SWEBOK® V3.0

Guide to the Software Engineering Body of Knowledge

Editors

Pierre Bourque
Richard E. (Dick) Fairley

IEEE

IEEE computer society

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

IEEE computer society

Copyright and Reprint Permissions. Educational or personal use of this material is permitted without fee provided such copies 1) are not made for profit or in lieu of purchasing copies for classes, and that this notice and a full citation to the original work appear on the first page of the copy and 2) do not imply IEEE endorsement of any third-party products or services. Permission to reprint/republish this material for commercial, advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from IEEE by writing to the IEEE Intellectual Property Rights Office, 445 Hoes Lane, Piscataway, NJ 08854-4141 or pubs-permissions@ieee.org.

Reference to any specific commercial products, process, or service does not imply endorsement by IEEE. The views and opinions expressed in this work do not necessarily reflect those of IEEE.

IEEE makes this document available on an "as is" basis and makes no warranty, express or implied, as to the accuracy, capability, efficiency merchantability, or functioning of this document. In no event will IEEE be liable for any general, consequential, indirect, incidental, exemplary, or special damages, even if IEEE has been advised of the possibility of such damages.

Copyright © 2014 IEEE. All rights reserved.
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 xvii
Foreword to the 2004 Edition xix
Editors xxi
Coeditors xxi
Contributing Editors xxi
Change Control Board xxi
Knowledge Area Editors xxiii
Knowledge Area Editors of Previous SWEBOK Versions xxv
Review Team xxvii
Acknowledgements xxix
Professional Activities Board, 2013 Membership xxix
Motions Regarding the Approval of SWEBOK Guide V3.0 xxx
Motions Regarding the Approval of SWEBOK Guide 2004 Version xxx
Introduction to the Guide xxxi

Chapter 1: Software Requirements 1-1
1. Software Requirements Fundamentals 1-1
1.1. Definition of a Software Requirement 1-1
1.2. Product and Process Requirements 1-2
1.3. Functional and Nonfunctional Requirements 1-3
1.4. Emergent Properties 1-3
1.5. Quantifiable Requirements 1-3
1.6. System Requirements and Software Requirements 1-3
2. Requirements Process 1-3
2.1. Process Models 1-4
2.2. Process Actors 1-4
2.3. Process Support and Management 1-4
2.4. Process Quality and Improvement 1-4
3. Requirements Elicitation 1-5
3.1. Requirements Sources 1-5
3.2. Elicitation Techniques 1-6
4. Requirements Analysis 1-7
4.1. Requirements Classification 1-7
4.2. Conceptual Modeling 1-8
4.3. Architectural Design and Requirements Allocation 1-9
4.4. Requirements Negotiation 1-9
4.5. Formal Analysis 1-10
5. Requirements Specification 1-10
5.1. System Definition Document 1-10
5.2. System Requirements Specification 1-10
5.3. Software Requirements Specification 1-11
6. Requirements Validation 1-11
6.1. Requirements Reviews 1-11
6.2. Prototyping 1-12

6.3. Model Validation *1-12*
6.4. Acceptance Tests *1-12*
7. Practical Considerations 1-12
7.1. Iterative Nature of the Requirements Process *1-13*
7.2. Change Management *1-13*
7.3. Requirements Attributes *1-13*
7.4. Requirements Tracing *1-14*
7.5. Measuring Requirements *1-14*
8. Software Requirements Tools 1-14
Matrix of Topics vs. Reference Material 1-15

Chapter 2: Software Design **2-1**
1. Software Design Fundamentals 2-2
1.1. General Design Concepts *2-2*
1.2. Context of Software Design *2-2*
1.3. Software Design Process *2-2*
1.4. Software Design Principles *2-3*
2. Key Issues in Software Design 2-3
2.1. Concurrency *2-4*
2.2. Control and Handling of Events *2-4*
2.3. Data Persistence *2-4*
2.4. Distribution of Components *2-4*
2.5. Error and Exception Handling and Fault Tolerance *2-4*
2.6. Interaction and Presentation *2-4*
2.7. Security *2-4*
3. Software Structure and Architecture 2-4
3.1. Architectural Structures and Viewpoints *2-5*
3.2. Architectural Styles *2-5*
3.3. Design Patterns *2-5*
3.4. Architecture Design Decisions *2-5*
3.5. Families of Programs and Frameworks *2-5*
4. User Interface Design 2-5
4.1. General User Interface Design Principles *2-6*
4.2. User Interface Design Issues *2-6*
4.3. The Design of User Interaction Modalities *2-6*
4.4. The Design of Information Presentation *2-6*
4.5. User Interface Design Process *2-7*
4.6. Localization and Internationalization *2-7*
4.7. Metaphors and Conceptual Models *2-7*
5. Software Design Quality Analysis and Evaluation 2-7
5.1. Quality Attributes *2-7*
5.2. Quality Analysis and Evaluation Techniques *2-8*
5.3. Measures *2-8*
6. Software Design Notations 2-8
6.1. Structural Descriptions (Static View) *2-8*
6.2. Behavioral Descriptions (Dynamic View) *2-9*
7. Software Design Strategies and Methods 2-10
7.1. General Strategies *2-10*
7.2. Function-Oriented (Structured) Design *2-10*
7.3. Object-Oriented Design *2-10*

7.4. Data Structure-Centered Design *2-10*
7.5. Component-Based Design (CBD) *2-10*
7.6. Other Methods *2-10*
8. Software Design Tools 2-11
Matrix of Topics vs. Reference Material 2-12

Chapter 3: Software Construction **3-1**
1. Software Construction Fundamentals 3-1
1.1. Minimizing Complexity *3-3*
1.2. Anticipating Change *3-3*
1.3. Constructing for Verification *3-3*
1.4. Reuse *3-3*
1.5. Standards in Construction *3-3*
2. Managing Construction 3-4
2.1. Construction in Life Cycle Models *3-4*
2.2. Construction Planning *3-4*
2.3. Construction Measurement *3-4*
3. Practical Considerations 3-5
3.1. Construction Design *3-5*
3.2. Construction Languages *3-5*
3.3. Coding *3-6*
3.4. Construction Testing *3-6*
3.5. Construction for Reuse *3-6*
3.6. Construction with Reuse *3-7*
3.7. Construction Quality *3-7*
3.8. Integration *3-7*
4. Construction Technologies 3-8
4.1. API Design and Use *3-8*
4.2. Object-Oriented Runtime Issues *3-8*
4.3. Parameterization and Generics *3-8*
4.4. Assertions, Design by Contract, and Defensive Programming *3-8*
4.5. Error Handling, Exception Handling, and Fault Tolerance *3-9*
4.6. Executable Models *3-9*
4.7. State-Based and Table-Driven Construction Techniques *3-9*
4.8. Runtime Configuration and Internationalization *3-10*
4.9. Grammar-Based Input Processing *3-10*
4.10. Concurrency Primitives *3-10*
4.11. Middleware *3-10*
4.12. Construction Methods for Distributed Software *3-11*
4.13. Constructing Heterogeneous Systems *3-11*
4.14. Performance Analysis and Tuning *3-11*
4.15. Platform Standards *3-11*
4.16. Test-First Programming *3-11*
5. Software Construction Tools 3-12
5.1. Development Environments *3-12*
5.2. GUI Builders *3-12*
5.3. Unit Testing Tools *3-12*
5.4. Profiling, Performance Analysis, and Slicing Tools *3-12*
Matrix of Topics vs. Reference Material 3-13

Chapter 4: Software Testing **4-1**
1. Software Testing Fundamentals 4-3
1.1. Testing-Related Terminology *4-3*
1.2. Key Issues *4-3*
1.3. Relationship of Testing to Other Activities *4-4*
2. Test Levels 4-5
2.1. The Target of the Test *4-5*
2.2. Objectives of Testing *4-5*
3. Test Techniques 4-7
3.1. Based on the Software Engineer's Intuition and Experience *4-8*
3.2. Input Domain-Based Techniques *4-8*
3.3. Code-Based Techniques *4-8*
3.4. Fault-Based Techniques *4-9*
3.5. Usage-Based Techniques *4-9*
3.6. Model-Based Testing Techniques *4-10*
3.7. Techniques Based on the Nature of the Application *4-10*
3.8. Selecting and Combining Techniques *4-11*
4. Test-Related Measures 4-11
4.1. Evaluation of the Program Under Test *4-11*
4.2. Evaluation of the Tests Performed *4-12*
5. Test Process 4-12
5.1. Practical Considerations *4-13*
5.2. Test Activities *4-14*
6. Software Testing Tools 4-15
6.1. Testing Tool Support *4-15*
6.2. Categories of Tools *4-15*
Matrix of Topics vs. Reference Material 4-17

Chapter 5: Software Maintenance **5-1**
1. Software Maintenance Fundamentals 5-1
1.1. Definitions and Terminology *5-1*
1.2. Nature of Maintenance *5-2*
1.3. Need for Maintenance *5-3*
1.4. Majority of Maintenance Costs *5-3*
1.5. Evolution of Software *5-3*
1.6. Categories of Maintenance *5-3*
2. Key Issues in Software Maintenance 5-4
2.1. Technical Issues *5-4*
2.2. Management Issues *5-5*
2.3. Maintenance Cost Estimation *5-6*
2.4. Software Maintenance Measurement *5-7*
3. Maintenance Process 5-7
3.1. Maintenance Processes *5-7*
3.2. Maintenance Activities *5-8*
4. Techniques for Maintenance 5-10
4.1. Program Comprehension *5-10*
4.2. Reengineering *5-10*
4.3. Reverse Engineering *5-10*
4.4. Migration *5-10*
4.5. Retirement *5-11*

5. Software Maintenance Tools 5-11
Matrix of Topics vs. Reference Material 5-12

Chapter 6: Software Configuration Management 6-1
1. Management of the SCM Process 6-2
1.1. Organizational Context for SCM 6-2
1.2. Constraints and Guidance for the SCM Process 6-3
1.3. Planning for SCM 6-3
1.4. SCM Plan 6-5
1.5. Surveillance of Software Configuration Management 6-5
2. Software Configuration Identification 6-6
2.1. Identifying Items to Be Controlled 6-6
2.2. Software Library 6-8
3. Software Configuration Control 6-8
3.1. Requesting, Evaluating, and Approving Software Changes 6-8
3.2. Implementing Software Changes 6-9
3.3. Deviations and Waivers 6-10
4. Software Configuration Status Accounting 6-10
4.1. Software Configuration Status Information 6-10
4.2. Software Configuration Status Reporting 6-10
5. Software Configuration Auditing 6-10
5.1. Software Functional Configuration Audit 6-11
5.2. Software Physical Configuration Audit 6-11
5.3. In-Process Audits of a Software Baseline 6-11
6. Software Release Management and Delivery 6-11
6.1. Software Building 6-11
6.2. Software Release Management 6-12
7. Software Configuration Management Tools 6-12
Matrix of Topics vs. Reference Material 6-13

Chapter 7: Software Engineering Management 7-1
1. Initiation and Scope Definition 7-4
1.1. Determination and Negotiation of Requirements 7-4
1.2. Feasibility Analysis 7-4
1.3. Process for the Review and Revision of Requirements 7-5
2. Software Project Planning 7-5
2.1. Process Planning 7-5
2.2. Determine Deliverables 7-5
2.3. Effort, Schedule, and Cost Estimation 7-6
2.4. Resource Allocation 7-6
2.5. Risk Management 7-6
2.6. Quality Management 7-6
2.7. Plan Management 7-7
3. Software Project Enactment 7-7
3.1. Implementation of Plans 7-7
3.2. Software Acquisition and Supplier Contract Management 7-7
3.3. Implementation of Measurement Process 7-7
3.4. Monitor Process 7-7
3.5. Control Process 7-8
3.6. Reporting 7-8

4. Review and Evaluation 7-8
4.1. Determining Satisfaction of Requirements 7-8
4.2. Reviewing and Evaluating Performance 7-9
5. Closure 7-9
5.1. Determining Closure 7-9
5.2. Closure Activities 7-9
6. Software Engineering Measurement 7-9
6.1. Establish and Sustain Measurement Commitment 7-9
6.2. Plan the Measurement Process 7-10
6.3. Perform the Measurement Process 7-11
6.4. Evaluate Measurement 7-11
7. Software Engineering Management Tools 7-11
Matrix of Topics vs. Reference Material 7-13

Chapter 8: Software Engineering Process 8-1
1. Software Process Definition 8-2
1.1. Software Process Management 8-3
1.2. Software Process Infrastructure 8-4
2. Software Life Cycles 8-4
2.1. Categories of Software Processes 8-5
2.2. Software Life Cycle Models 8-5
2.3. Software Process Adaptation 8-6
2.4. Practical Considerations 8-6
3. Software Process Assessment and Improvement 8-6
3.1. Software Process Assessment Models 8-7
3.2. Software Process Assessment Methods 8-7
3.3. Software Process Improvement Models 8-7
3.4. Continuous and Staged Software Process Ratings 8-8
4. Software Measurement 8-8
4.1. Software Process and Product Measurement 8-9
4.2. Quality of Measurement Results 8-10
4.3. Software Information Models 8-10
4.4. Software Process Measurement Techniques 8-11
5. Software Engineering Process Tools 8-12
Matrix of Topics vs. Reference Material 8-13

Chapter 9: Software Engineering Models and Methods 9-1
1. Modeling 9-1
1.1. Modeling Principles 9-2
1.2. Properties and Expression of Models 9-3
1.3. Syntax, Semantics, and Pragmatics 9-3
1.4. Preconditions, Postconditions, and Invariants 9-4
2. Types of Models 9-4
2.1. Information Modeling 9-5
2.2. Behavioral Modeling 9-5
2.3. Structure Modeling 9-5
3. Analysis of Models 9-5
3.1. Analyzing for Completeness 9-5
3.2. Analyzing for Consistency 9-6

3.3. Analyzing for Correctness *9-6*
3.4. Traceability *9-6*
3.5. Interaction Analysis *9-6*
4. Software Engineering Methods 9-7
4.1. Heuristic Methods *9-7*
4.2. Formal Methods *9-7*
4.3. Prototyping Methods *9-8*
4.4. Agile Methods *9-9*
Matrix of Topics vs. Reference Material 9-10

Chapter 10: Software Quality **10-1**
1. Software Quality Fundamentals 10-2
1.1. Software Engineering Culture and Ethics *10-2*
1.2. Value and Costs of Quality *10-3*
1.3. Models and Quality Characteristics *10-3*
1.4. Software Quality Improvement *10-4*
1.5. Software Safety *10-4*
2. Software Quality Management Processes 10-5
2.1. Software Quality Assurance *10-5*
2.2. Verification & Validation *10-6*
2.3. Reviews and Audits *10-6*
3. Practical Considerations 10-9
3.1. Software Quality Requirements *10-9*
3.2. Defect Characterization *10-10*
3.3. Software Quality Management Techniques *10-11*
3.4. Software Quality Measurement *10-12*
4. Software Quality Tools 10-12
Matrix of Topics vs. Reference Material 10-14

Chapter 11: Software Engineering Professional Practice **11-1**
1. Professionalism 11-2
1.1. Accreditation, Certification, and Licensing *11-3*
1.2. Codes of Ethics and Professional Conduct *11-4*
1.3. Nature and Role of Professional Societies *11-4*
1.4. Nature and Role of Software Engineering Standards *11-4*
1.5. Economic Impact of Software *11-5*
1.6. Employment Contracts *11-5*
1.7. Legal Issues *11-5*
1.8. Documentation *11-7*
1.9. Tradeoff Analysis *11-8*
2. Group Dynamics and Psychology 11-9
2.1. Dynamics of Working in Teams/Groups *11-9*
2.2. Individual Cognition *11-9*
2.3. Dealing with Problem Complexity *11-10*
2.4. Interacting with Stakeholders *11-10*
2.5. Dealing with Uncertainty and Ambiguity *11-10*
2.6. Dealing with Multicultural Environments *11-10*
3. Communication Skills 11-11
3.1. Reading, Understanding, and Summarizing *11-11*

3.2. Writing *11-11*
3.3. Team and Group Communication *11-11*
3.4. Presentation Skills *11-12*
Matrix of Topics vs. Reference Material 11-13

Chapter 12: Software Engineering Economics **12-1**
1. Software Engineering Economics Fundamentals 12-3
1.1. Finance *12-3*
1.2. Accounting *12-3*
1.3. Controlling *12-3*
1.4. Cash Flow *12-3*
1.5. Decision-Making Process *12-4*
1.6. Valuation *12-5*
1.7. Inflation *12-6*
1.8. Depreciation *12-6*
1.9. Taxation *12-6*
1.10. Time-Value of Money *12-6*
1.11. Efficiency *12-6*
1.12. Effectiveness *12-6*
1.13. Productivity *12-6*
2. Life Cycle Economics 12-7
2.1. Product *12-7*
2.2. Project *12-7*
2.3. Program *12-7*
2.4. Portfolio *12-7*
2.5. Product Life Cycle *12-7*
2.6. Project Life Cycle *12-7*
2.7. Proposals *12-8*
2.8. Investment Decisions *12-8*
2.9. Planning Horizon *12-8*
2.10. Price and Pricing *12-8*
2.11. Cost and Costing *12-9*
2.12. Performance Measurement *12-9*
2.13. Earned Value Management *12-9*
2.14. Termination Decisions *12-9*
2.15. Replacement and Retirement Decisions *12-10*
3. Risk and Uncertainty 12-10
3.1. Goals, Estimates, and Plans *12-10*
3.2. Estimation Techniques *12-11*
3.3. Addressing Uncertainty *12-11*
3.4. Prioritization *12-11*
3.5. Decisions under Risk *12-11*
3.6. Decisions under Uncertainty *12-12*
4. Economic Analysis Methods 12-12
4.1. For-Profit Decision Analysis *12-12*
4.2. Minimum Acceptable Rate of Return *12-13*
4.3. Return on Investment *12-13*
4.4. Return on Capital Employed *12-13*
4.5. Cost-Benefit Analysis *12-13*

4.6. Cost-Effectiveness Analysis *12-13*
4.7. Break-Even Analysis *12-13*
4.8. Business Case *12-13*
4.9. Multiple Attribute Evaluation *12-14*
4.10. Optimization Analysis *12-14*
5. Practical Considerations 12-14
5.1. The "Good Enough" Principle *12-14*
5.2. Friction-Free Economy *12-15*
5.3. Ecosystems *12-15*
5.4. Offshoring and Outsourcing *12-15*
Matrix of Topics vs. Reference Material 12-16

Chapter 13: Computing Foundations **13-1**
1. Problem Solving Techniques 13-3
1.1. Definition of Problem Solving *13-3*
1.2. Formulating the Real Problem *13-3*
1.3. Analyze the Problem *13-3*
1.4. Design a Solution Search Strategy *13-3*
1.5. Problem Solving Using Programs *13-3*
2. Abstraction 13-4
2.1. Levels of Abstraction *13-4*
2.2. Encapsulation *13-4*
2.3. Hierarchy *13-4*
2.4. Alternate Abstractions *13-5*
3. Programming Fundamentals 13-5
3.1. The Programming Process *13-5*
3.2. Programming Paradigms *13-5*
4. Programming Language Basics 13-6
4.1. Programming Language Overview *13-6*
4.2. Syntax and Semantics of Programming Languages *13-6*
4.3. Low-Level Programming Languages *13-7*
4.4. High-Level Programming Languages *13-7*
4.5. Declarative vs. Imperative Programming Languages *13-7*
5. Debugging Tools and Techniques 13-8
5.1. Types of Errors *13-8*
5.2. Debugging Techniques *13-8*
5.3. Debugging Tools *13-8*
6. Data Structure and Representation 13-9
6.1. Data Structure Overview *13-9*
6.2. Types of Data Structure *13-9*
6.3. Operations on Data Structures *13-9*
7. Algorithms and Complexity 13-10
7.1. Overview of Algorithms *13-10*
7.2. Attributes of Algorithms *13-10*
7.3. Algorithmic Analysis *13-10*
7.4. Algorithmic Design Strategies *13-11*
7.5. Algorithmic Analysis Strategies *13-11*
8. Basic Concept of a System 13-11
8.1. Emergent System Properties *13-11*

8.2. Systems Engineering *13-12*
8.3. Overview of a Computer System *13-12*
9. Computer Organization 13-13
9.1. Computer Organization Overview *13-13*
9.2. Digital Systems *13-13*
9.3. Digital Logic *13-13*
9.4. Computer Expression of Data *13-13*
9.5. The Central Processing Unit (CPU) *13-14*
9.6. Memory System Organization *13-14*
9.7. Input and Output (I/O) *13-14*
10. Compiler Basics 13-15
10.1. Compiler/Interpreter Overview *13-15*
10.2. Interpretation and Compilation *13-15*
10.3. The Compilation Process *13-15*
11. Operating Systems Basics 13-16
11.1. Operating Systems Overview *13-16*
11.2. Tasks of an Operating System *13-16*
11.3. Operating System Abstractions *13-17*
11.4. Operating Systems Classification *13-17*
12. Database Basics and Data Management 13-17
12.1. Entity and Schema *13-18*
12.2. Database Management Systems (DBMS) *13-18*
12.3. Database Query Language *13-18*
12.4. Tasks of DBMS Packages *13-18*
12.5. Data Management *13-19*
12.6. Data Mining *13-19*
13. Network Communication Basics 13-19
13.1. Types of Network *13-19*
13.2. Basic Network Components *13-19*
13.3. Networking Protocols and Standards *13-20*
13.4. The Internet *13-20*
13.5. Internet of Things *13-20*
13.6. Virtual Private Network (VPN) *13-21*
14. Parallel and Distributed Computing 13-21
14.1. Parallel and Distributed Computing Overview *13-21*
14.2. Difference between Parallel and Distributed Computing *13-21*
14.3. Parallel and Distributed Computing Models *13-21*
14.4. Main Issues in Distributed Computing *13-22*
15. Basic User Human Factors 13-22
15.1. Input and Output *13-22*
15.2. Error Messages *13-23*
15.3. Software Robustness *13-23*
16. Basic Developer Human Factors 13-23
16.1. Structure *13-24*
16.2. Comments *13-24*
17. Secure Software Development and Maintenance 13-24
17.1. Software Requirements Security *13-24*
17.2. Software Design Security *13-25*
17.3. Software Construction Security *13-25*
17.4. Software Testing Security *13-25*

17.5. Build Security into Software Engineering Process *13-25*
17.6. Software Security Guidelines *13-25*
Matrix of Topics vs. Reference Material 13-27

Chapter 14: Mathematical Foundations **14-1**
1. Set, Relations, Functions 14-1
1.1. Set Operations *14-2*
1.2. Properties of Set *14-3*
1.3. Relation and Function *14-4*
2. Basic Logic 14-5
2.1. Propositional Logic *14-5*
2.2. Predicate Logic *14-5*
3. Proof Techniques 14-6
3.1. Methods of Proving Theorems *14-6*
4. Basics of Counting 14-7
5. Graphs and Trees 14-8
5.1. Graphs *14-8*
5.2. Trees *14-10*
6. Discrete Probability 14-13
7. Finite State Machines 14-14
8. Grammars 14-15
8.1. Language Recognition *14-16*
9. Numerical Precision, Accuracy, and Errors 14-17
10. Number Theory 14-18
10.1. Divisibility *14-18*
10.2. Prime Number, GCD *14-19*
11. Algebraic Structures 14-19
11.1. Group *14-19*
11.2. Rings *14-20*
Matrix of Topics vs. Reference Material 14-21

Chapter 15: Engineering Foundations **15-1**
1. Empirical Methods and Experimental Techniques 15-1
1.1. Designed Experiment *15-1*
1.2. Observational Study *15-2*
1.3. Retrospective Study *15-2*
2. Statistical Analysis 15-2
2.1. Unit of Analysis (Sampling Units), Population, and Sample *15-2*
2.2. Concepts of Correlation and Regression *15-5*
3. Measurement 15-5
3.1. Levels (Scales) of Measurement *15-6*
3.2. Direct and Derived Measures *15-7*
3.3. Reliability and Validity *15-8*
3.4. Assessing Reliability *15-8*
4. Engineering Design 15-8
4.1. Engineering Design in Engineering Education *15-8*
4.2. Design as a Problem Solving Activity *15-9*
4.3. Steps Involved in Engineering Design *15-9*
5. Modeling, Simulation, and Prototyping 15-10
5.1. Modeling *15-10*

5.2. Simulation *15-11*
5.3. Prototyping *15-11*
6. Standards 15-12
7. Root Cause Analysis 15-12
7.1. Techniques for Conducting Root Cause Analysis *15-13*
Matrix of Topics vs. Reference Material 15-14

Appendix A: Knowledge Area Description Specifications **A-1**

Appendix B: IEEE and ISO/IEC Standards Supporting the Software Engineering Body of Knowledge (SWEBOK) **B-1**

Appendix C: Consolidated Reference List **C-1**

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

ATDD	Acceptance Test Driven Development
BDD	Behavior Driven Development
CIA	Confidentiality, Integrity, and Availability
FSM	Functional Size Measurement
INCOSE	International Council on Systems Engineering
JAD	Joint Application Development
JRP	Joint Requirements Planning
RUP	Rational Unified Process
SME	Subject Matter Expert
SysML	Systems Modeling Language
TDD	Test Driven Development
UML	Unified Modeling Language

INTRODUCTION

Software requirements should be viewed from two perspectives. The first is as an expression of the needs and constraints on a software product or project that contribute to the solution of a real-world problem. The second is that of the activities necessary to develop and maintain the requirements for a software product and for the project that constructs it. Both perspectives are presented in this knowledge area (KA).

Industry experts and researchers generally agree on the importance of software requirements for success. If a team does a poor job of determining the requirements, the project, the product or both are likely to suffer from added costs, delays, cancellations and defects. One reason is that each software product requirement generally leads to many design decisions. Each design decision generally leads to many code-level decisions. Each decision can involve several test decisions, as well. In other words, determining the requirements correctly is high-stakes work. If not detected and repaired early, missing, misinterpreted and incorrect requirements can induce exponentially cascading rework to correct them.

Real-world software projects tend to suffer from two primary requirements-related problems:

1. incompleteness: stakeholder requirements exist that are not revealed and communicated to the software engineers;
2. ambiguity: requirements are communicated in a way that is open to multiple interpretations, with only one possible interpretation being correct.

Beyond the obvious short-term role requirements play in initial software construction, they also play a less recognized but still important role in long-term maintenance. Upon receiving software without any supporting documentation, a software engineer has several means to determine what that code does, such as execute it, step through it with a debugger, hand-execute it, statically analyze it, and so on.

The challenge is determining what that code is *intended to do*. What is generally referred to as a *bug* — but is better called a *defect* — is simply an observable difference between what the software is intended to do and what it does. The role of requirements documentation throughout the service life of the software is to capture and communicate intent for software engineers who maintain the code but might not have been its original authors.

The Software Requirements KA concerns developing software requirements and managing those requirements over the software's service life. This KA provides an understanding that software requirements:

- are not necessarily a discrete front-end activity of the software development life cycle but rather a process initiated at a project's beginning that often continues to be refined throughout the software's entire service life;
- need to be tailored to the organization and project context.

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

The Software Requirements KA is most closely related to the Software Architecture, Software Design, Software Construction, Software Testing, and Software Maintenance KAs, as well as to the models topic in the Software Engineering Models and Methods KA, in that there can be high value in specifying requirements in model form.

This KA is also related to the Software Life Cycles topic in the Software Engineering Process KA, in that this KA's focus is on *what* and *how* requirements work can and should be done, whereas the project's life cycle determines *when* that work is done. For example, in a waterfall life cycle, all requirements work is essentially done in a discrete *Requirements phase* and is expected to be substantially complete before any architecture, design and construction work occurs in subsequent phases. Under the Rational Unified Process (RUP), initial, high-level requirements work is done during an *Inception phase*, and further detailing is done during one or more *Elaboration phases*. In an Agile life cycle, requirements work is done incrementally, just in time, as each additional element of functionality is constructed.

The *whats* and *hows* of software requirements work on a project should be determined by the nature of the software constructed, not by the life cycle under which it is constructed. Insofar as requirements documentation captures and communicates the software's intent, downstream maintainers should not be able to discern the life cycle used in earlier development from the form of those requirements alone.

This KA is also related, but somewhat less so, to the Software Configuration Management, Software Engineering Management and Software Quality KAs.

BREAKDOWN OF TOPICS FOR SOFTWARE REQUIREMENTS

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

1. Software Requirements Fundamentals

1.1. Definition of a Software Requirement [1*, c1pp5-6] [2*, c4p102]

Formally, a *software requirement* has been defined as [28]:

- a condition or capability needed by a user to solve a problem or achieve an objective;

Figure 1.1. Breakdown of Topics for the Software Requirements KA

- a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed document;
- a documented representation or capability as in (1) or (2) above.

This formal definition is extended in this KA to include expressions of a software project's needs and constraints.

At its most basic, a software requirement is a property that must be exhibited to solve a real-world problem. It might aim to automate all or part of a task supporting an organization's business policies and processes, correct existing software's shortcomings, or control a device — just a few of the many problems for which software solutions are possible.

Business policies and processes, as well as device functions, are often very complex. By extension, software requirements are often a complex combination of requirements from various stakeholders at different organizational levels who are involved or connected with some aspect of the environment in which the software will operate.

Clients, customers and users usually impose requirements. However, other third parties, like regulatory authorities and, in some cases, the software organization or the project itself, might also impose requirements. (See also [5, c1] [6, c1] [9, c4].)

1.2. Categories of Software Requirements [1*, c1pp7-12] [2*, s4.1]

Figure 1.2 shows the categories of software requirements defined in this KA and the relationships among those categories. (See also [5, c1] [6, c1] [9, c4].) Each category is further described below.

1.3. Software Product Requirements and Software Project Requirements [1*, c1pp14-15]

Software product requirements specify the software's expected form, fit or function. *Software project requirements* — also called *process requirements* — constrain the project that constructs the software. Project requirements often constrain cost, schedule and/or staffing but can also

constrain other aspects of a software project, such as maintenance. Software project requirements can be captured in a project charter or other high-level project initiation document. They are most relevant to how the project is managed (see the Software Engineering Management KA) or what life cycle process should be used (see the Software Engineering Process KA). This KA does not discuss software project requirements further.

Figure 1.2. Categories of Software Requirements

1.4. Functional Requirements [1*, c1p9] [2*, s4.1.1]

Functional requirements specify observable behaviors that the software is to provide — policies to be enforced and processes to be carried out. Example policies in banking software might be "an account shall always have at least one customer as its owner," and "the balance of an account shall never be negative." Example processes could specify the meanings of depositing money into an account, withdrawing money from an account and transferring money from one account to another.

Even highly technical (nonbusiness-oriented) software, such as software that implements the Transmission Control Protocol/Internet Protocol (TCP/IP) network communications protocol, has policies and processes: "a Port shall be able to exist with zero, one, or many associated Connections, but a Connection shall exist on exactly one associated Port," "acceptable states of a Connection shall be 'listen,' 'syn sent,' 'established,' 'closing,' . . . ," and "if the time-to-live of a Segment reaches zero, that Segment shall be deleted." (See [5, c1] [6, c10] [9, c4].)

1.5. Nonfunctional Requirements [1*, c1pp10-11] [2*, s4.1.2]

Nonfunctional requirements in some way constrain the technologies to be used in the implementation: What computing platform(s)? What database engine(s)? How accurate do results need to be? How quickly must results be presented? How many records of a certain type need to be stored? Some nonfunctional requirements might relate to the operation of the software. (See the Operation and Maintenance KA.) (See also [5, c1] [6, c11] [9, c4].)

The nonfunctional requirements can be further divided into technology constraints and quality of service constraints.

1.6. Technology Constraints

These requirements mandate — or prohibit — use of specific, named automation technologies or defined infrastructures. Examples are requirements to use specific computing platforms (e.g., Windows™, MacOS™, Android OS™, iOS™), programming languages (e.g., Java, C++, C#, Python), compatibility with specific web browsers (e.g., Chrome™, Safari™, Internet Explorer™, Edge™), given database engines (e.g., Oracle™, SQL Server™, MySQL™), and general technologies (e.g., Reduced Instruction Set Computer (RISC), Relational Database. A requirement prohibiting use of pointers would be another example. (See also [9, c4].)

1.7. Quality of Service Constraints

These requirements do not constrain the use of specific, named technologies. Instead, these specify acceptable performance levels an automated solution must exhibit. Examples are response time, throughput, accuracy, reliability and scalability.

International Standards Organization/International Electrotechnical Commission (ISO/IEC) 25010: "System and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models" [27] contains a large list of the kinds of service qualities that can be relevant for software. (See also [9, c4].) Security is also a particularly important topic where requirements tend to be overlooked. (See the Security KA for details on the kinds of specific security requirements that should be considered.) (See also [2*, c13].)

1.8. Why Categorize Requirements This Way?

Categorizing requirements this way is useful for the following reasons:

- requirements in one category tend to come from different sources than other categories;
- elicitation techniques often vary by source;
- analysis techniques vary by category;
- specification techniques vary by category;
- validation authorities vary by category;
- the different categories affect the resulting software in different ways.

In addition, organizing the requirements in these categories is beneficial in the following ways:

- complexity can be better managed because different areas can be addressed separately; software engineers can deal with policy and process complexities without worrying about automation technology issues at the same time (and vice versa). One large problem becomes two smaller ones. This is classic *divide and conquer* complexity management;
- distinct areas of expertise can be isolated; stakeholders, not software engineers, are the experts in the policies and processes to be automated. Software engineers, not stakeholders, are the technology experts. When a business expert is given interspersed functional and nonfunctional requirements for review or validation, they might give up because they don't understand — or even care about — the technology issues. The relevant requirements reviewer can focus on just the subset of requirements relevant to them.

The *Perfect Technology Filter* originally described in [18, c1-4] but also explained in [8] and [9, c4] helps separate functional from nonfunctional requirements. Simply put, functional requirements are those that would still need to be stated even if a computer with infinite speed, unlimited memory, zero cost, no failures, etc., existed on which to construct the software. All other software product requirements are constraints on automation technologies and are therefore nonfunctional.

Large systems often span more than one subject matter area, or domain. As explained in [9, c6], recursive design shows how nonfunctional requirements in a parent domain can become, or can induce, functional requirements in a child domain. For example, a nonfunctional requirement about user security in a parent banking domain can become or can induce functional requirements in a child security domain. Similarly, cross-cutting nonfunctional requirements about auditing and transaction management in a parent banking domain can become or induce functional requirements in a child auditing domain and a child transaction domain. Decomposing large systems into a set of related domains significantly reduces complexity.

1.9. System Requirements and Software Requirements

The International Council on Systems Engineering (INCOSE) defines a *system* as "an interacting combination of elements to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements" [24].

In some cases, it is either useful or mandatory to distinguish system requirements from software requirements. System requirements apply to larger systems — for example, an autonomous vehicle. Software requirements apply only to an element of software in that larger system. Some software requirements may be derived from system requirements. (See also [5, c1].)

1.10. Derived Requirements

In practice, *requirements* can be context-sensitive and can depend on perspective. An external stakeholder can impose a scope requirement, and this would be a requirement for the entire project — even if that project involves hundreds of software engineers. An architect's decision to use a pipes-and-filters architecture style would not be a requirement from the perspective of the overall project stakeholders, but a design decision. But that same decision, when seen from the perspective of a sub-team responsible for constructing a particular filter, would be considered a requirement.

The aerospace industry has long used the term *derived requirement* to mean a requirement that was not made by a stakeholder external to the overall project but that was imposed inside the larger development team. The architect's pipes-and-filters decision fits this definition. That choice would be seen as a design decision from the point of view of external stakeholders, but as a requirement for the sub-teams responsible for developing each filter. (See also [9, c4].)

1.11. Software Requirements Activities [1*, c1pp15-18] [2*, s4.2]
Figure 1.3 shows the requirements development and management activities.

Requirements development, as a whole, can be thought of as "reaching an agreement on what software is to be constructed." In contrast, requirements management can be considered "maintaining that agreement over time." Each activity is presented in this KA. Requirements development activities are presented as separate topics, with requirements management presented as a single topic. (See also [5, c1] [6, 2].)

Figure 1.3. Software Requirements Activities

2. Requirements Elicitation [1*, c6-7] [2*, s4.3]

The goal of requirements elicitation is to surface candidate requirements. It is also called *requirements capture*, *requirements discovery* or *requirements acquisition*. As stated earlier, one problem in requirements work on real-world software projects is incompleteness. This can be the result of inadequate elicitation. Although there is no guarantee that a set of requirements is complete, well-executed elicitation helps minimize incompleteness. (See also [5, c2-3] [6, c3-7].)

2.1. Requirements Sources [1*, c6] [2*, s4.3]
Requirements come — can be elicited — from many different sources. All potential requirements sources should be identified and evaluated. A *stakeholder* can be defined as any person, group or organization that:

- is actively involved in the project;
- is affected by the project's outcome;
- can influence the project's outcome.

Typical stakeholders for software projects include but are not limited to the following:

- clients — those who pay for the software to be constructed (e.g., organizational management);
- customers — those who decide whether a software product will be put into service;
- users — those who interact directly or indirectly with the software; users can often be further broken down into distinct user classes that vary in frequency of use, tasks performed, skill and knowledge level, privilege level, and so on;
- subject matter experts (SMEs);
- operations staff;
- first-level product support staff;
- relevant professional bodies;
- regulatory agencies;
- special interest groups;
- people who can be negatively affected if the project is successful;
- developers.

Stakeholder classes are groups of stakeholders that have similar perspectives and needs. Working on a software project in terms of stakeholder classes rather than with individual stakeholders can produce important, additional insight.

Many projects benefit from performing a stakeholder analysis to identify as many important stakeholder classes as possible. This reduces the possibility that the requirements are biased toward better-represented stakeholders and away from less well-represented stakeholders. The stakeholder analysis can also inform negotiation and conflict resolution when requirements from one stakeholder class conflict with requirements from another. (See also [5, c3] [6, c3].)

2.2. Common Requirements Elicitation Techniques [1*, c7] [2*, s4.3]

A wide variety of techniques can be used to elicit requirements from stakeholders. Some techniques work better with certain stakeholder classes than others. Common stakeholder elicitation techniques include the following:

- interviews;
- meetings, possibly including brainstorming;
- Joint Application Development (JAD) [13], Joint Requirements Planning (JRP) [14] and other facilitated workshops;
- protocol analysis;
- focus groups
- questionnaires and market surveys
- exploratory prototyping, including low-fidelity and high-fidelity user interface prototyping [1*, c15];
- user story mapping.

Elicitation can be difficult, and the software engineer needs to know that (for example) users might have difficulty describing their tasks, leave important information unstated or be unwilling or unable to cooperate. Elicitation is not a passive activity. Even if cooperative and articulate stakeholders are available, the software engineer must work hard to elicit the right information. Many product requirements are tacit or can be found only in information that has yet to be collected.

Requirements can also be elicited from sources other than stakeholders. Such sources and techniques include the following:

- previous versions of the system;
- defect tracking database for previous versions of the system;
- systems that interface with the system under development;
- competitive benchmarking;
- literature search;
- Quality Function Deployment (QFD)'s House of Quality [15];
- observation, where the software engineer studies the work and the environment where the work is being done;
- apprenticing, where the software engineer learns by doing the work;
- usage scenario descriptions;
- decomposition (e.g., capabilities into epics into features into stories);

- task analysis [16];
- design thinking (empathize, define, ideate, prototype, test) [17];
- ISO/IEC 25010: "System and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models" [27];
  - security requirements, as discussed in the Security KA;
  - applicable standards and regulations.

(See also [5, c3] [6, c4-7].)

3. Requirements Analysis [1*, c8-9]

Requirements are unlikely to be elicited in their final form. Further investigation is usually needed to reveal the full, true requirements suggested by the originally elicited information. Requirements analysis helps software developers understand the meaning and implications of candidate requirements, both individually and in the context of the overall set of requirements.

3.1. Basic Requirements Analysis [1*, c8-9]

The following list of desirable properties of requirements can guide basic requirements analysis. The software engineer seeks to establish any of these properties that do not hold yet. Each requirement should:

- be unambiguous (interpretable in only one way);
- be testable (quantified), meaning that compliance or noncompliance can be clearly demonstrated;
- be binding, meaning that clients are willing to pay for it and unwilling not to have it;
- represent true, actual stakeholder needs;
- use stakeholder vocabulary;
- be acceptable to all stakeholders.

The overall collection of requirements should be:

- complete, — The requirements adequately address boundary conditions, exception conditions and security needs;
- internally consistent — No requirement conflicts with any other;
- externally consistent — No requirement conflicts with any source material;
- feasible — A viable, cost-effective solution can be created within cost, schedule, staffing, and other constraints.

In some cases, an elicited statement represents a solution to be implemented rather than the true problem to be solved. This risks implementing a suboptimal solution. The *5-whys* technique (e.g., [3*, c4]) involves repeatedly asking, "Why is this the requirement?" to converge on the true problem. Repetition stops when the answer is, "If that isn't done, then the stakeholder's problem has not been solved." Often, the true problem is reached in two or three cycles, but the technique is called *5-whys* to incentivize engineers to push it as far as possible.

3.2. Economics of Quality of Service Constraints [3*]

Quality of service constraints can be particularly challenging. This is generally because engineers do not consider them from an economic perspective [9, c4]. Figure 1.4 illustrates the economic perspective of a typical quality of service constraint, such as capacity, throughput and reliability, where value increases with performance level. This curve is mirrored vertically for quality of service constraints whose value decreases as performance level increases (response time and mean time to repair would be examples).

Figure 1.4. Value as a Function of Level of Performance

Over the relevant range of performance levels, the stakeholders have a corresponding value if the system performs at that level. The value curve has two important points:

1. Perfection point — This is the most favorable level of performance, beyond which there is no additional benefit. Even if the system can perform better than the perfection point, the customer cannot use that capacity. For example, a social media system that supports more members than the world population would have this excess capacity.
2. Fail point — This is the least favorable level of performance, beyond which there is no further reduction in benefit. For example, the social media system might need to support at least a minimum market share to be viable as a platform.

A quantified requirement point, even if stated explicitly, is usually arbitrary. It is often based on what a client feels justified requesting, given what they are paying for the software. Even if the software engineers cannot construct a system that fully achieves the stated requirement point, the software typically still has value; it just has less value than the client expected. Further, the ability to exceed the requirement point can significantly increase value in some cases.

The cost to achieve a given performance level is usually a step function. First, for a given investment level, there is some maximum achievable performance level. Then, additional investment is needed, and that further investment enables performance up to a new, more favorable maximum. Figure 1.5 illustrates the most cost-effective performance level — the performance level with the maximum positive difference between the value at that performance level and the cost to achieve it.

Figure 1.5. Most Cost-Effective Level of Performance

(See the Software Engineering Economics KA or [3*] for more information on performing economic analyses such as this.)

The software engineer should pay particular attention to positive and negative relationships between quality of service constraints (e.g., Figure 14-2 in [1*, c14]). Some quality of service constraints are mutually supporting; improving one's performance level will automatically improve the other's performance level. For example, the more modifiable code is, the more reliable it tends to be, as both modifiability and reliability are, to a degree, a consequence of how clean the code is. On the other hand, the higher the code's speed, the less modifiable it might be, because high speed is often

achieved through optimizations that make the code more complex.

3.3. Formal Analysis [2*, s12.3.2-12.3.3]

Formal analysis has shown benefits in some application domains, particularly high-integrity systems (e.g., [5, c6]). The formal expression of requirements depends on the use of a specification language with formally defined semantics. Formality has two benefits. First, formal requirements are precise and concise, which (in principle) will reduce the possibility for misinterpretation. Second, formal requirements can be reasoned over, permitting desired properties of the specified software to be proved. This permits static validation that the software specified by the requirements does have the properties (e.g., absence of deadlock) that the customer, users and software engineer expect it to have.

This topic is related to Formal Methods in the Software Engineering Models and Methods KA.

3.4. Addressing Conflict in Requirements

When a project has more — and more diverse — stakeholders, conflicts among the requirements are more likely. One particularly important aspect of requirements analysis is identifying and managing such conflicts (e.g., [6, c17]). Once conflicting requirements have been identified, the engineer may consider two different approaches to managing that conflict (and possibly other approaches as well) and determine the most appropriate course of action.

One approach is to negotiate a resolution among the conflicting stakeholders. In most cases, it is unwise for the software engineer to make a unilateral decision, so it becomes necessary to consult with the stakeholders to reach a consensus resolution. It is often also important, for contractual reasons, that such decisions be traceable back to the customer. A specific example is *project scope management* — namely, balancing what's desired in the stated software product requirements with what can be accomplished given the project requirements of cost, schedule, staffing and other project-level constraints. There are many useful sources for information on negotiation and conflict resolution [25].

Another approach is to apply *product family development* (e.g., [20]). This involves separating requirements into two categories. The first category contains the *invariant requirements*. These are requirements that all stakeholders agree on. The second category contains the *variant requirements*, where conflict exists. The software engineer can focus on understanding the range of variations needed to satisfy all stakeholders. The software can be designed using *design to invariants* to accommodate the invariant requirements and *design for change* to incorporate customization points to configure an instance of the system to best fit relevant stakeholders. In a simple example, some users of a weather application require temperatures displayed in degrees Celsius while others require degrees Fahrenheit.

4. Requirements Specification [1*, c10-14, c20-26] [2*, s4.4, c5]

Requirements specification concerns recording the requirements so they can be both remembered and communicated. Requirements specification might be the most contentious topic in this KA. Debate centers on questions such as:

- should requirements be written down at all?
- if requirements are written down, what form should they take?
- if requirements are written down, should they also be maintained over time?

There are no standard answers to these questions; the answer to each can depend on factors such as the following:

- the software engineer's familiarity with the business domain;
- precedent for this kind of software;

- degree of risk (e.g., probability, severity) of incorrect requirements;
- staff turnover anticipated during the software's service life of the software;
- geographic distribution of the development team members;
- stakeholder involvement over the course of the project;
- whether the use of a packaged solution is anticipated;
- whether any design or construction will be outsourced;
- the degree of requirements-based testing expected;
- effort needed to use a candidate specification technique;
- accuracy needed from the requirements-based estimates;
- extent of requirements tracing necessary, if any;
- Contractual impositions of requirements specification content and format.

As established in this KA's introduction, the *whats* and *hows* of software requirements work on a project should be determined by the nature of the software constructed, not by the life cycle under which it is constructed. Downstream maintainers should not be able to discern the life cycle used in earlier development from the form of those requirements alone. The chosen life cycle's effect should be limited to the completeness of the requirements at any point in the project. Under a waterfall life cycle, the requirements are expected to be completely specified at the end of the Requirements phase. Under an Agile life cycle, the requirements are expected to grow continuously and not be complete until the project's end.

Some organizations have a culture of documenting requirements; some do not. Dynamic startup projects are often driven by a strong product vision and limited resources; their teams might view requirements documentation as unnecessary overhead. But as these products evolve and mature, software engineers often recognize 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 become important to long-term success. A project's approach to requirements in general, and to requirements specification in particular, may evolve over the service life of that software.

The most basic recommendation for requirements documentation is to base decisions on an *audience analysis*. Who are the different consumers who will need information from a requirements specification? What information will they need? How can that information be packaged and presented so that each consumer can get the information they need with the least effort?

There is a degree of overlap and dependency between requirements analysis and specification. Use of certain requirements specification techniques — particularly model-based requirements specifications — permit and encourage requirements analysis that can go beyond what has already been presented.

Documented software requirements should be subject to the same configuration management practices as the other deliverables of the software life cycle processes. (See the Configuration Management KA for a detailed discussion.) In addition, when practical, the individual requirements are also subject to configuration management, which is generally supported by a requirements management tool. (See Topic 8, Software Requirements Tools.)

There are several general categories of requirements specification techniques, each of which is discussed below. The requirements specification for a given project may also use various techniques. ISO/IEC/IEEE 29148 [26], as well as [1*, c10-14], [5, c4], [6, c16], and many others offer templates for requirements documentation.

4.1. Unstructured Natural Language Requirements Specification [1*, c11] [2*, s4.4.1]

Natural language requirements specifications express requirements in common, ordinary language. Natural language requirements specifications can be unstructured or structured.

A typical unstructured natural language requirements specification is a collection of statements in natural language, such as, "The system shall . . . ." For example, business rules are statements that define or constrain some aspect of the structure or the behavior of the business to be automated. "A student cannot register in next semester's courses if there remain any unpaid tuition fees" is an example of a business rule that serves as a requirement for a university's course-registration software. Some projects can publish a user manual as a satisfactory requirements specification, although there are limits to how effective this can be. (See also [5, c4] [26].)

4.2. Structured Natural Language Requirements Specification [1*, c8] [2*, s4.4.2]

Structured natural language requirements specifications impose constraints on how the requirements are expressed; the goal is to increase precision and conciseness.

The simplest example might be the actor-action format. The actor is the entity responsible for carrying out the action, and action is what needs to happen. A triggering event might precede the actor, and the action might be followed by an optional condition or qualification. The statement "When an order is shipped, the system shall create an Invoice unless the Order Terms are 'Prepaid'" uses actor-action format. The triggering event is "When an order is shipped." The actor is "the system." The action is "create an Invoice." The condition/qualification is "except the Order Terms are 'Prepaid'."

Another example is a use case specification template, as shown in Figure 1.6. (See [11] for guidelines on writing good use case specifications.)

Use case #66	Use case name: Reserve flight(s)
Triggering event(s)	Customer requests reservation(s) on flight(s)
Parameters	Passenger, itinerary, fare class, payment method(s)
Requires	Legal itinerary, fare class restrictions met
Guarantees	Seat(s) reserved for passenger on itinerary flight(s)
Normal course	Non-FF passenger, all domestic itinerary, Economy fare class, credit/debit card
Alternative course(s)	Is FF passenger: [None, Silver, Gold, Platinum, Elite] Itinerary: [all international, mixed domestic + international] Fare class: [Basic economy, Premium Economy, Business, First] Payment method: [Voucher, FF miles]
Exceptions	C/D card declined, voucher doesn't exist, voucher expired, FF account doesn't exist, insufficient miles in FF account

Figure 1.6. Example of Structured Natural Language Specification for a Single Use Case

The "As a <user> I want <capability> so that <benefit>" format and decision tables are other examples. (See also [5, c4] [6, c12, c16] [7, c2-5].)

4.3. Acceptance Criteria-Based Requirements Specification

This general approach includes two specific variants: Acceptance Test Driven Development (ATDD) and Behavior Driven Development (BDD).

ATDD [2*, s3.2.3, s8.2] is a part of the larger Test Driven Development (TDD) approach. (See the Software Testing KA.). The main idea of TDD is that test cases precede construction. Therefore, no new production code is written and no existing code is modified unless at least one test case fails, either at the unit test level or at the acceptance test level. The ATDD process has three steps:

1. A unit of functionality (e.g., a user story) is selected for implementation.

2. One or more software engineers, one or more business domain experts, and possibly one or more QA/test professionals meet — before any production design or construction work is done — to agree on a set of test cases that must pass to show that the unit of functionality has been correctly implemented.

3. At least one of those acceptance test cases must fail on the existing software. The existence of at least one failing test case gives the software engineer(s) permission

to create or modify production code to pass all of the agreed-upon test cases. This step might require several iterations. The code may also be refactored during this step.

When all acceptance test cases have passed, and presumably all unit and integration test cases as well, then the unit of functionality is deemed to have been completely and correctly implemented. The ATDD process returns to step 1, where a new unit of functionality is selected, and the cycle repeats.

ATDD might seem to be a testing technique rather than a requirements specification technique. On the other hand, a test case has the general form of "When given input that looks like X, <u>we expect the software to produce</u> results that look like Y." The key is the underlined phrase, "we expect the software to produce." If we simply modify that phrase to say, "the software shall produce," as in "When given input that looks like X, <u>the software shall produce</u> results that look like Y," what first looked like a test case now looks like a requirement. Technically, one acceptance test case can encompass more than one single requirement, but the general idea holds that the ATDD test cases are essentially precise, unambiguous statements of requirements.

The BDD approach [19] is slightly more structured, and business domain experts typically prefer it over ATDD because it is less technical in appearance. In BDD, the unit of functionality is described as a user story, in a form such as this: "As a <role>, I want <goal/desire> so that <benefit>." This leads to the identification and specification of a set of "scenarios" in this form: "Given <some context> [and <possibly more context>], when <stimulus> then <outcome> [and <possibly more outcomes>]."

If the story is "As a bank customer, I want to withdraw cash from the Automated Teller Machine (ATM) so that I can get money without going to the bank," one scenario could be that "the account has a sufficient balance." This scenario could be detailed as "Given the account balance is $500, and the customer's bank card is valid, and the automated teller machine contains enough money in its cash box, when the Account Holder requests $100, then the ATM should dispense $100 and the account balance should be $400, and the customer's bank card should be returned."

Another scenario could be that "the account has an insufficient balance" and could be detailed as "Given the account balance is $50, and the customer's bank card is valid, and the automated teller machine contains enough money in its cash box, when the Account Holder requests $100, then the ATM should not dispense any money, and the ATM should say there is an insufficient balance, the balance should remain at $50, and the customer's bank card should be returned."

The goal of BDD is to have a comprehensive set of scenarios for each unit of functionality. In the withdrawing cash situation, additional scenarios for "The Bank Customer's bank card has been disabled" and "The ATM does not contain enough money in its cash box" would be necessary.

The acceptance test cases are obvious from the BDD scenarios.

Acceptance criteria-based requirements specification directly addresses the requirements ambiguity problem. Natural languages are inherently ambiguous, but test case language is not. In acceptance-based criteria requirements specification, the requirements are written using test case language, which is very precise. On the other hand, this does not inherently solve the incompleteness problem. However, combining ATDD or BDD with appropriate functional test coverage criteria, such as Domain Testing, Boundary Value Analysis and Pairwise Testing (see the Software Testing KA), can reduce the likelihood of requirements incompleteness. (See also [9, c1, c12].)

4.4. Model-Based Requirements Specification [1*, c12] [2*, c5] [4*]

Another approach to avoiding the inherent ambiguity of natural languages is to use modeling languages such as selected elements of the Unified Modeling Language (UML) or Systems Modeling Language (SysML). Much like the blueprints used in building construction, these modeling languages can be used in a computing technology-free manner to precisely and concisely specify functional requirements [9, c1-2]. This topic is closely related to the Software Engineering Models and Methods KA. Requirements models fall into two general categories:

1. Structural models for specifying policies to be enforced: These are logical class models as described in, for example, [9, c8]. They are also called conceptual data models, logical data models and entity-relationship diagrams.
2. Behavioral models for specifying processes to be carried out: These models include use case modeling as described in [9, c7], interaction diagrams as described in [9, c9] and state modeling as described in [9, c10]. Other examples are UML activity diagrams and data-flow modeling, as described in [1*, c12-13], [8], [10] and [18].

Model-based requirements specifications vary in the degree of model formality. Consider the following:

- Agile modeling (see, for example, [10]) is the least formal. Agile models can be little more than rough sketches whose goal is to communicate important information rather than demonstrate proper use of modeling notations. In this type of modeling, the effect of the communication is considered more important than the form of the communication.
- Semiformal modeling, for example [9, c6-12], provides a definition of the modeling language semantics ([9, Appendix L]), but that definition has not been formally proved to be complete and consistent.
- Formal modeling, for example, Z, the Vienna Development Method (VDM), Specification and Description Language (SDL) and [5, c6] have very precisely defined semantics that allow specifications to be mechanically analyzed for the presence or absence of specific properties to help avoid critical reasoning errors. The term *correctness by construction* has been used for development in this context. (See the Formal Methods section in the Software Engineering Models and Methods KA.)

Generally, the more formal a requirements model is, the less ambiguous it is, so software engineers are less likely to misinterpret the requirements. More formal requirements models can also be:

- more concise and compact;
- easier to translate into code, possibly mechanically;
- used as a basis for deriving acceptance test cases.

One important message in [4*] is that while formal modeling languages are stronger than semiformal and Agile modeling, formal notations can burden both the model creator and human readers. Wing's compromise is to use formally defined underpinnings (e.g., in Z) for surface syntaxes that are easier to read and write (e.g., UML statecharts).

4.5. Additional Attributes of Requirements [1*, c27pp462-463]

Over and above the basic requirements statements already described, documenting additional attributes for some or all requirements can be useful. This supplemental detail can help software engineers better interpret and manage the requirements [6, c16]. Possible additional attributes include the following:

- tag to support requirements tracing;
- description (additional details about the requirement);
- rationale (why the requirement is important);
- source (role or name of the stakeholder who imposed this requirement);

- use case or relevant triggering event;
- type (classification or category of the requirement — e.g., functional, quality of service);
- dependencies;
- conflicts;
- acceptance criteria;
- priority (see Requirements Prioritization later in this KA);
- stability (see Requirements Stability and Volatility later in this KA);
- whether the requirement is common or a variant for product family development (e.g., [20]);
- supporting materials;
- the requirement's change history.

Gilb's Planguage (short for Planning Language) [7] recommends attributes such as scale, meter, minimum, target, outstanding, past, trend and record.

4.6. Incremental and Comprehensive Requirements Specification

Projects that explicitly document requirements take one of two approaches. One can be called *incremental specification*. In this approach, a version of the requirements specification contains only the differences — additions, modifications and deletions — from the previous version. An advantage of this approach is that it can produce a smaller volume of written specifications.

The other approach can be called *comprehensive specification*. In this approach, each version's requirements specification contains all requirements, not just changes from the previous version. An advantage of this approach is that a reader can understand all requirements in a single document instead of having to keep track of cumulative additions, modifications and deletions across a series of specifications.

Some organizations combine these two approaches, producing intermediate releases (e.g., x.1, x.2 and x.3) that are specified incrementally and major releases (e.g., 1.0, 2.0 and 3.0) that are specified comprehensively. The reader never needs to go any further back than the requirements specifications for the last major release to obtain the complete set of specifications.

5. Requirements Validation [1*, c17] [2*, s4.5]

Requirements validation concerns gaining confidence that the requirements represent the stakeholders' true needs as they are currently understood (and possibly documented). Key questions include the following:

- do these represent all requirements relevant at this time?
- are any stated requirements not representative of stakeholder needs?
- are these requirements appropriately stated?
- are the requirements understandable, consistent and complete?
- does the requirements documentation conform to relevant standards?

Three methods for requirements validation tend to be used: requirements reviews, simulation and execution, and prototyping. (See also [5, c5] [6, c17] [9, c12].)

5.1. Requirements Reviews [1*, c17pp332-342] [2*, c4p130]

The most common way to validate is by reviewing or inspecting a requirements document. One or more reviewers are asked to look for errors, omissions, invalid assumptions, lack of clarity and deviation from accepted practice. Review from multiple perspectives is preferred:

- clients, customers and users check that their wants and needs are completely and accurately represented;
- other software engineers with expertise in requirements specification check that the document is clear and conforms to applicable standards;

- software engineers who will do architecture, design or construction of the software that satisfies these requirements check that the document is sufficient to support their work.

Providing checklists, quality criteria or a "definition of done" to the reviewers can guide them to focus on specific aspects of the requirements specification. (See Reviews and Audits in the Software Quality KA.)

5.2. Simulation and Execution

Nontechnical stakeholders might not want to spend time reviewing a specification in detail. Some specifications can be subjected to simulation or actual execution in place of or in addition to human review. To the extent that the requirements are formally specified (e.g., in a model-based specification), software engineers can hand interpret that specification and "execute" the specification. Given a sufficient set of demonstration scenarios, stakeholders can be convinced that the specification defines their policies and processes completely and accurately. (See [9, c12].)

5.3. Prototyping [1*, c17p342] [2*, c4p130]
If the requirements specification is not in a form that allows direct simulation or execution, an alternative is to have a software engineer build a prototype that concretely demonstrates some important dimension of an implementation. This demonstrates the software engineer's interpretation of those requirements.

Prototypes can help expose software engineers' assumptions and, where needed, give useful feedback on why they are wrong. For example, a user interface's dynamic behavior might be better understood through an animated prototype than through textual description or graphical models. However, a danger of prototyping is that cosmetic issues or quality problems with the prototype can distract the reviewers' attention from the core underlying functionality. Prototypes can also be costly to develop. However, if a prototype helps engineers avoid the waste caused by trying to satisfy erroneous requirements, its cost can be more easily justified.

6. Requirements Management Activities [1*, c27-28] [2*, s4.6]

Requirements development, as a whole, can be thought of as "reaching an agreement on what software is to be constructed." (See Figure 1.3.) In contrast, requirements management can be thought of as "maintaining that agreement over time." This topic examines requirements management. (See also [5, c9].)

6.1. Requirements Scrubbing
The goal of requirements scrubbing [22, c14, c32] is to find the smallest set of simply stated requirements that will meet stakeholder needs. Doing so will reduce the size and complexity of the solution, thus minimizing the effort, cost and schedule to deliver it. Requirements scrubbing involves eliminating requirements that:

- are out of scope;
- would not yield an adequate return on investment;
- are not that important.

Another important part of the process is to simplify unnecessarily complicated requirements.

In waterfall and other plan-based life cycles, requirements scrubbing can be coordinated with requirements reviews for validation; scrubbing should occur just before the validation review. In Agile life cycles, scrubbing happens implicitly in iteration planning; only the highest-priority requirements are brought into a sprint (iteration).

6.2. Requirements Change Control [1*, c28] [2*, s4.6]
Change control is central to managing requirements. This topic is closely linked to the Software

Configuration Management KA. (Refer to that chapter for more information.)

Projects using waterfall or other plan-based life cycles should have an explicit requirements change control process that includes:

- a means to request changes to previously agreed-upon requirements;
- an optional impact analysis stage to more thoroughly examine benefits and costs of a requested change;
- a responsible person or group who decides to accept, reject, or defer each requested change;
- a means to notify all affected stakeholders of that decision;
- a means to track accepted changes to closure.

All stakeholders must understand and agree that accepting a change means accepting its impact on schedule, resources and/or commensurate change in scope elsewhere in the project.

In contrast, requirements change management happens implicitly in Agile life cycles. In these life cycles, any request to change previously agreed-upon requirements becomes just another item on the product backlog. A request will only become "accepted" when it is prioritized highly enough to make it into an iteration (a sprint). (See also [5, c9] [22, c17].)

6.3. Scope Matching

Scope matching [22, c14] involves ensuring that the scope of requirements to architect, design and construct does not exceed any cost, schedule or staffing constraints on the project. When requirements scope exceeds the cost, schedule or staffing constraints, then either that scope must be reduced (presumably by removing a sufficient number of the lowest-priority requirements), capacity must be increased (by extending the schedule or increasing the budget and/or staffing), or some appropriate combination thereof must be negotiated.

In waterfall and other plan-based life cycles, scope matching can be coordinated with requirements validation; the scope matching should occur just before the validation review. In Agile life cycles, as long as some variant of *velocity-based sprint planning* is done, then the only work allowed into a sprint/iteration will be the work that can reasonably be expected to be completed during that sprint/iteration.

7. Practical Considerations

7.1. Iterative Nature of the Requirements Process [2*, s4.2]

Requirements for typical software not only have wide breadth; they must also have significant depth. The tension created by simultaneous breadth-wise and depth-wise requirements in real-world projects often prompts teams to perform requirements activities iteratively. At some points, elicitation and analysis favor expanding the breadth of requirements knowledge, while at other points, expanding the depth is called for. In practice, it is highly unlikely that all requirements work can be done in a single pass through the subject matter. (See also [6, c2, c9].)

7.2. Requirements Prioritization [1*, c16]

Prioritizing requirements is useful throughout a software project because it helps focus software engineers on delivering the most valuable functionality soonest. It also helps support intelligent trade-off decisions involving conflict resolution and scope matching. Prioritized requirements also help in maintenance beyond the initial development project itself. Defects raised against higher-priority requirements should probably be repaired before defects raised against lower-priority ones.

A variety of prioritization schemes are available. Answering a few key questions can help engineers choose the best approach. The first question is "What factors are relevant in determining the

priority of one requirement over another?" The following factors might be relevant to a project:

- value; desirability; client, customer and user satisfaction;
- undesirability; client, customer and user dissatisfaction (Kano model, below);
- cost to deliver;
- cost to maintain over the software's service life;
- technical risk of implementation;
- risk that users will not use it even if implemented.

The Kano model, which underlies [6, c17], shows that considering only value, desirability or satisfaction can lead to erroneous priorities. A better understanding of priorities comes from considering how unhappy stakeholders would be if that requirement were not satisfied. For example, consider a project to develop an email client. Two candidate requirements might relate to:

1. Having an effective spam filter
2. Handling attachments on emails

Prioritization must weigh both the satisfaction users will experience from having certain features and the dissatisfaction they will experience if they lack certain features. For example, users are more likely to be happy with an effective spam filter than with the ability to handle attachments, so the spam filter would be given a higher priority based on the satisfaction criterion. On the other hand, the inability to handle attachments would make many users extremely unhappy — much more so than not having an effective spam filter. When considering happiness, or satisfaction, from implementing features combined with unhappiness (or dissatisfaction) from not implementing certain features, developers would generally give handling attachments a higher priority than the effective spam filter.

The second key question is "How can we convert the set of relevant factors into an expression of priority?" The formula

$$Priority = \frac{\left(Value * (1 - Risk)\right)}{Cost}$$

is just one example of an *objective function* to do so. The choice of measurement schemes for the relevant factors can impose constraints on the objective function. (See Measurement Theory in Computing Foundations).

Once the priority of the requirements has been determined, those priorities must be specified in a way that can be communicated to all stakeholders. Several ways to do this are possible, including the following:

- enumerated scale (e.g., must have, should have, nice to have);
- numerical scale (e.g., 1 . . . 10);
- Lists that sort the requirements in decreasing priority order.

Effective requirement prioritization focuses on finding groups of requirements with similar priorities rather than creating overly rigorous measurement scales or debating small differences.

7.3. Requirements Tracing [1*, c29]

Requirements tracing can serve two potentially useful purposes. One is to serve as an accounting exercise that documents consistency between pairs of related project work products. An important question might be "For each identified software requirement, are there identified design elements intended to satisfy it?" If no identified design elements can be found, then either that requirement is not satisfied in that design or the design is correct and one or more stated requirements can be deleted. Similarly, "For each identified design element, are there identified requirements that cause it to exist?" If no identified requirements can be found, then either that design element is

unnecessary or the stated requirements are incomplete.

The other purpose is to assist in impact analysis of a proposed requirement change. If a particular system requirement were to change, for example, that system requirement could be traced to its linked software requirements. Not all linked software requirements would need to change. But each software requirement that would be affected could be traced to its linked design elements. Again, not all linked design elements would need to change. But each design element affected could be traced to the linked code. The affected software requirements, design elements and code units could also be traced to their linked test cases for further impact analysis. This helps establish a "footprint" for the volume of work needed to incorporate that change to the system requirement.

Software requirements can be traced back to source documentation such as system requirements, standards documents and other relevant specifications. Software requirements can also be traced forward to design elements and requirements-based test cases. Finally, software requirements can also be traced forward to sections in a user manual describing the implemented functionality. (See also [23].)

7.4. Requirements Stability and Volatility [2*, s4.6]
Some requirements are very stable; they will probably never change over the software's service life. Some requirements are less stable; they might change over the service life but might not change during the development project. For example, in a banking application, requirements for functions to calculate and credit interest to customers' accounts are likely to be more stable than requirements to support different tax-free accounts. The former reflects a banking domain's fundamental feature (that accounts can earn interest). At the same time, the latter may be rendered obsolete by a change in government legislation. Finally, some requirements can be very unstable; they can change during the project — possibly more than once. It is useful to assess the likelihood that a requirement will change in a given time. Identifying potentially volatile requirements helps the software engineer establish a design more tolerant of change, (e.g., [20]). (See also [9, c4].)

7.5. Measuring Requirements
[1*, c19]
As a practical matter, it may be 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 new development project or the size of a change in requirements and in estimating the cost of development or maintenance tasks (e.g., [9, c23]), 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 can be found in the Software Engineering Process KA.

Many 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 and a requirements specification document as a whole can be derived from the desirable properties discussed in Section 3.1, Basic Requirements Analysis, earlier in this KA.

7.6. Requirements Process Quality and Improvement [1*, c31]
This topic concerns assessing the quality and improvement of the requirements process. Its purpose is to emphasize the key role of the requirements process in a software product's cost and timeliness and in customer satisfaction. Furthermore, it helps align the requirements process with quality standards and process improvement models for software and systems. Process quality and improvement are closely related to both the Software Quality KA and

Software Engineering Process KA, comprising the following:

- requirements process coverage by process improvement standards and models;
- requirements process measures and benchmarking;
- improvement planning and implementation;
- security/CIA (confidentiality, integrity, and availability) improvement/planning and implementation.

8. Software Requirements Tools [1*, c30]

Tools that help software engineers deal with software requirements fall broadly into three categories: requirements management tools, requirements modeling tools and functional test case generation tools, as discussed below.

8.1. Requirements Management Tools [1*, c30pp506-510]

Requirements management tools support various activities, including storing requirements attributes, tracing, document generation and change control. Indeed, tracing and change control might only be practical when supported by a tool. Because requirements management is fundamental to good requirements practice, many organizations have invested in tools. However, many more manage their requirements in more ad hoc and generally less satisfactory ways (e.g., spreadsheets). (See also [5, c8].)

8.2. Requirements Modeling Tools [1*, c30p506] [2*, s12.3.3]

At a minimum, a requirements modeling tool supports creating, modifying and publishing model-based requirements specifications. Some tools extend that by also providing static analysis (e.g., syntax correctness, completeness and consistency). Formal analysis requires tool support to be practicable for anything other than trivial systems, and tools generally fall into two categories: theorem provers or model checkers. In neither case can proof be fully automated, and the competence in formal reasoning needed to use the tools restricts the wider formal analysis. Some tools also dynamically execute a specification (simulation).

8.3. Functional Test Case Generation Tools

The more formally defined a requirements specification language is, the more likely it is that functional test cases can be at least partially derived mechanically. For example, converting BDD scenarios into test cases is not difficult. Another example involves state models. Positive test cases can be derived for each defined transition in that kind of model. Negative test cases can be derived from the state and event combinations that do not appear. (See Section 8.2, Testing Tools in the Testing KA, for more information.) A process for deriving test cases from UML requirements models can be found in [9, c12].

In the most general case, such tools can only generate test case inputs. Determining an expected result is not always possible. Additional business domain expertise might be necessary.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Wiegers 2013 [1*]	Sommerville 2018 [2*]	Tockey 2005 [3*]	Wing 1990 [4*]
1. Software Requirements Fundamentals				
1.1. Definition of a Software Requirement	c1pp5-6	c4p102		
1.2. Categories of Software Requirements	c1pp7-12	s4.1		
1.3. Software Product Requirements and Software Project Requirements	c1pp14-15			
1.4. Functional Requirements	c1p9	s4.1.1		
1.5. Nonfunctional Requirements	c1pp10-11	s4.1.2		
1.6. Technology Constraints				
1.7. Quality of Service Constraints				
1.8. Why Categorize Requirements This Way?				
1.9. System Requirements and Software Requirements				
1.10. Derived Requirements				
1.11. Software Requirements Activities	c1pp15-18	s4.2		
2. Requirements Elicitation				
2.1. Requirements Sources	c6	s4.3		
2.2. Common Requirements Elicitation Techniques	c7	s4.3		
3. Requirements Analysis				
3.1. Basic Requirements Analysis	c8-9			
3.2. Economics of Quality of Service Constraints			c1-27	
3.3. Formal Analysis		s12.3.2-12.3.3		
3.4. Addressing Conflict in Requirements				
4. Requirements Specification				
4.1. Unstructured Natural Language Requirements Specification	c11	s4.4.1		
4.2. Structured Natural Language Requirements Specification	c8	s4.4.2		
4.3. Acceptance Criteria-Based Requirements Specification		s3.2.3, s8.2		
4.4. Model-Based Requirements Specification	c12	c5		pp8-11
4.5. Additional Attributes of Requirements	c27pp462-463			
4.6. Incremental and Comprehensive				

Requirements Specification				
5. Requirements Validation				
5.1. Requirements Reviews	c17pp332-342	c4p130		
5.2. Simulation and Execution				
5.3. Prototyping	c17p342	c4p130		
6. Requirements Management Activities				
6.1. Requirements Scrubbing				
6.2. Requirements Change Control	c28	s4.6		
6.3. Scope Matching				
7. Practical Considerations				
7.1. Iterative Nature of the Requirements Process		s4.2		
7.2. Requirements Prioritization	c16			
7.3. Requirements Tracing	c29			
7.4. Requirements Stability and Volatility		s4.6		
7.5. Measuring Requirements	c19			
7.6. Requirements Process Quality and Improvement	c31			
8. Software Requirements Tools				
8.1. Requirements Management Tools	c30pp506-510			
8.2. Requirements Modeling Tools	c30p506	s12.3.3		
8.3. Functional Test Case Generation Tools				

FURTHER READING

P. LaPlante, *Requirements Engineering for Software and Systems* [5].

This book is one potential alternative to [1*], offering a comprehensive discussion of software requirements.

S. Robertson and J. Robertson, *Mastering the Requirements Process: Getting Requirements Right* [6].

This book is another potential alternative to [1*], offering a comprehensive discussion of software requirements.

T. Gilb, *Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage* [7].

This book presents a unique perspective on requirements, emphasizing requirements precision and completeness along with a strong business value-driven motivation.

K. Wiegers, *Software Development Pearls: Lessons from Fifty Years of Software Experience* [21].

This book is a compendium of important but often unrecognized key success factors based on Dr. Wiegers' extensive real-world experience. Chapter 2 is specific to software requirements.

R. Fisher and W. Ury, *Getting to Yes* [25].

This book is a classic reference on principled negotiation and conflict resolution that serves as one good basis for addressing inevitable conflict in software requirements when there are multiple stakeholders.

N. Ahmad, *Effects of Electronic Communication on the Elicitation of Tacit Knowledge in Interview Techniques for Small Software Developments* [29].

This doctoral thesis shows how using four different types of electronic communication tools to discuss interview agenda details with interviewees before conducting semi-structured interviews for requirements elicitation improved elicitation of tacit (hidden) knowledge.

REFERENCES

[1*] K. E. Wiegers and J. Beatty, *Software Requirements*, 3rd ed., Redmond, WA: Microsoft Press, 2013.

[2*] I. Sommerville, *Software Engineering*, 10th ed., New York: Addison-Wesley, 2018.

[3*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, Boston, MA: Addison-Wesley, 2005.

[4*] J. M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, vol. 23, no. 9, 1990, pp. 8, 10-23.

[5] P. LaPlante, *Requirements Engineering for Software and Systems*, 3rd ed., Boca Raton, FL: CRC Press, 2018.

[6] S. Robertson and J. Robertson, *Mastering the Requirements Process: Getting Requirements Right*, Upper Saddle River, NJ: Addison-Wesley, 2013.

[7] T. Gilb, *Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*, Oxford, UK: Elsevier Butterworth-Heinemann, 2005.

[8] E. Yourdon, *Modern Structured Analysis*, Englewood Cliffs, NJ: Prentice-Hall, 1989.

[9] S. Tockey, *How to Engineer Software*, Hoboken, NJ: Wiley, 2019.

[10] S. Ambler, *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*, Hoboken, NJ: Wiley, 2002.

[11] A. Cockburn, *Writing Effective Use Cases*, Upper Saddle River, NJ: Addison-Wesley, 2000.

[12] L. Constantine and L. Lockwood, *Software for Use*, Reading, MA: Addison-Wesley, 2000.

[13] J. Wood and D. Silver, *Joint Application Development*, New York, NY: Wiley, 1995.

[14] E. Gottesdiener, *Requirements by Collaboration*, Boston, MA: Addison-Wesley, 2002.

[15] J. Terninko, *Step by Step QFD*, 2nd ed., Boca Raton, FL: CRC Press, 1997.

[16] G. Salvendy, *Handbook of Human Factors*, 4th ed., Hoboken, NJ: Wiley, 2012.

[17] T. Brown and B. Katz, *Change by Design: How Design Thinking Transforms Organizations and Inspires Innovation*, Revised and updated ed., New York, NY: Harper Collins, 2019.

[18] S. McMenamin and J. Palmer, *Essential Systems Analysis*, New York, NY: Yourdon Press, 1984.

[19] J. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*, Shelter Island, NY: Manning Publications, 2015.

[20] D. Weiss and C. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Reading, MA: Addison-Wesley, 1999.

[21] K. Wiegers, *Software Development Pearls: Lessons from Fifty Years of Software Experience*, Boston, MA: Addison-Wesley Professional, 2021.

[22] S. McConnell, *Rapid Development*, Redmond, WA: Microsoft Press, 1996.

[23] O. Gotel and C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem," presented at the Proceedings of

the 1st International Conference on Requirements Engineering, 1994.

[24] INCOSE, *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*, 3.2.2 ed., San Diego, US: International Council on Systems Engineering, 2012.

[25] R. Fisher and W. Ury, *Getting to Yes*, 3rd ed., New York, NY: Penguin, 2011.

[26] ____, ISO/IEC/IEEE 29148 "Systems and software engineering – Life cycle processes – Requirements engineering," *International Standards Organization*, 2018.

[27] ____, ISO/IEC 25010: "System and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models," *International Standards Organization*, 2011.

[28] ____, Software and Systems Engineering Vocabulary, https://pascal.computer.org/sev_display/index.action.

[29] N. Ahmad, *Effects of Electronic Communication on the Elicitation of Tacit Knowledge in Interview Techniques for Small Software Developments*, doctoral thesis, University of Huddersfield, 2021.

chapter 2

v2.7

Software Architecture

ACRONYMS

AD	architecture description
ADL	architecture description language
API	application programming interface
ASR	architecturally significant requirement
IDL	interface description language
MVC	model view controller

INTRODUCTION

This chapter considers software architecture from several perspectives: concepts; representation and work products; context, process and methods; and analysis and evaluation.

In contrast to the previous edition, this edition creates a Software Architecture Knowledge Area (KA), separate from the Software Design KA, because of the significant interest and growth of the discipline since the 1990s.

BREAKDOWN OF TOPICS FOR SOFTWARE ARCHITECTURE

The breakdown of topics for the Software Architecture KA is shown in Fig. 1.

Fig. 1. Breakdown of Topics for the Software Architecture KA

1 SOFTWARE ARCHITECTURE FUNDAMENTALS

1.1 The Senses of "Architecture"

Software engineering and related disciplines use many senses of "architecture". First, "architecture" often refers to a discipline: the art and science of constructing things — in this case, software-intensive systems. The discipline involves concepts, principles, processes and methods the community has discovered and adopted.

Second, *architecture* refers to the various processes through which that discipline is realized. In this KA, we distinguish *architecture design* as a specific phase in the life cycle encompassing a particular set of activities, and we distinguish it from the wider *architecting* processes that span the life cycle. Both are discussed in topic *Software Architecture Processes*.

Third, "architecture" refers to the *outcome* of applying architectural design discipline and processes to devise architectures for software systems. Architectures as outcomes are expressed in *architecture descriptions*. This is discussed in topic *Software Architecture Description*. The concept of architecture has evolved, and many definitions are in use today. One early definition of architecture, from 1990, emphasized software structure:

Architecture. The organizational structure of a system or component. [IEEE Std 610.12–1990, *IEEE Glossary of Software Engineering Terminology*]

This definition did not do justice to evolving thinking about architecture; e.g., this definition does not allow us to distinguish the detailed design of a module from its Makefile. Either example reflects an *organizational structure* of the software system or component but should not be considered architecture. Moreover, emphasis on the structure was often limited to the code's structure and failed to encompass all the structures of the software system:

> The software architecture of a system is the set of structures needed to reason about the system. These structures comprise software elements, relations among them, and properties of both. [2*]

During the mid-1990s, however, software architecture emerged as a broader discipline involving a more generic study of software structures and architectures. Many software system structures are not directly reflected in the code structure. Both types of structure have implications for the system as a whole: What behaviors is the system capable of? What interactions does it have with other systems? How are properties like safety and security handled? The recognition that software contains many different structures has prompted discussion of a number of interesting concepts about software architecture (and software design more generally).

> **architecture** (of a system). fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution [15*]

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

Key ideas in that definition are the following: (1) Architecture is about what is *fundamental* to a software system; not every element, interconnection, or interface is considered fundamental. (2) Architecture considers a system *in its environment.* Much like building architecture, software architecture is outward-looking; it considers a system's context beyond its boundaries to consider the people, organizations, software, hardware and other devices with which the system must interact.

1.2 Stakeholders and Concerns

A software system has many *stakeholders* with varying roles and interests relative to that system. These varying interests are termed *concerns*, following Dijkstra's *separation of concerns*:

> Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained — on the contrary! — by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "[focusing] one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track-minded simultaneously. [9]

What is fundamental about a system varies according to stakeholders' concerns and roles. The software structures, therefore, also vary with stakeholder roles and concerns. (See also topic *Design Methods* in Software Design KA.)

A software system's customer is most interested in when the system will be ready and how much it will cost to build and operate. Users are most interested in what it does and how to use it. Designers and programmers building the system have their own concerns, such as whether the algorithm will meet the requirements. And those responsible for ensuring the system is safe to operate have different concerns.

Concerns encompass a broad range of issues, including those depicted in Fig. 2.

Many software considerations fall within the general idea of concerns and manifest in various forms, such as functional and "nonfunctional" requirements, emergent properties, quality attributes and "-ilities". Topic 2, *Software Architecture Description*, shows how concerns shape architecture and the work products describing those architectures.

affordability, agility, assurance, autonomy, availability, behavior, business goals and strategies, complexity, compliance with regulation, concurrency, control, cost, data accessibility, deployability, disposability, energy efficiency, evolvability, extensibility, feasibility, flexibility, functionality, information assurance, inter-process communication, interoperability, known limitations, maintainability, modifiability, modularity, openness, performance, privacy, quality of service, reliability, resource utilization, reusability, safety, scalability, schedule, security, system modes, software structure, subsystem integration, sustainability, system features, testability, usability, usage, user experience

Fig. 2. Examples of Architectural Concerns

1.3 Uses of Architecture

A principal use of a software system's architecture is to give those working with it a shared understanding of the system to guide its design and construction. An architecture also serves as a preliminary conception of the software system that provides a basis to analyze and evaluate alternatives. A third common usage is to enable reverse engineering (or *reverse architecting*) by helping those working with it to understand an existing software system before undertaking maintenance, enhancement or modification. To support these uses, the architecture should be documented (see topic *Software Architecture Description*).

Conway's Law posits that "organizations which design systems . . . are constrained to produce designs which are copies of the communication structures of these organizations." [8] This suggests that architectures often mirror the structure of the organizations that developed them. Depending on the software system and the organization, this can be a strength or a weakness. The architecture can enhance communication within a large team or compromise it. Each part of the organization can base its planning, costing and scheduling activities upon its knowledge of the architecture. Creating a well-planned and documented architecture is one approach to increasing the applicability and reusability of software designs and components. The architecture forms the basis for design families of programs or software product lines. This can be done by identifying commonalities among members of such families and by designing reusable and customizable components to account for the variability among family members.

2 Software Architecture Description

In topic 1, *Software Architecture Fundamentals*, a software architecture was defined as the fundamental concepts or properties of a software system in its environment. But each stakeholder can have a different notion of what is fundamental to that software system, given their perspective. Having a mental model of a system's architecture is perhaps fine for small systems and for individuals working alone. However, for large, complex systems developed and operated by teams, a tangible representation is invaluable, especially as the conception of the system evolves, and as people join or leave the team. Having a concrete representation as a work product can also serve as a basis to analyze the architecture, organize its design and guide its implementation. These work products are called *architecture descriptions (ADs)*.

An AD documents an architecture for a software system. It is targeted to those stakeholders of the system who have concerns about the software system which are answered by the architecture. As noted in topic 1, *Software Architecture Fundamentals*, a primary audience comprises the designers, engineers and programmers whose concerns pertain to constructing the system. For these stakeholders, the AD serves as a *blueprint* to guide the construction of the software system. For others, the AD is a basis for their work—for example, testing and quality assurance, certification, deployment, operation, and maintenance and future evolution.

Historically, ADs used text and informal diagrams to convey the architecture. However, the diversity of stakeholder audiences and their different concerns have led to a diversity of representations of the architecture.. Often, these representations are specialized based upon existing practices of the communities or disciplines involved to effectively address this variety of stakeholders and concerns (see Software Design KA and Software Engineering Models and Methods KA). These various renderings are called *architecture views*.

2.1 Architecture Views and Viewpoints

An *architecture view* represents one or more aspects of an architecture to address one or more concerns [23*]. Views address distinct concerns — for example, a logical view (depicts how the system will satisfy the functional requirements); a process view (depicts how the system will use concurrency); a physical view (depicts how the system is to be deployed and distributed) and a development view (depicts how the top-level design is broken down into implementation units, the dependencies among those units and how the implementation is to be constructed). Separating concerns by view allows interested stakeholders to focus on a few things at a time and offers a means of managing the architecture's understandability and overall complexity.

As architecture practice has evolved from the use of text and informal diagrams to the use of more rigorous representations. Each architecture view depicts architectural elements of the system using well-defined conventions, notations and models [23*]. The conventions for each view are documented as an *architecture viewpoint* [15]. Viewpoints guide the creation, interpretation and uses of architecture views. Each viewpoint links stakeholder audience concerns with a set of conventions. In model-based architecting, each view can be machine-checked against its viewpoint.

Common viewpoints include the module viewpoint, used to express a software system's implementation in terms of its modules and their organization [2*]; the component and connector viewpoint, used to express the software's large-scale runtime organization and interactions [2*]; the logical viewpoint, used to express fundamental concepts of the software's domain and capability [16]; the scenarios/use cases viewpoint, used to express how users interact with the system [16]; the information viewpoint, used to express a system's key information elements and how they are accessed and stored [23*]; and the deployment viewpoint, used to express how a system is configured and deployed for operation [23*]. Other documented viewpoints include viewpoints for availability, behavior, communications, exception handling, performance, reliability, safety and security.

Each viewpoint provides a vocabulary or language for talking about a set of concerns and the mechanisms for addressing them. The viewpoint language gives stakeholders a shared means of expression. Viewpoints need not be limited to one software system but are reusable by an organization or application community for many similar systems. When generic representations such as Unified Modeling Language (UML) are used, they can be specialized to the system, its domain or the organizations involved. (See section 2.3, *Architecture Description Languages and Architecture Frameworks*.)

Beyond specifying forms of representation, an architecture viewpoint can capture the ways of working within a discipline or community of practice. For example, a software reliability viewpoint captures existing practices from the software reliability community for identifying and analyzing reliability issues, formulating alternatives and synthesizing and representing solutions. Like engineering handbooks, generic and specialized viewpoints provide a means to document repeatable or reusable approaches to recurring software issues.

Architecture descriptions frequently use *multiple* architecture views to represent the diverse structures needed to address different stakeholders' various concerns. A consequence of introducing multiple views into an AD is a potential mismatch between the views. Are they consistent? Are they describing the same system? This has been called the *multiple views problem* [24]. To integrate views into a whole, linkages or other kinds of traceability are used to cross-reference view elements to achieve consistency [15]. Viewpoints often include rules for establishing consistency or other relationships among views.

2.2 Architecture Styles and Patterns

Inspired by its use in the long history of the architecture of buildings, an *architectural style* is a particular manner of construction yielding a software system's characteristic features. An architectural style often expresses a software system's large-scale organization. In contrast, an *architectural pattern* expresses a common solution to a recurring problem within the context of a software system. Patterns are discussed in section 4.4 of Software Design KA.

Various architectural styles and patterns have been documented [7, 24]:

- General structures (e.g., layered, call-and-return, pipes and filters, blackboard, services and microservices)
- Distributed systems (e.g., client-server, n-tier, broker, publish-subscribe)
- Method-driven (e.g., object-oriented, event-driven, data flow)
- User-computer interaction (e.g., model-view-controller, presentation-abstraction-control)
- Adaptive systems (e.g., microkernel, reflection and meta-level architectures)
- Virtual machines (e.g., interpreters, rule-based, process control)

There is no strict dividing line between styles and patterns. Roughly, a style is strategic: An architectural style applies to a whole system or major parts of a system, whereas patterns are often tactical; a pattern applies within a software system, perhaps repeatedly, to provide a solution to a particular computing problem in a given context. In relation to architecture viewpoints, which provide the languages for talking about various aspects of software systems, a unifying notion is that both patterns and styles are *idioms* in those languages for expressing particular aspects of architectures (and designs, see section 4.4 Software Design KA). An architectural pattern or style uses a vocabulary, drawn from the viewpoint's language, in a specified way, to talk about view elements, including element and relation types and their instances, and constraints on combining them [15, 24]. In this way, viewpoints, patterns and styles are mechanisms for codifying best practices to facilitate reuse.

2.3 Architecture Description Languages and Architecture Frameworks

An *architecture description language* (ADL) is a domain-specific language for expressing software architectures. ADLs arose from module interconnection languages [22] for programming in the large. Some ADLs target a single application domain or architectural style (such as MetaH for avionics systems in an event-driven style), others are wide spectrum to frame concerns across the enterprise (such as ArchiMate™). UML has frequently been used as an ADL. ADLs often provide capabilities beyond description to enable architecture analysis or code generation.

An *architecture framework* captures the "conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders" [15]. Frameworks codify best practices within a specific domain and are implemented as an interlocking set of viewpoints or ADLs. Examples are OMG Unified Architecture Framework (UAF®) and ISO Reference Model for Open Distributed Processing (RM-ODP).

2.4 Architecture as Significant Decisions

Architectural design is a creative process. During this activity, architects make many decisions that profoundly affect the architecture, the downstream development process and the software system. Many factors affect decision-making, including prominent concerns of stakeholders for the software system, its requirements, and the available resources during development and throughout the life cycle. The impact on quality attributes and trade-offs among competing quality attributes are often the basis for design decisions.

The architectural design activity produces a network of decisions as its outcome, with some decisions deriving from prior decisions. Decision analysis provides one approach to architecture evaluation. Decisions should be explicitly documented, along with an explanation of the rationale for each nontrivial decision.

Architecture rationale captures *why* an architectural decision was made. This includes assumptions made before the decision, alternatives considered, and trade-offs or criteria used to select an approach and reject others. Recording rejected decisions and the reasons for their rejection can also be useful. In the future, this could either prevent a software development from making a poor decision—one rejected earlier for forgotten reasons—or allow the development to recognize that relevant conditions have changed and that they can revisit the decision.

Architectural technical debt has been introduced to reflect that today's decisions for an architecture may have significant consequences later in the software system's life cycle. Decisions deferred can compromise its maintainability or the future evolvability, and that debt will have to be paid—typically by others, not necessarily by those who caused the debt. [Provide TD example] Such debt has an economic impact on the system's future development and operations. Architectural technical debt can be analyzed and managed, like other concerns, using models and viewpoints [17].

3 Software Architecture Process

This section outlines a general model of an architectural design process. It is used to demonstrate how architectural design fits into the general context of software engineering processes (see Software Engineering Process KA) and as a framework for understanding the many architecture methods currently in use.

3.1 Architecture in Context

Architecture occurs in several contexts. In the traditional life cycle, there is an architectural design stage driven by software system requirements (see Software Requirements KA). Some requirements will be *architectural drivers*, influencing major decisions about the architecture, while other requirements are deferred to subsequent stages of the software process, such as design or construction.

In product line or product family settings, a product line/family architecture is developed against a basic set of needs, requirements and other factors. That architecture will be the starting point for one or more product instances developed against specific product requirements, building upon the product baseline.

In agile approaches, there is not usually an architecture design stage. The only architecture description might be the code itself. In some agile practices, the software architecture is said to "emerge" from coding the system based on user stories through a rapid series of development cycles. Although this approach has had some success with user-centric information systems, it is difficult to ensure an adequate architecture *emerges* for other classes of applications, such as embedded and cyber-physical systems, when critical architectural properties might not be articulated by any user stories.

In enterprise and system-of-systems contexts, as in product lines and families, the overarching architecture (of the enterprise, system or product line/family) provides primary requirements and guidance on the form and constraints upon the software architecture. This baseline can be enforced through specifications, additional requirements, application programming interfaces (APIs) or conformance suites.

3.1.1 Relation of Architecture to Design

Design and architecture are often blurred. It has sometimes been said that architecture is the set of decisions that one cannot

trust to designers. In fact, architecture emerged out of software design as the discipline matured, largely since the 1990s. There are various contrasts: design often focuses on an established set of requirements, whereas architecture often must shape the requirements through negotiation with stakeholders and requirements analysis. In addition, architecture often must recognize and address a wider range of concerns that may or may not end up as requirements on the software system of interest.

3.2 Architectural Design

Architectural design is the application of design principles and methods within a process to create and document a software architecture. There are many architecture methods for carrying out this activity. This section describes a general model of architectural design underlying various architecture methods based upon [12].

Architectural design involves identifying a system's major components; their responsibilities, properties, and interfaces; and the relationships and interactions among them and with the environment. In architectural design, fundamentals of the system are decided, but other aspects, such as the internal details of major components are deferred.

Typical concerns in architectural design include the following:

- Overall architecture styles and computing paradigms
- Large-scale refinement of the system into key components
- Communication and interaction among components
- Allocation of concerns and design responsibilities to components
- Component interfaces
- Understanding and analysis of scaling and performance properties, resource consumption properties, and reliability properties
- Large-scale/system-wide approaches to dominating concerns (such as safety and security, where applicable)

An overview of architectural design is presented in Fig. 3.

Architectural design is iterative, comprising three major activities: analysis, synthesis and evaluation. Often, all three major activities are performed concurrently at various levels of granularity.

Fig. 3. A general model of architectural design

3.2.1 Architecture Analysis

Architecture analysis formulates architecturally significant requirements (ASRs): any ''requirement upon a software system which influences its architecture'' [19]. Architecture analysis is based on identified concerns and on understanding the software's context, including known requirements, stakeholder needs and the environment's constraints. ASRs reflect the design problems the architecture must solve. Often the combination of initial requirements and known constraints cannot be satisfied without consequences to cost, schedule, etc. In such cases, negotiation is used to modify incoming needs, requirements and expectations to make solutions possible. Architecture analysis produces ASRs, initial system-wide decisions and any overarching system principles derived from the context (see *Architecture in Context*).

3.2.2 Architecture Synthesis

Architecture synthesis develops candidate solutions in response to the outcomes of architecture analysis. Synthesis proceeds by working out detailed solutions to design problems identified by ASRs, and makes trade-offs to accommodate interactions between those solutions. These outcomes feed back to architecture analysis resulting in elaborated ASRs, principles and decisions which then lead to further detailed solution elements.

3.2.3 Architecture Evaluation

Architecture evaluation validates whether the chosen solutions satisfy ASRs and when and where rework is needed.

Architecture evaluation methods are discussed in topic 4 *Software Architecture Evaluation*.

3.3 Architecture Methods and Tactics

There are a number of documented architecture methods (see Further Reading for a list).

3.4 Architecting in the Large

Architectural design denotes as specific stage of the life cycle, but is only one part of software architecting. Software architecting does not occur in a vacuum, as noted in section 3.1 *Architecture in Context*, but in an environment that often includes other architectures. For example, an application architecture should conform to an enterprise architecture; to "play well" in a system of systems, the architecture of each constituent system should conform to the system of systems architecture. In such cases, these relations need to be reflected as ASRs on the software being architected. Many software architecting activities and principles are not limited to software but equally apply to systems and enterprise architecting [18]. Weinreich and Buchgeher have extended Hofmeister et al.'s model used in section 3.2 *Architectural Design* to include these activities [26]:

- *architecture implementation*: overseeing implementation and certifying that implementations conform to the architecture
- *architecture maintenance*: managing and extending the architecture following its implementation
- *architecture management*: managing an organization's portfolio of interrelated architectures
- *architecture knowledge management*: extracting, maintaining, sharing and exploiting reusable architecture assets, including decisions, lessons learned, specifications and documentation across the organization

4 Software Architecture Evaluation

Coordinate this section with Software Design Reviews and with Software Quality

4.1 Goodness in Architecture

Architecture analysis takes place throughout the process of creating and sustaining an architecture. *Architecture evaluation* is typically undertaken by third parties at determined milestones as a form of assessment.

Given the multi-concern, multi-disciplinary nature of software architecture, there are many aspects to what makes an architecture "good." The Roman architect Vitruvius posited that all buildings should have the attributes of *firmitas*, *utilitas* and *venustas* (translated from Latin as strength, utility and beauty).

Of a software system and its architecture, one can ask:

- Is it robust over its lifetime and possible evolution?
- Is it fit for its intended use?
- Is it feasible and cost-effective to construct software systems using this architecture?
- Is it, if not beautiful, then at least clear and understandable to those who must construct, use and maintain the software?

Each architecture concern may be a basis for evaluation. Evaluation is conducted against requirements (when available) or against need, expectations and norms (in other situations). A "good" architecture should address not only the distinct concerns of its stakeholders, but also the consequences of their interactions. For example: a secure architecture may be excessively costly to build and verify; an easy-to-build architecture may not be maintainable over the system's lifetime if it cannot incorporate new technologies.

4.2 Reasoning about Architectures

Each architecture concern has a distinct basis for evaluation. Evaluation is most effective when it is based upon robust, existing architecture descriptions. ADs can be queried, examined and analyzed. For example, evaluation of functionality or behavior benefits from having an explicit architecture view or other representation of that aspect of the system to study. Specialized concerns such as reliability, safety and security often rely on specialized representations from the respective discipline.

Often architecture documentation is unfinished, incomplete, out of date or nonexistent. In such cases, the evaluation effort must rely on the knowledge of participants as a primary information source.

Use cases are frequently used to check an architecture's completeness and consistency (see Software Engineering Models and Methods KA) by comparing the steps in the use case to the software architecture elements that would be involved in carrying out those steps [16].

(For a general framework for reasoning about various concerns, see Bass et al. [3].)

4.3 Architecture Reviews

Architecture reviews are an effective approach to assess an architecture's status and quality and identify risks by assessing one or more architecture concerns [1]. Many reviews are informal or expertise-based, and some are more structured, organized around a checklist of topics to cover. Parnas and Weiss proposed an effective approach to conducting reviews, called *active reviews* [21], where instead of checklists, each evaluation item entails a specific activity by a reviewer to obtain the needed information.

Many organizations have institutionalized architecture review practices. For example, an industry group developed a framework for defining, conducting and documenting architecture reviews and their outcomes [18].

4.4 Architecture Metrics

An *architecture metric* is a quantitative measure of a characteristic of an architecture. Various architecture metrics have been defined. Many of these originated as design or code metrics that have been "lifted" to apply to architecture. Metrics include component dependency, cyclicity and cyclomatic complexity, internal module complexity, module coupling and

cohesion, levels of nesting, and compliance with the use of patterns, styles and (required) APIs.

In continuous development paradigms (such as DevOps), other metrics have evolved that focus not on the architecture directly but on the responsiveness of the process, such as metrics for lead time for changes, deployment frequency, mean time to restore service, and change failure rate—as indicative of the state of the architecture.

MATRIX: TOPICS VS. REFERENCE MATERIAL

cX refers to chapter X

	Bass et al. [2*]	**Budgen [6*]**	**Rozanski Woods [23*]**	**Sommerville [25*]**	**See also**
Software Architecture Fundamentals			c2		
The senses of "architecture"	c1				[18]
Stakeholders and Concerns	c3-14		c8, c9		
Roles of Architecture	c24		c30		
Architecture Description	c22			c6	
Architecture Views and Viewpoints		c7	c3, c12, c13	c6.2	
Architectural Styles and Patterns		c6	c11	c6.3	[7]
Architecture Description Languages and Architecture Frameworks					[15]
Architecture as Significant Decisions			c8	c6.1	[16]
Architecture Processes			c7		
Architecture in Context					[18]
Architectural Design	c19-20				[12]
Architecture Methods and Tactics					see Further Reading
Architecting in the Large					[18]

Architecture Evaluation	c21		c14		
Goodness in Architecture	c2				
Reasoning about Architectures			c10		[3]
Architecture Reviews	c21				[1,19]
Architecture Metrics	c23				

FURTHER READINGS

Reviewers are encouraged to submit updates to this list.

Len Bass et al. Software Architecture in Practice. 4th ed. [2*]

This book introduces concepts and best practices of software architecture, meaning how software is structured and how the software's components interact. The book addresses several quality concerns in detail, including: Availability, Deployability, Energy Efficiency, Modifiability, Performance, Testability and Usability. The authors offer best practices focusing on architectural design, architecture description, architecture evaluation and managing architecture technical debt. They also emphasize the importance of the business context in which large software is designed. In doing so, they present software architecture in a real-world setting, reflecting both the opportunities and constraints that organizations encounter.

Philippe Kruchten, "The 4+1 View Model of Architecture" [16].

This seminal paper organizes an approach to architecture description using five architecture viewpoints. The first four are used to produce the logical view, the development view, the process view, and the physical view. These are integrated through selected use cases or scenarios to illustrate the architecture. Hence, the model results in 4+1 views. The views are used to describe the software as envisioned by different stakeholders—such as end-users, developers, and project managers.

Nick Rozanski & Eoin Woods, Software Systems Architecture [23*]

This is a handbook for the software systems architect. It develops key concepts of stakeholder, concern, architecture description, architecture viewpoint and architecture view, architecture patterns and styles, with examples. It provides an end-to-end architecting process. The authors provide a catalog of ready-to-use, practical viewpoints for the architect to employ that are applicable to a wide range of systems. The book is filled with guidance for applying these concepts and methods.

Simon Brown, Software Architecture for Developers [5]

Brown provides an overview of software architecture topics from the perspective of a developer. He discusses common architecture drivers (functional requirements, quality concerns, constraints and architecture principles. He has an in-depth discussion of the role of the architect in a development setting and requisite knowledge and skills for architects. He focuses on the practical issues of architecture in the delivery process and on managing risk. An appendix provides a case study.

George Fairbanks, Just Enough Software Architecture: A risk-driven approach 2010 [11]

Fairbanks offers a risk-driven approach to architecting within the context of development: do just enough software architecture to mitigate the identified risks where those risks could result from a small solution space, from extremely demanding quality requirements or from possible high-risk failures. The risk-driven approach is harmonious with low-ceremony and agile approaches. Architecting, as argued by Fairbanks, is not just for architects—but is relevant to all developers.

REFERENCES

[1] M. Ali Babar, and I. Gorton, "Software Architecture Review: The State of the Practice", IEEE *Computer*, July 2009.

[2] * L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th edition, 2021.

[3] L. Bass, J. Ivers, M.H. Klein, and P. Merson, Reasoning Frameworks, CMU/SEI-2005-TR-007, 2005.

[4] * F. Brooks, *The Design of Design*, Addison-Wesley, 2010.

[5] S. Brown, Software Architecture for Developers, 2018, http://leanpub.com/software-architecture-for-developers

[6] * D. Budgen, Software Design: Creating Solutions for Ill-Structured Problems, 3rd Edition, CRC Press, 2021.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern Oriented Software Architecture*, John Wiley & Sons, 1996.

[8] M.E. Conway, "How Do Committees Invent?" Datamation, 14(4), 28-31, 1968.

[9] E.W. Dijkstra, "On the role of scientific thought", 1974, available at http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html.

[10] P. Eeles, and P. Cripps, *The Process of Software Architecting*, Addison Wesley, 2010.

[11] G. Fairbanks, Just Enough Software Architecture: A Risk-Driven Approach, Marshall & Brainerd, 2010.

[12] C. Hofmeister, P.B. Kruchten, R.L. Nord, H. Obbink, A. Ran, and P. America, "A general model of software architecture design derived from five industrial approaches", The Journal of Systems and Software, 80, 106–126, 2007.

[13] C. Hofmeister, R.L. Nord, and D. Soni, *Applied Software Architecture*, Addison- Wesley, 2000.

[14] ISO/IEC/IEEE 24765:2017, Systems and software engineering — Vocabulary.

[15] ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description.

[16] P.B. Kruchten, The “4+1” View Model of Architecture, IEEE Software 12(6), 1995.

[17] Z. Li, P. Liang, P. Avgeriou, Architecture viewpoints for documenting architectural technical debt. Software Quality Assurance, Elsevier, 2016

[18] * M.W. Maier, and E. Rechtin, *The Art of Systems Architecting*, 3rd edition, CRC Press, 2021.

[19] H. Obbink, et al., *Report on Software Architecture Review and Assessment* (SARA), version 1.0, available at https://philippe.kruchten.com/architecture/SARAv1.pdf, 2002.

[20] D.L. Parnas, “On the criteria to be used in decomposing systems into modules”, Communications of the ACM 15(12), 1053-1058, 1972.

[21] D.L. Parnas, and D.M. Weiss, “Active Design Reviews: Principles and Practices”, Proceedings of 8th International Conference on Software Engineering, 215-222, 1985.

[22] R. Prieto-Diaz, and J.M. Neighbors, “Module Interconnection Languages”, Journal of Systems and Software, 6(4), 307–334, 1986.

[23] * N. Rozanski, and E. Woods, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd edition, Addison-Wesley, 2011.

[24] M. Shaw, and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[25] * I. Sommerville, *Software Engineering*, 10th edition, 2016.

[26] R. Weinreich, and G. Buchgeher, Towards supporting the software architecture life cycle, *The Journal of Systems and Software*, 85, 546–561, 2012.

chapter 3

v2.7

Software Design

ACRONYMS

API	application programming interface
AOD	aspect-oriented design
CBD	component-based design
CRC	class responsibility collaborator (or collaboration)
DFD	data flow diagram
DSL	domain-specific language
ERD	entity relationship diagram
FOSS	free and open source software
IDL	interface description language
MBD	model-based design
MDD	model-driven design
OO	object-oriented
PDL	program design language
SDD	software design description
UML	Unified Modeling Language

INTRODUCTION

This chapter considers software design from several perspectives—focusing on basic concepts, context and processes, software design qualities and strategies, and recording and evaluating designs.

Design is used in distinct but closely related ways to refer to (1) the *discipline* ("use of scientific principles, technical information, and imagination in the definition of a software system to perform [prespecified] functions with maximum economy and efficiency") [9]; (2) the *processes* for performing within that discipline; (3) the *result* of applying that discipline; and (4) the *stage* in the life cycle of a software system during which those processes yield those results.

Software design, viewed as a life cycle activity, is the application of software engineering discipline in which software requirements are analyzed to define the software's external characteristics and internal structure as the basis for the software's construction.

A *software design description* (SDD) documents the result of software design. It is a "representation of software created to facilitate analysis, planning, implementation, and decision-making. The software design description is used as a medium for communicating software design information and can be thought of as a blueprint or model of the system" [9].

The SDD, which may take many forms, encompasses the refinement of that software into components, the organization of those components, and the definition of interfaces among them and between the software and the outside world—to a level of detail that enables their construction.

Software design takes place in three stages:

- architectural design of the software system
- high-level or external-facing design of the system and its components
- detailed or internal-facing design

Architectural design is a part of architecting, discussed in the Software Architecture KA.

BREAKDOWN OF TOPICS FOR SOFTWARE DESIGN

The breakdown of topics for the Software Design KA is shown in Fig. 1.

Fig. 1. Breakdown of topics for the Software Design KA

1 SOFTWARE DESIGN FUNDAMENTALS

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

1.1 Design Thinking

Design is all around us, in the things and organizations that have been created to meet a need or solve a problem.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

In a 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. Many other notions and concepts help us understand design in its general sense: goals, constraints, alternatives, representations and solutions. (See also *Problem-Solving Techniques* in the Computing Foundations KA.)

Design thinking comprises two essentials: (1) understanding the need or problem and (2) devising a solution. Ross, Goodenough and Irvine offer an elaboration of design thinking appropriate to software:

> This process consists of five basic steps: (1) crystallize a purpose or objective; (2) formulate a concept for how the purpose can be achieved; (3) devise a mechanism that implements the conceptual structure; (4) introduce a notation for expressing the capabilities of the mechanism and invoking its use; (5) describe the usage of the notation in a specific problem context to invoke the mechanism so the purpose is achieved. [15]

This is particularly appropriate because much of software design consists of creating the necessary *vocabulary* to express a problem, express its solution and implement that solution. The steps emphasize the linguistic nature of software design problem solving. This is a recurring pattern we see throughout high-level design, detailed design and architecting (see *Architecting in the Large* in Software Architecture KA).

1.2 Context of Software Design

Software design is an important part of the software development process. To understand the role of software design is to see how it fits into the software development life cycle (see Software Process KA). To understand that context, it is important to understand the major characteristics and roles of software requirements, software construction, software testing, and software maintenance. The context varies with many factors, including degree of formality and stage of the life cycle.

Software design is the transformation of customer and other requirements, needs, and concerns into implementable design specifications. Its contexts include the following:

- Design interface with software requirements: The requirements establish a set of problems that the software design must solve.
- Design interface with software architecture: In cases where an architecture has been established, that architecture constrains the design by capturing fundamental aspects of the system: its major components and their interconnections, application programming interfaces (APIs), styles and patterns to be used, and architectural principles to be observed and enforced.
- Design interface with software construction: The software design must provide a guide to implementors on building the system.
- Design interface with software testing: Software design provides a foundation for an overall testing strategy and test cases that ensure that the design is properly implemented and operates as intended.

1.3 Key Issues in Software Design

Many key issues must be dealt with when designing software. Some are quality *concerns* that all software must address (performance, security, reliability, usability, etc.). Another important issue is how to refine, organize, interconnect and package software components. These issues are so fundamental that all design approaches address them in one way or another. (See topic *Stakeholders and Concerns* in Software Architecture KA, section 1.4 *Software Design Principles*, and topic 5 *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" [2]. Such issues, which often crosscut the system's functionality, are 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" [10].

1.4 Software Design Principles

A *principle* is "a fundamental truth or proposition that serves as the foundation for a system of belief or behavior or for a chain of reasoning." [Oxford English Dictionary]

Design principles provide direction or guidance for making decisions during design. Some principles originated during the early days of software engineering, others even predate the discipline, deriving from best practices in engineering unrelated to software. (See Engineering Foundations KA.) Software design principles are key notions that provide the basis for many different software design concepts, approaches and methods. The principles listed below apply to any of the three stages of design. Many of these principles are interrelated. Whether alone or used in combination with other principles, they are reflected elsewhere in software design to produce many concepts and constructs found in design capture, strategies and methods. This is itself an application of the design thinking process above. Software design principles include the following:

• *Abstraction* is "a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information" [9]."The abstraction principle . . . helps to identify essential properties common to superficially different entities" [15]. (See also topic *Abstraction* in the Computing Foundations KA.)

• *Separation of concerns (SoC).* A *design concern* is an "area of interest with respect to a software design" [9] that is relevant to one or more of its stakeholders. By identifying and separating concerns, the designer can focus on each concern for the system in isolation about which Dijkstra says "even if not perfectly possible, [SoC] is yet the only available technique for effective ordering of one's thoughts " [5] (See also topic *Stakeholders and Concerns* in Software Architecture KA.)

• *Modularization* (or *refinement* or *decomposition*) structures large software as comprising smaller components or units. Each component is named and has well-defined interfaces for its interactions with other components. Smaller components

are easier to understand and, therefore, to maintain. There are numerous modularization strategies. (See topic 5 Software Design Strategies and Methods.)

Traditionally, the goal is to place distinct functionalities and responsibilities in different components. Parnas advocated that each module in a system should have a single responsibility [12]. One way to think of modularization is as a special case of more general strategies, such as separation of concerns or divide and conquer. (see topic Problem-Solving Techniques in Computing Foundations).

• *Encapsulation* (or *information hiding*) builds upon the principles of abstraction and modularization so that nonessential information is less accessible, allowing users of the module to focus on the essential elements of the interface.

• *Separation of interface and implementation* is an application of encapsulation that involves defining a component by specifying its public interfaces, which are known to and accessible to clients; isolating the use of a component from the details of how that component is built. (See *Encapsulation* (or *information hiding*) above.)

• *Coupling* is defined as "a measure of the interdependence among modules in a computer program" [9]. Most design methods advocate that modules should be loosely or weakly coupled.

• *Cohesion* (or *localization*) is defined as "a measure of the strength of association of the elements within a module" [9]. Cohesion highlights organizing a module's constituents based on their relatedness. Most design methods advocate that modules should maximize their cohesion/locality.

• *Uniformity* is a principle of consistency across software components—common solutions should be produced to address common or recurring problems. These include naming schemes, notations and syntax, interfaces that define access to services and mechanisms, and ordering of elements and parameters. This can be achieved through conventions such as rules, formats and styles.

• *Completeness* (or *sufficiency*) means ensuring that a software component captures the important characteristics of an abstraction and leaves nothing out. Completeness takes various forms, perhaps the most important of which is design completeness against requirements: a design should be sufficient for designers to demonstrate how requirements will be met and how subsequent work will satisfy those requirements. Design should be complete with respect to the modes and states of the software.

• *Confirmability* means that information needed to verify that the software is correct, complete and fit for use is available. This is relevant for any software but is of particular importance for high-assurance software, such as software where security, reliability or safety-critical concerns are present. An SDD should be sufficient as a basis for verifying a design.

• *Other design principles*. Recently, with the increased appearance of autonomous systems, the use of machine learning and artificial intelligence, and, generally, systems with widening social impacts, approaches to *Ethically Aligned Design* have been developed to address concerns including universal human values, political self-determination, and data agency and technical dependability [7]. The general principles of Ethically Aligned Design are human rights, well-being, data agency, effectiveness, transparency, accountability, awareness of misuse, and competence.

2 Software Design Processes

Software design is generally considered a multistage process or activity. Software design can be divided into the following stages or phases. When necessary, we distinguish the phase from the general activity:

- Architectural design stage
- High-level design stage
- Detailed design stage

The architectural design stage addresses the fundamentals of the system as a whole and situated in its environment (see Software Architecture KA).

The high-level design stage is *outward-facing*—developing the top-level structure and organization of the software, identifying its various components and how that software system and its components interact with the environment and its elements.

The detailed design stage is *inward-facing*—specifying each component in sufficient detail to facilitate its construction and to meet its outside obligations, including how software components are further refined into modules and units.

Each stage reflects the basic pattern outlined in section 1.1 *Design Thinking*.

Not all stages are found in every software process. However, when present, each stage creates an obligation upon the next stage regarding the software underdevelopment.

Although software developers generally follow similar guidelines for what happens in each stage, there are no strict boundaries between stages regarding what must be done and when. For example, for many software systems, the choice of an algorithm to sort data will be deferred to programmers, within the constraints and guidance provided by the system's requirements, its architecture description or design specifications. However, for another software system, the existence of a suitable algorithm could be architecturally significant and must be determined early in the life cycle. Without that algorithm, there is no possibility of constructing the software to meet its requirements.

Some rules of thumb for each stage include the following:

- The architectural design stage defines a computational model, the major computational elements, and the important protocols and relationships among them. This stage develops strategies to address crosscutting concerns, such as performance, reliability, security and safety, and articulation of crosscutting decisions, including system-wide styles (e.g., a Model-View-Controller style versus, a Pipes-and-Filters style, together with the rationale for such decisions).

- The high-level design stage includes identification of the primary computational elements and significant relationships among them, with a focus on each major component's existence, role and interfaces. That definition should be sufficiently detailed to allow designers or programmers of client components to correctly and efficiently access each server's capabilities—without having to read its code.

- The detailed design stage defines each module's internal structure, focusing on detailing and justifying choices of algorithms, data access and data representation. The detailed design specifications should be sufficient to allow programmers to code each module during construction (see Software Construction KA). The code is a representation of the solution that is sufficiently detailed and complete that a compiler (ore interpreter) can execute it.

2.1 High-Level Design

High-level design specifies the interaction of a system's major components with one another and with the environment, including users, devices and other systems. High-level design addresses the following:

- External events and messages to which the system must respond
- Events and messages which the system must produce
- Specification of the data formats and protocols for events and messages
- Specification of the ordering and timing relationships between input events and messages, and output events and messages
- Data persistence (how data is stored and managed)

High-level design is undertaken within the envelope established by the system's software architecture (if any). Each of the above may be guided or constrained by architecture directives. For example, event signaling and messaging will use the protocols and modes of interaction established by the architecture. Data formats and protocols will use data and communication standards specified by the architecture. Absent an explicit architecture design stage, some of these directives will be established by the software requirements or decided during high-level design.

2.2 Detailed Design

The detailed design stage proceeds within the constraints established by the high-level design. It specifies major system components' internal characteristics, internal modules and their interconnections to other modules, services and processes they provide, computing properties, algorithms, and data access rules and data structures. This includes the following:

- Refinement of major system components into modules or program units, including opportunities for using off-the-shelf components and application frameworks
- Allocation of design responsibilities to modules and program units
- Interactions among modules
- Scope and visibility among components, modules and program units
- Component modes, component states and transitions among them
- Data and control interdependencies
- Data organization, packaging and implementation
- User interfaces
- Requisite algorithms and data structures

3 SOFTWARE DESIGN QUALITIES

Software requirements and architecture directives are intended to guide software toward certain characteristics or *design qualities*. Design qualities are an important subclass of concerns (see topic *Stakeholders and Concerns* in Software Architecture KA). One role of design principles (see section 1.4 *Software Design Principles*) is to help software achieve these qualities. Among the characteristics of interest to designers are the following:

3.1 Concurrency

Design for concurrency concerns how software is refined into concurrent units such as processes, tasks, and threads and the consequences of those decisions with respect to efficiency, atomicity, synchronization and scheduling.

3.2 Control and Event Handling

Event handling is concerned with how to organize control flow as well as how to handle reactive and temporal events through various mechanisms including synchronization, implicit invocation and callbacks.

3.3 Data Persistence

Data persistence concerns the storage and management of data throughout the system.

3.4 Distribution of Components

Distribution concerns how software components are distributed across hardware (including computers, networks and other devices) and how those components communicate while meeting performance, reliability, scalability, availability, monitorability, business continuity and other expectations.

3.5 Errors and Exception Handling, Fault Tolerance

This concern pertains to how to prevent, avoid, mitigate, tolerate and process errors and exceptional conditions.

3.6 Integration and Interoperability

This issue arises at the enterprise or system-of-systems level or for any complex software when heterogeneous systems or applications need to interwork through exchanges of data or accessing one another's services. Within a software system, the issue arises when components are designed using different frameworks, libraries or protocols.

3.7 *Assurance, Security and Safety*

High-assurance spans a number of software qualities, including security and safety concerns, pertaining to whether the software behaves as intended in critical situations. Design for security concerns how to prevent unauthorized disclosure, creation, change, deletion, or denial of access to information and other resources in the face of attacks upon the system or violations of system policies to limit damage; provide continuity of service; and assist repair and recovery. Design for safety pertains to managing the software's behavior in circumstances which might lead to harm to or loss of human life or damage to property or the environment.

3.8 *Variability*

Variability concerns permissible variations in software, as arises for example in software product lines and system families, to accommodate and manage multiple different deployments such as for different organizations or markets.

4 Recording Software Designs

The outcome of design processes is accumulated knowledge and work products recording that knowledge. Design work products capture (1) aspects of the problems to be solved, using the vocabulary of the domain; (2) a solution vocabulary for solving the design problems (see section 1.1 *Design Thinking*); (3) the major decisions that have been taken; and (4) explanations of the rationale for each nontrivial decision. Recording the rationale for important decisions enhances the software product's long-term maintainability when modifications or enhancements are considered (see section 4.6 *Design Rationale*). These work products, often termed *design descriptions* or *design specifications*, can take the form of texts, diagrams, models and prototypes that comprise the blueprints of the software to be implemented.

A fundamental aspect of software design is communication about the design to customers, other designers, implementers and other stakeholders. This is true whether the software is developed using agile, traditional or formal methods. The communication will vary depending upon the target audience, the level of detail being communicated, and relevance to the concerns of the stakeholders. While implementers are an important audience for design specifications, testing and quality assurance personnel, certification authorities, and requirements analysts will also use the design specifications in their work. Therefore, design specifications should have a clearly defined audience, subject and intended usage.

Designers can analyze and evaluate these work products to determine whether the design can meet the requirements and constraints on the software. Software design also examines and evaluates alternative solutions and trade-offs. In addition to using them as inputs and as the starting point for construction and testing, stakeholders can use the design work products to plan subsequent activities, such as system verification and validation.

As design concepts evolve, so do their representations (see section 1.1 *Design Thinking*); part of the design process involves creating appropriate vocabularies for problems and solutions. An informal sketch may be most appropriate for the early stages. It is useful to distinguish in-process ("working") specifications from final design products. The former are produced *by* the design team *for* the design team; the latter may be produced for known stakeholders or even for an unknown future audience.

Many notations exist to represent software design artifacts. Software design is often carried out using multiple types of notation. Two broad areas of concern are software structures and software behaviors. Some are used to describe a design's structural organization, others to represent the software's intended behavior. Below, they are categorized as notations for structural and behavioral concerns (see section 4.2 *Structural Design Descriptions* and section 4.3 *Behavioral Design Descriptions*, respectively). Certain notations are used mostly during architectural design and others mainly during detailed design; some are useful throughout all stages of software design. Some notations are closely tied to the context of specific design methods (see Software Design Strategies and Methods KA).

The Unified Modeling Language (UML) is a widely used family of notations addressing both structural and behavioral concerns and is used in all design stages, from architectural through detailed design [1].

4.1 *Model-Based Design*

Over the history of software engineering, including architecture and design, there has been an evolution from document-based artifacts to model-based artifacts. *Model-based design* (MBD) is an approach to recording designs where models play an important role.

This trend reflects the limitations of document-based artifacts and the increased capabilities of automated tools. Document-based artifacts use natural language and informal diagrams to convey designers' intentions, which might introduce ambiguity and incompleteness. Even when documents use well-defined formats, relevant information might be spread across documents, making understandability and analysis difficult. With MBD, appropriate tooling can gather and organize relevant information for use by designers and other stakeholders in an accessible form.

Modern tools have accelerated the trend from document to model-based artifacts. Tooling enables animation or simulation of various software aspects, analyses of what-if scenarios and trade-offs, and rapid prototyping. Tooling also facilitates continuous testing and integration approaches, enhanced and interactive traceability, and knowledge capture and management, which are inefficient or even infeasible with document-based approaches.

Model-driven development (MDD) is a development paradigm that uses models as the development process' *primary* artifacts (see *Software Engineering Models and Methods* KA).

4.2 *Structural Design Descriptions*

The following types of notation, most of which are graphical, are used to 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) and the allocation of responsibilities to components and modules:

- Class and object diagrams are used to represent a set of classes and objects and their interrelationships.
- Component diagrams are 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" [1]) and their interconnections. Component models evolved from earlier module interconnection languages into the package systems of programming languages like Ada and Java and the sophisticated module systems of current functional languages such as Haskell and Coq.
- Class responsibility collaborator cards (CRCs) are used to denote the names of components (classes), their responsibilities and the components they interact with to meet those responsibilities.
- Deployment diagrams are used to represent a set of physical nodes and their interconnections to model the physical aspects of software as deployed on hardware.
- Entity relationship diagrams (ERDs) are used to represent conceptual, logical and physical models of data as stored in information repositories.
- Interface description languages (IDLs) are programming-like languages used to define the interfaces (names and types of exported operations) of software components.
- Structure charts are used to describe the calling structure of programs (that is, they show which modules call, and are called by, which other modules).

4.3 Behavioral Design Descriptions

The following notations and languages, some graphical and some textual, are used to describe the dynamic behavior of software systems and their components. Many of these notations are useful mostly, but not exclusively, during detailed design. Moreover, behavioral descriptions can include rationale for design decisions (see section 4.6 *Design Rationale*).

- Activity diagrams are used to show flow of a computation from activity to activity. They also can represent concurrent activities.
- Communication diagrams are used to show the interactions among a group of objects; the emphasis is on the objects, their links and the messages they exchange on those links.
- Data flow diagrams (DFDs) are used to show data flow among computing elements. A DFD 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]. DFDs have other uses, such as security analysis, as they identify possible paths for attack and disclosure of confidential information.
- Decision tables and diagrams are used to represent complex combinations of conditions and actions.
- Flowcharts are used to represent the flow of control and the sequence of associated actions.
- Sequence diagrams are used to show the interactions among a group of objects, depicting the time ordering of messages passed among objects.
- State (transition) diagrams and statecharts are used to show transitions from state to state and how a component's behavior changes based on its current state and response to input events.
- Formal specification languages are predominantly textual languages founded upon basic notions from mathematics (for example, type, set, sequence, logical proposition) to rigorously and abstractly define software component interfaces and behavior, often in terms of pre- and post-conditions, invariants, type checking, and computational models (see section *Formal Methods* in Software Engineering Models and Methods KA).
- Pseudocode and program design languages (PDLs) are structured, programming language-like notations used to describe a procedure's processing behavior, generally at the detailed design stage. The use of these languages is less common today but is still found in the documentation of algorithms.

4.4 Design Patterns and Styles

Succinctly described, a pattern is "a common solution to a common problem in a given context" [6]. Architectural styles can be viewed as patterns "in the large," describing common solutions to architecture-level problems that pervade the software. Design patterns include the following:

- Creational patterns (e.g., builder, factory, prototype, singleton)
- Structural patterns (e.g., adapter, bridge, composite, decorator, façade, flyweight, proxy)
- Behavioral patterns (e.g., command, interpreter, iterator, mediator, memento, observer, peer-to-peer, publish-subscribe, state, strategy, template, visitor)

Design patterns and styles reflect idioms that have proven useful in solving particular design problems in the past. They arise at all stages of design, including architecture (see also section 2.2 *Architecture Styles and Patterns*, Software Architecture KA).

4.5 Specialized and Domain-specific languages

Not every design representation falls easily into the structure/behavior dichotomy. For example, user interface design mixes the structural layout of what a user might see with the behavioral logic of sequencing screens based upon user actions. Specialized concerns such as safety and reliability often have their own forms of representation that have evolved among specialists in those communities [16].

A recent trend has been the maturing of *domain-specific languages* (DSLs) and widely available tools to develop them. In this approach, part of the design process is codifying concepts and constructs of a specific application domain to create a

computer language for that domain so that representing the design using these constructs leads to an animated or executable implementation. DSLs blur the lines among modeling languages, design languages and programming languages in this approach.

There are DSLs and supporting tools for domains such as simulation; real-time, reactive and distributed systems; game development; user interfaces; test development; and language processing tools. The growth of DSLs has been facilitated by increasingly powerful grammar-driven tools that, given a language definition, can generate a graphical user interface, syntax checkers, code generators, compilers and linkers for the specialized language.

4.6 Design Rationale

A useful design outcome is insight into and explicit documentation of the major decisions taken, along with an explanation of the rationale for each decision. Design rationale captures *why* a design decision was made. This includes prior assumptions made, alternatives considered, and trade-offs and criteria analyzed to select one approach and reject others.

Although the reasons for decisions are likely to be obvious to the current design team, they can be less obvious to those who modify or maintain the system after deployment. Recording the rationale enhances the software product's long-term maintainability. Continuing to capture the rationale for changes during maintenance also contributes to the software's viability.

It can also be useful to capture rejected decisions and the reasons for rejection. Capturing these rationales can enable a team to revisit a previously rejected decision when assumptions, requirements or constraints change. The importance of rationale is visible, for example, in free and open-source software (FOSS) projects, which often involve large, distributed teams of developers with frequent turnover.

Design rationale may be captured as part of a software design description or as a companion artifact. Often rationale is captured in text, but other forms of representation can also be used, such as graphs that portray a design as an interconnected network of decisions.

5 Software Design Strategies and Methods

Various strategies and methods exist to structure and guide the design process; many of these evolved from programming styles or paradigms. In addition to embodying one or more general strategies, most design methods focus on making one or more design concepts (whether objects, methods or events) prominent as organizing themes for the software. These themes then guide the designers as to what to focus on first, how to proceed, and how to structure modules.

5.1 General Strategies

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

5.2 Function-Oriented (or Structured) Design

This is one of the classical software design methods. It focuses on refinement (or decomposition) to identify major software functions, elaborating them in a top- down manner. Structured design often follows structured analysis, producing DFDs and associated process descriptions. Various tools enable the automated translation of DFDs into high-level designs.

5.3 Data-Centered Design

Data-centered design starts from the data structures a program manipulates rather than from the functions it performs. The software designer specifies the input and output data structures and then develops program units that transform inputs into outputs. Various heuristics have been proposed to deal with special cases, such as cases where there is a mismatch between the input and output structures.

5.4 Object-Oriented Design

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

5.5 User-Centered Design

User-centered design is more than a design method; it is a multidisciplinary approach emphasizing a deep understanding of users and their needs as the basis for designing user experiences within the context of their organization and the tasks to be accomplished. It involves gathering user requirements, creating a user flow of tasks and decisions, creating prototypes or mockups representative of user interfaces, and evaluating the design solution against original requirements.

5.6 Component-Based Design (CBD)

A software component is an independent unit, having well-defined interfaces and dependencies that can be composed and deployed independently. CBD addresses issues related to providing, developing and integrating such components to improve reuse. CBD often emphasizes common APIs for all components and specialized APIs for specific services or responsibilities.

5.7 Event-Driven Design

Event-driven design focuses on the events and messages flowing among and within systems. Message-based systems frequently incorporate identifiable senders and receivers within the design. Event-driven systems may not identify senders and receivers explicitly—instead each module produces events while listening for any events they care about or need to respond to [11]. "Anonymous" asynchronous message and event processing are good strategies for scalable systems.

5.8 Aspect-Oriented Design (AOD)

AOD is a method by which software is constructed using aspects to implement the crosscutting concerns and extensions

identified in software requirements [9]. AOD evolved from object-oriented design and programming practices. Although it has yet to become a widespread design or programming paradigm, the aspect-oriented perspective is frequently used in application frameworks and software libraries where parameters of the framework or library can be configured with aspect declarations.

5.9 Constraint-Based Design

Constraints' role in the design process is to limit the size of a design space to exclude infeasible or unacceptable alternatives. Constraints accelerate design because they force a few early decisions. The constraints can reflect limits imposed on the hardware, software, data, operational procedures, interfaces or anything that affects the software. The constrained design space can then be explored with search or backtracking methods. Constraint-based design approaches are used in user interface design, gaming and other applications. In general, constraint satisfaction problems can be NP-hard; however, various kinds of constraint-based programming can be used to approximate or solve constraint problems.

5.10 Other Methods

Other approaches to design exist (see Software Engineering Models and Methods KA). For example, iterative and adaptive methods implement software increments and reduce the emphasis on rigorous software requirements and design.

Service-oriented methods builds distributed software using web services executed on distributed computers. Software systems are often constructed using services from different providers interconnect with standard protocols (e.g., HTTP, HTTPS, SOAP) designed to support service communication and service information exchange.

6 Software Design Quality Analysis and Evaluation

This section includes several quality analysis and evaluation topics specifically related to software design. Needs to be integrated with topic Review and Evaluation in Software Engineering Management KA.

6.1 Design Reviews and Audits

Does this belong in this KA or elsewhere?

6.2 Quality Attributes

Integrate with earlier discussion of Qualities and relate to Design Concerns

Various attributes contribute to the quality of a software design, including various "ilities" (maintainability, portability, testability, usability) and "nesses" (correctness, robustness). Qualities are a major subset concerns (see topic *Stakeholders and Concerns* in Software Architecture KA). Some qualities can be observed at runtime (e.g., performance, security, availability, functionality, usability); others cannot (e.g., modifiability, portability, reusability, testability); some (e.g., conceptual integrity, correctness, completeness) are intrinsic to the software.

6.3 Quality Analysis and Evaluation Techniques

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

- Software design reviews include informal and rigorous techniques to determine software qualities based on SDDs and other design artifacts for example, architecture reviews, design reviews and inspections; scenario-based techniques; requirements tracing.
- 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 predict 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 Software Engineering Models and Methods KA.)
- Simulation and prototyping: dynamic techniques to evaluate a design (for example, performance simulation or feasibility prototypes).

6.4 Measures and Metrics

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 are based upon the approach used for producing the design (see topic 5 Software Design Strategies and Methods). 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 (or hierarchical diagram) on which various measures can be calculated.
- 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 calculated.

6.5 Verification, Validation and Certification

Should this be in this KA or elsewhere?

MATRIX: TOPICS VS. REFERENCE MATERIAL

In table below, cX means chapter X

Software Design Fundamentals	Brooks [3*]	Budgen [4*]	Gamma et al. [6*]	Sommerville [16*]	See also	
Design Thinking	c1, c2, c3	c1, c2			[3, 15]	
Context of Software Design		c13, c14		c19, c20		
Key Issues in Software Design					[2, 10]	
Software Design Principles					[5, 9, 12, 15]	
Software Design Processes		c3		c2, c7		
High-level Design	c5	c6			[8]	
Detailed Design					[8]	
Software Design Qualities		c4				
Concurrency				c17		
Control and Handling of Events				c21		
Data Persistence				c6, c16		
Distribution of Components				c17		
Errors and Exception Handling, Fault Tolerance				c11		
Integration and Interoperability		c11, c14, c16				
Assurance, Security and Safety				c10–c14		
Variability						tbd
Recording Software Designs		c7, c8			[1]	
Model-based Design		c7.3		c5.5		
Structural Design Descriptions		c7, c10	c4	c5.3		

Behavioral Design Descriptions		c9, c10	c5	c5.4		
Design Patterns and Styles	c12	c15	c1, c2	c7.2	[6]	
Specialized and Domain-specific languages						tbd
Design Rationale	c16	c12		c6.1		
Software Design Strategies				c3		
General Strategies		c13				
Function-Oriented (“Structured”) Design						tbd
Data-Centered Design						tbd
Object-Oriented Design		c10				
User-Centered Design	c9					
Component-Based Design (CBD)		c11, c16		c16		
Event-Driven Design					[11]	
Aspect-Oriented Design					[10]	
Constraint-Based Design	c11					
Other Methods				c18–c21		
Software Design Quality Analysis and Evaluation		c17		c24		
Design Reviews and Audits		c5.3				
Quality Attributes						
Quality Analysis and Evaluation Techniques						
Measures and Metrics		c5, c17		c24.5		
Verification, Validation and Certification						tbd

FURTHER READINGS

TBD

REFERENCES

[1] * Booch, G, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 2nd edition, Addison-Wesley, 2005

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

[3] * Brooks, F. *The Design of Design*, Addison-Wesley, 2010

[4] * Budgen, D. Software Design: Creating Solutions for Ill-structured problems, 3rd Edition CRC Press, 2021

[5] Dijkstra, E. W., On the role of scientific thought. 1974. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

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

[7] IEEE Std 7000™-2021 IEEE Standard Model Process for Addressing Ethical Concerns during System Design

[8] ISO/IEC/IEEE 12207 Systems and software engineering — Software life cycle processes

[9] ISO/IEC/IEEE 24765 Systems and software engineering — Vocabulary

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

[11] Luckham, D., The power of events: an introduction to Complex Event Processing, Addison-Wesley, 2002

[12] Parnas, D.L. On the criteria to be used in decomposing systems into modules, *Communications of the ACM* **15**(12), 1053–1058, 1972

[13] Parnas, D.L. and Clements, P.C. A rational design process: How and why to fake it, *IEEE Transactions on Software Engineering* **12**(2), 251– 257, 1986

[14] Parnas, D.L. and Weiss, D.M. Active design reviews: Principles and practices, *Journal of Systems & Software* **7**, 259–265, 1987

[15] Ross, D.T., Goodenough, J.B., Irvine, A. Software Engineering: Process, Principles, and Goals, *IEEE Computer*, May 1975

[16] * Sommerville, I. Software Engineering, 10th edition, Pearson, 2016

CHAPTER 4
SOFTWARE CONSTRUCTION

ABBREVIATIONS

API	Application programming interface
ASIC	Application-specific integrated circuit
BaaS	Backend as a service
CI	Continuous integration
COTS	Commercial off-the-shelf
CSS	Cascading style sheets
DSL	Domain-specific language
DSP	Digital signal processor
ESB	Enterprise service bus
FPGA	Field programmable gate array
GUI	Graphical user interface
HTML 5	HyperText markup language version 5
IDE	Integrated development environment
IEEE	Institute of electrical and electronics engineers
ISO	International organization for standardization
J2EE	Java 2 platform, enterprise edition
KA	Knowledge area
MDA	Model-driven architecture
NPM	Node Package Manager
OMG	Object management group
PIM	Platform independent model
POSIX	Portable operating system interface
PSM	Platform-specific model
SDK	Software development kit
TDD	Test-driven development
UML	Unified Modeling Language
WYSIWYG	What you see is what you get

INTRODUCTION

Software construction refers to the detailed creation and maintenance of software through 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 the Software Design and Software Testing KAs because the software construction process involves significant 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 among design, construction and testing (if any) vary depending on the software life cycle processes used in a project.

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

Also, 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.

The Software Construction KA is also related to configuration management, quality,

project management and computing, and thus to the relevant KAs.

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

Second, while quality is important in all the KAs, code is a software project's ultimate deliverable, and code is produced during construction. Thus, the Software Quality KA is closely linked to the Software Construction KA.

Third, while project management involve various software development tasks, software construction typically produces the most deliverables of a software project. Thus, the Software Construction KA is closely linked to the Software Engineering Management KA.

Fourth, since software construction requires knowledge of algorithms and coding practices, this KA is closely related to the Computing Foundations KA, which concerns the computer science foundations supporting software product design and construction.

BREAKDOWN OF TOPICS FOR SOFTWARE ARCHITECTURE

The breakdown of topics for the Software Architecture KA is shown in エラー! 参照元が見つかりません。.

Fig. 1. Breakdown of Topics for the Software Construction KA

1. Software Construction Fundamentals

Software construction fundamentals include the following:

- Minimizing complexity
- Anticipating and embracing change
- Constructing for verification
- Reusing assets
- Applying standards in construction

The first four concepts apply to design as well as to construction. The following sections define these concepts and describe how they apply to construction.

1.1. Minimizing Complexity [1]

Most people have limited ability to hold complex structures and information in their working memories, especially over long periods. This greatly influences how people convey intent to computers and drives one of the key goals in software construction — to minimize complexity. The need to reduce complexity applies to essentially every aspect

of software construction and is particularly critical to testing software constructions.

Several types of complexity can affect software construction. Tools can be used to manage different aspects of the complexity of software components and their construction. For example, cyclomatic complexity is a static analysis measure of how difficult code is to test and understand. The tool, developed by Thomas J. McCabe, Sr., in 1976, calculates the number of linearly independent paths through a program's source code. Ideally, there should be at least that number of test cases. Other examples are tools like Make, which can build an application, or integrated development environments (IDEs) for entering, editing and compiling code. These tools help manage the complexity of the construction process.

In software construction, reduced complexity is achieved by creating simple and readable code rather than clever code. This is accomplished by using 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). Construction-focused quality techniques also support this (see section 3.6, Construction Quality).

1.2. Anticipating and Embracing Change [1, 2, 3]

Most software changes over time, and anticipating 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, enhancing a software product without disrupting the underlying structure. Anticipating change is supported by many specific techniques (see section 3.3, Coding).

Moreover, today's business environments require many organizations to deliver and deploy software more frequently, faster and more reliably. Anticipating specific, necessary changes can be difficult, so software engineers should be careful to build flexibility and adaptability into the software to incorporate changes with less difficulty. These software teams should embrace change by adopting agile development, practicing DevOps, and by adopting continuous delivery and deployment practices. Such practices align the software development process and management with an evolutionary environment.

1.3. Constructing for Verification [1]

Constructing for verification builds 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 difficult-to-understand language structures.

1.4. Reusing Assets [2]

Reuse means using existing assets to solve different problems. In software construction, typical assets that are reused include frameworks, libraries, modules, components, source code and commercial off-the-shelf (COTS) assets. Reuse has two closely related facets: *construction for reuse* and *construction with reuse*. The former means creating reusable software assets, whereas the latter means reusing software assets to construct a new solution. Reuse often transcends project boundaries, which means

reused assets can be constructed in other projects or organizations.

1.5. Applying Standards in Construction [1]

Applying external or internal development standards during construction helps achieve a project's efficiency, quality and cost objectives. 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 the following:

- Communication methods (e.g., standards for document formats and content)
- Programming languages (e.g., standards for languages like Java and C++)
- Coding standards (e.g., standards for naming conventions, layout and indentation)
- Platforms (e.g., interface standards for operating system calls)
- Tools (e.g., diagrammatic standards for notations like UML (Unified Modeling Language))

Use of external standards: Construction depends on 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 (e.g., object management group (OMG)) and international organizations (e.g., the institute of electrical and electronics engineers (IEEE) or the international organization for standardization (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 coordinating group activities, minimizing complexity, anticipating change and constructing for verification.

2. Managing Construction

2.1. Construction in Life Cycle Models [1, 2, 3]

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

Some models are more linear from the construction viewpoint, such as the waterfall and staged-delivery life cycle models. These models treat construction as an activity that occurs only after the completion of significant prerequisite work, including detailed requirements work, extensive design work and detailed planning. The more linear approaches emphasize the activities that precede construction (requirements and design) and create more distinct separations between activities. In these models, construction's main emphasis might be coding.

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

The practices of continuous delivery and deployment further mix coding, testing,

delivery and deployment activities. In these practices, software updates made during construction activities are continuously delivered and deployed into the production environment. The whole process is fully automated by a deployment pipeline that consists of various testing and deployment activities.

Consequently, what is considered construction depends 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. This choice 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 objective may also be addressed at the process, requirements and design levels, but the choice of construction method will influence them.

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

2.4. Managing Dependencies [2]

Software products often heavily rely on dependencies, including internal and external (commercial or open-source) dependencies, which allow developers to reuse common functionalities instead of reinventing the wheel and substantially improve developers' productivity. In addition, package managers (e.g., Maven in Java and NPM in JavaScript) are widely used to automate the process of installing, upgrading, configuring and removing dependencies.

The direct and indirect dependencies of software products constitute a dependency supply chain network. Any dependency in the supply chain network can introduce potential risk to software products and should be managed by developers or tools. Unnecessary dependencies should be avoided to improve build efficiency. License conflicts between dependencies and software products should be avoided to reduce legal risk. Propagation of dependencies' defects or vulnerabilities into software products should be avoided to improve the quality of software products. Regulations and monitoring mechanisms should be developed to prevent developers from introducing untrusted external dependencies.

3. Practical Considerations

Construction is an activity in which the software engineer often has to deal with sometimes chaotic, changing and even conflicting real-world constraints. Because of real-world constraints, practical considerations drive construction more than some other KAs, and software engineering is perhaps most craft-like in the construction activities compared with other activities.

3.1. Construction Design [1, 2]

Some projects allocate considerable design activity to construction, whereas others allocate design to a phase explicitly focused on design. Regardless of the exact allocation, some detailed design work occurs at the construction level, and that design work is dictated by constraints imposed by the real-world problem the software addresses.

Just as construction workers building a physical structure must make small modifications for unanticipated gaps in the builder's plans, software construction workers must make small or large modifications to flesh out software design details 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 at a smaller scale to 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 solution to a problem. Consequently, construction languages and their implementations (e.g., compilers) can affect software quality attributes such as performance, reliability and portability. As a result, they can seriously contribute to security vulnerabilities.

The simplest 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 some program generators' menu-style selection lists constitute another example of a configuration language.

Toolkit languages are used to build applications from 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 might be implied by a toolkit's set of interfaces.

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

Programming languages are the most flexible 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 greatly affect the likelihood of vulnerabilities being introduced during coding (e.g., uncritical use of C and C++ is questionable from a security viewpoint).

Three general notations are used for programming languages:

- 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 in patterns may have a sentence-like syntax. Properly used, each string should have a strong semantic connotation providing an immediate intuitive understanding of what happens when the software construction is executed.

Formal notations rely less on intuitive, everyday meanings of words and text strings and more on definitions backed by precise, unambiguous and formal (or mathematical) definitions. Formal construction notations and methods are at the semantic base of most 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 more on direct visual interpretation and placement of visual entities that represent the underlying software. Visual construction is 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 to build and "adjust" a visual interface to a program, the detailed behavior of which has an underlying definition.

Nowadays, domain-specific languages (DSLs) are widely used to build domain-specific applications. Unlike a general-purpose programming language, such as C/C++ or Java, a DSL is designed for the application construction of a particular domain. Therefore, a DSL usually can be defined based on a higher level of abstraction of the target domain and can be optimized for a specific class of problems. Furthermore, A DSL usually can be expressed by visual notations defined by domain-specific concepts and rules.

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 (e.g., input of bad data)
- Prevention of code-level security breaches (e.g., buffer overflows or array index bounds)
- Resource use through 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, 2]

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

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

Construction testing typically involves a subset of the various types of testing, 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 testing.

Two standards have been published on 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 and 2.1.2 in the Software Testing KA for more specialized reference material.

3.5.Reuse in Construction [2]

Reuse in construction includes both construction for reuse and construction with reuse.

Construction for reuse creates software with the potential to be reused in the future for the present project or for other projects with a broad-based, multisystem perspective. Construction for reuse is usually based on variability analysis and design. To avoid the problem of code clones, developers should 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 and design patterns
- 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

Construction with reuse means creating new software by reusing existing software assets. The most popular reuse method is to reuse code from the libraries provided by the language, platform, tools or an organizational repository. Aside from these, many applications developed today use open-source libraries. In addition, reused and off-the-shelf software often have the same (or better) quality requirements as newly developed software (e.g., security level requirements).

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

- Selecting reusable units, databases, test procedures or test data
- Evaluating code or test reusability
- Integrating reusable software assets into the current software
- Reporting reuse information on new code, test procedures or test data

The forms of reusable software assets are not limited to software artifacts that must be locally integrated. Nowadays, cloud services that provide various services through online interfaces such as RESTful application programming interfaces (APIs) are widely

used in applications. In the new cloud service model BaaS (backend as a service), applications delegate their backend implementations to cloud service providers — for example, utilities such as authentication, messaging and storage are usually provided by cloud providers.

Reuse is best practiced systematically, according to a well-defined, repeatable process. Systematic reuse can enable significant software productivity, quality and cost improvements. Systematic reuse is supported by methodologies such as software product line engineering and various software frameworks and platforms. Widely used frameworks such as Spring provide reusable infrastructures for enterprise applications so software teams can focus on application-specific business logic. Commercial platforms provide various reusable frameworks, libraries, components and tools to support application development to build their ecosystems.

3.6. Construction Quality [1, 2]

In addition to faults occurring during requirements and design activities, faults introduced during construction can cause serious quality problems (e.g., security vulnerabilities). These include not only faults in security functionality but also faults elsewhere that allow bypassing of the security functionality or create other security weaknesses or violations.

Numerous techniques exist to ensure the quality of code as it is constructed. The primary techniques used to ensure construction quality are the following:

- Unit testing and integration testing (see section 3.4, Construction Testing)
- Test-first development (see section 6.1.2 in the Software Testing KA)
- Use of assertions and defensive programming
- Debugging
- Inspections
- Technical reviews, including security-oriented reviews (see section 2.3 in the Software Quality KA)
- Static analysis (see section 2.2.1 of the Software Quality KA)

The specific technique or techniques selected depend on the software constructed and on the skill set of the software engineers performing the construction activities. Programmers should know good practices and common vulnerabilities (e.g., from widely recognized lists about common vulnerabilities). Automated static code analysis 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. These activities focus on 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.

3.7. Integration [1, 2, 3]

During construction, a key activity is integrating 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 are 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 (e.g., 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, including, for example, stubs, drivers and mock objects, is usually needed to enable incremental integration. In addition, by building and integrating one unit at a time (e.g., 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 and more fully tested units, among others.

Nowadays, continuous integration (CI) has been widely adopted in practice. A software team using CI integrates its work frequently, leading to multiple integrations per day. CI is usually automated by a pipeline that builds and tests each integration to detect errors and provide fast feedback.

3.8. Cross-Platform Development and Migration [4]

Some applications, such as mobile applications, heavily rely on specific platforms (e.g., Apple, Android), which usually include operating systems, development frameworks and APIs. To support multiple platforms, the developers need to develop and build an application separately for each target platform using the corresponding program language and software development kit (SDK). However, multi-platform development in this way requires more time and cost and might cause different user experiences between different implementations.

Cross-platform development allows the developers to develop an application using a universal language and export it to various platforms. This usually can be done in two ways for mobile applications. One way is to generate native applications using specific tools that can compile the universal language into platform-specific formats. The other is to develop hybrid applications that combine web applications developed using languages like hypertext markup language version 5 (HTML5) and cascading style sheets (CSS) and native containers or wrappers for various operations systems.

For applications that are not developed in this way, developers may consider migrating the applications from one platform to another. The migration usually involves translation of different programming languages and platform-specific APIs and can be partially automated by tools.

4. Construction Technologies

4.1. API Design and Use [5]

An *API* is a 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 make the API easy to learn and memorize, lead to readable code, be difficult 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, an API should be straightforward and stable, to facilitate client application development and maintenance.

API use involves selecting, learning, testing, integrating and possibly extending APIs provided by a library or framework (see section 3.5, Reuse in Construction).

4.2. Object-Oriented Runtime Issues [1]

Object-oriented languages support runtime mechanisms, including polymorphism and reflection. These runtime mechanisms increase the flexibility and adaptability of object-oriented programs.

Polymorphism is a language's ability to support general operations without knowing until runtime what kind of concrete objects the software will include. Because the program does not know the types of the objects in advance, the exact behavior is determined at runtime (called *dynamic binding*).

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

4.3. Parameterization, Templates and Generics [6]

Parameterized types, also known as *generics* (Ada, Eiffel) and *templates* (C++), enable a type or class definition without specifying all the other types used. The unspecified types are supplied as parameters at the point of use. Parameterized types provide a third way (besides 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 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 other problems. Assertions are typically compiled into the code at development time and are later compiled out of the code so 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. A contract precisely specifies the semantics of a routine and thus helps clarify 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]

How *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 as follows: 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 or 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 or containing their effects if recovery is not possible. The most common fault tolerance strategies include backing up and retrying, using auxiliary code and voting algorithms, and replacing an erroneous value with a phony value that will have a benign effect.

4.6. Executable Models [7]

Executable models abstract away the details of specific programming languages and decisions about the software's organization. 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. Furthermore, 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 is a way of constructing executable software.

Executable models are one foundation supporting the model-driven architecture (MDA) initiative of the OMG. An executable model is a way to 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 that uses finite-state machines to describe program behaviors. A state machine's transition graphs are used in all stages of software development (specification, implementation, debugging and documentation). The main idea is to construct computer programs in the same way technological processes are automated. 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 display information rather than convey information with logic statements (such as *if* and *case*). When 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 its variables' late binding time. For example, *runtime configuration* 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,* modifies 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 used, what kinds of strings are used, how to maintain the strings without changing the code and how to translate the strings into different languages with minimal impact on the processing code and the user interface.

4.9. Grammar-Based Input Processing [1, 8]

Grammar-based input processing involves syntax analysis, or *parsing*, of the input token stream. It creates 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. Next, the parser checks the symbol table for programmer-defined variables that populate the tree. After building the parse tree, the program uses it as an input to the computational processes.

4.10. Concurrency Primitives [9]

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 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 in 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 [5, 8]

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 using the middleware. Modern message-oriented middleware usually provides an enterprise service bus (ESB) that supports service-oriented interaction and communication among multiple software applications.

4.12. Construction Methods for Distributed and Cloud-Based Software [2, 9]

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

Distributed programming typically falls into several basic architectural categories: client-server, three-tier architecture, n-tier architecture, distributed objects, loose coupling or tight coupling (see section 5.6 in the Computing Foundations KA and section 2.2 in the Software Architecture KA).

Nowadays, more applications are migrated to the cloud. *Cloud-based software* often adopts microservice architecture and container-based deployment. In addition to traditional distributed software issues, cloud-based software developers also need to consider cloud infrastructure issues such as use of an API gateway, service registration and discovery.

4.13. Constructing Heterogeneous Systems [8]

Heterogeneous systems consist of various 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 combining several specification languages to design different system parts (hardware/software codesign). The key issues include multilanguage validation, co-simulation and interfacing.

During the hardware/software codesign, software 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 execution speed and size. *Performance analysis* investigates a program's behavior using information gathered as the program executes to identify possible hot spots in the program to be improved.

Code *tuning*, which improves performance at the code level, modifies correct code to make it run more efficiently. Code tuning usually involves only small 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 hot spots in a program.

4.15. Platform Standards [4, 8, 9]

Platform standards enable programmers to develop portable applications that can be executed in compatible environments without changes. Platform standards usually involve standard services and APIs that compatible platform implementations must use. Typical examples of platform standards are Java 2 platform, enterprise edition (J2EE); the portable operating system interface (POSIX) standard for operating systems, which represents a set of standards implemented primarily for Unix-based operating systems; and HTML5, which defines the standards for developing web applications that can run on different environments (e.g., Apple iOS, Android).

4.16. Test-First Programming [1, 2]

Test-first programming (also known as test-driven development (TDD)) is a popular development style in which test cases are written before 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.

4.17. Feedback Loop for Construction [3]

Early and continuous feedback for the construction activity is one of the most important advantages of agile development and DevOps. Agile development provides early feedback for construction through frequent iterations in the development process. DevOps provides even faster feedback from the operation, allowing the developers to learn how well their code performs in production environments. This fast feedback is achieved through techniques and practices in the DevOps pipeline, such as automated building and testing, canary release, and A/B testing.

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 programmers' choice of development environment can affect software construction efficiency and quality.

Besides basic code editing functions, modern IDEs often offer other features, like compilation and error detection 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.

Nowadays, cloud-based development environments are available in public or private cloud services. These environments can provide all the features of modern IDEs and even more (e.g., containerized building and deployment), powered by the cloud.

5.2. Visual Programming and Low-Code/Zero-Code Platforms [1]

Visual programming allows users to create programs by manipulating visual program elements graphically. As a visual programming tool, a GUI (graphical user interface) builder 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 that enables the developer to design forms and windows and manage the layout of the widgets with drag, drop and parameter setting features. Some GUI builders can

automatically generate the source code corresponding to the visual GUI design. Because GUI applications usually follow the event-driven style (in which events and event handling determine the program flow), 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 stand-alone GUI builders.

Visual programming and other rapid application development tools have evolved into *low-code/zero-code platforms*. These platforms allow developers to build complete applications visually through a drag-and-drop interface and with minimal hand-coding. They are usually based on the principles of model-driven design, visual programming and code generation. The difference between low-code development and zero-code development lies in hand-coding; the former requires a little hand-coding, whereas the latter requires practically none.

5.3. Unit Testing Tools [1, 2]

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

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 often each statement is executed or how much time the program spends on each statement or execution path. Profiling the code while it runs 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 computing the set of program statements (i.e., the program slice) that might affect the values of specified variables at some point of interest, which is called a slicing criterion. Program slicing can be used for locating error sources, program understanding and optimization analysis. Program slicing tools compute program slices for various programming languages using static or dynamic analysis methods.

MATRIX: TOPICS VS. REFERENCE MATERIAL

	McConnell, 2004 [1]	Sommerville, 2016 [2]	Kim et al., 2021 [3]	Heitkötter et al., 2013 [4]	Clements et al., 2010 [5]	Gamma et al., 1994 [6]	Mellor and Balcer, 2002 [7]	Null and Lobur, 2006 [8]	Silberschatz et al., 2008 [9]
1. Software Construction Fundamentals									
1.1. Minimizing Complexity	c2, c3, c7-c9, c24, c27, c28, c31, c32, c34								

1.2. Anticipating and Embracing Change	c3-c5, c24, c31, c32, c34	c1, c3, c9	c1						
1.3. Constructing for Verification	c8, c20-c23, c31, c34								
1.4. Reuse		c15							
1.5. Standards in Construction	c4								
2. Managing Construction									
2.1. Construction in Life Cycle Models	c2, c3, c27, c29	c3, c7	c1						
2.2. Construction Planning	c3, c4, c21, c27-c29								
2.3. Construction Measurement	c25, c28								
2.4. Managing Dependencies		c25							
3. Practical Considerations									
3.1. Construction Design	c3, c5, c24	c7							
3.2. Construction Languages	c4								
3.3. Coding	c5-c19, c25-c26								
3.4. Construction Testing	c22, c23	c8							
3.5. Reuse in Construction		c15, c16							
3.6. Construction Quality	c8, c20-c25	c8, c24							
3.7. Integration	c29	c8	c11						
3.8. Cross-Platform Development and Migration				c					
4. Construction Technologies									
4.1. API Design and Use					c7				
4.2. Object-Oriented Runtime Issues	c6, c7								
4.3. Parameterization, Templates 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	
4.10. Concurrency Primitives									c6
4.11. Middleware					c1			c8	
4.12. Construction Methods for Distributed and Cloud-Based Software		c17, c18							c2
4.13. Constructing Heterogeneous Systems								c9	
4.14. Performance Analysis and Tuning	c25, c26								
4.15. Platform Standards				c				c10	c1
4.16. Test-First Programming	c22	c8							
4.17. Feedback Loop for Construction			c3, c16						
5. Software Construction Tools									
5.1. Development Environments	c30								

5.2. Visual Programming and Low-Code/Zero-Code Platforms	c30								
5.3. Unit Testing Tools	c22	c8							
5.4. Profiling, Performance Analysis and Slicing Tools	c25, c26								

REFERENCES

[1] McConnell, S., *Code Complete*, 2nd edition, Redmond, WA: Microsoft Press, 2004.

[2] Sommerville, I., *Software Engineering*, 10th edition, Addison-Wesley, 2016.

[3] Kim, G., et al., *The DevOps Handbook: How to Create World-Class Agility, Reliability & Security in Technology Organizations*, 2nd edition, IT Revolution, 2021.

[4] Heitkötter, H., Hanschke, S., Majchrzak, T.A., Evaluating Cross-Platform Development Approaches for Mobile Applications, 2013, in Cordeiro, J., Krempels, K.H. (eds.), *Web Information Systems and Technologies*. WEBIST 2012. Lecture Notes in Business Information Processing, vol. 140, Springer, Berlin, Heidelberg.

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

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

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

[8] Null, L., and Lobur, J., *The Essentials of Computer Organization and Architecture*, 2nd edition, Sudbury, MA: Jones and Bartlett Publishers, 2006.

[9] Silberschatz, A., et al., *Operating System Concepts*, 8th edition, Hoboken, NJ: Wiley, 2008.

CHAPTER 5
SOFTWARE TESTING

ACRONYMS

AI	Artificial Intelligence
API	Application Program Interface
ARINC	Aeronautical Radio Incorporated
ATDD	Acceptance Test-Driven Development
CMM	Capability Maturity Model
CSS	Cascading Style Sheets
DICOM	Digital Imaging and Communications in Medicine
DL	Deep Learning
DU	Definition and Use
EBSE	Evidence-Based Software Engineering
ECS	Ecosystem
ETSI	European Telecommunications Standards Institute
FHIR	Fast Healthcare Interoperability Resources
GDPR	General Data Protection Regulation
GPS	Global Positioning System
GUI	Graphical User Interface
HIL	Hardware-In-the-Loop
HIPAA	Health Insurance Portability and Accountability Act
HL7	Health Level Seven
IoT	Internet of Things
KPI	Key Performance Indicator
MC/DC	Modified Condition Decision Coverage
ML	Machine Learning
MTTR	Mean Time to Recovery
OAT	Orthogonal Array Testing
ODC	Orthogonal Defect Classification
RUP	Rational Unified Process
SoS	System of Systems
SPI	Software Process Improvement
SPICE	Software Process Improvement and Capability Determination
SUT	System Under Test
TDD	Test-Driven Development
TMMi	Test Maturity Model integration
UI	User Interface

INTRODUCTION

Software testing consists of the *dynamic* validation that a *System Under Test* (*SUT)* provides *expected* behaviors on a *finite* set of *test cases* suitably *selected* from the usually infinite execution domain.

In the above statement, italicized words correspond to key issues in the Software Testing Knowledge Area (KA). Those terms are discussed below.

- *System Under Test*: This term can refer to the testing object, which could be a

program, a software product, an application, a service-oriented application (e.g., web services, microservices), middleware (HW/SW), a services composition, a system, a System of Systems (SoS), or an Ecosystem (ECS).

- *Test Case:* A test case is the specification of all the entities that are essential for the execution, such as inputs values, execution and timing conditions, testing procedure, and the expected outcomes (e.g., produced values, state changes, output messages). Input values alone are not always sufficient to specify the test cases since because the SUT might react to the same input with different behaviors, depending, for instance, on the SUT state or environmental conditions. A set of test cases is usually called a *test suite.*
- *Dynamic:* Dynamic validation requires executing the SUT on a test suite. Static techniques complement dynamic testing, and they are covered in the Software Quality KA.[1]
- *Finite:* Even in a simple SUT, executing all the possible test cases (i.e., exhaustive testing) could require months or years. Consequently, in practice, testing targets a subset of all possible test cases, determined by different criteria. Testing always implies a trade-off between limited resources and schedules on the one hand and inherently unlimited test requirements on the other.
- *Selected:* Identifying the most suitable selection criteria under given conditions is a complex problem. Different techniques can be considered and combined to tackle that problem, such as risk analysis, software requirements, cost reduction, quality attributes satisfaction, prioritization, and fault detection. The many proposed test techniques differ in how the test suite is selected, and software engineers must be aware that different selection criteria might yield vastly different degrees of effectiveness.
- *Expected*: For each executed test case, it must be possible, although it might not easy, to decide whether the observed SUT outcomes match the expected ones. Indeed, 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 foreseen behavior from implicit requirements or expectations. (See Section 4.3, Acceptance Criteria-Based Requirements Specification, in the Software Requirements KA.)

As reflected in this discussion, software testing is a pervasive and holistic activity involving all the steps of any process development life cycle (e.g., traditional, or left-shift development). The remainder of this chapter presents the basics of software testing, as well as its challenges, issues, and commonly accepted practices and solutions.

BREAKDOWN OF TOPICS FOR SOFTWARE TESTING

Figure 1 shows the breakdown of topics for the Software Testing KA. The Matrix of Topics vs. Reference Material provides a more detailed breakdown at the end of this KA.

The first topic, Software Testing Fundamentals, covers the basic definitions in software testing, the basic terminology and

[1] It is worth noting that terminology is not uniform among different communities, and some use the term *testing* to refer to static techniques as well.

key issues, and software testing's relationship with other activities.

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

Figure 1. Breakdown of Topics for the Software Testing KA

The Target of the Test and Objectives of Testing together determine how the test suite is identified, both regarding its consistency (How much testing is enough for achieving the stated objective?) and its composition (Which test cases should be selected for achieving the stated objective?). (However, usually "for achieving the stated objective" remains implicit, and only the first part of the two questions above is posed.) Criteria for addressing the first question are *test adequacy criteria*, whereas those used for 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. Therefore, the third topic covers generally accepted and standardized techniques.

Test-Related Measures are dealt with in the fourth topic, while the issues relative to the Test Process are covered in the fifth.

Software Testing in the Development Processes and the Application Domains is described in the sixth topic, and Testing of and Testing Through Emerging Technologies are described in the seventh topic. Finally, Software Testing Tools are presented in topic eight.

1. Software Testing Fundamentals

[1*, c1, c2, 2*, c8, 20*, c7]

This section provides an overview of the main testing issues and the relationship of testing to the other activities. Most of the testing terms used here are also defined. A more comprehensive overview of the testing and testing-related terminology can be found in the cited references.

1.1 Faults vs. Failures

[1*, c1s5, 2*, c1; 20*, c1s3]

Many terms are used in the software engineering literature to describe a malfunction: notably *fault* (see, for comparison, *defect* in Section 3.2, Defect Characterization, in the Software Quality KA), *failure* and *error*. It is essential to distinguish between the *cause* of a malfunction (for which the term *fault* is used here) and an undesired effect observed in the system's delivered service (a *failure*). Indeed, there might well be faults in the software that never manifest as failures. (See Theoretical and Practical Limitations of Testing in Section 1.2.8.) Thus, testing can reveal failures, but the faults causing them are what can and must be removed. However, a failure's cause cannot always be unequivocally identified. No theoretical criteria exist to definitively determine, in general, the fault that caused an observed failure. The fault might have to be modified to remove the failure, but other modifications might also work. To avoid ambiguity, we could refer to *failure-causing inputs* instead of faults — those sets of inputs that cause a failure to appear.

1.2. Key Issues

This subsection provides an overview of the main testing issues.

1.2.1. Test Case Creation

[1*, c12s1, c12s3, 2*, c8]

Test case creation or *generation* creates the test suite useful for testing the SUT for specific purposes (e.g., adequacy, accuracy, or assessment). Because test case generation is among the most important and intensive activities of software testing, it is usually supported by approaches, techniques, and tools to automate the process.

1.2.2. Test Selection and Adequacy Criteria

[1*, c1s14, c6s6, c12s7, 2*, c8]

A test selection criterion is a means of selecting test cases or determining that a test suite is sufficient for a specified purpose. Test case selection aims to reduce the cardinality of the test suites while keeping the same effectiveness in terms of coverage or fault detection rate. Test adequacy criteria can be used to decide when sufficient testing is accomplished.

1.2.3 Prioritization/Minimization

[6, part 2, part 3, c5]

Suitable strategies for test case selection or prioritization can be adopted to improve testing efficacy. Test case *prioritization* aims to define a test execution order according to some criteria (e.g., coverage, fault detection rate, similarity, and risk) so those tests with a higher priority are executed before those with a lower priority. Test case *minimization* usually aims to reduce a test suite by removing redundant test cases according to some criterion or purpose.

1.2.4. Purpose of Testing

[1*, c13s11, c11s4, 2*, c8]

Different well-defined purposes can guide testing activity; it is only by considering a specific purpose that a test suite can be generated (selected), executed and evaluated.

1.2.5. Assessment and Certification

[6, part 1, c5; 2*, c7, c25; 14]

Testing needs to focus on specific (mandatory) prescriptions, such as requirements, laws and standards. Test cases should be generated and executed to provide evidence useful for evaluating and/or certifying adherence to the selected prescriptions. Usually, *assessment* and *certification* of the test results include verifying that the test cases have been derived and generated using baseline requirements, adopting a configuration control process, and using repeatable processes.

1.2.6. Testing for Quality Improvement/Assurance

[1*, c16s2; 6, part1, c5; 15]

Testing has many aspects, including quality improvement and assurance. These characteristics involve planned and systematic supporting processes and activities leveraging confidence that the SUT fulfills established technical or quality requirements. Thus, quality *improvement* and *assurance* involve defining methods, tools, skills, and practice to achieve the specific quality level and objectives. The list of the main quality characteristics that testing can measure or assess is reported in ISO/IEC 25010:2011 [15]. (See also Section 1.3.2, Software Product Quality, in the Software Quality KA.)

1.2.7. The Oracle Problem

[1*, c1s9, c9s7]

An important testing component is the oracle. Indeed, a test is meaningful only if it is possible to decide its observed outcome. An *oracle* can be any human or mechanical agent that decides whether the SUT behaved correctly in each test and according to the expected outcomes. Consequently, the oracle provides a "pass" or "fail" verdict. The oracle cannot always decide; in these cases, the test output is classified as inconclusive. There are many kinds of oracles — for example, unambiguous requirements specifications, behavioral models, and code annotations. Automation of mechanized oracles can be difficult and expensive.

1.2.8. Theoretical and Practical Limitations

[1*, c2s7]

Testing theory warns against ascribing unjustified confidence to a series of successful tests. Unfortunately, most established results of the testing theory are negative results, in that they state what is not achieved as opposed to what is achieved. The most famous quotation on this point 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.

1.2.9. The Problem of Infeasible Paths

[1*, c4s7]

Infeasible paths are control flow paths that cannot be exercised by any input data (i.e., test cases). Managing the infeasible paths can help reduce the time and resources devoted to testing. They are a significant problem in path-based testing, particularly in the automated derivation of test cases to exercise control flow paths. Additionally, infeasible paths can also be connected to the analysis and detection process for security vulnerabilities and can improve accuracy.

1.2.10. 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 test suite will expose a failure *if* the software is faulty. Both meanings are important.

1.2.11 Test Execution and Automation

[6, part 1c4]

An important challenge of testing is to improve attainable automation, either by developing advanced techniques for generating the test inputs or, beyond test generation, by finding innovative support procedures to (fully) automate the different testing activities — for instance, to increase the number of test cases generated or executed.

1.2.12. Scalability

[1*, c8s7]

Scalability is the software's ability to increase and scale up on its nonfunctional requirements, such as load, number of transactions and volume of data. Scalability is also connected to the complexity of the platform and environment in which the program runs, such as distributed, wireless networks and virtualized environments, large-scale clusters, and mobile clouds.

1.2.13 Test Effectiveness

[1* c1s1; 2* c8s1; 14]

Evaluating the SUT, measuring a testing technique's efficacy, and judging whether testing can be stopped are important issues for software testing, and they may require defining and selecting the proper test effectiveness measures.

1.2.14 Controllability, Replication, and Generalization

[1* c12s12; 6 part 2 c7]

Specific aspects of testing include the following:

- *Controllability* refers to the transition of testing activities from the laboratory (i.e., controlled conditions) to reality (i.e., uncontrolled conditions).
- *Replication* refers to the ability for different people to perform the same testing activities. The purpose is to verify whether a given testing theory works, at least in the laboratory.
- The *generalization* of testing is connected to external validity — i.e., the extent to which the test approach can be applied to broader settings or target populations. The generalizability of the software testing can be important for managing the testing activities (in terms of cost and effort) and increasing confidence in the test results.

1.2.15. Off-Line vs. Online Testing

[16, c3]

The testing process can be executed in two settings: *off-line* and *online*. Usually, the

SUT is validated in an environment without external interaction in off-line testing, whereas the SUT interacts with the real application environment in online testing. The test cases are either manually or automatically derived in both cases, and the expected outcomes are used to assess the SUT.

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 viewpoint of software quality analysts and certifiers. For further discussion, see the following:

- Testing vs. Static Software Quality Management Techniques: See Section 2.2.1, Static Analysis Techniques, in the Software Quality KA.
- Testing vs. Quality Improvement/Assurance: See Section 1.3.2, Software Product Quality, in the Software Quality KA.
- Testing vs. Correctness Proofs and Formal Verification: See the Software Engineering Models and Methods KA.
- Testing vs. Debugging: See Construction Testing in the Software Construction KA and Debugging Tools and Techniques in the Computing Foundations KA.
- Testing vs. Program Construction: See Construction Testing in the Software Construction KA.
- Testing vs. Security: See the new KA: Software Security.
- Testing vs. Effort Estimation: See the Software Engineering Management KA.
- Testing vs. Legal Issues: See the Software Engineering Professional Practice KA.

2. Test Levels

[1*, c1s13; 2*, c8s1]

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

2.1. The Target of the Test

[1*, c1s13, 2*, c8s1]

The target of the test can vary depending on the SUT, the conditions of the environment and the budget/time devoted to the testing activity. Four test stages can be distinguished: unit, integration, system, and acceptance. These four test stages do not imply any development process, nor is any one of them assumed to be more important than the other three.

2.1.1. Unit Testing

[1*, c3, 2*, c8]

Unit testing verifies the functioning in isolation of SUT elements that are separately testable. Depending on the context, these could be the individual subprograms or components, a subsystem, or a composition of SUT components. Typically, but not always, the person who wrote the code conducts the unit testing.

2.1.2. Integration Testing

[1*, c7, 2*, c8]

Integration testing verifies the interactions among SUT elements (for instance, components, modules, or subsystems). Integration strategies involve the incremental (and systematic) integration of the SUT elements considering either identified functional threads or architecture specifications. Typical integration testing strategies are top-down, bottom-up, mixed (or sandwiched) and the big bang. They focus on different perspectives of the level at which

SUT elements are integrated. Integration testing is a continuous activity that can be performed at each development stage. It may target different aspects, such as interoperability (e.g., compatibility or configuration) of the SUT elements or with the external environment. External interfaces to other applications, utilities, hardware devices or operating environments can also be considered.

2.1.3. System Testing

[1*, c8, 2*, c8]

System testing concerns testing the behavior of the SUT (according to the definition of Section 1). Effective unit and integration testing should have identified many SUT defects. In addition, system testing is usually considered appropriate for assessing the non-functional system requirements, such as security, privacy, speed, accuracy, and reliability. (See Functional and Non-Functional Requirements in the Software Requirements KA and Software Quality Requirements in the Software Quality KA.)

2.1.4. Acceptance Testing

[1*, c1s7, 2*, c8s4]

Acceptance testing targets the deployment of a SUT. Its main goal is to verify that the SUT satisfies the requirements and the end-users' expectations. Generally, it is run by or with the end-users to perform those functions and tasks for which the software was built. For example, this testing activity could target usability testing or operational acceptance. Defining acceptance tests before implementing the corresponding functionality is a key activity of the Acceptance Test-Driven Development (ATDD). (See the Software Requirements KA, Section 4.3.)

2.2. Objectives of Testing

[1*, c1s7]

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

Testing can be aimed at verifying different properties. For example, 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 non-functional properties may be tested as well, including performance, reliability, and usability. (See Models and Quality Characteristics in the Software Quality KA.)

Other important testing objectives include but are not limited to reliability measurements, identification of security and privacy vulnerabilities, and usability evaluation; different approaches would be necessary depending on the objective. 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 cited in the literature.

2.2.1. Conformance Testing

[1*, c10s4]

Conformance testing aims to verify that the SUT conforms to standards, rules, specifications, requirements, design, processes, or practices.

2.2.2 Compliance Testing

[1*, c12s3]

Compliance testing aims to demonstrate the SUT's adherence to a law or regulation. Usually, compliance testing is forced by an external regulatory body.

2.2.3. Installation Testing

[1*, c12s2]

Often, after system and acceptance testing is completed and the SUT has been installed in the target environment, the SUT is verified. *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.

2.2.4. Alpha and Beta Testing

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

Before the SUT 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 testing and beta testing are often uncontrolled and are not always referred to in a test plan.

2.2.5. Regression Testing

[1*, c8s11, c13s3, 6 part1, c5]

According to the definition reported in [7], *regression testing* is the "selective retesting of a SUT to verify that modifications have not caused unintended effects and that the SUT still complies with its specified requirements." In practice, the approach is designed to show that the SUT still passes previously passed tests in a test suite (in fact, it is sometimes referred to as *non-regression testing*). In some cases, a trade-off 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. This can be quite time-consuming because of the many tests that might be executed. Regression testing can be conducted at each test level described in Section 2.1. It may involve functional and non-functional testing, such as reliability, accessibility, usability, maintainability, conversion, migration, and compatibility testing.

Regression testing may involve selection (see Section 1.2.2) and minimization (see Section 1.2.3) of test cases, as well as the adoption of prioritization approaches (see Section 2.2.6) to existing test suites.

Regression testing is a fundamental activity of Agile, DevOps, Test-Driven Development (TDD) and Continuous Development. It is usually performed after integration testing and before deployment to production or operation.

2.2.6. Prioritization Testing

[1*, c12s7]

Test case prioritization aims to schedule test cases to increase the rate of fault detection, the likelihood of revealing faults, the coverage of code under test and the SUT's reliability. Typically, prioritization testing relies on heuristics, and its performance might vary according to the SUT, the environment and the available test cases. Among the different prioritization proposals, similarity-based prioritization is one of the most commonly adopted. In this approach to prioritization, test cases are prioritized starting from those most dissimilar according to a predefined distance function.

2.2.7. Non-functional Testing

[2*, c8]

Non-Functional testing targets the validation of non-functional aspects (such as performance, usability or reliability), and it is performed at all test levels. At the state of the practice, there are hundreds of non-functional testing techniques that include but are not limited to the following:

- Performance Testing [6, part 1]: Performance testing verifies that the software meets the specified performance requirements and assesses performance characteristics (e.g., capacity and response time).

- Load Testing [6, part 1]: Load testing focuses on validating the SUT's behavior under load pressure conditions to discover problems (e.g., deadlocks, racing, buffer overflows and memory leaks) or reliability, stability, or robustness violations. It aims to assess the rate at which different service requests are submitted to the SUT.
- Stress Testing [1*, c8s8]: Stress testing aims to push the SUT beyond its capabilities by generating a load greater than what the system is expected to handle.
- Volume Testing [6, part 1]: Volume testing targets the assessment of the SUT's internal storage limitations and its ability to exchange data and information.
- Failover Testing [1*, c17s2, 2*, c8]: Failover testing validates the SUT's ability to manage heavy loads or unexpected failure to continue typical operations (e.g., by allocating extra resources). Failover testing is also connected with recoverability validation.
- Reliability Testing [1*, c15, 2*, c11]: Reliability testing evaluates the SUT's reliability by identifying and correcting faults. Reliability testing observes the SUT in operation or exercises the SUT by using test cases according to statistical models (operational profiles) of the different users' behaviors. Usually, reliability is assessed through reliability growth models. The continuous development processes (such as DevOps) are currently facilitating the adoption of reliability testing in the various iterations for improving final SUT quality.
- Compatibility Testing [6, part 1, 16, c3]: Compatibility testing is used to verify whether the software can collaborate with different hardware and software facilities or with different versions or releases.
- Scalability Testing [1*, c8s7, 2* c17]: Scalability testing evaluates the capability to use and learn the system and the user documentation. It also focuses on the system's effectiveness in supporting user tasks and the ability to recover from user errors. This testing is particularly important in distributed or high-performance systems.
- Elasticity Testing [2* c17]: Elasticity testing assesses the ability of the SUT (such as cloud and distributed systems) to rapidly expand or shrink compute, memory, and storage resources without compromising the capacity to meet peak utilization. Some elasticity testing objectives are to control behaviors, to identify the resources to be (un)allocated, to coordinate events in parallel, and to evaluate scalability.
- Infrastructure Testing [14, annex H]: Infrastructure testing tests and validates infrastructure components to reduce the chances of downtime and improve the performance of the IT infrastructure.
- 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."
- Recovery Testing [1*, c14s2]: Recovery testing is aimed at verifying software restart capabilities after a system crash or other disasters.

2.2.8. Security Testing

[2*, c13; 6, part4, annex A]

Security testing focuses on validating that the SUT is protected from external attacks. More precisely, it verifies the confidentiality, integrity and availability of the systems and their data. Usually, security testing includes validation against misuse and abuse of the software or system (negative testing). (See Security Testing in the Software Security KA.)

2.2.9. Privacy Testing

[2*, c13, c14]

Privacy testing is devoted to assessing the security and privacy of users' personal data to prevent local attacks. It specifically assesses privacy and information-sharing policies, as well as the validation of decentralized management of users' social profiles and data storage solutions. (See Legal Issue in the Software Engineering Professional Practice KA.)

2.2.10. Interface and Application Program Interface (API) Testing

[2*, c8s1; 20*, c7s12; 6, part 5c4, c7]

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

2.2.11. Configuration Testing

[1*, c8s5]

Where the SUT is built to serve different users, *configuration testing* verifies the software under specified configurations.

2.2.12. Usability and Human-Computer Interaction Testing

[2* c8s4; 10*, c6; 6, part 4, annex A]

The main task of *usability and human-computer interaction testing* is to evaluate how easy it is for end-users to learn to use the software. It might involve testing the software functions that support user tasks, the documentation that aids users, and the system's ability to recover from user errors. (See User-Centered Design in the Software Design KA.)

3. Test Techniques

[1*, c1s15; 6, part 4]

Over the years, different testing techniques have been developed to increase the SUT's overall quality [6, part 4]. These techniques attempt to propose systematic procedures and approaches for generating or selecting the most suitable test suites for detecting as many failures as possible.

Testing techniques can be classified by considering different key aspects such as specification, structure and experience [6, part 4]. In this case, additional information sources can be the faults to be discovered, the predicted use, the models, the nature of the application or the derived knowledge. In this classification, overlapping is possible, and one category might deal with the combined use of two or more techniques.

Alternative classifications that rely on the degree of information about the SUT are available in the literature. Indeed, in the specification-based techniques, also known as *black-box* techniques, the generation of test cases is based only on the SUT's input/output behavior, whereas in the structure-based, also called *white-box* (or *glass-box* or *clear-box*), techniques, the test cases are generated using the information about how the SUT has been designed or coded.

As some testing techniques are used more than others, the remainder of the section presents the standard testing techniques and those commonly adopted at the state of the practice.

3.1. Specification-Based Techniques

[1*, c6s2, 6 part 4]

The underlying idea of *specification-based techniques* (sometimes also called domain testing techniques) is to select a few test cases from the input domain that can detect specific categories of faults (also called domain errors). These techniques check whether the SUT can manage inputs within a certain range and return the required output.

3.1.1. Equivalence Partitioning

[1*, c9s4]

Equivalence partitioning involves partitioning the input domain into a collection of subsets (or equivalence 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 SUT 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 test suite (sometimes containing only one test case) is usually taken from each equivalence class.

3.1.2. Boundary-Value Analysis

[1*, c9s5; 6, part 4]

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.

3.1.3. Syntax Testing

[1*, c10s11, 2*, c5; 6, part 4]

The Syntax Testing techniques, also known as formal specification-based techniques, rely on the SUT specifications in a formal language. (See Formal Methods in the Software Engineering Models and Methods KA.) This representation permits automatic derivation of functional test cases and, at the same time, provides an oracle for checking test results.

3.1.4. Combinatorial Test Techniques

[1*, c9s3; 6, part 4]

The Combinatorial Test Techniques systematically derive the test cases that cover specific parameters of values or conditions. According to [6 part 4], the commonly used combinatorial test techniques are All-Combinations Testing, Pair-Wise Testing, Each Choice Testing and Base Choice Testing. All-combinations testing focuses on all the possible input combinations, whereas its subset, also called *t-wise* testing, considers every possible combination of *t* input. In this case, more than one pair is derived (i.e., by including higher-level combinations). Pair-wise testing is a specific combinatorial testing technique where test cases are derived by combining values of every pair of an input set. These techniques are also known as Orthogonal Array Testing (OAT).

3.1.5. Decision Table

[1*, c9s6; 1*, c13s6 6, part 4]

Decision tables (or trees) represent logical relationships between conditions (roughly, inputs) and actions (roughly, outputs). Usually, they are widely adopted for knowledge representation (e.g., Machine Learning (ML)). Test cases are systematically derived by considering every possible combination of conditions and their corresponding resultant actions. A related technique is *cause-effect graphing*. Currently,

shift-left development processes are taking advantage of this kind of testing technique because these techniques are useful for documenting the test results and factors that can affect them.

3.1.6. Cause-Effect Graphing

[1*, c1s6; 6, part 3; 6, part 4]

Cause-effect graphing techniques rely on logical networks that map a set of causes to a set of effects by systematically exploring the possible combinations of input conditions. They identify the effects and link the effects to their causes through model graphs. Cause-effect graphing techniques are used in testing because they allow specification analysis, the identification of the relevant input conditions or causes, the consequent transformations, and the output conditions.

3.1.7. State Transition Testing

[1*, c10, 6, part 4]

Techniques based on Finite-State Machines (*State Transition Testing* techniques in [6, part 4]) focus on representing the SUT with a finite-state machine. In this case, the test suite is derived to cover the states and transitions according to a specific coverage level.

3.1.8. Scenario-Based Testing

[2*, c8s3.2, c19s3.1; 6 part 4; 13]

A model in this context is an abstract (formal) representation of the SUT or its software requirements. (See Modeling in the Software Engineering Models and Methods KA.) *Scenario-based testing* is used to validate requirements, check their consistency and generate test cases focused on the SUT's behavioral aspects. (See Types of Models in the Software Engineering Models and Methods KA.) The key components of scenario-based testing are 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. Because of the complexity of the techniques, scenario-based testing approaches are often used with test automation harnesses.

Among scenario-based testing, workflow models can also be used to graphically represent the sequence of activities performed by humans and/or software applications. In this case, each sequence of actions constitutes one workflow (also called a scenario). Usually, it is important to ensure that both typical and alternate workflows are also tested. For example, business process testing is part of this scenario-based technique. In this case, special focus is on the roles in a workflow specification.

3.1.9. Random Testing

[1*, c9s7; 6, part 4]

In this approach, test cases are generated purely at random. This testing falls under the heading of input domain testing because the input domain must be known to be able to pick random points within it. *Random testing* provides a relatively simple approach for test automation. Enhanced forms of random testing (such as adaptive random testing) have been proposed in which other input selection criteria direct the random input sampling.

Currently, under the name of fuzzy testing, the random selection of invalid and unexpected inputs and data is extensively used in cybersecurity to find hackable software bugs, coding errors and security loopholes. (See also Sections 2.2.8 and 8.2.)

3.1.10. Evidence-Based

[16, c6s2]

Evidence-based software engineering (EBSE), which follows a rigorous research approach, is the *best* solution for a practical

problem. EBSE includes the following phases:

- Identifying the evidence and forming a question
- Tracking down the best evidence to answer the question
- Critically analyzing the evidence in the light of the problem that the evidence should help solve.

EBSE principles can also be applied to the testing process. For that purpose, the widely used approaches that allow identifying and aggregating evidence are systematic mapping studies and systematic reviews.

3.1.11. Forcing Exception

[7]

Test cases are specifically conceived for checking whether the SUT can manage a predefined set of exceptions/errors, such as data exception, operation exception, overflow exception, protection exception or underflow exception. Testing techniques usually focus on negative test scenarios (i.e., test cases that are able to force the generation of error messages).

3.2. Structure-Based Test Techniques

[6, part 4]

Structure-Based Test Techniques (sometimes called *code-based test techniques*) focus on the code and its structure. Structure-Based Test Techniques can be performed at different levels (such as code development, code inspection or unit testing) and can include static testing (such as code inspection, code walkthrough and code review), dynamic testing (like statement coverage, branch coverage and path coverage), or code complexity measurement (e.g., using techniques like cyclomatic complexity [18]).

3.2.1. Control Flow Testing

[1*, c4; 6, part 4]

Control flow testing covers all the statements, branches, decisions, branch conditions, modified condition decision coverage (MC/DC), blocks of statements or specific combinations of statements in a SUT. The strongest of the control flow-based criteria is path testing, which aims to execute all entry-to-exit control flow paths in a SUT's control flow graph. Because 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.

3.2.2. Data Flow Testing

[1*, c5; 6, part 4]

In *data flow testing*, the control flow graph is annotated with information about how the variables are defined, used, and killed (undefined). Commonly adopted data flow testing techniques are All-Definitions Testing, All-C-Uses Testing, All-P-Uses Testing, All-Uses Testing and All-DU-Paths Testing. The strongest data flow testing criterion is the All-DU-Paths Testing, where all definition and use (DU) paths need to be covered. This is because it requires executing, for each variable, every control flow path segment from a definition of that variable to the use of that definition. However, weaker strategies such as all-definitions and all-uses are used to reduce the number of paths required.

3.2.3. Reference Models for Structure-Based Test Techniques

[1*, c4]

Although not a technique, a SUT's control structure can be graphically represented using a flow graph to visualize *structure-based test 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.

3.3. Experience-Based Techniques

[6, part 1, part 4]

The generation of the most suitable test suite may depend on different factors, such as human knowledge of the SUT and its context and his/her experience and intuition. In the following section, the commonly adopted experience-based techniques are briefly introduced.

3.3.1. Error Guessing

[1*, c9s8; 6, part 4]

In *error guessing*, software engineers design test cases specifically to anticipate the most plausible faults in each SUT. Good sources of information are the history of faults discovered in earlier projects and the software engineer's expertise.

3.3.2. Exploratory Testing

[6, part 1]

Exploratory testing is defined as simultaneous learning, test design and test execution. The test cases are not defined in advance but are dynamically designed, executed and modified according to the collected evidence and test results, such as observed product behavior, peculiarities of the SUT, the domain and the environment, the failure process, the types of possible faults and failures, and the risk associated with a particular product. Usually, the intuition, knowledge, and expertise of the personnel in charge of performing the exploratory testing can affect the testing effectiveness. Exploratory testing is widely used in left-shift development (such as Agile). (See Section 5.4.2.)

3.3.3. Further Experience-Based Techniques

[6, part 4, 19]

At the state of the practice, *experience-based techniques* may include further approaches such as Ad Hoc-based, knowledge-based and ML-based testing techniques.

Ad Hoc testing is a widely used technique in which test cases are derived by relying on the software engineer's skill, intuition and experience with similar programs. It can be useful for identifying test cases that are not easily generated by more formalized techniques. Typical Ad Hoc methodologies are the following:

- Monkey testing runs randomly generated test cases to cause the program to stop.
- Buddy testing generates test cases by using internal architecture knowledge and testing specific knowledge.
- Pair testing involves two individuals. One generates and runs the test cases; the other observes and analyzes the testing process. Pair testing allows generating test cases with broad and better test coverage.
- Gamification aims to convert testing tasks to components of gameplay. By applying specific techniques (such as engaging practitioners or crowdsourcing complex testing tasks), gamification can substantially improve software testing practice and, consequently, SUT quality.
- Quick testing, in which a very small test suite is selected and executed, guarantees that no failure can be experienced as a result of SUT components that are not fully operational.
- Smoke testing (also known as Build Verification Testing) ensures that the SUT's core functionalities behave properly. It also guarantees that the SUT is operational before the planned

testing begins. In addition, smoke testing prevents failures because of the test environment (e.g., because artifacts or packages are not properly built). Smoke testing is also considered a special case of quick testing.

Knowledge-based testing and ML-based testing exploit (formal or informal) knowledge about the SUT or derive it from observations of SUT executions for defining its behavioral models (such as ontologies or decision tables) (see Section 3.6.1), rules, and non-functional properties. In addition, Knowledge-based testing and ML-based testing specify the testing needs and identify test objectives for which test cases are generated.

3.4. Fault-Based and Mutation Techniques

[1*, c1s14, 1* c3s5; 7]

With different degrees of formalization, *fault-based* testing techniques devise test cases specifically to reveal likely or predefined fault categories. A *fault model* can be introduced that classifies the different faults to better focus the test case generation or selection. In this context, a variety of platforms and development processes (e.g., waterfall, spiral and Agile) consider the Orthogonal Defect Classification (ODC) a valid methodology for collecting semantic information about the different defects and reducing the time and effort of the root cause analysis.

Mutation Testing was originally conceived as a technique to evaluate test suites (see Section 4.2, Evaluation of the Tests Performed) in which a mutant is a slightly modified version of the SUT (also called *gold*), differing from it by a small syntactic change. Every test case exercises both the gold version and all generated mutants. If a test case succeeds in identifying the difference between the gold version and a mutant, the latter is said to be "killed." The underlying assumption of mutation testing, the coupling effect, is that more complex but real faults will be found by looking for simple syntactic faults. For the technique to be effective, many mutants must be automatically generated and executed systematically [12]. Mutation testing is also a testing criterion in itself. Test cases 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 structure-based technique. Mutation testing has been used effectively for generating fuzzy testing. A more recent application of the mutation process is metamorphic testing, a technique that has become increasingly popular in addressing some ML systems' testing challenges. In this case, the modifications (called also morph) are applied to the inputs so a relationship can connect the previous input (and its output) to the new morphed input (and its output).

3.5. Usage-Based Techniques

[1*, c15s5]

Usage-based techniques usually rely on a usage model or profiles. In this case, the testing environment needs to represent the actual operational environment, and the sequence of test case execution should reproduce the SUT usage by the target stakeholder. Statistical sampling is used for simulating the execution of many test cases. Thus, sometimes, the term *random testing* is also associated with these techniques. Usage-based statistical testing is applied more during the acceptance testing stage.

3.5.1. Operational Profile

[1*, c15s5, 2*, c11]

Testing based on *operational profiles* aims at generating test cases that might estimate the reliability of the SUT or part of it. Therefore, the goal is to infer from the observed test

results the future reliability of the software (when it is in use). Because the established reliability strictly depends on the operating profile, the main difficulty (and cost) in using this testing approach comes from the operational profile derivation. Therefore, one possible solution is to assign to the input the probabilities or profiles according to their frequency of occurrence in actual operation.

3.5.2. User Observation Heuristics

[10*, c5, c7; 6, part 4, annex A]

Specialized heuristics, also called *usability inspection methods*, are applied to systematically observe system use under controlled conditions 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. Techniques Based on the Nature of the Application

[2* c16, c17, c18, c20, c21; 20*, c4s8; 14]

The above techniques apply to all kinds of software. Additional test derivation and execution techniques are based on the nature of the software being tested. Examples are the following:

- Object-oriented software
- Component-based software
- Web-based software
- Concurrent programs
- Protocol-based software
- Communication systems
- Real-time systems
- Safety-critical systems
- Service-oriented software
- Open-source software
- Embedded software
- Cloud-based software
- Blockchain-based software
- Big data-based software
- ML-based software
- Mobile apps
- Security and privacy-preserving software

In some cases, standards such as ISO/IEC/IEEE 29119 [6 part 4, part 5] provide examples and support for specifying test cases, automating their execution, and maintaining the test suites.

3.7. Selecting and Combining Techniques

[20*; 16; 6, part 5]

Combining different testing techniques has always been a well-grounded means to assure the required level of SUT quality. Currently, especially in left-shift developments, methodologies for adaptive combinations of testing techniques are part of the state of the practice. The goal is to improve the effectiveness of testing processes by learning from experience and, at the same time, adapting the technique selection to the current testing session.

3.7.1. Combining Functional and Structural

[1*, c9; 6, part 5]

Scenario-based and structure-based test techniques are often contrasted as *functional* vs. *structural* testing. These two approaches to test case selection are nowadays seen as complements, as they use different sources of information and have been shown to highlight different problems. Depending on the different organizational constraints, such as budgetary considerations, they could be combined.

3.7.2. Deterministic vs. Random

[1*, c9s6]

Test cases can be selected in a deterministic way, according to 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.

3.8. Techniques Based on Derived Knowledge

[2*, c19, c20; 20*, c7]

Testing techniques can integrate evidence and knowledge from different research areas and contexts. For this, approaches and methodologies are used to support testing activity and improve its effectiveness. Currently, innovative approaches include using digital twins or simulation methodologies and frameworks, exploiting ML and gamification facilities, and using (simulated) neuronal networks.

4. Test-Related Measures

[2*, c24s5; 20*, c10; 6, part 4]

Testing techniques are like tools that help in achieving specific test objectives. To evaluate whether a test objective is reached, well-defined measures are needed. Measurement is usually considered fundamental to quality analysis. Measurement may also be used to optimize test planning and execution. Test management can use several different process measures to monitor progress. (See Software Engineering Measurement in the Software Engineering Management KA for information on measurement programs. See Software Measurement in the Software Engineering Process KA for information on measures.)

According to the definition in [6, part 4], testing techniques can be classified according to the degree of coverage they can achieve. Coverage may vary from 0% to 100%, excluding possible infeasible tests (i.e., tests that cannot be executed). Thus, for each specification-based, structure-based, and experience-based test technique, the associated coverage measures, and the procedure for evaluating that coverage must be determined. Examples of coverage measures could be the percentage of branches covered in the program flow graph or the percentage of functional requirements exercised among those listed in the specifications document.

It is important to consider that monitoring facilities can dynamically compute the ratio between covered elements, and the total number may also be considered. Additionally, especially in the case of structure-based testing techniques, appropriate instrumentation of the SUT may also be necessary.

However, the proposed set of testing measures can also be classified from different viewpoints — from the point of view of those providing and allowing an evaluation of the SUT based on the observed test outputs and of those that evaluate the thoroughness or effectiveness of the executed test suites.

4.1. Evaluation of the SUT

[2*, c24s5]

Usually, indicators (i.e., measurable information) can be used to determine whether a SUT is performing as expected and achieving its expected outcomes. The indicators, sometimes known as Key Performance Indicators (KPIs), are strongly connected with the adopted evaluation measures, methods, data analysis and reporting.

4.1.1. SUT Measurements That Aid in Planning and Designing Tests

[20*, c10; 16, c6; 6 part 1]

All the testing measures proposed in [6 part 4] can be used for planning and guiding testing activities. Additionally, in the shift-left development process, specific measures, such as Deployment Frequency, Lead Time, Mean Time to Recovery (MTTR), and Change Failure Rate, are also commonly adopted to plan and manage the testing activities and results.

4.1.2. Fault Types, Classification and Statistics

[1* c13s4, c13s5, c13s6]

The testing literature is rich in classifications and taxonomies of faults that can be generic or specific to a context or quality attributes (such as the usability defect classification, the taxonomy of HW/SW security and privacy vulnerabilities and attacks, and the classification of cybersecurity risks). To make testing more effective, it is important to know which types of faults may be found in the SUT and the relative frequency with which these faults have occurred in the past. This information can be useful in making quality predictions and in process improvement (See Characterization in the Software Quality KA).

4.1.3. Fault Density

[1*, c13s4; 20*, c10s1]

Traditionally, a SUT can be evaluated by counting discovered faults as the ratio between the number of faults found and the SUT size. Because of the semantics-based definition of faults, additional measurements can be considered, such as fault depth (the minimal number of fault removals needed to make a SUT correct) and fault multiplicity (the number of atomic changes needed to repair a single fault).

4.1.4. Life Test, Reliability Evaluation

[1*, c15, 2*, c11; 20*, c1s3]

A statistical estimate of software reliability, which can be obtained by observing reliability achieved, can be used to evaluate a SUT and decide whether testing can be stopped or the SUT is mature enough to be a candidate for the next left-shift development release. Reliability evaluation is taking a pivotal role in the Cloud (and fog) contexts [24].

On the one hand, validation and verification proposals are focusing on maintaining the high level of reliability and availability required by the cloud (fog) services. On the other, testing activities are exploiting the computational power of the cloud (fog) environment to speed up the reliability evaluation and drastically reduce its costs.

4.1.5. Reliability Growth Models

[1*, c15, 2* c11s5]

Reliability growth models predict reliability based on observed 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 product's reliability will increase. 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

[6, part 6, c6]

The behavior of SUT is generally verified by executing test suites, which are pivotal in finding defects. Therefore, from both the researchers' and practitioners' perspectives, a fundamental part of software testing is comparing test suites. Usually, evaluating the test suites means comparing techniques for test case generation that produce the test cases. Different criteria are used for that purpose, such as coverage criteria or mutation analysis criteria.

4.2.1. Fault Injection

[1*, c2s5]

In *fault injection*, some faults are artificially introduced into the SUT before testing. When a test suite is executed, some of these injected faults are revealed, as are, possibly, some faults that were already there. In theory, depending on which and how many artificial faults are discovered, the testing effectiveness can be evaluated, and the remaining number of genuine faults can be estimated. In practice, statisticians question the distribution and representativeness of injected 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 because inserting faults into the SUT incurs the obvious risk of leaving them there.

4.2.2. Mutation Score

[1*, c3s5, 12]

In mutation testing, the test suite effectiveness measure is calculated as the ratio of killed mutants to the number of generated mutants. The higher the test suite effectiveness value, the better, since it indicates a stronger ability to discover the most real injected faults.

4.2.3. Comparison and Relative Effectiveness of Different Techniques

[1*, c1s7; 7; 15]

Relative effectiveness compares different testing techniques against a specific property, such as 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. Several studies have already been conducted to compare different techniques analytically and empirically according to each notion of property (or effectiveness) defined.

5. Test Process

[6 part 1, part 2, part 3; 2* c8]

Testing concepts, strategies, techniques and measures need to be integrated into a defined and controlled test planning process to test output evaluation. The test process supports testing and provides guidelines to those responsible for different testing activities to ensure the test objectives are met cost-effectively.

As described in [6 part 2], the test process is a multi-layered process activity that includes the test specification at the organizational, management and dynamic levels. The organizational test process defines the steps for creating and maintaining test specifications, such as organizational test policies, strategies, processes, procedures, and other assets [6 part 2].

The test management process defines the steps necessary for management: planning, monitoring and control, and completion.

Finally, the dynamic test process specifies the steps for design and implementation, environment setup and maintenance, execution, and test incident reporting.

In the remainder of this section, some practical considerations about the test process specification, management, and execution, as well as a summary of the test sub-processes and activities included in the organizational, management and dynamic levels as in [6 part 2], are provided.

5.1. Practical Considerations

[6, part 1]

Testing processes should allow the automation of different testing phases and should rely on the controllability, replicability, and risk/cost estimation of the performed activities. In the remainder of this section, commonly applied test steps are described, compatible with and applicable to all life cycle models. (See Software Life Cycles in the Software Engineering Process KA.)

5.1.1. Attitudes/Egoless Programming

[1*, c16; 2*, c3]

An important element of successful testing is a collaborative attitude toward testing and Quality Assurance (QA) activities. Managers have a key role in fostering a favorable reception toward failure discovery and correction during software development and maintenance. For instance, in shift-left development processes, such as Agile,

communication and collaboration among testers and developers are considered vital for achieving successful testing results.

5.1.2. Test Guides and Organizational Process

[1*, c12s1, 2* c8, 6 part 2, part 3; 20*, c7s3]

Various aims can guide the testing phases. 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 and backlog lists. Usually, the organization of the test process includes defining test policies (i.e., specifying the purpose, goals, and overall scope of testing) and test strategies (i.e., specifying the guidelines about how testing will be carried out). For instance, in left-shift developments, a test strategy should include at least the following data: the purposes (e.g., defined through user stories), the objectives (e.g., a test suite), the scope (the SUT), and the environment and methods (e.g., how, and where the test suite is run).

5.1.3. Test Management and Dynamic Test Processes

[1*, c12; 6 part 2, part 3, 20*, c7s3]

Test activities conducted at different levels (see Section 2, Test Levels) must be organized — with people, tools, policies, and measures — into a well-defined process integral to the life cycle. Test process management includes different subprocesses such as planning, monitoring, control, and completion, whereas the Dynamic test process includes test design and implementation, test environment set-up and maintenance, test execution, and test incident reporting.

5.1.4. Test Documentation

[1*, c8s12, 20*, c7s8; 6, part 3]

According to [6 part 3], documentation is integral to the formalization of the test process. Test documents can be classified into three hierarchical categories: organizational test documentation, test management documentation and dynamic test documentation. Organizational test documentation includes the information necessary for documenting the test policy and the organizational test strategies. Test management documentation includes the test plan, test status report and test completion report. Finally, dynamic test documentation includes the following documents: test specification (test design specification, test case specification and test procedure specification), test data requirements, test environment requirements, test data readiness report, test environment readiness report, and test execution documentation (such as actual results, test results, test execution log and incident report).

Test documentation should be produced and continuously updated with the same quality as other software engineering documentation. Test documentation should also be under the control of software configuration management. (See the Software Configuration Management KA.)

5.1.5. Test Team

[1*, c16; 2* c23s5; 6 part 2, 6 part 3]

Formalizing the testing process may also involve formalizing the testing team's organization. Considerations of cost, schedule, maturity levels of the involved organizations and criticality of the application can guide the decision. The testing team can be composed of members involved (or not) in the SUT development (i.e., having or not having an unbiased, independent perspective) or internal (or external) personnel. Nowadays, shift-left development does not strongly distinguish among testing team members because the test suite is defined and updated according to the SUT development and delivered code.

5.1.6. Test Process Measures

[1*, c18s3, 20*, c10; 6 part 1, part 2, part 3]

Managers use several measures for the resources spent on testing, as well as for the relative fault-finding effectiveness of the various test phases, to control and improve the testing process, as well as to provide information for managing process risks. Therefore, monitor and control testing must define required data and information and state how to obtain them. The test measures may cover the number of specified, executed, passed, and failed test cases, among other elements. These measures can also be combined with specific process metrics such as residual risk, cumulative defects open and closed, test case progress, and defect detection percentage. 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 risk analysis. Moreover, the resources deemed worth spending on testing should be commensurate with the application's use and criticality. Different techniques have different costs and yield different confidence levels in product reliability.

5.1.7. Test Monitoring and Control

[6 part 1, 6 part 2]

Monitoring and Control comprise an important sub-process of the test management process as in [6 part 2], useful for collecting data and information required during test management and assessment. Usually, monitoring and control activities are executed in parallel with the test execution, and sometimes, data collected might prompt revision of overall process planning. Monitoring assures that testing process activities comply with a specific test plan to mitigate the identified risks satisfactorily. During test monitoring and control, specific documentation (test reports) can regularly be produced to help assess and document the test activity.

5.1.8. Test Completion

[20*, c7s11; 6 part 3]

A decision must be made about how much testing is enough and when a test stage can be completed. Therefore, the purpose of *Test Completion*, a sub-process of the test management process as in [6 part 2], is to ensure that test requirements are satisfied and verified, test reports are completed, and test results are communicated to relevant stakeholders. Thoroughness measures, such as achieved code coverage or functional coverage, and estimates of fault density or 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 and Adequacy Criteria in Section 1.2, Key Issues.) As for the other activities, in this stage, specific documentation is produced (e.g., test completion report) and communicated to the relevant stakeholders.

5.1.9. Test Reusability

[20*, c3; 15]

It is necessary to add complexity and time for test planning and design to achieve reusability of the testing artifacts, such as the test case or execution environment, which is desired when test development is costly, time-consuming, and complex.

Test reusability collects and classifies the testing knowledge (test cases and test results) to make this information searchable and usable for creating new tests or re-executing an existing one. Suitable knowledge-based repositories should be configured and managed to test reusability so changes to software requirements or design can be reflected in changes to the tests.

Currently, the reusability of test cases is pivotal in feature-based or product-line

development and regression testing. Test reusability also relates to maintainability because reusability can reduce the cost and effort involved and improve a test's effectiveness.

5.2. Test Sub-Processes and Activities

[1*c1.12; 1*c12s9; 6 part 2]

In the remainder of this section, the main testing activities and sub-processes are briefly introduced.

5.2.1. Test Planning Process

[1*, c12s1, c12s8; 17; 6 part 2]

Like all other aspects of project management, testing activities must be planned. According to [6 part 2], key aspects of test planning include identification and coordination of personnel, identification of the test objective and completion criteria, definition of test facilities and equipment, creation and maintenance of all test-related documentation, and risk planning and management for possible undesirable outcomes. These activities can be organized at three different levels: (1) process management (i.e., identification of test policies, strategies, processes, and procedures), (2) organizational management (i.e., definition of the test phase, test type and test objective), and (3) design and implementation (i.e., definition of the test environment, the test execution process and monitoring, the completion process, and reporting).

5.2.2. Test Design and Implementation

[1*, c12s1, c12s3; 17]

Generation of test cases is based on the level of testing to be performed and the chosen testing techniques. According to the dynamic test process, as described in [6 part 2], preconditions of the test case generation are the identification of test objectives and the selection of the appropriate testing/demonstration techniques. Test generation focuses on implementing and executing test cases. It often relates to tooling (i.e., using specific software, also called a *test cases generator*). This software accepts inputs (such as source code, test criteria, specifications, or data structure definitions) and uses them to generate the test suites. Sometimes, a test case generator can determine expected results by using a specific oracle facility. This contributes to the full test automation of the overall testing process.

5.2.3. Test Environment Set-up and Maintenance

[1*, c12s6; 2* c8s1; 20* c13s2; 6 part 2; 17]

According to the dynamic test process, as described in [6 part 2], test environment development and setup involve identifying the testing infrastructure. This includes selecting or developing the facilities, hardware, software, firmware, and procedures to conduct the testing activity. The testing environment can be simulated, controlled, and executed in vitro or in vivo. Developing the test environment also involves setting up monitoring and logging facilities useful for documenting the testing activities and assessing the result obtained. The testing environment should be compatible with the other software engineering tools used.

5.2.4. Controlled Experiments and Test Execution

[1*, c12s7, 20* c4s7, 20* c5s6; 6 part 2]

Execution of tests should embody a basic principle of scientific controlled experimentation — everything done during testing should be performed and documented specifically and clearly enough that another person could replicate the results. Hence, testing should be performed following documented procedures using a clearly defined version of the SUT. Especially during acceptance testing, controlled experiments like A/B testing can also be performed to statistically evaluate user

preferences between different versions of the SUT.

5.2.5. Test Incident Reporting

[1*, c13s4, c13s9, c13s11; 2*, c8s3; 20*, c7s8; 6, part 3; 18]

According to the dynamic test process, as described in [6 part 2], testing incidents and reporting focus on the well-defined test data collection process (i.e., identifying when a test was conducted, who performed the test, what software configuration was used, and other relevant identification information). This process and the collected evidence can be leveraged for accountability purposes. Test reporting can involve suitable audit systems to identify unexpected or incorrect test results and record them in a problem reporting system. These data form the basis for later debugging and fixing the problems observed as failures during testing. Also, anomalies not classified as faults could be documented if they later become 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.)

Hence, the *Test Incident Reporting* process focuses on identifying the relevant stakeholders' incidents that could be used to determine what aspects of software testing and other processes need improvement and how effective previous approaches have been.

Part of the incident reporting is also evaluating test results to determine whether 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; sometimes they are 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.

5.3. Staffing

[1*, c16; 6, part 3]

According to [6 part 3], staffing includes defining roles, activities, and responsibilities, specifying hiring needs, and defining training needs. Staffing affects project risk because a lack of required expertise might undermine the teams' ability to meet deadlines and address changing requirements.

The roles, activities and responsibilities definition establishes the following roles and responsibilities: the activity leader and supporting personnel, the test-related roles and their corresponding responsibilities, and the person responsible for providing the test item(s).

Depending on the development lifecycle adopted, typical testing roles include but are not limited to scrum master/test lead, QA/test analyst, test designer, test security/performance engineer and consultant, test environment expert, test executor and test automation consultant or architect.

Hiring needs require the identification of specific requirements for which additional testing personnel are needed to complete the testing process (as well as when that personnel is needed and the desired skills). Depending on the business needs, staffing could take different forms, from internal transfers to external hires or even consultants and/or outsourced resources.

Finally, the training needs specification includes the definition of the required skill level. It also includes the specification of the training activities (such as classroom training, self-paced training, computer-based training, or mentoring) useful for providing the necessary skills to the selected staff.

6. Software Testing in the Development Processes and the Application Domains

[2*, c8, c15; 20*, c4s8, c7]

Whatever development process is adopted, testing remains a fundamental activity. However, specific testing activities or terminologies could be used in some cases, such as the adopted development life cycle and/or the application domain. In the remainder of this section, peculiarities about testing inside the different software development processes and the main application domains are briefly introduced.

6.1. Testing Inside Software Development Processes

[2*, c8; 20*, c7]

In the remainder of this section, peculiarities of testing inside the different development processes are provided.

6.1.1. Testing in Traditional Processes

[1* c18; 20*, c7]

There are a variety of traditional processes, essentially based on the SUT development principles, that can be adopted within the organization. Sequential, V, spiral model and iterative are just some of the processes commonly applied. (Software Life Cycles in the Software Engineering Process KA provides a detailed description of each.) However, in all these processes, testing is just one perceived activity; it is sometimes performed at the end of the process, with a tangible risk of SUT development failure in case of deviation of the end-user needs or assessment issues. During recent years, to evaluate and control the overall quality of the SUT, initiatives such as Test Maturity Model integration (TMMi) and Software Process Improvement (SPI) have been established. As a result, different existing frameworks have been updated or improved for the purpose, such as Software Process Improvement and Capability Determination (SPICE), Capability Maturity Model (CMM), and Rational Unified Process (RUP).

For instance, CMM is one of the most referenced models; it can guide key SUT stakeholders in gaining control of their development and maintenance processes. It is, in fact, a well-defined set of best practices in software testing that improves SUT quality by increasing customer satisfaction.

Presented in the early 2000s, the RUP model can be seen as a predecessor of the left-shift movement. RUP encourages testing early by offering several mechanisms to integrate testing more closely with the software development effort, making testing a distinct discipline. Furthermore, RUP promotes an iterative development approach for continuously verifying quality. It also enables use cases and risk to drive SUT development and allows strategic change management. Indeed, RUP groups the SUT increments and SUT iterations into four phases: inception, elaboration, construction, and transition.

Nowadays, RUP can be considered both Iterative and Agile — Iterative because all the core activities are repeated throughout the SUT development project, and Agile because the defined phases of the chosen lifecycle can be repeated until the SUT meets requirements (both functional and non-functional), achieves the defined objectives, and guarantees the target quality.

6.1.2. Testing in Line with Left-Shift Movement

[2*, c3, c8s2; 6, part 1; 16, c3, c5]

The *left-shift testing movement* promotes the adoption of testing in the early stages of software development to detect and remove faults as early as possible to increase overall SUT quality and reduce the cost and risks of testing activities. Currently, different

development life cycles, such as Agile, DevOps and TDD, belong to the left-shift movement. (See Agile Methods in the Software Engineering Process KA.)

In left-shift-based development, different testing aspects should be considered:

A. The internal code quality: Regression, prioritization, security, and privacy could be the primary objectives of the internal code quality (Section 2.2). Usually, unit testing and integration testing are the targeted levels (Section 2.1), whereas structure-based is the main testing technique (Section 3.2).
B. Business needs: Compliance and conformance, usability, security, and privacy are just a subset of the possible objectives of the business needs aspect (Section 2.2). Concerning this aspect, testing focuses more on the system and acceptance test levels and on end-user expectations, as well as usage-based (Section 3.5) and scenario-based techniques (Section 3.1.8).
C. Perceived quality: Alpha, beta, installation, usability, security, and privacy could be the primary objectives of the internal perceived quality (Section 2.2). Perceived quality usually focuses on the acceptance test level and is achieved by applying techniques based on software engineering's intuition and experience (Section 3.3) and usage-based and fault-based techniques, such as mutation testing (Section 3.4).
D. Quality assurance: Performance installations, security, and privacy conformance and compliance are some main objectives of quality assurance (Section 2.2). This aspect may involve all testing levels, and the selection of the testing technique depends on the objective and the level chosen.

Here, some examples of testing inside the different left-shift movements implementation are provided:

- In Agile process development, testing activities involve all stakeholders (such as customers and team personnel) and target the identification of where improvements could be made in future interactions. Managing the risk of regression defects, meeting changing requirements, and managing their impact on test artifacts are also objectives of the Agile testing process. Typically, test automation is used to manage the regression risk, and exploratory testing may be used to manage a lack of detailed requirements.
- In TDD, the test cases mainly target the software requirements specifications and acceptance, and they are generated in advance of the code being written. The tests are based on the user stories and implemented using automated component testing tools. Indeed, TDD is a practice that requires defining and maintaining unit tests and can help clarify the user needs and software requirements specifications.
- In testing automated builds and continuous integration (for instance, DevOps), the SUT is continuously developed, integrated, delivered and monitored. In this process, regression testing is continuously performed to timely identify and correct development and integration issues. Additionally, quick testing techniques, such as smoke testing, are commonly used during continuous integration to guarantee that the SUT

is testable before it is released to the operational stage.

6.2. Testing in the Application Domains

[2*, c15; 20*, c4s8]

Usually, an application domain is strictly connected to a certain reality. Therefore, testing approaches could be tailored to the needs of the domain and customized to the adopted technologies.

Below, we provide an overview of different aspects and solutions for software testing applied within several domain-specific environments:

- Automotive domain testing: Due to the complexity of automotive systems, this testing involves aspects of almost every software component and its interaction with hardware. Security testing, simulation testing, reliability/life cycle testing, integrated systems testing, data acquisition and signal analysis testing, quality testing and inspection, and stress/strain testing are just some of the various testing performed in this domain. Several supporting standards are currently available to guide and manage automotive testing according to the peculiarity, the component, or the quality aspect that should be assessed. Autosar [2] and Automotive SPICE [3] are examples.
- Internet of Things (IoT) domain testing: This testing involves application development, device management, system management, heterogeneity management, data management, and tools for analysis, deployment, monitoring, visualization and research. Additionally, security, privacy, communications and user/component interaction should be considered in the quality assessment. For example, guidelines and specific conformance test suites for cybersecurity assessment of the IoT SUT are detailed in the European Telecommunications Standards Institute (ETSI) standards.[4]
- Legal domain testing: One of the most important aspects in the legal domain is the management of highly sensitive users; therefore, security, privacy and trust are the most common areas of focus for testing. Additionally, because of the copious data collected and exchanged, performance testing of the data repository, testing to show accurate communication and integration testing, as well as consistency and compliance testing, should also be done. Finally, because the legal domain is characterized by specific nomenclature and jargon, involving legal domain experts in test case generation is common practice to ensure a focus on desired characteristics and quality.
- Mobile domain testing: This testing is usually for usability, functional, configuration and consistency assessment. Mobile-specific aspects such as screen resolution, Global Positioning

[2] https://www.autosar.org/
[3] https://www.automotivespice.com/
[4] https://www.etsi.org/

System (GPS), operating systems, and device manufacturers should also be considered during testing activity. Finally, the type of mobile applications (native or web apps) and their interactions need to be tested. For example, the W3C Web and Mobile Interest Group [5] provides facilities, guidelines and ad hoc test suites useful for developing and testing web-based content, applications and services.

- Avionics domain testing [6] : Usually, avionics systems include several independent or loosely coupled components and commercial off-the-shelf products. Those forces testing to include very general processes and approaches applicable at both the system and the process levels. Functional and non-functional, integration, communication operational, stress, safety, and security testing are examples of possible approaches. As in other domains, supporting standards such as Aeronautical Radio Incorporated (ARINC) Standards and ASTM F3153-15 can be used for reference.
- Healthcare domain testing: Healthcare domain testing should ensure quality in areas such as secure and reliable data exchange, stable performance, privacy, and safety. Interoperability, usability, performance and compliance with industry regulations, as well as security and safety standards (such as the Health Level Seven (HL7), [7] Fast Healthcare Interoperability Resources (FHIR), [8] Digital Imaging and Communications in Medicine (DICOM), [9] Health Insurance Portability and Accountability Act (HIPAA),[10] and the General Data Protection Regulation (GDPR) [11]) should also be considered.
- Embedded domain testing: Because software and hardware are tightly coupled in embedded systems, testing activity should assess functional and non-functional attributes of both software and hardware.
- Graphical User Interface (GUI) testing: GUI testing involves assessing the UI (User Interface) (i.e., the elements of the user objects that we can see). Thus, GUI testing targets the design pattern, images, alignment, spellings, and the overall look and feel of the UI. Testing approaches based on finite-state machines, goal-driven approaches, approaches based on abstractions and model-based approaches can be considered.
- Gaming: Gaming applications and software are currently a very active sector of software production, causing increased demand for new approaches and ways to ensure their quality and security. Among the specific testing techniques, playtesting is one of the most adopted. In this

[5] https://www.w3.org/2013/07/webmobile-ig-charter.html
[6] www.astm.org.
[7] https://www.hl7.org/
[8] http://fhir.org/
[9] https://www.dicomstandard.org/
[10] https://www.hhs.gov/hipaa/.
[11] https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679.

case, real gamers repeat quality control methods at many points of the game execution or design process. GUI testing, functionality testing, security testing, console testing, compliance testing and performance testing can also be considered.

- Real-time domain testing: Real-time testing usually focuses on assessing timing constraints and deterministic behavior. Usually, unit, integration and system testing approaches can be adopted. Communication, interaction and behavioral testing can also be performed.
- Service Oriented Architecture (SOA) testing: This testing focuses mainly on correctly implementing business processes and involves unit and integration testing approaches. Structure-based, specification-based and security testing can be applied. The testing activity might vary according to the environment, organization and set of requirements that should be satisfied.
- Finance domain testing: This testing covers a wide range of aspects, from managing financial requirements to assessing financial applications and software programs. As in other domains, domain-specific knowledge (such as that held by, for example, banks, credit unions, insurance companies, credit card companies, consumer finance businesses, investment funds and stock brokerages) could be necessary to apply the testing process effectively and efficiently. Customer satisfaction, usability, security, privacy, third-party component and apps integrations, real-time issues, and performance are some of the most important challenges in this domain.

7. Testing of and Testing Through Emerging Technologies

In recent decades, software development was driven by emerging trends such as the widespread diffusion of mobile technology, cloud infrastructures adoption, big data analysis and the software as a service paradigm, which highlighted new constraints and challenges for testing.

7.1. Testing of Emerging Technologies

- Testing Artificial Intelligence (AI), ML/Deep Learning (DL) [19]: AI, ML and DL are successfully being applied in practice. Sooner or later, most business applications will have some form of AI, ML or DL. Because of their peculiarities, testing such applications is challenging and might be very expensive. AI, ML or DL testing refers to any activity designed to reveal AI, ML or DL bugs.
    - Three main aspects should be considered in defining bugs and testing in this scenario: the required conditions (correctness, robustness, security, and privacy); the AI, ML or DL items (e.g., a bug might exist in the data, the learning program, or the framework used); and the involved testing activities (test case generation, test oracle identification and

definition, and test case adequacy criteria).

  - In all these applications, a prototype model is first generated based on historical data. Then, off-line testing, such as cross-validation, is conducted to verify that the generated model satisfies the required conditions. Usually, after deployment, the model is used for prediction purposes by generating new data. Finally, the generated data is analyzed through online testing to evaluate how the model interacts with user behaviors.

- Testing blockchain [21]: The commonly used testing techniques for validating blockchains and related applications such as smart contracts are stress testing, penetration testing and property testing. However, depending on the specific situation, different aspects should be considered during the testing of a blockchain-based SUT, such as the following:
- Platform type: The level of validation depends on the type of platform used for implementation — public or private. The latter requires a much greater testing effort.
- Connection with other applications: Integration testing should be performed to check consistency when the blockchain works with various applications.
- Performance: Performance testing should be conducted when performance issues are a concern. Specific strategies to handle many transactions should be conceived to guarantee a satisfactory performance level. Qualitative and quantitative metrics, such as average transaction validation latency and security, should also be considered.
- Testing the cloud [1*, c10s10, 2*, c18]: Testing the cloud validates the quality of applications and infrastructures deployed in the cloud by considering both functional and non-functional properties. The focus is to identify problems posed by systems residing in the cloud. Therefore, testing activities use techniques to validate cloud-based services' performance, scalability, elasticity and security. Moreover, testing should also focus on compatibility and interoperability among heterogeneous cloud resources when different deployment models are used (e.g., private, public or hybrid).
- Testing concurrent and distributed applications [1*, c10s10, 2*, c17]: One main aspect of testing dynamic, complex, distributed or concurrent applications is dealing with multiple operating systems and updates, multiple browser platforms and versions, different types of hardware, and many users. For such testing, it's difficult to use testing approaches based on the classical hierarchy between components or systems; instead, solutions based on input/output, dependency threads, or dynamic relations often work better. Additionally, the possibility of continuous integration and deployment of the different components forces the testing process to include approaches for managing continuous test operation, injection, monitoring and reporting according to the time, bandwidth usage, throughput, and adaptability constraints. Finally, there is still the

need for solutions that allow the reusability of testing knowledge, architectures, and code to make the testing activity more effective and less expensive.

7.2. Testing Through Emerging Technologies

- Testing through ML [19]: AI, ML or DL techniques are successfully used to reduce the effort involved in several activities in software engineering (such as behavior extraction, testing or bug fixing). These techniques aid both researchers and practitioners in adopting and identifying appropriate methods for their desired applications. There is a growing interest in adopting ML techniques in software testing because most software testing issues are being formulated as ML learning problems. Indeed, AI, ML or DL is intensively used in almost all software, such as test case design, the oracle problem, test case evaluation, test case prioritization and refinement, and mutation testing automation. Indeed, they can reduce maintenance efforts and improve the overall SUT quality because of their ability to analyze large amounts of data for classifying, triaging and prioritizing bugs more efficiently. From a DevOps perspective, AI, ML and DL solutions can be used in SUT automation authoring and execution phases of test cases, as well as in the post-execution test analysis that identifies trends, patterns and impact on SUT testing activity.

- Testing through blockchain [21]: Testing becomes complicated when different teams, domain experts and users need to work together in collaborative, large-scale systems and complex software systems to achieve a common goal. This is mainly because of the time constraint, data sharing policies, acceptance criteria and trusted coordination among the teams involved in the testing process. Blockchain technologies can be exploited to improve software testing efficiency and avoid using centralized authority to manage different testing activities. This can help ensure distributed data management, tamper resistance, auditability, and automatic requirement compliance to improve the quality of software testing and development. Blockchain-based approaches for trusted test case repository management and to support test-based software and security testing are also considered.

- Testing through the cloud [23]: Testing through the cloud refers to SUT testing performed by leveraging scalable cloud technologies. Usually, the cloud is used for testing purposes wherever large-scale simulations and elastic resources are necessary. Indeed, this can affect cost reduction, development, and maintenance of the testing infrastructure (scaffolding), and online validation of systems, such as ML-based SUT. A particular situation is the testing of the cloud through the cloud itself. This is an example of the intersection between testing of and testing through emerging technologies. The applications and infrastructures deployed in the cloud can be tested, exploiting the cloud's bandwidth.

- Testing through simulation [1*, c3s9]: Simulation is an important technology for testing activity because it represents a valid means for evaluating SUT execution under

critical situations or disasters or assessing specific behaviors or recovering activities. The complexity of the testing approach might vary according to the complexity of the simulation system adopted and might involve closed-loop testing; assessing the devices, communications and interface; and use of real-time data (e.g., voltage, current and breaker status). Simulation testing can be applied to each development level and might involve mathematical, formal representation of the real system, environment, network conditions and control devices. Simulation testing is currently adopted in many application domains. Especially in the automotive and embedded domain, among the different proposals, one of the emerging solutions for simulation testing is hardware-in-the-loop (HIL) simulation testing. In this case, real signals sent to the SUT to simulate reality and to test and design the iteration are continuously performed while the real-world system is being used.

- Testing through crowdsourcing [22]: Crowdsourced testing (also known as crowdtesting) is an emerging approach for involving users and experts in the testing activity. Thus, crowdsourcing uses represent the dispersed, temporary workforce of multiple individual testers. Testing through crowdsourcing is mainly used for testing mobile applications because it ensures technology diversity and customer-centric validation. However, crowdtesting is not a substitute for in-house SUT validation. It represents a valid means of detecting failures and issues because it involves many individuals (testers) in different locations, who are using different technologies in different conditions and who have different skills and knowledge.

8. Software Testing Tools

[1*, c12s11, 20*, c7]

Several testing tools focus on the SUT peculiarities and needs. This section describes the main issues and challenges concerning testing tools and provides an overview of their currently identified categories.

8.1. Testing Tool Support and Selection

[1*, c12s11, 20*, c7]

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

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

8.2. Categories of Tools

[1*, c1, c3, c4, c7, c8, c9, c12]

Several classifications of testing tools mainly describe their functionalities, such as the following:

- *Test harnesses* (drivers, stubs) [1*, c3s9] provide a controlled environment in which tests can be

launched and the test outputs can be logged. Drivers and stubs are provided to execute parts of a SUT to simulate calling and called modules.

- *Test generators* [1*, c12s11] assist in generating test cases. That generation can be random, path-based, model-based or a mix thereof.
- *Capture/replay tools* [1*, c12s11] automatically re-execute or replay previously executed tests that 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.
- *Coverage analyzers and instrumenters* [1*, c4] work together. Coverage analyzers assess which and how many entities of the program flow graph have been exercised among all those required by the selected test coverage criterion. The analysis can be done through SUT 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 re-execution of a test suite after a section of software has been modified. They can also help select a test subset according to the change made.
- *Reliability evaluation tools* [9*, c8] support test results analysis and graphical visualization to assess reliability-related measures according to selected models.
- *Injection-based tools* [1*, c3, c7s7] focus on introducing or reproducing specific problems to confirm that the SUT behaves suitably under the corresponding condition. That can involve managing some input or triggering of events. Usually, two categories of injection-based tools are considered: attack injection and fault injection.
- *Simulation-based tools* [1*, c3s9] verify and validate selected properties. Usually, they exploit specific models to enable the automated execution of scenarios to assess whether the SUT operates as expected or to predict how the SUT would respond to defined inputs. Typical simulation-based tools are classified into tools for verification, tools for collaboration, tools for optimization, tools for testing automated systems and tools for evaluating software concepts.
- *Security testing tools* [1*, c8s3, c12s11] focus on specific security vulnerabilities. Among these are tools for attack injection, penetration testing and fuzzy testing.
- *Test management tools* [1*, c12s11] include all the supporting tools that assure efficient and effective test management and data collection.
- *Cross-browser testing tools* [1*, c8s3] enable the tester to quickly build and run user interface test cases across desktop, mobile and web applications to check whether the SUT looks and works as expected on every device and browser.
- *Load testing tools* [1*, c3] collect valuable data and evidence for SUT performance evaluations.
- *Defect tracking tools* [1*, c3] help keep track of detected faults during the SUT development projects. These tools behave as tracking systems and usually allow end users to enter fault reports directly.
- *Mobile testing tools* [1*, c8s3] support the implementation and testing of mobile apps by allowing several repeated UI tests over the application

platform, development on real mobile devices or emulators, testing of the mobile apps on real-time implementations and collection of data for specific QA measures.

- *API testing tools* [1*, c7s2] check whether the applications meet functionality, performance, reliability and security expectations throughout the automation of specific API tests.
- *CSS validator tools* [1*, c7s2] validate Cascading Style Sheets (CSS) code and discover errors, issues and warnings that can be fixed. The CSS Validation Service, provided by W3C for free, is one of the most used validators in practice that helps both web designers and web developers check CSS.
- *Web application testing tools* [1*, c8s3], also referred to as web testing tools, support validating the functionality and the performance of web-based SUTs before their deployment into production. These tools provide relevant insight and data for different stakeholders, such as developers, servers, and infrastructure administrators. From a DevOps perspective, these tools address issues, or bugs before SUTs are available to end users.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	1*	2*	10*	20*
1. Software Testing Fundamentals	c1, c2	c8		c7
1.1. Faults vs. Failures	c1s5	c1		c1s3
1.2. Key Issues				
1.2.1. Test Case Creation	c12s1, c12s3	c8		
1.2.2. Test Selection and Adequacy Criteria	c1s14, c6s6, c12s7	c8		
1.2.3. Prioritization/Minimization				
1.2.4. Purpose of Testing	c13s11, c11s4	c8		
1.2.5. Assessment and Certification		c7, c25		
1.2.6. Testing for Quality Improvement/Assurance	c16s2			
1.2.7. The Oracle Problem	c1s9, c9s7			
1.2.8. Theoretical and Practical Limitations	c2s7			
1.2.9. The Problem of Infeasible Paths	c4s7			
1.2.10. Testability	c17s2			
1.2.11. Test Execution and Automation				
1.2.12. Scalability	c8s7			
1.2.13. Test Effectiveness	c1s1	c8s1		
1.2.14. Controllability, Replication and Generalization	c12s12			
1.2.15. Off-line vs. Online Testing				
1.3. Relationship of Testing to Other Activities				
2. Test Levels	c1s13	c8s1		
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		
2.1.4. Acceptance Testing	c1s7	c8s4		
2.2. Objectives of Testing	c1s7			
2.2.1. Conformance Testing	c10s4			
2.2.2. Compliance Testing	c12s3			
2.2.3. Installation Testing	c12s2			
2.2.4. Alpha and Beta Testing	c13s7, c16s6	c8s4		
2.2.5. Regression Testing	c8s11, c13s3			
2.2.6. Prioritization Testing	c12s7			
2.2.7. Non-functional testing	c8s7, c14s2, c17s2	c8, c17		
2.2.8. Security Testing		c13		
2.2.9. Privacy Testing		c13, c14		
2.2.10. Interface and API Testing		c8s1		c7s12
2.2.11. Configuration Testing	c8s5			
2.2.12. Usability and Human-Computer Interaction Testing		c8s4	c6	
3. Test Techniques	c1s15			
3.1. Specification-Based Techniques	c6s2			
3.1.1. Equivalence Partitioning	c9s4			
3.1.2. Boundary-Value Analysis	c9s5			
3.1.3. Syntax Testing	c10s11	c5		
3.1.4. Combinatorial Test Techniques	c9s3			
3.1.5. Decision Table	c9s6, c13s6			
3.1.6. Cause-Effect Graphing	c1s6			
3.1.7. State Transition Testing	c10			

3.1.8. Scenario Testing		c8s3.2, c19s3.1		
3.1.9. Random Testing	c9s7			
3.1.10. Evidence-Based				
3.1.11. Forcing Exception				
3.2. Structure-Based Test Techniques				
3.2.1. Control Flow Testing	c4			
3.2.2. Data Flow Testing	c5			
3.2.3. Reference Models for Structure-Based Test Techniques	c4			
3.3. Experience-Based Techniques				
3.3.1. Error Guessing	c9s8			
3.3.2. Exploratory Testing				
3.3.3. Further Experience-Based Techniques				
3.4. Fault-Based and Mutation Techniques	c1s14, c3s5			
3.5. Usage-Based Techniques	c15s5			
3.5.1. Operational Profile	c15s5	c11		
3.5.2. User Observation Heuristics	c5, c7			
3.6. Techniques Based on the Nature of the Application		c16, c17, c18, c20, c21		c4s8
3.7. Selecting and Combining Techniques				c7s12
3.7.1. Combining Functional and Structural	c9			
3.7.2. Deterministic vs. Random	c9s6			
3.8. Techniques Based on Derived Knowledge		c19, c20		c7
4. Test-Related Measures		c24s5		c10
4.1. Evaluation of the SUT		c24s5		
4.1.1. SUT Measurements That Aid in Planning and Designing Tests			c10	
4.1.2. Fault Types, Classification and Statistics	c13s4, c13s5, c13s6			
4.1.3. Fault Density	c13s4			c10s1
4.1.4. Life Test, Reliability Evaluation	c15	c11		c1s3
4.1.5. Reliability Growth Models	c15	c11s5		
4.2. Evaluation of the Tests Performed				
4.2.1. Fault Injection	c2s5			
4.2.2. Mutation Score	c3s5			
4.2.3. Comparison and Relative Effectiveness of Different Techniques	c1s7			
5. Test Process		c8		
5.1. Practical Considerations				
5.1.1. Attitudes/Egoless Programming	c16	c3		
5.1.2. Test Guides and Organizational Process	c12s1	c8		c7s3
5.1.3. Test Management and Dynamic Test Processes	c12			c7s3
5.1.4. Test Documentation	c8s12			c7s8
5.1.5. Test Team	c16	c23s5		
5.1.6. Test Process Measures	c18s3			c10
5.1.7. Test Monitoring and Control				
5.1.8. Test Completion				c7s11
5.1.9. Test Reusability		c3		
5.2. Test Sub-Processes and Activities	c12s9, c1s12			
5.2.1. Test Planning Process	c12s1, c12s8			
5.2.2. Test Design and Implementation	c12s1, c12s3			

5.2.3. Test Environment Set-up and Maintenance	c12s6	c8s1		c13s2
5.2.4. Controlled Experiments and Test Execution	c12s7			c4s7, c5s6
5.2.5. Test Incident Reporting	c13s4, c13s9, c13s11	c8s3		c7s8
5.3. Staffing	c16			
6. Software Testing in the Development Processes and the Application Domains		c8, c15		c4s8, c7
6.1. Testing Inside Software Development Processes		c8		c7
6.1.1. Testing in Traditional Processes	c18			c7
6.1.2. Testing in Line with Left-Shift Movement		c3, c8s2		
6.2. Testing in the Application Domains		c15		c4s8
7. Testing of and Testing Through Emerging Technologies				
7.1. Testing of Emerging Technologies				
7.2. Testing Through Emerging Technologies				
8. Software Testing Tools	c12s11			c7
8.1. Testing Tool Support and Selection	c12s11			c7
8.2. Categories of Tools	c1, c3, c4, c7, c8, c9, c12, c15			

References

[1*] Naik, S., and Tripathy P., *Software Testing and Quality Assurance: Theory and Practice*, ed: Wiley, 2008, p. 648.

[2*] Sommerville, I., *Software Engineering*, 10th ed., Addison-Wesley, 2016.

[3] ~~Lyu,, M.R., Ed., *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, McGraw-Hill, 1996.~~

~~[4] Zhu, H., et al., Software Unit Test Coverage and Adequacy, *ACM Computing Surveys,* vol. 29, Dec. 1997, pp. 366-427.~~

[5] Dijkstra, E.W., *Notes on Structured Programming*, Technological University, Eindhoven, 1970.

[6] ISO/IEC/IEEE 29119 — System and software engineering — Software testing, 2016.

[7] ISO/IEC/IEEE International Standard — Systems and software engineering — Vocabulary, in *ISO/IEC/IEEE 24765:2017(E)*, Aug. 28, 2017, pp. 1-541.

[8] ~~Yoo, S., and M. Harman, Regression Testing Minimization, Selection and Prioritization: A Survey, *Software Testing Verification and Reliability,* vol. 22, Mar. 2012, pp. 67-120.~~

~~[9*] Kan,, S.H., *Metrics and Models in Software Quality Engineering*, 2nd ed. Boston: Addison-Wesley, 2002~~.

[10*] Nielsen, J., *Usability Engineering*, 1st ed., Boston: Morgan Kaufmann, 1993.

~~[11] T. Y. Chen, et al., Adaptive Random Testing: The ART of Test Case Diversity, *Journal of Systems and Software,* vol. 83, Jan. 2010, pp. 60-66.~~

[12] Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., and Harman, M., Chapter Six — Mutation Testing Advances: An Analysis and Survey, *Adv. Comput.* 112, 2019: 275-378.

[13] Utting, M., Legeard, B., Bouquet, F., Fourneret, E., Peureux, F., and Vernotte, A., Recent advances in model-based testing, *Advances in Computers*, 101, 2016, pp. 53-120.

[14] IEEE Std 1012-2016, IEEE Standard for System, Software, and Hardware Verification, and Validation, ed. 2016.

[15] ISO/IEC 25010:2011, Systems and software engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models, ed. 2011.

[16] IEEE 2675-2021, IEEE Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package, and Deployment, ed. 2021.

[17] Software Engineering Competency Model (SWECOM), v1.0, 2014.

[18] IEEE Std 1028-2008, IEEE Standard for Software Reviews and Audits, ed. 2008.

[19] Riccio, V., Jahangirova, G., Stocco, A., et al., Testing machine learning based systems: A systematic mapping, *Empir Software Eng*, 25, 2020, pp. 5193-5254.

[20*] Laporte, C.Y., and April, A., *Software Quality Assurance*, IEEE Computer Society Press, 1st ed., 2018.

[21] Demi, S., Colomo-Palacios, R., and Sánchez-Gordón, M., Software Engineering Applications Enabled by Blockchain Technology: A Systematic Mapping Study, *Applied Sciences*, 11(7), 2021, pp. 2960.

[22] Mao, Ke, Capra, Licia, Harman, Mark, and Jia, Yue. A survey of the

use of crowdsourcing in software engineering, *Journal of Systems and Software*, 126, 2017, pp. 57-84.

[23] Bertolino, A., Angelis, G.D., Gallego, M., García, B., Gortázar, F., Lonetti, F., and Marchetti, E., A systematic review on cloud testing, *ACM Computing Surveys (CSUR)*, 52(5), 2019, pp. 1-42.

[24] Achary, Rathnakar, and Pethuru, Raj, *Cloud Reliability Engineering: Technologies and Tools*, CRC Press, 2021.

CHAPTER 6

SOFTWARE CONFIGURATION MANAGEMENT

ACRONYMS

CCB	Configuration Control Board
CM	Configuration Management
FCA	Functional Configuration Audit
PCA	Physical Configuration Audit
SCCB	Software Configuration Control Board
SCI	Software Configuration Item
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SCR	Software Change Request
SCSA	Software Configuration Status Accounting
SDD	Software Design Document
SEI/ CMMI	Software Engineering Institute's Capability Maturity Model Integration
SQA	Software Quality Assurance
SRS	Software Requirement Specification

INTRODUCTION

A system can be defined as the combination of interacting elements organized to achieve one or more stated purposes [1]. The configuration of a system is the functional and physical characteristics of hardware or software as set forth in technical documentation or achieved in a product [1]; it can also be thought of as a collection of specific versions of hardware, firmware, or software items combined according to specific build procedures to serve a particular purpose. Configuration management (CM), then, is the discipline of identifying the configuration of a system at distinct points in time for the purpose of systematically controlling changes to the configuration and maintaining the integrity and traceability of the configuration throughout the system life cycle. It is formally defined as

> A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements. [1]

Software configuration management (SCM) is a supporting-software life cycle process that benefits project management, development and maintenance activities, quality assurance activities, as well as the customers and users of the end product.

The concepts of configuration management apply to all items to be controlled, although there are some differences in implementation between hardware CM and software CM.

SCM is closely related to the software quality assurance (SQA) activity. As defined in the Software Quality knowledge area (KA), SQA processes provide assurance that the software products and processes in the project life cycle conform to their specified requirements by planning, enacting, and performing a set of activities to provide adequate confidence that quality is being built into the software. SCM activities help in accomplishing these SQA goals. In some project contexts, specific SQA requirements prescribe certain SCM activities.

Figure 6.1. Breakdown of Topics for the Software Configuration Management KA

The SCM activities are management and planning of the SCM process, software configuration identification, software configuration control, software configuration status accounting, software configuration auditing, and software release management and delivery.

The Software Configuration Management KA is related to all the other KAs, since the object of configuration management is the artifact produced and used throughout the software engineering process.

BREAKDOWN OF TOPICS FOR SOFTWARE CONFIGURATION MANAGEMENT

The breakdown of topics for the Software Configuration Management KA is shown in Figure 6.1.

1. Management of the SCM Process

SCM controls the evolution and integrity of a product by identifying its elements; managing and controlling change; and verifying, recording, and reporting on configuration information. From the software engineer's perspective, SCM facilitates development and change implementation activities. A successful SCM implementation requires careful planning and management. This, in turn, requires an understanding of the organizational context for, and the constraints placed on, the design and implementation of the SCM process.

1.1. Organizational Context for SCM

[2*, c6, ann. D] [3*, introduction] [4*, c29]

To plan an SCM process for a project, it is necessary to understand the organizational context and the relationships among organizational elements. SCM interacts with several other activities or organizational elements.

The organizational elements responsible for the software engineering supporting processes may be structured in various ways. Although the responsibility for performing certain SCM tasks might be assigned to other parts of the organization (such as the development organization), the overall responsibility for SCM often rests with a distinct organizational element or designated individual.

Software is frequently developed as part of a larger system containing hardware and firmware elements. In this case, SCM activities take place

in parallel with hardware and firmware CM activities and must be consistent with system-level CM. Note that firmware contains hardware and software; therefore, both hardware and software CM concepts are applicable.

SCM might interface with an organization's quality assurance activity on issues such as records management and nonconforming items. Regarding the former, some items under SCM control might also be project records subject to provisions of the organization's quality assurance program. Managing nonconforming items is usually the responsibility of the quality assurance activity; however, SCM might assist with tracking and reporting on software configuration items falling into this category.

Perhaps the closest relationship is with the software development and maintenance organizations. It is within this context that many of the software configuration control tasks are conducted. Frequently, the same tools support development, maintenance, and SCM purposes.

1.2. Constraints and Guidance for the SCM Process

[2*, c6, ann. D, ann. E] [3*, c2, c5] [5*, c19s2.2]

Constraints affecting, and guidance for, the SCM process come from a number of sources. Policies and procedures set forth at corporate or other organizational levels might influence or prescribe the design and implementation of the SCM process for a given project. In addition, the contract between the acquirer and the supplier might contain provisions affecting the SCM process. For example, certain configuration audits might be required, or it might be specified that certain items be placed under CM. When software products to be developed have the potential to affect public safety, external regulatory bodies may impose constraints. Finally, the particular software life cycle process chosen for a software project and the level of formalism selected to implement the software affect the design and implementation of the SCM process.

Guidance for designing and implementing an SCM process can also be obtained from "best practice," as reflected in the standards on software engineering issued by the various standards organizations (see Appendix B on standards).

1.3. Planning for SCM

[2*, c6, ann. D, ann. E] [3*, c23] [4*, c29]

The planning of an SCM process for a given project should be consistent with the organizational context, applicable constraints, commonly accepted guidance, and the nature of the project (for example, size, safety criticality, and security). The major activities covered are software configuration identification, software configuration control, software configuration status accounting, software configuration auditing, and software release management and delivery. In addition, issues such as organization and responsibilities, resources and schedules, tool selection and implementation, vendor and subcontractor control, and interface control are typically considered. The results of the planning activity are recorded in an SCM Plan (SCMP), which is typically subject to SQA review and audit.

Branching and merging strategies should be carefully planned and communicated, since they impact many SCM activities. From an SCM standpoint, a branch is defined as a set of evolving source file versions [1]. Merging consists in combining different changes to the same file [1]. This typically occurs when more than one person changes a configuration item. There are many branching and merging strategies in common use (see the Further Readings section for additional discussion).

The software development life cycle model (see Software Life Cycle Models in the Software Engineering Process KA) also impacts SCM activities, and SCM planning should take this into account. For instance, continuous integration is a common practice in many software development approaches. It is typically characterized by frequent build-test-deploy cycles. SCM activities must be planned accordingly.

1.3.1. SCM Organization and Responsibilities

[2*, ann. Ds5, ann. Ds6] [3*, c10-11] [4*, introduction, c29]

To prevent confusion about who will perform given SCM activities or tasks, organizational

roles to be involved in the SCM process need to be clearly identified. Specific responsibilities for given SCM activities or tasks also need to be assigned to organizational entities, either by title or by organizational element. The overall authority and reporting channels for SCM should also be identified, although this might be accomplished at the project management or quality assurance planning stage.

1.3.2. SCM Resources and Schedules
[2*, ann. Ds8] [3*, c23]

Planning for SCM identifies the staff and tools involved in carrying out SCM activities and tasks. It addresses scheduling questions by establishing necessary sequences of SCM tasks and identifying their relationships to the project schedules and milestones established at the project management planning stage. Any training requirements necessary for implementing the plans and training new staff members are also specified.

1.3.3. Tool Selection and Implementation
[3*, c26s2, c26s6] [4*, c29s5]

As for any area of software engineering, the selection and implementation of SCM tools should be carefully planned. The following questions should be considered:

- Organization: what motivates tool acquisition from an organizational perspective?
- Tools: can we use commercial tools or develop them ourselves?
- Environment: what are the constraints imposed by the organization and its technical context?
- Legacy: how will projects use (or not) the new tools?
- Financing: who will pay for the tools' acquisition, maintenance, training, and customization?
- Scope: how will the new tools be deployed—for instance, through the entire organization or only on specific projects?
- Ownership: who is responsible for the introduction of new tools?
- Future: what is the plan for the tools' use in the future?
- Change: how adaptable are the tools?
- Branching and merging: are the tools' capabilities compatible with the planned branching and merging strategies?
- Integration: do the various SCM tools integrate among themselves? With other tools in use in the organization?
- Migration: can the repository maintained by the version control tool be ported to another version control tool while maintaining complete history of the configuration items it contains?

SCM typically requires a set of tools, as opposed to a single tool. Such tool sets are sometimes referred to as workbenches. In such a context, another important consideration in planning for tool selection is determining if the SCM workbench will be *open* (in other words, tools from different suppliers will be used in different activities of the SCM process) or *integrated* (where elements of the workbench are designed to work together).

The size of the organization and the type of projects involved may also impact tool selection (see topic 7, Software Configuration Management Tools).

1.3.4. Vendor/Subcontractor Control
[2*, c13] [3*, c13s9, c14s2]

A software project might acquire or make use of purchased software products, such as compilers or other tools. SCM planning considers if and how these items will be taken under configuration control (for example, integrated into the project libraries) and how changes or updates will be evaluated and managed.

Similar considerations apply to subcontracted software. When using subcontracted software, both the SCM requirements to be imposed on the subcontractor's SCM process as part of the subcontract and the means for monitoring compliance need to be established. The latter includes consideration of what SCM information must be available for effective compliance monitoring.

1.3.5. Interface Control

[2*, c12] [3*, c24s4]

When a software item will interface with another software or hardware item, a change to either item can affect the other. Planning for the SCM process considers how the interfacing items will be identified and how changes to the items will be managed and communicated. The SCM role may be part of a larger, system-level process for interface specification and control; it may involve interface specifications, interface control plans, and interface control documents. In this case, SCM planning for interface control takes place within the context of the system-level process.

1.4. SCM Plan

[2*, ann. D] [3*, c23] [4*, c29s1]

The results of SCM planning for a given project are recorded in a software configuration management plan (SCMP), a "living document" which serves as a reference for the SCM process. It is maintained (that is, updated and approved) as necessary during the software life cycle. In implementing the SCMP, it is typically necessary to develop a number of more detailed, subordinate procedures defining how specific requirements will be carried out during day-to-day activities—for example, which branching strategies will be used and how frequently builds occur and automated tests of all kinds are run.

Guidance on the creation and maintenance of an SCMP, based on the information produced by the planning activity, is available from a number of sources, such as [2*]. This reference provides requirements for the information to be contained in an SCMP; it also defines and describes six categories of SCM information to be included in an SCMP:

- Introduction (purpose, scope, terms used)
- SCM Management (organization, responsibilities, authorities, applicable policies, directives, and procedures)
- SCM Activities (configuration identification, configuration control, and so on)
- SCM Schedules (coordination with other project activities)
- SCM Resources (tools, physical resources, and human resources)
- SCMP Maintenance.

1.5. Surveillance of Software Configuration Management

[3*, c11s3]

After the SCM process has been implemented, some degree of surveillance may be necessary to ensure that the provisions of the SCMP are properly carried out. There are likely to be specific SQA requirements for ensuring compliance with specified SCM processes and procedures. The person responsible for SCM ensures that those with the assigned responsibility perform the defined SCM tasks correctly. The software quality assurance authority, as part of a compliance auditing activity, might also perform this surveillance.

The use of integrated SCM tools with process control capability can make the surveillance task easier. Some tools facilitate process compliance while providing flexibility for the software engineer to adapt procedures. Other tools enforce process, leaving the software engineer with less flexibility. Surveillance requirements and the level of flexibility to be provided to the software engineer are important considerations in tool selection.

1.5.1. SCM Measures and Measurement

[3*, c9s2, c25s2–s3]

SCM measures can be designed to provide specific information on the evolving product or to provide insight into the functioning of the SCM process. A related goal of monitoring the SCM process is to discover opportunities for process improvement. Measurements of SCM processes provide a good means for monitoring the effectiveness of SCM activities on an ongoing basis. These measurements are useful in characterizing the current state of the process as well as in providing a basis for making comparisons over time. Analysis of the measurements may produce

insights leading to process changes and corresponding updates to the SCMP.

Software libraries and the various SCM tool capabilities provide sources for extracting information about the characteristics of the SCM process (as well as providing project and management information). For example, information about the time required to accomplish various types of changes would be useful in an evaluation of the criteria for determining what levels of authority are optimal for authorizing certain types of changes and for estimating future changes.

Care must be taken to keep the focus of the surveillance on the insights that can be gained from the measurements, not on the measurements themselves. Discussion of software process and product measurement is presented in the Software Engineering Process KA. Software measurement programs are described in the Software Engineering Management KA.

1.5.2. In-Process Audits of SCM

[3*, c1s1]

Audits can be carried out during the software engineering process to investigate the current status of specific elements of the configuration or to assess the implementation of the SCM process. In-process auditing of SCM provides a more formal mechanism for monitoring selected aspects of the process and may be coordinated with the SQA function (see topic 5, Software Configuration Auditing).

2. Software Configuration Identification

[2*, c8] [4*, c29s1.1]

Software configuration identification identifies items to be controlled, establishes identification schemes for the items and their versions, and establishes the tools and techniques to be used in acquiring and managing controlled items. These activities provide the basis for the other SCM activities.

2.1. Identifying Items to Be Controlled

[2*, c8s2.2] [4*, c29s1.1]

One of the first steps in controlling change is identifying the software items to be controlled. This involves understanding the software configuration within the context of the system configuration, selecting software configuration items, developing a strategy for labeling software items and describing their relationships, and identifying both the baselines to be used and the procedure for a baseline's acquisition of the items.

2.1.1. Software Configuration

[1, c3]

Software configuration is the functional and physical characteristics of hardware or software as set forth in technical documentation or achieved in a product. It can be viewed as part of an overall system configuration.

2.1.2. Software Configuration Item

[4*, c29s1.1]

A configuration item (CI) is an item or aggregation of hardware or software or both that is designed to be managed as a single entity. A software configuration item (SCI) is a software entity that has been established as a configuration item [1]. The SCM typically controls a variety of items in addition to the code itself. Software items with potential to become SCIs include plans, specifications and design documentation, testing materials, software tools, source and executable code, code libraries, data and data dictionaries, and documentation for installation, maintenance, operations, and software use.

Selecting SCIs is an important process in which a balance must be achieved between providing adequate visibility for project control purposes and providing a manageable number of controlled items.

2.1.3. Software Configuration Item Relationships

[3*, c7s4]

Structural relationships among the selected SCIs, and their constituent parts, affect other SCM activities or tasks, such as software building or analyzing the impact of proposed changes. Proper tracking of these relationships is also important for supporting traceability. The design of the identification scheme for SCIs

Figure 6.2. Acquisition of Items

should consider the need to map identified items to the software structure, as well as the need to support the evolution of the software items and their relationships.

2.1.4. Software Version

[1, c3] [4*, c29s3]

Software items evolve as a software project proceeds. A version of a software item is an identified instance of an item. It can be thought of as a state of an evolving item. A variant is a version of a program resulting from the application of software diversity.

2.1.5. Baseline

[1, c3]

A software baseline is a formally approved version of a configuration item (regardless of media) that is formally designated and fixed at a specific time during the configuration item's life cycle. The term is also used to refer to a particular version of a software configuration item that has been agreed on. In either case, the baseline can only be changed through formal change control procedures. A baseline, together with all approved changes to the baseline, represents the current approved configuration.

Commonly used baselines include functional, allocated, developmental, and product baselines. The functional baseline corresponds to the reviewed system requirements. The allocated baseline corresponds to the reviewed software requirements specification and software interface requirements specification. The developmental baseline represents the evolving software configuration at selected times during the software life cycle. Change authority for this baseline typically rests primarily with the development organization but may be shared with other organizations (for example, SCM or Test). The product baseline corresponds to the completed software product delivered for system integration. The baselines to be used for a given project, along with the associated levels of authority needed for change approval, are typically identified in the SCMP.

2.1.6. Acquiring Software Configuration Items

[3*, c18]

Software configuration items are placed under SCM control at different times; that is, they are incorporated into a particular baseline at a particular point in the software life cycle. The triggering event is the completion of some form of formal acceptance task, such as a formal review. Figure 6.2 characterizes the growth of baselined items as the life cycle proceeds. This figure is based on the waterfall model for purposes of illustration only; the subscripts used in the figure indicate versions

of the evolving items. The software change request (SCR) is described in section 3.1.

In acquiring an SCI, its origin and initial integrity must be established. Following the acquisition of an SCI, changes to the item must be formally approved as appropriate for the SCI and the baseline involved, as defined in the SCMP. Following approval, the item is incorporated into the software baseline according to the appropriate procedure.

2.2. Software Library

[3*, c1s3] [4*, c29s1.2]

A software library is a controlled collection of software and related documentation designed to aid in software development, use, or maintenance [1]. It is also instrumental in software release management and delivery activities. Several types of libraries might be used, each corresponding to the software item's particular level of maturity. For example, a working library could support coding and a project support library could support testing, while a master library could be used for finished products. An appropriate level of SCM control (associated baseline and level of authority for change) is associated with each library. Security, in terms of access control and the backup facilities, is a key aspect of library management.

The tool(s) used for each library must support the SCM control needs for that library—both in terms of controlling SCIs and controlling access to the library. At the working library level, this is a code management capability serving developers, maintainers, and SCM. It is focused on managing the versions of software items while supporting the activities of multiple developers. At higher levels of control, access is more restricted and SCM is the primary user.

These libraries are also an important source of information for measurements of work and progress.

3. Software Configuration Control

[2*, c9] [4*, c29s2]

Software configuration control is concerned with managing changes during the software life cycle. It covers the process for determining what changes to make, the authority for approving certain changes, support for the implementation of those changes, and the concept of formal deviations from project requirements as well as waivers of them. Information derived from these activities is useful in measuring change traffic and breakage as well as aspects of rework.

3.1. Requesting, Evaluating, and Approving Software Changes

[2*, c9s2.4] [4*, c29s2]

The first step in managing changes to controlled items is determining what changes to make. The software change request process (see a typical flow of a change request process in Figure 6.3) provides formal procedures for submitting and recording change requests, evaluating the potential cost and impact of a proposed change, and accepting, modifying, deferring, or rejecting the proposed change. A change request (CR) is a request to expand or reduce the project scope; modify policies, processes, plans, or procedures; modify costs or budgets; or revise schedules [1]. Requests for changes to software configuration items may be originated by anyone at any point in the software life cycle and may include a suggested solution and requested priority. One source of a CR is the initiation of corrective action in response to problem reports. Regardless of the source, the type of change (for example, defect or enhancement) is usually recorded on the Software CR (SCR).

This provides an opportunity for tracking defects and collecting change activity measurements by change type. Once an SCR is received, a technical evaluation (also known as an impact analysis) is performed to determine the extent of the modifications that would be necessary should the change request be accepted. A good understanding of the relationships among software (and, possibly, hardware) items is important for this task. Finally, an established authority—commensurate with the affected baseline, the SCI involved, and the nature of the change—will evaluate the technical and managerial aspects of the change request and either accept, modify, reject, or defer the proposed change.

Figure 6.3. Flow of a Change Control Process

3.1.1. Software Configuration Control Board
[2*, c9s2.2] [3*, c11s1] [4*, c29s2]

The authority for accepting or rejecting proposed changes rests with an entity typically known as a Configuration Control Board (CCB). In smaller projects, this authority may actually reside with the leader or an assigned individual rather than a multiperson board. There can be multiple levels of change authority depending on a variety of criteria—such as the criticality of the item involved, the nature of the change (for example, impact on budget and schedule), or the project's current point in the life cycle. The composition of the CCBs used for a given system varies depending on these criteria (an SCM representative would always be present). All stakeholders, appropriate to the level of the CCB, are represented. When the scope of authority of a CCB is strictly software, it is known as a Software Configuration Control Board (SCCB). The activities of the CCB are typically subject to software quality audit or review.

3.1.2. Software Change Request Process
[3*, c1s4, c8s4]

An effective software change request (SCR) process requires the use of supporting tools and procedures for originating change requests, enforcing the flow of the change process, capturing CCB decisions, and reporting change process information. A link between this tool capability and the problem-reporting system can facilitate the tracking of solutions for reported problems.

3.2. Implementing Software Changes
[4*, c29]

Approved SCRs are implemented using the defined software procedures in accordance with the applicable schedule requirements. Since a number of approved SCRs might be implemented simultaneously, it is necessary to provide a means for tracking which SCRs are incorporated into particular software versions and baselines. As part of the closure of the change process, completed changes may undergo configuration audits and software quality verification—this includes ensuring that only approved changes have been made. The software change request process described above will typically document the SCM (and other) approval information for the change.

Changes may be supported by source code version control tools. These tools allow a team of software engineers, or a single software engineer, to track and document changes to the source code. These tools provide a single repository for storing the source code, can prevent more than one software engineer from editing the same module at the same time, and record all changes made to the

source code. Software engineers check modules out of the repository, make changes, document the changes, and then save the edited modules in the repository. If needed, changes can also be discarded, restoring a previous baseline. More powerful tools can support parallel development and geographically distributed environments. These tools may be manifested as separate, specialized applications under the control of an independent SCM group. They may also appear as an integrated part of the software engineering environment. Finally, they may be as elementary as a rudimentary change control system provided with an operating system.

3.3. Deviations and Waivers

[1, c3]

The constraints imposed on a software engineering effort or the specifications produced during the development activities might contain provisions that cannot be satisfied at the designated point in the life cycle. A deviation is a written authorization, granted prior to the manufacture of an item, to depart from a particular performance or design requirement for a specific number of units or a specific period of time. A waiver is a written authorization to accept a configuration item or other designated item that is found, during production or after having been submitted for inspection, to depart from specified requirements but is nevertheless considered suitable for use as-is or after rework by an approved method. In these cases, a formal process is used for gaining approval for deviations from, or waivers of, the provisions.

4. Software Configuration Status Accounting

[2*, c10]

Software configuration status accounting (SCSA) is an element of configuration management consisting of the recording and reporting of information needed to manage a configuration effectively.

4.1. Software Configuration Status Information

[2*, c10s2.1]

The SCSA activity designs and operates a system for the capture and reporting of necessary information as the life cycle proceeds. As in any information system, the configuration status information to be managed for the evolving configurations must be identified, collected, and maintained. Various information and measurements are needed to support the SCM process and to meet the configuration status reporting needs of management, software engineering, and other related activities. The types of information available include the approved configuration identification as well as the identification and current implementation status of changes, deviations, and waivers.

Some form of automated tool support is necessary to accomplish the SCSA data collection and reporting tasks; this could be a database capability, a stand-alone tool, or a capability of a larger, integrated tool environment.

4.2. Software Configuration Status Reporting

[2*, c10s2.4] [3*, c1s5, c9s1, c17]

Reported information can be used by various organizational and project elements—including the development team, the maintenance team, project management, and software quality activities. Reporting can take the form of ad hoc queries to answer specific questions or the periodic production of predesigned reports. Some information produced by the status accounting activity during the course of the life cycle might become quality assurance records.

In addition to reporting the current status of the configuration, the information obtained by the SCSA can serve as a basis of various measurements. Examples include the number of change requests per SCI and the average time needed to implement a change request.

5. Software Configuration Auditing

[2*, c11]

A software audit is an independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements, or other criteria [1]. Audits are conducted according to a well-defined process consisting of various auditor roles and responsibilities. Consequently, each audit must be carefully planned. An audit can require a number of individuals to perform a variety of tasks over a fairly short period of time. Tools to support

the planning and conduct of an audit can greatly facilitate the process.

Software configuration auditing determines the extent to which an item satisfies the required functional and physical characteristics. Informal audits of this type can be conducted at key points in the life cycle. Two types of formal audits might be required by the governing contract (for example, in contracts covering critical software): the Functional Configuration Audit (FCA) and the Physical Configuration Audit (PCA). Successful completion of these audits can be a prerequisite for the establishment of the product baseline.

5.1. Software Functional Configuration Audit

[2*, c11s2.1]

The purpose of the software FCA is to ensure that the audited software item is consistent with its governing specifications. The output of the software verification and validation activities (see Verification and Validation in the Software Quality KA) is a key input to this audit.

5.2. Software Physical Configuration Audit

[2*, c11s2.2]

The purpose of the software physical configuration audit (PCA) is to ensure that the design and reference documentation is consistent with the as-built software product.

5.3. In-Process Audits of a Software Baseline

[2*, c11s2.3]

As mentioned above, audits can be carried out during the development process to investigate the current status of specific elements of the configuration. In this case, an audit could be applied to sampled baseline items to ensure that performance is consistent with specifications or to ensure that evolving documentation continues to be consistent with the developing baseline item.

6. Software Release Management and Delivery

[2*, c14] [3*, c8s2]

In this context, *release* refers to the distribution of a software configuration item outside the development activity; this includes internal releases as well as distribution to customers. When different versions of a software item are available for delivery (such as versions for different platforms or versions with varying capabilities), it is frequently necessary to recreate specific versions and package the correct materials for delivery of the version. The software library is a key element in accomplishing release and delivery tasks.

6.1. Software Building

[4*, c29s4]

Software building is the activity of combining the correct versions of software configuration items, using the appropriate configuration data, into an executable program for delivery to a customer or other recipient, such as the testing activity. For systems with hardware or firmware, the executable program is delivered to the system-building activity. Build instructions ensure that the proper build steps are taken in the correct sequence. In addition to building software for new releases, it is usually also necessary for SCM to have the capability to reproduce previous releases for recovery, testing, maintenance, or additional release purposes.

Software is built using particular versions of supporting tools, such as compilers (see Compiler Basics in the Computing Foundations KA). It might be necessary to rebuild an exact copy of a previously built software configuration item. In this case, supporting tools and associated build instructions need to be under SCM control to ensure availability of the correct versions of the tools.

A tool capability is useful for selecting the correct versions of software items for a given target environment and for automating the process of building the software from the selected versions and appropriate configuration data. For projects with parallel or distributed development environments, this tool capability is necessary. Most software engineering environments provide this capability. These tools vary in complexity from requiring the software engineer to learn a specialized scripting language to graphics-oriented approaches that hide much of the complexity of an “intelligent” build facility.

The build process and products are often subject to software quality verification. Outputs of

the build process might be needed for future reference and may become quality assurance records.

6.2. Software Release Management

[4*, c29s3.2]

Software release management encompasses the identification, packaging, and delivery of the elements of a product—for example, an executable program, documentation, release notes, and configuration data. Given that product changes can occur on a continuing basis, one concern for release management is determining when to issue a release. The severity of the problems addressed by the release and measurements of the fault densities of prior releases affect this decision. The packaging task must identify which product items are to be delivered and then select the correct variants of those items, given the intended application of the product. The information documenting the physical contents of a release is known as a version description document. The release notes typically describe new capabilities, known problems, and platform requirements necessary for proper product operation. The package to be released also contains installation or upgrading instructions. The latter can be complicated by the fact that some current users might have versions that are several releases old. In some cases, release management might be required in order to track distribution of the product to various customers or target systems—for example, in a case where the supplier was required to notify a customer of newly reported problems. Finally, a mechanism to ensure the integrity of the released item can be implemented—for example by releasing a digital signature with it.

A tool capability is needed for supporting these release management functions. It is useful to have a connection with the tool capability supporting the change request process in order to map release contents to the SCRs that have been received. This tool capability might also maintain information on various target platforms and on various customer environments.

7. Software Configuration Management Tools

[3*, c26s1] [4*, c8s2]

When discussing software configuration management tools, it is helpful to classify them. SCM tools can be divided into three classes in terms of the scope at which they provide support: individual support, project-related support, and companywide-process support.

Individual support tools are appropriate and typically sufficient for small organizations or development groups without variants of their software products or other complex SCM requirements. They include:

- Version control tools: track, document, and store individual configuration items such as source code and external documentation.
- Build handling tools: in their simplest form, such tools compile and link an executable version of the software. More advanced building tools extract the latest version from the version control software, perform quality checks, run regression tests, and produce various forms of reports, among other tasks.
- Change control tools: mainly support the control of change requests and events notification (for example, change request status changes, milestones reached).

Project-related support tools mainly support workspace management for development teams and integrators; they are typically able to support distributed development environments. Such tools are appropriate for medium to large organizations with variants of their software products and parallel development but no certification requirements.

Companywide-process support tools can typically automate portions of a companywide process, providing support for workflow managements, roles, and responsibilities. They are able to handle many items, data, and life cycles. Such tools add to project-related support by supporting a more formal development process, including certification requirements.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	IEEE 828-2012 [2*]	Hass 2003 [3*]	Moore 2006 [5*]	Sommerville 2011 [4*]
1. Management of the SCM Process				
1.1. Organizational Context for SCM	c6, ann.D	introduction		c29
1.2. Constraints and Guidance for the SCM Process	c6, ann.D, ann.E	c2	c19s2.2	c29 intro
1.3. Planning for SCM	c6, ann.D, ann.E	c23		c29
1.3.1. SCM Organization and Responsibilities	ann.Ds5–6	c10–11		c29 intro
1.3.2. SCM Resources and Schedules	ann.Ds8	c23		
1.3.3. Tool Selection and Implementation		c26s2; s6		c29s5
1.3.4. Vendor/Subcontractor Control	c13	c13s9–c14s2		
1.3.5. Interface Control	c12	c24s4		
1.4. SCM Plan	ann.D	c23		c29s1
1.5. Surveillance of Software Configuration Management		c11s3		
1.5.1. SCM Measures and Measurement		c9s2; c25s2–s3		
1.5.2. In-Process Audits of SCM		c1s1		
2. Software Configuration Identification				c29s1.1
2.1. Identifying Items to Be Controlled	c8s2.2			c29s1.1
2.1.1. Software Configuration				
2.1.2. Software Configuration Item				c29s1.1
2.1.3. Software Configuration Item Relationships		c7s4		
2.1.4. Software Version				c29s3

	IEEE 828-2012 [2*]	Hass 2003 [3*]	Moore 2006 [5*]	Sommerville 2011 [4*]
2.1.5. Baseline				
2.1.6. Acquiring Software Configuration Items		c18		
2.2. Software Library		c1s3		c29s1.2
3. Software Configuration Control	c9			c29s2
3.1. Requesting, Evaluating, and Approving Software Changes	c9s2.4			c29s2
3.1.1. Software Configuration Control Board	c9s2.2	c11s1		c29s2
3.1.2. Software Change Request Process		c1s4, c8s4		
3.2. Implementing Software Changes				c29
3.3. Deviations and Waivers				
4. Software Configuration Status Accounting	c10			
4.1. Software Configuration Status Information	c10s2.1			
4.2. Software Configuration Status Reporting	c10s2.4	c1s5, c9s1, c17		
5. Software Configuration Auditing	c11			
5.1. Software Functional Configuration Audit	c11s2.1			
5.2. Software Physical Configuration Audit	c11s2.2			
5.3. In-Process Audits of a Software Baseline	c11s2.3			
6. Software Release Management and Delivery	c14	c8s2		c29s3
6.1. Software Building				c29s4
6.2. Software Release Management				c29s3.2
7. Software Configuration Management Tools		c26s1		

FURTHER READINGS

Stephen P. Berczuk and Brad Appleton, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration* [6].

This book expresses useful SCM practices and strategies as patterns. The patterns can be implemented using various tools, but they are expressed in a tool-agnostic fashion.

"CMMI for Development," Version 1.3, pp. 137–147 [7].

This model presents a collection of best practices to help software development organizations improve their processes. At maturity level 2, it suggests configuration management activities.

REFERENCES

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

[2*] *IEEE Std. 828-2012, Standard for Configuration Management in Systems and Software Engineering*, IEEE, 2012.

[3*] A.M.J. Hass, *Configuration Management Principles and Practices*, 1st ed., Addison-Wesley, 2003.

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

[5*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006.

[6] S.P. Berczuk and B. Appleton, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, Addison-Wesley Professional, 2003.

[7] CMMI Product Team, "CMMI for Development, Version 1.3," Software Engineering Institute, 2010; http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=9661.

CHAPTER 7

SOFTWARE ENGINEERING MANAGEMENT

ACRONYMS

PMBOK® Guide	*Guide to the Project Management Body of Knowledge*
SDLC	Software Development Life Cycle
SEM	Software Engineering Management
SQA	Software Quality Assurance
SWX	*Software Extension to the PMBOK® Guide*
WBS	Work Breakdown Structure

INTRODUCTION

Software engineering management can be defined as the application of management activities—planning, coordinating, measuring, monitoring, controlling, and reporting[1]—to ensure that software products and software engineering services are delivered efficiently, effectively, and to the benefit of stakeholders. The related discipline of management is an important element of all the knowledge areas (KAs), but it is of course more relevant to this KA than to other KAs. Measurement is also an important aspect of all KAs; the topic of measurement programs is presented in this KA.

In one sense, it should be possible to manage a software engineering project in the same way other complex endeavors are managed. However, there are aspects specific to software projects and software life cycle processes that complicate effective management, including these:

- Clients often don't know what is needed or what is feasible.
- Clients often lack appreciation for the complexities inherent in software engineering, particularly regarding the impact of changing requirements.
- It is likely that increased understanding and changing conditions will generate new or changed software requirements.
- As a result of changing requirements, software is often built using an iterative process rather than as a sequence of closed tasks.
- Software engineering necessarily incorporates creativity and discipline. Maintaining an appropriate balance between the two is sometimes difficult.
- The degree of novelty and complexity is often high.
- There is often a rapid rate of change in the underlying technology.

Software engineering management activities occur at three levels: organizational and infrastructure management, project management, and management of the measurement program. The last two are covered in detail in this KA description. However, this is not to diminish the importance of organizational and infrastructure management issues. It is generally agreed that software organizational engineering managers should be conversant with the project management and software measurement knowledge described in this KA. They should also possess some target domain knowledge. Likewise, it is also helpful if managers of complex projects and programs in which software is a component of the system architecture are aware of the differences that software processes introduce into project management and project measurement.

1 The terms Initiating, Planning, Executing, Monitoring and Controlling, and Closing are used to describe process groups in the *PMBOK® Guide* and *SWX*.

Figure 7.1. Breakdown of Topics for the Software Engineering Management KA

Other aspects of organizational management exert an impact on software engineering (for example, organizational policies and procedures that provide the framework in which software engineering projects are undertaken). These policies and procedures may need to be adjusted by the requirements for effective software development and maintenance. In addition, a number of policies specific to software engineering may need to be in place or established for effective management of software engineering at the organizational level. For example, policies are usually necessary to establish specific organization-wide processes or procedures for software engineering tasks such as software design, software construction, estimating, monitoring, and reporting. Such policies are important for effective long-term management of software engineering projects across an organization (for example, establishing a consistent basis by which to analyze past project performance and implement improvements).

Another important aspect of organizational management is personnel management policies and procedures for hiring, training, and mentoring personnel for career development, not only at the project level, but also to the longer-term success of an organization. Software engineering personnel may present unique training or personnel management challenges (for example, maintaining currency in a context where the underlying technology undergoes rapid and continuous change).

Communication management is also often mentioned as an overlooked but important aspect of the performance of individuals in a field where precise understanding of user needs, software requirements, and software designs is necessary. Furthermore, portfolio management, which provides an overall view, not only of software currently under development in various projects and programs (integrated projects), but also of software planned and currently in use in an organization, is desirable. Also, software reuse is a key

factor in maintaining and improving productivity and competitiveness. Effective reuse requires a strategic vision that reflects the advantages and disadvantages of reuse.

In addition to understanding the aspects of management that are uniquely influenced by software projects, software engineers should have some knowledge of the more general aspects of management that are discussed in this KA (even in the first few years after graduation).

Attributes of organizational culture and behavior, plus management of other functional areas of the enterprise, have an influence, albeit indirectly, on an organization's software engineering processes.

Extensive information concerning software project management can be found in the *Guide to the Project Management Body of Knowledge (PMBOK® Guide)* and the *Software Extension to the PMBOK® Guide* (*SWX*) [1] [2]. Each of these guides includes ten project management KAs: project integration management, project scope management, project time management, project cost management, project quality management, project human resource management, project communications management, project risk management, project procurement management, and project stakeholder management. Each KA has direct relevance to this Software Engineering Management KA.

Additional information is also provided in the other references and further readings for this KA.

This Software Engineering Management KA consists of the software project management processes in the first five topics in Figure 7.1 (Initiation and Scope Definition, Software Project Planning, Software Project Enactment, Review and Evaluation, Closure), plus Software Engineering Measurement in the sixth topic and Software Engineering Management Tools in the seventh topic. While project management and measurement management are often regarded as being separate, and indeed each does possess many unique attributes, the close relationship has led to combined treatment in this KA.

Unfortunately, a common perception of the software industry is that software products are delivered late, over budget, of poor quality, and with incomplete functionality. Measurement-informed management—a basic principle of any true engineering discipline (see Measurement in the Engineering Foundations KA)—can help improve the perception and the reality. In essence, management without measurement (qualitative and quantitative) suggests a lack of discipline, and measurement without management suggests a lack of purpose or context. Effective management requires a combination of both measurement and experience.

The following working definitions are adopted here:

- *Management* is a system of processes and controls required to achieve the strategic objectives set by the organization.
- *Measurement* refers to the assignment of values and labels to software engineering work products, processes, and resources plus the models that are derived from them, whether these models are developed using statistical or other techniques [3* , c7, c8].

The software engineering project management sections in this KA make extensive use of the software engineering measurement section.

This KA is closely related to others in the *SWEBOK Guide*, and reading the following KA descriptions in conjunction with this one will be particularly helpful:

- The Engineering Foundations KA describes some general concepts of measurement that are directly applicable to the Software Engineering Measurement section of this KA. In addition, the concepts and techniques presented in the Statistical Analysis section of the Engineering Foundations KA apply directly to many topics in this KA.
- The Software Requirements KA describes some of the activities that should be performed during the Initiation and Scope definition phase of the project.
- The Software Configuration Management KA deals with identification, control, status accounting, and auditing of software configurations along with software release management and delivery and software configuration management tools.

- The Software Engineering Process KA describes software life cycle models and the relationships between processes and work products.
- The Software Quality KA emphasizes quality as a goal of management and as an aim of many software engineering activities.
- The Software Engineering Economics KA discusses how to make software-related decisions in a business context.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING MANAGEMENT

Because most software development life cycle models require similar activities that may be executed in different ways, the breakdown of topics is activity-based. That breakdown is shown in Figure 7.1. The elements of the top-level breakdown shown in that figure are the activities that are usually performed when a software development project is being managed, independent of the software development life cycle model (see Software Life Cycle Models in the Software Engineering Process KA) that has been chosen for a specific project. There is no intent in this breakdown to recommend a specific life cycle model. The breakdown implies only what happens and does not imply when, how, or how many times each activity occurs. The seven topics are:

- Initiation and Scope Definition, which deal with the decision to embark on a software engineering project;
- Software Project Planning, which addresses the activities undertaken to prepare for a successful software engineering project from the management perspective;
- Software Project Enactment, which deals with generally accepted software engineering management activities that occur during the execution of a software engineering project;
- Review and Evaluation, which deal with ensuring that technical, schedule, cost, and quality engineering activities are satisfactory;
- Closure, which addresses the activities accomplished to complete a project;
- Software Engineering Measurement, which deals with the effective development and implementation of measurement programs in software engineering organizations;
- Software Engineering Management Tools, which describes the selection and use of tools for managing a software engineering project.

1. Initiation and Scope Definition

The focus of these activities is on effective determination of software requirements using various elicitation methods and the assessment of project feasibility from a variety of standpoints. Once project feasibility has been established, the remaining tasks within this section are the specification of requirements and selection of the processes for revision and review of requirements.

1.1. Determination and Negotiation of Requirements

[3*, c3]

Determining and negotiating requirements set the visible boundaries for the set of tasks being undertaken (see the Software Requirements KA). Activities include requirements elicitation, analysis, specification, and validation. Methods and techniques should be selected and applied, taking into account the various stakeholder perspectives. This leads to the determination of project scope in order to meet objectives and satisfy constraints.

1.2. Feasibility Analysis

[4*, c4]

The purpose of feasibility analysis is to develop a clear description of project objectives and evaluate alternative approaches in order to determine whether the proposed project is the best alternative given the constraints of technology, resources, finances, and social/political considerations. An initial project and product scope statement, project deliverables, project duration constraints, and an estimate of resources needed should be prepared.

Resources include a sufficient number of people who have the needed skills, facilities, infrastructure, and support (either internally or externally). Feasibility analysis often requires approximate estimations of effort and cost based on appropriate methods (see section 2.3, Effort, Schedule, and Cost Estimation).

1.3. Process for the Review and Revision of Requirements

[3*, c3]

Given the inevitability of change, stakeholders should agree on the means by which requirements and scope are to be reviewed and revised (for example, change management procedures, iterative cycle retrospectives). This clearly implies that scope and requirements will not be "set in stone" but can and should be revisited at predetermined points as the project unfolds (for example, at the time when backlog priorities are created or at milestone reviews). If changes are accepted, then some form of traceability analysis and risk analysis should be used to ascertain the impact of those changes (see section 2.5, Risk Management, and Software Configuration Control in the Software Configuration Management KA).

A managed-change approach can also form the basis for evaluation of success during closure of an incremental cycle or an entire project, based on changes that have occurred along the way (see topic 5, Closure).

2. Software Project Planning

The first step in software project planning should be selection of an appropriate software development life cycle model and perhaps tailoring it based on project scope, software requirements, and a risk assessment. Other factors to be considered include the nature of the application domain, functional and technical complexity, and software quality requirements (see Software Quality Requirements in the Software Quality KA).

In all SDLCs, risk assessment should be an element of initial project planning, and the "risk profile" of the project should be discussed and accepted by all relevant stakeholders. Software quality management processes (see Software Quality Management Processes in the Software Quality KA) should be determined as part of the planning process and result in procedures and responsibilities for software quality assurance, verification and validation, reviews, and audits (see the Software Quality KA). Processes and responsibilities for ongoing review and revision of the project plan and related plans should also be clearly stated and agreed upon.

2.1. Process Planning

[3*, c3, c4, c5] [5*, c1]

Software development life cycle (SDLC) models span a continuum from predictive to adaptive (see Software Life Cycle Models in the Software Engineering Process KA). Predictive SDLCs are characterized by development of detailed software requirements, detailed project planning, and minimal planning for iteration among development phases. Adaptive SDLCs are designed to accommodate emergent software requirements and iterative adjustment of plans. A highly predictive SDLC executes the first five processes listed in Figure 7.1 in a linear sequence with revisions to earlier phases only as necessary. Adaptive SDLCs are characterized by iterative development cycles. SDLCs in the mid-range of the SDLC continuum produce increments of functionality on either a preplanned schedule (on the predictive side of the continuum) or as the products of frequently updated development cycles (on the adaptive side of the continuum).

Well-known SDLCs include the waterfall, incremental, and spiral models plus various forms of agile software development [2] [3*, c2].

Relevant methods (see the Software Engineering Models and Methods KA) and tools should be selected as part of planning. Automated tools that will be used throughout the project should also be planned for and acquired. Tools may include tools for project scheduling, software requirements, software design, software construction, software maintenance, software configuration management, software engineering process, software quality, and others. While many of these tools should be selected based primarily on the technical considerations discussed in other KAs, some of them are closely related to the management considerations discussed in this chapter.

2.2. Determine Deliverables

[3*, c4, c5, c6]

The work product(s) of each project activity (for example, software architecture design documents, inspection reports, tested software) should be identified and characterized. Opportunities to reuse software components from previous projects or to utilize off-the-shelf software products

should be evaluated. Procurement of software and use of third parties to develop deliverables should be planned and suppliers selected (see section 3.2, Software Acquisition and Supplier Contract Management).

2.3. Effort, Schedule, and Cost Estimation

[3*, c6]

The estimated range of effort required for a project, or parts of a project, can be determined using a calibrated estimation model based on historical size and effort data (when available) and other relevant methods such as expert judgment and analogy. Task dependencies can be established and potential opportunities for completing tasks concurrently and sequentially can be identified and documented using a Gantt chart, for example. For predictive SDLC projects, the expected schedule of tasks with projected start times, durations, and end times is typically produced during planning. For adaptive SDLC projects, an overall estimate of effort and schedule is typically developed from the initial understanding of the requirements, or, alternatively, constraints on overall effort and schedule may be specified and used to determine an initial estimate of the number of iterative cycles and estimates of effort and other resources allocated to each cycle.

Resource requirements (for example, people and tools) can be translated into cost estimates. Initial estimation of effort, schedule, and cost is an iterative activity that should be negotiated and revised among affected stakeholders until consensus is reached on resources and time available for project completion.

2.4. Resource Allocation

[3*, c5, c10, c11]

Equipment, facilities, and people should be allocated to the identified tasks, including the allocation of responsibilities for completion of various elements of a project and the overall project. A matrix that shows who is responsible for, accountable for, consulted about, and informed about each of the tasks can be used. Resource allocation is based on, and constrained by, the availability of resources and their optimal use, as well as by issues relating to personnel (for example, productivity of individuals and teams, team dynamics, and team structures).

2.5. Risk Management

[3*, c9] [5*, c5]

Risk and uncertainty are related but distinct concepts. Uncertainty results from lack of information. Risk is characterized by the probability of an event that will result in a negative impact plus a characterization of the negative impact on a project. Risk is often the result of uncertainty. The converse of risk is opportunity, which is characterized by the probability that an event having a positive outcome might occur.

Risk management entails identification of risk factors and analysis of the probability and potential impact of each risk factor, prioritization of risk factors, and development of risk mitigation strategies to reduce the probability and minimize the negative impact if a risk factor becomes a problem. Risk assessment methods (for example, expert judgment, historical data, decision trees, and process simulations) can sometimes be used in order to identify and evaluate risk factors.

Project abandonment conditions can also be determined at this point in discussion with all relevant stakeholders. Software-unique aspects of risk, such as software engineers' tendency to add unneeded features, or the risks related to software's intangible nature, can influence risk management of a software project. Particular attention should be paid to the management of risks related to software quality requirements such as safety or security (see the Software Quality KA). Risk management should be done not only at the beginning of a project, but also at periodic intervals throughout the project life cycle.

2.6. Quality Management

[3*, c4] [4*, c24]

Software quality requirements should be identified, perhaps in both quantitative and qualitative terms, for a software project and the associated work products. Thresholds for acceptable quality measurements should be set for each software quality requirement based on stakeholder needs

and expectations. Procedures concerned with ongoing Software Quality Assurance (SQA) and quality improvement throughout the development process, and for verification and validation of the deliverable software product, should also be specified during quality planning (for example, technical reviews and inspections or demonstrations of completed functionality; see the Software Quality KA).

2.7. Plan Management

[3*, c4]

For software projects, where change is an expectation, plans should be managed. Managing the project plan should thus be planned. Plans and processes selected for software development should be systematically monitored, reviewed, reported, and, when appropriate, revised. Plans associated with supporting processes (for example, documentation, software configuration management, and problem resolution) also should be managed. Reporting, monitoring, and controlling a project should fit within the selected SDLC and the realities of the project; plans should account for the various artifacts that will be used to manage the project.

3. Software Project Enactment

During software project enactment (also known as project execution) plans are implemented and the processes embodied in the plans are enacted. Throughout, there should be a focus on adherence to the selected SDLC processes, with an overriding expectation that adherence will lead to the successful satisfaction of stakeholder requirements and achievement of the project's objectives. Fundamental to enactment are the ongoing management activities of monitoring, controlling, and reporting.

3.1. Implementation of Plans

[4*, c2]

Project activities should be undertaken in accordance with the project plan and supporting plans. Resources (for example, personnel, technology, and funding) are utilized and work products (for example, software design, software code, and software test cases) are generated.

3.2. Software Acquisition and Supplier Contract Management

[3*, c3, c4]

Software acquisition and supplier contract management is concerned with issues involved in contracting with customers of the software development organization who acquire the deliverable work products and with suppliers who supply products or services to the software engineering organization.

This may involve selection of appropriate kinds of contracts, such as fixed price, time and materials, cost plus fixed fee, or cost plus incentive fee. Agreements with customers and suppliers typically specify the scope of work and the deliverables and include clauses such as penalties for late delivery or nondelivery and intellectual property agreements that specify what the supplier or suppliers are providing and what the acquirer is paying for, plus what will be delivered to and owned by the acquirer. For software being developed by suppliers (both internal to or external to the software development organization), agreements commonly indicate software quality requirements for acceptance of the delivered software.

After the agreement has been put in place, execution of the project in compliance with the terms of the agreement should be managed (see chapter 12 of SWX, Software Procurement Management, for more information on this topic [2]).

3.3. Implementation of Measurement Process

[3*, c7]

The measurement process should be enacted during the software project to ensure that relevant and useful data are collected (see sections 6.2, Plan the Measurement Process, and 6.3, Perform the Measurement Process).

3.4. Monitor Process

[3*, c8]

Adherence to the project plan and related plans should be assessed continually and at

predetermined intervals. Also, outputs and completion criteria for each task should be assessed. Deliverables should be evaluated in terms of their required characteristics (for example, via inspections or by demonstrating working functionality). Effort expenditure, schedule adherence, and costs to date should be analyzed, and resource usage examined. The project risk profile (see section 2.5, Risk Management) should be revisited, and adherence to software quality requirements evaluated (see Software Quality Requirements in the Software Quality KA).

Measurement data should be analyzed (see Statistical Analysis in the Engineering Foundations KA). Variance analysis based on the deviation of actual from expected outcomes and values should be determined. This may include cost overruns, schedule slippage, or other similar measures. Outlier identification and analysis of quality and other measurement data should be performed (for example, defect analysis; see Software Quality Measurement in the Software Quality KA). Risk exposures should be recalculated (see section 2.5, Risk Management). These activities can enable problem detection and exception identification based on thresholds that have been exceeded. Outcomes should be reported when thresholds have been exceeded, or as necessary.

3.5. Control Process

[3*, c7, c8]

The outcomes of project monitoring activities provide the basis on which decisions can be made. Where appropriate, and when the probability and impact of risk factors are understood, changes can be made to the project. This may take the form of corrective action (for example, retesting certain software components); it may involve incorporating additional actions (for example, deciding to use prototyping to assist in software requirements validation; see Prototyping in the Software Requirements KA); and/or it may entail revision of the project plan and other project documents (for example, the software requirements specification) to accommodate unanticipated events and their implications.

In some instances, the control process may lead to abandonment of the project. In all cases, software configuration control and software configuration management procedures should be adhered to (see the Software Configuration Management KA), decisions should be documented and communicated to all relevant parties, plans should be revisited and revised when necessary, and relevant data recorded (see section 6.3, Perform the Measurement Process).

3.6. Reporting

[3*, c11]

At specified and agreed-upon times, progress to date should be reported—both within the organization (for example, to a project steering committee) and to external stakeholders (for example, clients or users). Reports should focus on the information needs of the target audience as opposed to the detailed status reporting within the project team.

4. Review and Evaluation

At prespecified times and as needed, overall progress towards achievement of the stated objectives and satisfaction of stakeholder (user and customer) requirements should be evaluated. Similarly, assessments of the effectiveness of the software process, the personnel involved, and the tools and methods employed should also be undertaken regularly and as determined by circumstances.

4.1. Determining Satisfaction of Requirements

[4*, c8]

Because achieving stakeholder satisfaction is a principal goal of the software engineering manager, progress towards this goal should be assessed periodically. Progress should be assessed on achievement of major project milestones (for example, completion of software design architecture or completion of a software technical review), or upon completion of an iterative development cycle that results in a product increment. Variances from software requirements should be identified and appropriate actions should be taken.

As in the control process activity above (see section 3.5, Control Process), software configuration

control and software configuration management procedures should be followed (see the Software Configuration Management KA), decisions documented and communicated to all relevant parties, plans revisited and revised where necessary, and relevant data recorded (see section 6.3, Perform the Measurement Process).

4.2. Reviewing and Evaluating Performance

[3*, c8, c10]

Periodic performance reviews for project personnel can provide insights as to the likelihood of adherence to plans and processes as well as possible areas of difficulty (for example, team member conflicts). The various methods, tools, and techniques employed should be evaluated for their effectiveness and appropriateness, and the process being used by the project should also be systematically and periodically assessed for relevance, utility, and efficacy in the project context. Where appropriate, changes should be made and managed.

5. Closure

An entire project, a major phase of a project, or an iterative development cycle reaches closure when all the plans and processes have been enacted and completed. The criteria for project, phase, or iteration success should be evaluated. Once closure is established, archival, retrospective, and process improvement activities can be performed.

5.1. Determining Closure

[1, s3.7, s4.6]

Closure occurs when the specified tasks for a project, a phase, or an iteration have been completed and satisfactory achievement of the completion criteria has been confirmed. Software requirements can be confirmed as satisfied or not, and the degree of achieving the objectives can be determined. Closure processes should involve relevant stakeholders and result in documentation of relevant stakeholders' acceptance; any known problems should be documented.

5.2. Closure Activities

[2, s3.7, s4.8]

After closure has been confirmed, archiving of project materials should be accomplished in accordance with stakeholder agreed-upon methods, location, and duration—possibly including destruction of sensitive information, software, and the medium on which copies are resident. The organization's measurement database should be updated with relevant project data. A project, phase, or iteration retrospective analysis should be undertaken so that issues, problems, risks, and opportunities encountered can be analyzed (see topic 4, Review and Evaluation). Lessons learned should be drawn from the project and fed into organizational learning and improvement endeavors.

6. Software Engineering Measurement

The importance of measurement and its role in better management and engineering practices is widely acknowledged (see Measurement in the Engineering Foundations KA). Effective measurement has become one of the cornerstones of organizational maturity. Measurement can be applied to organizations, projects, processes, and work products. In this section the focus is on the application of measurement at the levels of projects, processes, and work products.

This section follows the IEEE 15939:2008 standard [6], which describes a process to define the activities and tasks necessary to implement a software measurement process. The standard also includes a measurement information model.

6.1. Establish and Sustain Measurement Commitment

[7*, c1, c2][2]

- Requirements for measurement. Each measurement endeavor should be guided by organizational objectives and driven by a set of measurement requirements established by

2 Please note that these two chapters can be downloaded free of charge from www.psmsc.com/PSMBook.asp.

the organization and the project (for example, an organizational objective might be "first-to-market with new products").

- Scope of measurement. The organizational unit to which each measurement requirement is to be applied should be established. This may consist of a functional area, a single project, a single site, or an entire enterprise. The temporal scope of the measurement effort should also be considered because time series of some measurements may be required; for example, to calibrate estimation models (see section 2.3, Effort, Schedule, and Cost Estimation).
- Team commitment to measurement. The commitment should be formally established, communicated, and supported by resources (see next item).
- Resources for measurement. An organization's commitment to measurement is an essential factor for success, as evidenced by the assignment of resources for implementing the measurement process. Assigning resources includes allocation of responsibility for the various tasks of the measurement process (such as analyst and librarian). Adequate funding, training, tools, and support to conduct the process should also be allocated.

6.2. Plan the Measurement Process

[7*, c1, c2]

- Characterize the organizational unit. The organizational unit provides the context for measurement, so the organizational context should be made explicit, including the constraints that the organization imposes on the measurement process. The characterization can be stated in terms of organizational processes, application domains, technology, organizational interfaces, and organizational structure.
- Identify information needs. Information needs are based on the goals, constraints, risks, and problems of the organizational unit. They may be derived from business, organizational, regulatory, and/or product objectives. They should be identified and prioritized. Then a subset of objectives to be addressed can be selected, documented, communicated, and reviewed by stakeholders.
- Select measures. Candidate measures should be selected, with clear links to the information needs. Measures should be selected based on the priorities of the information needs and other criteria such as cost of collection, degree of process disruption during collection, ease of obtaining accurate, consistent data, and ease of analysis and reporting. Because internal quality characteristics (see Models and Quality Characteristics in the Software Quality KA) are often not contained in the contractually binding software requirements, it is important to consider measuring the internal quality of the software to provide an early indicator of potential issues that may impact external stakeholders.
- Define data collection, analysis, and reporting procedures. This encompasses collection procedures and schedules, storage, verification, analysis, reporting, and configuration management of data.
- Select criteria for evaluating the information products. Criteria for evaluation are influenced by the technical and business objectives of the organizational unit. Information products include those associated with the product being produced, as well as those associated with the processes being used to manage and measure the project.
- Provide resources for measurement tasks. The measurement plan should be reviewed and approved by the appropriate stakeholders to include all data collection procedures; storage, analysis, and reporting procedures; evaluation criteria; schedules; and responsibilities. Criteria for reviewing these artifacts should have been established at the organizational-unit level or higher and should be used as the basis for these reviews. Such criteria should take into consideration previous experience, availability of resources, and potential disruptions to projects when changes from current practices are proposed. Approval demonstrates commitment to the measurement process.
- Identify resources to be made available for implementing the planned and approved

measurement tasks. Resource availability may be staged in cases where changes are to be piloted before widespread deployment. Consideration should be paid to the resources necessary for successful deployment of new procedures or measures.

- Acquire and deploy supporting technologies. This includes evaluation of available supporting technologies, selection of the most appropriate technologies, acquisition of those technologies, and deployment of those technologies.

6.3. Perform the Measurement Process

[7*, c1, c2]

- Integrate measurement procedures with relevant software processes. The measurement procedures, such as data collection, should be integrated into the software processes they are measuring. This may involve changing current software processes to accommodate data collection or generation activities. It may also involve analysis of current software processes to minimize additional effort and evaluation of the effect on employees to ensure that the measurement procedures will be accepted. Morale issues and other human factors should be considered. In addition, the measurement procedures should be communicated to those providing the data. Training and support may also need to be provided. Data analysis and reporting procedures are typically integrated into organizational and/or project processes in a similar manner.
- Collect data. Data should be collected, verified, and stored. Collection can sometimes be automated by using software engineering management tools (see topic 7, Software Engineering Management Tools) to analyze data and develop reports. Data may be aggregated, transformed, or recoded as part of the analysis process, using a degree of rigor appropriate to the nature of the data and the information needs. The results of this analysis are typically indicators such as graphs, numbers, or other indications that will be interpreted, resulting in conclusions and recommendations to be presented to stakeholders (see Statistical Analysis in the Engineering Foundations KA). The results and conclusions are usually reviewed, using a process defined by the organization (which may be formal or informal). Data providers and measurement users should participate in reviewing the data to ensure that they are meaningful and accurate and that they can result in reasonable actions.
- Communicate results. Information products should be documented and communicated to users and stakeholders.

6.4. Evaluate Measurement

[7*, c1, c2]

- Evaluate information products and the measurement process against specified evaluation criteria and determine strengths and weaknesses of the information products or process, respectively. Evaluation may be performed by an internal process or an external audit; it should include feedback from measurement users. Lessons learned should be recorded in an appropriate database.
- Identify potential improvements. Such improvements may be changes in the format of indicators, changes in units measured, or reclassification of measurement categories. The costs and benefits of potential improvements should be determined and appropriate improvement actions should be reported.
- Communicate proposed improvements to the measurement process owner and stakeholders for review and approval. Also, lack of potential improvements should be communicated if the analysis fails to identify any improvements.

7. Software Engineering Management Tools

[3*, c5, c6, c7]

Software engineering management tools are often used to provide visibility and control of software engineering management processes. Some tools are automated while others are manually implemented. There has been a recent trend towards the use of integrated suites of software engineering tools that are used throughout a project to plan, collect and record, monitor and control, and

report project and product information. Tools can be divided into the following categories:

Project Planning and Tracking Tools. Project planning and tracking tools can be used to estimate project effort and cost and to prepare project schedules. Some projects use automated estimation tools that accept as input the estimated size and other characteristics of a software product and produce estimates of the required total effort, schedule, and cost. Planning tools also include automated scheduling tools that analyze the tasks within a work breakdown structure, their estimated durations, their precedence relationships, and the resources assigned to each task to produce a schedule in the form of a Gantt chart.

Tracking tools can be used to track project milestones, regularly scheduled project status meetings, scheduled iteration cycles, product demonstrations, and/or action items.

Risk Management Tools. Risk management tools (see section 2.5, Risk Management) can be used to track risk identification, estimation, and monitoring. These tools include the use of approaches such as simulation or decision trees to analyze the effect of costs versus payoffs and subjective estimates of the probabilities of risk events. Monte Carlo simulation tools can be used to produce probability distributions of effort, schedule, and risk by combining multiple input probability distributions in an algorithmic manner.

Communications Tools. Communication tools can assist in providing timely and consistent information to relevant stakeholders involved in a project. These tools can include things like email notifications and broadcasts to team members and stakeholders. They also include communication of minutes from regularly scheduled project meetings, daily stand-up meetings, plus charts showing progress, backlogs, and maintenance request resolutions.

Measurement Tools. Measurement tools support activities related to the software measurement program (see topic 6, Software Engineering Measurement). There are few completely automated tools in this category. Measurement tools used to gather, analyze, and report project measurement data may be based on spreadsheets developed by project team members or organizational employees.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Fairley 2009 [3*]	Sommerville 2011 [4*]	Boehm and Turner 2003 [5*]	McGarry et al. 2001 [7*]
1. Initiation and Scope Definition				
1.1. Determination and Negotiation of Requirements	c3			
1.2. Feasibility Analysis		c4		
1.3. Process for the Review and Revision of Requirements	c3			
2. Software Project Planning				
2.1. Process Planning	c2, c3, c4, c5		c1	
2.2. Determine Deliverables	c4, c5, c6			
2.3. Effort, Schedule, and Cost Estimation	c6			
2.4. Resource Allocation	c5, c10, c11			
2.5. Risk Management	c9		c5	
2.6. Quality Management	c4	c24		
2.7. Plan Management	c4			
3. Software Project Enactment				
3.1. Implementation of Plans		c2		
3.2. Software Acquisition and Supplier Contract Management	c3, c4			
3.3. Implementation of Measurement Process	c7			
3.4. Monitor Process	c8			
3.5. Control Process	c7, c8			
3.6. Reporting	c11			
4. Review and Evaluation				
4.1. Determining Satisfaction of Requirements				
4.2. Reviewing and Evaluating Performance	c8, c10			

	Fairley 2009 [3*]	Sommerville 2011 [4*]	Boehm and Turner 2003 [5*]	McGarry et al. 2001 [7*]
5. Closure				
5.1. Determining Closure				
5.2. Closure Activities				
6. Software Engineering Measurement				
6.1. Establish and Sustain Measurement Commitment				c1, c2
6.2. Plan the Measurement Process				c1, c2
6.3. Perform the Measurement Process				c1, c2
6.4. Evaluate Measurement				c1, c2
7. Software Engineering Management Tools	c5, c6, c7			

FURTHER READINGS

A Guide to the Project Management Body of Knowledge (PMBOK® Guide) [1].

The *PMBOK® Guide* provides guidelines for managing individual projects and defines project management-related concepts. It also describes the project management life cycle and its related processes, as well as the project life cycle. It is a globally recognized guide for the project management profession.

Software Extension to the Guide to the Project Management Body of Knowledge (PMBOK® Guide) [2].

SWX provides adaptations and extensions to the generic practices of project management documented in the *PMBOK® Guide* for managing software projects. The primary contribution of this extension to the *PMBOK® Guide* is a description of processes that are applicable for managing adaptive life cycle software projects.

IEEE Standard Adoption of ISO/IEC 15939 [6].

This international standard identifies a process that supports defining a suitable set of measures to address specific information needs. It identifies the activities and tasks that are necessary to successfully identify, define, select, apply, and improve measurement within an overall project or organizational measurement structure.

J. McDonald, *Managing the Development of Software Intensive Systems*, Wiley, 2010 [8].

This textbook provides an introduction to project management for beginning software and hardware developers plus unique advanced material for experienced project managers. Case studies are included for planning and managing verification and validation for large software projects, complex software, and hardware systems, as well as inspection results and testing metrics to monitor project status.

REFERENCES

[1] Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK(R) Guide)*, 5th ed., Project Management Institute, 2013.

[2] Project Management Institute and IEEE Computer Society, *Software Extension to the PMBOK® Guide Fifth Edition*, Project Management Institute, 2013.

[3*] R.E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

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

[5*] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2003.

[6] *IEEE Std. 15939-2008 Standard Adoption of ISO/IEC 15939:2007 Systems and Software Engineering—Measurement Process*, IEEE, 2008.

[7*] J. McGarry et al., *Practical Software Measurement: Objective Information for Decision Makers*, Addison-Wesley Professional, 2001.

[8] J. McDonald, *Managing the Development of Software Intensive Systems*, John Wiley and Sons, Inc., 2010.

CHAPTER 8

SOFTWARE ENGINEERING PROCESS

ACRONYMS

BPMN	Business Process Modeling Notation
CASE	Computer-Assisted Software Engineering
CM	Configuration Management
CMMI	Capability Maturity Model Integration
GQM	Goal-Question-Metric
IDEF0	Integration Definition
LOE	Level of Effort
ODC	Orthogonal Defect Classification
SDLC	Software Development Life Cycle
SPLC	Software Product Life Cycle
UML	Unified Modeling Language

INTRODUCTION

An engineering process consists of a set of interrelated activities that transform one or more inputs into outputs while consuming resources to accomplish the transformation. Many of the processes of traditional engineering disciplines (e.g., electrical, mechanical, civil, chemical) are concerned with transforming energy and physical entities from one form into another, as in a hydroelectric dam that transforms potential energy into electrical energy or a petroleum refinery that uses chemical processes to transform crude oil into gasoline.

In this knowledge area (KA), software engineering processes are concerned with work activities accomplished by software engineers to develop, maintain, and operate software, such as requirements, design, construction, testing, configuration management, and other software engineering processes. For readability, "software engineering process" will be referred to as "software process" in this KA. In addition, please note that "software process" denotes work activities—not the execution process for implemented software.

Software processes are specified for a number of reasons: to facilitate human understanding, communication, and coordination; to aid management of software projects; to measure and improve the quality of software products in an efficient manner; to support process improvement; and to provide a basis for automated support of process execution.

SWEBOK KAs closely related to this Software Engineering Process KA include Software Engineering Management, Software Engineering Models and Methods, and Software Quality; the Measurement and Root Cause Analysis topic found in the Engineering Foundations KA is also closely related. Software Engineering Management is concerned with tailoring, adapting, and implementing software processes for a specific software project (see Process Planning in the Software Engineering Management KA). Models and methods support a systematic approach to software development and modification.

The Software Quality KA is concerned with the planning, assurance, and control processes for project and product quality. Measurement and measurement results in the Engineering Foundations KA are essential for evaluating and controlling software processes.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING PROCESS

As illustrated in Figure 8.1, this KA is concerned with software process definition, software life cycles, software process assessment and improvement, software measurement, and software engineering process tools.

Figure 8.1. Breakdown of Topics for the Software Engineering Process KA

1. Software Process Definition

[1*, p177] [2*, p295] [3*, p28–29, p36, c5]

This topic is concerned with a definition of software process, software process management, and software process infrastructure.

As stated above, a software process is a set of interrelated activities and tasks that transform input work products into output work products. At minimum, the description of a software process includes required inputs, transforming work activities, and outputs generated. As illustrated in Figure 8.2, a software process may also include its entry and exit criteria and decomposition of the work activities into tasks, which are the smallest units of work subject to management accountability. A process input may be a triggering event or the output of another process. Entry criteria should be satisfied before a process can commence. All specified conditions should be satisfied before a process can be successfully concluded, including the acceptance criteria for the output work product or work products.

A software process may include subprocesses. For example, software requirements validation is a process used to determine whether the requirements will provide an adequate basis for software development; it is a subprocess of the software requirements process. Inputs for requirements validation are typically a software requirements specification and the resources needed to perform validation (personnel, validation tools, sufficient time). The tasks of the requirements validation activity might include requirements reviews, prototyping, and model validation. These tasks involve work assignments for individuals and teams. The output of requirements validation is typically a validated software requirements specification that provides inputs to the software design and software testing processes. Requirements validation and other subprocesses of the software requirements process are often interleaved and iterated in various ways;

Figure 8.2. Elements of a Software Process

the software requirements process and its subprocesses may be entered and exited multiple times during software development or modification.

Complete definition of a software process may also include the roles and competencies, IT support, software engineering techniques and tools, and work environment needed to perform the process, as well as the approaches and measures (Key Performance Indicators) used to determine the efficiency and effectiveness of performing the process.

In addition, a software process may include interleaved technical, collaborative, and administrative activities.

Notations for defining software processes include textual lists of constituent activities and tasks described in natural language; data-flow diagrams; state charts; BPMN; IDEF0; Petri nets; and UML activity diagrams. The transforming tasks within a process may be defined as procedures; a procedure may be specified as an ordered set of steps or, alternatively, as a checklist of the work to be accomplished in performing a task.

It must be emphasized that there is no best software process or set of software processes. Software processes must be selected, adapted, and applied as appropriate for each project and each organizational context. No ideal process, or set of processes, exists.

1.1. Software Process Management

[3*, s26.1] [4*, p453–454]

Two objectives of software process management are to realize the efficiency and effectiveness that result from a systematic approach to accomplishing software processes and producing work products—be it at the individual, project, or organizational level—and to introduce new or improved processes.

Processes are changed with the expectation that a new or modified process will improve the efficiency and/or effectiveness of the process and the quality of the resulting work products. Changing to a new process, improving an existing process, organizational change, and infrastructure change (technology insertion or changes in tools) are closely related, as all are usually initiated with the goal of improving the cost, development schedule, or quality of the software products. Process change has impacts not only for the software product; they often lead to organizational change. Changing a process or introducing a new process can have ripple effects throughout an organization. For example, changes in IT infrastructure tools and technology often require process changes.

Existing processes may be modified when other new processes are deployed for the first time (for example, introducing an inspection activity within a software development project will likely impact the software testing process—see Reviews and Audits in the Software Quality KA and in the Software Testing KA). These situations can also be termed "process evolution." If the modifications are extensive, then changes in the organizational culture and business model will likely be necessary to accommodate the process changes.

1.2. Software Process Infrastructure

[2*, p183, p186] [4*, p437–438]

Establishing, implementing, and managing software processes and software life cycle models often occurs at the level of individual software projects. However, systematic application of software processes and software life cycle models across an organization can provide benefits to all software work within the organization, although it requires commitment at the organizational level. A software process infrastructure can provide process definitions, policies for interpreting and applying the processes, and descriptions of the procedures to be used to implement the processes. Additionally, a software process infrastructure may provide funding, tools, training, and staff members who have been assigned responsibilities for establishing and maintaining the software process infrastructure.

Software process infrastructure varies, depending on the size and complexity of the organization and the projects undertaken within the organization. Small, simple organizations and projects have small, simple infrastructure needs. Large, complex organizations and projects, by necessity, have larger and more complex software process infrastructures. In the latter case, various organizational units may be established (such as a software engineering process group or a steering committee) to oversee implementation and improvement of the software processes.

A common misperception is that establishing a software process infrastructure and implementing repeatable software processes will add time and cost to software development and maintenance. There is a cost associated with introducing or improving a software process; however, experience has shown that implementing systematic improvement of software processes tends to result in lower cost through improved efficiency, avoidance of rework, and more reliable and affordable software. Process performance thus influences software product quality.

2. Software Life Cycles

[1*, c2] [2*, p190]

This topic addresses categories of software processes, software life cycle models, software process adaptation, and practical considerations. A software development life cycle (SDLC) includes the software processes used to specify and transform software requirements into a deliverable software product. A software product life cycle (SPLC) includes a software development life cycle plus additional software processes that provide for deployment, maintenance, support, evolution, retirement, and all other inception-to-retirement processes for a software product, including the software configuration management and software quality assurance processes that are applied throughout a software product life cycle. A software product life cycle may include multiple software development life cycles for evolving and enhancing the software.

Individual software processes have no temporal ordering among them. The temporal relationships among software processes are provided by a software life cycle model: either an SDLC or SPLC. Life cycle models typically emphasize the key software processes within the model and their temporal and logical interdependencies and relationships. Detailed definitions of the software processes in a life cycle model may be provided directly or by reference to other documents.

In addition to conveying the temporal and logical relationships among software processes, the software development life cycle model (or models used within an organization) includes the control mechanisms for applying entry and exit criteria (e.g., project reviews, customer approvals, software testing, quality thresholds, demonstrations, team consensus). The output of one software process often provides the input for others (e.g., software requirements provide input for a software architectural design process and the software construction and software testing processes). Concurrent execution of several software process activities may produce a shared output (e.g., the interface specifications for interfaces among multiple software components developed by different teams). Some software processes may be regarded as less effective unless other software processes are being performed at the same time (e.g., software test planning during software requirements analysis can improve the software requirements).

2.1. Categories of Software Processes
[1*, Preface] [2* , p294–295] [3*, c22–c24]

Many distinct software processes have been defined for use in the various parts of the software development and software maintenance life cycles. These processes can be categorized as follows:

1. *Primary processes* include software processes for development, operation, and maintenance of software.
2. *Supporting processes* are applied intermittently or continuously throughout a software product life cycle to support primary processes; they include software processes such as configuration management, quality assurance, and verification and validation.
3. *Organizational processes* provide support for software engineering; they include training, process measurement analysis, infrastructure management, portfolio and reuse management, organizational process improvement, and management of software life cycle models.
4. *Cross-project processes,* such as reuse, software product line, and domain engineering; they involve more than a single software project in an organization.

Software processes in addition to those listed above include the following.

Project management processes include processes for planning and estimating, resource management, measuring and controlling, leading, managing risk, managing stakeholders, and coordinating the primary, supporting, organizational, and cross-project processes of software development and maintenance projects.

Software processes are also developed for particular needs, such as process activities that address software quality characteristics (see the Software Quality KA). For example, security concerns during software development may necessitate one or more software processes to protect the security of the development environment and reduce the risk of malicious acts. Software processes may also be developed to provide adequate grounds for establishing confidence in the integrity of the software.

2.2. Software Life Cycle Models
[1*, c2] [2*, s3.2] [3*, s2.1] [5]

The intangible and malleable nature of software permits a wide variety of software development life cycle models, ranging from linear models in which the phases of software development are accomplished sequentially with feedback and iteration as needed followed by integration, testing, and delivery of a single product; to iterative models in which software is developed in increments of increasing functionality on iterative cycles; to agile models that typically involve frequent demonstrations of working software to a customer or user representative who directs development of the software in short iterative cycles that produce small increments of working, deliverable software. Incremental, iterative, and agile models can deliver early subsets of working software into the user environment, if desired.

Linear SDLC models are sometimes referred to as predictive software development life cycle models, while iterative and agile SDLCs are referred to as adaptive software development life cycle models. It should be noted that various maintenance activities during an SPLC can be conducted using different SDLC models, as appropriate to the maintenance activities.

A distinguishing feature of the various software development life cycle models is the way in which software requirements are managed. Linear development models typically develop a complete set of software requirements, to the extent possible, during project initiation and planning. The software requirements are then rigorously controlled. Changes to the software requirements are based on change requests that are processed by a change control board (see Requesting, Evaluating and Approving Software Changes in the Change Control Board in the Software Configuration Management KA). An incremental model produces successive increments of working, deliverable software based on partitioning of the software requirements to be implemented in each of the increments. The software requirements may be rigorously controlled, as in a linear model, or there may be some flexibility in revising the software requirements as the software product evolves. Agile models may define product scope and high-level features initially; however, agile

models are designed to facilitate evolution of the software requirements during the project.

It must be emphasized that the continuum of SDLCs from linear to agile is not a thin, straight line. Elements of different approaches may be incorporated into a specific model; for example, an incremental software development life cycle model may incorporate sequential software requirements and design phases but permit considerable flexibility in revising the software requirements and architecture during software construction.

2.3. Software Process Adaptation

[1*, s2.7] [2*, p51]

Predefined SDLCs, SPLCs, and individual software processes often need to be adapted (or "tailored") to better serve local needs. Organizational context, innovations in technology, project size, product criticality, regulatory requirements, industry practices, and corporate culture may determine needed adaptations. Adaptation of individual software processes and software life cycle models (development and product) may consist of adding more details to software processes, activities, tasks, and procedures to address critical concerns. It may consist of using an alternate set of activities that achieves the purpose and outcomes of the software process. Adaptation may also include omitting software processes or activities from a development or product life cycle model that are clearly inapplicable to the scope of work to be accomplished.

2.4. Practical Considerations

[2*, p188–190]

In practice, software processes and activities are often interleaved, overlapped, and applied concurrently. Software life cycle models that specify discrete software processes, with rigorously specified entry and exit criteria and prescribed boundaries and interfaces, should be recognized as idealizations that must be adapted to reflect the realities of software development and maintenance within the organizational context and business environment.

Another practical consideration: software processes (such as configuration management, construction, and testing) can be adapted to facilitate operation, support, maintenance, migration, and retirement of the software.

Additional factors to be considered when defining and tailoring a software life cycle model include required conformance to standards, directives, and policies; customer demands; criticality of the software product; and organizational maturity and competencies. Other factors include the nature of the work (e.g., modification of existing software versus new development) and the application domain (e.g., aerospace versus hotel management).

3. Software Process Assessment and Improvement

[2*, p188, p194] [3*, c26] [4*, p397, c15]

This topic addresses software process assessment models, software process assessment methods, software process improvement models, and continuous and staged process ratings. Software process assessments are used to evaluate the form and content of a software process, which may be specified by a standardized set of criteria. In some instances, the terms "process appraisal" and "capability evaluation" are used instead of process assessment. Capability evaluations are typically performed by an acquirer (or potential acquirer) or by an external agent on behalf of an acquirer (or potential acquirer). The results are used as an indicator of whether the software processes used by a supplier (or potential supplier) are acceptable to the acquirer. Performance appraisals are typically performed within an organization to identify software processes in need of improvement or to determine whether a process (or processes) satisfies the criteria at a given level of process capability or maturity.

Process assessments are performed at the levels of entire organizations, organizational units within organizations, and individual projects. Assessment may involve issues such as assessing whether software process entry and exit criteria are being met, to review risk factors and risk management, or to identify lessons learned. Process assessment is carried out using both an assessment model and an assessment method. The model can provide a norm for a benchmarking

comparison among projects within an organization and among organizations.

A process audit differs from a process assessment. Assessments are performed to determine levels of capability or maturity and to identify software processes to be improved. Audits are typically conducted to ascertain compliance with policies and standards. Audits provide management visibility into the actual operations being performed in the organization so that accurate and meaningful decisions can be made concerning issues that are impacting a development project, a maintenance activity, or a software-related topic.

Success factors for software process assessment and improvement within software engineering organizations include management sponsorship, planning, training, experienced and capable leaders, team commitment, expectation management, the use of change agents, plus pilot projects and experimentation with tools. Additional factors include independence of the assessor and the timeliness of the assessment.

3.1. Software Process Assessment Models

[2*, s4.5, s4.6] [3*, s26.5] [4*, p44–48]

Software process assessment models typically include assessment criteria for software processes that are regarded as constituting good practices. These practices may address software development processes only, or they may also include topics such as software maintenance, software project management, systems engineering, or human resources management.

3.2. Software Process Assessment Methods

[1*, p322–331] [3*, s26.3] [4*, p44–48, s16.4] [6]

A software process assessment method can be qualitative or quantitative. Qualitative assessments rely on the judgment of experts; quantitative assessments assign numerical scores to software processes based on analysis of objective evidence that indicates attainment of the goals and outcomes of a defined software process. For example, a quantitative assessment of the software inspection process might be performed by examining the procedural steps followed and results obtained plus data concerning defects found and time required to find and fix the defects as compared to software testing.

A typical method of software process assessment includes planning, fact-finding (by collecting evidence through questionnaires, interviews, and observation of work practices), collection and validation of process data, and analysis and reporting. Process assessments may rely on the subjective, qualitative judgment of the assessor, or on the objective presence or absence of defined artifacts, records, and other evidence.

The activities performed during a software process assessment and the distribution of effort for assessment activities are different depending on the purpose of the software process assessment. Software process assessments may be undertaken to develop capability ratings used to make recommendations for process improvements or may be undertaken to obtain a process maturity rating in order to qualify for a contract or award.

The quality of assessment results depends on the software process assessment method, the integrity and quality of the obtained data, the assessment team's capability and objectivity, and the evidence examined during the assessment. The goal of a software process assessment is to gain insight that will establish the current status of a process or processes and provide a basis for process improvement; performing a software process assessment by following a checklist for conformance without gaining insight adds little value.

3.3. Software Process Improvement Models

[2*, p187–188] [3*, s26.5] [4*, s2.7]

Software process improvement models emphasize iterative cycles of continuous improvement. A software process improvement cycle typically involves the subprocesses of measuring, analyzing, and changing. The Plan-Do-Check-Act model is a well-known iterative approach to software process improvement. Improvement activities include identifying and prioritizing desired improvements (planning); introducing an improvement, including change management and training (doing); evaluating the improvement

as compared to previous or exemplary process results and costs (checking); and making further modifications (acting). The Plan-Do-Check-Act process improvement model can be applied, for example, to improve software processes that enhance defect prevention.

3.4. Continuous and Staged Software Process Ratings

[1*, p28–34] [3*, s26.5] [4*, p39–45]

Software process capability and software process maturity are typically rated using five or six levels to characterize the capability or maturity of the software processes used within an organization.

A *continuous* rating system involves assigning a rating to each software process of interest; a *staged* rating system is established by assigning the same maturity rating to all of the software processes within a specified process level. A representation of continuous and staged process levels is provided in Table 8.1. Continuous models typically use a level 0 rating; staged models typically do not.

Table 8.1. Software Process Rating Levels

Level	Continuous Representation of Capability Levels	Staged Representation of Maturity Levels
0	Incomplete	
1	Performed	Initial
2	Managed	Managed
3	Defined	Defined
4		Quantitatively Managed
5		Optimizing

In Table 8.1, level 0 indicates that a software process is incompletely performed or may not be performed. At level 1, a software process is being performed (capability rating), or the software processes in a maturity level 1 group are being performed but on an ad hoc, informal basis. At level 2, a software process (capability rating) or the processes in maturity level 2 are being performed in a manner that provides management visibility into intermediate work products and can exert some control over transitions between processes. At level 3, a single software process or the processes in a maturity level 3 group plus the process or processes in maturity level 2 are well defined (perhaps in organizational policies and procedures) and are being repeated across different projects. Level 3 of process capability or maturity provides the basis for process improvement across an organization because the process is (or processes are) conducted in a similar manner. This allows collection of performance data in a uniform manner across multiple projects. At maturity level 4, quantitative measures can be applied and used for process assessment; statistical analysis may be used. At maturity level 5, the mechanisms for continuous process improvements are applied.

Continuous and staged representations can be used to determine the order in which software processes are to be improved. In the continuous representation, the different capability levels for different software processes provide a guideline for determining the order in which software processes will be improved. In the staged representation, satisfying the goals of a set of software processes within a maturity level is accomplished for that maturity level, which provides a foundation for improving all of the software processes at the next higher level.

4. Software Measurement

[3*, s26.2] [4*, s18.1.1]

This topic addresses software process and product measurement, quality of measurement results, software information models, and software process measurement techniques (see Measurement in the Engineering Foundations KA).

Before a new process is implemented or a current process is modified, measurement results for the current situation should be obtained to provide a baseline for comparison between the current situation and the new situation. For example, before introducing the software inspection process, effort required to fix defects discovered by testing should be measured. Following an initial start-up period after the inspection process is introduced, the combined effort of inspection

plus testing can be compared to the previous amount of effort required for testing alone. Similar considerations apply if a process is changed.

4.1. Software Process and Product Measurement [1*, s6.3, p273] [3*, s26.2, p638]

Software process and product measurement are concerned with determining the efficiency and effectiveness of a software process, activity, or task. The *efficiency* of a software process, activity, or task is the ratio of resources actually consumed to resources expected or desired to be consumed in accomplishing a software process, activity, or task (see Efficiency in the Software Engineering Economics KA). Effort (or equivalent cost) is the primary measure of resources for most software processes, activities, and tasks; it is measured in units such as person-hours, person-days, staff-weeks, or staff-months of effort or in equivalent monetary units—such as euros or dollars.

Effectiveness is the ratio of actual output to expected output produced by a software process, activity, or task; for example, actual number of defects detected and corrected during software testing to expected number of defects to be detected and corrected—perhaps based on historical data for similar projects (see Effectiveness in the Software Engineering Economics KA). Note that measurement of software process effectiveness requires measurement of the relevant product attributes; for example, measurement of software defects discovered and corrected during software testing.

One must take care when measuring product attributes for the purpose of determining process effectiveness. For example, the number of defects detected and corrected by testing may not achieve the expected number of defects and thus provide a misleadingly low effectiveness measure, either because the software being tested is of better-than-usual quality or perhaps because introduction of a newly introduced upstream inspection process has reduced the remaining number of defects in the software.

Product measures that may be important in determining the effectiveness of software processes include product complexity, total defects, defect density, and the quality of requirements, design documentation, and other related work products.

Also note that efficiency and effectiveness are independent concepts. An effective software process can be inefficient in achieving a desired software process result; for example, the amount of effort expended to find and fix software defects could be very high and result in low efficiency, as compared to expectations.

An efficient process can be ineffective in accomplishing the desired transformation of input work products into output work products; for example, failure to find and correct a sufficient number of software defects during the testing process.

Causes of low efficiency and/or low effectiveness in the way a software process, activity, or task is executed might include one or more of the following problems: deficient input work products, inexperienced personnel, lack of adequate tools and infrastructure, learning a new process, a complex product, or an unfamiliar product domain. The efficiency and effectiveness of software process execution are also affected (either positively or negatively) by factors such as turnover in software personnel, schedule changes, a new customer representative, or a new organizational policy.

In software engineering, productivity in performing a process, activity, or task is the ratio of output produced divided by resources consumed; for example, the number of software defects discovered and corrected divided by person-hours of effort (see Productivity in the Software Engineering Economics KA). Accurate measurement of productivity must include total effort used to satisfy the exit criteria of a software process, activity, or task; for example, the effort required to correct defects discovered during software testing must be included in software development productivity.

Calculation of productivity must account for the context in which the work is accomplished. For example, the effort to correct discovered defects will be included in the productivity calculation of a software team if team members correct the defects they find—as in unit testing by software developers or in a cross-functional agile team. Or the productivity calculation may include either the effort of the software

developers or the effort of an independent testing team, depending on who fixes the defects found by the independent testers. Note that this example refers to the effort of teams of developers or teams of testers and not to individuals. Software productivity calculated at the level of individuals can be misleading because of the many factors that can affect the individual productivity of software engineers.

Standardized definitions and counting rules for measurement of software processes and work products are necessary to provide standardized measurement results across projects within an organization, to populate a repository of historical data that can be analyzed to identify software processes that need to be improved, and to build predictive models based on accumulated data. In the example above, definitions of software defects and staff-hours of testing effort plus counting rules for defects and effort would be necessary to obtain satisfactory measurement results.

The extent to which the software process is institutionalized is important; failure to institutionalize a software process may explain why "good" software processes do not always produce anticipated results. Software processes may be institutionalized by adoption within the local organizational unit or across larger units of an enterprise.

4.2. Quality of Measurement Results

[4*, s3.4–3.7]

The quality of process and product measurement results is primarily determined by the reliability and validity of the measured results. Measurements that do not satisfy these quality criteria can result in incorrect interpretations and faulty software process improvement initiatives. Other desirable properties of software measurements include ease of collection, analysis, and presentation plus a strong correlation between cause and effect.

The Software Engineering Measurement topic in the Software Engineering Management KA describes a process for implementing a software measurement program.

4.3. Software Information Models

[1*, p310–311] [3*, p712–713] [4*, s19.2]

Software information models allow modeling, analysis, and prediction of software process and software product attributes to provide answers to relevant questions and achieve process and product improvement goals. Needed data can be collected and retained in a repository; the data can be analyzed and models can be constructed. Validation and refinement of software information models occur during software projects and after projects are completed to ensure that the level of accuracy is sufficient and that their limitations are known and understood. Software information models may also be developed for contexts other than software projects; for example, a software information model might be developed for processes that apply across an organization, such as software configuration management or software quality assurance processes at the organizational level.

Analysis-driven software information model building involves the development, calibration, and evaluation of a model. A software information model is developed by establishing a hypothesized transformation of input variables into desired outputs; for example, product size and complexity might be transformed into estimated effort needed to develop a software product using a regression equation developed from observed data from past projects. A model is calibrated by adjusting parameters in the model to match observed results from past projects; for example, the exponent in a nonlinear regression model might be changed by applying the regression equation to a different set of past projects other than the projects used to develop the model.

A model is evaluated by comparing computed results to actual outcomes for a different set of similar data. There are three possible evaluation outcomes:

1. results computed for a different data set vary widely from actual outcomes for that data set, in which case the derived model is not applicable for the new data set and should not be applied to analyze or make predictions for future projects;

2. results computed for a new data set are close to actual outcomes for that data set, in which case minor adjustments are made to the parameters of the model to improve agreement;
3. results computed for the new data set and subsequent data sets are very close and no adjustments to the model are needed.

Continuous evaluation of the model may indicate a need for adjustments over time as the context in which the model is applied changes.

The Goals/Questions/Metrics (GQM) method was originally intended for establishing measurement activities, but it can also be used to guide analysis and improvement of software processes.

It can be used to guide analysis-driven software information model building; results obtained from the software information model can be used to guide process improvement.

The following example illustrates application of the GQM method:

- Goal: Reduce the average change request processing time by 10% within six months.
- Question 1-1: What is the baseline change request processing time?
- Metric 1-1-1: Average of change request processing times on starting date
- Metric 1-1-2: Standard deviation of change request processing times on starting date
- Question 1-2: What is the current change request processing time?
- Metric 1-2-1: Average of change request processing times currently
- Metric 1-2-2: Standard deviation of change request processing times currently

4.4. Software Process Measurement Techniques

[1*, c8]

Software process measurement techniques are used to collect process data and work product data, transform the data into useful information, and analyze the information to identify process activities that are candidates for improvement. In some cases, new software processes may be needed.

Process measurement techniques also provide the information needed to measure the effects of process improvement initiatives. Process measurement techniques can be used to collect both quantitative and qualitative data.

4.4.1. Quantitative Process Measurement Techniques

[4*, s5.1, s5.7, s9.8]

The purpose of quantitative process measurement techniques is to collect, transform, and analyze quantitative process and work product data that can be used to indicate where process improvements are needed and to assess the results of process improvement initiatives. Quantitative process measurement techniques are used to collect and analyze data in numerical form to which mathematical and statistical techniques can be applied.

Quantitative process data can be collected as a byproduct of software processes. For example, the number of defects discovered during software testing and the staff-hours expended can be collected by direct measurement, and the productivity of defect discovery can be derived by calculating defects discovered per staff-hour.

Basic tools for quality control can be used to analyze quantitative process measurement data (e.g., check sheets, Pareto diagrams, histograms, scatter diagrams, run charts, control charts, and cause-and-effect diagrams) (see Root Cause Analysis in the Engineering Foundations KA). In addition, various statistical techniques can be used that range from calculation of medians and means to multivariate analysis methods (see Statistical Analysis in the Engineering Foundations KA).

Data collected using quantitative process measurement techniques can also be used as inputs to simulation models (see Modeling, Prototyping, and Simulation in the Engineering Foundations KA); these models can be used to assess the impact of various approaches to software process improvement.

Orthogonal Defect Classification (ODC) can be used to analyze quantitative process measurement data. ODC can be used to group detected defects into categories and link the defects in

each category to the software process or software processes where a group of defects originated (see Defect Characterization in the Software Quality KA). Software interface defects, for example, may have originated during an inadequate software design process; improving the software design process will reduce the number of software interface defects. ODC can provide quantitative data for applying root cause analysis.

Statistical Process Control can be used to track process stability, or the lack of process stability, using control charts.

4.4.2. Qualitative Process Measurement Techniques

[1*, s6.4]

Qualitative process measurement techniques—including interviews, questionnaires, and expert judgment—can be used to augment quantitative process measurement techniques. Group consensus techniques, including the Delphi technique, can be used to obtain consensus among groups of stakeholders.

5. Software Engineering Process Tools

[1*, s8.7]

Software process tools support many of the notations used to define, implement, and manage individual software processes and software life cycle models. They include editors for notations such as data-flow diagrams, state charts, BPMN, IDEF0 diagrams, Petri nets, and UML activity diagrams. In some cases, software process tools allow different types of analyses and simulations (for example, discrete event simulation). In addition, general purpose business tools, such as a spreadsheet, may be useful.

Computer-Assisted Software Engineering (CASE) tools can reinforce the use of integrated processes, support the execution of process definitions, and provide guidance to humans in performing well-defined processes. Simple tools such as word processors and spreadsheets can be used to prepare textual descriptions of processes, activities, and tasks; these tools also support traceability among the inputs and outputs of multiple software processes (such as stakeholder needs analysis, software requirements specification, software architecture, and software detailed design) as well as the results of software processes such as documentation, software components, test cases, and problem reports.

Most of the knowledge areas in this *Guide* describe specialized tools that can be used to manage the processes within that KA. In particular, see the Software Configuration Management KA for a discussion of software configuration management tools that can be used to manage the construction, integration, and release processes for software products. Other tools, such as those for requirements management and testing, are described in the appropriate KAs.

Software process tools can support projects that involve geographically dispersed (virtual) teams. Increasingly, software process tools are available through cloud computing facilities as well as through dedicated infrastructures.

A project control panel or dashboard can display selected process and product attributes for software projects and indicate measurements that are within control limits and those needing corrective action.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Fairley 2009 [1*]	Moore 2009 [2*]	Sommerville 2011 [3*]	Kan 2003 [4*]
1. Software Process Definition	p177	p295	p28–29, p36, c5	
1.1. Software Process Management			s26.1	p453–454
1.2. Software Process Infrastructure		p183, p186		p437–438
2. Software Life Cycles	c2	p190		
2.1. Categories of Software Processes	preface	p294–295	c22, c23, c24	
2.2. Software Life Cycle Models	c2	s3.2	s2.1	
2.3. Software Process Adaptation	s2.7	p51		
2.4. Practical Considerations		p188–190		
3. Software Process Assessment and Improvement		p188, p194	c26	p397, c15
3.1. Software Process Assessment Models		s4.5, s4.6	s26.5	p44–48
3.2. Software Process Assessment Methods	p322–331		s26.3	p44–48, s16.4
3.3. Software Process Improvement Models		p187–188	s26.5	s2.7
3.4. Continuous and Staged Ratings	p28–34		s26.5	p39–45
4. Software Measurement			s26.2	s18.1.1
4.1. Software Process and Product Measurement	s6.3, p273		s26.2, p638	
4.2. Quality of Measurement Results				s3.4, s3.5, s3.6, s3.7
4.3. Software Information Models	p310–311		p. 712–713	s19.2
4.4. Software Process Measurement Techniques	s6.4, c8			s5.1, s5.7, s9.8
5. Software Engineering Process Tools	s8.7			

FURTHER READINGS

Software Extension to the Guide to the Project Management Body of Knowledge® (SWX) [5].

SWX provides adaptations and extensions to the generic practices of project management documented in the *PMBOK® Guide* for managing software projects. The primary contribution of this extension to the *PMBOK® Guide* is description of processes that are applicable for managing adaptive life cycle software projects.

D. Gibson, D. Goldenson, and K. Kost, "Performance Results of CMMI-Based Process Improvement" [6].

This technical report summarizes publicly available empirical evidence about the performance results that can occur as a consequence of CMMI-based process improvement. The report contains a series of brief case descriptions that were created with collaboration from representatives from 10 organizations that have achieved notable quantitative performance results through their CMMI-based improvement efforts.

CMMI® for Development, Version 1.3 [7].

CMMI® for Development, Version 1.3 provides an integrated set of process guidelines for developing and improving products and services. These guidelines include best practices for developing and improving products and services to meet the needs of customers and end users.

ISO/IEC 15504-1:2004 Information technology—Process assessment—Part 1: Concepts and vocabulary [8].

This standard, commonly known as SPICE (Software Process Improvement and Capability Determination), includes multiple parts. Part 1 provides concepts and vocabulary for software development processes and related business-management functions. Other parts of 15504 define the requirements and procedures for performing process assessments.

REFERENCES

[1*] R.E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

[2*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006.

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

[4*] S.H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed., Addison-Wesley, 2002.

[5] Project Management Institute and IEEE Computer Society, *Software Extension to the PMBOK® Guide Fifth Edition*, ed: Project Management Institute, 2013.

[6] D. Gibson, D. Goldenson, and K. Kost, "Performance Results of CMMI-Based Process Improvement," Software Engineering Institute, 2006; http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=8065.

[7] CMMI Product Team, "CMMI for Development, Version 1.3," Software Engineering Institute, 2010; http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=9661.

[8] *ISO/IEC 15504-1:2004, Information Technology—Process Assessment—Part 1: Concepts and Vocabulary*, ISO/IEC, 2004.

CHAPTER 9

SOFTWARE ENGINEERING MANAGEMENT

ABBREVIATIONS

PMBOK® Guide	*Guide to the Project Management Body of Knowledge*
SDLC	Software development life cycle
SEM	Software engineering management
SQA	Software quality assurance
SWX	*Software Extension to the PMBOK® Guide*
WBS	Work breakdown structure
PSM	Practical Software and Systems Measurement

INTRODUCTION

Software engineering management (SEM) can be defined as a collection of work activities involved with planning, estimating, measuring, controlling, coordinating, leading and managing risk factors for a software project to ensure that software products and software engineering services are delivered efficiently, effectively and to the stakeholders' benefit [3]. Although project management and measurement management are often seen as separate areas, and each possesses many unique attributes, the close relationship between the two has led to their combined treatment in this knowledge area (KA). other complex endeavors are managed. However, software engineers use different process

In one sense, it should be possible to manage a software engineering project in the same way models, technical processes and problem-solving styles than other engineers do, making these choices based on their education and experience and on the differences between physical and software attributes. Software system elements are logical constructions expressed in logarithmic form, while physical system elements are realized in mechanical, electrical, chemical, biological and other physical media. Software is intangible because it has no physical properties and is malleable because of the relative ease with which code can be modified. Obtaining the desired effect by modifying software code might not be easy, but code modifications, per se, are straightforward compared with the modification of physical elements that have already been constructed [12].

As software and software-embedded systems become bigger, more complex and more intertwined, software management and engineering roles are evolving in response [10], because skilled individuals must actively develop and maintain these systems. Consider the following: hardware is different from software (and not all software is the same). Hardware can be developed, procured and maintained in a linear fashion. Software is an enduring capability that must be supported and continuously improved throughout its life cycle [13]. Furthermore, the malleable nature of software allows iteration among and interleaving of the development phases (to a much greater

degree than is possible when developing physical artifacts).

Software is made by people and for people, so digital talent matters. Software projects are increasingly important, and their ongoing success largely depends on people with the right skills, knowledge and abilities. Therefore, people management activities, such as team and teamwork, leadership, communication, and coordination activities, are important to project success.

Other issues can complicate effective management of software projects and software life cycle processes, including the following:

- Clients often do not know what is needed or what is feasible.
- Increased understanding and changing conditions will likely generate new or changed software requirements.
- Clients often do not appreciate the complexities inherent in software engineering, particularly regarding the impact of changing requirements.
- As a result of changing requirements and software malleability, software is often built iteratively rather than as a linear sequence of phases.
- Software is nominally an enduring capability that must be supported and continuously improved throughout its lifecycle
- Software construction differs from hardware implementation in that design is usually part of software construction, whereas in hardware-oriented systems, design precedes hardware implementation to "get it right" prior to procurement or fabrication of hardware [12].
- Software engineering necessarily incorporates creativity and discipline. Maintaining an appropriate balance between the two is sometimes difficult [5].
- The development of software capabilities often involves a high degree of novelty and complexity.
- Typically, the underlying technology has a high rate of change.
- Computer software has become a key component of most modern systems. Software has been elevated to a highly prominent role because of its flexibility and relatively low replication cost compared with hardware.
- Speed and cycle time are important metrics for managing software. Software capabilities are often delivered at increasing speed to satisfy business and mission needs [13].

SEM activities occur on three levels: organizational and infrastructure management, project management, and management of the measurement program. The last two are covered in detail in this KA description. This fact does not diminish the importance of organizational and infrastructure management issues but rather points out that software organizational engineering managers should be conversant with the project management and software measurement knowledge described in this KA. They should also possess some target domain knowledge. Likewise, it also helps for managers of complex projects and programs where software is part of the system architecture to know what issues software engineering processes (versus other types of engineering processes) introduce into project management and project measurement.

Other aspects of organizational management affect software engineering — for example, organizational policies and procedures that provide the framework for software engineering projects. These policies and procedures might need to be adjusted for effective software development and maintenance requirements. In addition,

several policies specific to software engineering might need to be in place or established for the effective management of software engineering at the organizational level. For example, policies are usually necessary to establish specific organization-wide processes or procedures for software engineering tasks such as software design, software construction, estimating, monitoring and reporting. Such policies are important for effective long-term management of software engineering projects across an organization (e.g., one such policy could establish a consistent basis for analyzing past project performance and implementing improvements).

Another important aspect of organizational management is the use of personnel management policies and procedures for hiring, training and mentoring — not only for a project's success, but also for the organization's long-term success. Given the projected scarcity of skilled software engineers, it is important to provide an environment that attracts and retains good talent. Software engineering personnel can present unique training or personnel management challenges (e.g., maintaining currency in a context where the underlying technology undergoes rapid and continuous change) as part of career development.

Communication management is also often mentioned as an overlooked but important aspect of success in a field where a precise understanding of user needs, software requirements and software designs is necessary. Furthermore, portfolio management is desirable, which provides an overall view of software under development in various projects and programs (integrated projects) of planned software, and of software already in use in an organization. Also, software reuse can be a key factor in maintaining and improving productivity and competitiveness. Effective reuse requires a strategic vision that reflects the advantages and disadvantages of reuse.

Software engineers should have a sound understanding of the aspects of management that are unique to software projects, and they should also have some knowledge of the more general aspects of management discussed in this KA (even in the first few years after graduation).

Certain attributes of organizational culture and behavior, as well as management of functional areas of the enterprise outside the immediate software engineering realm, can influence an organization's software engineering processes, albeit indirectly. Software projects are often targeted at changing the way people work — but culture change is difficult, complicated and unlikely to succeed without a significant effort. For this reason, leadership is an important attribute for program managers, as they often need to lead the charge for digital transformation. They might need to galvanize their teams and other stakeholders to bring their very best to every project pursuing major change.

Extensive information concerning software project management can be found in the *Guide to the Project Management Body of Knowledge (PMBOK® Guide)* and the *Software Extension to the PMBOK® Guide* (*SWX*) [1, 2]. Each of these guides includes 10 project management KAs: project integration management, project scope management, project time/schedule management, project cost management, project quality management, project resource/human management, project communications management, project risk management, project procurement management and project stakeholder management. Each KA has direct relevance to this SEM KA.

Additional information is also provided in this KA's references and list of further reading.

This SEM KA discusses the software project management processes shown as the first five topics in Figure 1 (Initiation and Scope Definition, Software Project Planning, Software Project Enactment, Review and Evaluation, Closure), as well as Software Engineering Measurement (the sixth topic shown in the figure) and Software Engineering Management Tools (the seventh topic).

Unfortunately, a common perception of the software industry is that software products often are delivered late, are over budget, are of poor quality and have incomplete functionality. Measurement-informed management — a basic principle of any true engineering discipline (see Measurement in the Engineering Foundations KA) — can help improve perception and reality. In essence, management without measurement (qualitative and quantitative) suggests a lack of discipline, and measurement without management suggests a lack of purpose or context. To be effective, software engineers must use both measurement and management.

The following working definitions are adopted here:

- *Management* is a system of processes and controls required to achieve the strategic objectives set by the organization.
- *Measurement* refers to the assignment of values and labels to software engineering work products, processes and resources, plus the models derived from them, whether these models are developed using statistical or other techniques [3*, c7, c8].

The software engineering project management sections in this KA use the Software Engineering Measurement section extensively.

This KA is closely related to others in the *SWEBOK Guide*; reading the following KA descriptions will be particularly helpful in understanding this one:

- The Engineering Foundations KA describes some general measurement concepts that directly apply to the Software Engineering Measurement section of this KA. In addition, the concepts and techniques presented in the Statistical Analysis section of the Engineering Foundations KA apply directly to many topics in this KA.
- The Software Requirements KA describes activities that should be performed during the project's Initiation and Scope Definition phase.
- The Software Configuration Management KA deals with the identification, control, status accounting and auditing of software configurations, along with software release management and delivery and software configuration management tools.
- The Software Engineering Process KA describes software life cycle models and the relationships between processes and work products.
- The Software Quality KA emphasizes quality as a management goal and as an aim of many software engineering activities.
- The Software Engineering Economics KA discusses how to make software-related decisions in a business context.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING MANAGEMENT

Because most software development life cycle (SDLC) models require similar activities that may be executed in different ways, the topic breakdown, shown in Figure 1, is activity-based. The top-level elements shown in the figure are activities that are usually performed when a software development project is being managed, regardless of which SDLC model is being

used (see Software Life Cycle Models in the Software Engineering Process KA). This breakdown does not recommend a specific life cycle model. However, it is important to note the impact the choice of software can have on business success and the associated development of software life cycle models to accommodate changing business needs.

Delivery speed, continuous adaptation and frequent modular upgrades to deliver new capabilities are often key business differentiators and project management imperative [11, 13]. These imperatives should be balanced with risk management activities.

Several software life cycle process models have been and are being developed to shorten development cycles in response to the changing business needs. Most of these processes involve Agile SDLC approaches [14]. The Agile approach assumes that teams can develop high-quality, adaptive software using continuous design improvement principles and testing based on rapid feedback and change. In comparison, the traditional approach assumes that software-intensive systems are fully specifiable and predictable and can be built through meticulous and extensive planning. The management style associated with the Agile approach emphasizes leadership and collaboration at the team level, whereas the management style of the highly predictive approach is more formal (top-down). Many Agile approaches integrate different management approaches.

For example, Dev/Sec/Ops is a culture and an Agile approach to modern software delivery that aligns development (Dev), security (Sec) and operations (Ops) groups into an integrated team focused on continuous, incremental delivery of capabilities. The main characteristic of Dev/Sec/Ops is that this approach automates, continuously monitors and applies security at all phases of the software life cycle: plan, develop, build, test, release, deliver, deploy, operate and monitor. In Dev/Sec/Ops, testing and security are shifted to the left through automated unit, functional, integration and security testing. This is a key Dev/Sec/Ops differentiator; security/quality assurance (QA) and other nonfunctional and functional capabilities are tested and built simultaneously [11, 14]. Whereas Dev/Sec/Ops encompasses the culture and processes that enable rapid, continual delivery of cyber-resilient systems, complex software-embedded systems can have additional demands that must also be integrated into the Dev/Sec/Ops culture and processes, such as safety. Elevating these demands to be on par with Dev/Sec/Ops highlights the importance of incorporating quality into all program aspects. The complexity of the end-to-end Dev/Sec/Ops tools and of using emerging technologies such as artificial intelligence (AI) and machine learning (ML) to leverage those tools adds another dimension [15].

It is important to understand the difference between *phases* and *activities* and why an activities breakdown is used. The Project Management Institute (PMI) describes a *phase* this way: "The completion and approval of one or more deliverables characterizes a project phase." A *deliverable* is a measurable, verifiable work product such as a specification, feasibility study report, detailed design document or working prototype. Some deliverables correspond to part of the project management process, whereas others are the end products or components of the end products for which the project was conceived. The deliverables, and hence the phases, are part of a generally sequential process designed to ensure proper control of the project and to attain the desired product or service, which is the project's

objective. From a project management perspective, phases help accomplish project objectives and maintain control over the project.

The activity-based breakdown in Figure 1 shows what happens but does not imply when, how or how many times each activity occurs. The seven topics are the following:

- Initiation and Scope Definition, which deals with the decision to embark on a software engineering project
- Software Project Planning, which addresses the activities undertaken to prepare for a successful software engineering project from the management perspective
- Software Project Enactment, which deals with generally accepted SEM activities that occur during a software engineering project's execution
- Review and Evaluation, which deals with ensuring that technical, schedule, cost and quality engineering activities are satisfactory
- Closure, which addresses the activities accomplished to complete a project
- Software Engineering Measurement, which deals with the effective development and implementation of measurement programs in software engineering organizations
- Software Engineering Management Tools, which describes the selection and use of tools for managing a software engineering project

Figure 1. Breakdown of Topics for the Software Engineering Management KA

1. Initiation and Scope Definition

These activities focus on determining software requirements using various elicitation methods and assessing project feasibility from various standpoints. Once project feasibility has been established, the remaining tasks in this section are specifying requirements and selecting the processes for requirements revision and review.

1.1. Determination and Negotiation of Requirements [3, c3]*

Determining and negotiating the project requirements are the overarching goals of the tasks undertaken during this phase (see the Software Architecture KA and Software Requirements KA). Activities include requirements elicitation, analysis, specification and validation. Methods and techniques should be selected and applied considering the various stakeholder perspectives. Requirements provide the basis for all that follows on a software project and are captured in a Project Charter or other high-level project initiation document.

1.2. Feasibility Analysis [4, c5]*

The purpose of the feasibility analysis is to develop a clear description of project objectives and to evaluate alternative approaches to determine whether the proposed project solution is the best approach, given the constraints of technology, resources, finances and social/political considerations. An initial project and product scope statement, project deliverables, project

duration constraints, and an estimation of resources needed should be prepared.

Resources (which can be internal or external to the organization) include infrastructure, support, and people with the necessary core competencies. The feasibility analysis often requires estimations of effort and cost based on appropriate methods. (See Section 2.3, Effort, Schedule and Cost Estimation.)

An initial work breakdown structure (WBS) and context diagram may be developed during the project's Initiation and Scope Definition phase activities. Breaking work into smaller tasks is a common productivity technique that makes the work more manageable and approachable. As the project tool that uses this technique, WBS is an important project management document. The WBS integrates scope, cost and schedule baselines, ensuring that project plans align. An engineering context diagram defines the boundary between the system (or a part of the system) and its environment, showing the entities interacting with it. This document is important in defining management and technical interfaces and trade-offs that must be considered [1]. While engineers are developing the WBS, they should consider all configuration items as tasks to have under control.

1.3. Process for the Review and Revision of Requirements [3, c3]*

Given the inevitability of change, stakeholders should agree on how requirements and scope will be reviewed and revised (e.g., change management and trade-off procedures, iterative cycle retrospectives). This indicates that scope and requirements will not be "set in stone" but can and should be revisited at predetermined points as the project unfolds (for example, at the time when backlog priorities are created or at milestone reviews). If changes are accepted, then traceability analysis and risk analysis should be used to ascertain the impact of those changes. (See Section 2.5, Risk Management, and Software Configuration Control in the Software Configuration Management KA.)

A managed-change approach can also form the basis for evaluating success during closure of an incremental cycle or an entire project, based on changes that occurred along the way. (See Topic 5, Closure).

2. Software Project Planning

A key step in software project planning should be selecting an appropriate SDLC model and, perhaps, tailoring it based on project scope, software requirements and a risk assessment. The *SWX* [2] states that project life cycles occupy a continuum from predictive to adaptive. Factors that characterize the positions of software project life cycles within the continuum include (but are not limited to) the various ways requirements and plans are handled, how risk and cost are managed, and key stakeholder involvement. Highly predictive software project life cycles emphasize requirements specification and detailed planning during the project's initiation and planning phases. Detailed plans based on a known architecture, requirements and constraints are used to reduce risk and cost. Milestones are planned, versus continuous key stakeholder involvement. Highly adaptive software project life cycles, on the other hand, are characterized by progressive requirements specification based on short iterative development cycles. Risk and cost are reduced by progressive evolution of initial plans, and key stakeholders are continuously involved [2].

Other factors to consider include the nature of the application domain, functional and technical complexity, and software quality requirements. (See Software Quality Requirements in the Software Quality KA.)

In all SDLCs, risk assessment should be an element of initial project planning, and the "risk profile" of the project should be discussed and accepted by all relevant stakeholders. Software quality management processes (see Software Quality Management Processes in the Software Quality KA) should be determined as part of the planning process; this planning should establish procedures and responsibilities for software quality assurance (SQA), verification and validation, reviews, and audits. (See the Software Quality KA.) Processes and responsibilities for ongoing review and revision of the project plan and related plans should also be clearly stated and agreed upon.

2.1. Process Planning [3, c3, c4, c5], [5*, c1]*

SDLC models span a continuum from predictive to adaptive. (See Software Life Cycle Models in the Software Engineering Process KA.) Predictive SDLCs are characterized by the development of detailed software architecture and software requirements, detailed project planning, and minimal planning for iteration among development phases. Adaptive SDLCs are designed to accommodate emergent software requirements and iterative adjustment of plans. A highly predictive SDLC executes the first five processes listed in Figure 1 in a linear sequence with revisions to earlier phases only as necessary. Adaptive SDLCs are characterized by iterative development cycles. SDLCs in the midrange of the SDLC continuum produce increments of functionality on either a preplanned schedule (on the predictive side of the continuum) or as the products of frequently updated development cycles (on the adaptive side of the continuum).

Well-known SDLCs include the waterfall, incremental and spiral models, plus various Agile software development approaches [2, 11] [3*, c2].

Relevant methods (see the Software Engineering Models and Methods KA) and tools should be selected as part of planning. Automated tools that will be used throughout the project should also be planned for and acquired. Tools might include those for project scheduling, software requirements, software design, software construction, software maintenance, software configuration management, software engineering process and software quality, among others. Many of these tools should be selected based primarily on the technical considerations discussed in other KAs, but some of those concerns are closely related to the management considerations discussed in this chapter.

2.2. Determine Deliverables [3, c4, c5, c6]*

Each project activity's work product(s) (e.g., software architecture design documents, inspection reports, tested software) should be identified and characterized. Opportunities to reuse software components from previous projects or to use off-the-shelf software products should be evaluated. Software procurement and use of third parties to develop deliverables should be planned and suppliers selected. (See Section 3.2, Software Acquisition and Supplier Contract Management.)

2.3. Effort, Schedule and Cost Estimation

The topic of estimation in general is addressed in the Software Engineering Economics KA. Questions like "What is estimation?" and "Why do we estimate?" are addressed there. This section addresses management-specific estimation topics.

Estimating costs for software projects is an error-prone process. The effort required for any given software project depends almost entirely on human elements: individuals' experience and capabilities, team members' interactions, and the culture of the software development environment. Dynamic environmental factors, such as rapid technology evolution, changing and emergent requirements, and the intangible nature of the product, also significantly affect cost management. Estimating costs when this much variability exists is difficult even when significant historical data exists. Software project managers should use multiple estimation approaches and then reconcile the differences among the estimates [3, 10, 11].

When data is available, the estimated range of effort required for a project, or parts of a project, can be determined using a calibrated estimation model based on historical size and effort data. It is best to also use bottom-up estimation techniques based on estimates from those who will accomplish the work and historical data based on similar projects [2]. Task dependencies can be established, and potential opportunities for completing tasks concurrently and sequentially can be identified and documented, using a Gantt chart, for example. In predictive SDLC projects, the expected schedule of tasks, with projected start times, durations and end times, is typically produced during planning. In adaptive SDLC projects, an overall estimate of effort and schedule is typically developed from the initial understanding of the requirements, or, alternatively, constraints on overall effort and schedule may be specified and used to determine an initial estimate of the number of iterative cycles and estimates of effort and other resources allocated to each cycle.

Resource requirements (for example, people and tools needed) can usually be translated into cost estimates. The estimation of effort, schedule and cost is an iterative activity that should be negotiated and revised among affected stakeholders until consensus is reached on resources and time available for project completion. Project managers should constantly monitor stakeholder requirements and changes as they evolve to analyze their impact on the project cost and schedule. This is usually more important in Agile software development projects, where stakeholder requirements are dynamic because changes might occur rapidly as the project progresses.

2.4. Resource Allocation [3, c5, c10, c11]*

Equipment, facilities and people should be allocated to the identified tasks, including allocating responsibilities for completing various project elements and the overall project. A matrix that shows who is responsible for, accountable for, consulted about and informed about each task can be used. Resource allocation is based on and constrained by the availability of resources and their optimal use, and by issues relating to personnel (e.g., productivity of individuals and teams, team dynamics, and team structures).

2.5. Risk Management [3, c9] [5*, c5]*

Risk and uncertainty are related but distinct concepts. Uncertainty results from a lack of information. Risk is characterized by the probability of an event that will result in a negative impact and by how that impact would affect the project. Uncertainty often creates risk The converse of risk is opportunity, which is characterized by the probability that an event having a positive outcome might occur.

Risk management entails identifying risk factors, analyzing probability and potential impact of each risk factor, prioritizing risk factors, and developing risk mitigation strategies to reduce the probability of a

negative event and to minimize the negative impact if a risk factor becomes a problem. Risk management data can be used to represent the project risk profile; this data is often part of a risk register. A risk register is a document used as a risk management tool. It can be used to fulfill regulatory compliance, serving as a repository for all risks identified and for additional information about each risk [2]. Risk assessment methods (e.g., expert judgment, historical data, decision trees and process simulations) can sometimes be used to identify and evaluate risk factors.

Project abandonment conditions can also be determined with all relevant stakeholders. Software-unique aspects of risk, such as software engineers' tendency to add unneeded features or the risks related to software's intangible nature, can influence risk management for software projects. Particular attention should be paid to managing risks related to software quality requirements such as safety or security [11]. (See the Software Quality KA.) Risk management should be done not only at the beginning of a project, but also at periodic intervals throughout the project life cycle.

2.6. Quality Management [3, c4] [4*, c24]*

According to the *PMBOK® Guide*, Project quality management includes the performing organization's processes and activities that determine quality policies, objectives and responsibilities so the project will satisfy the needs for which it was undertaken. This section discusses additional considerations for managing software project quality [1]. Software quality requirements for a software project and the associated work products should be identified, perhaps both quantitatively and qualitatively. Quality attributes of software include but are not limited to safety, security, reliability, availability, performance, ease of use and ease of modification. *SWX* Section 1.9 lists quality attributes that are important for software users (e.g., efficiency, safety, security, reliability, availability) and quality attributes that are important to software developers and maintainers (e.g., maintainability is important to those who provide sustainment services) [1]. The International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) 25000 series of standards provides extensive lists of software quality attributes that align with different stakeholder needs [2]. This alignment is consistent with ISO/IEC 15939 and Practical Software and Systems Measurement (PSM) [2, 9.11].

Large portions of system functionality are shifting from hardware to software to capitalize on the increased flexibility and speed of component delivery that software can provide. However, with these benefits come other challenges — for example, the need for increased management of software quality requirements (e.g., cybersecurity) throughout the SDLC [11]. Thresholds for acceptable quality measurements should be set for each software quality requirement based on stakeholder needs and expectations. Procedures concerned with ongoing SQA and quality improvement throughout the development process and with verifying and validating the deliverable software product should also be specified during quality planning (e.g., technical reviews and inspections or demonstrations of completed functionality). (See the Software Quality KA.)

2.7. Plan Management [3, c4]*

Plans should be developed and managed for software projects when change is expected. The magnitude of the planning effort and the plan's content should be determined partly by the risk of not developing the plan. The

management of the project plan should itself be planned. Plans and processes selected for software development should be systematically monitored, reviewed, reported and, when appropriate, revised. Plans associated with supporting processes (e.g., documentation, software configuration management, and problem resolution) also should be managed. Reporting, monitoring and controlling a project should fit within the selected SDLC and the realities of the project. Plans should account for the various artifacts that will be used to manage the project.

Project managers of predictive life cycle software projects put substantial effort into up-front development of the project plan and integration of subsidiary plans developed by support personnel from other organizational units (e.g., estimation specialists in the Project Management Office (PMO)). In adaptive projects, managers spend less effort up front on developing detailed scope, cost and schedule plans. But significant effort is typically spent defining monitor and control processes, such as requirements traceability, to ensure coordination among the project members or teams as the emerging plans are implemented [2].

3. Software Project Enactment

During software project enactment (also known as *project execution*), plans are implemented, and the processes embodied in the plans are enacted. Throughout, there should be a focus on adherence to the selected SDLC processes, with an overriding expectation that adherence will satisfy stakeholder requirements and achieve the project's objectives. Fundamental to enactment are the ongoing management activities of monitoring, controlling and reporting.

3.1. Implementation of Plans [4, c2]*

Project activities should follow the project plan and supporting plans. Project activities use resources (personnel, technology and funding) and generate work products (software design, software code and software test cases).

3.2. Software Acquisition and Supplier Contract Management [3, c3, c4]*

Software acquisition and supplier contract management concern issues involved in contracting with customers of the software development organization who acquire the deliverable work products and with suppliers who supply products or services to the software engineering organization.

Software acquisition is common practice in software development projects, with integrated development environments (IDEs) and package libraries allowing software engineers to acquire third-party libraries with minimal steps, facilitating the assessment of risk, legality and suitability. However, software is no longer exclusively acquired as a shrink-wrapped product via a complex supply chain process and purchasing route. The ease of acquiring software has resulted in a common attack surface and led to security vulnerabilities. Organizations should consider introducing technical or procedural controls to minimize risk potentially exposed by unfiltered access to external library repositories.

The different software acquisition classes include commercial off-the-shelf (COTS) software — an existing product acquired "as is" from another software vendor, with applicable license terms; software developed exclusively for the organization by another party — typically contracted and sometimes a customization of COTS software; open source software — nominally free, although

the organization may purchase enhanced support or maintenance and must review the license for restrictions on use; customer loaned software — typically to provide simulation or integration with another system element; software as a service (SaaS) — which might include software the organization rents to fulfill a particular need (for example, a cloud-based hosting, source control or development environment).

Software projects typically use different acquisition approaches to obtain the necessary software components. However, regardless of how the software components are obtained, the following activities must be performed: verifying that each component is complete, correct and consistent concerning the architectural design and software requirements for that component; integrating the components; verifying that the integrated components are correct, complete and consistent concerning the architectural design and the software requirements; and validating that the integrated components will satisfy their intended purpose when used in their intended operating environment.

Different acquisition approaches (for obtaining software components) require different approaches to managing the project. For example, custom development requires detailed planning for the numbers and skills of the software developers, organizing the development team(s), allocating requirements to the teams, specifying project metrics to be collected, monitoring progress, and applying corrective actions when actual progress does not agree with planned progress. Licensing components involves evaluating candidate components; selecting appropriate components; and negotiating terms, conditions, and delivery dates for the selected components.

This might involve selecting appropriate contracts, such as fixed price, time-and-materials, cost plus fixed fee, and cost-plus incentive fee. Agreements with customers and suppliers typically specify the scope of work and the deliverables. The agreements can also include special clauses, such as clauses establishing penalties for late delivery or nondelivery, and intellectual property agreements that specify what the suppliers are providing and what the acquirer is paying for, plus what will be delivered to and owned by the acquirer. For software developed by suppliers (both those internal to and those external to the software development organization), agreements commonly establish software quality requirements.

After the agreement has been put in place, executing the project in compliance with the terms of the agreement should be managed. (See Chapter 12, *Software Extension to the PMBOK® Guide* (*SWX*), Software Procurement Management, for more information on this topic [2].)

3.3. Implementation of Measurement Process [3, c7]*

The measurement process should be enacted during the software project to ensure that relevant and useful data is collected. (See Sections 6.2, Plan the Measurement Process, and 6.3, Perform the Measurement Process.)

3.4. Monitor Process [3, c8]*

Adherence to the project plan and related plans should be assessed continually and at predetermined intervals. Outputs and completion criteria for each task should also be assessed. Deliverables should be evaluated for their required characteristics (for example, via inspections or by demonstrating working functionality). Effort expenditure, schedule adherence, costs to date, and resource use should be analyzed. The project risk profile (see Section 2.5, Risk

Management) should be revisited, and adherence to software quality requirements should be evaluated (see Software Quality Requirements in the Software Quality KA).

Measurement data should be analyzed. (See Statistical Analysis in the Engineering Foundations KA.) Variance analysis should be conducted to determine deviation of actual from expected outcomes and values. This analysis might examine cost overruns, schedule slippage or other measures. Outlier identification and analysis of quality and other measurement data should be performed (e.g., defect analysis). (See Software Quality Measurement in the Software Quality KA.) Risk exposures should be recalculated. (See Section 2.5, Risk Management.) These activities can enable problem detection and exception identification based on thresholds that have been exceeded. Outcomes should be reported as necessary or when thresholds have been exceeded. For example, the timely identification, mitigation, and resolution of software security vulnerabilities and weaknesses that exceed expectations can affect the system's security posture [11].

3.5. Control Process [3, c7, c8]*

Project monitoring activities provide the basis for making decisions. Where appropriate, and when the probability and impact of risk factors are understood, changes can be made to the project. This may take the form of corrective action (e.g., retesting certain software components). It might involve incorporating additional actions (e.g., deciding to use prototyping to assist in software requirements validation; see Prototyping in the Software Requirements KA). It might also entail revising the project plan and other project documents (e.g., the software requirements specification) to accommodate unanticipated events and their implications.

In some instances, the control process might lead to abandonment of the project. In all cases, the software development team should adhere to software configuration control and software configuration management procedures. (See the Software Configuration Management KA.) Decisions should be documented and communicated to all relevant parties, plans should be revisited and revised when necessary, and relevant data should be recorded. (See Section 6.3, Perform the Measurement Process.)

3.6. Reporting [3, c11]*

Progress to date should be reported at specified and agreed-upon times both within the organization (e.g., to a project steering committee) and to external stakeholders (e.g., clients or users). Reports should focus on the information needs of the target audience as opposed to the detailed status reporting within the project team.

4. Review and Evaluation

At prespecified times and as needed, overall progress toward the stated objectives and satisfaction of stakeholder (user and customer) requirements should be evaluated. Similarly, assessments of the effectiveness of the software process, the personnel involved, and the tools and methods used should also be undertaken regularly and as circumstances demand.

4.1. Determining Satisfaction of Requirements [4, c8]*

Achieving stakeholder satisfaction is a principal goal of the software engineering manager. Progress toward this goal should be assessed periodically. Progress should be assessed upon achieving a major project milestone (e.g., completing software design architecture or completing a software technical review) or upon completion of an

iterative development cycle that results in a product increment. Variances from software requirements should be identified, and appropriate actions should be taken.

As in the control process activity above (see Section 3.5, Control Process), software configuration control and software configuration management procedures should be followed (see the Software Configuration Management KA). Decisions should be documented and communicated to all relevant parties; plans should be revisited and revised as necessary; and relevant data should be recorded (see Section 6.3, Perform the Measurement Process).

4.2. Reviewing and Evaluating Performance [3, c8, c10]*

Periodic performance reviews for project personnel can provide insight into the likelihood of adherence to plans and processes and possible areas of difficulty (e.g., team member conflicts). The various project methods, tools and techniques should be evaluated for effectiveness and appropriateness. The project's process should also be systematically and periodically assessed for relevance, utility and efficacy. Where appropriate, project changes should be made and managed.

5. Closure

An entire project, a major project phase or an iterative development cycle reaches closure when all the plans and processes have been enacted and completed. The criteria for project, phase or iteration success should then be evaluated. Once closure has been established, archival, retrospective and process improvement activities can be performed.

5.1. Determining Closure [1, s3.7, s4.6]

Closure occurs when the specified tasks for a project, a phase or an iteration have been completed and satisfactory achievement of the completion criteria has been confirmed. Software requirements can be confirmed as satisfied or not, and the degree of achieving the objectives can be determined. Closure processes should involve relevant stakeholders and document relevant stakeholders' acceptance; any known problems should be documented.

5.2. Closure Activities [2, s3.7, s4.8]

After closure has been confirmed, project materials should be archived in accordance with stakeholder agreed-upon rules for archival methods, location and duration — possibly including destruction of sensitive information, software and the medium on which copies are resident. For example, these rules could require that during closure, all data is removed and destroyed from any devices that contain relevant information before physical disposal of the devices (e.g., the hard drives of personal computers, servers, mainframes, personal digital assistants (PDAs), routers, firewalls, switches, tapes, diskettes, CDs, DVDs, cell phones, printers, Universal Serial Bus (USB) data storage devices).

The organization's measurement database should be updated with relevant project data. A project, phase or iteration retrospective analysis should be undertaken so that issues, problems, risks and opportunities encountered can be analyzed. (See Topic 4, Review and Evaluation.) Lessons learned should be drawn from the project and fed into organizational learning and improvement endeavors.

6. Software Engineering Measurement

The importance of software engineering measurement for good management and

engineering practices is widely acknowledged. (See Measurement in the Engineering Foundations KA.) Effective software engineering measurement has become one of the cornerstones of organizational maturity. Measurement can be applied to organizations, projects, processes and work products. This section focuses on applying measurement at the levels of projects, processes and work products.

This section follows the Institute of Electrical and Electronics Engineers (IEEE) 15939:2008 standard [6], which describes a process to define the activities and tasks necessary to implement a software measurement process. The standard also includes a measurement information model. This model in the PSM Continuous Iterative Development Measurement Framework report is also elaborated for SDLC approaches [9].

6.1. Establish and Sustain Measurement Commitment [7, c1, c2]*[1]

- Establish measurement requirements. Each measurement endeavor should be guided by organizational objectives and driven by a set of measurement requirements established by the organization and the project (e.g., an organizational objective might be first to market).
- Establish the scope of measurement. The project team should establish the organizational unit to which each measurement requirement is to be applied. This might be a functional area, a single project, a single site or an entire enterprise. The temporal scope of the measurement effort should also be considered because the time series of some measurements might be required (e.g., to calibrate estimation models). (See Section 2.3, Effort, Schedule and Cost Estimation.)
- Establish the team's commitment to measurement. The commitment should be formally established, communicated and supported by resources.
- Commit measurement resources. An organization's commitment to measurement is an essential factor for success, as evidenced by the assignment of resources for implementing the measurement process. Assigning resources includes allocation of responsibility for the various tasks of the measurement process (such as analyst and librarian). Adequate funding, training, tools and support to conduct the process should also be allocated.

6.2. Plan the Measurement Process [7, c1, c2]*[1]

- Characterize the organizational unit. The organizational unit provides the context for measurement, so the organizational context should be explicit, including the organization's constraints on the measurement process. The characterization can be stated in terms of organizational processes, application domains, technology, organizational interfaces and organizational structure.

- Identify information needs. Information needs are based on the organizational unit's goals, constraints, risks, and problems and may be derived from business, organizational, regulatory and/or

[1] These two chapters can be downloaded free of charge from http://www.psmsc.com/PSMBook.asp.

product objectives. Stakeholders should identify, prioritize, document, communicate and review these needs.

- Select measures. Select candidate measures, with clear links to the information needs. Select measures based on the priorities of the information needs and other criteria such as cost of collection; degree of process disruption during collection; ease of obtaining accurate, consistent data; and ease of analysis and reporting. Internal quality characteristics (see Models and Quality Characteristics in the Software Quality KA) are often not contained in the contractually binding software requirements. Therefore, consider measuring the software's internal quality to provide an early indicator of potential issues that might affect external stakeholders.

- Define data collection, analysis and reporting procedures. This encompasses collection procedures and schedules, storage, verification, analysis, reporting and data configuration management.

- Select criteria for evaluating the information products. The organizational unit's technical and business objectives influence evaluation criteria. Information products include those associated with the product produced and those associated with the processes used to manage and measure the project.

- Provide resources for measurement tasks. The appropriate stakeholders should review and approve the measurement plan to include all data collection procedures; storage, analysis, and reporting procedures; evaluation criteria; schedules; and responsibilities. Criteria for reviewing these artifacts should be established at the organizational unit level or higher and should be used as the basis for these reviews. Such criteria should consider experience, resource availability and potential disruptions to projects when changes from current practices are proposed. Approval demonstrates commitment to the measurement process.

    - Identify resources to be made available for implementing the planned and approved measurement tasks. Resource availability may be staged in cases where changes are piloted before widespread deployment. Consider the resources necessary for successful deployment of new procedures or measures.
    - Acquire and deploy supporting technologies. This includes evaluating available supporting technologies, selecting the most appropriate technologies, acquiring those technologies and deploying those technologies.

6.3. Perform the Measurement Process [7, c1, c2]*

Integrate measurement procedures with relevant software processes. The measurement procedures, such as data collection, should be integrated into the software processes they measure. This might involve changing current software processes to accommodate data collection or generation activities. It might also involve analyzing current software processes to minimize additional effort and evaluating the effect on

employees to ensure acceptance of the measurement procedures. Consider morale issues and other human factors. In addition, communicate the measurement procedures to those providing the data. Training and support might also be needed. Data analysis and reporting procedures are typically integrated similarly into organizational and project processes.

Collect data. Data should be collected, verified and stored. Collection can sometimes be automated by using SEM tools (see Topic 7, Software Engineering Management Tools) to analyze data and develop reports. Data may be aggregated, transformed or recorded as part of the analysis, using a degree of rigor appropriate to the nature of the data and the information needs. This analysis typically produces graphs, numbers or other indicators that inform conclusions and recommendations to present to stakeholders. (See Statistical Analysis in the Engineering Foundations KA.) The results and conclusions are reviewed using the organization's formal or informal process. Data providers and measurement users should participate in reviewing the data to ensure it is meaningful and accurate and can result in reasonable actions.

Communicate results. Document and communicate information products to users and stakeholders.

6.4. Evaluate Measurement [7, c1, c2]*

Evaluate information products and the measurement process against specified evaluation criteria, and determine the strengths and weaknesses of the information products or process. An internal process or an external audit can be used to perform the evaluation, including feedback from measurement users. Record lessons learned in an appropriate database.

Identify potential improvements. Such improvements might be changes in the format of indicators, changes in units measured or reclassification of measurement categories. Determine potential improvements' costs and benefits, and report appropriate improvement actions.

Communicate proposed improvements to the measurement process owner and stakeholders for review and approval. Also, communicate the lack of potential improvements if the analysis fails to identify any.

7. Software Engineering Management Tools [3*, c5, c6, c7]

SEM tools are often used to provide visibility and control of SEM processes. Some tools are automated, whereas others are manually implemented. In addition, there has been a recent trend toward using integrated suites of software engineering tools throughout a project to plan, collect and record, monitor and control, and report project and product information. Tools can be divided into the following categories:

Project planning and tracking tools. Project planning and tracking tools can be used to estimate project effort and cost and to prepare project schedules. For example, some projects use automated estimation tools that use a software product's estimated size and other characteristics as input and then estimate the required total effort, schedule and cost. Planning tools also include automated scheduling tools that analyze the WBS tasks, their estimated durations, their precedence relationships and the resources assigned to each task to produce a Gantt chart.

Tracking tools can be used to track project milestones, regularly scheduled project status

meetings, scheduled iteration cycles, product demonstrations and action items.

Risk management tools. Risk management tools (see Section 2.5, Risk Management*)* can be used to track risk identification, analysis and monitoring. These tools include simulation or decision trees to analyze the effect of costs versus payoffs and subjective estimates of the probabilities of risk events. For example, Monte Carlo simulation tools can be used to produce probability distributions of effort, schedule and risk by algorithmically combining multiple input probability distributions.

Communication tools. Communication tools can help provide timely and consistent information to relevant stakeholders involved in a project. Examples of such tools are email notifications and broadcasts to team members and stakeholders; regular communications of meeting minutes; and charts showing progress, backlogs, and maintenance request resolutions.

Measurement tools. Measurement tools support activities related to the software measurement program. (See Topic 6, Software Engineering Measurement.) There are few completely automated tools in this category. Measurement tools to gather, analyze and report project measurement data may be based on spreadsheets developed by project team members or organizational employees.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Fairley 2009 [3*]	Sommerville 2011 [4*]	Boehm and Turner 2003 [5*]	McGarry et al. 2001 [7*]
1. Initiation and Scope Definition				
1.1. Determination and Negotiation of Requirements	c3			
1.2. Feasibility Analysis		c4		
1.3. Process for the Review and Revision of Requirements	c3			
2. Software Project Planning				
2.1. Process Planning	c2, c3, c4, c5		c1	
2.2. Determine Deliverables	c4, c5, c6			
2.3. Effort, Schedule and Cost Estimation	c6			
2.4. Resource Allocation	c5, c10, c11			
2.5. Risk Management	c9		c5	
2.6. Quality Management	c4	c24		
2.7. Plan Management	c4			
3. Software Project Enactment				
3.1. Implementation of Plans		c2		
3.2. Software Acquisition and Supplier Contract Management	c3, c4			
3.3. Implementation of Measurement Process	c7			
3.4. Monitor Process	c8			
3.5. Control Process	c7, c8			
3.6. Reporting	c11			
4. Review and Evaluation				
4.1. Determining Satisfaction of Requirements				
4.2. Reviewing and Evaluating Performance	c8, c10			
5. Closure				
5.1. Determining Closure				
5.2. Closure Activities				
6. Software Engineering Measurement				
6.1. Establish and Sustain Measurement Commitment				c1, c2
6.2. Plan the Measurement Process				c1, c2
6.3. Perform the Measurement Process				c1, c2
6.4. Evaluate Measurement				c1, c2
7. Software Engineering Management Tools	c5, c6, c7			

FURTHER READING

A Guide to the Project Management Body of Knowledge (*PMBOK® Guide*) [1].

The *PMBOK® Guide* provides guidelines for managing individual projects and defines project management-related concepts. It also describes the project management life cycle and its related processes, and the project life cycle. It is a globally recognized guide for the project management profession.

Software Extension to the Project Management Body of Knowledge (*PMBOK®*) *Guide* [2].

SWX provides adaptations of and extensions to the generic practices of project management documented in the *PMBOK® Guide* for managing software projects. The primary contribution of this extension to the *PMBOK® Guide* is a description of processes for managing adaptive life cycle software projects.

IEEE Standard Adoption of ISO/IEC 15939 [6].

This international standard identifies a process that supports defining suitable measures to address specific information needs. It identifies the activities and tasks necessary to successfully identify, define, select, apply and improve measurement within an overall project or organizational measurement structure.

J. McDonald, *Managing the Development of Software Intensive Systems*, Wiley, 2010 [8].

This textbook introduces project management for beginning software and hardware developers, plus unique advanced material for experienced project managers. Case studies are included for planning and managing verification and validation for large software projects and complex software and hardware systems, as well as inspection results and testing metrics to monitor project status.

REFERENCES

[1] Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*, 6th ed., Newton Square, PA: Project Management Institute, 2017.

[2] *Software Extension to the Project Management Body of Knowledge (PMBOK® Guide)*, Fifth Edition, Project Management Institute, 2013.

[3*] R. E. Fairley, *Managing and Leading Software Projects*. Hoboken, NJ: Wiley IEEE Computer Society Press, 2009.

[4*] I. Sommerville, *Software Engineering*, 10th ed., New York: Addison-Wesley, 2015.

[5*] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley, 2003.

[6] IEEE, *IEEE Standard Adoption of ISO/IEC 15939: 2007 Systems and Software Engineering Measurement Process*, ed: IEEE, 2017.

[7*] J. McGarry, et al., *Practical Software Measurement: Objective Information for Decision Makers*, Addison-Wesley Professional, 2001.

[8] J. McDonald, *Managing the Development of Software-Intensive Systems*. Hoboken, NJ: John Wiley and Sons, Inc., 2010.

[9] *Practical Software and Systems Measurement Continuous Iterative Development Measurement Framework Parts 1-3: Concepts, Definitions, Principles, and Measures*, Version 2.1, April 15, 2021.

[10] Dr. Sarah Sheard, Mickael Bouyaud, Macaulay Osaisai, Jeannine Siviy, and Dr. Ken Nidiffer. *"Book Club" Guides a Working Group to Create INCOSE System-Software Interface Products*, *INSIGHT*, Volume 24, Issue 2, 2021.

[11] Dr. Kenneth E. Nidiffer, Dr. Carol Woody, Timothy A. Chick, *Program Manager's Guidebook for Software Assurance, Special Report*, CMU/SEI-2018-SR-025, Software Solutions and CERT Divisions, Software Engineering Institute/Carnegie Mellon University, August 2018.

[12] Dr. R. E. Fairley, *Systems Engineering of Software-Enabled Systems*, ISBN 978-1-119-53501-0, 2019.

[13] Defense Innovation Board, *Software Is Never Done: Refactoring the Acquisition Code for Competitive Advantage Defense*, v3.3, March 12, 2019.

[14] "DevOps: Building Reliable and Secure Systems Including Application Build, Package, and Deployment," *IEEE Standard*, 2675-2021, 2021.

[15] Murali Chemuturi and Thomas Cagley, *Mastering Software Project Management: Best Practices, Tools and Techniques*, J. Ross Publishing, July 2010.

CHAPTER 10

SOFTWARE QUALITY

ACRONYMS

CMMI	Capability Maturity Model Integration
CoSQ	Cost of Software Quality
COTS	Commercial Off-the-Shelf Software
FMEA	Failure Mode and Effects Analysis
FTA	Fault Tree Analysis
PDCA	Plan-Do-Check-Act
PDSA	Plan-Do-Study-Act
QFD	Quality Function Deployment
SPI	Software Process Improvement
SQA	Software Quality Assurance
SQC	Software Quality Control
SQM	Software Quality Management
TQM	Total Quality Management
V&V	Verification and Validation

INTRODUCTION

What is software quality, and why is it so important that it is included in many knowledge areas (KAs) of the *SWEBOK Guide*?

One reason is that the term *software quality* is overloaded. Software quality may refer: to desirable characteristics of software products, to the extent to which a particular software product possess those characteristics, and to processes, tools, and techniques used to achieve those characteristics. Over the years, authors and organizations have defined the term quality differently. To Phil Crosby, it was "conformance to requirements" [1]. Watts Humphrey refers to it as "achieving excellent levels of "fitness for use" [2]. Meanwhile, IBM coined the phrase "market-driven quality," where the "customer is the final arbiter" [3*, p8].

More recently, software quality is defined as the "capability of software product to satisfy stated and implied needs under specified conditions" [4] and as "the degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations" [5]. Both definitions embrace the premise of conformance to requirements. Neither refers to types of requirements (e.g., functional, reliability, performance, dependability, or any other characteristic). Significantly, however, these definitions emphasize that quality is dependent upon requirements.

These definitions also illustrate another reason for the prevalence of software quality throughout this *Guide*: a frequent ambiguity of *software quality* versus *software quality requirements* ("the *-ilities*" is a common shorthand). Software quality requirements are actually attributes of (or constraints on) functional requirements (what the system does). Software requirements may also specify resource usage, a communication protocol, or many other characteristics. This KA attempts clarity by using *software quality* in the broadest sense from the definitions above and by using *software quality requirements* as constraints on functional requirements. Software quality is achieved by conformance to all requirements regardless of what characteristic is specified or how requirements are grouped or named.

Software quality is also considered in many of the SWEBOK KAs because it is a basic parameter of a software engineering effort. For all engineered products, the primary goal is delivering maximum stakeholder value, while balancing the constraints of development cost and schedule; this is sometimes characterized as "fitness for

Figure 10.1. Breakdown of Topics for the Software Quality KA

use." Stakeholder value is expressed in requirements. For software products, stakeholders could value price (what they pay for the product), lead time (how fast they get the product), and software quality.

This KA addresses definitions and provides an overview of practices, tools, and techniques for defining software quality and for appraising the state of software quality during development, maintenance, and deployment. Cited references provide additional details.

BREAKDOWN OF TOPICS FOR SOFTWARE QUALITY

The breakdown of topics for the Software Quality KA is presented in Figure 10.1.

1. Software Quality Fundamentals

Reaching agreement on what constitutes quality for all stakeholders and clearly communicating that agreement to software engineers require that the many aspects of quality be formally defined and discussed.

A software engineer should understand quality concepts, characteristics, values, and their application to the software under development or maintenance. The important concept is that the software requirements define the required quality attributes of the software. Software requirements influence the measurement methods and acceptance criteria for assessing the degree to which the software and related documentation achieve the desired quality levels.

1.1. Software Engineering Culture and Ethics

[3*, c1s4] [6*, c2s3.5]

Software engineers are expected to share a commitment to software quality as part of their culture. A healthy software engineering culture includes many characteristics, including the understanding that tradeoffs among cost, schedule, and quality are a basic tenant of the engineering of any product. A strong software engineering ethic assumes

that engineers accurately report information, conditions, and outcomes related to quality.

Ethics also play a significant role in software quality, the culture, and the attitudes of software engineers. The IEEE Computer Society and the ACM have developed a code of ethics and professional practice (see Codes of Ethics and Professional Conduct in the Software Engineering Professional Practice KA).

1.2. Value and Costs of Quality [7*, c17, c22]

Defining and then achieving software quality is not simple. Quality characteristics may or may not be required, or they may be required to a greater or lesser degree, and tradeoffs may be made among them. To help determine the level of software quality, i.e., achieving stakeholder value, this section presents cost of software quality (CoSQ): a set of measurements derived from the economic assessment of software quality development and maintenance processes. The CoSQ measurements are examples of process measurements that may be used to infer characteristics of a product.

The premise underlying the CoSQ is that the level of quality in a software product can be inferred from the cost of activities related to dealing with the consequences of poor quality. Poor quality means that the software product does not fully "satisfy stated and implied needs" or "established requirements." There are four cost of quality categories: prevention, appraisal, internal failure, and external failure.

Prevention costs include investments in software process improvement efforts, quality infrastructure, quality tools, training, audits, and management reviews. These costs are usually not specific to a project; they span the organization. Appraisal costs arise from project activities that find defects. These appraisal activities can be categorized into costs of reviews (design, peer) and costs of testing (software unit testing, software integration, system level testing, acceptance testing); appraisal costs would be extended to subcontracted software suppliers. Costs of internal failures are those that are incurred to fix defects found during appraisal activities and discovered prior to delivery of the software product to the customer. External failure costs include activities to respond to software problems discovered after delivery to the customer.

Software engineers should be able to use CoSQ methods to ascertain levels of software quality and should also be able to present quality alternatives and their costs so that tradeoffs between cost, schedule, and delivery of stakeholder value can be made.

1.3. Models and Quality Characteristics [3*, c24s1] [7*, c2s4] [8*, c17]

Terminology for software quality characteristics differs from one taxonomy (or model of software quality) to another, each model perhaps having a different number of hierarchical levels and a different total number of characteristics. Various authors have produced models of software quality characteristics or attributes that can be useful for discussing, planning, and rating the quality of software products. ISO/IEC 25010: 2011 [4] defines product quality and quality in use as two related quality models. Appendix B in the *SWEBOK Guide* provides a list of applicable standards for each KA. Standards for this KA cover various ways of characterizing software quality.

1.3.1. Software Process Quality

Software quality management and software engineering process quality have a direct bearing on the quality of the software product.

Models and criteria that evaluate the capabilities of software organizations are primarily project organization and management considerations and, as such, are covered in the Software Engineering Management and Software Engineering Process KAs.

It is not possible to completely distinguish process quality from product quality because process outcomes include products. Determining whether a process has the capability to consistently produce products of desired quality is not simple.

The software engineering process, discussed in the Software Engineering Process KA, influences the quality characteristics of software products, which in turn affect quality as perceived by stakeholders.

1.3.2. Software Product Quality

The software engineer, first of all, must determine the real purpose of the software. In this regard, stakeholder requirements are paramount, and they include quality requirements in addition to functional requirements. Thus, software engineers have a responsibility to elicit quality requirements that may not be explicit at the outset and to understand their importance as well as the level of difficulty in attaining them. All software development processes (e.g., eliciting requirements, designing, constructing, building, checking, improving quality) are designed with these quality requirements in mind and may carry additional development costs if attributes such as safety, security, and dependability are important. The additional development costs help ensure that quality obtained can be traded off against the anticipated benefits.

The term work-product means any artifact that is the outcome of a process used to create the final software product. Examples of a work-product include a system/subsystem specification, a software requirements specification for a software component of a system, a software design description, source code, software test documentation, or reports. While some treatments of quality are described in terms of final software and system performance, sound engineering practice requires that intermediate work-products relevant to quality be evaluated throughout the software engineering process.

1.4. Software Quality Improvement

[3*, c1s4] [9*, c24] [10*, c11s2.4]

The quality of software products can be improved through preventative processes or an iterative process of continual improvement, which requires management control, coordination, and feedback from many concurrent processes: (1) the software life cycle processes, (2) the process of fault/defect detection, removal, and prevention, and (3) the quality improvement process.

The theory and concepts behind quality improvement—such as building in quality through the prevention and early detection of defects, continual improvement, and stakeholder focus—are pertinent to software engineering. These concepts are based on the work of experts in quality who have stated that the quality of a product is directly linked to the quality of the process used to create it. Approaches such as the Deming improvement cycle of Plan-Do-Check-Act (PDCA), evolutionary delivery, kaizen, and quality function deployment (QFD) offer techniques to specify quality objectives and determine whether they are met. The Software Engineering Institute's IDEAL is another method [7*]. Quality management is now recognized by the *SWEBOK Guide* as an important discipline.

Management sponsorship supports process and product evaluations and the resulting findings. Then an improvement program is developed identifying detailed actions and improvement projects to be addressed in a feasible time frame. Management support implies that each improvement project has enough resources to achieve the goal defined for it. Management sponsorship is solicited frequently by implementing proactive communication activities.

1.5. Software Safety

[9*, c11s3]

Safety-critical systems are those in which a system failure could harm human life, other living things, physical structures, or the environment. The software in these systems is safety-critical. There are increasing numbers of applications of safety-critical software in a growing number of industries. Examples of systems with safety-critical software include mass transit systems, chemical manufacturing plants, and medical devices. The failure of software in these systems could have catastrophic effects. There are industry standards, such as DO-178C [11], and emerging processes, tools, and techniques for developing safetycritical software. The intent of these standards, tools, and techniques is to reduce the risk of injecting faults into the software and thus improve software reliability.

Safety-critical software can be categorized as direct or indirect. Direct is that software embedded in a safety-critical system, such as the flight control computer of an aircraft. Indirect includes software applications used to develop safety-critical software. Indirect software is included in software engineering environments and software test environments.

Three complementary techniques for reducing the risk of failure are avoidance, detection and removal, and damage limitation. These techniques impact software functional requirements, software performance requirements, and development processes. Increasing levels of risk imply increasing levels of software quality assurance and control techniques such as inspections. Higher risk levels may necessitate more thorough inspections of requirements, design, and code or the use of more formal analytical techniques. Another technique for managing and controlling software risk is building assurance cases. An assurance case is a reasoned, auditable artifact created to support the contention that its claim or claims are satisfied. It contains the following and their relationships: one or more claims about properties; arguments that logically link the evidence and any assumptions to the claims; and a body of evidence and assumptions supporting these arguments [12].

2. Software Quality Management Processes

Software quality management is the collection of all processes that ensure that software products, services, and life cycle process implementations meet organizational software quality objectives and achieve stakeholder satisfaction [13, 14]. SQM defines processes, process owners, requirements for the processes, measurements of the processes and their outputs, and feedback channels throughout the whole software life cycle.

SQM comprises four subcategories: software quality planning, software quality assurance (SQA), software quality control (SQC), and software process improvement (SPI). Software quality planning includes determining which quality standards are to be used, defining specific quality goals, and estimating the effort and schedule of software quality activities. In some cases, software quality planning also includes defining the software quality processes to be used. SQA activities define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes [5]. SQC activities examine specific project artifacts (documents and executables) to determine whether they comply with standards established for the project (including requirements, constraints, designs, contracts, and plans). SQC evaluates intermediate products as well as the final products.

The fourth SQM category dealing with improvement has various names within the software industry, including SPI, software quality improvement, and software corrective and preventive action. The activities in this category seek to improve process effectiveness, efficiency, and other characteristics with the ultimate goal of improving software quality. Although SPI could be included in any of the first three categories, an increasing number of organizations organize SPI into a separate category that may span across many projects (see the Software Engineering Process KA).

Software quality processes consist of tasks and techniques to indicate how software plans (e.g., software management, development, quality management, or configuration management plans) are being implemented and how well the intermediate and final products are meeting their specified requirements. Results from these tasks are assembled in reports for management before corrective action is taken. The management of an SQM process is tasked with ensuring that the results of these reports are accurate.

Risk management can also play an important role in delivering quality software. Incorporating disciplined risk analysis and management techniques into the software life cycle processes can help improve product quality (see the Software Engineering Management KA for related material on risk management).

2.1. Software Quality Assurance

[7*, c4–c6, c11, c12, c26–27]

To quell a widespread misunderstanding, software quality assurance is not testing. software quality assurance (SQA) is a set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate and produce software products of suitable quality for their intended purposes. A key attribute of SQA is the objectivity of the SQA function with respect to the project. The SQA function may also be organizationally independent of the project; that is, free from technical, managerial, and

financial pressures from the project [5]. SQA has two aspects: product assurance and process assurance, which are explained in section 2.3.

The software quality plan (in some industry sectors it is termed the software quality assurance plan) defines the activities and tasks employed to ensure that software developed for a specific product satisfies the project's established requirements and user needs within project cost and schedule constraints and is commensurate with project risks. The SQAP first ensures that quality targets are clearly defined and understood.

The SQA plan's quality activities and tasks are specified with their costs, resource requirements, objectives, and schedule in relation to related objectives in the software engineering management, software development, and software maintenance plans. The SQA plan should be consistent with the software configuration management plan (see the Software Configuration Management KA). The SQA plan identifies documents, standards, practices, and conventions governing the project and how these items are checked and monitored to ensure adequacy and compliance. The SQA plan also identifies measures; statistical techniques; procedures for problem reporting and corrective action; resources such as tools, techniques, and methodologies; security for physical media; training; and SQA reporting and documentation. Moreover, the SQA plan addresses the software quality assurance activities of any other type of activity described in the software plans—such as procurement of supplier software for the project, commercial off-the-shelf software (COTS) installation, and service after delivery of the software. It can also contain acceptance criteria as well as reporting and management activities that are critical to software quality.

2.2. Verification & Validation

[9*, c2s2.3, c8, c15s1.1, c21s3.3]

As stated in [15],

> The purpose of V&V is to help the development organization build quality into the system during the life cycle. V&V processes provide an objective assessment of products and processes throughout the life cycle. This assessment demonstrates whether the requirements are correct, complete, accurate, consistent, and testable. The V&V processes determine whether the development products of a given activity conform to the requirements of that activity and whether the product satisfies its intended use and user needs.

Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Validation is an attempt to ensure that the right product is built—that is, the product fulfills its specific intended purpose. Both the verification process and the validation process begin early in the development or maintenance phase. They provide an examination of key product features in relation to both the product's immediate predecessor and the specifications to be met.

The purpose of planning V&V is to ensure that each resource, role, and responsibility is clearly assigned. The resulting V&V plan documents describe the various resources and their roles and activities, as well as the techniques and tools to be used. An understanding of the different purposes of each V&V activity helps in the careful planning of the techniques and resources needed to fulfill their purposes. The plan also addresses the management, communication, policies, and procedures of the V&V activities and their interaction, as well as defect reporting and documentation requirements.

2.3. Reviews and Audits

[9*, c24s3] [16*]

Reviews and audit processes are broadly defined as static—meaning that no software programs or models are executed—examination of software engineering artifacts with respect to standards that have been established by the organization or project for those artifacts. Different types of reviews and audits are distinguished by their purpose, levels of independence, tools and techniques, roles, and by the subject of the activity. Product assurance and process assurance audits are typically conducted by software quality assurance (SQA) personnel who are independent of development

teams. Management reviews are conducted by organizational or project management. The engineering staff conducts technical reviews.

- Management reviews evaluate actual project results with respect to plans.
- Technical reviews (including inspections, walkthrough, and desk checking) examine engineering work-products.
- Process assurance audits. SQA process assurance activities make certain that the processes used to develop, install, operate, and maintain software conform to contracts, comply with any imposed laws, rules, and regulations and are adequate, efficient and effective for their intended purpose [5].
- Product assurance audits. SQA product assurance activities make certain to provide evidence that software products and related documentation are identified in and comply with contracts; and ensure that nonconformances are identified and addressed [5].

2.3.1. Management Reviews

As stated in [16*],

> The purpose of a management review is to monitor progress, determine the status of plans and schedules, and evaluate the effectiveness of management processes, tools and techniques. Management reviews compare actual project results against plans to determine the status of projects or maintenance efforts. The main parameters of management reviews are project cost, schedule, scope, and quality. Management reviews evaluate decisions about corrective actions, changes in the allocation of resources, or changes to the scope of the project.

Inputs to management reviews may include audit reports, progress reports, V&V reports, and plans of many types, including risk management, project management, software configuration management, software safety, and risk assessment, among others. (Refer to the Software Engineering Management and the Software Configuration Management KAs for related material.)

2.3.2. Technical Reviews

As stated in [16*],

> The purpose of a technical review is to evaluate a software product by a team of qualified personnel to determine its suitability for its intended use and identify discrepancies from specifications and standards. It provides management with evidence to confirm the technical status of the project.

Although any work-product can be reviewed, technical reviews are performed on the main software engineering work-products of software requirements and software design.

Purpose, roles, activities, and most importantly the level of formality distinguish different types of technical reviews. Inspections are the most formal, walkthroughs less, and pair reviews or desk checks are the least formal.

Examples of specific roles include a decision maker (i.e., software lead), a review leader, a recorder, and checkers (technical staff members who examine the work-products). Reviews are also distinguished by whether meetings (face to face or electronic) are included in the process. In some review methods checkers solitarily examine work-products and send their results back to a coordinator. In other methods checkers work cooperatively in meetings. A technical review may require that mandatory inputs be in place in order to proceed:

- Statement of objectives
- Specific software product
- Specific project management plan
- Issues list associated with this product
- Technical review procedure.

The team follows the documented review procedure. The technical review is completed once all the activities listed in the examination have been completed.

Technical reviews of source code may include a wide variety of concerns such as analysis of algorithms, utilization of critical computer resources, adherence to coding standards, structure and

organization of code for testability, and safety-critical considerations.

Note that technical reviews of source code or design models such as UML are also termed static analysis (see topic 3, Practical Considerations).

2.3.3. Inspections

"The purpose of an inspection is to detect and identify software product anomalies" [16*]. Some important differentiators of inspections as compared to other types of technical reviews are these:

1. Rules. Inspections are based upon examining a work-product with respect to a defined set of criteria specified by the organization. Sets of rules can be defined for different types of workproducts (e.g., rules for requirements, architecture descriptions, source code).
2. Sampling. Rather that attempt to examine every word and figure in a document, the inspection process allows checkers to evaluate defined subsets (samples) of the documents under review.
3. Peer. Individuals holding management positions over members of the inspection team do not participate in the inspection. This is a key distinction between peer review and management review.
4. Led. An impartial moderator who is trained in inspection techniques leads inspection meetings.
5. Meeting. The inspection process includes meetings (face to face or electronic) conducted by a moderator according to a formal procedure in which inspection team members report the anomalies they have found and other issues.

Software inspections always involve the author of an intermediate or final product; other reviews might not. Inspections also include an inspection leader, a recorder, a reader, and a few (two to five) checkers (inspectors). The members of an inspection team may possess different expertise, such as domain expertise, software design method expertise, or programming language expertise. Inspections are usually conducted on one relatively small section of the product at a time (samples). Each team member examines the software product and other review inputs prior to the review meeting, perhaps by applying an analytical technique (see section 3.3.3) to a small section of the product or to the entire product with a focus on only one aspect—e.g., interfaces. During the inspection, the moderator conducts the session and verifies that everyone has prepared for the inspection and conducts the session. The inspection recorder documents anomalies found. A set of rules, with criteria and questions germane to the issues of interest, is a common tool used in inspections. The resulting list often classifies the anomalies (see section 3.2, Defect Characterization) and is reviewed for completeness and accuracy by the team. The inspection exit decision corresponds to one of the following options:

1. Accept with no or, at most, minor reworking
2. Accept with rework verification
3. Reinspect.

2.3.4. Walkthroughs

As stated in [16*],

> The purpose of a systematic walk-through is to evaluate a software product. A walk-through may be conducted for the purpose of educating an audience regarding a software product.

Walkthroughs are distinguished from inspections. The main difference is that the author presents the work-product to the other participants in a meeting (face to face or electronic). Unlike an inspection, the meeting participants may not have necessarily seen the material prior to the meeting. The meetings may be conducted less formally. The author takes the role of explaining and showing the material to participants and solicits feedback. Like inspections, walkthroughs may be conducted on any type of work-product including project plan, requirements, design, source code, and test reports.

2.3.5. Process Assurance and Product Assurance Audits

As stated in [16*],

> The purpose of a software audit is to provide an independent evaluation of the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures.

Process assurance audits determine the adequacy of plans, schedules, and requirements to achieve project objectives [5]. The audit is a formally organized activity with participants having specific roles—such as lead auditor, another auditor, a recorder, or an initiator—and including a representative of the audited organization. Audits identify instances of nonconformance and produce a report requiring the team to take corrective action.

While there may be many formal names for reviews and audits, such as those identified in the standard [16*], the important point is that they can occur on almost any product at any stage of the development or maintenance process.

3. Practical Considerations

3.1. Software Quality Requirements

[9*, c11s1] [18*, c12]
[17*, c15s3.2.2, c15s3.3.1, c16s9.10]

3.1.1. Influence Factors

Various factors influence planning, management, and selection of SQM activities and techniques, including

- the domain of the system in which the software resides; the system functions could be safety-critical, mission-critical, business-critical, security-critical
- the physical environment in which the software system resides
- system and software functional (what the system does) and quality (how well the system performs its functions) requirements
- the commercial (external) or standard (internal) components to be used in the system
- the specific software engineering standards applicable
- the methods and software tools to be used for development and maintenance and for quality evaluation and improvement
- the budget, staff, project organization, plans, and scheduling of all processes
- the intended users and use of the system
- the integrity level of the system.

Information on these factors influences how the SQM processes are organized and documented, how specific SQM activities are selected, what resources are needed, and which of those resources impose bounds on the efforts.

3.1.2. Dependability

In cases where system failure may have extremely severe consequences, overall dependability (hardware, software, and human or operational) is the main quality requirement over and above basic functionality. This is the case for the following reasons: system failures affect a large number of people; users often reject systems that are unreliable, unsafe, or insecure; system failure costs may be enormous; and undependable systems may cause information loss. System and software dependability include such characteristics as availability, reliability, safety, and security. When developing dependable software, tools and techniques can be applied to reduce the risk of injecting faults into the intermediate deliverables or the final software product. Verification, validation, and testing processes, techniques, methods, and tools identify faults that impact dependability as early as possible in the life cycle. Additionally, mechanisms may need to be in place in the software to guard against external attacks and to tolerate faults.

3.1.3. Integrity Levels of Software

Defining integrity levels is a method of risk management.

> Software integrity levels are a range of values that represent software complexity, criticality, risk, safety level, security level,

desired performance, reliability, or other project-unique characteristics that define the importance of the software to the user and acquirer. The characteristics used to determine software integrity level vary depending on the intended application and use of the system. The software is a part of the system, and its integrity level is to be determined as a part of that system.

The assigned software integrity levels may change as the software evolves. Design, coding, procedural, and technology features implemented in the system or software can raise or lower the assigned software integrity levels. The software integrity levels established for a project result from agreements among the acquirer, supplier, developer, and independent assurance authorities. A software integrity level scheme is a tool used in determining software integrity levels. [5]

As noted in [17*], "the integrity levels can be applied during development to allocate additional verification and validation efforts to high-integrity components."

3.2. Defect Characterization

[3*, c3s3, c8s8, c10s2]

Software quality evaluation (i.e., software quality control) techniques find defects, faults and failures. Characterizing these techniques leads to an understanding of the product, facilitates corrections to the process or the product, and informs management and other stakeholders of the status of the process or product. Many taxonomies exist and, while attempts have been made to gain consensus, the literature indicates that there are quite a few in use. Defect characterization is also used in audits and reviews, with the review leader often presenting a list of issues provided by team members for consideration at a review meeting.

As new design methods and languages evolve, along with advances in overall software technologies, new classes of defects appear, and a great deal of effort is required to interpret previously defined classes. When tracking defects, the software engineer is interested in not only the number of defects but also the types. Information alone, without some classification, may not be sufficient to identify the underlying causes of the defects. Specific types of problems need to be grouped to identify trends over time. The point is to establish a defect taxonomy that is meaningful to the organization and to software engineers.

Software quality control activities discover information at all stages of software development and maintenance. In some cases, the word *defect* is overloaded to refer to different types of anomalies. However, different engineering cultures and standards may use somewhat different meanings for these terms. The variety of terms prompts this section to provide a widely used set of definitions [19]:

- *Computational Error*: "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition."
- *Error*: "A human action that produces an incorrect result." A slip or mistake that a person makes. Also called human error.
- *Defect*: An "imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced." A defect is caused by a person committing an error.
- *Fault*: A defect in source code. An "incorrect step, process, or data definition in computer program." The encoding of a human error in source code. Fault is the formal name of a bug.
- *Failure*: An "event in which a system or system component does not perform a required function within specified limits." A failure is produced when a fault is encountered by the processor under specified conditions.

Using these definitions three widely used software quality measurements are defect density (number of defects per unit size of documents), fault density (number of faults per 1K lines of code), and failure intensity (failures per use-hour or per test-hour). Reliability models are built from failure data collected during software testing or from software in service and thus can be used to estimate the probability of future failures and to assist in decisions on when to stop testing.

One probable action resulting from SQM findings is to remove the defects from the product under examination (e.g., find and fix bugs, create new build). Other activities attempt to eliminate

the causes of the defects—for example, root cause analysis (RCA). RCA activities include analyzing and summarizing the findings to find root causes and using measurement techniques to improve the product and the process as well as to track the defects and their removal. Process improvement is primarily discussed in the Software Engineering Process KA, with the SQM process being a source of information.

Data on inadequacies and defects found by software quality control techniques may be lost unless they are recorded. For some techniques (e.g., technical reviews, audits, inspections), recorders are present to set down such information, along with issues and decisions. When automated tools are used (see topic 4, Software Quality Tools), the tool output may provide the defect information. Reports about defects are provided to the management of the organization.

3.3. Software Quality Management Techniques

[7*, c7s3] [8*, c17] [9*, c12s5, c15s1, p417] [16*]

Software quality control techniques can be categorized in many ways, but a straightforward approach uses just two categories: static and dynamic. Dynamic techniques involve executing the software; static techniques involve analyzing documents and source code but not executing the software.

3.3.1. Static Techniques

Static techniques examine software documentation (including requirements, interface specifications, designs, and models) and software source code without executing the code. There are many tools and techniques for statically examining software work-products (see section 2.3.2). In addition, tools that analyze source code control flow and search for dead code are considered to be static analysis tools because they do not involve executing the software code.

Other, more formal, types of analytical techniques are known as formal methods. They are notably used to verify software requirements and designs. They have mostly been used in the verification of crucial parts of critical systems, such as specific security and safety requirements. (See also Formal Methods in the Software Engineering Models and Methods KA.)

3.3.2. Dynamic Techniques

Dynamic techniques involve executing the software code. Different kinds of dynamic techniques are performed throughout the development and maintenance of software. Generally, these are testing techniques, but techniques such as simulation and model analysis may be considered dynamic (see the Software Engineering Models and Methods KA). Code reading is considered a static technique, but experienced software engineers may execute the code as they read through it. Code reading may utilize dynamic techniques. This discrepancy in categorizing indicates that people with different roles and experience in the organization may consider and apply these techniques differently.

Different groups may perform testing during software development, including groups independent of the development team. The Software Testing KA is devoted entirely to this subject.

3.3.3. Testing

Two types of testing may fall under V&V because of their responsibility for the quality of the materials used in the project:

- Evaluation and tests of tools to be used on the project
- Conformance tests (or review of conformance tests) of components and COTS products to be used in the product.

Sometimes an independent (third-party or IV&V) organization may be tasked to perform testing or to monitor the test process V&V may be called upon to evaluate the testing itself: adequacy of plans, processes, and procedures, and adequacy and accuracy of results.

The third party is not the developer, nor is it associated with the development of the product. Instead, the third party is an independent facility, usually accredited by some body of authority. Their purpose is to test a product for conformance to a specific set of requirements (see the Software Testing KA).

3.4. Software Quality Measurement

[3*, c4] [8*, c17] [9*, p90]

Software quality measurements are used to support decision-making. With the increasing sophistication of software, questions of quality go beyond whether or not the software works to how well it achieves measurable quality goals.

Decisions supported by software quality measurement include determining levels of software quality (notably because models of software product quality include measures to determine the degree to which the software product achieves quality goals); managerial questions about effort, cost, and schedule; determining when to stop testing and release a product (see Termination under section 5.1, Practical Considerations, in the Software Testing KA); and determining the efficacy of process improvement efforts.

The cost of SQM processes is an issue frequently raised in deciding how a project or a software development and maintenance group should be organized. Often, generic models of cost are used, which are based on when a defect is found and how much effort it takes to fix the defect relative to finding the defect earlier in the development process. Software quality measurement data collected internally may give a better picture of cost within this project or organization.

While the software quality measurement data may be useful in itself (e.g., the number of defective requirements or the proportion of defective requirements), mathematical and graphical techniques can be applied to aid in the interpretation of the measures (see the Engineering Foundations KA). These techniques include

- descriptive statistics based (e.g., Pareto analysis, run charts, scatter plots, normal distribution)
- statistical tests (e.g., the binomial test, chi-squared test)
- trend analysis (e.g., control charts; see *The Quality Toolbox* in the list of further readings)
- prediction (e.g., reliability models).

Descriptive statistics-based techniques and tests often provide a snapshot of the more troublesome areas of the software product under examination. The resulting charts and graphs are visualization aids, which the decision makers can use to focus resources and conduct process improvements where they appear to be most needed. Results from trend analysis may indicate that a schedule is being met, such as in testing, or that certain classes of faults may become more likely to occur unless some corrective action is taken in development. The predictive techniques assist in estimating testing effort and schedule and in predicting failures. More discussion on measurement in general appears in the Software Engineering Process and Software Engineering Management KAs. More specific information on testing measurement is presented in the Software Testing KA.

Software quality measurement includes measuring defect occurrences and applying statistical methods to understand the types of defects that occur most frequently. This information may be used by software process improvement for determining methods to prevent, reduce, or eliminate their recurrence. They also aid in understanding trends, how well detection and containment techniques are working, and how well the development and maintenance processes are progressing.

From these measurement methods, defect profiles can be developed for a specific application domain. Then, for the next software project within that organization, the profiles can be used to guide the SQM processes—that is, to expend the effort where problems are most likely to occur. Similarly, benchmarks, or defect counts typical of that domain, may serve as one aid in determining when the product is ready for delivery. Discussion on using data from SQM to improve development and maintenance processes appears in the Software Engineering Management and Software Engineering Process KAs.

4. Software Quality Tools

Software quality tools include static and dynamic analysis tools. Static analysis tools input source code, perform syntactical and semantic analysis without executing the code, and present results to users. There is a large variety in the depth, thoroughness, and scope of static analysis tools that

can be applied to artifacts including models, in addition to source code. (See the Software Construction, Software Testing, and Software Maintenance KAs for descriptions of dynamic analysis tools.)

Categories of static analysis tools include the following:

- Tools that facilitate and partially automate reviews and inspections of documents and code. These tools can route work to different participants in order to partially automate and control a review process. They allow users to enter defects found during inspections and reviews for later removal.
- Some tools help organizations perform software safety hazard analysis. These tools provide, e.g., automated support for failure mode and effects analysis (FMEA) and fault tree analysis (FTA).
- Tools that support tracking of software problems provide for entry of anomalies discovered during software testing and subsequent analysis, disposition, and resolution. Some tools include support for workflow and for tracking the status of problem resolution.
- Tools that analyze data captured from software engineering environments and software test environments and produce visual displays of quantified data in the form of graphs, charts, and tables. These tools sometimes include the functionality to perform statistical analysis on data sets (for the purpose of discerning trends and making forecasts). Some of these tools provide defect and removal injection rates; defect densities; yields; distribution of defect injection and removal for each of the life cycle phases.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Kan 2002 [3*]	Bott et al. 2000 [6*]	Galin 2004 [7*]	Naik and Tripathy 2008 [8*]	Sommerville 2011 [9*]	Voland 2003 [10*]	IEEE Std. 1028-2008 [16*]	Moore 2006 [17*]	Wiegers 2003 [18*]
1. Software Quality Fundamentals									
1.1. Software Engineering Culture and Ethics	c1s4	c2s3.5							
1.2. Value and Cost of Quality			c17, c22						
1.3. Models and Quality Characteristics	c24s1		c2s4	c17					
1.4. Software Quality Improvement	c1s4				c24	c11 s2.4			
1.5. Software Safety					c11s3				
2. Software Quality Management Processes									
2.1. Software Quality Assurance			c4–c6, c11, c26–27						
2.2. Verification and Validation					c2 s2.3, c8, c15 s1.1, c21 s3.3				
2.3. Reviews and Audits					c24s3		*		

	Kan 2002 [3*]	Bott et al. 2000 [6*]	Galin 2004 [7*]	Naik and Tripathy 2008 [8*]	Sommerville 2011 [9*]	Voland 2003 [10*]	IEEE Std. 1028-2008 [16*]	Moore 2006 [17*]	Wiegers 2003 [18*]
3. Software Quality Practical Considerations									
3.1. Software Quality Requirements					c11s1			c15 s3.2.2, c15 s3.3.1, c16 s9.10	c12
3.2. Defect Characterization	c3s3, c8s8, c10s2								
3.3. SQM Techniques			c7s3	c17	c12s5, c15s1, p417		*		
3.4. Software Quality Measurement	c4			c17	p90				
4. Software Quality Tools									

FURTHER READINGS

N. Leveson, *Safeware: System Safety and Computers* [20].

This book describes the importance of software safety practices and how these practices can be incorporated into software development projects.

T. Gilb, *Principles of Software Engineering Management* [21].

This is one of the first books on iterative and incremental development techniques. The Evo Method defines quantified goals, frequent time-boxed iterations, measurements of progress toward goals, and adaptation of plans based on actual results.

T. Gilb and D. Graham, *Software Inspection* [22].

This book introduces measurement and statistical sampling for reviews and defects. It presents techniques that produce quantified results for reducing defects, improving productivity, tracking projects, and creating documentation.

K.E. Wiegers, *Peer Reviews in Software: A Practical Guide* [23].

This book provides clear, succinct explanations of different peer review methods distinguished by level of formality and effectiveness. Pragmatic guidance for implementing the methods and how to select which methods are appropriate for given circumstances is provided.

N.R. Tague, *The Quality Toolbox*, 2nd ed., [24].

Provides a pragmatic how-to explanation of a comprehensive set of methods, tools, and techniques for solving quality improvement problems. Includes the seven basic quality control tools and many others.

IEEE Std. P730-2013 Draft Standard for Software Quality Assurance Processes [5].

This draft standard expands the SQA processes identified in IEEE/ISO/IEC 12207-2008. P730 establishes standards for initiating, planning, controlling, and executing the software quality assurance processes of a software development or maintenance project. Approval of this draft standard is expected in 2014.

REFERENCES

[1] P.B. Crosby, *Quality Is Free*, McGraw-Hill, 1979.

[2] W. Humphrey, *Managing the Software Process*, Addison-Wesley, 1989.

[3*] S.H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed., Addison-Wesley, 2002.

[4] *ISO/IEC 25010:2011 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—Systems and Software Quality Models*, ISO/IEC, 2011.

[5] *IEEE P730™/D8 Draft Standard for Software Quality Assurance Processes*, IEEE, 2012.

[6*] F. Bott et al., *Professional Issues in Software Engineering*, 3rd ed., Taylor & Francis, 2000.

[7*] D. Galin, *Software Quality Assurance: From Theory to Implementation*, Pearson Education Limited, 2004.

[8*] S. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*, Wiley-Spektrum, 2008.

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

[10*] G. Voland, *Engineering by Design*, 2nd ed., Prentice Hall, 2003.

[11] *RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics, 2011.

[12] *IEEE Std. 15026.1-2011 Trial-Use Standard Adoption of ISO/IEC TR 15026-1:2010 Systems and Software Engineering—Systems and Software Assurance—Part 1: Concepts and Vocabulary*, IEEE, 2011.

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

[14] *ISO 9000:2005 Quality Management Systems—Fundamentals and Vocabulary*, ISO, 2005.

[15] *IEEE Std. 1012-2012 Standard for System and Software Verification and Validation*, IEEE, 2012.

[16*] *IEEE Std. 1028-2008, Software Reviews and Audits*, IEEE, 2008.

[17*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006.

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

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

[20] N. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley Professional, 1995.

[21] T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley Professional, 1988.

[22] T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley Professional, 1993.

[23] K. Wiegers, *Peer Reviews in Software: A Practical Guide*, Addison-Wesley Professional, 2001.

[24] N.R. Tague, *The Quality Toolbox*, 2nd ed., ASQ Quality Press, 2010.

CHAPTER 11

SOFTWARE ENGINEERING PROFESSIONAL PRACTICE

ACRONYMS

ACM	Association for Computing Machinery
BCS	British Computer Society
CSDA	Certified Software Development Associate
CSDP	Certified Software Development Professional
IEC	International Electrotechnical Commission
IEEE CS	IEEE Computer Society
IFIP	International. Federation for Information Processing
IP	Intellectual Property
ISO	International Organization for Standardization
NDA	Non-Disclosure Agreement
WIPO	World Intellectual Property Organization
WTO	World Trade Organization

INTRODUCTION

The Software Engineering Professional Practice knowledge area (KA) is concerned with the knowledge, skills, and attitudes that software engineers must possess to practice software engineering in a professional, responsible, and ethical manner. Because of the widespread applications of software products in social and personal life, the quality of software products can have profound impact on our personal well-being and societal harmony. Software engineers must handle unique engineering problems, producing software with known characteristics and reliability. This requirement calls for software engineers who possess a proper set of knowledge, skills, training, and experience in professional practice.

The term "professional practice" refers to a way of conducting services so as to achieve certain standards or criteria in both the process of performing a service and the end product resulting from the service. These standards and criteria can include both technical and nontechnical aspects. The concept of professional practice can be viewed as being more applicable within those professions that have a generally accepted body of knowledge; codes of ethics and professional conduct with penalties for violations; accepted processes for accreditation, certification, and licensing; and professional societies to provide and administer all of these. Admission to these professional societies is often predicated on a prescribed combination of education and experience.

A software engineer maintains a professional practice by performing all work in accordance with generally accepted practices, standards, and guidelines notably set forth by the applicable professional society. For example, the Association for Computing Machinery (ACM) and IEEE Computer Society (IEEE CS) have established a Software Engineering Code of Ethics and Professional Practice. Both the British Computer Society (BCS) and the International Federation for Information Processing (IFIP) have established similar professional practice standards. ISO/IEC and IEEE have further provided internationally accepted software engineering standards (see Appendix B of this *Guide*). IEEE CS has established two international certification programs (CSDA, CSDP) and a corresponding *Guide to the Software Engineering Body of Knowledge* (*SWEBOK Guide*). All of these are

Figure 11.1. Breakdown of Topics for the Software Engineering Professional Practice KA

elements that lay the foundation for of the professional practice of software engineering.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING PROFESSIONAL PRACTICE

The Software Engineering Professional Practice KA's breakdown of topics is shown in Figure 11.1. The subareas presented in this KA are professionalism, group dynamics and psychology, and communication skills.

1. Professionalism

A software engineer displays professionalism notably through adherence to codes of ethics and professional conduct and to standards and

practices that are established by the engineer's professional community.

The professional community is often represented by one or more professional societies; those societies publish codes of ethics and professional conduct as well as criteria for admittance to the community. Those criteria form the basis for accreditation and licensing activities and may be used as a measure to determine engineering competence or negligence.

1.1. Accreditation, Certification, and Licensing [1*, c1s4.1, c1s5.1–c1s5.4]

1.1.1. Accreditation

Accreditation is a process to certify the competency, authority, or credibility of an organization. Accredited schools or programs are assured to adhere to particular standards and maintain certain qualities. In many countries, the basic means by which engineers acquire knowledge is through completion of an accredited course of study. Often, engineering accreditation is performed by a government organization, such as the ministry of education. Such countries with government accreditations include China, France, Germany, Israel, Italy, and Russia.

In other countries, however, the accreditation process is independent of government and performed by private membership associations. For example, in the United States, engineering accreditation is performed by an organization known as ABET. An organization known as CSAB serving as a participating body of ABET is the lead society within ABET for the accreditation of degree programs in software engineering.

While the process of accreditation may be different for each country and jurisdiction, the general meaning is the same. For an institution's course of study to be accredited means that "the accreditation body recognizes an educational institution as maintaining standards that qualify the graduates for admission to higher or more specialized institutions or for professional practice" [2].

1.1.2. Certification

Certification refers to the confirmation of a person's particular characteristics. A common type of certification is professional certification, where a person is certified as being able to complete an activity in a certain discipline at a stated level of competency. Professional certification also can also verify the holder's ability to meet professional standards and to apply professional judgment in solving or addressing problems. Professional certification can also involve the verification of prescribed knowledge, the mastering of best practice and proven methodologies, and the amount of professional experience.

An engineer usually obtains certification by passing an examination in conjunction with other experience-based criteria. These examinations are often administered by nongovernmental organizations, such as professional societies.

In software engineering, certification testifies to one's qualification as a software engineer. For example, the IEEE CS has enacted two certification programs (CSDA and CSDP) designed to confirm a software engineer's knowledge of standard software engineering practices and to advance one's career. A lack of certification does not exclude the individual from working as a software engineer. Currently certification in software engineering is completely voluntary. In fact, most software engineers are not certified under any program.

1.1.3. Licensing

"Licensing" is the action of giving a person the authorization to perform certain kinds of activities and take responsibility for resultant engineering products. The noun "license" refers to both that authorization and the document recording that authorization. Governmental authorities or statutory bodies usually issue licenses.

Obtaining a license to practice requires not only that an individual meets a certain standard, but also that they do so with a certain ability to practice or operate. Sometimes there is an entry-level requirement which sets the minimum skills and capabilities to practice, but as the professional moves through his or her career, the required skills and capabilities change and evolve.

In general, engineers are licensed as a means of protecting the public from unqualified individuals. In some countries, no one can practice as a professional engineer unless licensed; or further, no

company may offer "engineering services" unless at least one licensed engineer is employed there.

1.2. Codes of Ethics and Professional Conduct

[1*, c1s6–c1s9] [3*, c8] [4*, c1s2] [5*, c33] [6*]

Codes of ethics and professional conduct comprise the values and behavior that an engineer's professional practice and decisions should embody.

The professional community establishes codes of ethics and professional conduct. They exist in the context of, and are adjusted to agree with, societal norms and local laws. Therefore, codes of ethics and professional conduct present guidance in the face of conflicting imperatives.

Once established, codes of ethics and professional conduct are enforced by the profession, as represented by professional societies or by a statutory body.

Violations may be acts of commission, such as concealing inadequate work, disclosing confidential information, falsifying information, or misrepresenting one's abilities. They may also occur through omission, including failure to disclose risks or to provide important information, failure to give proper credit or to acknowledge references, and failure to represent client interests. Violations of codes of ethics and professional conduct may result in penalties and possible expulsion from professional status.

A code of ethics and professional conduct for software engineering was approved by the ACM Council and the IEEE CS Board of Governors in 1999 [6*]. According to the short version of this code:

> Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the eight principles concerning the public, client and employer, product, judgment, management, profession, colleagues, and self, respectively.

Since standards and codes of ethics and professional conduct may be introduced, modified, or replaced at any time, individual software engineers bear the responsibility for their own continuing study to stay current in their professional practice.

1.3. Nature and Role of Professional Societies

[1*, c1s1–c1s2] [4*, c1s2] [5*, c35s1]

Professional societies are comprised of a mix of practitioners and academics. These societies serve to define, advance, and regulate their corresponding professions. Professional societies help to establish professional standards as well as codes of ethics and professional conduct. For this reason, they also engage in related activities, which include

- establishing and promulgating a body of generally accepted knowledge;
- accrediting, certifying, and licensing;
- dispensing disciplinary actions;
- advancing the profession through conferences, training, and publications.

Participation in professional societies assists the individual engineer in maintaining and sharpening their professional knowledge and relevancy and in expanding and maintaining their professional network.

1.4. Nature and Role of Software Engineering Standards

[1*, c5s3.2, c10s2.1] [5*, c32s6] [7*, c1s2]

Software engineering standards cover a remarkable variety of topics. They provide guidelines for the practice of software engineering and processes to be used during development, maintenance, and support of software. By establishing a consensual body of knowledge and experience, software engineering standards establish a basis upon which further guidelines may be developed. Appendix B of this *Guide* provides guidance on IEEE and ISO/IEC software engineering standards that support the knowledge areas of this *Guide*.

The benefits of software engineering standards are many and include improving software quality,

helping avoid errors, protecting both software producers and users, increasing professional discipline, and helping technology transition.

1.5. Economic Impact of Software

[3*, c10s8] [4*, c1s1.1] [8*, c1]

Software has economic effects at the individual, business, and societal levels. Software "success" may be determined by the suitability of a product for a recognized problem as well as by its effectiveness when applied to that problem.

At the individual level, an engineer's continuing employment may depend on their ability and willingness to interpret and execute tasks in meeting customers' or employers' needs and expectations. The customer or employer's financial situation may in turn be positively or negatively affected by the purchase of software.

At the business level, software properly applied to a problem can eliminate months of work and translate to elevated profits or more effective organizations. Moreover, organizations that acquire or provide successful software may be a boon to the society in which they operate by providing both employment and improved services. However, the development or acquisition costs of software can also equate to those of any major acquisition.

At the societal level, direct impacts of software success or failure include or exclude accidents, interruptions, and loss of service. Indirect impacts include the success or failure of the organization that acquired or produced the software, increased or decreased societal productivity, harmonious or disruptive social order, and even the saving or loss of property and life.

1.6. Employment Contracts

[1*, c7]

Software engineering services may be provided under a variety of client-engineer relationships. The software engineering work may be solicited as company-to-customer supplier, engineer-to-customer consultancy, direct hire, or even volunteering. In all of these situations, the customer and supplier agree that a product or service will be provided in return for some sort of consideration. Here, we are most concerned with the engineer-to-customer arrangement and its attendant agreements or contracts, whether they are of the direct-hire or consultant variety, and the issues they typically address.

A common concern in software engineering contracts is confidentiality. Employers derive commercial advantage from intellectual property, so they strive to protect that property from disclosure. Therefore, software engineers are often required to sign non-disclosure (NDA) or intellectual property (IP) agreements as a precondition to work. These agreements typically apply to information the software engineer could only gain through association with the customer. The terms of these agreements may extend past termination of the association.

Another concern is IP ownership. Rights to software engineering assets—products, innovations, inventions, discoveries, and ideas—may reside with the employer or customer, either under explicit contract terms or relevant laws, if those assets are obtained during the term of the software engineer's relationship with that employer or customer. Contracts differ in the ownership of assets created using non-employer-owned equipment or information.

Finally, contracts can also specify among other elements the location at which work is to be performed; standards to which that work will be held; the system configuration to be used for development; limitations of the software engineer's and employer's liability; a communication matrix and/or escalation plan; and administrative details such as rates, frequency of compensation, working hours, and working conditions.

1.7. Legal Issues

[1*, c6, c11] [3*, c5s3–c5s4] [9*, c1s10]

Legal issues surrounding software engineering professional practice notably include matters related to standards, trademarks, patents, copyrights, trade secrets, professional liability, legal requirements, trade compliance, and cybercrime. It is therefore beneficial to possess knowledge of these issues and their applicability.

Legal issues are jurisdictionally based; software engineers must consult attorneys who

specialize in the type and jurisdiction of any identified legal issues.

1.7.1. Standards

Software engineering standards establish guidelines for generally accepted practices and minimum requirements for products and services provided by a software engineer. Appendix B of this *Guide* provides guidance on software engineering standards that are applicable to each KA.

Standards are valuable sources of requirements and assistance during the everyday conduct of software engineering activities. Adherence to standards facilitates discipline by enumerating minimal characteristics of products and practice. That discipline helps to mitigate subconscious assumptions or overconfidence in a design. For these reasons, organizations performing software engineering activities often include conformance to standards as part of their organizational policies. Further, adherence to standards is a major component of defense from legal action or from allegations of malpractice.

1.7.2. Trademarks

A trademark relates to any word, name, symbol, or device that is used in business transactions. It is used "to indicate the source or origin of the goods" [2].

Trademark protection protects names, logos, images, and packaging. However, if a name, image, or other trademarked asset becomes a generic term, then trademark protection is nullified.

The World Intellectual Property Organization (WIPO) is the authority that frames the rules and regulations on trademarks. WIPO is the United Nations agency dedicated to the use of intellectual property as a means of stimulating innovation and creativity.

1.7.3. Patents

Patents protect an inventor's right to manufacture and sell an idea. A patent consists of a set of exclusive rights granted by a sovereign government to an individual, group of individuals, or organization for a limited period of time. Patents are an old form of idea-ownership protection and date back to the 15th century.

Application for a patent entails careful records of the process that led to the invention. Patent attorneys are helpful in writing patent disclosure claims in a manner most likely to protect the software engineer's rights.

Note that, if inventions are made during the course of a software engineering contract, ownership may belong to the employer or customer or be jointly held, rather than belong to the software engineer.

There are rules concerning what is and is not patentable. In many countries, software code is not patentable, although software algorithms may be. Existing and filed patent applications can be searched at WIPO.

1.7.4. Copyrights

Most governments in the world give exclusive rights of an original work to its creator, usually for a limited time, enacted as a copyright. Copyrights protect the way an idea is presented—not the idea itself. For example, they may protect the particular wording of an account of an historical event, whereas the event itself is not protected. Copyrights are long-term and renewable; they date back to the 17th century.

1.7.5. Trade Secrets

In many countries, an intellectual asset such as a formula, algorithm, process, design, method, pattern, instrument, or compilation of information may be considered a "trade secret," provided that these assets are not generally known and may provide a business some economic advantage. The designation of "trade secret" provides legal protection if the asset is stolen. This protection is not subject to a time limit. However, if another party derives or discovers the same asset legally, then the asset is no longer protected and the other party will also possess all rights to use it.

1.7.6. Professional Liability

It is common for software engineers to be concerned with matters of professional liability. As

an individual provides services to a client or employer, it is vital to adhere to standards and generally accepted practices, thereby protecting against allegations or proceedings of or related to malpractice, negligence, or incompetence.

For engineers, including software engineers, professional liability is related to product liability. Under the laws and rules governing in their jurisdiction, engineers may be held to account for failing to fully and conscientiously follow recommended practice; this is known as "negligence." They may also be subject to laws governing "strict liability" and either implied or express warranty, where, by selling the product, the engineer is held to warrant that the product is both suitable and safe for use. In some countries (for example, in the US), "privity" (the idea that one could only sue the person selling the product) is no longer a defense against liability actions.

Legal suits for liability can be brought under tort law in the US allowing anyone who is harmed to recover their loss even if no guarantees were made. Because it is difficult to measure the suitability or safety of software, failure to take due care can be used to prove negligence on the part of software engineers. A defense against such an allegation is to show that standards and generally accepted practices were followed in the development of the product.

1.7.7. Legal Requirements

Software engineers must operate within the confines of local, national, and international legal frameworks. Therefore, software engineers must be aware of legal requirements for

- registration and licensing—including examination, education, experience, and training requirements;
- contractual agreements;
- noncontractual legalities, such as those governing liability;
- Basic information on the international legal framework can be accessed from the World Trade Organization (WTO).

1.7.8. Trade Compliance

All software professionals must be aware of legal restrictions on import, export, or reexport of goods, services, and technology in the jurisdictions in which they work. The considerations include export controls and classification, transfer of goods, acquisition of necessary governmental licenses for foreign use of hardware and software, services and technology by sanctioned nation, enterprise or individual entities, and import restrictions and duties. Trade experts should be consulted for detailed compliance guidance.

1.7.9. Cybercrime

Cybercrime refers to any crime that involves a computer, computer software, computer networks, or embedded software controlling a system. The computer or software may have been used in the commission of a crime or it may have been the target. This category of crime includes fraud, unauthorized access, spam, obscene or offensive content, threats, harassment, theft of sensitive personal data or trade secrets, and use of one computer to damage or infiltrate other networked computers and automated system controls.

Computer and software users commit fraud by altering electronic data to facilitate illegal activity. Forms of unauthorized access include hacking, eavesdropping, and using computer systems in a way that is concealed from their owners.

Many countries have separate laws to cover cybercrimes, but it has sometimes been difficult to prosecute cybercrimes due to a lack of precisely framed statutes. The software engineer has a professional obligation to consider the threat of cybercrime and to understand how the software system will protect or endanger software and user information from accidental or malicious access, use, modification, destruction, or disclosure.

1.8. Documentation

[1*, c10s5.8] [3*, c1s5] [5*, c32]

Providing clear, thorough, and accurate documentation is the responsibility of each software engineer. The adequacy of documentation is

judged by different criteria based on the needs of the various stakeholder audiences.

Good documentation complies with accepted standards and guidelines. In particular, software engineers should document

- relevant facts,
- significant risks and tradeoffs, and
- warnings of undesirable or dangerous consequences from use or misuse of the software.

Software engineers should avoid

- certifying or approving unacceptable products,
- disclosing confidential information, or
- falsifying facts or data.

In addition, software engineers and their managers should notably provide the following documentation for use by other elements of the software development organization:

- software requirements specifications, software design documents, details on the software engineering tools used, software test specifications and results, and details on the adopted software engineering methods;
- problems encountered during the development process.

For external stakeholders (customer, users, others) software documentation should notably provide

- information needed to determine if the software is likely to meet the customer's and users' needs,
- description of the safe, and unsafe, use of the software,
- description of the protection of sensitive information created by or stored using the software, and
- clear identification of warnings and critical procedures.

Use of software may include installation, operation, administration, and performance of other functions by various groups of users and support personnel. If the customer will acquire ownership of the software source code or the right to modify the code, the software engineer should provide documentation of the functional specifications, the software design, the test suite, and the necessary operating environment for the software.

The minimum length of time documents should be kept is the duration of the software products' life cycle or the time required by relevant organizational or regulatory requirements.

1.9. Tradeoff Analysis

[3*, c1s2, c10] [9*, c9s5.10]

Within the practice of software engineering, a software engineer often has to choose between alternative problem solutions. The outcome of these choices is determined by the software engineer's professional evaluation of the risks, costs, and benefits of alternatives, in cooperation with stakeholders. The software engineer's evaluation is called "tradeoff analysis." Tradeoff analysis notably enables the identification of competing and complementary software requirements in order to prioritize the final set of requirements defining the software to be constructed (see Requirements Negotiation in the Software Requirements KA and Determination and Negotiation of Requirements in the Software Engineering Management KA).

In the case of an ongoing software development project that is late or over budget, tradeoff analysis is often conducted to decide which software requirements can be relaxed or dropped given the effects thereof.

A first step in a tradeoff analysis is establishing design goals (see Engineering Design in the Engineering Foundations KA) and setting the relative importance of those goals. This permits identification of the solution that most nearly meets those goals; this means that the way the goals are stated is critically important.

Design goals may include minimization of monetary cost and maximization of reliability, performance, or some other criteria on a wide range of dimensions. However, it is difficult to formulate a tradeoff analysis of cost against risk, especially where primary production and secondary risk-based costs must be traded against each other.

A software engineer must conduct a tradeoff analysis in an ethical manner—notably by being objective and impartial when selecting criteria for comparison of alternative problem solutions and when assigning weights or importance to these criteria. Any conflict of interest must be disclosed up front.

2. Group Dynamics and Psychology

Engineering work is very often conducted in the context of teamwork. A software engineer must be able to interact cooperatively and constructively with others to first determine and then meet both needs and expectations. Knowledge of group dynamics and psychology is an asset when interacting with customers, coworkers, suppliers, and subordinates to solve software engineering problems.

2.1. Dynamics of Working in Teams/Groups

[3*, c1s6] [9*, c1s3.5, c10]

Software engineers must work with others. On one hand, they work internally in engineering teams; on the other hand, they work with customers, members of the public, regulators, and other stakeholders. Performing teams—those that demonstrate consistent quality of work and progress toward goals—are cohesive and possess a cooperative, honest, and focused atmosphere. Individual and team goals are aligned so that the members naturally commit to and feel ownership of shared outcomes.

Team members facilitate this atmosphere by being intellectually honest, making use of group thinking, admitting ignorance, and acknowledging mistakes. They share responsibility, rewards, and workload fairly. They take care to communicate clearly, directly to each other and in documents, as well as in source code, so that information is accessible to everyone. Peer reviews about work products are framed in a constructive and nonpersonal way (see Reviews and Audits in the Software Quality KA). This allows all the members to pursue a cycle of continuous improvement and growth without personal risk. In general, members of cohesive teams demonstrate respect for each other and their leader.

One point to emphasize is that software engineers must be able to work in multidisciplinary environments and in varied application domains. Since today software is everywhere, from a phone to a car, software is impacting people's lives far beyond the more traditional concept of software made for information management in a business environment.

2.2. Individual Cognition

[3*, c1s6.5] [5*, c33]

Engineers desire to solve problems. The ability to solve problems effectively and efficiently is what every engineer strives for. However, the limits and processes of individual cognition affect problem solving. In software engineering, notably due to the highly abstract nature of software itself, individual cognition plays a very prominent role in problem solving.

In general, an individual's (in particular, a software engineer's) ability to decompose a problem and creatively develop a solution can be inhibited by

- need for more knowledge,
- subconscious assumptions,
- volume of data,
- fear of failure or consequence of failure,
- culture, either application domain or organizational,
- lack of ability to express the problem,
- perceived working atmosphere, and
- emotional status of the individual.

The impact of these inhibiting factors can be reduced by cultivating good problem solving habits that minimize the impact of misleading assumptions. The ability to focus is vital, as is intellectual humility: both allow a software engineer to suspend personal considerations and consult with others freely, which is especially important when working in teams.

There is a set of basic methods engineers use to facilitate problem solving (see Problem Solving Techniques in the Computing Foundations KA). Breaking down problems and solving them one piece at a time reduces cognitive overload. Taking advantage of professional curiosity and pursuing continuous professional development

through training and study add skills and knowledge to the software engineer's portfolio; reading, networking, and experimenting with new tools, techniques, and methods are all valid means of professional development.

2.3. Dealing with Problem Complexity

[3*, c3s2] [5*, c33]

Many, if not most, software engineering problems are too complex and difficult to address as a whole or to be tackled by individual software engineers. When such circumstances arise, the usual means to adopt is teamwork and problem decomposition (see Problem Solving Techniques in the Computing Foundations KA).

Teams work together to deal with complex and large problems by sharing burdens and drawing upon each other's knowledge and creativity. When software engineers work in teams, different views and abilities of the individual engineers complement each other and help build a solution that is otherwise difficult to come by. Some specific teamwork examples to software engineering are pair programming (see Agile Methods in the Software Engineering Models and Methods KA) and code review (see Reviews and Audits in the Software Quality KA).

2.4. Interacting with Stakeholders

[9*, c2s3.1]

Success of a software engineering endeavor depends upon positive interactions with stakeholders. They should provide support, information, and feedback at all stages of the software life cycle process. For example, during the early stages, it is critical to identify all stakeholders and discover how the product will affect them, so that sufficient definition of the stakeholder requirements can be properly and completely captured.

During development, stakeholders may provide feedback on specifications and/or early versions of the software, change of priority, as well as clarification of detailed or new software requirements. Last, during software maintenance and until the end of product life, stakeholders provide feedback on evolving or new requirements as well problem reports so that the software may be extended and improved.

Therefore, it is vital to maintain open and productive communication with stakeholders for the duration of the software product's lifetime.

2.5. Dealing with Uncertainty and Ambiguity

[4*, c24s4, c26s2] [9*, c9s4]

As with engineers of other fields, software engineers must often deal with and resolve uncertainty and ambiguities while providing services and developing products. The software engineer must attack and reduce or eliminate any lack of clarity that is an obstacle to performing work.

Often, uncertainty is simply a reflection of lack of knowledge. In this case, investigation through recourse to formal sources such as textbooks and professional journals, interviews with stakeholders, or consultation with teammates and peers can overcome it.

When uncertainty or ambiguity cannot be overcome easily, software engineers or organizations may choose to regard it as a project risk. In this case, work estimates or pricing are adjusted to mitigate the anticipated cost of addressing it (see Risk Management in the Software Engineering Management KA).

2.6. Dealing with Multicultural Environments

[9*, c10s7]

Multicultural environments can have an impact on the dynamics of a group. This is especially true when the group is geographically separated or communication is infrequent, since such separation elevates the importance of each contact. Intercultural communication is even more difficult if the difference in time zones make oral communication less frequent.

Multicultural environments are quite prevalent in software engineering, perhaps more than in other fields of engineering, due to the strong trend of international outsourcing and the easy shipment of software components instantaneously across the globe. For example, it is rather common for a software project to be divided into pieces across national and cultural borders, and it is also quite common for a software project team to consist of people from diverse cultural backgrounds.

For a software project to be a success, team members must achieve a level of tolerance,

acknowledging that some rules depend on societal norms and that not all societies derive the same solutions and expectations.

This tolerance and accompanying understanding can be facilitated by the support of leadership and management. More frequent communication, including face-to-face meetings, can help to mitigate geographical and cultural divisions, promote cohesiveness, and raise productivity. Also, being able to communicate with teammates in their native language could be very beneficial.

3. Communication Skills

It is vital that a software engineer communicate well, both orally and in reading and writing. Successful attainment of software requirements and deadlines depends on developing clear understanding between the software engineer and customers, supervisors, coworkers, and suppliers. Optimal problem solving is made possible through the ability to investigate, comprehend, and summarize information. Customer product acceptance and safe product usage depend on the provision of relevant training and documentation. It follows that the software engineer's own career success is affected by the ability to consistently provide oral and written communication effectively and on time.

3.1. Reading, Understanding, and Summarizing

[5*, c33s3]

Software engineers are able to read and understand technical material. Technical material includes reference books, manuals, research papers, and program source code.

Reading is not only a primary way of improving skills, but also a way of gathering information necessary for the completion of engineering goals. A software engineer sifts through accumulated information, filtering out the pieces that will be most helpful. Customers may request that a software engineer summarize the results of such information gathering for them, simplifying or explaining it so that they may make the final choice between competing solutions.

Reading and comprehending source code is also a component of information gathering and problem solving. When modifying, extending, or rewriting software, it is critical to understand both its implementation directly derived from the presented code and its design, which must often be inferred.

3.2. Writing

[3*, c1s5]

Software engineers are able to produce written products as required by customer requests or generally accepted practice. These written products may include source code, software project plans, software requirement documents, risk analyses, software design documents, software test plans, user manuals, technical reports and evaluations, justifications, diagrams and charts, and so forth.

Writing clearly and concisely is very important because often it is the primary method of communication among relevant parties. In all cases, written software engineering products must be written so that they are accessible, understandable and relevant for their intended audience(s).

3.3. Team and Group Communication

[3*, c1s6.8] [4*, c22s3] [5*, c27s1] [9*, c10s4]

Effective communication among team and group members is essential to a collaborative software engineering effort. Stakeholders must be consulted, decisions must be made, and plans must be generated. The greater the number of team and group members, the greater the need to communicate.

The number of communication paths, however, grows quadratically with the addition of each team member. Further, team members are unlikely to communicate with anyone perceived to be removed from them by more than two degrees (levels). This problem can be more serious when software engineering endeavors or organizations are spread across national and continental borders.

Some communication can be accomplished in writing. Software documentation is a common substitute for direct interaction. Email is another but, although it is useful, it is not always enough; also, if one sends too many messages, it becomes difficult to identify the important information. Increasingly, organizations are using enterprise

collaboration tools to share information. In addition, the use of electronic information stores, accessible to all team members, for organizational policies, standards, common engineering procedures, and project-specific information, can be most beneficial.

Some software engineering teams focus on face-to-face interaction and promote such interaction by office space arrangement. Although private offices improve individual productivity, colocating team members in physical or virtual forms and providing communal work areas is important to collaborative efforts.

3.4. Presentation Skills

[3*, c1s5] [4*, c22] [9*, c10s7–c10s8]

Software engineers rely on their presentation skills during software life cycle processes. For example, during the software requirements phase, software engineers may walk customers and teammates through software requirements and conduct formal requirements reviews (see Requirement Reviews in the Software Requirements KA). During and after software design, software construction, and software maintenance, software engineers lead reviews, product walkthroughs (see Review and Audits in the Software Quality KA), and training. All of these require the ability to present technical information to groups and solicit ideas or feedback.

The software engineer's ability to convey concepts effectively in a presentation therefore influences product acceptance, management, and customer support; it also influences the ability of stakeholders to comprehend and assist in the product effort. This knowledge needs to be archived in the form of slides, knowledge write-up, technical whitepapers, and any other material utilized for knowledge creation.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Bott et al. 2000 [1*]	Voland 2003 [3*]	Sommerville 2011 [4*]	McConnell 2004 [5*]	IEEE-CS/ACM 1999 [6*]	Moore 2006 [7*]	Tockey 2004 [8*]	Fairley 2009 [9*]
1. Professionalism								
1.1. Accreditation, Certification, and Licensing	c1s4.1, c1s5.1–c1s5.4							
1.2. Codes of Ethics and Professional Conduct	c1s6–c1s9	c8	c1s2	c33	*			
1.3. Nature and Role of Professional Societies	c1s1–c1s2		c1s2	c35s1				
1.4. Nature and Role of Software Engineering Standards	c5s3.2, c10s2.1			c32s6		c1s2		
1.5. Economic Impact of Software		c10s8	c1s1.1				c1	
1.6. Employment Contracts	c7							
1.7. Legal Issues	c6, c11	c5s3–c5s4						c1s10
1.8. Documentation	c10s5.8	c1s5		c32				
1.9. Tradeoff Analysis		c1s2, c10						c9s5.10
2. Group Dynamics and Psychology								
2.1. Dynamics of Working in Teams/ Groups		c1s6						c1s3.5, c10
2.2. Individual Cognition		c1s6.5		c33				
2.3. 2.3 Dealing with Problem Complexity		c3s2		c33				
2.4. Interacting with Stakeholders								c2s3.1

	Bott et al. 2000 [1*]	**Voland 2003 [3*]**	**Sommerville 2011 [4*]**	**McConnell 2004 [5*]**	**IEEE-CS/ACM 1999 [6*]**	**Moore 2006 [7*]**	**Tockey 2004 [8*]**	**Fairley 2009 [9*]**
2.5. Dealing with Uncertainty and Ambiguity			c24s4, c26s2					c9s4
2.6. Dealing with Multicultural Environments								c10s7
3. Communication Skills								
3.1. Reading, Understanding, and Summarizing				c33s3				
3.2. Writing		c1s5						
3.3. Team and Group Communication		c1s6.8	c22s3	c27s1				c10s4
3.4. Presentation Skills		c1s5	c22					c10s7–c10s8

FURTHER READINGS

Gerald M. Weinberg, *The Psychology of Computer Programming* [10].

This was the first major book to address programming as an individual and team effort and became a classic in the field.

Kinney and Lange, P.A., *Intellectual Property Law for Business Lawyers* [11].

This book covers IP laws in the US. It not only talks about what the IP law is; it also explains why it looks the way it does.

REFERENCES

[1*] F. Bott et al., *Professional Issues in Software Engineering*, 3rd ed., Taylor & Francis, 2000.

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

[3*] G. Voland, *Engineering by Design*, 2nd ed., Prentice Hall, 2003.

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

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

[6*] IEEE CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices, "Software Engineering Code of Ethics and Professional Practice (Version 5.2)," 1999; www.acm.org/serving/se/code.htm.

[7*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006.

[8*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, Addison-Wesley, 2004.

[9*] R.E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

[10] G.M. Weinberg, *The Psychology of Computer Programming: Silver Anniversary Edition*, Dorset House, 1998.

[11] Kinney and Lange, P.A., *Intellectual Property Law for Business Lawyers*, Thomson West, 2013.

CHAPTER 12
SOFTWARE QUALITY

ABBREVIATIONS

CI/CD	Continuous Integration/Continuous Delivery
CoSQ	Cost of Software Quality
COTS	Commercial Off-The-Shelf
FMEA	Failure Mode and Effects Analysis
FTA	Fault Tree Analysis
IV&V	Independent Verification and Validation
PDCA	Plan-Do-Check-Act
PSP	Personal Software Process
QFD	Quality Function Deployment
RCA	Root Cause Analysis
SCM	Software Configuration Management
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SQC	Software Quality Control
SQM	Software Quality Management
VSEs	Very Small Entities
V&V	Verification and Validation

INTRODUCTION

What is software quality, and why is it so important that it is included in many knowledge areas (KAs) of the *SWEBOK Guide*? One reason is that the term *software quality* is overloaded. Software quality may refer to the desirable characteristics of software products, to the extent to which a particular software product has those characteristics (software product quality), and to the processes, tools and techniques used to achieve those characteristics (software process quality). Over the years, authors and organizations have defined the term *quality* differently. Phil Crosby defined quality as "conformance to requirements" [2]. Watts Humphrey referred to it as "achieving excellent levels of "fitness for use" [3]. Meanwhile, IBM coined the phrase "market-driven quality," where the "customer is the final arbiter" [4, p. 8].

More recently, software (product) quality has been defined as the "capability of a software product to satisfy stated and implied needs under specified conditions" [4] and as "the degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations" [6]. Both definitions embrace the premise of conformance to requirements. Neither refers to different types of requirements (requirements categorized according to functionality, reliability, performance, dependability or any other characteristic). Significantly, however, these definitions emphasize that quality is an important characteristic of requirements.

These definitions also illustrate another reason for the recurring discussions about software quality throughout the *SWEBOK Guide* — the often-unclear distinction between *software quality* and *software quality requirements* ("the *-ilities*" is a common shorthand for these terms). Software quality requirements (Quality of Service Constraints in the Software Requirements KA) are attributes of (or constraints on) functional requirements (what the system does). Software requirements may also specify resource use, a communication

protocol or many other characteristics (Technology Constraints in the Software Requirements KA). This KA attempts to clarify by using *software quality* in the broadest sense from the definitions above and by using *software quality requirements* as constraints on functional requirements. Software product quality is achieved by conforming to all requirements regardless of specified characteristics or grouping or naming of requirements.

Software quality is also discussed in many other SWEBOK Knowledge Areas because it is a basic parameter of a software engineering effort. The primary goal for all engineered products is to deliver maximum stakeholder value while balancing the constraints of development and maintenance costs and schedule, sometimes characterized as *fitness for use*. Stakeholder value is expressed in requirements. For software products, stakeholders could value price (what they pay for the product), lead time (how fast they get the product), and software quality. (See the Software Requirements KA for a broader discussion of this.)

The software process quality aspect, which is implied by the above, must be made explicit. The quality of a software process can be also observed in process characteristics such as efficiency, effectiveness, usability, and learnability. Defects in that process will likely show up as defects in the resulting software product, as well.

Finally, the Agile and DevOps movements aim at improving the software process and product quality by promoting quick iteration feedback loop and eliminating organizational silos by collocating users and software engineers. Other practices like pair programming and the automation of development, testing, and operations services also bring value, improve efficiency, and can detect defects early. (Refer to the Process KA for Agile life cycles and the Software Operations KA for more information on DevOps processes.)

This KA provides an overview of practices, tools and techniques for understanding software quality and planning and appraising the state of software quality during development, maintenance and operation, from both a software product perspective and

a software process perspective. Cited references provide additional details.

BREAKDOWN OF TOPICS FOR SOFTWARE QUALITY

The breakdown of topics for the Software Quality KA is presented in Figure 1.

Figure 1. Breakdown of Topics for Software Quality

1. Software Quality Fundamentals [1*, c1s1.3]

Agreeing on what constitutes software quality for all stakeholders and communicating that agreement to software engineers requires that the many aspects of quality be formally defined and communicated. The main challenges the software engineer faces to ensure quality include the following:

- Difficulty in clearly defining requirements;
- Maintaining effective communication with the client/user;
- Deviations from specifications;
- Architecture and design errors;
- Coding errors;
- Noncompliance with current processes/procedures;
- Inadequate work product reviews and tests;
- Documentation errors.

Software quality is defined as "conformance to established requirements; the capability of a software product to satisfy stated and implied needs when under specified conditions" [6]. It is further defined "by the degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent

stakeholder needs, wants, and expectations" [6].

A software engineer should understand software quality concepts, characteristics, and values and their application to the many development, maintenance and operation activities. An important concept is that the software requirements are expected to define the required software quality attributes. Furthermore, software requirements influence the measurement methods and acceptance criteria for assessing how the software and related work products achieve the desired quality levels. Another important concept is that software quality should be planned early and assessed at many milestones during the software life cycle. Finally, how to adapt software quality assurance (SQA) activities to accommodate Agile software development is presented in detail in the Institute of Electrical and Electronics Engineers (IEEE) Standard 730:2014 [6].

1.1. Software Engineering Culture and Ethics [1*, c1s1.6] [1*, c2] [5*]

An organization's culture affects how software engineers influence software quality. As Iberle explains [19], software engineering practices vary depending on the business model (e.g., custom, mass-market, commercial, firmware) and the industry where the software engineers work. Software engineers are expected to share a commitment to software quality in the context of their industry and as part of their culture. A healthy software engineering culture includes many characteristics, such as the understanding that trade-offs among cost, schedule and quality are a basic tenet of any product's engineering. A strong software engineering ethic assumes that engineers accurately report information, conditions and outcomes related to quality. Ethics also play a significant role in software quality in the professional culture of engineering, and in the attitudes of software engineers. The IEEE Computer Society and the Association for Computing Machinery (ACM) have developed a code of ethics and professional practice. (See Codes of Ethics and Professional Conduct in the Software Engineering Professional Practice KA.)

1.2. Value and Costs of Quality [1*, c2s2.2]

One major factor driving resistance to implementing SQA is its perceived high cost. However, not implementing basic SQA activities can be costly as well. Software engineers have to inform their administration of the risks they take when they are not fully committed to quality. This can be done by explaining the cost of software quality concepts to management. Cost of software quality (CoSQ) is defined as the sum of the following project costs:

- Cost of planning and construction activities (e.g., planning, designing, development);
- Cost of prevention activities (process improvement, tools, training);
- Cost of appraisal activities for defect detection (e.g., reviews, audits, testing);
- Cost of nonconformance rework (internal failure cost and external failure cost).

The CoSQ can be broken down into two top-level categories: *conformance cost* and *nonconformance cost*. Conformance cost is the total of all investments in error and defect detection (*appraisal*) and prevention activities. Appraisal costs arise from project activities that are intended to find errors and defects. These include testing (as detailed in the Software Testing KA) and reviews and audits (as detailed later in this KA). Appraisal costs extend to subcontracted software

suppliers, if any. Prevention costs include investments in software process improvement (SPI) efforts, quality infrastructure, quality tools, work product templates and training. These costs might not be specific to a project; they often span the larger organization.

Nonconformance cost is the total of all spending dealing with errors and defects that have been detected. Internal costs are those incurred to repair errors and defects found during appraisal activities and discovered before the software product is delivered to the customer. External costs include those incurred responding to software failures discovered after delivery to the customer. External costs include the rework needed to repair and test an updated release. However, the financial impact on the customer who encounters a failure is just as important. For example, the customer's lost productivity, lost data and potential loss of reputation in the marketplace must be acknowledged and accounted for. Software engineers should seek the optimal CoSQ — the minimal total cost for a specified quality level.

1.3. Process and Product Quality

Software quality encompasses several perspectives: the software process quality, the software end-product quality and the software work products (also called *intermediary products*) quality. This section covers each perspective of software quality knowledge a software engineer must have.

1.3.1. Software Process Quality [1*, c3s3.2–s3.3; c9]

Crosby [2] and Humphrey [3] have demonstrated that software quality management (SQM) and software engineering process quality have a direct effect on the quality of the final software product. (Models and criteria that evaluate and improve the capabilities of software organizations are primarily project organization and management considerations and, as such, are covered in the Software Engineering Management and Software Engineering Process KAs.) International Organization for Standardization (ISO) 9000 [10] proposes another process quality perspective, where a management system that oversees the processes' actors, activities, controls, input and outputs ensures the quality of outputs (e.g., work products and final product). A *management system* is defined as a "set of interrelated or interacting elements of an organization to establish policies and objectives, and processes to achieve those objectives" [10]. This perspective requires software engineering organizations to take the time to describe their policies, processes and procedures with enough detail that software engineer roles and responsibilities are clear during life cycle activities (as detailed in the Software Engineering Process KA).

SQA activities, listed in the IEEE 730:2014 Standard [6], describe the many quality assurance activities that should be conducted early in a software project's life cycle to ensure quality. Software engineers should be aware of the need to plan and execute SQA activities at certain project milestones and keep records of their execution. These activities consist of document and code reviews as well as verification and validation (V&V) activities, including testing (as detailed in Section 2.2 of this KA), which evaluate the output of a process's compliance with its requirements and specifications.

1.3.2. Software Product Quality [1*, s3.2–s3.3] [7,c4s4.1.2]

First, the software engineer must determine the real purpose of the software to be designed and constructed. Stakeholder

requirements are paramount here. They include quality requirements (called *Quality of Service Constraints* in the Software Requirements KA) and functional requirements. Thus, software engineers are responsible for eliciting quality requirements that might not be explicit at the outset and for understanding their importance and the difficulty in defining them, measuring them and establishing them for final acceptance. Software engineers should understand how to define quality requirements as well as their quality targets to ensure they can be effectively be measured at the acceptance stage of the project. During the project planning, software engineers must keep these quality requirements in mind. They must also anticipate potential additional development costs if attributes such as safety, security and dependability are important.

An international standard on what constitutes a software product's many measurable quality characteristics was reached and is described in the ISO/IEC (International Electrotechnical Commission) 25010:2011 [4]. This standard proposes several software product quality models, consisting of characteristics and sub-characteristics, for software product quality and software quality in use. These software characteristics are commonly called *product quality requirements*, which are nonfunctional software requirements [7* C4s4.1.2]. Software engineers should know the many software characteristics that can be planned, implemented and measured during software construction (e.g., functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability). Software engineers should also know that certain quality characteristics have conflicting impacts. For example, trying to augment the security characteristic by encrypting data might adversely affect the performance characteristic. This international standard also proposes a general data quality model that focuses on data quality as part of a computer system and defines quality characteristics for target data used by humans and systems.

Another software product quality perspective is the quality of work products. The term *work product* means any artifact resulting from a process used to create the final software product. Work products include system/subsystem specifications, software requirements specifications for a system's software components, software design descriptions, source code, software test documentation and test reports. Sound engineering practice requires that intermediate work products relevant to quality be evaluated using work product reviews and inspections (discussed later in this chapter) throughout the software engineering process.

1.4. Software Quality Improvement [1*, s9.9] [2] [3]

Software product quality can be improved using lean principles as well as an iterative process of continual improvement, which includes management control, coordination of activities, and feedback from many concurrent processes: (1) the process of improving the software life cycle processes; (2) the process of fault/defect categorization, detection, removal and prevention; and (3) a personal improvement process. (Refer to the Process KA for more information about software process assessment and improvement of the software life cycle processes.)

The theory and concepts behind quality improvement — such as building quality through the prevention and early detection of defects, continual improvement, and stakeholder focus — are also pertinent to software engineering. These concepts are based on the work of experts in quality who

have stated that a product's quality is directly linked to the quality of the process used to create it. Improvement models such as the Deming improvement cycle of plan-do-check-act (PDCA), evolutionary delivery, Kaizen and techniques like quality function deployment (QFD) offer ways to specify quality objectives and determine whether they are met. The ISO/IEC Technical Specification TS 33061:202 [22] Standard defines a process assessment model for software life cycle processes using five process capabilities levels (from level 0: incomplete to level 5: optimizing process). Additionally, software engineers can assess the maturity of their SQA activities in their software projects using the IEEE 730:2014 Standard guidance [6] (refer to Annex D). Management sponsorship supports process and product evaluations and the resulting findings. Then, an improvement program is developed identifying detailed actions and improvement projects to be addressed in a feasible time frame. Management support implies that each improvement project has enough resources to achieve the goal defined for it. Management sponsorship is solicited frequently by implementing proactive communication activities.

Finally, since software engineering is a complex process, it cannot be reduced to a cookbook of procedures. So instead, Humphrey proposed the personal software process (PSP) for software engineers to assess their skills and knowledge and continually improve them.

1.5. Software Dependability and Integrity Levels [1*, c4s4.8.2, c7s7.3.3] [11]

Software-intensive and safety-critical systems are those in which a system failure could harm human life, other living things, physical structures or the environment. The software in these systems is considered safety-critical and requires the use of systematic methods and tools to ensure its high level of quality. A growing number of industries are using increasing numbers of safety-critical software, including transportation systems, chemical and nuclear plants and medical devices. Software failure in these systems could have catastrophic effects. Engineers use industry standards, such as software considerations in airborne systems and equipment certification DO-178C [8] and railway applications EN 50128 [18], and emerging processes, tools, and techniques to develop safety-critical software more safely. These standards, tools and techniques reduce the risk of injecting faults into the software and thus improve software availability, reliability and maintainability. Software engineers and their managers must understand the threats and issues and develop the skills needed to anticipate and prevent accidents before they occur [15].

Safety-critical software can be categorized as *direct* or *indirect*. Direct software is embedded in a safety-critical system, such as an aircraft's flight control computer. Indirect software includes software applications used to develop safety-critical software. Indirect software is included in software engineering environments and software test environments.

Three complementary techniques for reducing failure risk are avoidance, detection and removal, and damage limitation. These techniques impact software functional requirements, performance requirements and development processes. Increasing risk implies increasing SQA and more rigorous review techniques such as inspections [16]. Higher risk levels might necessitate more thorough inspections of requirements, design, and code, or the use of more formal verification and validation techniques. Another technique for managing and controlling software risk is building assurance cases. An *assurance case* is a reasoned, auditable artifact created to support

the contention that its claim or claims are satisfied. It contains the following relationships: one or more claims about properties, arguments that logically link the evidence and any assumptions to the claims, and a body of evidence and assumptions supporting these arguments [9].

1.5.1. Dependability [7, c10]

In cases where system failure may have severe consequences, overall dependability (e.g., hardware, software, and human or operational dependability) is the main quality requirement, aside from basic software functionality, for the following reasons: System failures affect a large number of people; users often reject systems that are unreliable, unsafe, or insecure; system failure costs could be enormous; and undependable systems might cause information loss. Many standards address different perspectives of dependability, such as reliability and availability. System and software dependability regroups several related quality characteristics: availability, reliability, maintainability and supportability, risk assessment, and safety and security [21]. When developing dependable software, engineers can apply tools and techniques to reduce the risk of injecting faults into the intermediate deliverables or the final software product. They can use static, dynamic, or formal methods for verification and validation (V&V), and testing processes, as well as other specialized techniques, methods, and tools to identify defects that affect dependability as early as possible in the software life cycle [7*, c10.5]. Additionally, they may have to incorporate specific mechanisms into the software to guard against external attacks and to tolerate faults during its operation.

1.5.2. Integrity Levels of Software [1*, c4s4.8.2, c7s7.3.3] [11]

Defining integrity levels is a method of risk management. An *integrity level* is "a value representing project-unique characteristics (e.g., complexity, criticality, risk, safety level, security level, desired performance, and reliability) that define the importance of the system, software, or hardware to the user" [11]. The characteristics used to determine software integrity level vary depending on the intended application and use of the system. The software is a part of the system, and its integrity level is determined as a part of that system.

The assigned software integrity levels might change as the software evolves. Design, coding, procedural and technology features implemented in the system or software can raise or lower the assigned software integrity levels. The software integrity levels established for a project result from agreements among the acquirer, supplier, developer and independent assurance authorities. A *software integrity level scheme* is used to determine software integrity levels [11].

Software engineers should know that in certain safety-critical industries, such as avionics, railways, nuclear power, medical devices and many others, industry-specific guidance can require a certain level of independence for software quality activities and can assign minimum V&V techniques to be used by integrity level (example of such techniques are: usability analysis, algorithm analysis, boundary value analysis, data flow analysis, walk-through review [11, Table 2a and 3a][26]).

2. Software Quality Management Processes

"Software quality management (SQM) is concerned with ensuring that developed

software systems are 'fit for purpose.' That is, systems should meet the needs of their users, should perform efficiently and reliably, and should be delivered on time and within budget" [7 c24]. SQM is also the collection of all processes that ensure that software products, services and life cycle process implementations meet organizational software quality objectives and achieve stakeholder satisfaction. Software engineers can learn about these processes in the many software engineering standards, models and certifications available and used widely in the industry. (See Section 2.4.) SQM defines processes, process owners, requirements for the processes, measurements of the processes and their outputs, and feedback channels throughout the whole software life cycle. SQM comprises many key activities: SQA, V&V, reviews and audits, software configuration management (SCM), and policies, procedures, and processes.

Software quality planning includes determining which quality standards and models are to be used, defining specific quality goals, estimating the effort to be used to achieve each goal; and deciding at what milestone the software quality activity should take place. In some cases, software quality planning also includes defining the software quality processes to be used.

The fourth SQM category dealing with improvement has various names within the software industry, including SPI, software quality improvement, and software corrective and preventive action. The activities in this category seek to improve process effectiveness, efficiency and other characteristics to improve software quality. For example, although SPI could be included in any of the first three categories, many organizations organize SPI into a separate category that might span many projects. (See the Software Engineering Process KA.)

Software quality processes consist of tasks and techniques to indicate how software plans (e.g., software management, development, quality management or configuration management plans) are implemented and how well the intermediate and final products meet their specified requirements. Results from these tasks are assembled in reports for management before corrective action is taken. SQM process management is tasked with ensuring that the report results are accurate.

Risk management can also play an important role in delivering quality software. Incorporating disciplined risk analysis and management techniques into the software life cycle processes can help improve product quality. (See the Software Engineering Management KA for related material on risk management.)

2.1. Software Quality Assurance [1*, c1s1.5] [6]

Software quality assurance (SQA) is defined as "a set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes" [6]. To correct a common misunderstanding, SQA is not only testing of a software. A key attribute of SQA, in critical systems, is the objectivity of the SQA function concerning the quality of a software product. In this case, the SQA function might also be organizationally independent of the project; that is, free from technical, managerial and financial pressures [6]. SQA has two aspects: product assurance and process assurance, which are explained in Section 2.3.

The software quality plan (in some industry sectors, it is termed the software quality assurance plan (SQAP)) defines the activities

and tasks used to ensure that software developed for a specific product satisfies the project's established requirements and user needs within project cost and schedule constraints and is commensurate with project risks. The SQAP first ensures that quality targets are clearly defined and understood.

The SQAP's quality activities and tasks are specified, along with their costs, resource requirements, objectives and schedule in relation to related objectives, in the software engineering management, software development and software maintenance plans. The SQAP should be consistent with the SCM plan. (See the Software Configuration Management KA.) The SQAP identifies documents, standards, practices and conventions governing the project and how these items are checked and monitored to ensure adequacy and compliance. The SQAP also identifies measures; statistical techniques; procedures for problem reporting and corrective action; resources such as tools, techniques, and methodologies; security for physical media; training; and SQA reporting and documentation. Moreover, the SQAP addresses the SQA activities of any other type of activity described in the software plans — such as procurement of supplier software for the project, commercial off-the-shelf (COTS) software installation and service after software delivery. It can also contain acceptance criteria and reporting and management activities that are critical to software quality.

2.2. *Verification and Validation* [1*, c7] [11]

Verification ensures that the product is built correctly in that the output products of a life cycle phase meet the specifications imposed on them in previous phases. Verification is defined as "the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase" [11]. Alternatively, *validation* ensures that the right product is built — the product fulfills its specific intended purpose. It is defined as "the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements." These two processes are depicted in Figure 2.

Figure 2. Difference Between Verification and Validation

The purpose of V&V is to help the development organization build quality into the software throughout the development life cycle. V&V includes software testing tasks. Software testing is a necessary activity to ensure product quality. However, in most cases, software testing is insufficient to establish confidence that the software fits its intended use. V&V tasks listed in IEEE Standard 1012:2016 [11] objectively assess products and processes throughout the life cycle. This assessment demonstrates whether the requirements are correct, complete, accurate, consistent and testable. The verification process and the validation process should begin early in development or maintenance. This prevents defects late in the life cycle, which would incur rework and significantly increase costs. Software engineers should identify the product integrity level and ensure the minimum V&V tasks are assigned for key product features concerning both the product's immediate predecessor and the planned specifications. (Refer to Table 2a of IEEE Standard 1012:2016 [11].) Optional V&V tasks are

also listed and can improve software product quality. Keeping a record of the traceability among software work products can help augment the quality of the V&V activities. *Traceability* is defined as the "ability to trace the history, application or location of an object" [14].

Early planning of V&V activities ensures that each resource, role and responsibility is clearly assigned. The resulting V&V plan documents the various resources and their roles and SQA activities, as well as the techniques and tools to be used. Software engineers should choose and apply the proper V&V task depending on the software integrity level. (Refer to Section 3.1.3.) V&V can also be executed by an independent organization for very critical software. Independent verification and validation (IV&V) is defined as "V&V performed by an organization that is technically, managerially, and financially independent of the development organization" [11].

Software V&V tasks can be sorted into static, dynamic and formal tasks [20]. Dynamic techniques involve executing the software; static techniques involve analyzing documents and source code but not executing the software; formal techniques use mathematics and formal specification languages.

2.2.1. Static Analysis Techniques

Static analysis techniques directly analyze a work product's content and structure (including requirements, interface specifications, designs and models) without executing the software. Tools and techniques for statically examining software work can help software engineers in this task. For example, code reading, peer review of a work product, and static analysis of source code control flow are considered static techniques because they do not involve executing the software code.

2.2.2. Dynamic Analysis Techniques

Dynamic analysis techniques involve executing or simulating the software code, looking for errors and defects. Different dynamic techniques are performed throughout software development, maintenance and operation. Generally, these are testing techniques, but simulation and model analysis are considered dynamic analysis techniques. (See the Software Engineering Models and Methods KA.) In addition, black box testing is considered a dynamic analysis technique, as the software engineer analyzes the output received following the entry of inputs. (See the Software Testing KA.)

2.2.3. Formal Analysis Techniques

Formal analysis techniques (also called *formal methods*) are "mathematical approaches to software development where you define a formal model of the software. You may then formally analyze this model to search for errors and inconsistencies" [7*, c10s10.5]. Sometimes, the software requirements may be written using a more formal specification language known as formal methods. They are notably used to verify software requirements and designs. They have mostly been used to verify crucial parts of critical systems, such as specific security and safety requirements. (See also Formal Methods in the Software Engineering Models and Methods KA.) Different groups may perform testing during software development, including groups independent of the development team. The Software Testing KA is devoted entirely to this subject.

2.3. Technical Reviews and Audits [1*, c5, c6] [23, s4, s5]

Technical reviews are peer reviews of work products and SQA activities conducted during development, maintenance and operations [17]. Peer reviews are defined as

"the review of work products performed by peers during development of the work products to identify defects for removal" [14]. For example, during software development, a code review (often done by using a pull request technique/tool) occurs when a peer reviews the code, often at the software developer's request, before it can be merged into a project.

Reviews are valuable because they can identify issues early in development or even before a component is designed. Fixing a defect in a component that has been coded is much more expensive than catching it beforehand. Review and audit processes are broadly defined as static activities, meaning that no software or models are executed. Instead, they examine software engineering artifacts (also called *intermediary* or *work products*) concerning standards established by the organization or project for those artifacts.

Different types of work product reviews (e.g., formal and informal) are distinguished by purpose, level of independence, tools and techniques used, roles involved, and by the subject of the activity. Reviews play important roles in software quality, in SCM, and in the sharing of knowledge among colleagues. However, these different roles share a single purpose — to ensure the quality of the delivered products. Reviews should be part of the software engineering culture and should be planned, executed and documented during the software life cycle. In Agile life cycles, pair programming invites continuous reviews. Different review types for work products are described in the ISO/IEC 20246:2017 Standard [12]:

- Ad hoc reviews — unstructured reviews where each reviewer is expected to find as many defects as possible of any type;
- Checklist-based reviews — systematic reviews identifying issues based on checklists;
- Scenario-based reviews — reviews where reviewers are provided with structured guidelines on how to read through the work product under review;
- Perspective-based reviews — reviews where reviewers take on different stakeholder viewpoints and review the work product from that stakeholder's viewpoint;
- Role-based reviews — reviews in which the reviewer evaluates the work product from the perspective of various stakeholder roles, which might differ from their daily role.

Audits are more formal activities that are often mandated to be performed by third parties to ensure independence. In mature organizations, technical reviews and audits are fully integrated with the overall project plans. Therefore, technical reviews and audits should be planned, approved and conducted. Although a project audit often addresses the whole project's current state, technical reviews can also be more focused and address a specific project phase [24]. For example:

- System requirements reviews help ensure that the level of understanding of top-level system requirements is adequate to support further requirements analysis and design activities and that the system can proceed into initial system design with acceptable risk;
- System functional or preliminary design reviews help ensure that the system under review can proceed into preliminary or detailed design with acceptable risk and that all system requirements and functional performance requirements derived

from the approved preliminary system specification are defined and consistent with the project budget, program schedule, risk, and other program and system constraints;

- Preliminary design reviews help ensure that the preliminary design for the system under review is sufficiently mature and ready to proceed into detailed design and can meet the stated performance requirements within program budget, schedule, risk and other program and system constraints;
- Test readiness reviews assess test objectives, test methods and procedures, test scope, safety, readiness for the project test and evaluation, and whether test resources have been properly identified and obtained;
- Production readiness reviews ascertain that the system design is ready for production and that the project has accomplished adequate production planning for entering production.

2.4. *Standards, Models and Certifications* [1*, c4] [7, c24s24.2]

Sound use of software engineering software standards and software process assessment and improvement improves software quality. Foremost, software engineers should know the key software engineering standards that apply to their specific industry. As Iberle discussed [19], the practices software engineers use vary greatly depending on the industry, business model and organizational culture where they work. The PDCA paradigm differ from standards in that it often propose "best practices" for software engineers from a specific perspective. (Refer to the Software Engineering Process KA for more information about the PDCA paradigm for software.)

Other industry "best practices" models such as the Control Objectives for Information and Related Technologies (COBIT) for information technology governance, the Project Management Body of Knowledge (PMBOK®) for project management, the Business Analysis Body of Knowledge (BABOK®), the Information Technology Infrastructure Library (ITIL) for IT services and the Capability Maturity Model Integration (CMMI) propose software related practices that can improve the quality of software processes and products. Software organizations can also consider the possible advantages of obtaining registrations or certifications (e.g., ISO 9001 for quality, ISO 27001 for security, ISO 20000 for operations), and software engineers can also obtain Scrum and Scaled Agile Framework® (SAFe®) certifications for Agile processes. The use of these models and certifications have been shown to augment stakeholders' confidence that the software engineers' knowledge and skills are up to date and recognized internationally.

2.5. *Policies, Processes and Procedures* [1*, c9]

Establishing and observing software quality policies, processes and policies can improve software quality. An approved organizational policy, about software quality, help in guiding projects and products development decisions as well as behavior of personnel. Software engineers should promote the use of graphically represented processes and procedures that implement the quality policy and explain the roles, activities to be executed and the expected results of key software engineering processes. Processes should be documented with its user in mind and identify where quality controls are to be verified. Procedures support process execution. (Refer

to the Software Engineering Process KA for more information about this topic.)

2.6. *Software Configuration Management* [1*, c8] [7, c25]

SCM is an important activity to ensure the quality of work products and software. *Configuration management* is defined as the "discipline applying technical and administrative direction and surveillance to:

- identify and document the functional and physical characteristics of a configuration item;
- control changes to those characteristics;
- record and report change processing and implementation status;
- verify compliance with specified requirements."

Software engineers should identify which work products and software artifacts require configuration management. In addition, they should be familiar with source code versioning processes, which involve keeping track of all versions of the software and ensuring that changes different developers make do not interfere with one another, and they should know how to operate the version control tool kit. (Refer to the Software Configuration Management KA for more information about this process.)

3. Practical Considerations

3.1. *Software Quality Requirements* [1*, c3, c14] [13*,c14]

Various factors influence planning, management and selection of SQM activities and techniques, including the following:

- The domain of the system in which the software resides (for example, consider whether the system functions are safety-critical, mission-critical, business-critical or security-critical);
- The physical environment in which the software system resides;
- System and software functional and quality requirements (requirements for what the system does and how well the system performs those functions);
- The commercial (external) or standard (internal) components used in the system;
- The specific applicable software engineering standards;
- The methods and software tools to be used for development, maintenance, quality evaluation, and improvement;
- The budget, staff, project organization, plans and scheduling of all processes;
- The intended users and use of the system;
- The system's dependability and integrity level.

Information about these factors will influence how the SQM processes are organized and documented, how specific SQM activities are selected, what resources are needed, and which of those resources impose limits on the efforts.

3.2. *Defect Characterization* [1*, c1s3]

Software quality control (SQC) techniques find errors, defects and failures. Software engineers use this specific terminology to better locate and collect statistics about the life cycle stage where these anomalies have occurred in the software or its artefacts. This facilitate where the investigations and corrections must be done and better inform

management and other stakeholders of possible improvements to be done to the software engineering process and testing status and progress. When tracking errors, defects and failures, the software engineer is interested in the number and types of incidents. Numbers alone, without classification, might be insufficient to identify the underlying causes and thus to prevent them in the future. Therefore, software engineers established a meaningful taxonomy to describe and categorize such anomalies.

SQC activities discover anomalies at all stages of software development and maintenance. In some cases, the word *defect* is overloaded with too many meanings, as engineers and others use the word to refer to all different types of anomalies. However, different engineering cultures and standards often understand "defect" and other terms as having more specific meanings. To avoid confusion, software engineers should use the meaning provided by their standards [14]:

- *Error*: "A human action that produces an incorrect result." Also called *human error*;
- *Defect*: (*synonym of a fault*) An "imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced." A defect is inserted when a person developing the software makes an error. It hides in the software until (and if) it is discovered;
- *Failure*: The "termination of the ability of a system to perform a required function or its inability to perform within previously specified limits; an externally visible deviation from the system's specification event in which a system or system component does not perform a required function within specified limits." A failure is produced when the software executes a defect.

Three widely used software quality measurements, using these definitions, are error density (number of errors per unit size of documents/software), defect density (number of defects found divided by the size of the software), and failure rate (mean time to failure). Reliability models are built from failure data collected during software testing or from software in service and thus can be used to estimate the probability of future failures and assist in decisions about when to stop testing. One probable action resulting from peer reviews and testing findings is to remove these errors and defects early from the work product under examination.

Other SQM activities attempt to eliminate their causes (e.g., root cause analysis (RCA)). RCA activities include analyzing and summarizing the findings to find root causes and using measurement techniques to improve the software engineering processes, techniques and tools. (Process improvement is primarily discussed in the Software Engineering Process KA.)

Data on errors and defects found during SQA and control techniques may be lost unless they are recorded. For some techniques (e.g., peer reviews and inspections), software engineers are present to record such data and to address issues and make decisions. In addition, when automated tools are used (see Topic 4, Software Quality Tools), the tool output may provide defect trends reports that can be provided to the organization's management.

3.3. Software Quality Control and Testing [1*, c7s7.10]

Testing is considered a software quality control (SQC) activity and part of a software development project's V&V processes. Testing is an important activity to ensure software quality. Software testing is one of

many verification activities that confirm that software development output meets input requirements. IEEE 730:2014 [6] lists the many testing and retesting activities software engineers should plan, execute and record. It also recommends that testing completion criteria be set. Software engineers should plan the testing activities, including levels, techniques, measures and tools. (Refer to the Testing KA for details about the knowledge software engineers should have about software testing.)

3.4. Software Quality Measurement [1*, c10] [7, c24s24.5]

Software quality measurements are used to support decision-making. With the increasing sophistication of software, quality questions go beyond whether the software works to how well it achieves measurable quality goals. Quantifying some attribute of software can help engineers evaluate its quality or the quality of its process. (Process measurement is described in detail in the Process KA.)

Software quality measurement helps engineers make determinations about software quality (because models of software product quality include measures to determine the degree to which the software product achieves quality goals); managerial questions about effort, cost and schedule; when to stop testing and release a product (see Test-Related Measures in the Software Testing KA); and the efficacy of process improvement efforts.

The CoSQ assurance activities are an issue frequently raised in deciding how a project or a software development and maintenance organization should be organized. Often, generic models of cost are used; these models are based on when a defect is found and how much effort it takes to fix the defect relative to finding the defect earlier in development. Software quality measurement data collected internally may offer a better picture of cost within the project or organization. Although the software quality measurement data may be useful by itself (e.g., the number of defective requirements or the proportion of defective requirements), mathematical and graphical techniques can help project stakeholders interpret the measures. (See the Engineering Mathematical Foundations KA.) These techniques include the following:

- Descriptive statistics-based analysis (e.g., Pareto analysis, run charts, scatter plots, normal distribution);
- Statistical tests (e.g., the binomial test, chi-squared test);
- Trend analysis (e.g., control charts; see *The Quality Toolbox* in Further Reading);
- Prediction (e.g., reliability models).

Descriptive statistics-based techniques and tests often provide a snapshot of the more troublesome areas of the software product under examination. The resulting charts and graphs are visualization aids decision-makers can use to focus resources and conduct process improvements where they seem most needed. Results from trend analysis may indicate that a schedule is being or that certain classes of faults may become more likely unless some corrective action is taken in development. The predictive techniques help estimate testing effort and schedule and predict failures. (More discussion on measurement in general appears in the Software Engineering Process and Software Engineering Management KAs. More specific information on testing measurement is presented in the Software Testing KA.)

Software quality measurement includes measuring defect occurrences and applying statistical methods to understand what types of defects occur most frequently. This information can be used in SPI to determine methods to prevent, reduce or eliminate defect recurrence. The information also helps engineers understand trends, how well

detection and containment techniques are working, and how well the development and maintenance processes are progressing. They can use these measurement methods to develop defect profiles for a specific application domain. Then, for the next software project within that organization, the profiles can be used to guide the SQM processes — that is, to focus effort on where problems are most likely to occur. Similarly, benchmarks, or defect counts typical of that domain, may help engineers determine when the product is ready for delivery. (Discussion about using measurement data to improve development and maintenance processes appears in the Software Engineering Management and Software Engineering Process KAs.)

3.5. *Software Quality for Very Small Entities* [1*, c4s4.7.6–4.7.7, c8s8.10, c10s10.7]

Small software organizations (those with 25 or fewer employees) can also achieve software quality. Software engineers in these organizations should be aware that software quality standards have been adapted to their small scale. The ISO/IEC 29110 [22] set of standards and guides is available at no cost and includes relevant activities for four profiles of organizations that develop critical software:

1. Entry level (e.g., start-ups);
2. Basic level (e.g., one software project);
3. Intermediate level (e.g., more than one project and team);
4. Advanced level (e.g., highly technological projects).

4. **Software Quality Tools** [1*, c3s3.2.3, c7s7.8.1, c7s7.11]

Software tools improve software quality. Simple tools can be forms and checklists (e.g., a requirements traceability matrix or a code review checklist). But automated tools can also be of great help to improve software efficiency and quality. Examples of automated tools are tools that allow code versioning/branching (e.g., Git) and pull requests for code review. DevOps tools in services/scripts like on-demand environments, continuous integration/continuous delivery (CI/CD), code quality assessment, and automated testing are important contributors to software quality. (See the Software Operations KA discussion about tools.)

These tools are known as static and dynamic analysis tools. Static analysis tools input source code, perform syntactical and semantic analysis without executing the code, and present results to users. There is a large variety in the depth, thoroughness and scope of static analysis tools that can be applied to artifacts, including models, and source code. (See the Software Construction, Software Testing, and Software Maintenance KAs for descriptions of dynamic analysis tools.) Categories of static analysis tools include the following:

- Tools that facilitate and partially automate reviews and inspections of documents and code. These tools can route work to different participants in order to partially automate and control the review process. In addition, they allow users to enter defects found during inspections and reviews for later removal;
- Tools that help organizations perform software safety hazard analysis. These tools provide, for example, automated support for failure mode and effects analysis (FMEA) and fault tree analysis (FTA);
- Tools that support tracking of software problems. These tools enable entry of anomalies discovered during software testing and subsequent analysis, disposition and resolution. Some tools

include support for workflow and for tracking problem resolution status;

- Tools that analyze data captured from software engineering environments and software test environments and produce visual displays of quantified data in graphs, charts and tables. These tools sometimes include the functionality to perform statistical analysis on data sets (to discern trends and make forecasts). Some of these tools provide defect and removal injection rates, defect densities, yields, and distribution of defect injection and removal for each life cycle phase.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

Topic	**Laporte and April 2018 [1*]**	**Sommerville 2016 [7*]**	**IEEE std 730-2014 [6]**	**IEEE std 1012-2016 [11]**	**ISO/IEC 20246:2017 [12]**	**IEEE Software Code of Ethics [5*]**	**Wiegers 2013 [13*]**
1. Software Quality Fundamentals							
1.1. Software Engineering Culture and Ethics	Ch. 1s1.6, Ch. 2					X	
1.2. Value and Costs of Quality	Ch. 2s2.2						
1.3. Process and Product Quality	Ch. 3s3.2 Ch. 3s3.3 Ch. 9						
1.4. Software Quality Improvement		Ch. 2s	s4.7 Annex D				
1.5. Software Dependability and Integrity Levels	Ch. 4s4.8.2 Ch. 7s7.3.3			Ch. 5 Annex B			
2. Software Quality Management Process							
2.1. Software Quality Assurance	Ch. 1s1.5		s4 s5				
2.2. Verification and Validation	Ch. 7			Ch. 4, 6, 7, 9, 12		x	

2.3. Technical Reviews and Audits	Ch. 5 Ch. 6		s5.4.1				
2.4. Standards, Models and Certifications	Ch. 4		s5.4.2 s5.4.6				
2.5. Policies, Processes and Procedures	Ch. 9		s5.3.1.3 Tables C.15, C.16, C25				
2.6. Software Configuration Management	Ch. 8		Tables C.15, C.16, C25, C26	Ch. 7s5	x		
3. Practical Considerations							
3.1. Software Quality Requirements	Ch. 3		s4.3	Ch. 8s2 Ch. 9s2			Ch. 14
3.2. Defect Characterization	Ch. 1s3						
3.3. Quality Control and Testing	Ch. 7s7			Ch. 9s5—9s8 Cg. 9s10			
3.4. Software Quality Measurement	Ch. 10		s5.4.6 s5.5.5	Annex E			
3.5. Software Quality for Very Small Entities	Ch. 4s4.7 Ch. 8s8.10 Ch. 10s10.7		Annex G				
4. Software Quality Tools	c3s3.2.3, c7s7.8.1, c7s7.11		Annex H Table C.9, C.10			x	

FURTHER READING

IEEE 730-2014, "IEEE Standard for Software Quality Assurance Processes," 2014 [6].

Requirements for initiating, planning, controlling and executing the Software Quality Assurance processes of a software

development or maintenance project are established in this standard.

IEEE Std 1012-2016, "IEEE Standard for System, Software, and Hardware Verification and Validation," 2016 [11].

Verification and validation (V&V) processes are used to determine whether the development products of a given activity conform to that activity's requirements and whether the product satisfies its intended use and user needs. V&V life cycle process requirements are specified for different integrity levels.

ISO/IEC Std 20246-2017, "Software and Systems Engineering — Work Product Reviews," 2017 [12].

This international standard establishes a generic framework for work product reviews that can be referenced and used by all organizations involved in the management, development, testing and maintenance of systems and software.

N. Leveson, *Safeware: System Safety and Computers* [15].

This book describes the importance of software safety practices and how these practices can be incorporated into software development projects.

T. Gilb and D. Graham, *Software Inspection* [16].

This book introduces measurement and statistical sampling for reviews and defects. It presents techniques that produce quantified results for reducing defects, improving productivity, tracking projects and creating documentation.

K. E. Wiegers, *Peer Reviews in Software: A Practical Guide* [17*].

This book provides clear, succinct explanations of different peer review methods distinguished by level of formality and effectiveness. It provides pragmatic guidance for implementing the methods and for determining which methods are appropriate for given circumstances.

REFERENCES

[1*] C. Y. Laporte, A. April, *Software Quality Assurance*, IEEE Press, 2018.

[2] P. B. Crosby, *Quality Is Free*, McGraw-Hill, 1979.

[3] W. Humphrey, *Managing the Software Process*, Addison-Wesley, 1989.

[4] ISO/IEC, "ISO/IEC 25010:2011 Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — Systems and Software Quality Models," ed., 2011.

[5*] IEEE CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices, "Software Engineering Code of Ethics and Professional Practice https://www.computer.org/education/code-of-ethics.

[6] IEEE, "IEEE 730 Standard for Software Quality Assurance Processes," ed., IEEE, 2014.

[7*] I. Sommerville, *Software Engineering*, 10th ed., New York: Addison-Wesley, 2016.

[8] RTCA, "DO-178C, Software Considerations in Airborne Systems and Equipment Certification," ed., January 5,

2012. Also known as ED-12C in EUROCAE.

[9] ISO/IEC, "ISO/IEC 15026-1:2019 Systems and Software Engineering — Systems and Software Assurance — Part 1: Concepts and Vocabulary," ed., ISO/IEC, 2019.

[10] ISO, "ISO 9000:2015 Quality Management Systems — Fundamentals and Vocabulary," ed., ISO, 2015.

[11] IEEE, "IEEE Std. 1012:2016, Standard for System and Software Verification and Validation," IEEE, 2016.

[12] ISO/IEC 20246:2017, "Software and systems engineering — Work product reviews," ed., 2017.

[13*] K. E. Wiegers, *Software Requirements*, 3rd ed., Redmond, WA: Microsoft Press, 2013.

[14] IEEE/ISO/IEC, "IEEE/ISO/IEC 24765: Systems and Software Engineering — Vocabulary," 1st ed., 2017.

[15] N. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley Professional, 1995.

[16] T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley Professional, 1993.

[17*] K. Wiegers, *Peer Reviews in Software: A Practical Guide*, Addison-Wesley Professional, 2001.

[18] BS EN 50128:2011+A2:2020, "Standard for Railway Applications – Communications, Signaling and Processing Systems – Software for Railway Control and Protection Systems," British-Adopted European Standard, August 10, 2020.

[19] K. Iberle, *They don't care about quality*, proceedings of STAR East, Orlando, United States, 2013, available at https://kiberle.com/publications/.

[20] D. Wallace, L. M. Ippolito, B. B. Cuthill, *Reference Information for the Software Verification and Validation Process*, National Institute of Standards and Technology (NIST), U.D. Department of Commerce, Special Publication 500-234, 1996, available at https://www.nist.gov/publications/reference-information-software-verification-and-validation-process.

[21] IEC 60300-1:2014, "Dependability Management — Part 1: Guidance for Management and Application," version 3, September 25, 2014.

[22] ISO/IEC TR 29110-1:2016, "System and Software Engineering — Lifecycle Profiles for Very Small Entities (VSEs) — Part 1: Overview Dependability Management — Part 1: Guidance for Management," 2nd ed., 2016-06.

[23] ISO/IEC TS 33061:2021, "Information technology — Process assessment — Process Assessment Model for Software Life Cycle Processes," 2021-04.

[24] IEEE Std 15288.2:2014, "IEEE Standard for Technical Reviews and Audits on Defense Programs."

CHAPTER 13

SOFTWARE SECURITY

ACRONYMS

CC	Common Criteria
SD LC	Secure development life cycle

INTRODUCTION

Security has become a significant issue in software development because of increasing malicious activity targeting computer systems. In addition to the usual correctness and reliability concerns, software developers must pay attention to the security of the software they develop. Secure software development builds security by following a set of established and/or recommended rules and practices. Secure software maintenance complements secure software development by ensuring that no security problems are introduced during software maintenance.

BREAKDOWN OF TOPICS FOR SOFTWARE SECURITY

The breakdown of topics for the Software Security knowledge area (KA) is shown in Figure 13.1.

Figure 13.1. The Breakdown of Topics for the Software Security KA

1. Software Security Fundamentals [37, 9]

A generally accepted belief about software security is that it is much better to design security into software than to patch it in after the software is developed. To design security into software, one must consider every development life cycle stage. Secure software development involves software requirements security, software design security, software construction security and software testing security. In addition, security must be considered during software maintenance, as security faults and loopholes can be and often are introduced during maintenance.

1.1. Software Security [10]*

Security is a product quality characteristic representing the degree to which a product or system protects information and data so that persons or other products or systems have data access appropriate to their types and levels of authorization [10]. (For more information about product quality, refer to the Software Quality KA.)

1.2. Information Security [11]*

Information security preserves confidentiality, integrity and availability of information. Other properties, such as authenticity, accountability, non-repudiation and reliability can also be involved [11]. *Confidentiality* is the property of ensuring that information is not disclosed to unauthorized individuals, entities or processes. *Integrity* is the property of accuracy and completeness. *Availability* is the property of being accessible and usable on demand by an authorized entity. Software engineers should define the security properties of their software and maintain them throughout the software development life cycle.

1.3. Cybersecurity [12][38]*

Cybersecurity addresses security issues in cyberspace, including the following:

- Social engineering attacks
- Hacking
- The proliferation of malicious software (*malware*)
- Spyware
- Other potentially unwanted software [12]

Software engineers should consider the mitigation of such risks as part of software development.

2. Security Management and Organization [1*, c7][13]

Security governance and management are most effective when they are systematic; in other words, when they are woven into the culture and fabric of organizational behaviors and actions. Project managers need to elevate software security from a stand-alone technical concern to an enterprise issue [1].

2.1. Capability Maturity Model [3, c22][14]*

Many organizations practice security engineering in the development of computer programs, including operating systems, functions that manage and enforce security, packaged software products, middleware, and applications. Therefore, a diverse array of individuals must know how to apply appropriate methods and practices, including product developers, service providers, system integrators, system administrators and even security specialists. Systems Security Engineering — Capability Maturity Model (SSE-CMM), which helps measure the process capability of an organization that performs risk assessments [14], can be an important tool.

2.2. Information Security Management System [15]*

International Organization of Standardization/International Electrotechnical Commission (ISO/IEC) 27001:2013 specifies the requirements for establishing, implementing, maintaining and continually improving an information security management system (ISMS) within the organizational context [15]. Software that is part of an ISMS in an organization complies with the security policy. In other words, some software security properties come from the policy for ISMS.

2.3. Agile Practice for Software Security [4,c15,c16]*

Agile teams need to understand and adopt security practices and take more responsibility for their systems' security. Security professionals must learn to accept change, work faster and more iteratively, and think about security risks and how to manage risks in incremental terms. Finally, and most important, security needs to become an enabler instead of a blocker. The keys to a successful Agile security program are the involvement of the security team and developers, enablement, automation, and agility to keep up with Agile teams [4].

3. Software Security Engineering and Processes

3.1. Security Engineering and Secure Development Life Cycle (SDLC) [1, c1][16*][36]*

Software is only as secure as its development process. Security must be built into software engineering to ensure software security. The SDLC concept is one trend that aims to do this. SDLC uses a classical spiral model that views security holistically from the perspective of the software life cycle and ensures that security is inherent in software design and development, not an afterthought later in production. The SDLC process is claimed to reduce software maintenance costs and increase software reliability against security-related faults.

Recently, DevSecOps (meaning the integration of development, security and operations) has emerged. Beyond SDLC, DevSecOps includes an approach to culture, automation and platform design to make the software life cycle as Agile and responsible as Agile development and continuous integration (CI).

3.2. Common Criteria for Information Technology Security Evaluation [3*, c22, c25][34][35]

Security evaluation establishes confidence in the security functionality of IT products and the assurance measures applied to them. The evaluation results may help consumers determine whether IT products meet their security needs. ISO/IEC 15408, named Common Criteria (CC) for Information Technology Security Evaluation, is useful as a guide for developing, evaluating and/or procuring IT products with security functionality [34].

CC addresses the protection of assets from unauthorized disclosure, modification or loss of use. The categories of protection relating to these three types of security failure are commonly called *confidentiality*, *integrity* and *availability*, respectively.

4. Security Engineering for Software Systems [1*,c1,c3][3*,c1,c3]

4.1. Security Requirements [1,c3][2*,c2][3*,c20,c30][18]*

Software requirements security deals with clarifying and specifying security policy and objectives into software requirements, laying the foundation for security considerations in software development. Factors to consider in this phase include software requirements and threats or risks. The former refers to the specific functions required for security; the latter refers to the possible ways that software security is threatened.

4.2. Security Design [1,c4][2*,c5][3*,c20,c31][17,40]*

Security design concerns how to prevent unauthorized disclosure, creation, change, deletion or denial of access to information and other resources. It also concerns 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; ensuring the proper use of cryptology is also important.

Software design security deals with the design of software modules that fit together to meet the security objectives specified in the security requirements. This step clarifies the details of security considerations and develops the specific steps for implementation. Factors considered may include frameworks and access modes that set up the overall security monitoring/enforcement strategies, as well as the individual policy enforcement mechanisms.

4.3. Security Patterns [1,c4][19, 20, 21]*

A *security pattern* describes a particular recurring security problem that arises in a specific context and presents a well-proven generic solution [21].

4.4. Construction for Security [1,c5][3*,c20,c31][22, 23, 24]*

Software construction security concerns how to write programming code for specific situations to address security considerations. The term software construction security can mean different things to different people. It can mean the way a specific function is coded so that the code itself is secure, or it can mean the coding of security into software. Unfortunately, most people entangle the two meanings without distinction. One reason for such confusion is that it is unclear how to ensure a specific coding is secure. For example, in the C programming language, the expressions “i<<1” (shift the binary representation of i’s value to the left by one bit) and “2*” (multiply the value of variable i by constant 2) mean the same thing semantically, but do they have the same security ramifications?

The answer could be different for different combinations of ISAs and compilers. Because of this lack of understanding, software construction security — in its current state — mostly refers to the second aspect mentioned above: the coding of security into software. Coding of security into the software can be achieved by following recommended rules. A few such rules follow:

- Structure the process so that all sections requiring extra privileges are modules. The modules should be as small as possible and perform only the tasks that require those privileges.
- Ensure that any assumptions in the program are validated. If this is not possible, document them for the installers and maintainers so they know the assumptions attackers will try to invalidate.
- Ensure that the program does not share objects in memory with any other program.

- Check every function's error status. Do not recover unless neither the error's cause nor its effects affect any security considerations. The program should restore the state of the software to the state it had before the process began and then terminate.

Although there are no bulletproof ways to achieve secure software development, some general guidelines exist that can be helpful. These guidelines span every phase of the software development life cycle. The Computer Emergency Response Team (CERT) publishes reputable guidelines, and the following are its top 10 software security practices (the details can be found in [22]):

1. Validate input.
2. Heed compiler warnings.
3. Architect and design for security policies.
4. Keep it simple.
5. Default deny.
6. Adhere to the principle of least privilege.
7. Sanitize data sent to other software.
8. Practice defense in depth.
9. Use effective quality assurance techniques.
10. Adopt a software construction security standard.

4.5. Security Testing [1,c5][2*,c7][3*,c24,c31][26, 27]*

Security testing ensures that the implemented software meets the security requirements. It also verifies that the software implementation contains none of the known vulnerabilities. Whereas general software testing methods can handle the former, the latter requires security-specific testing methods. (For more information about testing, please refer to the Software Testing KA.)

There are two approaches to security-specific testing. One is to detect programming language or implementation-specific vulnerabilities by static analysis of the source code, which can be automated using tools. The other is the *penetration test*, also known as the *ethical hacking test*. It detects vulnerabilities in software behavior. It can be automated using tools such as web application scanners and fuzzing tools. Security experts who are skilled in the latest security in the application domain should conduct these tests.

4.6. Vulnerability Management [1,c5][3*,c24][28,29, 30]*

Using sound coding practices can help substantially reduce software defects commonly introduced during implementation [1]. Such common security defects are categorized and shared with databases: the Common Vulnerabilities and Exposures (CVE) [28], Common Weakness Enumeration (CWE) [29], and Common Attack Pattern Enumeration and Classification (CAPEC) [30]. Programmers can refer to these databases for security implementation, and some tools are available to check common vulnerabilities in codes.

5. Software Security Tools

5.1. Security Vulnerability Checking Tools [1,c6][25]*

Some source code analysis tools check codes to detect security problems such as security vulnerabilities. However, identifying security vulnerabilities is complicated by the fact that they often appear in hard-to-produce software states or crop up in unusual circumstances [1].

Security analysis tools can help, but they cannot find all vulnerabilities.

5.2. Penetration Testing Tools [2,c4]*

Penetration testing tests a system in its final production environment. These tools submit malformed, malicious and random data to a system's entry points in an attempt reveal faults — a process commonly called *fuzzing* [2].

6. Domain-Specific Software Security

6.1. Security for Container and Cloud [31]

Cloud infrastructure and services are often inexpensive and easy to provision, which can quickly lead to having many assets strewn all over the world and forgotten. These forgotten assets are like a ticking time bomb, waiting to explode into a security incident [31].

One important difference with cloud environments is that physical assets and protection are generally not a concern. Developers can gleefully outsource asset tags, anti-tailgating, slab-to-slab barriers, placement of data center windows, cameras, and other physical security and physical asset tracking controls [31].

6.2. Security for IoT Software [32,33]

As part of today's IoT (Internet of Things), systems are interconnected with many other devices, especially back-end systems suffering from all the well-known security flaws inherent in today's business IT. Attackers gaining access to business IT platforms, for instance, by exploiting browser vulnerabilities, will likely also gain access to weakly protected IoT industrial devices. This can cause severe damage, including safety incidents. Hence, the introduction of a massive number of end points from the consumer or industrial environment creates fertile ground for the exploitation of weak links. Hardening these end points, securing device-to-device communications, and ensuring device and information credibility in what until now have been closed, homogeneous systems present new challenges. Comprehensive risk and threat analysis methods, as well as management tools for IoT platforms, are required [33].

6.3. Security for Machine Learning-Based Application [39,c8]

Although machine learning techniques are widely used in many systems, machine learning presents a specific vulnerability: Attackers can change the decisions of machine learning models. There are two kinds of attacks: model poisoning, which attacks training data, and evasion, which attacks inputs to trained models [39].

MATRIX OF TOPICS VS. REFERENCE MATERIAL

Topic	Allen et al. 2018 [1*]	McGrow 2006 [2*]	Bishop 2019 [3*]	Bell 2017 [4*]

Software Security

1. Software Security Fundamentals				
1.1. Software Security				
1.2. Information Security				
1.3. Cybersecurity			Ch. 23	
2. Security Management and Organization	Ch. 7			
2.1. Capability Maturity Model			Ch. 22	
2.2. Information Security Management System				
2.3. Agile Practice for Software Security				Ch. 15, Ch. 16
3. Software Security Engineering and Processes				Ch. 9
3.1. Security Engineering and Secure Development Life Cycle	Ch. 1			Ch. 4
3.2. Common Criteria for Information Technology Security Evaluation			Ch. 22, Ch. 25	
4. Security Engineering for Software Systems			Ch. 1, Ch. 15, Ch. 1, Ch. 3	

4.1. Security Requirements	Ch. 3	Ch. 2	Ch. 20, Ch. 31	Ch. 5, Ch. 8
4.2. Security Design	Ch. 4	Ch. 5	Ch. 20, Ch. 31	Ch. 8
4.3. Security Patterns	Ch. 4			
4.4. Construction for Security	Ch. 5		Ch. 20, Ch. 31	
4.5. Security Testing	Ch. 5		Ch. 24, Ch. 31	Ch. 10, Ch. 11
4.6. Vulnerability Management			Ch. 24	Ch. 6
5. Software Security Tools				
5.1. Security Vulnerability Checking Tools	Ch. 6			Ch. 6
5.2. Penetration Testing Tools	Ch. 4		Ch. 31	Ch. 11, Ch. 12
6. Domain-Specific Software Security				
6.1. Security for Container and Cloud				
6.2. Security for IoT Software				

6.3. Security for Machine Learning-Based Application				

FURTHER READING

- [A] John Viega, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2011.
- [B] Loren Kohnfelder, *Designing Secure Software: A Guide for Developers*, No Starch Press, 2021.
- [C] C. Warren Axelrod, *Engineering Safe and Secure Software Systems*, Artech House Publishers, 2012.

REFERENCES

- [1*] Julia H. Allen, Sean J. Barnum, Robert J. Ellison, Gary McGraw, and Nancy R. Mead, *Software Security Engineering: A Guide for Project Managers*, Addison-Wesley Professional, 2008.
- [2*] Gary McGraw, *Software Security: Building Security In*, Addison-Wesley Professional, 2006.
- [3*] Matt Bishop, *Computer Security*, 2nd Edition, Addison-Wesley Professional, 2019.
- [4*] Laura Bell, Michael Brunton-Spall, Rich Smith, Jim Bird, *Agile Application Security*, O'Reilly, 2017.
- [5] Tony Hsiang-Chih Hsu, *Hands-On Security in DevOps: Ensure continuous security, deployment, and delivery with DevSecOps*, Packt Publishing, 2018.
- [6] Tony Hsiang-Chih Hsu, *Practical Security Automation and Testing: Tools and techniques for automated security scanning and testing in DevSecOps*, Packt Publishing, 2019.
- [7] Glenn Wilson, *DevSecOps: A leader's guide to producing secure software without compromising flow, feedback and continuous improvement*, Rethink Press, 2020.
- [8] Liz Rice, *Container Security: Fundamental Technology Concepts That Protect Containerized Applications*, O'Reilly & Associates Inc., 2020.
- [9] ISO/IEC/JTC1 SC27 Standards: Trustworthiness, Cryptography, Data security, Cryptography, Security evaluation and testing, Security control, Identity management and privacy technologies.
- [10*] ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.
- [11*] ISO/IEC 27000:2018 Information technology — Security techniques — Information security management systems — Overview and vocabulary.
- [12*] ISO/IEC 27032:2012 Information technology — Security techniques — Guidelines for cybersecurity.

- [13] ISO/IEC 19770-1:2017 Information technology — IT asset management — Part 1: IT asset management systems — Requirements.
- [14] ISO/IEC 21827:2008 Information technology — Security techniques — Systems Security Engineering — Capability Maturity Model (SSE-CMM).
- [15*] ISO/IEC 27001:2013 Information technology — Security techniques — Information security management systems — Requirements.
- [16] Michael Howard and Steve Lipner, *The Security Development Lifecycle*, Microsoft Press, 2006.
- [17] Frank Swiderski, Window Snyder, *Threat Modeling: Design for Security*, Wiley, 2014.
- [18] Donald Firesmith, "Security use cases," *Journal of Object Technology*, Vol. 2, No. 1, pp. 53-64, 2003.
- [19] Eduardo Fernandez-Buglioni, *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*, Wiley, 2013.
- [20] Christopher Nagappan, Ramesh Lai, Ray Steel, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, Prentice Hall, 2005.
- [21] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad, *Security Patterns: Integrating Security and Systems Engineering*, Wiley, 2006.
- [22] Robert C. Seacord, *The CERT C Secure Coding Standard*, Addison-Wesley Professional, 2008.
- [23] Robert C. Seacord, *Secure Coding in C and C++*, Addison-Wesley Professional, 2013.
- [24] David Long, Fred Mohindra, Dhruv Seacord, Robert C. Sutherland, Dean F. Svoboda, *The CERT Oracle Secure Coding Standard for Java*, 2011.
- [25] Jon Erickson, *Hacking: The Art of Exploitation*, 2nd Edition, No Starch Press, 2008.
- [26] Karen Scarfone, Murugiah Souppaya, Amanda Cody, Angela Orebaugh, *Technical Guide to Information Security Testing and Assessment*, NIST SP800-115, 2008.
- [27] PCI Security Standards Council, PCI DSS: Payment Card Industry Data Security Standard, Version 3.2, 2017.
- [28] *MITRE*, "Common Vulnerabilities and Exposures (CVE)," https://cve.mitre.org/.
- [29] *MITRE*, "Common Weakness Enumeration (CWE)," https://cwe.mitre.org/.
- [30] *MITRE*, "Common Attack Pattern Enumeration and Classification (CAPEC)," https://capec.mitre.org/.
- [31] Chris Dotson, *Practical Cloud Security*, O'Reilly, 2019.
- [32] "Internet of Things Security Best Practices," *IEEE*, 2017, https://internetinitiative.ieee.org/resources/reports-presentations-publications.

- [33] "IoT 2020: Smart and secure IoT platform," *IEC*, 2016, https://www.iec.ch/basecamp/iot-2020-smart-and-secure-iot-platform.

- [34] ISO/IEC 15408-1:2009 Information technology — Security techniques — Evaluation criteria for IT security — Part 1: Introduction and general model.

- [35] ISO/IEC 18045:2008 Information technology — Security techniques — Methodology for IT security evaluation.

- [36] DoD Enterprise DevSecOps, https://software.af.mil/dsop/documents/.

- [37] Chuck Easttom, *Computer Security Fundamentals*, 4th Edition, Pearson IT Certification, 2019.

- [38] Yuri Diogenes, Erdal Ozkaya, *Cybersecurity — Attack and Defense Strategies*, Second Edition, Packt Publishing, 2019.

- [39] Clarence Chio, David Freeman, *Machine Learning and Security: Protecting Systems with Data and Algorithms*, O'Reilly, 2018.

- [40] Ian Sommerville, *Software Engineering*, 10th ed., Addison-Wesley, 2016.

CHAPTER 14

SOFTWARE ENGINEERING PROFESSIONAL PRACTICE

ACRONYMS

ACM	Association for Computing Machinery
CCPA	The California Consumer Privacy Act
EEA	European Economic Area
ENAEE	European Network for Accreditation of Engineering Education
EU	European Union
GDPR	The General Data Protection Regulation
IEA	International Engineering Alliance
IEC	International Electrotechnical Commission
IEEE-CS	Institute for Electrical and Electronics Engineers — Computer Society
IFIP	International Federation for Information Processing
IP	Intellectual property
ISO	International Organization for Standardization
NDA	Nondisclosure agreement
UI/UX	User interface/user experience
WIPO	World Intellectual Property Organization
WTO	World Trade Organization

INTRODUCTION

The Software Engineering Professional Practice knowledge area (KA) is concerned with the knowledge, skills, and attitudes software engineers must possess to practice software engineering in a professional, responsible and ethical manner. Because of the widespread applications of software products in social and personal life, software product quality can profoundly affect personal well-being and societal harmony. Software engineers must handle unique engineering problems to produce software with known characteristics and reliability. This requirement calls for software engineers who possess the proper knowledge, skills, training, and experience in professional practice.

Professional practice refers to a way of conducting services to achieve certain standards or criteria in both the process of performing a service and the end product resulting from the service. These standards and criteria can include both technical and non-technical aspects. The concept of professional practice is especially applicable to professions with a generally accepted body of knowledge; code of ethics and professional conduct with penalties for violations; accepted processes for accreditation, certification, qualification, and licensing; and professional societies to provide and administer all these. Admission to these professional societies is often predicated on a prescribed combination of education and experience.

A software engineer maintains professional practice by performing all work following generally accepted practices, standards, and guidelines set forth by the applicable professional society, such as the Association for Computing Machinery (ACM), Institute for

Electrical and Electronics Engineers (IEEE), or International Federation for Information Processing (IFIP). The Institute for Electrical and Electronics Engineers — Computer Society (IEEE-CS), International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), and ISO/IEC/IEEE provide internationally accepted software engineering standards. All of these standards and guidelines comprise the foundation of the professional practice of software engineering.

Figure 14.1. Breakdown of Topics for the Software Engineering Professional Practice KA

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING PROFESSIONAL PRACTICE

The Software Engineering Professional Practice KA's breakdown of topics is shown in Figure 14.1.
The subareas presented in this KA are professionalism, group dynamics and psychology, and communication skills.

1. Professionalism

A software engineer displays professionalism notably by adhering to a code of ethics and professional conduct and to standards and practices established by the engineer's professional community.

One or more professional societies often represent a professional community, and this is the case for the engineering community. Those societies publish a code of ethics and professional conduct as well as criteria for admittance to the community. Those criteria form the basis for accreditation and licensing activities and may determine engineering competence or negligence.

As software is used more widely and deeply in society, stakeholders' requirements have likewise become wider and deeper. And as software has become socially vital, software engineers have worked to base the user interface/user experience (UI/UX) on socially inclusive concepts.

1.1. Accreditation, Certification and Qualification, and Licensing [1, cls4-s5, cls10] [2] [4*, c12s10] [6] [7] [8] [9]*

1.1.1. Accreditation

Accreditation certifies an organization's competency, authority, or credibility. Accredited schools or programs have shown that they adhere to particular standards and maintain certain qualities. In many countries, the basic means by which engineers acquire knowledge is by completing an accredited course of study. Often, the accreditation process is independent of the government and is performed by private membership associations. There are two major global accreditation organizations. One is the International Engineering Alliance (IEA), of which the Washington Accord is a constituent. The other is the European Network for Accreditation of Engineering Education (ENAEE), which administers EUR-ACE®, the label awarded to engineering degree programs at the bachelor's and master's levels, listed by the European Commission.

Although the accreditation process might differ for each country and jurisdiction, the general meaning is the same. Accreditation of an institution's course of study means "the accreditation body recognizes an educational institution as maintaining standards that qualify the graduates for admission to higher or more specialized institutions or professional practice."

1.1.2. Certification and Qualification

ISO/IEC 24773-1 Software and Systems Engineering — Certification of Software and Systems Engineering Professionals — Part 1: General Requirements define certification and qualification. Certification contains recertification. Qualification is similar to certification but does not require re-qualification. *Certification* refers to the confirmation of a person's particular characteristics. A common type of certification is professional certification, which certifies a person as being able to complete an activity in a certain discipline at a stated level of

competency. Professional certification can verify the holder's ability to meet professional standards and to apply professional judgment in solving or addressing problems. Professional certification can also verify prescribed knowledge, mastery of best practices and proven methodologies, and professional experience.

An engineer usually obtains certification by passing an examination in addition to meeting other experience-based criteria. Nongovernmental organizations, such as professional societies, often administer these examinations. In software engineering, certification testifies to one's capability as a software engineer.

The IEEE-CS has enacted one qualification program, called Associate Software Developer Certification, and two certification programs, Professional Software Developer Certification and Professional Software Engineering Master Certification. The qualification and certification programs are designed to confirm a software engineer's knowledge of standard software engineering practices and to advance the engineer's career. A lack of qualification or certification does not exclude the individual from working as a software engineer. Qualification or certification in software engineering is voluntary. Most software engineers are not qualified or certified under any program.

1.1.3. Licensing

Licensing authorizes a person to perform certain activities and take responsibility for resultant engineering products. The noun *license* refers to both that authorization and the document recording that authorization. Governmental authorities or statutory bodies usually issue licenses.

Obtaining a license to practice requires an individual to meet a certain standard at a certain ability to practice or operate. Sometimes an entry-level requirement sets the minimum skills and capabilities to practice, and as the professional moves through their career, the required skills and capabilities change and evolve.

Engineers are licensed to protect the public from unqualified individuals. In some countries, no one can practice as a professional engineer unless licensed; further, no company may offer "engineering services" unless at least one licensed engineer is employed there.

1.2. Codes of Ethics and Professional Conduct

[1*, cls7-cls9, c10s2, Appendix] [3*, c8] [4*, cls2] [5*, c33] [10] [11] [13*]

Codes of ethics and professional conduct describe the values and behavior that an engineer's professional practice and decisions should embody. The professional community establishes a code of ethics and professional conduct. This code exists in the context of societal norms and local laws and is adjusted to agree with those norms and laws as needed. A code of ethics and professional conduct can offer guidance in the face of conflicting imperatives. More than one such code serves the professional engineering community. For example, in 1999, IEEE-CS and ACM launched a joint Software Engineering Ethics and Professional Practices Task Force to publish a code of ethics. In 2018, ACM published its ACM Code of Ethics and Professional Conduct, and in 2020, IEEE published its IEEE Code of Ethics. Then, in 2021, IFIP published its Code of Ethics and Professional Conduct, adapted from ACM's Code of Ethics and Professional Conduct.

Once established, codes of ethics and professional conduct are enforced by the profession, as represented by professional societies or by a statutory body. Violations may be acts of commission, such as concealing inadequate work, disclosing confidential

information, falsifying information, or misrepresenting abilities. They may also occur through omission, including failure to disclose risks or provide important information, failure to give proper credit or acknowledge references, and failure to represent client interests. Violations of a code of ethics and professional conduct may result in penalties and possible expulsion from professional status.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession. Following their commitment to the health, safety, and welfare of the public, software engineers shall adhere to the eight principles concerning the public, client and employer, product, judgment, management, profession, colleagues, and self.

Since the code of ethics and professional conduct may be introduced, modified, or replaced at any time, individual software engineers are responsible for continuing their studies to stay current in their professional practice.

1.3. Nature and Role of Professional Societies [1, c2s3] [4*, c1s2] [5*, c35s1]*

Professional societies comprise a mix of practitioners and academics. These societies define, advance, and regulate their corresponding professions. Professional societies help establish professional standards as well as codes of ethics and professional conduct. They also engage in related activities, which include the following:

- Establishing and promulgating a body of generally accepted knowledge
- Accrediting, certifying, and licensing
- Dispensing disciplinary actions
- Advancing the profession through conferences, training, and publications

Participation in professional societies assists individual engineers in maintaining and sharpening their professional knowledge and relevancy and in expanding and maintaining their professional network.

1.4. Nature and Role of Software Engineering Standards [1, c10s2] [2] [4*] [5*, c32s6]*

Software engineering standards cover a remarkable variety of topics. They provide guidelines for the practice of software engineering and for processes to be used during the development, maintenance, and support of software. By establishing a common body of knowledge and experience, software engineering standards establish a basis on which further guidelines may be developed. Appendix B of this *Guide* presents IEEE, ISO/IEC, and ISO/IEC/IEEE software engineering standards that support this *Guide*'s KAs.

The benefits of software engineering standards are many; for example, such standards improve software quality, help prevent errors, protect both software producers and users, increase professional discipline, and support the technology transition.

1.5.Economic Impact of Software [3, c1s1, c10s8] [4*, c1s1] [13*]*

The software has economic effects at the individual, business, and societal levels. For example, software "success" may be determined by a product's suitability for a recognized problem and by its effectiveness when applied to that problem. At the individual level, an engineer's continuing employment may depend on their ability and willingness to interpret and execute tasks in meeting customers' or employers' needs and expectations. The customer's or the employer's financial situation may be positively or negatively affected by software purchases.

At the business level, software properly applied to a problem can eliminate months of work and translate to elevated profits or more effective organizations. Organizations that acquire or provide successful software may become a boon to the society in which they operate by providing both employment and improved services. However, the software's development or acquisition costs can be considerable, like those of any major acquisition.

At the societal level, direct impacts of software success or failure include the avoidance or experience of accidents, interruptions, and loss of service. Indirect impacts include the success or failure of the organization that acquired or produced the software, increased or decreased societal productivity, harmonious or disruptive social order, and even the saving or loss of property or life. In addition, as digitalization progresses, easier and faster access to the information needed may bring higher social value.

1.6. Employment Contracts [1, c6, c7] [10] [11] [12]*

Software engineering services may be provided under a variety of client-engineer relationships. For example, the work may be done through a company-to-customer supplier arrangement, an engineer-to-customer consultancy arrangement, a direct-hire, or even through volunteering. In these situations, the customer and supplier agree that a product or service will be provided in return for some consideration. Here, we are most concerned with engineer-to-customer arrangements and their attendant agreements or contracts, whether they are of the direct-hire or consultant variety, and the issues they typically address.

A common concern in software engineering contracts is confidentiality. Employers derive commercial advantage from intellectual property (IP), so they strive to protect that property from disclosure. Therefore, software engineers are often required to sign nondisclosure agreements (NDA) or IP agreements as a precondition to working. These agreements typically apply to information the software engineer could gain only through association with the customer. The terms of these agreements may extend past the association's termination.

Another concern is IP ownership. Rights to software engineering assets — products, innovations, inventions, discoveries, and ideas — may reside with the employer or customer, under explicit contract terms or relevant laws, if those assets are obtained during the software engineer's relationship with that employer or customer. Contracts differ in the ownership of assets created using non-employer-owned equipment or information.

Finally, contracts can also specify, among other elements:

- The location at which work is performed
- Standards to which that work will be held
- The system configuration used for development
- Limitations of the software engineer's and employer's liability
- A communication matrix and/or escalation plan
- Administrative details such as rates, frequency of compensation, working hours, and working conditions

1.7. Legal Issues [1, c6, c11] [2] [3*, c5s3–c5s4] [4*, c12s3, c13s2]*

Legal issues surrounding software engineering professional practice include matters related to standards, trademarks, patents, copyrights, trade secrets, professional liability, legal requirements, trade compliance, cybercrime, and data privacy. It is therefore beneficial to know these issues and their applicability. In addition, legal issues are jurisdictionally based,

so software engineers must consult attorneys who specialize in the type and jurisdiction of any identified legal issues.

1.7.1. Standards

Software engineering standards establish guidelines for generally accepted practices and minimum requirements for products and services provided by a software engineer. Appendix B of this *Guide* presents software engineering standards for each KA.

Standards are valuable sources of information about requirements and other guidance that can support software engineers in everyday activities. Adherence to standards promotes discipline by enumerating minimal characteristics of products and practices. That discipline helps mitigate subconscious assumptions or overconfidence in a design. For these reasons, organizations performing software engineering activities often include conformance to standards as part of their organizational policies. Further, adherence to standards provides a defense from legal action or allegations of malpractice.

1.7.2. Trademarks

A *trademark* relates to any word, name, symbol, or device used in business transactions. It is used "to indicate the source or origin of the goods." Trademark protection protects names, logos, images, and packaging. However, if a name, image, or other trademarked asset becomes a generic term, trademark protection is nullified.

The World Intellectual Property Organization (WIPO) is the authority that frames the rules and regulations on trademarks. WIPO is the United Nations agency dedicated to protecting the use of IP as a means of stimulating innovation and creativity.

1.7.3. Patents

Patents protect an inventor's right to manufacture and sell an idea. A patent consists of exclusive rights granted by a sovereign government to an individual, group of individuals, or organization for a limited period. Patents are an old form of idea-ownership protection and date to the 15th century.

Application for a patent entail keeping and producing careful records of the process that led to the invention. In addition, patent attorneys help write patent disclosure claims in a manner most likely to protect the software engineer's rights. Note that if inventions are made during a software engineering contract, ownership may belong to the employer or customer or be jointly held rather than belong to the software engineer.

Rules vary concerning what is and what is not patentable. In many countries, software code is not patentable, but software algorithms may be. Existing and filed patent applications can be found at WIPO.

1.7.4. Copyrights

Most governments give exclusive rights of an original work to its creator, usually for a limited time, enacted as copyright. Copyrights protect the way an idea is presented — not the idea itself. For example, they may protect the particular wording of an account of a historical event, whereas the event itself is not protected. Copyrights are long-term and renewable. As a form of IP, they date to the 17th century.

1.7.5. Trade Secrets

In many countries, an intellectual asset such as a formula, algorithm, process, design, method, pattern, instrument, or compilation of information may be considered a *trade secret*, provided the asset is not generally known and may provide a business with some economic advantage. The "trade secret" designation provides legal protection if the asset is stolen. This protection is not subject to a time limit. However, if another party derives or discovers

the same asset legally, then the asset is no longer protected and the other party will also possess all rights to use it.

1.7.6. Professional Liability

It is common for software engineers to be concerned with professional liability matters. As engineers provide services to a client or employer, it is crucial that they adhere to standards and generally accepted practices to protect themselves against allegations of or proceedings related to malpractice, negligence, or incompetence.

For engineers (including software engineers), professional liability is related to product liability. Under the laws and rules of their jurisdiction, engineers may be held accountable for failing to fully and conscientiously follow recommended practice; this is known as *negligence*. They may also be subject to laws governing *strict liability* and implied or express warranty, where, by selling the product, the engineer is held to warrant that the product is both suitable and safe for use. In some countries (e.g., in the US), *privity* (a doctrine under which one can sue only the person selling the product) is no longer a defense against liability actions.

Legal suits for liability can be brought under tort law in the US, allowing anyone who is harmed to recover their loss even if no guarantees were made. Because it is difficult to measure the suitability or safety of software, failure to take due care can be used to prove negligence on the part of software engineers. Engineers can defend themselves against such an allegation by showing that they followed standards and generally accepted practices in developing the product.

1.7.7. Legal Requirements

Software engineers must operate within local, national and international legal frameworks. Therefore, software engineers must know the legal requirements for the following:

- Registration and licensing, including examination, education, experience, and training requirements
- Contractual agreements
- Noncontractual legalities, such as those governing liability

Basic information on the international legal framework can be accessed from the World Trade Organization (WTO).

1.7.8. Trade Compliance

All software professionals must be aware of legal restrictions on the import, export, or re-export of goods, services, and technology in the jurisdictions in which they work. Such rules often concern export controls and classification; transfer of goods; acquisition of necessary governmental licenses for foreign use of hardware and software; services and technology by sanctioned nations, enterprises, or individual entities; and import restrictions and duties. Trade experts should be consulted for detailed compliance guidance.

1.7.9. Cybercrime

Cybercrime refers to any crime that involves a computer, computer software, computer networks, or embedded software controlling a system. The computer or software may have been used in the commission of a crime or have been the target of the crime. This category of crime includes fraud, unauthorized access, spam, obscene or offensive content, threats, harassment, theft of sensitive personal data or trade secrets, and use of one computer to damage or infiltrate other computers and automated system controls.

Computer and software users commit fraud by altering electronic data to facilitate illegal activity. Forms of unauthorized access include hacking, eavesdropping, and using computer systems in a way that is concealed from their owners. Many countries have laws that specifically cover cybercrimes, but

many do not have effective statutes, making cybercrime difficult to prosecute in some cases. The software engineer has a professional obligation to consider the threat of cybercrime and to consider how the software system's security will protect the software and user information from accidental or malicious access, use, modification, destruction, or disclosure.

Dark patterns are deceptive UI/UX interactions designed to mislead or trick users into making them do something they may not want to do. These patterns do not have the users' interests in mind and aim for exploitability rather than usability. Creating dark patterns is not good ethical practice. Software engineers should be responsible for their actions and be transparent with users instead of manipulating them.

1.7.10. Data Privacy

Software engineers should know that data privacy is a key legal requirement in many countries. The General Data Protection Regulation (GDPR), adopted on April 14, 2016, and enforceable since May 25, 2018, regulates data protection and privacy in the European Union (EU) and the European Economic Area (EEA). It also addresses the transfer of personal data outside the EU and EEA areas. The GDPR's primary aim is to enhance individuals' control and rights over their data and to simplify the regulatory environment for international business.

The regulation became a model for many national laws outside the EU, including the UK, Chile, Japan, Brazil, South Korea, Argentina, and Kenya. The California Consumer Privacy Act (CCPA), adopted on June 28, 2018, has many similarities with the GDPR.

1.8. Documentation [1, c10s5.8] [3*, c1s5] [4*] [5*, c32]*

Providing clear, thorough, and accurate documentation is the responsibility of each software engineer. The adequacy of documentation is judged according to different criteria, based on stakeholder needs. Good documentation complies with accepted standards and guidelines. In particular, software engineers should document the following:

- Relevant facts
- Significant risks and trade-offs
- Warnings of undesirable or dangerous consequences from the use or misuse of the software

Software engineers should avoid:

- Certifying or approving unacceptable products
- Disclosing confidential information
- Falsifying facts or data

In addition, software engineers and their managers should provide the following documentation for other elements of the software development organization to use:

- Software requirements specifications, software design documents, details on the software engineering tools used, software test specifications and results, and details about the adopted software engineering methods
- Problems encountered during the development process

For external stakeholders (customers, users, others), software documentation should provide the following:

- Information needed to determine whether the software is likely to meet customer and user needs
- Description of safe and unsafe use of the software

- Explanation of how to protect sensitive information created by or stored using the software
- Clear identification of warnings and critical procedures

Software use may include installation, operation, administration, and performance of other functions by various groups of users and support personnel. If the customer will acquire ownership of the software source code or the right to modify the code, the software engineer should provide documentation of the functional specifications, the software design, the test suite, and the necessary operating environment for the software. Documents should be kept for at least as long as the software product's life cycle or the time required by relevant organizational or regulatory requirements.

1.9. Trade-Off Analysis [3, c1s2, c10] [4*, c7s2, c13s4] [13*, c9s5.10]*

A software engineer often has to choose between alternative problem solutions. The outcome of these choices is determined by the software engineer's professional evaluation of each alternative's risks, costs, and benefits in cooperation with stakeholders. The software engineer's evaluation is called *trade-off analysis*. Trade-off analysis notably identifies competing and complementary software requirements to prioritize the final requirements defining the software to be constructed. (See Requirements Negotiation in the Software Requirements KA and Determination and Negotiation of Requirements in the Software Engineering Management KA.)

When an ongoing software development project is late or over budget, a trade-off analysis is often conducted to decide which software requirements can be relaxed or dropped given the effects thereof. The first step in a trade-off analysis is establishing design goals (see Engineering Design in the Engineering Foundations KA) and setting the relative importance of those goals. This permits the identification of the solution that most nearly meets those goals; this means that the way the goals are stated is critically important.

Design goals may include minimizing monetary cost and maximizing reliability, performance, or other criteria on various dimensions. However, it is difficult to formulate a trade-off analysis of cost against risk, especially where primary production and secondary risk-based costs must be weighed against each other.

A software engineer must ethically conduct a trade-off analysis — notably by being objective and impartial when selecting criteria for comparing alternative problem solutions and assigning weights or importance to these criteria. In addition, any conflict of interest must be disclosed upfront.

2. Group Dynamics and Psychology

Engineering work is often conducted in teams. A software engineer must interact cooperatively and constructively with others to first determine and then meet needs and expectations. Knowledge of group dynamics and psychology is an asset when interacting with customers, coworkers, suppliers, and subordinates to solve software engineering problems.

2.1. Dynamics of Working in Teams/Groups [3, c1s6] [14*, c1s3.5, c10]*

Software engineers must work with others. On the one hand, they work internally in engineering teams; on the other hand, they work with customers, members of the public, regulators, and other stakeholders. Performing teams — those who demonstrate a consistent quality of work and progress toward goals — are cohesive and possess a cooperative, honest and focused atmosphere. Individual and team goals are aligned so the members naturally

commit to and feel ownership of shared outcomes.

Team members facilitate this atmosphere by being intellectually honest, using group thinking, admitting ignorance, and acknowledging mistakes. They share responsibility, rewards, and workload fairly. They communicate clearly and directly to one another and in documents and source code so information is accessible to everyone. Peer reviews about work products are framed in a constructive and nonpersonal way. (See Reviews and Audits in the Software Quality KA.) This allows all the members to pursue a continuous improvement and growth cycle without personal risk. Members of cohesive teams demonstrate respect for one another and their leader.

One point to emphasize is that software engineers must be able to work in multidisciplinary environments and varied application domains. Because software is everywhere, from phones to cars, it affects people's lives far beyond the more traditional concept of software made for information management in a business environment.

2.2. Individual Cognition [3, c1s6.5] [5*, c33]*

Engineers want to solve problems. Every engineer strives to solve problems effectively and efficiently. However, the limits and processes of individual cognition affect problem-solving. Individual cognition plays a prominent role in problem-solving in software engineering, in part because of the highly abstract nature of software itself.

An individual's (in particular, a software engineer's) ability to decompose a problem and creatively develop a solution can be inhibited by the following:

- The need for more knowledge
- Subconscious assumptions
- The volume of data
- Fear of failure or the consequence of failure
- Culture, either of the application domain or the organization
- Lack of ability to express the problem
- Perceived working atmosphere
- The individual's emotional status

The effects of these inhibiting factors can be reduced by cultivating good problem-solving habits that minimize the impact of misleading assumptions. The ability to focus is crucial, as is intellectual humility. Both allow a software engineer to suspend personal considerations and consult with others freely, which is especially important when working in teams.

Engineers use basic methods to facilitate problem-solving. (See Problem-Solving Techniques in the Computing Foundations KA.) Breaking down problems and solving them one piece at a time reduces cognitive overload. By taking advantage of professional curiosity and pursuing continuous professional development, engineers gain skills and knowledge. Reading, networking, and experimenting with new tools, techniques and methods are all valid means of professional development.

2.3. Dealing with Problem Complexity [3, c3s2] [4*, c1s1, c20s1-s5] [5*, c33]*

Many, if not most, software engineering problems are too complex and difficult to address as a whole or to be tackled by individual software engineers. When such circumstances arise, engineers typically use teamwork and problem decomposition. (See Problem-Solving Techniques in the Computing Foundations KA.)

Teams work together to deal with large, complex problems by sharing burdens and drawing on one another's knowledge and creativity. When software engineers work in

teams, individual engineers' different views and abilities complement one another and help build a solution otherwise difficult to come by. Some teamwork examples in software engineering are pair programming (see Agile Methods in the Software Engineering Models and Methods KA) and code review (see Reviews and Audits in the Software Quality KA).

Interacting with Stakeholders [4]*

The success of a software engineering endeavor depends on positive interactions with stakeholders. Stakeholders should provide support, information, and feedback at all stages of the software life cycle. For example, during the early stages, it is critical to identify all stakeholders and discover how the product will affect them to properly capture a sufficient definition of stakeholder requirements.

In Agile software development, the involvement of stakeholders is even more essential than in other types of development. First, during development, stakeholders may provide feedback on specifications or early versions of the software, changes of priority, and clarification of detailed or new software requirements. Last, during software maintenance and until the end of product life, stakeholders can provide feedback on evolving or new requirements and problem reports so the software can be extended and improved. Clearly, regular stakeholder involvement will lead to a better application. It is vital to maintain open and productive communication with stakeholders during the software product's life cycle.

2.5. Dealing with Uncertainty and Ambiguity [4, c4s1, c4s4, c11s5, c24s5] [14*, c9s4]*

As with engineers in other fields, software engineers must often deal with and resolve uncertainty and ambiguities while providing services and developing products. The software engineer must reduce or eliminate any lack of clarity that is an obstacle to performing work.

Often, uncertainty reflects a lack of knowledge. If that is the case, investigating the issue by reading formal sources such as textbooks and professional journals, interviewing stakeholders, or consulting with teammates and peers can likely solve the problem.

When uncertainty or ambiguity cannot be overcome easily, software engineers or organizations may regard it as a project risk. When this is the case, work estimates or pricing are adjusted to mitigate the anticipated cost of addressing it. (See Risk Management in the Software Engineering Management KA.)

2.6. Dealing with Equity, Diversity, and Inclusivity [4] [14*, c10s7]*

The equity, diversity, and inclusivity environment can affect a group's dynamics. This is especially true when the group is geographically separated or communication is infrequent because such separation elevates the importance of each contact. Intercultural communication is even more difficult if the difference in time zones makes oral communication less frequent.

Multicultural environments are prevalent in software engineering, perhaps more than in other engineering fields, because of the strong trend of international outsourcing and the easy shipment of software components instantaneously around the globe. For example, it is common for a software project to be divided into pieces across national and cultural borders. It is also common for a software project team to consist of people from diverse cultural backgrounds.

For a software project to succeed, team members must embrace tolerance of different cultural and social norms, acknowledging that

not all societies have the same social expectations. The support of leadership and management can facilitate tolerance and understanding. More frequent communication, including face-to-face meetings, can help mitigate geographical and cultural divisions, promote cohesiveness, and raise productivity. Also, communicating with teammates in their native language could be beneficial.

In the software industry, gender bias is still prevalent. Implementing broader recruiting strategies, specific and measurable performance evaluation criteria, and transparent procedures for assigning compensation can reduce gender inequality in the software industry. These trends can contribute to building a diverse environment for all software engineers, regardless of their gender.

3. Communication Skills

A software engineer must communicate well, both orally and in reading and writing. To meet software requirements and deadlines, engineers must establish clear communication with customers, supervisors, coworkers, and suppliers. Optimal problem-solving is made possible through the ability to investigate, comprehend and summarize information. Customer product acceptance and safe product use depend on relevant training and documentation. The software engineer's career success is affected by consistently providing oral and written communication effectively and on time.

3.1. Reading, Understanding, and Summarizing [4, c4s5] [5*, c33s3]*

Software engineers must be able to read and understand technical material. Technical material includes reference books, manuals, research papers, and program source code.

Reading is not only a primary way of improving skills but also a way of gathering information for completing engineering goals. A software engineer sifts through accumulated information, focusing on the pieces that will be most helpful. Customers may request that a software engineer summarize the results of such information-gathering for them, simplifying or explaining it so that they can make the final choice among competing solutions.

Reading and comprehending source code are also components of information-gathering and problem-solving. For example, when engineers modify, extend or rewrite software, they must understand both its implementation, directly derived from the presented code, and its design, which must often be inferred.

3.2. Writing [3, c1s5] [4*, c4s2-s3]*

Software engineers can produce written products requested by customers or required by generally accepted practice. These written products may include source code, software project plans, software requirement documents, risk analyses, software design documents, software test plans, user manuals, technical reports and evaluations, justifications, diagrams and charts, and so forth.

Clear, concise writing is important because writing is often the primary method of communication among relevant parties. In all cases, written software engineering products must be accessible, understandable, and relevant to their intended audience.

3.3. Team and Group Communication [3, c1s6.8] [4*, c22s3] [5*, c27s1] [14*, c10s4]*

Effective communication among team and group members is essential to a collaborative software engineering effort. Stakeholders must be consulted; decisions must be made, and plans must be generated. The greater the

number of team and group members, the greater the need to communicate.

However, the number of communication paths grows quadratically with the addition of each team member. Furthermore, team members are unlikely to communicate with anyone perceived to be removed from them by more than two degrees (levels). This problem can be more serious when software engineering endeavors or organizations are spread across national and continental borders.

Some communication can be accomplished in writing. Software documentation is a common substitute for direct interaction. Email is another, but although it is useful, it is not always enough. Also, if one receives too many messages, it becomes difficult to identify the important information. Increasingly, organizations are using enterprise collaboration tools to share information. In addition, electronic information stores, accessible to all team members for organizational policies, standards, common engineering procedures, and project-specific information, can be beneficial.

Some software engineering teams focus on face-to-face interaction and promote such interaction through office space arrangements. Although private offices improve individual productivity, other arrangements, such as co-locating team members in physical or virtual spaces and providing communal work areas, can boost collaborative efforts.

3.4. Presentation Skills [3, c1s5] [4*, c22] [14*, c10s7–c10s8]*

Software engineers rely on their presentation skills during software life cycle processes. For example, software engineers may walk customers and teammates through software requirements during the phase and conduct formal requirements reviews. (See Requirement Reviews in the Software Requirements KA.) During and after software design, software construction, and software maintenance, software engineers lead reviews, product walkthroughs (see Review and Audits in the Software Quality KA), and training. These require the ability to present technical information to groups and solicit ideas or feedback.

Therefore, the software engineer's ability to convey concepts effectively in a presentation influences product acceptance, management, and customer support. It also influences the ability of stakeholders to comprehend and assist in the product effort. This knowledge needs to be archived in slides, knowledge write-ups, technical white papers, and other material used for knowledge creation.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Bott et al. 2000 [1*]	Voland, 2003 [3*]	Sommerville,2016 [4*]	McConnel, 2004 [5*]	Tockey, 2004 [13*]	Fairley, 2009 [14*]
1. Professionalism						
1.1. Accreditation, Certification and Qualification, and Licensing	c1s5, c1s10		c12s10			
1.2. Codes of Ethics and Professional Conduct	c1s7-s9, c10s2, App		c1s2			
1.3. Nature and Role of Professional Societies	c2s3		c1s2	c35s1		
1.4. Nature and Role of Software Engineering Standards	c10s2,		*	c32s6		
1.5. Economic Impact of Software		c1s1, c10s8	c1s1		*	
1.6. Employment Contracts	c6, c7		*			
1.7. Legal Issues	c6, c11	c5s3-s4,	c12s3, c13s2			
1.8. Documentation	c10s5	c1s5	*	c32		
1.9. Trade-Off Analysis		c1s2, c10	c7s2, c13s4		c9s5-10	
2. Group Dynamics and Psychology						
2.1. Dynamics of Working in Teams/Groups		c1s6				c1s3-5, c10
2.2. Individual Cognition		c1s6		c33		
2.3. Dealing with		c3s2	c1s1, c20s1-s5			

Problem Complexity						
2.4. Interacting with Stakeholders			*			
2.5. Dealing with Uncertainty and Ambiguity			**c4s1, c4s4, c11s5 c24s5**			**c9s4**
2.6. Dealing with Equity, Diversity, and Inclusivity			*			**c10s7**
3. Communication Skills						
3.1. Reading, Understanding, and Summarizing			**c4s5**	**c33s3**		
3.2. Writing		**c1s5**	**c4s2-s3**			
3.3. Team and Group Communication		**c1s6**	**c22s3**	**c27s1**		**c10s4**
3.4. Presentation Skills		**c1s5**	**c22**			**c10s7-s8**

FURTHER READING

Gerald M. Weinberg, *The Psychology of Computer Programming* [15].

This was the first major book to address programming as an individual and team effort; it has become a classic in the field.

Kinney and Lange, P.A., *Intellectual Property Law for Business Lawyers* [16].

This book covers IP laws in the US. It not only talks about what the IP law is; it also explains why it looks the way it does.

REFERENCES

[1*] F. Bott et al., *Professional Issues in Software Engineering*, 3rd ed., Taylor & Francis, 2000.

[2] Appendix B of this Guide.

[3*] G. Voland, *Engineering by Design*, 2nd ed., Prentice-Hall, 2003.

[4*] I. Sommerville, *Software Engineering*, 10th ed., Addison-Wesley, 2016.

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

[6] 25 Years Washington Accord, IEC, 2014.

[7] EUR-ACE, 2017.

[8] ISO/IEC 24773-1, Software and Systems Engineering – Certification of Software and Systems Engineering Professionals — Part 1: General Requirements.

[9] Software Professional Certification Program IEEE-CS, https://www.computer.org/education/certifications.

[10] ACM Code of Ethics and Professional Conduct, 2018.

[11] IEEE Code of Ethics, 2020.

[12] IFIP Code of Ethics and Professional Conduct, 2021.

[13*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, Addison-Wesley, 2004.

[14*] R. E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

[15] G. M. Weinberg, *The Psychology of Computer Programming: Silver Anniversary Edition*, Dorset House, 1998.

[16] Kinney and Lange, P.A., *Intellectual Property Law for Business Lawyers*, Thomson West, 2013

CHAPTER 15

SOFTWARE ENGINEERING ECONOMICS

ACRONYMS

IRR	Internal Rate of Return
MARR	Minimum Acceptable Rate of Return
SDLC	Software Development Life Cycle
SPLC	Software Product Life Cycle
ROI	Return on Investment
SEI	Software Engineering Institute
TCO	Total Cost of Ownership

INTRODUCTION

Software is ubiquitous and has become essential for many organizations. It serves organizations in the following ways:

- as a lever to reach an organization's business or strategic goals;
- as a catalyst of organizational know-how to improve value.

Both aspects lead directly to critical software engineering demands:

- increased productivity;
- long-term sustainability;
- innovation;
- competitiveness;
- alignment with organizational goals.

Software engineering economics helps software engineers work in ways that satisfy these critical demands. The Introduction to *SWEBOK Guide* explains that engineering economics is a key element of all recognized engineering disciplines. Economics is the science of choice, not the science of money. Engineering economics is the applied microeconomics branch of economics. It asks the fundamental question, "Is it in the best interest of this enterprise to invest its limited resources in this technical endeavor, or would the same investment produce a higher return elsewhere?" Paraphrasing the definition in [1], engineering is "finding the balance between what is technically feasible and what is economically acceptable."

Software engineering must be *value-based*. Neutral — or worse, negative — value from an organization's investment in software is not sustainable. Software engineering economics aligns software technical decisions with the organization's business goals.

"The organization" will at least include the organization where the software engineer is employed. However, when the software engineer is involved in work for any third party, such as through an external digital transformation contract or other "works for hire" situation, the business goals of that third party are also relevant.

In all types of organizations — for-profit, nonprofit and government — a value-based approach translates into long-term sustainability. In for-profit organizations this means achieving a tangible return on the software investment. In nonprofit organizations, this means achieving the maximum benefit for the least cost.

Software technical decisions, like an organization's decision to use a preexisting library or to develop its own, may appear easy from a purely technical perspective. However, they can have serious implications for the business viability of a software project as well as the product itself. Most software practitioners wonder whether such concerns apply to them. But economic decision-making is fundamental to engineering. Someone who cannot make decisions from both a technical and an economic perspective cannot be considered a true engineer.

Software engineering economics applies to decisions across the entire software product life cycle (SPLC), from the pre-project decision to develop the software to end-of-life decisions for existing software. It also applies to decisions at all levels of technical detail. For example, all following questions involve an economic perspective:

- can a client organization benefit from a digital transformation?
- does a project proposal (a tender) align with a client's business goals?
- should certain software functionality be bought or built?
- should certain requirements be included in scope or not?
- what is the most efficient, cost-effective architecture and design?
- what is an optimal load-balancing strategy for a cloud-based deployment that provides adequate response time to clients without incurring unnecessary operational cost?
- how much risk-based testing is enough?
- is it better to refactor, redevelop or just live with code that has high technical debt?
- is it better to focus maintenance on adding new functionality or on fixing known defects?
- would the value of early delivery of partial functionality gained by using an Agile process outweigh the overhead of rework and continuous testing inherent in iterative approaches?

The Software Engineering Economics Knowledge Area (KA) is directly or indirectly related to all other KAs in this *Guide*.

This KA also takes the position that the more traditional, purely financial view of engineering economics needs to be broadened [2]. Value does not always derive from money alone; value can also derive from "unquantifiables" like corporate citizenship, employee well-being, environmental friendliness, customer loyalty and so on. Therefore, software engineering decisions must also consider relevant unquantifiable criteria.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING ECONOMICS

The breakdown of topics for the Software Engineering Economics KA is shown in Figure 15.1.

Figure 15.1. Breakdown of Topics for the Software Engineering Economics KA

1. Software Engineering Economics Fundamentals

1.1. Proposals

[3*, c3pp23-24]

Software engineering decisions begin with the concept of a *proposal* — a single, separate course of action to be considered (e.g., carrying out a particular software development project or not). Another proposal could be to enhance an existing software component; another might be to redevelop that same software from scratch. In deciding what algorithm to use in implementing a certain function, each candidate considered is a proposal. Every proposal represents a binary unit of choice — the software engineer either carries out that proposal or chooses not to. Software engineering economics aims to identify the proposals best aligned with the organization's goals.

1.2. Cash Flow

[3*, c3pp24-32]

Engineers must evaluate a proposal from a financial perspective to make a meaningful decision about it. The concepts of *cash flow instance* and *cash flow stream* describe the financial perspective of proposals.

A cash flow instance is a specific amount of money flowing into or out of the organization at a specific time as a direct result of carrying out a proposal. For example, in a proposal to develop and launch product X, the payment for new computers, if needed, would be an example of an outgoing cash flow instance. Money would need to be spent to carry out that proposal. The sales income from product X in the 11th month after market launch would be an example of an incoming cash flow instance. Money would come in because of carrying out the proposal.

A cash flow stream is the set of cash flow instances over time caused by carrying out that proposal. The cash flow stream is that proposal's complete financial view. How much money goes out? When does it go out? How much money comes in? When does it come in? If the cash flow stream for Proposal A is more desirable than the cash flow stream for Proposal B, then — all other things being equal — the organization is financially better off carrying out Proposal A than Proposal B. Thus, the cash flow stream is an important element of engineering decision-making.

A *cash flow diagram* is a picture of a cash flow stream. The cash flow diagram quickly summarizes the financial view of a proposal. Figure 15.2 shows an example cash flow diagram.

Figure 15.2. A Cash Flow Diagram

The cash flow stream is shown in two dimensions. Time runs from left to right and amounts of money run up and down. The horizontal axis is divided into units representing years, months, weeks, etc., as appropriate for the proposal. Each net cash flow instance is drawn at a left-to-right position relative to its timing. The amount of the cash flow instance is shown as an upward or downward arrow. Upward arrows indicate that money is coming in (income), whereas downward arrows indicate that money is spent (expense). The arrow's length is usually proportional to the net amount.

1.3. Time-Value of Money

[3*, c5-6]

One of the most fundamental concepts in economics — and therefore, in business decisions — is that money has time-value: Its value changes over time. A specific amount of money right now almost always has a value different from the same amount at some other time. This concept has been around since the earliest recorded human history and is commonly expressed as *interest*.

1.4. Equivalence

[3*, c7]

Due to the time-value of money, two or more cash flows are equivalent only when they equal the same amount of money at the same time. Therefore, comparing cash flows makes sense only when they are expressed in the same time frame. Then, lack of equivalence between the two cash flows can be determined accurately and can serve as the basis for choice. The proposal with the highest value in the same time frame is the most financially desirable.

1.5. Bases for Comparison

[3*, c8]

A *basis for comparison* is a shared frame of reference for comparing two or more cash flow streams. It uses equivalence to meaningfully compare two or more proposals. Several bases for comparison exist, including the following:

- present worth;
- future worth;
- annual equivalent;
- internal rate of return (IRR);
- discounted payback period.

1.6. Alternatives

[3*, c9]

Often, an organization could carry out more than one proposal if it wanted to. But there might be important relationships between proposals that need to be considered. Maybe Proposal Y can only be carried out if Proposal X is also carried out. Or maybe Proposal P cannot be carried out if Proposal Q is carried out, nor could Q be carried out if P were. Decisions are easier when there are mutually exclusive paths — A, or B, or C, or another project, and no others. This topic explains how to turn any set of proposals, with their interrelationships, into a set of mutually exclusive alternatives. The cash flow stream for any alternative is the sum of the cash flow streams for all the proposals it contains. The choice can then be made among these alternatives.

One special case is known as the *do-nothing alternative*. Sometimes the best course of action is not to carry out any of the proposals being considered. The do-nothing alternative represents that case. It doesn't mean do nothing at all; it means

"do something else, something that's not in this set of choices." The do-nothing alternative should be considered in most, but not all, situations.

1.7. Intangible Assets

Intangible assets, also known as *knowledge assets*, are any knowledge that lies in the non-visible side of an organization but affects that organization's financial performance. According to International Valuation Standards (IVS) 210 § 20.1, "an intangible asset is a non-monetary asset that manifests itself by its economic properties. It does not have physical substance but grants rights and economic benefits to its owner" [4].

This can include, but is not limited to, policies, procedures, tools and specifications, as well as organizational culture, experience and know-how.

Knowing the organization's intangible assets helps the software engineer better understand how proposals may affect or be affected by organizational realities. Otherwise, hidden risks and opportunities that could influence proposals' success or failure might not be exposed.

The skills needed to consider intangible assets are the following:

- intangible assets identification and valuation [Skills Framework for the Information Age (SFIA), category Strategy and Architecture, subcategory Business strategy and planning];
- knowledge management [SFIA, category Strategy and Architecture, subcategory Business strategy and planning].

Identifying and characterizing intangible assets are discussed in more detail later in this KA.

1.8. Business Model

Peter Drucker, a founder of modern management, defines a good business model as one that answers these questions [5]:

- "Who is the customer?"
- "What does the customer value?"
- "How do we make money?"
- "What is the underlying economic logic that explains how we can deliver value to customers at an appropriate cost?"

Understanding the organization's business model — as well as its intangible assets — helps the software engineer better understand how proposals may affect or be affected by organizational realities. Analyzing the business model can help the software engineer identify hidden risks and opportunities that could influence a proposal's success or failure [6].

The skills needed to consider intangible assets are knowledge management and intangible assets identification and valuation [SFIA, category Strategy and Architecture, subcategory Business strategy and planning].

2. The Engineering Decision-Making Process

2.1. Process Overview

[3*, c4pp35-36]

Figure 15.3 provides an overview of the engineering decision-making process.

Figure 15.3. The Engineering Decision-Making Process

The process is shown as stepwise and sequential; however, it can be more fluid in practice. Steps can be done iteratively, can overlap and can even occur in different sequences. Just be sure not to skip any step or execute it poorly.

When the consequences of a wrong decision are significant, such as a go/no-go decision for a large project, more time, effort and care should be spent in this process. All steps should be completed thoroughly and carefully. ISO 12207 [7] and ISO

15288 [8] recommend two additional early activities, which can be important in high-consequence decisions:

- define the decision management strategy — this strategy might specify roles, responsibilities, procedures and tools;
- identify relevant stakeholders, which might include appropriate subject matter experts.

When the consequences of a wrong decision are small, such as the consequences of selecting a minor algorithm or data structure, less time, effort and care can be spent, but the same general process is followed. Each step is discussed in more detail below.

2.2. Understand the Real Problem

[3*, c4pp37-39]

The best solution to a problem can come only from thoroughly understanding the real problem to be solved. This step's key aspects include the use of an interrogative technique such as the "5 Whys" technique and a consideration of the broader context surrounding the problem. The Empathize stage in Design Thinking [9] (to consider intangible assets) and looking closely at the organization's business model are examples of considering that broader context.

2.3. Identify All Reasonable Technically Feasible Solutions

[3*, c4pp40-41]

The goal of engineering decision-making is to find the best solution. However, the best solution must first be identified as a candidate before it can be selected as the best. If the best solution is not among the set of solutions being considered, it cannot be selected. The importance of creative thinking in this step cannot be overstated when the consequences of a wrong decision are high.

For some potential solutions, or candidates, prototyping is a useful way to verify technical feasibility. Peer review can also help verify technical feasibility and possibly spur the identification of even more candidates.

On the one hand, adding candidates increases the probability that the best one is in the set. On the other hand, each adds cost to the decision-making process. Software engineers must use their best judgment in deciding when they have enough candidates. These candidates are the "proposals," as defined in the Fundamentals topic.

2.4. Define the Selection Criteria

[3*, c4pp39-40, c26pp441-442]

Engineering decisions almost always consider the financial perspective. However, other decision criteria can also be relevant; when this is the case, the decision is a multiple-attribute decision. For example, an environmentally conscious organization may choose a less economical solution if it is more eco-friendly. In many cases, the greater the consequences of a wrong decision, the more selection criteria need to be considered.

As much as possible, each criterion should be expressed objectively. Ideally, those objective terms will be expressed as a monetary value — but not necessarily. What is the "value" of a clean river? It might not make sense to value a river by multiplying the price per kilogram of fish by an estimate of the number of fish in the river. Decision criteria that can't be expressed objectively are called "unquantifiables," "irreducibles" or "intangibles."

Defining the decision criteria can be a subjective task. Too many criteria could make the analysis unwieldy. On the other hand, too few criteria might not differentiate well between proposals and could thus lead to a suboptimal choice. The potential for a better decision provided by including more criteria must be balanced against the extra effort required to analyze the criteria.

To the extent that money is a selection criterion, the context of the decision will constrain the decision-maker to a for-profit, nonprofit or present economy decision analysis, as explained in topics 3, 4 and 5, later in this KA.

2.5. Evaluate Each Alternative Against the Selection Criteria

[3*, c4pp41-42]

Each alternative is evaluated against each selection criterion. When a selection criterion involves money, each alternative must be judged from the same viewpoint. Use the same basis for comparison (present worth, future worth, IRR, etc., in for-profit decisions; benefit-cost ratio or cost-effectiveness in nonprofit decisions, etc.), the same planning horizon, and consider the same kinds of costs and incomes. An example decision might be buying and adapting an off-the-shelf software product versus building a custom application from scratch. Considering costs for a longer time frame for one proposal than for the other will make the one using the shorter time frame seem like the better choice even though it might not be better over the same time frame.

2.6. Select the Preferred Alternative

[3*, c4p42, c26pp447-458]

If the only selection criterion is money, the alternative with the highest present worth, future worth, etc., will be chosen. When there are multiple criteria, a variety of techniques can be used to evaluate the criteria together. Multiple-attribute decision-making is detailed later in this KA.

Engineering decisions are based on estimates (discussed later in this KA). The accuracy of an estimate is limited in theory and in practice, and the degree of inaccuracy depends on the specifics of the situation [3*, c21pp344-356]. If the degree of inaccuracy is high enough, that inaccuracy could change the resulting decision. The following techniques [3*, c23] can help engineers address these situations:

- consider ranges of estimates;
- perform a sensitivity analysis;
- delay final decisions.

In addition, two categories of techniques address multiple potential outcomes from a decision:

- decision-making-under-risk techniques [3*, c24] are used when probabilities can be assigned to the different potential outcomes. Specific techniques include expected value decision-making, expectation variance and decision-making, Monte Carlo analysis, decision trees, and the expected value of perfect information;
- decision-making-under-uncertainty techniques [3*, c25] are used when probabilities cannot be assigned to the different potential outcomes. Specific techniques include the Laplace Rule, the Maximin Rule, the Maximax Rule, the Hurwicz Rule and the Minimax Regret Rule.

High-consequence decisions may benefit from formally recording the selected alternative and the justification for why that alternative was selected.

2.7. Monitor the Performance of the Selected Alternative

[3*, c4pp42-43]

Because estimation is a fundamental element of engineering decision-making, the quality of the decision depends on the quality of the estimates. Bad estimates can easily lead to bad decisions. The software engineer needs to "close the loop" on estimates by comparing them to the actual outcomes. Otherwise, no one will ever know if the estimates were good [3*, c21pp356-358]. This also helps improve estimation over time. Understanding what drives differences between estimates and actual outcomes helps engineers refine estimation techniques to produce more accurate estimates in the future.

3. For-Profit Decision-Making

For-profit decision techniques apply when the organization's goal is profit — which is the case in most companies.

Figure 15.4 shows the process for identifying the financially best alternative out of a set of proposals. Arranging alternatives in order of increasing initial investment and then selecting strictly better candidates means that, all other considerations being equal, the alternative with the smaller initial investment will be chosen. The "Is the next candidate strictly better?" decision is made in terms of the appropriate basis for comparison: present worth, future worth, IRR, etc.

Figure 15.4. The For-Profit Decision-Making Process

3.1. Minimum Acceptable Rate of Return

[3*, c10pp141-143]

The *minimum acceptable rate of return* (MARR) is the lowest IRR the organization would consider a good investment. Generally, it would not be wise to invest in an activity with a return of 10% when another activity returns 20%. The MARR is a statement that the organization is confident it can achieve at least that rate of return. The MARR represents the organization's opportunity cost for investments. By investing in some alternative, the organization explicitly decides not to invest that same money somewhere else. If the organization is already confident it can achieve a known rate of return, alternatives should be chosen only if their rate of return is at least that high. A simple way to account for that opportunity cost is to use the MARR as the interest rate in the basis for comparison.

3.2. Economic Life

[3*, c11pp160-164]

When an organization invests in a particular alternative, money is tied up in that alternative — so-called *frozen assets*. The economic impact of frozen assets typically starts high and decreases over time. On the other hand, operating and maintenance costs tend to start low and increase over time. An alternative's total cost is the sum of those two costs. At first, frozen asset costs dominate; later, operating and maintenance costs dominate. At some point, the sum of the two costs is minimized; this is the *economic life* or *minimum cost lifetime*.

3.3. Planning Horizon

[3*, c11]

To properly compare a proposal with a four-year life to a proposal with a six-year life, the economic effects of either cutting the six-year proposal by two years or investing the profits from the four-year proposal for another two years need to be addressed. The *planning horizon*, sometimes known as the *study period*, is the consistent time frame over which all proposals in the same decision are considered. Aspects such as economic life and the time frame over which reasonable estimates can be made need to be factored into establishing a planning horizon. Once the planning horizon is established, several techniques are available for putting proposals with different life spans into that planning horizon.

3.4. Replacement Decisions

[3*, c12pp171-178] [8*, c9]

A replacement decision happens when an organization already has a particular asset and is considering replacing it with a different asset (e.g., deciding between maintaining and supporting legacy software or redeveloping it from the ground up). Replacement decisions use the same for-profit decision process, but there are two additional important considerations: sunk cost and salvage value.

3.5. Retirement Decisions

[3*, c12pp178-181] [8*, c9]

Retirement decisions are about getting out of an activity altogether, such as when a software company considers not selling a software product anymore or a hardware manufacturer considers not building and selling a particular computer model any

longer. Retirement decisions can be preplanned or happen spontaneously (e.g., when performance targets are not achieved). Retirement decisions can be influenced by lock-in factors such as technology dependency and high exit costs.

3.6. Advanced For-Profit Decision Considerations

[3*, c13-17]

The above concepts and techniques are often sufficient to make a good for-profit decision. However, particularly when the consequences of a wrong decision are high, additional considerations may need to be factored into the decision analysis, including the following:

- inflation or deflation;
- depreciation;
- income taxes.

4. Nonprofit Decision-Making

The for-profit decision techniques don't apply when the organization's goal isn't profit — which is the case in government and nonprofit organizations. These organizations have a different goal, so different decision techniques are needed. The two techniques are benefit-cost analysis and cost-effectiveness analysis (discussed below).

4.1. Benefit-Cost Analysis

[3*, c18pp303-311]

Benefit-cost analysis is one of the most widely used methods for evaluating proposals in nonprofit organizations. A proposal's financial benefits are divided by its costs. Any proposal with a benefit-cost ratio of less than 1.0 can usually be rejected without further analysis because it would cost more than it would benefit the organization. Additional considerations are necessary when two or more proposals are considered at the same time.

4.2. Cost-Effectiveness Analysis

[3*, c18pp311-314]

Cost-effectiveness analysis shares much of the philosophy and methodology of benefit-cost analysis. There are two versions of cost-effectiveness analysis. The fixed-cost version seeks to maximize benefit given a fixed upper bound on cost. The fixed-effectiveness version seeks to minimize the cost to achieve a fixed goal.

5. Present Economy Decision-Making

This subset of engineering decision-making is called *present economy* because it does not involve the time-value of money (future economy). The two forms of present economy decisions are presented below.

5.1. Break-Even Analysis

[3*, c19]

Given functions describing the costs of two or more proposals, break-even analysis helps engineers choose between them by identifying points where those cost functions are equal. Below a break-even point, one proposal is preferred, and above that point, the other is preferred. For example, consider a choice between two cloud service providers. One provider has a lower fixed cost per month with a higher incremental fee for use, whereas the other has a higher fixed monthly cost with a lower incremental fee for use. Break-even analysis identifies the use level where the costs are the same. The organization's expected use level can be compared to the break-even point to identify the lower-cost provider.

5.2. Optimization Analysis

[3*, c20]

Optimization analysis studies one or more cost functions over a range of values to find the point where overall cost is lowest. Software's classic space-time trade-off is an example of optimization; an algorithm that runs faster often uses more memory. Optimization balances the value of faster run time against the cost of the additional memory.

6. Multiple-Attribute Decision-Making

[3*, c26]

Most topics presented in this KA so far have discussed decisions based on a single criterion — money. The alternative with the best present worth, the best incremental IRR, the best incremental benefit-cost ratio, etc., is the one selected. Aside from technical feasibility, money is usually the most important decision criterion, but it's certainly not always the only one. Often, other criteria, other "attributes," need to be considered that can't be cast in terms of money. Multiple-attribute decision-making techniques allow nonmonetary criteria to be factored into the decision.

A variety of techniques can be used to address multiple criteria, including nonmonetary criteria. These techniques fall into two categories.

6.1. Compensatory Techniques

[3*, c26pp449-458]

Also called *single-dimensioned techniques*, the techniques in this category collapse all criteria into a single figure of merit. This category is called *compensatory* because, for any given alternative, a lower score in one criterion can be compensated by — traded off against — a higher score in other criteria. Compensatory techniques include nondimensional scaling, additive weighting and analytic hierarchy process (AHP).

Gilb's Impact Estimation [11] and the Software Engineering Institute's (SEI) Architectural Tradeoff Analysis Method (ATAM) [12] are examples of compensatory multiple-attribute decision-making techniques focused on identifying the best software design.

6.2. Non-Compensatory Techniques

[3*, c26pp447-449]

Also called *fully dimensioned techniques*, the techniques in this category do not allow trade-offs among the criteria. Each criterion is treated as a separate entity in the selection process. Non-compensatory techniques include dominance, satisficing and lexicography.

7. Identifying and Characterizing Intangible Assets

The intangible side of an organization is the valuable knowledge residing within it. This includes employees' knowledge about processes, structures, procedures, etc. (tacit, or implicit, knowledge), as well as institutional knowledge recorded in various organizational resources (explicit knowledge). These assets are usually hidden, the way most of an iceberg is underwater. The intangible assets must be explicitly considered in many decisions, particularly when the consequences of a wrong decision are high for the client, no matter if the client is internal or external to the organization for which the software project is being done. organization.

If these assets are not adequately considered, software engineers risk developing a software solution that does not fit the client organization. Only when the intangible assets are explicitly considered will the risk of a poorly fitting software solution be minimized. The Strategic Intangible Process Assets Characterization (SIPAC) method [13] has been used to good effect to accomplish this. SIPAC steps are outlined in the following subsections.

7.1. Identify Processes and Define Business Goals

Start by understanding the organization's business processes and business goals. If the organization has well-documented processes, these should be used; otherwise, a deliberate survey will be necessary.

Business goals can include, but are not limited to, the following:

- maintaining growth and continuity of the organization;
- meeting financial objectives;
- meeting responsibility to employees;
- meeting responsibility to society;
- managing market position.

7.2. Identify Intangible Assets Linked with Business Goals

The next step is to comprehensively identify the intangible assets. Common examples are policies, documented business processes, checklists, lessons learned, templates, standards, procedures, plans and training materials. Organizations develop or acquire these assets to meet their business goals. The assets represent investments that provide business value. One effective approach to identifying an organization's intangible assets is to start with a taxonomy such as described in the following reference [14]. The goal is to identify as many intangible assets as possible that serve as a lever to achieve the business goals identified in the previous step. In practice, this can be an iterative process where reviewing the already-identified assets reveals the existence of others. A practical way to do it is by focusing iteratively on the 11 generic intangible assets (GIAs) described in [6].

Possible GIAs represent all potential parts of any business that can be involved in a digital transformation. Focusing on one or more of them allows the software engineer to better understand and frame the project's impact. Focusing iteratively on the 11 GIAs, the software engineer will select the type of GIA to be considered and, with this, elicit the specific intangible assets associated with each GIA. In addition to identifying specific intangible assets, a qualitative relative "importance" must be added to each one as it is identified. The importance is a value between 1 and 5 (1 for lower importance and 5 for higher importance), depending on how well the asset supports achieving the business objectives. The intangible assets with the highest importance are likely the most suitable target for the client organization.

7.3. Identify Software Products That Support Intangible Assets

Software products that support specific intangible assets will be part of the digital transformation proposal to be presented to the client to help them decide what digital transformation to implement.

To identify products related to specific intangible assets, the software engineer can choose from the following:

- discovering them all at a single time by using the methodology of Osterwalder [13], which promotes innovation by generating a value map with the stakeholders' emerging needs, mapping the products to the specific intangible assets;
- listing them if they are known and then mapping them to specific intangible assets;
- iteratively working with the individual intangible assets by (1) selecting a specific intangible asset and (2) identifying the products, continuing until all specific intangible assets have been analyzed.

A single product can support more than one specific intangible asset, and each specific intangible asset can be supported by many products.

7.4. Define and Measure Indicators

This step defines and measures the indicators that will be used to characterize how the intangible assets (through the software products that support those intangible assets) help meet business goals through describing, implementing or improving the identified products. Quality indicators assess specific intangible asset characteristics or features. Impact indicators assess how much the specific intangible assets contribute to processes or business goals.

Indicators must be normalized and standardized to operate correctly.

7.5. Intangible Asset Characterization

Based on the information gathered, the software engineer determines the value of the identified specific intangible assets based on their quality and impact. Specific intangible assets may be characterized in terms of their impact on business goals and their quality as organizational assets. There are three important characterization cases:

- case 1: specific intangible assets with both impact and quality indicators (Warning, Replaceable, Evolving or Stable);

- case 2: specific intangible assets with only quality indicators (Acceptable or Unacceptable);

- case 3: specific intangible assets with only impact indicators (Acceptable or Unacceptable).

The three characterization cases are shown in Figure 15.5. The quadrants represent the "states" constituting different levels of characterization. The lines separating the quadrants are thresholds of impact and quality that define the point at which the impact or quality of a specific intangible asset may be considered acceptable or not for each organization. These thresholds are established for every client organization and specify what level of organizational performance, quality, and impact they will demand from their knowledge/intangible assets.

Figure 15.5. Extended Characterization of Specific Intangible Assets

The characterization uses information from standardized-normalized indicators to assess the identified intangible assets. This assessment generates a descriptive value that will determine the asset's general state of health from a quantitative perspective.

Quality quantitative assessment

The quality valuation considers only the indicators of the type ***quality*** of an intangible asset and calculates a general valuation of it. To evaluate the subset of quality indicators, given a set of ***q*** quality indicators for an intangible asset ***n***, the valuation of the quality is given according to Equation 1.

Equation 1. Quality Assessment for a Knowledge Asset

$$Q_{VAL}^{n} = \frac{\sum_{i=1}^{q} X_i^n}{q}$$

Where $\boldsymbol{X_i^n}$ is each of the ***q*** normalized indicators of quality that the intangible asset ***n*** has.

If ***n*** is the number of indicators of quality and ***n*** is the number of indicators of impact, the assessments of quality and impact are given as the average of the normalized values of the corresponding indicators.

$$Qval = \frac{\sum_{i=1}^{n} X_{NORM}^{i}}{n}$$

Above, ***Qval*** is the average of the normalized values of the quality indicators of a corresponding specific intangible asset.

Impact quantitative assessment

An intangible asset's impact valuation is an assessment that considers only the normalized indicators that are classified as "impact" indicators. To evaluate the subset of impact indicators, given a set of ***p*** normalized impact indicators for an intangible asset ***n***, the valuation is given as stated in Equation 2:

Equation 2. Impact Assessment for a Specific Intangible Asset

$$I_{VAL}^{n} = \frac{\sum_{i=1}^{p} X_i^n}{p}$$

Where $\boldsymbol{X_i^n}$ is each of the ***p*** normalized indicators of impact that the knowledge asset ***n*** has.

$$Ival = \frac{\sum_{i=1}^{m} X_{NORM}^{i}}{n}$$

Where ***Ival*** is the average of the normalized values of the impact indicators of a corresponding knowledge asset.

Finally, the linear value of an intangible asset characterization is given by the impact and quality valuations (***Qval*** and ***Ival***), following these rules, assuming that both quality and impact are equally important, so ***KAval*** (the valuation of the intangible asset) is given by:

$$\textbf{If } \exists\, \boldsymbol{Qval} \cap \exists\, \boldsymbol{Ival}, \textbf{ then } \boldsymbol{KAval} = \frac{\boldsymbol{Qval+Ival}}{\mathbf{2}}$$

$$\textbf{If } \exists\, \boldsymbol{Qval} \cap \nexists\, \boldsymbol{Ival}, \textbf{ then } \boldsymbol{KAval} = \boldsymbol{Qval}$$

$$\textbf{If } \nexists\, \boldsymbol{Qval} \cap \exists\, \boldsymbol{Ival}, \textbf{ then } \boldsymbol{KAval} = \boldsymbol{Ival}$$

This linear value represents an intangible asset's general state based on the state of its indicators. It uses the algebraic mean of the standardized and normalized indicators to represent the assets' general state on a scale [-1, 1] and based on the corresponding interpretation thresholds.

7.6. Link Specific Intangible Assets with the Business Model

Visualizing the client business model, enriched with the intangible assets status allocated into that model, gives organizational leadership a clear understanding of the important relationships among proposed software solutions, intangible assets, the business model and the business goals. The software engineer can clearly show which proposed solution generates the most value for the business. An example is shown in [6].

7.7. Decision-Making

The next step in the decision-making process is to prioritize and choose the software products that interest the client organization most. There is no simple rule; several criteria must be considered:

- the intangible asset's impact on business goals (defined in previous steps);
- the characterization reached (defined in previous steps);
- the impact of intangibles assets status on the competitors of the organization under improvement;
- the intangible asset's impact on the business model;
- cost to implement the products;
- time to implement the products;
- complexity of the products.

All criteria must be considered to decide what software products should be developed for the client organization, making this a multiple-attribute decision. (See 6., Multiple-Attribute Decision-Making.)

Upon considering all relevant criteria, the organization can see the risks of implementing a software solution to automate processes that are either not very valuable or not in good shape. Instead, the software engineer can offer, in a transparent way, proposals that better satisfy the organization's business needs.

This approach can be useful whenever an engineering decision needs to be made, but it is particularly critical in the pre-project stage when there is a need to present the client organization with proposals that are best for business value.

8. Estimation

[3*, c21-26]

An estimate analytically predicts some quantity, like a project's size, cost or schedule. Many other quantities are also estimated in software engineering, such as the average number of active client sessions a cloud service needs to support, the number of times a function will be called during execution of a section of code, or the number of delivered defects in a software product.

Software engineers do not estimate purely for the sake of estimating. Software engineers estimate to make decisions when critical quantities are unknown. For example, a decision to buy a functionality or build it within the organization would certainly be based on the cost of building it. But the actual cost of building it cannot be precisely known until the

organization does build it. Key information needed to make engineering decisions is usually not known when those decisions need to be made. Instead, decisions are based on estimates. Behind every estimate is one or more decisions.

Given that estimates are predictions, there is a nonzero probability that the actual outcome will differ from the estimate. All estimates are inherently uncertain. Sometimes, the uncertainty is large, and sometimes it is small. But it is always there. Fortunately, estimates need not be perfect; they need only to be good enough to lead the decision-maker to make the right decision.

The Software Engineering Code of Ethics and Professional Practice [16] states, "3.09. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work and provide an uncertainty assessment of these estimates" (underlining added for emphasis). (See [3*, c21pp358-361].)

Estimation is covered extensively in [17], [18] and [3*]. Several general techniques exist, and each is overviewed here. All specific estimation techniques use one or a combination of these general techniques.

8.1. Expert Judgment

[3*, c22pp367-369]

In expert judgment estimation, the estimate is based purely on the estimator's professional opinion. It is the simplest technique and is always available, and it is particularly useful when the other techniques aren't available. The downside is that this technique produces the least accurate estimates.

8.2. Analogy

[3*, c22pp369-371]

Estimation by analogy assumes that if the thing estimated is similar to something already known, then the estimate for the new thing can be based on the actual result for the similar thing, with allowances for relevant differences. The steps in estimation by analogy are as follows:

1. Understand the thing to be estimated.
2. Find a suitable analogy for which actual results are known.
3. List differences between the thing being estimated and the analogy that could significantly affect the outcome.
4. Estimate the magnitude of each identified difference.
5. Build the estimate from the analogy's actual result and adjustments for the identified differences.

Estimation by analogy produces more accurate results than expert judgment, and it is still relatively quick and easy. On the other hand, an appropriate analogy for which accurate results are known must be available for this approach to work.

8.3. Decomposition

[3*, c22pp371-374]

Sometimes called *bottom-up estimation*, the steps in estimation by decomposition are:

1. Break the thing to be estimated into successively smaller pieces until the smallest pieces can be reasonably estimated.
2. Estimate those smallest pieces.
3. Add up the estimates for the smallest pieces to build the estimate for the whole.
4. If the estimates for the smallest pieces don't include allowances for significant cross-cutting factors, then find a way to address those factors. For example, when estimating a software project from its design, the estimates for the design elements may not include allowances for requirements work, integration work, testing work and user documentation work.

Estimation by decomposition assumes that overestimates of lowest-level pieces will cancel out corresponding underestimates of other pieces and lead to a more accurate estimate of the whole. The primary disadvantages are the following:

- it can be a lot more work than any other technique;
- if the bottom-level estimates are biased either high or low, the canceling effect doesn't happen.

8.4. Parametric

[3*, c22pp374-377]

Also called *estimation by statistical methods*, parametric estimation takes advantage of a known, mathematical relationship between the thing being estimated and one or more observable factors about that thing, like calculating the cost to build a building as a function of its floor space. The estimation model is an equation: First, count the observable factors, and then plug them into the equation to get the resulting estimate.

Parametric estimates are typically the most accurate, the most defendable and the easiest to use, provided the equation has been developed and validated. The disadvantage is that developing and validating such an equation requires an adequate base of accurate historical data along with some nontrivial mathematics and statistics.

8.5. Multiple Estimates

[3*, c22pp377-379]

When the consequences of a wrong decision are small, it can be acceptable to base the decision on a single estimate from a single estimator using a single estimation technique. However, when the consequences of a wrong decision are significant, investing extra effort in developing more than one estimate can be worthwhile.

To use this approach, engineers estimate the same thing using different techniques, possibly by different estimators. Then, they look for convergence or divergence among those multiple estimates. Convergence suggests the individual estimates are probably accurate, and any of them could be used to make the decision. Divergence suggests that one or more important factors might have been overlooked. Finding the factors that caused the divergence and reestimating to produce converging results often lead to a better estimate and thus a better decision.

9. Practical Considerations

9.1. Business Case

The business case is the consolidated, documented information summarizing and explaining a recommended business decision from different perspectives (cost, benefit, risk and so on) for a decision-maker and other relevant stakeholders. It's used to assess a product's potential value, which can be used as a basis for an investment decision.

9.2. Multiple-Currency Analysis

When a decision analysis involves cross-border finances, currency exchange rate variations may need to be considered. This is often done using historical data.

9.3. Systems Thinking

The ecosystem in which software engineers develop their professional life is complex. To understand the whole picture around a client organization and form a holistic view of the scenarios they analyze, software engineers can use systems thinking methodologies. This approach helps the software engineer create a complete set of possible scenarios in which the software to be provided could be useful and, with this information, explain to the client how the software solution can be a value provider for the organization. Sources for system thinking methodologies are [19] and [20]. A way to connect systems thinking methodologies with the development of a business model to understand the pillars of the client organization can be reached here [21].

10. Related Concepts

This topic includes concepts the software engineer may want to bear in mind.

10.1. Accounting

[3*, c15pp234-245]

Accounting is part of finance. It allows people whose money is used to run an organization to know the results of their investment: Did they get the profit they were expecting? In for-profit organizations, this relates to the tangible return on investment (ROI), while in nonprofit and governmental organizations,

as well as for-profit organizations, it translates into sustainably staying in business. Accounting's primary role is to measure the organization's actual financial performance and to communicate financial information about a business entity to stakeholders, such as shareholders, financial auditors and investors. Communication generally takes the form of financial statements showing the economic resources to be controlled. The right information — relevant and reliable to the user — must be presented. Information and its timing are partially governed by risk management and governance policies. Accounting systems are also a rich source of historical data for estimating.

Software engineers must be conscious of the software's importance as a driver of business accounts in the digital era.

10.2. Cost and Costing

[3*, c15pp245-259]

A cost is the money used to produce something and, hence, is no longer available for use. In economics, a cost is an alternative that is given up as a result of a decision.

Sunk cost refers to unrecoverable expenses that have occurred, which can cause emotional hurdles looking forward. From a traditional economics viewpoint, sunk costs should not be considered in decision-making. Opportunity cost is the cost of an alternative that must be forgone to pursue another alternative.

Costing is part of finance and product management. It is the process of determining the cost based on expenses (e.g., production, software engineering, distribution, rework) and on the target cost to be competitive and successful in a market. The target cost can be below the actual estimated cost. The planning and controlling of these costs (called *cost management*) is important and should always be included in costing.

An important concept in costing is the total cost of ownership (TCO). This holds true especially for software because there are many not-so-obvious costs related to SPLC activities after initial product development. TCO for a software product is defined as the total cost for acquiring that product, activating it and keeping it running. These costs can be grouped as direct and indirect costs. TCO is an accounting method that is crucial in making sound economic decisions.

10.3. Finance

Finance is the branch of economics concerned with allocating, managing, acquiring and investing resources. Finance is an element of every organization, including software engineering organizations.

The field of finance deals with the concepts of time, money, and risk, and how they are interrelated. It also deals with how money is spent and budgeted. Corporate finance is concerned with funding an organization's activities. Generally, this involves balancing risk and profitability while attempting to maximize an organization's wealth and the value of its stock. This applies primarily to for-profit organizations but also to nonprofit organizations. The latter needs finances to ensure sustainability, if not to make a tangible profit. To do this, an organization must:

- identify organizational goals, time horizons, risk factors, tax considerations and financial constraints;
- identify and implement the appropriate business strategy, such as which portfolio and investment decisions to take, how to manage cash flow and where to get the funding;
- measure financial performance, such as cash flow and ROI, and take corrective actions in case of deviation from objectives and strategy.

Provided that many organizations use software development or acquisition to stay competitive, the software engineer must be conscious of the importance of software to business finances.

10.4. Controlling

Controlling is the element of finance and accounting that involves measuring and correcting performance. It ensures that an organization's objectives and plans are accomplished. Controlling cost is a specialized

branch of controlling used to detect variances of actual costs from planned costs.

In software engineering, this concept is referred to as processes and products control and evolution. While the organization is seen as an entity with its own goals, and control of the organizational goals is seen as separate, software engineers must consider control of the organization part of their job by ensuring alignment of their software with business goals.

10.5. Efficiency and Effectiveness

[10*, c22pp422-23]

Economic efficiency of a process, activity or task is the ratio of resources consumed to resources expected to be consumed. Efficiency means "doing things right." An efficient behavior, like an effective behavior, delivers results and minimizes effort. Factors affecting efficiency in software engineering include product complexity, quality requirements, time pressure, process capability, team distribution, interruptions, feature churn, tools and programming language.

Effectiveness is about having impact. It is the relationship between achieved objectives and defined objectives. *Effectiveness* means "doing the right things." Effectiveness looks only at whether defined objectives are reached — not at how they are reached.

10.6. Productivity

[10*, c23pp689]

Productivity is the ratio of output to input from an economic perspective. *Output* is the value delivered. *Input* covers all resources (e.g., effort) spent to generate the output. Productivity combines efficiency and effectiveness from a value-oriented perspective. Maximizing productivity is about generating the highest value with the lowest resource consumption.

10.7. Product or Service

A *product* is a tangible economic good (or output) created in a process that transforms product factors (or inputs) into an output. A *service* is an intangible resource, like consulting. When sold, a product or service is a deliverable that creates both a value and an experience for its consumers. A product or service can be a combination of systems, solutions and materials delivered internally (e.g., an in-house IT solution) or externally (e.g., a software application), either as is or as a component for another product (e.g., embedded software).

10.8. Project

[22*, c2s2.4]

A *project* is "a temporary endeavor undertaken to create a unique product, service, or result" [23]. In software engineering, different project types are distinguished (e.g., product development, outsourced services, software maintenance, service creation, and so on). During its life cycle, a software product may require many projects. For example, during the product conception phase, a project might be conducted to determine customer need and market requirements; during maintenance, a project might be conducted to produce the next version of a product.

10.9. Program

A *program* is "a group of related projects, subprograms, and program activities managed in a coordinated way to obtain benefits not available from managing them individually" [23]. Programs are often used to identify and manage different deliveries to a single customer or market over a time horizon of several years.

10.10. Portfolio

Portfolios are "projects, programs, sub-portfolios, and operations managed as a group to achieve strategic objectives" [23]. Portfolios are used to group and then manage simultaneously all assets within a business line or organization. Having an entire portfolio to consider helps ensure that the broader impacts of decisions are considered, such as the decision to allocate resources to a specific project, which means that the same resources will not be available for the other projects in the portfolio.

10.11. Product Life Cycle

An SPLC includes all activities needed to define, build, operate, maintain and retire a software product or service and its variants. The SPLC activities of "operate," "maintain" and "retire" occur in a much longer time frame than initial software development (the software development life cycle (SDLC)). (See Software Life Cycle Models in the Software Engineering Process KA.) Also, the operate-maintain-retire activities of an SPLC consume more total effort and other resources than the SDLC activities. (See Majority of Maintenance Costs in the Software Maintenance KA.) The value contributed by a software product or associated services can be objectively determined during the "operate and maintain" time frame. Software engineering economics should be concerned with all SPLC activities, including activities that take place after the initial product release.

10.12. Project Life Cycle

Project life cycle activities typically involve five process groups: Initiating, Planning, Executing, Monitoring and controlling, and Closing [24]. (See the Software Engineering Management KA.) The activities within a software project life cycle are often interleaved, overlapped and iterated in various ways [20*, c2] [25]. (See the Software Engineering Process KA.) For instance, Agile product development within an SPLC involves multiple iterations that produce increments of deliverable software. An SPLC should include risk management and synchronization with different suppliers (if any) while providing auditable decision-making information (e.g., to comply with product liability needs or governance regulations). The software project life cycle and the SPLC are interrelated; an SPLC may include several SDLCs.

10.13. Price and Pricing

[10*, c23s23.1]

A *price* is what is paid in exchange for a good or service. Price is a fundamental aspect of financial modeling and is one of the four Ps of the marketing mix. The other three Ps are product, promotion and place. Price is the only revenue-generating element among the four Ps; the rest are costs.

Pricing is an element of finance and marketing. It determines what a company will receive in exchange for its products. Pricing factors include manufacturing cost, market placement, competition, market condition and product quality. Pricing applies prices to products and services based on factors such as fixed amount, quantity break, promotion or sales campaign, specific vendor quote, shipment or invoice date, combination of multiple orders, service offerings, and many others. The consumer's needs can be converted into demand only if the consumer has the willingness and capacity to buy the product. Thus, pricing is crucial in marketing. Pricing is initially done during the project initiation phase and is a part of the "go" decision-making process.

10.14. Prioritization

Prioritization involves ranking alternatives based on common criteria to deliver the best value. For example, in software engineering projects, software requirements are often prioritized to deliver the most value to the client within the constraints of schedule, budget, resources, and technology, or to allow the team to build the product in increments, where the first increments provide the highest value to the customer. (See Requirements Prioritization in the Software Requirements KA and Software Life Cycle Models in the Software Engineering Process KA.) Prioritizing alternatives is at least implicit in the discussion in 2.6., Select the Preferred Alternative, but is explicit when a compensatory technique is used, as described in 7.6., Multiple-Attribute Decision-Making.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Tockey 2005 [3*]	Sommerville 2016 [10*]	Fairley 2009 [22*]
1. Software Engineering Economics Fundamentals			
1.1. Proposals	c3pp23-24		
1.2. Cash Flow	c3pp24-32		
1.3. Time-Value of Money	c5-6		
1.4. Equivalence	c7		
1.5. Bases for Comparison	c8		
1.6. Alternatives	c9		
1.7. Intangible Assets			
1.8. Business Model			
2. The Engineering Decision-Making Process			
2.1. Process Overview	c4pp35-36		
2.2. Understand the Real Problem	c4pp37-39		
2.3. Identify All Reasonable Technically Feasible Solutions	c4pp40-41		
2.4. Define the Selection Criteria	c4pp39-40, c26pp441-442		
2.5. Evaluate Each Alternative Against the Selection Criteria	c4pp41-42		
2.6. Select the Preferred Alternative	c4p42, c26pp447-458		

	Tockey 2005 [3*]	Sommerville 2016 [10*]	Fairley 2009 [22*]
2.7. Monitor the Performance of the Selected Alternative	c4pp42-43		
3. For-Profit Decision-Making			
3.1. Minimum Acceptable Rate of Return	c10pp141-143		
3.2. Economic Life	c11pp160-164		
3.3. Planning Horizon	c11		
3.4. Replacement Decisions	c12pp171-178	c9	
3.5. Retirement Decisions	c12pp178-181	c9	
3.6. Advanced For-Profit Decision Considerations	c13-17		
4. Nonprofit Decision-Making			
4.1. Benefit-Cost Analysis	c18pp303-311		
4.2. Cost-Effectiveness Analysis	c18pp311-314		
5. Present Economy Decision-Making			
5.1. Break-Even Analysis	c19		
5.2. Optimization Analysis	c20		
6. Multiple-Attribute Decision-Making			
6.1. Compensatory Techniques	c26pp449-458		
6.2. Non-Compensatory Techniques	c26pp447-449		

	Tockey 2005 [3*]	Sommerville 2016 [10*]	Fairley 2009 [22*]
7. Identifying and Characterizing Intangible Assets			
7.1. Identify Processes and Define Business Goals			
7.2. Identify Intangible Assets Linked with Business Goals			
7.3. Identify Software Products That Support Intangible Assets			
7.4. Define and Measure Indicators			
7.5. Intangible Asset Characterization			
7.6. Link Specific Intangible Assets with the Business Model			
7.7. Decision-Making			
8. Estimation			
8.1. Expert Judgment	c22pp367-369		
8.2. Analogy	c22pp369-371		
8.3. Decomposition	c22pp371-374		
8.4. Parametric	c22pp374-377		
8.5. Multiple Estimates	c22pp377-379		
9. Practical Considerations			
9.1. Business Case			
9.2. Multiple-Currency Analysis			

	Tockey 2005 [3*]	Sommerville 2016 [10*]	Fairley 2009 [22*]
9.3. Systems Thinking			
10. Related Concepts			
10.1. Accounting	c15pp234-245		
10.2. Cost and Costing	c15pp245-259		
10.3. Finance			
10.4. Controlling			
10.5. Efficiency and Effectiveness		c22pp422-23	
10.6. Productivity		c23pp689	
10.7. Product or Service			
10.8. Project			c2s2.4
10.9. Program			
10.10. Portfolio			
10.11. Product Life Cycle			
10.12. Project Life Cycle			
10.13. Price and Pricing		c23s23.1	
10.14. Prioritization			

FURTHER READINGS

Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)* [24].

The *PMBOK® Guide* provides guidelines for managing individual projects and defines project management-related concepts. It also describes the project management life cycle and its related processes, as well as the project life cycle. It is a globally recognized guide for the project management profession.

Project Management Institute and IEEE Computer Society, *Software Extension to the Guide to the Project Management Body of Knowledge (S WX)* [25].

SWX provides adaptations and extensions to the generic practices of project management documented in the *PMBOK® Guide* for managing software projects. The primary contribution of this extension to the *PMBOK® Guide* is its description of processes that are applicable to managing adaptive life cycle software projects.

B.W. Boehm, *Software Engineering Economics* [26].

This book is classic reading on software engineering economics. It provides an overview of business thinking in software engineering. Although the examples and figures are dated, it is still worth reading.

C. Ebert and R. Dumke, *Software Measurement* [27].

This book provides an overview of quantitative methods in software engineering, starting with measurement theory and proceeding to performance management and business decision-making.

D.J. Reifer, *Making the Software Business Case: Improvement by the Numbers* [28].

This book is classic reading on making a business case in software and IT industries. Many useful examples illustrate how the business case is formulated and quantified.

REFERENCES

[1] E. DeGarmo et al., *Engineering Economy*, 9th ed., Englewood Cliffs, NJ: Prentice Hall, 1993.

[2] P. Rodriguez, C. Urquhart, and E. Mendes. "A Theory of Value for Value-based Feature Selection in Software Engineering," *IEEE Transactions on Software Engineering*, 1, 2020.

[3*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, Boston, MA: Addison-Wesley, 2005.

[4] *International Valuation Standards (IVS)*, 2019, Norwich: Page Bros, ISBN: 978-0-9931513-3-3-0.

[5] K. Voigt, O. Buliga, and K. Michl, Business model pioneers: How innovators successfully implement new business models, 2017.

[6] M.-I. Sanchez-Segura, G.-L. Dugarte-Peña, A. Amescua-Seco, and F. Medina-Domínguez, "Exploring how the intangible side of an organization impacts its business model," Kybernetes, Vol. 50 No. 10, pp. 2790-2822. 2021. https://doi.org/10.1108/K-05-2020-0302

[7] ISO/IEC/IEEE International Standard – Systems and software engineering – Software life cycle processes – Part 2: Relation and mapping between

ISO/IEC/IEEE 12207:2017 and ISO/IEC 12207:2008. IEEE, 2020, pp. 1-278, doi: 10.1109/IEEESTD.2020.9238529.

[8*] ISO/IEC/IEEE 15288 First edition 2015-05-15: ISO/IEC/IEEE International Standard – Systems and software engineering – System life cycle processes, IEEE.

[9] T. Brown and B. Katz, *Change by Design: How Design Thinking Transforms Organizations and Inspires Innovation*, Revised and updated ed., New York, NY: Harper Collins, 2019.

[10*] I. Sommerville, *Software Engineering*, 10th ed., New York: Addison-Wesley, 2016.

[11] T. Gilb, *Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*, Oxford, UK: Elsevier Butterworth-Heinemann, 2005.

[12] R. Kazman, M. Klein, and P. Clements, "ATAM^{SM}: Method for Architecture Evaluation," CMU/SEI-2000-TR-004, Software Engineering Institute, August 2000.

[13] M.I. Sanchez-Segura, A. Ruiz-Robles, F. Medina-Domínguez, and G.L. Dugarte-Peña. "Strategic characterization of process assets based on asset quality and business impact," *Industrial Management and Data Systems*, 117(8), 1720-1734. https://doi.org/10.1108/IMDS-10-2016-0422, 2017.

[14] M.I. Sanchez-Segura, A. Ruiz Robles, F. Medina-Domínguez. "Uncovering hidden process assets: A case study." *Information Systems Frontiers*. https://www.springerprofessional.de/en/uncovering-hidden-process-assets-a-case-study/11724394, 2016.

[15] A. Osterwalder, Y. Pigneur, G. Bernarda, A. Smith, and T. Papadakos, *Value proposition design*, Wiley, 2015.

[16] D. Gotterbarn, K. Miller, and S. Rogerson, "Software engineering code of ethics," *Commun. ACM* 40, 11, 110-118, doi: 10.1145/265684.265699, 1997.

[17] S. McConnell, *Software Estimation Demystifying the Black Art*, 1st ed., Microsoft Press, 2009.

[18] R.D. Stutzke, *Estimating software-intensive systems projects, products, and processes*, 1st ed., Addison-Wesley, 2005.

[19] M. Ben-Eli, *Understanding Systems. Systems Innovation*, http://bit.ly/2XNlh3D, 2019.

[20] J. Sterman, *Business Dynamics: Systems Thinking and Modeling for a Complex World*, McGraw-Hill, 2000.

[21] S. Pereira, G. Medina, et al., "System thinking and business model canvas for collaborative business models design," *IFIP Advances in Information and Communication Technology*, Vol. 488, pp. 461-468, 2016.

[22*] R.E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

[23] Project Management Institute, Inc., *PMI Lexicon of Project Management Term,* 2012.

[24] Project Management Institute, *A Guide to the Project Management Body o f Knowledge (PMBOK® Guide)*, 5th ed., Project Management Institute, 2013.

[25] Project Management Institute and IEEE Computer Society, *Software Extension to the PMBOK® Guide Fifth Edition*, ed: Project Management Institute, 2013.

[26] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[27] C. Ebert and R. Dumke, *Software Measurement*, Springer, 2007.

[28] D.J. Reifer, *Making the Software Business Case: Impro vement by the Numbers*, Addison-Wesley, 2002.

APPENDIX A

KNOWLEDGE AREA DESCRIPTION SPECIFICATIONS

INTRODUCTION

This document presents the specifications provided to the Knowledge Area Editors (KA Editors) regarding the Knowledge Area Descriptions (KA Descriptions) of the Version 3 (V3) edition of the *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide)*. This document will also enable readers, reviewers, and users to clearly understand what specifications were used when developing this version of the *SWEBOK Guide*.

This document begins by situating the *SWEBOK Guide* as a foundational document for the IEEE Computer Society suite of software engineering products and more widely within the software engineering community at large. The role of the baseline and the Change Control Board is then described. Criteria and requirements are defined for the breakdowns of topics, for the rationale underlying these breakdowns and the succinct description of topics, and for reference materials. Important input documents are also identified, and their role within the project is explained. Noncontent issues such as submission format and style guidelines are also discussed.

THE SWEBOK GUIDE IS A FOUNDATIONAL DOCUMENT FOR THE IEEE COMPUTER SOCIETY SUITE OF SOFTWARE ENGINEERING PRODUCTS

The *SWEBOK Guide* is an IEEE Computer Society flagship and structural document for the IEEE Computer Society suite of software engineering products. The *SWEBOK Guide* is also more widely recognized as a foundational document within the software engineering community at large notably through the official recognition of the 2004 Version as ISO/IEC Technical Report 19759:2005. The list of knowledge areas (KAs) and the breakdown of topics within each KA is described and detailed in the introduction of this *SWEBOK Guide*.

Consequently, the *SWEBOK Guide* is foundational to other initiatives within the IEEE Computer Society:

a) The list of KAs and the breakdown of topics within each KA are also adopted by the software engineering certification and associated professional development products offered by the IEEE Computer Society (see www.computer.org/certification).
b) The list of KAs and the breakdown of topics are also foundational to the software engineering curricula guidelines developed or endorsed by the IEEE Computer Society (www.computer.org/portal/web/education/Curricula).
c) The Consolidated Reference List (see Appendix C), meaning the list of recommended reference materials (to the level of section number) that accompanies the breakdown of topics within each KA is also adopted by the software engineering certification and associated professional development products offered by the IEEE Computer Society.

BASELINE AND CHANGE CONTROL BOARD

Due to the structural nature of the *SWEBOK Guide* and its adoption by other products, a baseline was developed at the outset of the project comprised of the list of KAs, the breakdown of

topics within each KA, and the Consolidated Reference List.

A Change Control Board (CCB) has been in place for the development of this version to handle all change requests to this baseline coming from the KA Editors, arising during the review process, or otherwise. Change requests must be approved both by the *SWEBOK Guide* Editors and by the CCB before being implemented. This CCB is comprised of members of the initiatives listed above and acting under the authority of the Software and Systems Engineering Committee of the IEEE Computer Society Professional Activities Board.

CRITERIA AND REQUIREMENTS FOR THE BREAKDOWN OF TOPICS WITHIN A KNOWLEDGE AREA

a) KA Editors are instructed to adopt the baseline breakdown of topics.
b) The breakdown of topics is expected to be "reasonable," not "perfect."
c) The breakdown of topics within a KA must decompose the subset of the Software Engineering Body of Knowledge that is "generally recognized." See below for a more detailed discussion of this point.
d) The breakdown of topics within a KA must not presume specific application domains, business needs, sizes of organizations, organizational structures, management philosophies, software life cycle models, software technologies, or software development methods.
e) The breakdown of topics must, as much as possible, be compatible with the various schools of thought within software engineering.
f) The breakdown of topics within a KA must be compatible with the breakdown of software engineering generally found in industry and in the software engineering literature and standards.
g) The breakdown of topics is expected to be as inclusive as possible.
h) The *SWEBOK Guide* adopts the position that even though the following "themes" are common across all Knowledge Areas, they are also an integral part of all Knowledge Areas and therefore must be incorporated into the proposed breakdown of topics of each Knowledge Area. These common themes are measurement, quality (in general), and security.
i) The breakdown of topics should be at most two or three levels deep. Even though no upper or lower limit is imposed on the number of topics within each KA, a reasonable and manageable number of topics is expected to be included in each KA. Emphasis should also be put on the selection of the topics themselves rather than on their organization in an appropriate hierarchy.
j) Topic names must be significant enough to be meaningful even when cited outside the *SWEBOK Guide*.
k) The description of a KA will include a chart (in tree form) describing the knowledge breakdown.

CRITERIA AND REQUIREMENTS FOR DESCRIBING TOPICS

Topics need only be sufficiently described so the reader can select the appropriate reference material according to his/her needs. Topic descriptions must not be prescriptive.

CRITERIA AND REQUIREMENTS FOR REFERENCE MATERIAL

a) KA Editors are instructed to use the references (to the level of section number) allocated to their KA by the Consolidated Reference List as their Recommended References.
b) There are three categories of reference material:

» Recommended References. The set of Recommended References (to the level of section number) is collectively known as the Consolidated Reference List.
» Further Readings.
» Additional references cited in the KA Description (for example, the source of a quotation or reference material in support of a rationale behind a particular argument).

c) The *SWEBOK Guide* is intended by definition to be selective in its choice of topics and associated reference material. The list of reference material should be clearly viewed as an "informed and reasonable selection" rather than as a definitive list.

d) Reference material can be book chapters, refereed journal papers, refereed conference papers, refereed technical or industrial reports, or any other type of recognized artifact. References to another KA, subarea, or topic are also permitted.

e) Reference material must be generally available and must not be confidential in nature.

f) Reference material must be in English.

g) Criteria and requirements for recommended reference material or Consolidated Reference List:

» Collectively the list of Recommended References should be

i. complete: covering the entire scope of the *SWEBOK Guide*
ii. sufficient: providing enough information to describe "generally accepted" knowledge
iii. consistent: not providing contradictory knowledge nor conflicting practices
iv. credible: recognized as providing expert treatment
v. current: treating the subject in a manner that is commensurate with currently generally accepted knowledge
vi. succinct: as short as possible (both in number of reference items and in total page count) without failing other objectives.

» Recommended reference material must be identified for each topic. Each recommended reference item may of course cover multiple topics. Exceptionally, a topic may be self-descriptive and not cite a reference material item (for example, a topic that is a definition or a topic for which the description itself without any cited reference material is sufficient for the objectives of the *SWEBOK Guide*).

» Each reference to the recommended reference material should be as precise as possible by identifying what specific chapter or section is relevant.

» A matrix of reference material (to the level of section number) versus topics must be provided.

» A reasonable amount of recommended reference material must be identified for each KA. The following guidelines should be used in determining how much is reasonable:

i. If the recommended reference material were written in a coherent manner that followed the proposed breakdown of topics and in a uniform style (for example, in a new book based on the proposed KA description), an average target across all KAs for the number of pages would be 750. However, this target may not be attainable when selecting existing reference material due to differences in style and overlap and redundancy between the selected reference materials.
ii. In other words, the target for the number of pages for the entire collection of recommended references of the *SWEBOK Guide* is in the range of 10,000 to 15,000 pages.
iii. Another way of viewing this is that the amount of recommended reference material would be reasonable if it consisted of the study material on this KA for a software engineering licensing exam that a graduate would pass after completing four years of work experience.

h) Additional reference material can be included by the KA Editor in a "Further Readings" list:

» These further readings must be related to the topics in the breakdown rather than, for example, to more advanced topics.
» The list must be annotated (within 1 paragraph per reference) as to why this reference material was included in the list of further readings. Further readings could include: new versions of an existing reference already included in the recommended references, alternative viewpoints on a KA, or a seminal treatment of a KA.
» A general guideline to be followed is 10 or fewer further readings per KA.
» There is no matrix of the reference materials listed in further readings and the breakdown of topics.

i) Criteria and requirements regarding additional references cited in the KA Description:

» The *SWEBOK Guide* is not a research document and its readership will be varied. Therefore, a delicate balance must be maintained between ensuring a high level of readability within the document while maintaining its technical excellence. Additional reference material should therefore only be brought in by the KA Editor if it is necessary to the discussion. Examples are to identify the source of a quotation or to cite reference item in support of a rationale behind a particular and important argument.

COMMON STRUCTURE

KA descriptions should use the following structure:

- Acronyms
- Introduction
- Breakdown of Topics of the KA (including a figure describing the breakdown)
- Matrix of Topics vs. Reference Material
- List of Further Readings
- References

WHAT DO WE MEAN BY "GENERALLY RECOGNIZED KNOWLEDGE"?

The Software Engineering Body of Knowledge is an all-inclusive term that describes the sum of knowledge within the profession of software engineering. However, the *SWEBOK Guide* seeks to identify and describe that subset of the body of knowledge that is generally recognized or, in other words, the core body of knowledge. To better illustrate what "generally recognized" knowledge is relative to other types of knowledge, Figure A.1 proposes a three-category schema for classifying knowledge.

The Project Management Institute in its *Guide to the Project Management Body of Knowledge* defines "generally recognized" knowledge for project management as being:

> that subset of the project management body of knowledge generally recognized as good practice. "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. "Good practice" means there is general agreement that the application of these skills, tools, and techniques can enhance the chances of success over a wide range of projects. "Good practice" does not mean that the knowledge described should always be applied uniformly to all projects; the organization and/or project management team is responsible for determining what is appropriate for any given project. [1]

"Generally accepted" knowledge could also be viewed as knowledge to be included in the study material of a software engineering licensing exam (in the USA) that a graduate would take after completing four years of work experience. These two definitions should be seen as complementary.

KA Editors are also expected to be somewhat forward looking in their interpretation by taking into consideration not only what is "generally recognized" today and but what they expect will be "generally recognized" in a 3- to 5-year timeframe.

Specialized Practices Used Only for Certain Types of Software	**Generally Recognized** Established traditional practices recommended by many organizations
	Advanced and Research Innovative practices tested and used only by some organizations and concepts still being developed and tested in research organizations

Figure A.1. Categories of Knowledge

LENGTH OF KA DESCRIPTION

KA Descriptions are to be roughly 10 to 20 pages using the formatting template for papers published in conference proceedings of the IEEE Computer Society. This includes text, references, appendices, tables, etc. This, of course, does not include the reference materials themselves.

IMPORTANT RELATED DOCUMENTS

1. *Graduate Software Engineering 2009 (GSwE2009): Curriculum Guidelines for Graduate Degree Programs in Software Engineering*, 2009; www.gswe2009.org. [2]

This document "provides guidelines and recommendations" for defining the curricula of a professional master's level program in software engineering. The *SWEBOK Guide* is identified as a "primary reference" in developing the body of knowledge underlying these guidelines. This document has been officially endorsed by the IEEE Computer Society and sponsored by the Association for Computing Machinery.

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].

This standard is considered the key standard regarding the definition of life cycle processes and has been adopted by the two main standardization bodies in software engineering: ISO/IEC JTC1/SC7 and the IEEE Computer Society Software and Systems Engineering Standards Committees. It also has been designated as a pivotal standard by the Software and System Engineering Standards Committee (S2ESC) of the IEEE.

Even though we do not intend that the *Guide to the Software Engineering Body of Knowledge* be fully 12207-conformant, this standard remains a key input to the *SWEBOK Guide*, and special care will be taken throughout the *SWEBOK Guide* regarding the compatibility of the *Guide* with the 12207 standard.

3. J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006. [4*]

This book describes the scope, roles, uses, and development trends of the most widely used software engineering standards. It concentrates on important software engineering activities—quality and project management, system engineering, dependability, and safety. The analysis and regrouping of the standard collections exposes the reader to key relationships between standards.

Even though the *SWEBOK Guide* is not a software engineering standard per se, special care will be taken throughout the document regarding the compatibility of the *Guide* with the current IEEE and ISO/IEC Systems and Software Engineering Standards Collection.

4. *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, IEEE Computer Society and Association for Computing Machinery, 2004; http://sites.computer.org/ccse/SE2004Volume.pdf. [5]

This document describes curriculum guidelines for an undergraduate degree in software engineering. The *SWEBOK Guide* is identified as being "one of the primary sources" in developing the body of knowledge underlying these guidelines.

5. *ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary*, ISO/IEC/IEEE, 2010; www.computer.org/sevocab. [6]

The hierarchy of references for terminology is *Merriam Webster's Collegiate Dictionary* (11th ed.) [7], IEEE/ISO/IEC 24765 [6], and new proposed definitions if required.

6. "Certification and Training for Software Professionals," IEEE Computer Society, 2013; www.computer.org/certification. [8]

Information on the certification and associated professional development products developed and offered by the IEEE Computer Society for professionals in the field of software engineering can be found on this website. The *SWEBOK Guide* is foundational to these products.

STYLE AND TECHNICAL GUIDELINES

- KA Descriptions should conform to the Word template available at www.computer.org/portal/web/cscps/formatting.
- KA Descriptions are expected to follow the IEEE Computer Society Style Guide (www.computer.org/portal/web/publications/styleguide).
- Files are to be submitted in Microsoft Word format.
- All citations of reference material are to be produced using EndNote Web as indicated in the instructions provided to KA Editors in this regard.

OTHER DETAILED GUIDELINES

When referencing the *Guide to the Software Engineering Body of Knowledge*, use the title "*SWEBOK Guide.*"

For the purpose of simplicity, avoid footnotes and try to include their content in the main text.

Use explicit references to standards, as opposed to simply inserting numbers referencing items in the bibliography. We believe this approach allows the reader to be better exposed to the source and scope of a standard.

The text accompanying figures and tables should be self-explanatory or have enough related text. This would ensure that the reader knows what the figures and tables mean.

To make sure that some information in the *SWEBOK Guide* does not become rapidly obsolete and due to its generic nature, please avoid directly naming tools and products. Instead, try to name their functions.

EDITING

Editors of the *SWEBOK Guide* as well as professional copy editors will edit KA Descriptions. Editing includes copy editing (grammar, punctuation, and capitalization), style editing (conformance to the Computer Society style guide), and content editing (flow, meaning, clarity, directness, and organization). The final editing will be a collaborative process in which the Editors of the *SWEBOK Guide* and the KA Editors work together to achieve a concise, well-worded, and useful KA Description.

RELEASE OF COPYRIGHT

All intellectual property rights associated with the *SWEBOK Guide* will remain with the IEEE. KA Editors must sign a copyright release form.

It is also understood that the *SWEBOK Guide* will continue to be available free of charge in the public domain in at least one format, provided by the IEEE Computer Society through web technology or by other means.

For more information, see www.computer.org/copyright.htm.

REFERENCES

[1] Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK(R) Guide)*, 5th ed., Project Management Institute, 2013.

[2] Integrated Software and Systems Engineering Curriculum (iSSEc) Project, *Graduate Software Engineering 2009 (GSwE2009): Curriculum Guidelines for Graduate Degree Programs in Software Engineering*, Stevens Institute of Technology, 2009; www.gswe2009.org.

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

[4*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006.

[5] Joint Task Force on Computing Curricula, IEEE Computer Society and Association for Computing Machinery, *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, 2004; http://sites.computer.org/ccse/SE2004Volume.pdf.

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

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

[8] IEEE Computer Society, "Certification and Training for Software Professionals," 2013; www.computer.org/certification.

APPENDIX B

IEEE AND ISO/IEC STANDARDS SUPPORTING THE SOFTWARE ENGINEERING BODY OF KNOWLEDGE (SWEBOK)

Some might say that the supply of software engineering standards far exceeds the demand. One seldom listens to a briefing on the subject without suffering some apparently obligatory joke that there are too many of them. However, the existence of standards takes a very large (possibly infinite) trade space of alternatives and reduces that space to a smaller set of choices—a huge advantage for users. Nevertheless, it can still be difficult to choose from dozens of alternatives, so supplementary guidance, like this appendix, can be helpful. A summary list of the standards mentioned in this appendix appears at the end.

To reduce tedium in reading, a few simplifications and abridgements are made in this appendix:

- ISO/IEC JTC 1/SC 7 maintains nearly two hundred standards on the subject. IEEE maintains about fifty. The two organizations are in the tenth year of a systematic program to coordinate and integrate their collections. In general, this article will focus on the standards that are recognized by both organizations, taking this condition as evidence that wide agreement has been obtained. Other standards will be mentioned briefly.
- Standards tend to have long, taxonomical titles. If there were a single standard for building an automobile, the one for your Camry probably would be titled something like, "Vehicle, internal combustion, four-wheel, passenger, sedan." Also, modern standards organizations provide their standards from databases. Like any database, these sometimes contain errors, particularly for the titles. So this article will often paraphrase the title of the standard or simply use its number. In obtaining a standard of interest, the reader should rely on the number, not the title, given in this article. For reasons of consistency, the article will use the IEEE's convention for the capitalization of titles—nouns, pronouns, adjectives, verbs, adverbs, and first and last words have an initial capital letter—despite the fact that IEEE and ISO/IEC use differing conventions.
- Because these standards are being continually revised to take account of new technologies and usage patterns, this article will be obsolescent before it is published. Therefore, it will occasionally discuss standards that have not yet been published, if they are likely to assume significant importance.
- Explicit trademarks are omitted. Suffice it to say that IEEE places a trademark on all of its standards' designations.

There are some other conventions of interest:

- In both IEEE and ISO/IEC, standards for *systems* engineering are maintained by the same committee as those for *software* engineering. Many of the standards apply to both. So, instead of making fine distinctions, this article will deal with both.
- On the other hand, both S2ESC and SC 7 (see below for descriptions of these organizations) are responsible for standards that don't qualify as "engineering." In the US and many other countries, the services of a licensed engineer are required when a product might affect public safety, health,

and welfare as opposed to affecting merely the pocketbook of the client. This appendix will respect that distinction and ignore standards that appear to be merely economic in consequence.

- User documentation is assumed to be developed similarly to software. For example, a standard concerning the design of user documentation is described in the Software Design KA.
- Some jointly developed standards are explicitly labeled as joint developments, e.g., ISO/IEC/IEEE 24765. In other cases, the standards have different designations in the two organizations. Examples include

  » IEEE Std. 12207:2008 (a.k.a. ISO/IEC 12207:2008), where "a.k.a." ("also known as") is this appendix's abbreviation to note the designation in the other organization;
  » IEEE Std. 15939:2008 Standard Adoption of ISO/IEC 15939:2007, an adoption by IEEE of a standard developed in ISO/IEC;
  » IEEE Std. 1220:2005 (a.k.a. ISO/IEC 26702:2007), a "fast-track" by ISO/IEC of a standard developed in IEEE.

  In each of these cases, the standards are substantively identical in the two organizations, differing only in front matter and, occasionally, added informational material.

A summary list of all of the mentioned standards is provided at the end of this appendix.

ISO/IEC JTC 1/SC 7, SOFTWARE AND SYSTEMS ENGINEERING

ISO/IEC JTC 1/SC 7 is the major source of international standards on software and systems engineering. Its name is formed taxonomically. Joint Technical Committee 1 (JTC 1) is a child of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC); it has the scope of "information technology" and subdivides its work among a number of subcommittees; Subcommittee 7 (SC 7) is the one responsible for software and systems engineering. SC 7, and its working groups, meets twice a year, attracting delegations representing the national standards bodies of participating nations. Each nation follows its own procedures for determining national positions and each nation has the responsibility of determining whether an ISO/IEC standard should be adopted as a national standard.

SC 7 creates three types of documents:

- International Standards: Documents containing requirements that must be satisfied in order to claim conformance.
- Technical Specifications (formerly called Technical Reports, type 1 and type 2): Documents published in a preliminary manner while work continues.
- Technical Reports (formerly called Technical Reports, type 3): Documents inherently unsuited to be standards, usually because they are descriptive rather than prescriptive.

The key thing to remember is that only the first category counts as a consensus standard. The reader can easily recognize the others by the suffix TS or TR prepended to the number of the document.

IEEE SOFTWARE AND SYSTEMS ENGINEERING STANDARDS COMMITTEE (S2ESC)

IEEE is the world's largest organization of technical professionals, with about 400,000 members in more than 160 countries. The publication of standards is performed by the IEEE Standards Association (IEEE-SA), but the committees that draft and sponsor the standards are in the various IEEE societies; S2ESC is a part of the IEEE Computer Society. IEEE is a global standards maker because its standards are used in many different countries. Despite its international membership (about 50% non-US), though, the IEEE-SA routinely submits its standards to the American National Standards Institute (ANSI) for endorsement as "American National Standards." Some S2ESC standards are developed within S2ESC, some are developed jointly with SC 7, and some are adopted after being developed by SC 7.

IEEE-SA publishes three types of "standards":

- Standards, with a preponderance of the verb "shall"
- Recommended Practices, with a preponderance of the verb "should"
- Guides, with a preponderance of the verb "may."

All three of these compare to ISO/IEC standards. IEEE-SA does have the concept of a "Trial-Use" standard, which is roughly comparable to an ISO/IEC Technical Specification. However, it has nothing comparable to an ISO/IEC Technical Report; one would look elsewhere in IEEE for documents of this ilk.

THE STANDARDS

The remainder of this article allocates the selected standards to relevant knowledge areas (KAs) of the *SWEBOK Guide*. There is a section for each KA. Within each section, the relevant standards are listed—the ones that principally apply to the KA as well as others that principally apply to other KAs but which are also related to the current one. Following each standard is a brief summary. In most cases, the summary is a quotation or paraphrase of the abstract or other introductory material from the text of the standard.

Most of the standards easily fit into one KA. Some fit into more than one; in such cases, a cross-reference is provided. Two standards apply to all KAs, so they are listed in a category called "General." All of the standards related to computer-aided software engineering (CASE) tools and environments are listed in the Software Engineering Models and Methods KA section.

GENERAL

The first two standards are so central that they could be slotted into all of the KAs. Two more are described in the Software Engineering Process KA, but are mentioned here because they provide a helpful framework and because the descriptions of several other standards refer to them.

ISO/IEC TR 19759 is the *SWEBOK Guide* itself. It's not an IEEE standard because, lacking prescriptive verbs, it doesn't satisfy the criteria for any of the IEEE categories. In ISO/IEC, it is a "technical report"—defined as a document inherently unsuited to be a standard. The 2004 IEEE *SWEBOK Guide* was adopted by ISO/IEC without change. Presumably, ISO/IEC will adopt Version 3 of the *SWEBOK Guide*.

ISO/IEC TR 19759:2005 Software Engineering—*Guide to the Software Engineering Body of Knowledge* (SWEBOK)

Applies to all KAs

ISO/IEC 19759:2005, a *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, identifies and describes that subset of the body of knowledge that is generally accepted, even though software engineers must be knowledgeable not only in software engineering, but also, of course, in other related disciplines. SWEBOK is an all-inclusive term that describes the sum of knowledge within the profession of software engineering.

The text of the *SWEBOK Guide* is freely available at www.swebok.org/. The ISO/IEC adoption of the *Guide* is freely available at http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html.

ISO/IEC/IEEE 24765 provides a shared vocabulary for the systems and software engineering standards of both SC 7 and S2ESC.

ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary

Applies to all KAs

ISO/IEC/IEEE 24765:2010 provides a common vocabulary applicable to all systems and software engineering work. It was prepared to collect and support the standardization of terminology. ISO/IEC/IEEE 24765:2010 is intended to serve as a useful reference for those in the information technology field and to encourage the use of systems and software engineering standards prepared by ISO and liaison organizations IEEE Computer Society and Project Management Institute. ISO/IEC/IEEE 24765:2010 includes references to the

active source standards for each definition so that the use of the term can be further explored.

The vocabulary is descriptive, rather than prescriptive; it gathers up all of the definitions from all of the relevant standards, as well as a few other sources, rather than choosing among competing definitions.

The content of the 24765 standard is freely accessible online at www.computer.org/sevocab.

Two standards, 12207 and 15288, provide a complete set of processes for the entire life cycle of a system or a software product. The two standards are aligned for concurrent use on a single project or in a single organization. They are mentioned here because they are often used as a framework for explaining or localizing the role of other standards in the life cycle.

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

See Software Engineering Process KA

IEEE Std. 15288-2008 (a.k.a. ISO/IEC 15288:2008) Standard for Systems and *S*oftware Engineering—System Life Cycle Processes

See Software Engineering Process KA

SOFTWARE REQUIREMENTS

The primary standard for software and systems requirements engineering is a new one that replaced several existing IEEE standards. It provides a broad view of requirements engineering across the entire life cycle.

ISO/IEC/IEEE 29148:2011 Systems and Software Engineering—Life Cycle Processes—Requirements Engineering

ISO/IEC/IEEE 29148:2011 contains provisions for the processes and products related to the engineering of requirements for systems and software products and services throughout the life cycle. It defines the construct of a good requirement, provides attributes and characteristics of requirements, and discusses the iterative and recursive application of requirements processes throughout the life cycle. ISO/IEC/IEEE 29148:2011 provides additional guidance in the application of requirements engineering and management processes for requirements-related activities in ISO/IEC 12207:2008 and ISO/IEC 15288:2008. Information items applicable to the engineering of requirements and their content are defined. The content of ISO/IEC/IEEE 29148:2011 can be added to the existing set of requirements-related life cycle processes defined by ISO/IEC 12207:2008 or ISO/IEC 15288:2008, or it can be used independently.

A multipart ISO/IEC standard provides principles and methods for "sizing" software based on its requirements. The functional size is often useful in the denominator of measurements of quality and productivity in software development. It may also play a role in contracting for service-level agreements.

ISO/IEC 14143 [six parts] Information Technology—Software Measurement—Functional Size Measurement

ISO/IEC 14143 describes FSM (functional size measurement). The concepts of functional size measurement (FSM) are designed to overcome the limitations of earlier methods of sizing software by shifting the focus away from measuring how the software is implemented to measuring size in terms of the functions required by the user.

FSM is often known as "function point counting." The four standards listed below are alternative methods for function point counting—all meet the requirements of ISO/IEC 14143. The dominant method, in terms of market share, is the IFPUG method, described in ISO/IEC 20926. Other methods are variations intended to improve the validity of the count in various circumstances. For example, ISO/IEC 19761—COSMIC is

notably intended to be used on software with a real-time component.

ISO/IEC 19761:2011 Software Engineering—COSMIC: A Functional Size Measurement Method

ISO/IEC 20926:2009 Software and Systems Engineering—Software Measurement—IFPUG Functional Size Measurement Method

ISO/IEC 20968:2002 Software Engineering—Mk II Function Point Analysis—Counting Practices Manual

ISO/IEC 24570:2005 Software Engineering—NESMA Functional Size Measurement Method Version 2.1—Definitions and Counting Guidelines for the Application of Function Point Analysis

---

Sometimes requirements are described in natural language, but sometimes they are described in formal or semiformal notations. The objective of the Unified Modeling Language (UML) is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes. The two parts of ISO/IEC 19505 define UML, revision 2. The older ISO/IEC 19501 is an earlier version of UML. They are mentioned here because they are often used to model requirements.

ISO/IEC 19501:2005 Information Technology—Open Distributed Processing—Unified Modeling Language (UML) Version 1.4.2

See Software Engineering Models and Methods KA

ISO/IEC 19505:2012 [two parts] Information Technology—Object Management Group Unified Modeling Language (OMG UML)

See Software Engineering Models and Methods KA

SOFTWARE DESIGN

The software design KA includes both software architectural design (for determining the relationships among the items of the software and detailed design (for describing the individual items). ISO/IEC/IEEE 42010 concerns the description of architecture for systems and software.

ISO/IEC/IEEE 42010:2011 Systems and Software Engineering—Architecture Description

ISO/IEC/IEEE 42010:2011 addresses the creation, analysis, and sustainment of architectures of systems through the use of architecture descriptions. A conceptual model of architecture description is established. The required contents of an architecture description are specified. Architecture viewpoints, architecture frameworks and architecture description languages are introduced for codifying conventions and common practices of architecture description. The required content of architecture viewpoints, architecture frameworks and architecture description languages is specified. Annexes provide the motivation and background for key concepts and terminology and examples of applying ISO/IEC/IEEE 42010:2011.

Like ISO/IEC/IEEE 42010, the next standard treats software "design" as an abstraction, independent of its representation in a document. Accordingly, the standard places provisions on the description of design, rather than on design itself.

IEEE Std. 1016-2009 Standard for Information Technology—Systems Design—Software Design Descriptions

This standard describes software designs and establishes the information content and organization of a software design description (SDD). An SDD is a representation of a software design to be used for recording design information and communicating that design information to key design

stakeholders. This standard is intended for use in design situations in which an explicit software design description is to be prepared. These situations include traditional software construction activities (when design leads to code) and reverse engineering situations (when a design description is recovered from an existing implementation). This standard can be applied to commercial, scientific, or military software that runs on digital computers. Applicability is not restricted by the size, complexity, or criticality of the software. This standard can be applied to the description of high-level and detailed designs. This standard does not prescribe specific methodologies for design, configuration management, or quality assurance. This standard does not require the use of any particular design languages, but establishes requirements on the selection of design languages for use in an SDD. This standard can be applied to the preparation of SDDs captured as paper documents, automated databases, software development tools, or other media.

By convention, this appendix treats user documentation as a part of a software system. Therefore, the various aspects of user documentation—its design, its testing, and so forth—are allocated to different KAs. The next standard deals with the design of user documentation.

IEEE Std. 26514-2010 Standard Adoption of ISO/IEC 26514:2008 Systems and Software Engineering—Requirements for Designers and Developers of User Documentation

This standard provides requirements for the design and development of software user documentation as part of the life cycle processes. It defines the documentation process from the viewpoint of the documentation developer and also covers the documentation product. It specifies the structure, content, and format for user documentation and also provides informative guidance for user documentation style. It is independent of the software tools that may be used to produce documentation and applies to both printed documentation and onscreen documentation. Much of this standard is also applicable to user documentation for systems including hardware.

SOFTWARE CONSTRUCTION

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

There are few standards on the details of software coding. It has been found through (mostly bad) experience that coding conventions are not appropriate for standardization because, in most cases, the real benefit comes from the consistency of applying an arbitrary convention rather than the convention itself. So, although coding conventions are a good idea, it is generally left to the organization or the project to develop such a standard.

Nevertheless, the subject of secure coding has attracted attention in recent years because some coding idioms are insecure in the face of attack. A Technical Report prepared by ISO/IEC JTC 1/SC 22 (programming languages) describes vulnerabilities in programming languages and how they can be avoided.

ISO/IEC TR 24772:2013 Information Technology—Programming Languages—Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

ISO/IEC TR 24772:2013 specifies software programming language vulnerabilities to be avoided in the development of systems where assured behavior is required for security, safety, mission-critical, and business-critical software. In general, this guidance is applicable to the software developed, reviewed, or maintained for any application.

Vulnerabilities are described in a generic manner that is applicable to a broad range of programming languages. Annexes relate the generic guidance to a selection of specific programming languages.

The Technical Report is freely available at http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html.

Two standards are mentioned here because unit testing is often regarded as an activity of software construction. IEEE and ISO/IEC are cooperating in the development of a four-part joint standard, 29119, that will provide a comprehensive treatment of testing and supplant IEEE Std. 1008.

IEEE Std. 1008-1987 Standard for Software Unit Testing

See Software Testing KA

ISO/IEC/IEEE 29119 [four parts] (Draft) Software and Systems Engineering—Software Testing

See Software Testing KA

The next standard provides for the development of user documentation during an agile development process. It is mentioned here because agile development is sometimes regarded as construction.

ISO/IEC/IEEE 26515:2012 Systems and Software Engineering—Developing User Documentation in an Agile Environment

See Software Engineering Models and Methods KA

Coding is not the only way to create a software product. Often code (as well as requirements and design) is reused from previous projects or engineered for reuse in future projects. IEEE Std. 1517 is mentioned here because it provides a common framework for extending the system and software life cycle processes of IEEE Std. 12207:2008 to include the systematic practice of reuse.

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

See Software Engineering Process KA

SOFTWARE TESTING

Oddly, there are few standards for testing. IEEE Std. 829 is the most comprehensive.

IEEE Std. 829-2008 Standard for Software and System Test Documentation

Test processes determine whether the development products of a given activity conform to the requirements of that activity and whether the system and/or software satisfies its intended use and user needs. Testing process tasks are specified for different integrity levels. These process tasks determine the appropriate breadth and depth of test documentation. The documentation elements for each type of test documentation can then be selected. The scope of testing encompasses software-based systems, computer software, hardware, and their interfaces. This standard applies to software-based systems being developed, maintained, or reused (legacy, commercial off-the-shelf, nondevelopmental items). The term "software" also includes firmware, microcode, and documentation. Test processes can include inspection, analysis, demonstration, verification, and validation of software and software-based system products.

IEEE Std. 1008 focuses on unit testing.

IEEE Std. 1008-1987 Standard for Software Unit Testing

The primary objective is to specify a standard approach to software unit testing that can be used as a basis for sound software engineering practice. A second objective is to describe the software engineering concepts and testing assumptions on which the standard approach is based. A third objective is to provide guidance and resource information to assist with the implementation and usage of the standard unit testing approach.

IEEE and ISO/IEC JTC 1/SC 7 are cooperating in a project to develop a single comprehensive standard that covers all aspects of testing. One can hope for publication of the four-part standard by 2014. Portions of the content remain controversial. One taxonomical issue is whether "static methods"—such as inspection, review, and static analysis—should fall within the scope of "testing" or should be distinguished as "verification and validation." Although the resolution of the issue is probably of little importance to users of the standard, it assumes great importance to the standards-writers who must manage an integrated suite of interoperating standards.

ISO/IEC/IEEE 29119 [four parts] (Draft) Software and Systems Engineering—Software Testing

The purpose of ISO/IEC 29119 Software Testing is to define an internationally agreed standard for software testing that can be used by any organization when performing any form of software testing.

Testing of user documentation is described in the next standard, providing requirements for the test and review of software user documentation as part of the life cycle processes. It defines the documentation process from the viewpoint of the documentation tester and reviewer. It is relevant to roles involved in testing and development of software and user documentation, including project managers, usability experts, and information developers in addition to testers and reviewers.

IEEE Std. 26513-2010 Standard Adoption of ISO/IEC 26513:2009 Systems and Software Engineering—Requirements for Testers and Reviewers of Documentation

ISO/IEC 26513 provides the minimum requirements for the testing and reviewing of user documentation, including both printed and onscreen documents used in the work environment by the users of systems software. It applies to printed user manuals, online help, tutorials, and user reference documentation.

It specifies processes for use in testing and reviewing of user documentation. It is not limited to the test and review phase of the life cycle, but includes activities throughout the information management and documentation management processes.

Two standards are mentioned here because some sources consider software verification and validation to be taxonomically included in testing.

IEEE Std. 1012-2012 Standard for System and Software Verification and Validation

See Software Quality KA

IEEE Std. 1044-2009 Standard for Classification for Software Anomalies

See Software Quality KA

SOFTWARE MAINTENANCE

This standard—the result of harmonizing distinct IEEE and ISO/IEC standards on the subject—describes a single comprehensive process for the management and execution of software maintenance. It expands on the provisions of the software maintenance process provided in ISO/IEC/IEEE 12207.

IEEE Std. 14764-2006 (a.k.a. ISO/IEC 14764:2006) Standard for Software Engineering—Software Life Cycle Processes—Maintenance

ISO/IEC 14764:2006 describes in greater detail management of the maintenance process described in ISO/IEC 12207, including amendments. It also establishes definitions for the various types of maintenance. ISO/IEC 14764:2006 provides guidance that applies to planning, execution and control, review and evaluation, and closure of the maintenance process. The scope of ISO/IEC 14764:2006 includes maintenance for multiple software products with the same maintenance resources. "Maintenance" in ISO/IEC 14764:2006 means software maintenance unless otherwise stated.

ISO/IEC 14764:2006 provides the framework within which generic and specific software maintenance plans may be executed, evaluated, and tailored to the maintenance scope and magnitude of given software products. It provides the framework, precise terminology, and processes to allow the consistent application of technology (tools, techniques, and methods) to software maintenance.

It does not address the operation of software and the operational functions, e.g., backup, recovery, and system administration, which are normally performed by those who operate the software.

ISO/IEC 14764:2006 is written primarily for maintainers of software and additionally for those responsible for development and quality assurance. It may also be used by acquirers and users of systems containing software, who may provide inputs to the maintenance plan.

SOFTWARE CONFIGURATION MANAGEMENT

There is one standard for configuration management.

IEEE Std. 828-2012 Standard for Configuration Management in Systems and Software Engineering

This standard establishes the minimum requirements for processes for configuration management (CM) in systems and software engineering. The application of this standard applies to any form, class, or type of software or system. This revision of the standard expands the previous version to explain CM, including identifying and acquiring configuration items, controlling changes, reporting the status of configuration items, as well as software builds and release engineering. Its predecessor defined only the contents of a software configuration management plan. This standard addresses what CM activities are to be done, when they are to happen in the life cycle, and what planning and resources are required. It also describes the content areas for a CM plan. The standard supports ISO/IEC/IEEE 12207:2008 and ISO/IEC/IEEE 15288:2008 and adheres to the terminology in ISO/IEC/IEEE Std. 24765 and the information item requirements of IEEE Std. 15939.

ISO/IEC JTC 1/SC 7 has not yet determined what action it should take regarding the new IEEE Std. 828. There are issues concerning the extent of compatibility with ISO/IEC/IEEE 12207 and other standards in the SC 7 suite. It should be noted, though, that SC 7 does not have a competing standard.

SOFTWARE ENGINEERING MANAGEMENT

Most readers will interpret the phrase "software engineering management" to mean the management of a *project* that concerns software. There are at least two possible extensions to this generalization, though. Some software activities are managed according to a service-level agreement (SLA). SLAs do not meet the criteria for "project" according to some definitions. Also, it has become generally agreed that some management of software should occur in the organization at a level above the project, so that all projects can benefit from a common investment. A commonly cited example is the provision of software processes and tooling by the organization.

Software project management can be regarded as a specialization of "project management"—often regarded as a distinct discipline. The Project Management Institute's *Guide to the Project Management Body of Knowledge (PMBOK® Guide)* is often regarded as the authoritative source for this knowledge. From time to time, IEEE adopts the most recent version of the *PMBOK® Guide* as an IEEE standard.

IEEE Std. 1490-2011 Guide—Adoption of the Project Management Institute (PMI®) Standard, A Guide to the Project Management Body of Knowledge (PMBOK® Guide)—Fourth Edition

The *PMBOK® Guide* identifies that subset of the project management body of knowledge generally recognized as good practice. "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. "Good practice" means there is general agreement that the application of these skills, tools, and techniques can enhance the chances of success over a wide range of projects. Good practice does not mean the knowledge described should always be applied uniformly to all projects; the organization and/or project management team is responsible for determining what is appropriate for any given project. The *PMBOK® Guide* also provides and promotes a common vocabulary within the project management profession for discussing, writing, and applying project management concepts. Such a standard vocabulary is an essential element of a professional discipline. The Project Management Institute (PMI) views this standard as a foundational project management reference for its professional development programs and certifications.

The 2008 revisions of ISO/IEC/IEEE 12207 and 15288 provide project management processes for software and systems and relate them to organization-level processes as well as technical processes. The jointly developed 16326 standard, replacing two older standards, expands those provisions with guidance for application.

ISO/IEC/IEEE 16326:2009 Systems and Software Engineering—Life Cycle Processes—Project Management

ISO/IEC/IEEE 16326:2009 provides normative content specifications for project management plans covering software projects and software-intensive system projects. It also provides detailed discussion and advice on applying a set of project processes that are common to both the software and system life cycle as covered by ISO/IEC 12207:2008 (IEEE Std. 12207-2008) and ISO/IEC 15288:2008 (IEEE Std. 15288-2008), respectively. The discussion and advice are intended to aid in the preparation of the normative content of project management plans. ISO/IEC/IEEE 16326:2009 is the result of the harmonization of ISO/IEC TR 16326:1999 and IEEE Std. 1058-1998.

Particularly in high-technology applications and high-consequence projects, the management of risk is an important aspect of the overall project management responsibilities. This standard deals with that subject.

IEEE Std. 16085-2006 (a.k.a. ISO/IEC 16085:2006) Standard for Systems and Software Engineering—Software Life Cycle Processes—Risk Management

ISO/IEC 16085:2006 defines a process for the management of risk in the life cycle. It can be added to the existing set of system and software life cycle processes defined by ISO/IEC 15288 and ISO/IEC 12207, or it can be used independently.

ISO/IEC 16085:2006 can be applied equally to systems and software.

The purpose of risk management is to identify potential managerial and technical problems before they occur so that actions can be taken that reduce or eliminate the probability and/or impact of these problems should they occur. It is a critical tool for continuously determining the feasibility of project plans, for improving the search for and identification of potential problems that can affect life cycle activities and the quality and performance of products, and for improving the active management of projects.

The analysis of risk and risk mitigation depends crucially upon measurement. This international standard provides an elaboration of the measurement process from ISO/IEC/IEEE 15288:2008 and ISO/IEC/IEEE 12207:2008.

IEEE Std. 15939-2008 Standard Adoption of ISO/IEC 15939:2007 Systems and Software Engineering—Measurement Process

ISO/IEC 15939 defines a measurement process applicable to system and software engineering and management disciplines. The process is described through a model that defines the activities of the measurement process that are required to adequately specify what measurement information is required, how the measures and analysis results are to be applied, and how to determine

if the analysis results are valid. The measurement process is flexible, tailorable, and adaptable to the needs of different users.

ISO/IEC 15939:2007 identifies a process that supports defining a suitable set of measures that address specific information needs. It identifies the activities and tasks that are necessary to successfully identify, define, select, apply, and improve measurement within an overall project or organizational measurement structure. It also provides definitions for measurement terms commonly used within the system and software industries.

Software projects often require the development of user documentation. Management of the project, therefore, includes management of the documentation effort.

ISO/IEC/IEEE 26511:2012 Systems and Software Engineering—Requirements for Managers of User Documentation

ISO/IEC/IEEE 26511:2012 specifies procedures for managing user documentation throughout the software life cycle. It applies to people or organizations producing suites of documentation, to those undertaking a single documentation project, and to documentation produced internally, as well as to documentation contracted to outside service organizations. It provides an overview of the software documentation and information management processes, and also presents aspects of portfolio planning and content management that user documentation managers apply. It covers management activities in starting a project, including setting up procedures and specifications, establishing infrastructure, and building a team. It includes examples of roles needed on a user documentation team. It addresses measurements and estimates needed for management control, and the use of supporting processes such as change management, schedule and cost control, resource management, and quality management and process improvement. It includes requirements for key documents produced for user documentation management, including documentation plans and documentation management plans. ISO/IEC/IEEE 26511:2012 is independent of the software tools that may be used to produce or manage documentation, and applies to both printed documentation and onscreen documentation. Much of its guidance is applicable to user documentation for systems including hardware as well as software.

Sometimes software or system components are acquired rather than developed.

IEEE Std. 1062-1998 Recommended Practice for Software Acquisition

A set of useful quality practices that can be selected and applied during one or more steps in a software acquisition process is described. This recommended practice can be applied to software that runs on any computer system regardless of the size, complexity, or criticality of the software, but is more suited for use on modified-off-the-shelf software and fully developed software.

Sometimes user documentation is acquired regardless of whether the software it describes was acquired. The following standard deals with that subject.

ISO/IEC/IEEE 26512:2011 Systems and Software Engineering—Requirements for Acquirers and Suppliers of User Documentation

ISO/IEC/IEEE 26512:2011 was developed to assist users of ISO/IEC/IEEE 15288:2008 or ISO/IEC/IEEE 12207:2008 to acquire or supply software user documentation as part of the software life cycle processes. It defines the documentation process from the acquirer's standpoint and the supplier's standpoint. ISO/IEC/IEEE 26512:2011 covers the requirements for information items used in the acquisition of user documentation products: the acquisition plan, document specification, statement of work, request for proposals, and proposal. It provides an overview of the software user documentation and information management processes which may require acquisition and supply of software user documentation products and services. It addresses the preparation of requirements for

software user documentation. These requirements are central to the user documentation specification and statement of work. It includes requirements for primary document outputs of the acquisition and supply process: the request for proposal and the proposal for user documentation products and services. It also discusses the use of a documentation management plan and a document plan as they arise in the acquisition and supply processes. ISO/IEC/IEEE 26512:2011 is independent of the software tools that may be used to produce documentation and applies to both printed documentation and onscreen documentation. Much of its guidance is applicable to user documentation for systems including hardware as well as software.

The next two standards are mentioned here because they supply information used in management decision-making.

IEEE Std. 1028-2008 Standard for Software Reviews and Audits

See Software Quality KA

IEEE Std. 1061-1998 Standard for Software Quality Metrics Methodology

See Software Quality KA

The next standard is mentioned because it includes the manager's role in developing user documentation in an agile project.

ISO/IEC/IEEE 26515:2012 Systems and Software Engineering—Developing User Documentation in an Agile Environment

See Software Engineering Models and Methods KA

SOFTWARE ENGINEERING PROCESS

Software and systems engineering processes are central to the standardization of those two disciplines—not just because many are interested in process improvement, but also because processes are effective for the description of improved practices. For example, one might propose an improved practice for software requirements analysis. A naïve treatment might relate the description to an early stage of the life cycle model. A superior approach is to describe the practice in the context of a process that can be applied at any stage of the life cycle. The requirements analysis process, for example, is necessary for the development stage, for maintenance, and often for retirement, so an improved practice described in terms of the requirements analysis process can be applied to any of those stages.

The two key standards are ISO/IEC/IEEE 12207, *Software Life Cycle Processes*, and ISO/IEC/IEEE 15288, *System Life Cycle Processes*. The two standards have distinct histories, but they were both revised in 2008 to align their processes, permitting their interoperable use across a wide spectrum of projects ranging from a standalone software component to a system with negligible software content. Both are being revised again with the intent of containing an identical list of processes, but with provisions specialized for the respective disciplines.

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

ISO/IEC 12207:2008 establishes a common framework for software life cycle processes, with well-defined terminology that can be referenced by the software industry.

ISO/IEC 12207:2008 applies to the acquisition of systems and software products and services and to the supply, development, operation, maintenance, and disposal of software products and the software portion of a system, whether performed internally or externally to an organization. Those aspects of system definition needed to provide the context for software products and services are included.

ISO/IEC 12207:2008 also provides a process that can be employed for defining, controlling, and improving software life cycle processes.

The processes, activities and tasks of ISO/IEC 12207:2008—either alone or in conjunction with ISO/IEC 15288—may also be applied during the acquisition of a system that contains software.

IEEE Std. 15288-2008 (a.k.a. ISO/IEC 15288:2008) Standard for Systems and Software Engineering—System Life Cycle Processes

ISO/IEC 15288:2008 establishes a common framework for describing the life cycle of systems created by humans. It defines a set of processes and associated terminology. These processes can be applied at any level in the hierarchy of a system's structure. Selected sets of these processes can be applied throughout the life cycle for managing and performing the stages of a system's life cycle. This is accomplished through the involvement of all interested parties, with the ultimate goal of achieving customer satisfaction.

ISO/IEC 15288:2008 also provides processes that support the definition, control, and improvement of the life cycle processes used within an organization or a project. Organizations and projects can use these life cycle processes when acquiring and supplying systems.

ISO/IEC 15288:2008 concerns those systems that are man-made and may be configured with one or more of the following: hardware, software, data, humans, processes (e.g., processes for providing service to users), procedures (e.g., operator instructions), facilities, materials, and naturally occurring entities. When a system element is software, the software life cycle processes documented in ISO/IEC 12207:2008 may be used to implement that system element.

ISO/IEC 15288:2008 and ISO/IEC 12207:2008 are harmonized for concurrent use on a single project or in a single organization.

Those two standards specify that processes may produce items of information but do not prescribe their content or format. The next standard provides help with that.

ISO/IEC/IEEE 15289:2011 Systems and Software Engineering—Content of Life-Cycle Information Products (Documentation)

ISO/IEC/IEEE 15289:2011 provides requirements for identifying and planning the specific information items (information products, documentation) to be developed and revised during systems and software life cycles and service management processes. It specifies the purpose and content of all identified systems and software data records and life cycle information items, as well as records and information items for information technology service management. The information item contents are defined according to generic document types (description, plan, policy, procedure, report, request, and specification) and the specific purpose of the document. For simplicity of reference, each information item is described as if it were published as a separate document. However, information items may be unpublished but available in a repository for reference, divided into separate documents or volumes, or combined with other information items into one document. ISO/IEC/IEEE 15289:2011 is based on the life cycle processes specified in ISO/IEC 12207:2008 (IEEE Std. 12207-2008) and ISO/IEC 15288:2008 (IEEE Std. 15288-2008), and the service management processes specified in ISO/IEC 20000-1:2005 and ISO/IEC 20000-2:2005.

The next two guides provide supplementary information helpful in applying 12207 and 15288.

IEEE Std. 24748.2-2012 Guide—Adoption of ISO/IEC TR 24748-2:2011 Systems and Software Engineering—Life Cycle Management—Part 2: Guide to the Application of ISO/IEC 15288 (System Life Cycle Processes)

ISO/IEC TR 24748-2 is a guide for the application of ISO/IEC 15288:2008. It addresses system, life cycle, process, organizational, project, and adaptation concepts, principally through reference to ISO/IEC TR 24748-1 and ISO/IEC 15288:2008. It then gives guidance on applying ISO/IEC 15288:2008 from the aspects of strategy, planning, application in organizations, and application on projects.

IEEE Std. 24748.3-2012 Guide—Adoption of ISO/IEC TR 24748-3:2011 Systems and Software

Engineering—Life Cycle Management—Part 3: Guide to the Application of ISO/IEC 12207 (Software Life Cycle Processes)

ISO/IEC TR 24748-3 is a guide for the application of ISO/IEC 12207:2008. It addresses system, life cycle, process, organizational, project, and adaptation concepts, principally through reference to ISO/IEC TR 24748-1 and ISO/IEC 12207:2008. It gives guidance on applying ISO/IEC 12207:2008 from the aspects of strategy, planning, application in organizations, and application on projects.

The 12207 and 15288 standards provide processes covering the life cycle, but they do not provide a standard life cycle model (waterfall, incremental delivery, prototype-driven, etc). Selecting an appropriate life cycle model for a project is a major concern of ISO/IEC 24748-1.

IEEE Std. 24748.1-2011 Guide—Adoption of ISO/IEC TR 24748-1:2010 Systems and Software Engineering—Life Cycle Management—Part 1: Guide for Life Cycle Management

ISO/IEC TR 24748-1 provides information on life cycle concepts and descriptions of the purposes and outcomes of representative life cycle stages. It also illustrates the use of a life cycle model for systems in the context of ISO/IEC 15288 and provides a corresponding illustration of the use of a life cycle model for software in the context of ISO/IEC 12207. ISO/IEC TR 24748-1 additionally provides detailed discussion and advice on adapting a life cycle model for use in a specific project and organizational environment. It further provides guidance on life cycle model use by domains, disciplines and specialties. ISO/IEC TR 24748-1 gives a detailed comparison between prior and current versions of ISO/IEC 12207 and ISO/IEC 15288 as well as advice on transitioning from prior to current versions and on using their application guides. The discussion and advice are intended to provide a reference model for life cycle models, facilitate use of the updated ISO/IEC 15288 and ISO/IEC 12207, and provide a framework for the development of updated application guides for those International Standards. ISO/IEC TR 24748-1 is a result of the alignment stage of the harmonization of ISO/IEC 12207 and ISO/IEC 15288.

The next standard extends the provisions of ISO/IEC/IEEE 12207 to deal with systematic software reuse.

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

A common framework for extending the system and software life cycle processes of IEEE Std. 12207:2008 to include the systematic practice of reuse is provided. The processes, activities, and tasks to be applied during each life cycle process to enable a system and/or product to be constructed from reusable assets are specified. The processes, activities, and tasks to enable the identification, construction, maintenance, and management of assets supplied are also specified.

IEEE Std. 1220 has been widely applied as a systems engineering process and was adopted by ISO/IEC with the number 26702. Unfortunately, the standard is not completely compatible with ISO/IEC/IEEE 15288 and is being revised to solve that problem. The result will be published as ISO/IEC/IEEE 24748-4.

IEEE Std. 1220-2005 (a.k.a. ISO/IEC 26702:2007) Standard for Application and Management of the Systems Engineering Process

ISO/IEC 26702 defines the interdisciplinary tasks which are required throughout a system's life cycle to transform customer needs, requirements, and constraints into a system solution. In addition, it specifies the requirements for the systems engineering process and its application throughout the product life cycle. ISO/IEC 26702:2007 focuses on engineering activities necessary to guide product development, while ensuring

that the product is properly designed to make it affordable to produce, own, operate, maintain, and eventually dispose of without undue risk to health or the environment.

Since SC 7 and IEEE have written so many process standards, one may not be surprised to learn that their model for process description is recorded in a Technical Report.

IEEE Std. 24774-2012 Guide—Adoption of ISO/IEC TR 24474:2010 Systems and Software Engineering—Life Cycle Management—Guidelines for Process Description

An increasing number of international, national, and industry standards describe process models. These models are developed for a range of purposes including process implementation and assessment. The terms and descriptions used in such models vary in format, content, and level of prescription. ISO/IEC TR 24774:2010 presents guidelines for the elements used most frequently in describing a process: the title, purpose, outcomes, activities, task, and information item. Whilst the primary purpose of ISO/IEC TR 24774:2010 is to encourage consistency in standard process reference models, the guidelines it provides can be applied to any process model developed for any purpose.

A very small entity (VSE) is an enterprise, an organization, a department, or a project having up to 25 people. The ISO/IEC 29110 series "profiles" large standards, such as ISO/IEC 12207 for software and ISO/IEC 15288 for systems, into smaller ones for VSEs. ISO 29110 is applicable to VSEs that do not develop critical systems or critical software. Profiles provide a roadmap allowing a start-up to grow a step at a time using the ISO 29110 management and engineering guides.

ISO/IEC 29110 set of standards and technical reports are targeted by audience such as VSEs, customers, or auditors. ISO/IEC 29110 is not intended to preclude the use of different life cycles approaches such as waterfall, iterative, incremental, evolutionary, or agile.

A VSE could obtain an ISO/IEC 29110 Certification. The set of technical reports is available at no cost on the ISO website. Many ISO 29110 documents are available in English, Spanish, Portuguese, Japanese, and French.

ISO/IEC TR 29110-5-1-2:2011 Software Engineering—Lifecycle Profiles for Very Small Entities (VSEs)—Part 5-1-2: Management and Engineering Guide: Generic Profile Group: Basic Profile

ISO/IEC TR 29110-5-1-2:2011 is applicable to very small entities (VSEs). A VSE is defined as an enterprise, organization, department, or project having up to 25 people. A set of standards and guides has been developed according to a set of VSEs' characteristics and needs. The guides are based on subsets of appropriate standards elements, referred to as VSE profiles. The purpose of a VSE profile is to define a subset of ISO/IEC international standards relevant to the VSEs' context.

ISO/IEC TR 29110-5-1-2:2011 provides the management and engineering guide to the basic VSE profile applicable to VSEs that do not develop critical software. The generic profile group does not imply any specific application domain.

The next standard may be viewed as an alternative to 12207 for individual projects. The 1074 standard explains how to define processes for use on a given project. The 12207 and 15288 standards, however, focus on defining processes for organizational adoption and repeated use on many projects. The current 1074 is the update of a standard that was a predecessor of 12207.

IEEE Std. 1074-2006 Standard for Developing a Software Project Life Cycle Process

This standard provides a process for creating a software project life cycle process (SPLCP). It is primarily directed at the process architect for a given software project.

All of the standards described so far in this section provide a basis for *defining* processes. Some users are interested in *assessing* and improving their processes after implementation. The 15504 series provides for process assessment; it is currently being revised and renumbered 330xx.

ISO/IEC 15504 [ten parts] Information Technology—Process Assessment

ISO/IEC 15504-2:2003 defines the requirements for performing process assessment as a basis for use in process improvement and capability determination.

Process assessment is based on a two-dimensional model containing a process dimension and a capability dimension. The process dimension is provided by an external process reference model (such as 12207 or 15288), which defines a set of processes characterized by statements of process purpose and process outcomes. The capability dimension consists of a measurement framework comprising six process capability levels and their associated process attributes.

The assessment output consists of a set of process attribute ratings for each process assessed, termed the process profile, and may also include the capability level achieved by that process.

ISO/IEC 15504-2:2003 identifies the measurement framework for process capability and the requirements for

- performing an assessment;
- process reference models;
- process assessment models;
- verifying conformity of process assessment.

The requirements for process assessment defined in ISO/IEC 15504-2:2003 form a structure that

- facilitates self-assessment;
- provides a basis for use in process improvement and capability determination;
- takes into account the context in which the assessed process is implemented;
- produces a process rating;
- addresses the ability of the process to achieve its purpose;
- is applicable across all application domains and sizes of organization; and
- may provide an objective benchmark between organizations.

The minimum set of requirements defined in ISO/IEC 15504-2:2003 ensures that assessment results are objective, impartial, consistent, repeatable, and representative of the assessed processes. Results of conformant process assessments may be compared when the scopes of the assessments are considered to be similar; for guidance on this matter, refer to ISO/IEC 15504-4.

Several other standards are mentioned here because they are written as elaborations of the processes of 12207 or 15288. They are allocated to other KAs because each one deals with topics described in those other KAs.

IEEE Std. 828-2012 Standard for Configuration Management in Systems and Software Engineering

See Software Configuration Management KA

IEEE Std. 14764-2006 (a.k.a. ISO/IEC 14764:2006) Standard for Software Engineering—Software Life Cycle Processes—Maintenance

See Software Maintenance KA

ISO/IEC 15026-4:2012 Systems and Software Engineering—Systems and Software Assurance—Part 4: Assurance in the Life Cycle

See Software Quality KA

IEEE Std. 15939-2008 Standard Adoption of ISO/IEC 15939:2007 Systems and Software Engineering—Measurement Process

See Software Engineering Management KA

ISO/IEC 15940:2006 Information Technology—Software Engineering Environment Services

See Software Engineering Models and Methods KA

IEEE Std. 16085-2006 (a.k.a. ISO/IEC 16085:2006) Standard for Systems and Software Engineering—Software Life Cycle Processes—Risk Management

See Software Engineering Management KA

ISO/IEC/IEEE 16326:2009 Systems and Software Engineering—Life Cycle Processes—Project Management

See Software Engineering Management KA

ISO/IEC/IEEE 29148:2011 Systems and Software Engineering—Life Cycle Processes—Requirements Engineering

See Software Requirements KA

Some users desire process standards usable for IT operations or IT service management. The ISO/IEC 20000 series describe IT service management. The processes are less rigorously defined than those of the aforementioned engineering standards, but may be preferable for situations where the risks of failure involve money or customer satisfaction rather than public health, safety, and welfare. The ISO/IEC 20000 series now extend to many parts. The foundation of the series, ISO/IEC 20000-1, is briefly described below.

ISO/IEC 20000-1:2011 Information Technology—Service Management—Part 1: Service Management System Requirements

ISO/IEC 20000-1:2011 is a service management system (SMS) standard. It specifies requirements for the service provider to plan, establish, implement, operate, monitor, review, maintain, and improve an SMS. The requirements include the design, transition, delivery and improvement of services to fulfill agreed service requirements.

IEEE has adopted the first two parts of the ISO/IEC 20000 series.

SOFTWARE ENGINEERING MODELS AND METHODS

Some approaches to software engineering use methods that cut across large parts of the life cycle, rather than focusing on specific processes. "Chief Programmer" was one traditional example. "Agile development" (actually an example of traditional incremental delivery) is a current example. Neither S2ESC nor SC 7 has a standard for agile development, but there is a standard for developing user documentation in an agile project.

ISO/IEC/IEEE 26515:2012 Systems and Software Engineering—Developing User Documentation in an Agile Environment

ISO/IEC/IEEE 26515:2012 specifies the way in which user documentation can be developed in agile development projects. It is intended for use in all organizations that are using agile development or are considering implementing their projects using these techniques. It applies to people or organizations producing suites of documentation, to those undertaking a single documentation project, and to documentation produced internally, as well as to documentation contracted to outside service organizations. ISO/IEC/IEEE 26515:2012 addresses the relationship between the user documentation process and the life cycle documentation process in agile development. It describes how the information developer or project manager may plan and manage the user documentation development in an agile environment. It is intended neither to encourage nor to discourage the use of any particular agile development tools or methods.

Many methodologies are based on semiformal descriptions of the software to be constructed. These range from simple descriptive notations to models that can be manipulated and tested and, in some cases, can generate code. Two relatively old techniques start the list; the first has been widely applied for modeling processes and workflows.

IEEE Std. 1320.1-1998 Standard for Functional Modeling Language—Syntax and Semantics for IDEF0

IDEF0 function modeling is designed to represent the decisions, actions, and activities of an existing or prospective organization or system. IDEF0 graphics and accompanying texts are presented in an organized and systematic way to gain

understanding, support analysis, provide logic for potential changes, specify requirements, and support system-level design and integration activities. IDEF0 may be used to model a wide variety of systems, composed of people, machines, materials, computers, and information of all varieties, and structured by the relationships among them, both automated and nonautomated. For new systems, IDEF0 may be used first to define requirements and to specify the functions to be carried out by the future system. As the basis of this architecture, IDEF0 may then be used to design an implementation that meets these requirements and performs these functions. For existing systems, IDEF0 can be used to analyze the functions that the system performs and to record the means by which these are done.

IEEE Std. 1320.2-1998 Standard for Conceptual Modeling Language—Syntax and Semantics for IDEF1X97 (IDEFobject)

IDEF1X 97 consists of two conceptual modeling languages. The key-style language supports data/information modeling and is downward compatible with the US government's 1993 standard, FIPS PUB 184. The identity-style language is based on the object model with declarative rules and constraints. IDEF1X 97 identity style includes constructs for the distinct but related components of object abstraction: interface, requests, and realization; utilizes graphics to state the interface; and defines a declarative, directly executable rule and constraint language for requests and realizations. IDEF1X 97 conceptual modeling supports implementation by relational databases, extended relational databases, object databases, and object programming languages. IDEF1X 97 is formally defined in terms of first order logic. A procedure is given whereby any valid IDEF1X 97 model can be transformed into an equivalent theory in first order logic. That procedure is then applied to a metamodel of IDEF1X 97 to define the valid set of IDEF1X 97 models.

In recent years, the UML notation has become popular for modeling software-intensive systems. The next two standards provide two versions of the UML language.

ISO/IEC 19501:2005 Information Technology—Open Distributed Processing—Unified Modeling Language (UML) Version 1.4.2

ISO/IEC 19501 describes the Unified Modeling Language (UML), a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

ISO/IEC 19505:2012 [two parts] Information Technology—Object Management Group Unified Modeling Language (OMG UML)

ISO/IEC 19505 defines the Unified Modeling Language (UML), revision 2. The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes.

Two more standards build on the base of UML to provide additional modeling capabilities:

ISO/IEC 19506:2012 Information Technology—Object Management Group Architecture-Driven Modernization (ADM)—Knowledge Discovery Meta-Model (KDM)

ISO/IEC 19506:2012 defines a metamodel for representing existing software assets, their associations, and operational environments, referred to as the knowledge discovery metamodel (KDM). This is the first in the series of specifications related to software assurance (SwA) and architecture-driven modernization (ADM) activities. KDM facilitates

projects that involve existing software systems by insuring interoperability and exchange of data between tools provided by different vendors.

ISO/IEC 19507:2012 Information Technology—Object Management Group Object Constraint Language (OCL)

ISO/IEC 19507:2012 defines the Object Constraint Language (OCL), version 2.3.1. OCL version 2.3.1 is the version of OCL that is aligned with UML 2.3 and MOF 2.0.

Some organizations invest in software engineering environments (SEE) to assist in the construction of software. An SEE, per se, is not a replacement for sound processes. However, a suitable SEE must support the processes that have been chosen by the organization.

ISO/IEC 15940:2006 Information Technology—Software Engineering Environment Services

ISO/IEC 15940:2006 defines software engineering environment (SEE) services conceptually in a reference model that can be adapted to any SEEs to automate one or more software engineering activities. It describes services that support the process definitions as in ISO/IEC 12207 so that the set of SEE services is compatible with ISO/IEC 12207. ISO/IEC 15940:2006 can be used either as a general reference or to define an automated software process.

The selection of tooling for a software engineering environment is itself a difficult task. Two standards provide some assistance. ISO/IEC 14102:2008 defines both a set of processes and a structured set of computer-aided software engineering (CASE) tool characteristics for use in the technical evaluation and the ultimate selection of a CASE tool.

IEEE Std. 14102-2010 Standard Adoption of ISO/IEC 14102:2008 Information Technology—Guideline for the Evaluation and Selection of CASE Tools

Within systems and software engineering, computer-aided software engineering (CASE) tools represent a major part of the supporting technologies used to develop and maintain information technology systems. Their selection must be carried out with careful consideration of both the technical and management requirements.

ISO/IEC 14102:2008 defines both a set of processes and a structured set of CASE tool characteristics for use in the technical evaluation and the ultimate selection of a CASE tool. It follows the software product evaluation model defined in ISO/IEC 14598-5:1998.

ISO/IEC 14102:2008 adopts the general model of software product quality characteristics and subcharacteristics defined in ISO/IEC 9126-1:2001 and extends these when the software product is a CASE tool; it provides product characteristics unique to CASE tools.

The next document provides guidance on how to adopt CASE tools, once selected.

IEEE Std. 14471-2010 Guide—Adoption of ISO/IEC TR 14471:2007 Information Technology—Software Engineering—Guidelines for the Adoption of CASE Tools

The purpose of ISO/IEC TR 14471:2007 is to provide a recommended practice for CASE adoption. It provides guidance in establishing processes and activities that are to be applied for the successful adoption of CASE technology. The use of ISO/IEC TR 14471:2007 will help to maximize the return and minimize the risk of investing in CASE technology. However, ISO/IEC TR 14471:2007 does not establish compliance criteria.

It is best used in conjunction with ISO/IEC 14102 for CASE tool evaluation and selection. It neither dictates nor advocates particular development standards, software processes, design methods, methodologies, techniques, programming languages, or life cycle paradigms.

Within a software engineering environment, it is important for the various tools to interoperate. The following standards provide a scheme for interconnection.

IEEE Std. 1175.1-2002 Guide for CASE Tool Interconnections—Classification and Description

IEEE Std. 1175.2-2006 Recommended Practice for CASE Tool Interconnection—Characterization of Interconnections

IEEE Std. 1175.3-2004 Standard for CASE Tool Interconnections—Reference Model for Specifying Software Behavior

IEEE Std. 1175.4-2008 Standard for CASE Tool Interconnections—Reference Model for Specifying System Behavior

The purpose of this family of standards is to specify a common set of modeling concepts based on those found in commercial CASE tools for describing the operational behavior of a software system. These standards establish a uniform, integrated model of software concepts related to software functionality. They also provide a textual syntax for expressing the common properties (attributes and relationships) of those concepts as they have been used to model software behavior.

SOFTWARE QUALITY

One viewpoint of software quality starts with ISO 9001, *Quality Management Requirements*, dealing with quality policy throughout an organization. The terminology of that standard may be unfamiliar to software professionals, and quality management auditors may be unfamiliar with software jargon. The following standard describes the relationship between ISO 9001 and ISO/IEC 12207. Unfortunately, the current version refers to obsolete editions of both; a replacement is in progress:

IEEE Std. 90003-2008 Guide—Adoption of ISO/IEC 90003:2004 Software Engineering—Guidelines for the Application of ISO 9001:2000 to Computer Software

ISO/IEC 90003 provides guidance for organizations in the application of ISO 9001:2000 to the acquisition, supply, development, operation, and maintenance of computer software and related support services. ISO/IEC 90003:2004 does not add to or otherwise change the requirements of ISO 9001:2000.

The guidelines provided in ISO/IEC 90003:2004 are not intended to be used as assessment criteria in quality management system registration/certification.

The application of ISO/IEC 90003:2004 is appropriate to software that is

- part of a commercial contract with another organization,
- a product available for a market sector,
- used to support the processes of an organization,
- embedded in a hardware product, or
- related to software services.

Some organizations may be involved in all the above activities; others may specialize in one area. Whatever the situation, the organization's quality management system should cover all aspects (software related and nonsoftware related) of the business.

ISO/IEC 90003:2004 identifies the issues which should be addressed and is independent of the technology, life cycle models, development processes, sequence of activities, and organizational structure used by an organization. Additional guidance and frequent references to the ISO/IEC JTC 1/SC 7 software engineering standards are provided to assist in the application of ISO 9001:2000: in particular, ISO/IEC 12207, ISO/IEC TR 9126, ISO/IEC 14598, ISO/IEC 15939, and ISO/IEC TR 15504.

The ISO 9001 approach posits an organization-level quality management process paired with project-level quality assurance planning to achieve the organizational goals. IEEE 730 describes project-level quality planning. It is

currently aligned with an obsolete edition of 12207, but a revision is being prepared.

IEEE Std. 730-2002 Standard for Software Quality Assurance Plans

The standard specifies the format and content of software quality assurance plans.

Another viewpoint of software quality begins with enumerating the desired characteristics of a software product and selecting measures or other evaluations to determine if the desired level of characteristics has been achieved. The so-called SQuaRE (software product quality requirements and evaluation) series of SC 7 standards covers this approach in great detail.

ISO/IEC 25000 through 25099 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)

A few of the SQuaRE standards are selected below for particular attention. The first is the overall guide to the series.

ISO/IEC 25000:2005 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Guide to SQuaRE

ISO/IEC 25000:2005 provides guidance for the use of the new series of international standards named Software product Quality Requirements and Evaluation (SQuaRE). The purpose of this guide is to provide a general overview of SQuaRE contents, common reference models, and definitions, as well as the relationship among the documents, allowing users of this guide a good understanding of those international standards. This document contains an explanation of the transition process between the old ISO/IEC 9126 and the 14598 series and SQuaRE, and also presents information on how to use the ISO/IEC 9126 and 14598 series in their previous form.

SQuaRE provides

- terms and definitions,
- reference models,
- guides
- standards for requirements specification, planning and management, measurement, and evaluation purposes.

The next SQuaRE standard provides a taxonomy of software quality characteristics that may be useful in selecting characteristics relevant to a specific project:

ISO/IEC 25010:2011 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models

ISO/IEC 25010:2011 defines the following:

1. A quality in-use model composed of five characteristics (some of which are further subdivided into subcharacteristics) that relate to the outcome of interaction when a product is used in a particular context of use. This system model is applicable to the complete human-computer system, including both computer systems in use and software products in use.
2. A product quality model composed of eight characteristics (which are further subdivided into subcharacteristics) that relate to static properties of software and dynamic properties of the computer system. The model is applicable to both computer systems and software products.

The characteristics defined by both models are relevant to all software products and computer systems. The characteristics and subcharacteristics provide consistent terminology for specifying, measuring, and evaluating system and software product quality. They also provide a set of quality characteristics against which stated quality requirements can be compared for completeness.

Although the scope of the product quality model is intended to be software and computer systems, many of the characteristics are also relevant to wider systems and services.

ISO/IEC 25012 contains a model for data quality that is complementary to this model.

The scope of the models excludes purely functional properties, but it does include functional suitability.

The scope of application of the quality models includes supporting specification and evaluation of software and software-intensive computer systems from different perspectives by those who are associated with their acquisition, requirements, development, use, evaluation, support, maintenance, quality assurance and control, and audit. The models can, for example, be used by developers, acquirers, quality assurance and control staff, and independent evaluators, particularly those responsible for specifying and evaluating software product quality. Activities during product development that can benefit from the use of the quality models include

- identifying software and system requirements;
- validating the comprehensiveness of a requirements definition;
- identifying software and system design objectives;
- identifying software and system testing objectives;
- identifying quality control criteria as part of quality assurance;
- identifying acceptance criteria for a software product and/or software-intensive computer system;
- establishing measures of quality characteristics in support of these activities.

Some documents in the SQuaRE series deal specifically with the characteristic of usability. The Common Industry Format (CIF) for usability reporting began at the US National Institute for Standards and Technology (NIST) and was moved into ISO/IEC JTC 1/SC 7 for purposes of standardization.

ISO/IEC 25060 through 25064 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Common Industry Format (CIF) for Usability

A family of international standards, named the Common Industry Formats (CIF), documents the specification and evaluation of the usability of interactive systems. It provides a general overview of the CIF framework and contents, definitions, and the relationship of the framework elements. The intended users of the framework are identified, as well as the situations in which the framework may be applied. The assumptions and constraints of the framework are also enumerated.

The framework content includes the following:

- consistent terminology and classification of specification, evaluation, and reporting;
- a definition of the type and scope of formats and the high-level structure to be used for documenting required information and the results of evaluation.

The CIF family of standards is applicable to software and hardware products used for predefined tasks. The information items are intended to be used as part of system-level documentation resulting from development processes such as those in ISO 9241-210 and ISO/IEC JTC 1/SC 7 process standards.

The CIF family focuses on documenting those elements needed for design and development of usable systems, rather than prescribing a specific process. It is intended to be used in conjunction with existing international standards, including ISO 9241, ISO 20282, ISO/IEC 9126, and the SQuaRE series (ISO/IEC 25000 to ISO/IEC 25099).

The CIF family of standards does not prescribe any kind of method, life cycle or process.

Not everyone agrees with the taxonomy of quality characteristics in ISO/IEC 25010. That standard has a quality factor called "reliability" that has subfactors of maturity, availability, fault tolerance, and recoverability. IEC TC 65, which has responsibility for standards on "dependability," defines that term as a nonquantitative composite of reliability, maintainability, and maintenance support. Others use the term "reliability"

to denote a measure defined by a mathematical equation. The disagreement over the use of these words means that the standards on the subject are inherently unaligned. A few will be noted below, but the words like those noted above may mean different things in different standards.

IEEE Std. 982.1-2005 Standard for Dictionary of Measures of the Software Aspects of Dependability

A standard dictionary of measures of the software aspects of dependability for assessing and predicting the reliability, maintainability, and availability of any software system; in particular, it applies to mission critical software systems.

IEEE Std. 1633-2008 Recommended Practice for Software Reliability

The methods for assessing and predicting the reliability of software, based on a life cycle approach to software reliability engineering, are prescribed in this recommended practice. It provides information necessary for the application of software reliability (SR) measurement to a project, lays a foundation for building consistent methods, and establishes the basic principle for collecting the data needed to assess and predict the reliability of software. The recommended practice prescribes how any user can participate in SR assessments and predictions.

IEEE has an overall standard for software product quality that has a scope similar to the ISO/IEC 250xx series described previously. Its terminology differs from the ISO/IEC series, but it is substantially more compact.

IEEE Std. 1061-1998 Standard for Software Quality Metrics Methodology

A methodology for establishing quality requirements and identifying, implementing, analyzing, and validating the process and product software quality metrics is defined. The methodology spans the entire software life cycle.

One approach to achieving software quality is to perform an extensive program of verification and validation. IEEE Std. 1012 is probably the world's most widely applied standard on this subject. A revision was recently published.

IEEE Std. 1012-2012 Standard for System and Software Verification and Validation

Verification and validation (V&V) processes are used to determine whether the development products of a given activity conform to the requirements of that activity and whether the product satisfies its intended use and user needs. V&V life cycle process requirements are specified for different integrity levels. The scope of V&V processes encompasses systems, software, and hardware, and it includes their interfaces. This standard applies to systems, software, and hardware being developed, maintained, or reused [legacy, commercial off-the-shelf (COTS), nondevelopmental items]. The term software also includes firmware and microcode, and each of the terms system, software, and hardware includes documentation. V&V processes include the analysis, evaluation, review, inspection, assessment, and testing of products.

There are other standards that support the verification and validation processes. One describes techniques for performing reviews and audits during a software project.

IEEE Std. 1028-2008 Standard for Software Reviews and Audits

Five types of software reviews and audits, together with procedures required for the execution of each type, are defined in this standard. This standard is concerned only with the reviews and audits; procedures for determining the necessity of a review or audit are not defined, and the disposition of the results of the review or audit is not specified. Types included are management reviews, technical reviews, inspections, walkthroughs, and audits.

In many cases, a database of software anomalies is used to support verification and validation activities. The following standard suggests how anomalies should be classified.

IEEE Std. 1044-2009 Standard for Classification for Software Anomalies

This standard provides a uniform approach to the classification of software anomalies, regardless of when they originate or when they are encountered within the project, product, or system life cycle. Classification data can be used for a variety of purposes, including defect causal analysis, project management, and software process improvement (e.g., to reduce the likelihood of defect insertion and/or increase the likelihood of early defect detection).

In some systems, one particular property of the software is so important that it requires special treatment beyond that provided by a conventional verification and validation program. The emerging term for this sort of treatment is "systems and software assurance." Examples include safety, privacy, high security, and ultrareliability. The 15026 standard is under development to deal with such situations. The first part of the four-part standard provides terminology and concepts used in the remaining parts. It was first written before the other parts and is now being revised for complete agreement with the others.

IEEE Std. 15026.1-2011 Trial-Use Standard Adoption of ISO/IEC TR 15026-1:2010 Systems and Software Engineering—Systems and Software Assurance—Part 1: Concepts and Vocabulary

This trial-use standard adopts ISO/IEC TR 15026-1:2010, which defines terms and establishes an extensive and organized set of concepts and their relationships for software and systems assurance, thereby establishing a basis for shared understanding of the concepts and principles central to ISO/IEC 15026 across its user communities. It provides information to users of the subsequent parts of ISO/IEC 15026, including the use of each part and the combined use of multiple parts. Coverage of assurance for a service being operated and managed on an ongoing basis is not covered in ISO/IEC 15026.

The second part of the standard describes the structure of an "assurance case," which is intended as a structured argument that the critical property has been achieved. It is a generalization of various domain-specific constructs like "safety cases."

IEEE Std. 15026.2-2011 Standard Adoption of ISO/IEC 15026-2:2011 Systems and Software Engineering—Systems and Software Assurance—Part 2: Assurance Case

ISO/IEC 15026-2:2011 is adopted by this standard. ISO/IEC 15026-2:2011 specifies minimum requirements for the structure and contents of an assurance case to improve the consistency and comparability of assurance cases and to facilitate stakeholder communications, engineering decisions, and other uses of assurance cases. An assurance case includes a top-level claim for a property of a system or product (or set of claims), systematic argumentation regarding this claim, and the evidence and explicit assumptions that underlie this argumentation. Arguing through multiple levels of subordinate claims, this structured argumentation connects the top-level claim to the evidence and assumptions. Assurance cases are generally developed to support claims in areas such as safety, reliability, maintainability, human factors, operability, and security, although these assurance cases are often called by more specific names, e.g., safety case or reliability and maintainability (R&M) case. ISO/IEC 15026-2:2011 does not place requirements on the quality of the contents of an assurance case and does not require the use of a particular terminology or graphical representation. Likewise, it places no requirements on the means of physical implementation of the data, including no requirements for redundancy or colocation.

In many systems, some portions are critical to achieving the desired property while others are only

incidental. For example, the flight control system of an airliner is critical to safety, but the microwave oven is not. Conventionally, the various portions are assigned "criticality levels" to indicate their significance to the overall achievement of the property. The third part of ISO/IEC 15026 describes how that is done. This part will be revised for better fit with the remainder of the 15026 standard.

ISO/IEC 15026-3:2011 Systems and Software Engineering—Systems and Software Assurance—Part 3: System Integrity Levels

ISO/IEC 15026-3:2011 specifies the concept of integrity levels with corresponding integrity level requirements that are required to be met in order to show the achievement of the integrity level. It places requirements on and recommends methods for defining and using integrity levels and their integrity level requirements, including the assignment of integrity levels to systems, software products, their elements, and relevant external dependences.

ISO/IEC 15026-3:2011 is applicable to systems and software and is intended for use by:

- definers of integrity levels such as industry and professional organizations, standards organizations, and government agencies;
- users of integrity levels such as developers and maintainers, suppliers and acquirers, users, and assessors of systems or software, and for the administrative and technical support of systems and/or software products.

One important use of integrity levels is by suppliers and acquirers in agreements; for example, to aid in assuring safety, economic, or security characteristics of a delivered system or product.

ISO/IEC 15026-3:2011 does not prescribe a specific set of integrity levels or their integrity level requirements. In addition, it does not prescribe the way in which integrity level use is integrated with the overall system or software engineering life cycle processes.

ISO/IEC 15026-3:2011 can be used alone or with other parts of ISO/IEC 15026. It can be used with a variety of technical and specialized risk analysis and development approaches. ISO/IEC TR 15026-1 provides additional information and references to aid users of ISO/IEC 15026-3:2011.

ISO/IEC 15026-3:2011 does not require the use of the assurance cases described by ISO/IEC 15026-2 but describes how integrity levels and assurance cases can work together, especially in the definition of specifications for integrity levels or by using integrity levels within a portion of an assurance case.

The final part of 15026 provides additional guidance for executing the life cycle processes of 12207 and 15288 when a system or software is required to achieve an important property.

ISO/IEC 15026-4:2012 Systems and Software Engineering—Systems and Software Assurance—Part 4: Assurance in the Life Cycle

This part of ISO/IEC 15026 gives guidance and recommendations for conducting selected processes, activities and tasks for systems and software products requiring assurance claims for properties selected for special attention, called critical properties. This part of ISO/IEC 15026 specifies a property-independent list of processes, activities, and tasks to achieve the claim and show the achievement of the claim. This part of ISO/IEC 15026 establishes the processes, activities, tasks, guidance, and recommendations in the context of a defined life cycle model and set of life cycle processes for system and/or software life cycle management.

The next standard deals with a property—safety—that is often identified as critical. It was originally developed in cooperation with the US nuclear power industry.

IEEE Std. 1228-1994 Standard for Software Safety Plans

The minimum acceptable requirements for the content of a software safety plan are established. This standard applies to the software safety plan used for the development, procurement, maintenance, and retirement of safety-critical software.

This standard requires that the plan be prepared within the context of the system safety program. Only the safety aspects of the software are included. This standard does not contain special provisions required for software used in distributed systems or in parallel processors.

Classical treatments suggest that "verification" deals with static evaluation methods and that "testing" deals with dynamic evaluation methods. Recent treatments, including ISO/IEC draft 29119, are blurring this distinction, though, so testing standards are mentioned here.

IEEE Std. 829-2008 Standard for Software and System Test Documentation

See Software Testing KA

IEEE Std. 1008-1987 Standard for Software Unit Testing

See Software Testing KA

IEEE Std. 26513-2010 Standard Adoption of ISO/IEC 26513:2009 Systems and Software Engineering—Requirements for Testers and Reviewers of Documentation

See Software Testing KA

ISO/IEC/IEEE 29119 [four parts] (Draft) Software and Systems Engineering—Software Testing

See Software Testing KA

SOFTWARE ENGINEERING PROFESSIONAL PRACTICE

IEEE is a provider of products related to the certification of professional practitioners of software engineering. The first has already been described, the *Guide to the Software Engineering Body of Knowledge*. The *SWEBOK Guide* has been adopted by ISO/IEC as an outline of the knowledge that professional software engineers should have.

ISO/IEC TR 19759:2005 Software Engineering—Guide to the Software Engineering Body of Knowledge (SWEBOK)

See General

An SC 7 standard provides a framework for comparisons among certifications of software engineering professionals. That standard states that the areas considered in certification must be mapped to the *SWEBOK Guide*.

ISO/IEC 24773:2008 Software Engineering—Certification of Software Engineering Professionals

ISO/IEC 24773:2008 establishes a framework for comparison of schemes for certifying software engineering professionals. A certification scheme is a set of certification requirements for software engineering professionals. ISO/IEC 24773:2008 specifies the items that a scheme is required to contain and indicates what should be defined for each item.

ISO/IEC 24773:2008 will facilitate the portability of software engineering professional certifications between different countries or organizations. At present, different countries and organizations have adopted different approaches on the topic, which are implemented by means of regulations and bylaws. The intention of ISO/IEC 24773:2008 is to be open to these individual approaches by providing a framework for expressing them in a common scheme that can lead to understanding.

SC 7 is currently drafting a guide that will supplement 24773.

SOFTWARE ENGINEERING ECONOMICS

No standards are allocated to this KA.

COMPUTING FOUNDATIONS

No standards are allocated to this KA.

MATHEMATICAL FOUNDATIONS

No standards are allocated to this KA.

ENGINEERING FOUNDATIONS

No standards are allocated to this KA.

STAYING CURRENT

This article was obsolescent the moment it was drafted. Some readers will need to know how to get current designations and descriptions of standards. This section describes some helpful resources.

WHERE TO FIND STANDARDS

The list of standards published for ISO/IEC JTC 1/SC 7 can be found at www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_tc_browse.htm?commid=45086.

Because the URL might change, readers might have to navigate to the list. Begin at www.iso.org/iso/store.htm, then click on "browse standards catalogue," then "browse by TC," then "JTC 1," then "SC 7."

Finding the current list of standards for S2ESC is a bit more difficult. Begin at http://standards.ieee.org/. In the search box under "Find Standards," type "S2ESC." This should produce a list of published standards for which S2ESC is responsible.

Keep in mind that the searchable databases are compilations. Like any such database, they can contain errors that lead to incomplete search results.

WHERE TO OBTAIN THE STANDARDS

Some readers will want to obtain standards described in this article. The first thing to know is that some international standards are available free for individual use. The current list of ISO/IEC standards available under these terms is located at http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html.

One of the publicly available standards is the ISO/IEC adoption of the *SWEBOK Guide*, ISO/IEC 19759.

The definitions contained in ISO/IEC/IEEE 24765, *System and Software Vocabulary*, are freely available at www.computer.org/sevocab.

However, the vast majority of standards are not free. ISO/IEC standards are generally purchased from the national standards organization of the country in which one lives. For example, in the US, international standards can be purchased from the American National Standards Institute at http://webstore.ansi.org/. Alternatively, standards can be purchased directly from ISO/IEC at www.iso.org/iso/store.htm. It should be noted that each individual nation is free to set its own prices, so it may be helpful to check both sources.

IEEE standards may be available to you for free if your employer or library has a subscription to IEEE Xplore: http://ieeexplore.ieee.org/. Some subscriptions to Xplore provide access only to the abstracts of standards; the full text may then be purchased via Xplore. Alternatively, standards may be purchased via the IEEE standards store at www.techstreet.com/ieeegate.html. It should be noted that IEEE-SA sometimes bundles standards into groups available at a substantial discount.

Finally, the reader should note that standards that IEEE has adopted from ISO/IEC, standards that ISO/IEC has "fast-tracked" from IEEE, and standards that were jointly developed or revised are available from both sources. For all standards described in this article, the IEEE version and the ISO/IEC version are substantively identical. The respective versions may have different front and back matter but the bodies are identical.

WHERE TO SEE THE SWEBOK GUIDE

The *SWEBOK Guide* is published under an IEEE copyright. The current version of the *SWEBOK Guide* is available free to the public at www.swebok.org/. The ISO/IEC adoption of the *SWEBOK Guide*, ISO/IEC TR 19759, is one of the freely available standards.

SUMMARY LIST OF THE STANDARDS

Number and Title (listed in order of number)	Most Relevant KA
IEEE Std. 730-2002 Standard for Software Quality Assurance Plans	SW Quality
IEEE Std. 828-2012 Standard for Configuration Management in Systems and Software Engineering	SW Configuration Management
IEEE Std. 829-2008 Standard for Software and System Test Documentation	SW Testing
IEEE Std. 982.1-2005 Standard for Dictionary of Measures of the Software Aspects of Dependability	SW Quality
IEEE Std. 1008-1987 Standard for Software Unit Testing	SW Testing
IEEE Std. 1012-2012 Standard for System and Software Verification and Validation	SW Quality
IEEE Std. 1016-2009 Standard for Information Technology—Systems Design—Software Design Descriptions	SW Design
IEEE Std. 1028-2008 Standard for Software Reviews and Audits	SW Quality
IEEE Std. 1044-2009 Standard for Classification for Software Anomalies	SW Quality
IEEE Std. 1061-1998 Standard for Software Quality Metrics Methodology	SW Quality
IEEE Std. 1062-1998 Recommended Practice for Software Acquisition	SW Engineering Management
IEEE Std. 1074-2006 Standard for Developing a Software Project Life Cycle Process	SW Engineering Process
IEEE Std. 1175.1-2002 Guide for CASE Tool Interconnections—Classification and Description	SW Engineering Models and Methods
IEEE Std. 1175.2-2006 Recommended Practice for CASE Tool Interconnection—Characterization of Interconnections	SW Engineering Models and Methods
IEEE Std. 1175.3-2004 Standard for CASE Tool Interconnections—Reference Model for Specifying Software Behavior	SW Engineering Models and Methods
IEEE Std. 1175.4-2008 Standard for CASE Tool Interconnections—Reference Model for Specifying System Behavior	SW Engineering Models and Methods
IEEE Std. 1220-2005 (a.k.a. ISO/IEC 26702:2007) Standard for Application and Management of the Systems Engineering Process	SW Engineering Process
IEEE Std. 1228-1994 Standard for Software Safety Plans	SW Quality
IEEE Std. 1320.1-1998 Standard for Functional Modeling Language—Syntax and Semantics for IDEF0	SW Engineering Models and Methods
IEEE Std. 1320.2-1998 Standard for Conceptual Modeling Language—Syntax and Semantics for IDEF1X97 (IDEFobject)	SW Engineering Models and Methods
IEEE Std. 1490-2011 Guide—Adoption of the Project Management Institute (PMI®) Standard, A Guide to the Project Management Body of Knowledge (PMBOK® Guide)—Fourth Edition	SW Engineering Management
IEEE Std. 1517-2010 Standard for Information Technology—System and Software Life Cycle Processes—Reuse Processes	SW Engineering Process

Number and Title (listed in order of number)	Most Relevant KA
IEEE Std. 1633-2008 Recommended Practice for Software Reliability	SW Quality
IEEE Std. 12207-2008 (a.k.a. ISO/IEC 12207:2008) Standard for Systems and Software Engineering—Software Life Cycle Processes	SW Engineering Process
IEEE Std. 14102-2010 Standard Adoption of ISO/IEC 14102:2008 Information Technology—Guideline for the Evaluation and Selection of CASE Tools	SW Engineering Models and Methods
ISO/IEC 14143 [six parts] Information Technology—Software Measurement—Functional Size Measurement	SW Requirements
IEEE Std. 14471-2010 Guide—Adoption of ISO/IEC TR 14471:2007 Information Technology—Software Engineering—Guidelines for the Adoption of CASE Tools	SW Engineering Models and Methods
IEEE Std. 14764-2006 (a.k.a. ISO/IEC 14764:2006) Standard for Software Engineering—Software Life Cycle Processes—Maintenance	SW Maintenance
IEEE Std. 15026.1-2011 Trial-Use Standard Adoption of ISO/IEC TR 15026-1:2010 Systems and Software Engineering—Systems and Software Assurance—Part 1: Concepts and Vocabulary	SW Quality
IEEE Std. 15026.2-2011 Standard Adoption of ISO/IEC 15026-2:2011 Systems and Software Engineering—Systems and Software Assurance—Part 2: Assurance Case	SW Quality
ISO/IEC 15026-3 Systems and Software Engineering—Systems and Software Assurance—Part 3: System Integrity Levels	SW Quality
ISO/IEC 15026-4:2012 Systems and Software Engineering—Systems and Software Assurance—Part 4: Assurance in the Life Cycle	SW Quality
IEEE Std. 15288-2008 (a.k.a. ISO/IEC 15288:2008) Standard for Systems and Software Engineering—System Life Cycle Processes	SW Engineering Process
ISO/IEC/IEEE 15289:2011 Systems and Software Engineering—Content of Life-Cycle Information Products (Documentation)	SW Engineering Process
ISO/IEC 15504 [ten parts] Information Technology—Process Assessment	SW Engineering Process
IEEE Std. 15939-2008 Standard Adoption of ISO/IEC 15939:2007 Systems and Software Engineering—Measurement Process	SW Engineering Management
ISO/IEC 15940:2006 Information Technology—Software Engineering Environment Services	SW Engineering Models and Methods
IEEE Std. 16085-2006 (a.k.a. ISO/IEC 16085:2006) Standard for Systems and Software Engineering—Software Life Cycle Processes—Risk Management	SW Engineering Management
ISO/IEC/IEEE 16326:2009 Systems and Software Engineering—Life Cycle Processes—Project Management	SW Engineering Management
ISO/IEC 19501:2005 Information Technology—Open Distributed Processing—Unified Modeling Language (UML) Version 1.4.2	SW Engineering Models and Methods

Number and Title (listed in order of number)	Most Relevant KA
ISO/IEC 19505:2012 [two parts] Information Technology—Object Management Group Unified Modeling Language (OMG UML)	SW Engineering Models and Methods
ISO/IEC 19506:2012 Information Technology—Object Management Group Architecture-Driven Modernization (ADM)—Knowledge Discovery Meta-Model (KDM)	SW Engineering Models and Methods
ISO/IEC 19507:2012 Information Technology—Object Management Group Object Constraint Language (OCL)	SW Engineering Models and Methods
ISO/IEC TR 19759:2005 Software Engineering—Guide to the Software Engineering Body of Knowledge (SWEBOK)	[General]
ISO/IEC 19761:2011 Software Engineering—COSMIC: A Functional Size Measurement Method	SW Requirements
ISO/IEC 20000-1:2011 Information Technology—Service Management—Part 1: Service management system requirements	SW Engineering Process
ISO/IEC 20926:2009 Software and Systems Engineering—Software Measurement—IFPUG Functional Size Measurement Method	SW Requirements
ISO/IEC 20968:2002 Software Engineering—Mk II Function Point Analysis—Counting Practices Manual	SW Requirements
ISO/IEC 24570:2005 Software Engineering—NESMA Functional Size Measurement Method Version 2.1—Definitions and Counting Guidelines for the Application of Function Point Analysis	SW Requirements
IEEE Std. 24748.1-2011 Guide—Adoption of ISO/IEC TR 24748-1:2010 Systems and Software Engineering—Life Cycle Management—Part 1: Guide for Life Cycle Management	SW Engineering Process
IEEE Std. 24748.2-2012 Guide—Adoption of ISO/IEC TR 24748-2:2011 Systems and Software Engineering—Life Cycle Management—Part 2: Guide to the Application of ISO/IEC 15288 (System Life Cycle Processes)	SW Engineering Process
IEEE Std. 24748-3:2012 Guide—Adoption of ISO/IEC TR 24748-3:2011 Systems and Software Engineering—Life Cycle Management—Part 3: Guide to the Application of ISO/IEC 12207 (Software Life Cycle Processes)	SW Engineering Process
ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary	[General]
ISO/IEC TR 24772:2013 Information technology—Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use	SW Construction
ISO/IEC 24773:2008 Software Engineering—Certification of Software Engineering Professionals	SW Engineering Professional Practice
IEEE Std. 24774:2012 Guide—Adoption of ISO/IEC TR 24474:2010 Systems and Software Engineering—Life Cycle Management—Guidelines for Process Description	SW Engineering Process
ISO/IEC 25000:2005 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Guide to SQuaRE	SW Quality

Number and Title (listed in order of number)	Most Relevant KA
ISO/IEC 25000 through 25099 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)	SW Quality
ISO/IEC 25010:2011 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models	SW Quality
ISO/IEC 25060 through 25064 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Common Industry Format (CIF) for Usability	SW Quality
ISO/IEC/IEEE 26511:2012 Systems and Software Engineering—Requirements for Managers of User Documentation	SW Engineering Management
ISO/IEC/IEEE 26512:2011 Systems and Software Engineering—Requirements for Acquirers and Suppliers of User Documentation	SW Engineering Management
IEEE Std. 26513-2010 Standard Adoption of ISO/IEC 26513:2009 Systems and Software Engineering—Requirements for Testers and Reviewers of Documentation	SW Testing
IEEE Std. 26514-2010 Standard Adoption of ISO/IEC 26514:2008 Systems and Software Engineering—Requirements for Designers and Developers of User Documentation	SW Design
ISO/IEC/IEEE 26515:2012 Systems and Software Engineering—Developing User Documentation in an Agile Environment	SW Engineering Models and Methods
ISO/IEC 29110 [several parts] Software Engineering—Lifecycle Profiles for Very Small Entities (VSE)	SW Engineering Process
ISO/IEC/IEEE 29119 [four parts] (Draft) Software and Systems Engineering—Software Testing	SW Testing
ISO/IEC/IEEE 29148:2011 Systems and Software Engineering—Life Cycle Processes—Requirements Engineering	SW Requirements
ISO/IEC/IEEE 42010:2011 Systems and Software Engineering—Architecture Description	SW Design
IEEE Std. 90003:2008 Guide—Adoption of ISO/IEC 90003:2004 Software Engineering—Guidelines for the Application of ISO 9001:2000 to Computer Software	SW Quality

APPENDIX C

CONSOLIDATED REFERENCE LIST

The Consolidated Reference List identifies all recommended reference materials (to the level of section number) that accompany the breakdown of topics within each knowledge area (KA). This Consolidated Reference List is adopted by the software engineering certification and associated professional development products offered by the IEEE Computer Society. KA Editors used the references allocated to their KA by the Consolidated Reference List as their Recommended References.

Collectively this Consolidated Reference List is

- Complete: Covering the entire scope of the *SWEBOK Guide*.
- Sufficient: Providing enough information to describe "generally accepted" knowledge.
- Consistent: Not providing contradictory knowledge nor conflicting practices.
- Credible: Recognized as providing expert treatment.
- Current: Treating the subject in a manner that is commensurate with currently generally accepted knowledge.
- Succinct: As short as possible (both in number of reference items and in total page count) without failing other objectives.

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

[2*] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley, 2002.

[3*] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2003.

[4*] F. Bott et al., *Professional Issues in Software Engineering*, 3rd ed., Taylor & Francis, 2000.

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

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

[7*] E.W. Cheney and D.R. Kincaid, *Numerical Mathematics and Computing*, 6th ed., Brooks/Cole, 2007.

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

[9*] R.E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

[10*] D. Galin, *Software Quality Assurance: From Theory to Implementation*, Pearson Education Limited, 2004.

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

[12*] P. Grubb and A.A. Takang, *Software Maintenance: Concepts and Practice*, 2nd ed., World Scientific Publishing, 2003.

[13*] A.M.J. Hass, *Configuration Management Principles and Practices*, 1st ed., Addison-Wesley, 2003.

[14*] E. Horowitz et al., *Computer Algorithms*, 2nd ed., Silicon Press, 2007.

[15*] IEEE CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices, "Software Engineering Code of Ethics and Professional Practice (Version 5.2)," 1999; www.acm.org/serving/se/code.htm.

[16*] *IEEE Std. 828-2012, Standard for Configuration Management in Systems and Software Engineering*, IEEE, 2012.

[17*] *IEEE Std. 1028-2008, Software Reviews and Audits*, IEEE, 2008.

[18*] *ISO/IEC 14764 IEEE Std. 14764-2006, Software Engineering—Software Life Cycle Processes—Maintenance*, IEEE, 2006.

[19*] S.H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed., Addison-Wesley, 2002.

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

[21*] J. McGarry et al., *Practical Software Measurement: Objective Information for Decision Makers*, Addison-Wesley Professional, 2001.

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

[23*] D.C. Montgomery and G.C. Runger, *Applied Statistics and Probability for Engineers*, 4th ed., Wiley, 2007.

[24*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, 1st ed., Wiley-IEEE Computer Society Press, 2006.

[25*] S. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*, Wiley-Spektrum, 2008.

[26*] J. Nielsen, Usability Engineering, 1st ed., Morgan Kaufmann, 1993.

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

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

[29*] K. Rosen, Discrete Mathematics and Its Applications, 7th ed., McGraw-Hill, 2011.

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

[31*] H.M. Sneed, "Offering Software Maintenance as an Offshore Service," *Proc. IEEE Int'l Conf. Software Maintenance* (ICSM 08), IEEE, 2008, pp. 1–5.

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

[33*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, 1st ed., Addison-Wesley, 2004.

[34*] G. Voland, *Engineering by Design*, 2nd ed., Prentice Hall, 2003.

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

[36*] J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, vol. 23, no. 9, 1990, pp. 8, 10–23.