Guide to the Software Engineering Body of Knowledge

A Stone Man Version

(Version 0.7)

SWEBOK

April 2000

A project of the Software Engineering Coordinating Committee
(Joint IEEE Computer Society - ACM committee)

IEEE COMPUTER SOCIETY

acm

Corporate support by:

BOEING

NIST

National Research Council Canada
Conseil national de recherches Canada

Rational
the e-development company™

Raytheon

SAP™

Project managed by:

UQÀM

Executive Editors:

Alain Abran, Université du Québec à Montréal
James W. Moore, The MITRE Corp.

Editors:

Pierre Bourque, Université du Québec à Montréal
Robert Dupuis, Université du Québec à Montréal

Chair of the Software Engineering Coordinating Committee

Leonard L. Tripp, IEEE Computer Society

PREFACE TO THE SWEBOK GUIDE

1. Software engineering is an emerging discipline but there are unmistakable trends indicating an increasing level of maturity:

2. - McMaster University (Canada), the Rochester Institute of Technology (US), the University of Sheffield (UK), the University of New South Wales (Australia) and other universities around the world now offer undergraduate degrees in software engineering.

3. - The Software Capability Maturity Model and ISO 9000 are used to certify organizational capability for software engineering.

4. - In the US, the Computer Science Accreditation Board (CSAB) and the Accreditation Board for Engineering and Technology (ABET) are cooperating closely and CSAB is expected to be lead society for the accreditation of university software engineering programs.

5. - The Canadian Information Processing Society has published criteria to accredit software engineering undergraduate university programs.

6. - The Texas Board of Professional Engineers has begun to license professional software engineers.

7. - The Association of Professional Engineers and Geoscientists of British Columbia (APEGBC) has begun registering software professional engineers and the Professional Engineers of Ontario (PEO) has also announced requirements for licensing.

8. - The Association for Computing Machinery (ACM) and the Computer Society of the Institute of Electrical and Electronics Engineers (IEEE) have jointly developed and adopted a Code of Ethics for software engineering professionals.

9. All of these efforts are based upon the presumption that there is a Body of Knowledge that should be mastered by practicing software engineers. This Body of Knowledge exists in the literature that has accumulated over the past thirty years. This book provides a Guide to that Body of Knowledge.

10. Purpose

11. The purpose of this Guide is to provide a consensually-validated characterization of the bounds of the software engineering discipline and to provide a topical access to the Body of Knowledge supporting that discipline. The Body of Knowledge is subdivided into ten Knowledge Areas (KA) and the descriptions of the KAs are designed to discriminate among the various important concepts, permitting readers to find their way quickly to subjects of interest. Upon finding a subject, readers are referred to key papers or book chapters selected because they succinctly present the knowledge.

12. In browsing the Guide, readers will note that the content is markedly different from Computer Science. Just as electrical engineering is based upon the science of physics, software engineering should be based upon computer science. In both cases, though, the emphasis is necessarily different. Scientists extend our knowledge of the laws of nature while engineers apply those laws of nature to build useful artifacts. Therefore, the emphasis of the Guide is placed upon the construction of useful software artifacts.

13. Readers will also notice that many important aspects of information technology, such as specific programming languages, relational databases and networks, are also not covered in the Guide. This is a consequence of an engineering-based approach. In all fields—not only computing—the designers of engineering curricula have realized that specific technologies are replaced much more rapidly than the engineering work force. An engineer must be equipped with the essential knowledge that supports the selection of the appropriate technology at the appropriate time in the appropriate circumstance. For example, software systems might be built in Fortran using functional decomposition or in C++ using object-oriented techniques. The techniques for integrating and configuring instances of those systems would be quite different. But, the principles and objectives of configuration management remain the same. The Guide therefore does not focus on the rapidly changing technologies.

14. These exclusions demonstrate that this Guide is necessarily incomplete. Practicing software engineers will need to know many things about computer science, project management and systems engineering—to name a few—that fall outside the Body of Knowledge characterized by this Guide. However, stating that this information should be known by software engineers is not the same as stating that this knowledge falls within the bounds of the software engineering discipline. Instead, it should be stated that software engineers need to know some things taken from other disciplines—and that is the approach adopted by this Guide. So, this Guide characterizes the Body of Knowledge falling within the scope of software engineering and provides references to relevant information from other disciplines.

15. The emphasis on engineering practice leads the Guide toward a strong relationship with the normative literature. Most of the computer science, information technology and software engineering literature provides information useful to software engineers, but a relatively small portion is normative. A normative document prescribes what an engineer should do in a specified situation rather than providing information that might be helpful. The normative literature is validated by consensus formed among practitioners and is concentrated in standards and related documents. From the beginning, the SWEBOK project was conceived as having a strong relationship to the normative literature of software engineering. The two major standards bodies for software engineering (IEEE Software Engineering Standards Committee and ISO/IEC JTC1/SC7) are represented in the project. Ultimately, we hope that software engineering practice standards will contain principles traceable to the SWEBOK Guide.

16. Intended Audience

17. The Guide is oriented toward a variety of audiences. It aims to serve public and private organizations in need of a consistent view of software engineering for defining education and training requirements, classifying jobs, and developing performance evaluation policies. It also addresses practicing software engineers and the officials responsible for making public policy regarding licensing and professional guidelines. In addition, professional societies and educators defining the certification rules, accreditation policies for university curricula, and guidelines for professional practice will benefit from SWEBOK, as well as the students learning the software engineering profession and educators and trainers engaged in defining curricula and course content.

18. Evolution of the Guide

19. At this time, the SWEBOK project (http://www.swebok.org) is nearing the end of the second of its three phases—called the Stoneman. An early prototype, Strawman, demonstrated how the project might be organized. Development of the Ironman version will commence after we gain insight through trial application of the Stoneman Guide.

20. Since 1993, the IEEE Computer Society and the ACM have cooperated in promoting the professionalization of software engineering through their joint Software Engineering Coordinating Committee (SWECC). The Code of Ethics mentioned previously was completed under stewardship of the SWECC primarily through volunteer efforts.

21. The SWEBOK project's scope, the variety of communities involved, and the need for broad participation suggested a need for full-time rather than volunteer management. For this purpose, the SWECC contracted the Software Engineering Management Research Laboratory at the Université du Québec à Montréal to manage the effort. It operates under SWECC supervision.

22. The project team developed two important principles for guiding the project: *transparency* and *consensus*. By transparency, we mean that the development process is itself documented, published, and publicized so that important decisions and status are visible to all concerned parties. By consensus, we mean that the only practical method for legitimizing a statement of this kind is through broad participation and agreement by all significant sectors of the relevant community. By the time the Stoneman version of the Guide is completed, literally hundreds of contributors and reviewers will have touched the product in some manner. By the time the third phase—the Ironman—is completed, the number of participants will number in the thousands and additional efforts will have been made to reach communities less likely to have participated in the current review process.

23. Like any software project, the SWEBOK project has many stakeholders—some of which are formally represented. An Industrial Advisory Board, composed of representatives from industry (Boeing, National Institute of Standards

and Technology, National Research Council of Canada, Rational Software, Raytheon Systems, and SAP Labs-Canada) and professional societies (IEEE Computer Society and ACM), provides financial support for the project. The IAB's generous support permits us to make the products of the SWEBOK project publicly available without any charge (visit http://www.swebok.org). IAB membership is supplemented with related standards bodies (IEEE Software Engineering Standards Committee and ISO/IEC JTC1/SC7) and related projects (the Computing Curricula 2001 initiative). The IAB reviews and approves the project plans, oversees consensus building and review processes, promotes the pr oject, and lends credibility to the effort. In general, it ensures the relevance of the effort to real-world needs.

24. We realize, however, that an implicit body of knowledge already exists in textbooks on software engineering. Thus, to ensure we fully take advantage of the current literature, Steve McConnell, Roger Pressman, and Ian Sommerville—the authors of the three best-selling textbooks on software engineering—have agreed to serve on a Panel of Experts, acting as a voice of experience. In addition, the extensive review process involves feedback from relevant communities. In all cases, we seek international participation to maintain a broad scope of relevance.

25. We organized the development of the Stoneman version into three public review cycles. The first review cycle focused on the soundness of the proposed breakdown of topics within each KA. Thirty-four domain experts completed this review cycle in April 1999. The reviewer comments, as well as the identities of the reviewers, are available on the project's Web site.

26. In the second review cycle completed in October 1999, a considerably larger group of professionals, organized into review viewpoints, answered a detailed questionnaire for each KA description. The viewpoints (for example, individual practitioners, educators, and makers of public policy) were formulated to ensure relevance to the Guide's various intended audiences. A discussion of the major changes that were applied after this review cycle can be found in Appendix E. Additionally, five thousand comments and their individual disposition supplied by roughly 200 reviewers and the identities of the reviewers are all publicly available and can be searched on the project's Web site.

27. The focus of the third review cycle will be on the correctness and utility of the Guide and will be conducted on the entire Guide as an integrated document rather than on each KA separately. This review cycle will be completed in the Spring of 2000 by individuals and organizations representing a cross-section of potential interest groups.

28. Limitations and Next Steps

29. Even though the current version 0.7 of the Guide has gone through an elaborate development and review process, the following limitations of this process must be recognized and stated:

30. ◆ So far, roughly two hundred and fifty software engineering professionals from 25 countries and representing various viewpoints have participated in the project. Even though this is a significant number of competent software engineering professionals, we cannot and do not claim that this sample is representative of the entire software engineering community from around the world and across all industry sectors

31. ◆ Even though complementary definitions of what constitutes "generally accepted knowledge" have been developed, the identification of which topics meet this definition within each Knowledge Area remains a matter for continued consensus formation

32. ◆ The amount of literature that has been published on software engineering is considerable and any selection of reference material remains a matter of judgment. In the case of the SWEBOK, references were selected because they are written in English, readily available, easily readable, and—, taken as a group—, provide coverage of the topics within the KA

33. ◆ important and highly relevant reference material written in other languages than English have been omitted from the selected reference material

34. ◆ Only two out of three review cycles for the Stoneman version have been completed. Please note that this is the first review cycle of the entire Guide as an integrated document

35. - The Guide has not yet been "field-tested" by its intended audience. For example, no one yet to our knowledge has attempted to define a software engineering undergraduate curricula based on this Guide nor has any industry group or organization yet written job descriptions from the Guide.

36. Additionally, one must consider that

37. - Software engineering is an emerging discipline. This is especially true if you compare it to certain more established engineering disciplines. This means notably that the boundaries between the Knowledge Areas of software engineering and between software engineering and its Related Disciplines remain a matter for continued consensus formation;

38. The contents of this Guide must therefore be viewed as an "informed and reasonable" characterization of the software engineering Body of Knowledge and as baseline document for the Ironman phase. Additionally, please note that the Guide is not attempting nor does it claim to replace or amend in any way laws, rules and procedures that have been defined by official public policy makers around the world regarding the practice and definition of engineering and software engineering in particular.

39. To address these limitations, the next (Ironman) phase will begin by monitoring and gathering feedback on actual usage of the Stoneman Guide by the various intended audiences for a period of roughly two years. Based on the gathered feedback, development of the Ironman version would be initiated in the third year and would follow a still to be determined development and review process. Those interested in performing experimental applications of the Guide are invited to contact the project team.

Alain Abran
Université du Québec à Montréal

Executive Editors of the Guide to the Software Engineering Body of Knowledge

James W. Moore
The MITRE Corporation

Pierre Bourque
Université du Québec à Montréal

Editors of the Guide to the Software Engineering Body of Knowledge

Robert Dupuis
Université du Québec à Montréal

Leonard Tripp
1999 President
IEEE Computer Society

Chair of the Joint IEEE Computer Society – ACM Software Engineering Coordinating Committee

April 2000

The SWEBOK project web site is http://www.swebok.org/

40. Acknowledgments

41. The SWEBOK editorial team gratefully acknowledges the support provided by the members of the Industrial Advisory Board. Funding for this project is provided by the Association for Computing Machinery, Boeing, the IEEE Computer Society, the National Institute of Standards and Technology, the National Research Council of Canada, Rational Software, Raytheon, and SAP Labs (Canada). The team also appreciates the important work performed by the Knowledge Area specialists. We also wish to thank the following members of the project team at the Université du Québec à Montréal: Simon Bouchard, François Cossette, Michèle Hébert, Vinh T. Ho, Julie Hudon, Louis Martin, Luis Moliníé, Évariste Valery Bevo Wandji and Sybille Wolff. Finally, the team acknowledges the indispensable contribution of the hundreds of reviewers who have participated so far. (Please note that the complete list of reviewers is available on www.swebok.org and will be included here in the final version)

TABLE OF CONTENTS

Important Notice

In this version of the Stoneman Guide, all paragraphs and entries in tables are numbered so that reviewers can identify precisely where in the Guide a recommended change is applicable. This numbering schema will be removed in the final version of the Stoneman Guide.

CHAPTER 1
INTRODUCTION TO THE GUIDE

1. In spite of the millions of software professionals worldwide and the ubiquitous presence of software in our society, software engineering has not yet reached the status of a legitimate engineering discipline and a recognized profession.

2. Since 1993, the IEEE Computer Society and the ACM have been actively promoting software engineering as a profession and a legitimate engineering discipline, notably through their Software Engineering Coordinating Committee (SWECC).

3. Achieving consensus by the profession on a core body of knowledge is a key milestone in all disciplines and has been identified by the Committee as crucial for the evolution of software engineering toward a professional status. This Guide, wr itten under the auspices of this committee, is the part of a multi-year project designed to reach this consensus.

4. What is software engineering?

5. The IEEE Computer Society defines software engineering as1:

6. > "(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

7. > (2) The study of approaches as in (1)."[2]

8. What is a recognized profession?

9. For software engineering to be known as a legitimate engineering discipline and a recognized profession, consensus on a core body of knowledge is imperative. This fact is well illustrated by Starr when he defines what can be considered a legitimate discipline and a recognized profession. In his Pulitzer-prize-winning book on the history of the medical profession in the USA, he states that:

10. > "the legitimation of professional authority involves three distinctive claims: first, that the knowledge and competence of the professional have been validated by a community of his or her peers; second, that this consensually validated knowledge rests on rational, scientific grounds; and third, that the professional's judgment and advice are oriented toward a set of substantive values, such as health. These aspects of legitimacy correspond to the kinds of attributes — collegial, cognitive and moral — usually cited in the term "profession."[3]

11. The software engineering profession is still immature

12. But what are the characteristics of a profession? Gary Ford and Norman Gibbs studied several recognized professions including medicine, law, engineering and accounting[5]. They concluded that an engineering profession is characterized by several components:

13. ◆ An initial *professional education* in a curriculum validated by society through *accreditation;*

14. ◆ Registration of fitness to practice via voluntary *certification* or mandatory *licensing*;

15. ◆ Specialized *skill development* and *continuing professional education*;

16. ◆ Communal support via a *professional society;*

17. ◆ A commitment to norms of conduct often prescribed in a *code of ethics.*

[1] Of course, there are many other definitions of software engineering. Since this effort is being conducted under a joint committee of the ACM and the IEEE Computer Society and since this definition was agreed upon by a wide consensus within the Computer Society, it was adopted at the outset of the Stoneman phase.

[2] "IEEE Standard Glossary of Software Engineering Terminology," IEEE, Piscataway, NJ std 610.12-1990, 1990.

[3] P. Starr, The Social Transformation of American Medicine: Basic Books, 1982. p. 15.

[4] P. Naur and B. Randell, "Software Engineering," presented at Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 1968.

[5] G. Ford and N. E. Gibbs, "*A Mature Profession of Software Engineering*," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical CMU/SEI -96-TR-004, January 1996.

18. This Guide directly supports the first three of these components. Articulating a Body of Knowledge is an essential step toward developing a profession because it represents a broad consensus regarding what a software engineering professional should know. Without such a consensus, no licensing examination can be validated, no curriculum can prepare an individual for an examination, and no criteria can be formulated for accrediting a curriculum. The development of the consensus is the vital prerequisite for all of these.[6]

19. What are the objectives of the project?

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

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

22. 1. Promote a consistent view of software engineering worldwide.

23. 2. Clarify the place—and set the boundary—of software engineering with respect to other disciplines such as computer science, project management, computer engineering, and mathematics.

24. 3. Characterize the contents of the software engineering discipline.

25. 4. Provide a topical access to the Software Engineering Body of Knowledge.

26. 5. Provide a foundation for curriculum development and individual certification and licensing material.

27. The first of these objectives, the consistent worldwide view of software engineering was supported by a development process that has engaged approximately 200 reviewers so far from 25 countries. (More information regarding the development process can be found in the Preface. Professional and learned societies and public agencies involved in software engineering were officially contacted, made aware of this project and invited to participate in the review process. Knowledge Area Specialists or chapter authors were recruted from North America, the Pacific Rim and Europe. Presentations on the project were made to various international venues and more are scheduled for the upcoming year.

28. The second of the objectives, the desire to set a boundary, motivates the fundamental organization of the Guide. The material that is recognized as being within software engineering is organized into the ten Knowledge Areas listed in Table 1. Each of the ten KAs is treated as a chapter in this Guide. In establishing a boundary, it is also important to identify what disciplines share a boundary and often a common intersection with software engineering. To this end, the guide also recognizes seven related disciplines, listed in Table 2. Software engineers should of course know material from these fields (and the KA descriptions may make references to the fields). It is not however an objective of the SWEBOK Guide to characterize the knowledge of the related disciplines but rather what is viewed as specific to software engineering.

29. Table 1. The SWEBOK knowledge areas (ka).

30. Software requirements

31. Software design

32. Software construction

33. Software testing

34. Software maintenance

35. Software configuration management

36. Software engineering management

37. Software engineering tools and methods

38. Software engineering process

39. Software quality

40. Table 2. Related disciplines.

41. Cognitive sciences and human factors

42. Computer engineering

43. Computer science

44. Management and management science

45. Mathematics

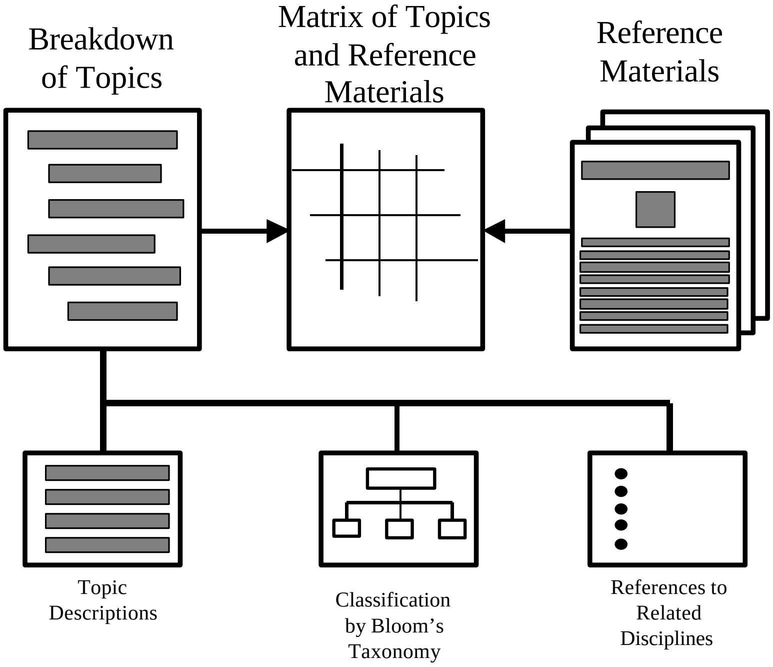
46. Project management

47. Systems engineering

[6] Regarding the final two components, it should be recognized that the SWEBOK guide is a joint project of the Software Engineering Coordinating Committee (SWECC) jointly sponsored by the ACM and the IEEE Computer Society. The SWECC has already developed and published a code of ethics.

48. Hierarchical organization

49. The organization of the Knowledge Area Descriptions or chapters, shown in Figure 1, 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 Knowledge Area Specialists regarding the contents of the Knowledge Area Descriptions can be found in Appendix A.


Breakdown of Topics
Matrix of Topics and Reference Materials
Reference Materials
Topic Descriptions
Classification by Bloom's Taxonomy
References to Related Disciplines

50. **Figure 1. The organization of a KA description.**

51. The Guide uses a hierarchical organization to decompose each KA into a set of topics with recognizable labels. A two- or 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 does 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 for the reader to successfully find reference material. After all, the Body of Knowledge is found in the reference materials, not in the Guide itself.

52. Reference materials and a matrix

53. To provide a topical access to the Knowledge—the fourth of the project's objectives—the Guide identifies reference materials for each KA including book chapters, refereed papers, or other well-recognized sources of authoritative information. Each KA description also includes a matrix that relates the reference materials to the listed topics. The total volume of cited literature is intended to be suitable for mastery through the completion of an undergraduate education plus four years of experience.

54. 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. Materials were selected, in part, because— taken as a collection—they provide coverage of the described topics.

55. Depth of Treatment

56. From the outset, the question arose as to the depth of treatment the Guide should provide. We adopted an approach that supports the fifth of the project's objectives—providing a foundation for curriculum development, certification and licensing. We applied a criterion of *generally accepted* knowledge, which we had to distinguish from advanced and research

knowledge (on the grounds of maturity) and from specialized knowledge (on the grounds of generality of application). The generally accepted knowledge applies to most projects most of the time, and widespread consensus validates its value and effectiveness.[7]

57. However, generally accepted knowledge does not imply that one should apply the designated knowledge uniformly 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 a software engineering licensing examination that graduates would take after gaining four years of work experience. Although this criterion is specific to the American style of education and does not necessarily apply to other countries, we deem it useful. However, both definitions of generally accepted knowledge should be seen as complementary.

58. Additionally, the descriptions are somewhat forward-looking—we're considering not only what is generally accepted today but also what could be generally accepted in three to five years.

59. Ratings

60. As an aid notably to curriculum developers and in support of the project's fifth objective, the Guide rates each topic with one of a set of pedagogical categories commonly attributed to Benjamin Bloom[8]. The concept is that educational objectives can be classified into six categories representing increasing depth: knowledge, comprehension, application, analysis, synthesis, and evaluation Results of this exercise for all KAs can be found in Appendix C. This Appendix must however not be viewed as a definitive classification but much more as a jumpstart document for curriculum developers.

61. KAs from related disciplines

62. A list of disciplines (Related Disciplines) that share a common boundary with software engineering can be found in Appendix B. Appendix B also identifies from as authoritative a source as possible a list of KAs of these Related Disciplines.

63. In support of the project's fifth objective, KAs of Related Disciplines that were deemed relevant to SWEBOK KAs are identified in Appendix D. Although these KAs of Related Disciplines are merely identified without additional description or references, they should aid curriculum developers.

64. Appendix D must however be viewed as a jumpstart document and as aid to curriculum developers rather than as a definitive list of relevant Knowledge Areas of Related Disciplines.

65. THE KNOWLEDGE AREAS

66. Figure 2 maps out the 10 KAs and the important topics incorporated within them. The first five KAs are presented in traditional lifecycle sequence. The subsequent are presented in alphabetical order. This is identical to the sequence in which they are presented in the Guide. Brief summaries of the KA descriptions appear next.

67. Software requirements

68. The software requirements (see Figure 2a) KA is concerned with the acquisition, analysis specification and management of software requirements. It is broken down into six subareas that correspond approximately to process tasks that are enacted iteratively rather than sequentially.

69. The requirements engineering process subarea introduces the requirements engineering process, orients the remaining five subareas, and shows how requirements engineering dovetails with the overall software engineering process. This section also deals with contractual and project organization issues.

70. The requirements elicitation subarea covers what is sometimes termed requirements capture, discovery, or acquisition. It is concerned with where requirements come from and how they can be collected by the requirements engineer. Requirements elicitation is the first stage in building an understanding of the problem the software must solve. It is fundamentally a human activity, and it identifies the stakeholders and establishes relationships between the development team and customer.

[7] Project Management Institute, A Guide to the Project Management Body of Knowledge, Upper Darby, PA, 1996, http://www.pmi.org/publictn/pmboktoc.htm/

[8] See http://www.valdosta.peachnet.edu/~whuitt/psy702/cogsys/bloom.html for Bloom's taxonomy.

71. The requirements analysis subarea is concerned with the process of analyzing requirements to detect and resolve conflicts between them, to discover the boundaries of the system and how it must interact with its environment; the requirements analysis subarea also discusses the elaboration from system requirements to software requirements. The software requirements specification subarea is concerned with the structure, quality and verification of the requirements document.

72. The requirements validation subarea is concerned with checking for omissions, conflicts, and ambiguities and with ensuring that the requirements follow prescribed quality standards. The requirements should be necessary, sufficient, and described in a way that leaves as little room as possible for misinterpretation.

73. The requirements management subarea spans the whole software life cycle. It is fundamentally about change management and maintaining the requirements in a state that accurately mirrors the software to be—or that has been—built.

74. Software design

75. Design (see Figure 2b) transforms (software) requirements—typically stated in terms relevant to the problem domain—into a description explaining how to solve the software-related aspects of the problem. It describes how the system is decomposed and organized into components, and it describes the interfaces between these components. Design also refines the description of these components into a level of detail suitable for allowing their construction.

76. Basic concepts of software design constitute the first subarea of this KA. Software architecture is the next subarea and includes topics on structures and viewpoints, architectural styles and patterns, design patterns and families of programs and frameworks. Design quality analysis and evaluation constitute the next subarea and is divided into quality attributes, quality analysis and evaluation tools, and metrics.

77. The design notations subarea discusses notations for structural and behavioral descriptions. Design strategies and methods constitute the last subarea, and it contains four main topics: general strategies, function-oriented design, object-oriented design, data-structure-centered design and other methods.

78. Software construction

79. Software construction (see Figure 2c) is a fundamental act of software engineering; programmers must construct working, meaningful software through coding, self-validation, and self-testing (unit testing). Far from being a simple mechanistic translation of good design in working software, software construction burrows deeply into difficult issues of software engineering.

80. The breakdown of topics for this KA adopts two complementary views of software construction. The first view comprises three major styles of software construction interfaces: linguistic, formal, and visual. For each style, topics are listed according to four basic principles of organization that strongly affect the way software construction is performed: reducing complexity, anticipating diversity, structuring for validation, and using external standards.

81. For example, the topics listed under anticipation of diversity for linguistic software construction interfaces are information hiding, embedded documentation, complete and sufficient method sets, object-oriented class inheritance, creation of “glue” languages for linking legacy components, table-driven software, configuration files, and self-describing software and hardware.

Software testing

82. Software testing (see Figure 2d) consists of dynamically verifying a program’s behavior on a finite set of test cases—suitably selected from the usually infinite domain of executions—against the specified expected behavior. These and other basic concepts and definitions constitute the first subarea of this KA.

83. This KA divides the test levels subarea into two orthogonal breakdowns; the first of which is organized according to the traditional phases for testing large software systems. The second breakdown concerns testing for specific conditions or properties.

84. The next subarea describes the knowledge relevant to several generally accepted test techniques. It classifies these techniques as being intuition-based, specification-based, code-based, fault-based, usage-based, or based on the nature of the application. An alternative breakdown of test techniques as being white-box or black-box is also presented. Test-related measures are dealt with in their own subarea.

85. The next subarea expands on issues relative to the management of the test process, including management concerns and test activities

86. Software maintenance

87. Software maintenance (see Figure 2e) is defined as the totality of activities required to provide cost-effective support to a software system. Activities are performed during the predelivery stage as well as the postdelivery stage. Predelivery activities include planning for postdelivery operations, supportability, and logistics determination. Postdelivery activities include software modification, training, and operating a help desk.

88. The introduction to software maintenance subarea discusses the need for software maintenance and the categories of maintenance. The maintenance activities subarea addresses the unique and supporting activities of maintenance as well as maintenance planning. As with software development, process is critical to the success and understanding of software maintenance. The next subarea discusses standard maintenance process models. Organizing the maintenance area might differ from development; the subarea on organizational aspects discusses the differences.

89. Software maintenance present unique and different technical and managerial problems for software engineering, as addressed in the problems of software maintenance subarea. Cost is always a critical topic when discussing software maintenance. The subarea on maintenance cost and maintenance cost estimation concerns life-cycle costs as well as costs for individual evolution and maintenance tasks. The maintenance measurements subarea addresses the topics of quality and metrics. The final subarea, techniques for maintenance, aggregates many subtopics that the KA description otherwise fails to address.

90. Software configuration management

91. We can define a system as a collection of components organized to accomplish a specific function and/or set of functions. A system's configuration is the function or physical characteristics of hardware, firmware, software, or a combination thereof as set forth in technical documentation and achieved in a product. Configuration management, then, is the discipline of identifying the configuration at distinct points in time to systematically control its changes and to maintain its integrity and traceability throughout the system life cycle.

92. The concepts of configuration management apply to all items requiring control, though there are differences in implementation between hardware configuration management and software configuration management. The primary activities of software configuration management are used as the framework for organizing and describing the topics of this KA (see Figure 2f). These primary activities are the management of the software configuration management process; software configuration identification, control, status accounting, and auditing; and software release management and delivery.

93. Software engineering management

94. The software engineering management (see Figure 2g) KA addresses the management of software development projects and the measurement and modeling of such projects. It consists of eight subareas, from measurement, to organizational management and coordination and then to six additional subareas organized by lifecycle phases. The measurement subarea addresses five main topics: measurement program goals, measuring software and its development, measurement selection, data collection, and metric models.

95. The organizational management and coordination subarea considers the notion of portfolio management, acquisition decisions and management, policy management, personnel management and communications. The remaining subareas are organized according to stages in the project development life cycle: initiation and scope definition, planning, enactment, review and evaluation, project close out and post-closure activities.

96. An alternative classification of these topics is also proposed in the KA description based on common themes.

97. Software engineering process

98. The software engineering process (see Figure 2h) covers the definition, implementation, measurement, management, change, and improvement of software processes. The first subarea—basic concepts and definitions—establishes the KA concepts and terminology.

99. The process infrastructure subarea is concerned with putting in place an infrastructure for software process engineering. Topics are the experience factory and software engineering

process groups. The process measurement subarea discusses quantitative techniques to diagnose software processes; to identify strengths and weaknesses. This can be performed to initiate process implementation and change, and afterwards to evaluate the consequences of process implementation and change.

100. The process definition subarea is concerned with defining processes in the form of models, plus the automated support that is available for the modeling task, and for enacting the models during the software process. The next subarea, qualitative process analysis regards qualitative techniques to analyze software processes, to identify strengths and weaknesses. This can be performed to initiate process implementation and change, and afterwards to evaluate the consequences of process implementation and change.

101. The process implementation and change subarea contains topics that regard the deployment of processes for the first time and with the change of existing processes. It focuses on organizational change. It describes the paradigms, infrastructure, and critical success factors necessary for successful process implementation and change. Within the scope of this subarea, it also presents some conceptual issues about the evaluation of process change.

102. Software engineering tools and methods

103. The Software Engineering Tools and Methods (see Figure 2i) Knowledge Area covers two topics that cut across the other KAs: software tools and development methods. Software tools are the computer-based tools intended to assist the software engineering process. Tools are often designed to support particular methods, reducing the administrative load associated with applying the method manually. Like methods, they are intended to make development more systematic, and they vary in scope from supporting individual tasks to encompassing the complete life cycle. The top-level partitioning of the software tools subarea uses the list of KAs of this Guide as its structure. The remaining categories cover infrastructure support and other miscellaneous topics.

104. Development methods impose structure on the software development and maintenance activity with the goal of making the activity systematic and ultimately more successful. Methods usually provide a notation and vocabulary, procedures for performing identifiable tasks, and guidelines for checking both the process and product. Development methods vary widely in scope, from a single life-cycle phase to the complete life cycle. The Guide divides this subarea into three nondisjointed main topics: heuristic methods dealing with informal approaches, formal methods dealing with mathematically based approaches, and prototyping methods dealing with approaches based on various forms of prototyping. The fourth main topic, miscellaneous, covers issues not previously covered

105. Software quality

106. Production of quality products is key to customer satisfaction. Software without the requisite features and degree of quality is an indicator of failed (or at least flawed) software engineering. However, even with the best of software engineering processes, requirement specifications can miss customer needs, code can fail to fulfill requirements, and subtle errors can lie undetected until they cause minor or major problems—even catastrophic failures. This KA (see Figure 2j) therefore discusses the knowledge related to software quality assurance and software verification and validation activities.

107. The goal of software engineering is a quality product, but quality itself can mean different things. The first subarea, software quality concepts, discusses measuring the value of quality, quality attributes as defined in ISO 9126, dependability, special types of systems and quality needs, and the quality attributes for the engineering process.

108. The software quality assurance process provides assurance that the software products and processes in the project life cycle conform to their specified requirements and adhere to their established plans. The software verification and validation process determines whether products of a given development or maintenance activity conform to the requirements of that activity and those imposed by previous activities, and whether the final software product (through its evolution) satisfies its intended use and user needs. These form three additional subareas.

109. The last subarea discusses measurement as applied to software quality assurance and verification and validation.

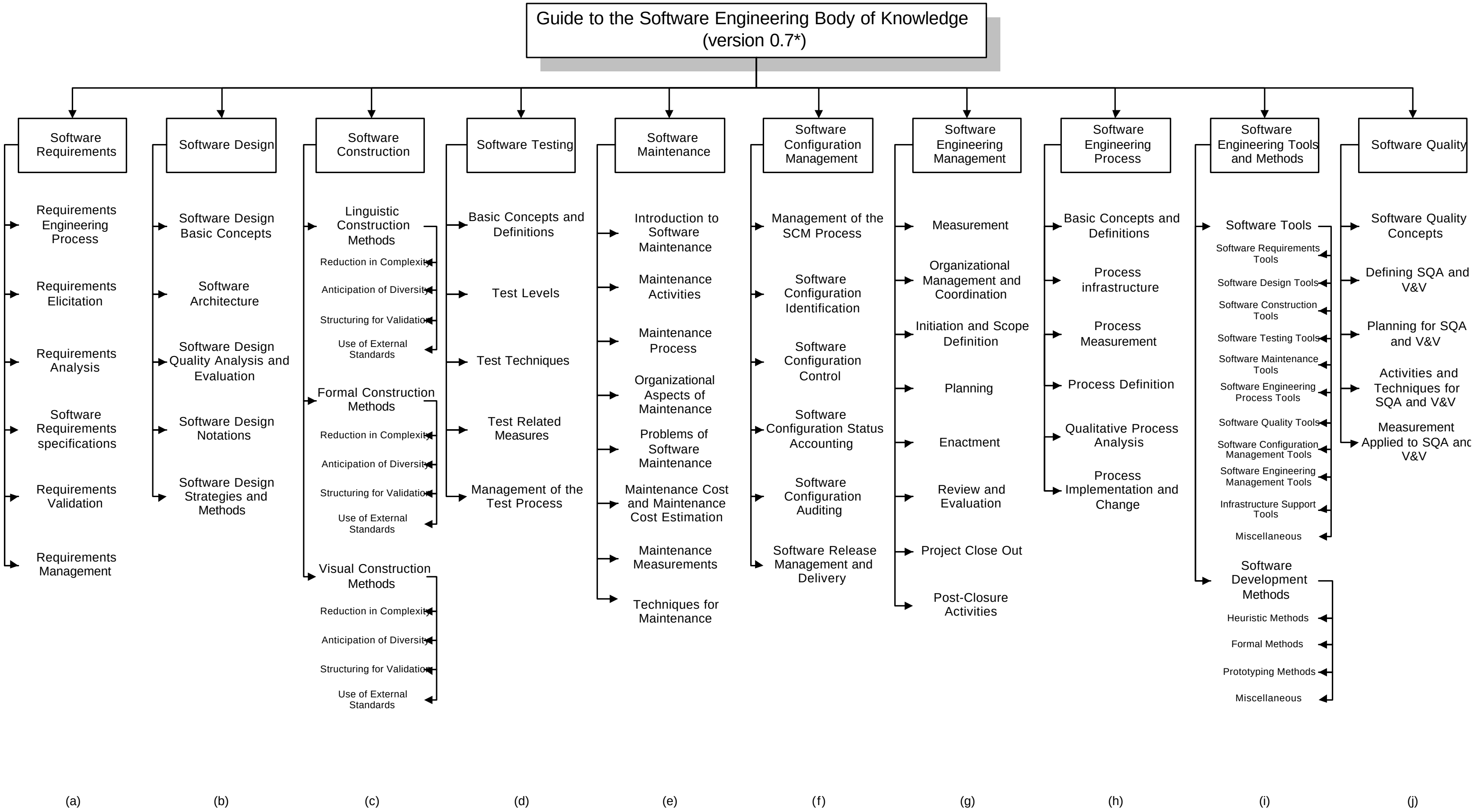

Guide to the Software Engineering Body of Knowledge (version 0.7*)
Software Requirements
Requirements Engineering Process
Requirements Elicitation
Requirements Analysis
Software Requirements specifications
Requirements Validation
Requirements Management
Software Design
Software Design Basic Concepts
Software Architecture
Software Design Quality Analysis and Evaluation
Software Design Notations
Software Design Strategies and Methods
Software Construction
Linguistic Construction Methods
Reduction in Complexity
Anticipation of Diversity
Structuring for Validation
Use of External Standards
Formal Construction Methods
Reduction in Complexity
Anticipation of Diversity
Structuring for Validation
Use of External Standards
Visual Construction Methods
Reduction in Complexity
Anticipation of Diversity
Structuring for Validation
Use of External Standards
Software Testing
Basic Concepts and Definitions
Test Levels
Test Techniques
Test Related Measures
Management of the Test Process
Software Maintenance
Introduction to Software Maintenance
Maintenance Activities
Maintenance Process
Organizational Aspects of Maintenance
Problems of Software Maintenance
Maintenance Cost and Maintenance Cost Estimation
Maintenance Measurements
Techniques for Maintenance
Software Configuration Management
Management of the SCM Process
Software Configuration Identification
Software Configuration Control
Software Configuration Status Accounting
Software Configuration Auditing
Software Release Management and Delivery
Software Engineering Management
Measurement
Organizational Management and Coordination
Initiation and Scope Definition
Planning
Enactment
Review and Evaluation
Project Close Out
Post-Closure Activities
Software Engineering Process
Basic Concepts and Definitions
Process infrastructure
Process Measurement
Process Definition
Qualitative Process Analysis
Process Implementation and Change
Software Engineering Tools and Methods
Software Tools
Software Requirements Tools
Software Design Tools
Software Construction Tools
Software Testing Tools
Software Maintenance Tools
Software Engineering Process Tools
Software Quality Tools
Software Configuration Management Tools
Software Engineering Management Tools
Infrastructure Support Tools
Miscellaneous
Software Development Methods
Heuristic Methods
Formal Methods
Prototyping Methods
Miscellaneous
Software Quality
Software Quality Concepts
Defining SQA and V&V
Planning for SQA and V&V
Activities and Techniques for SQA and V&V
Measurement Applied to SQA anc V&V
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)

Figure 2. A mapping of the Guide to the Software Engineering Body of Knowledge

* This refers to the interim draft version number of the Stoneman Guide.

CHAPTER 2
SOFTWARE REQUIREMENTS

Pete Sawyer and Gerald Kotonya
Computing Department,
Lancaster University
United Kingdom
{sawyer} {gerald}@comp.lancs.ac.uk

TABLE OF CONTENTS

1. 1. INTRODUCTION

2. This document proposes a breakdown of the SWEBOK Software Requirements Knowledge Area. The knowledge area is concerned with the acquisition, analysis specification and management of software requirements. It is widely acknowledged within the software industry that software projects are critically vulnerable when these activities are performed poorly. This has led to the widespread use of the term 'requirements engineering' to denote the systematic handling of requirements. This is the term we use in the rest of this document. Software requirements are one of the products of the requirements engineering process.

3. Software requirements express the requirements and constraints on a software product that contributes to the satisfaction of some 'need' in the real world. This need may, for example, be to solve some business problem or exploit a business opportunity offered by a new market. It is important to understand that, except where the problem is motivated by technology (such as the identification of a new market created by a new technology), the problem is an artifact of the problem domain and is generally technology-neutral. The software product alone may satisfy this need (for example, if it is a desktop application), or it may be a component (for example, a speech compression module in a mobile phone) of a software-intensive system for which satisfaction of the need is an emergent property. In fundamental terms, the way in which the requirements are handled for stand-alone products and components of software-intensive systems is the same. It is just that in systems like the mobile phone, only a subset of the requirements are allocated to software.

4. One of the main objectives of requirements engineering is to discover how to partition the system; to identify which requirements should be allocated to which components. In some systems, all the components will be implemented in software. Others will comprise a mixture of technologies. Almost all will have human users and sometimes it makes sense to consider these 'components' of the system to which requirements should be allocated (for example, to save costs or to exploit human adaptability and resourcefulness). Because of this requirements engineering is fundamentally an activity of systems engineering rather than one that is specific to software engineering. In this respect, the term 'software requirements engineering' is misleading because it implies a narrow scope concerned only with the handling of requirements that have already been acquired and allocated to software components. Since it is increasingly common for practicing software

engineers to participate in the elicitation and allocation of requirements, it is essential that the scope of the knowledge area extends to the whole of the requirements engineering process. To underscore this the prefix 'software' is omitted from requirements engineering in the remainder of this document.

5. One of the fundamental tenets of good software engineering is that there is good communication between system users and system developers. It is the requirements engineer who is the conduit for this communication. They must mediate between the domain of the system user (and other stakeholders) and the technical world of the software engineer. This requires that they possess technical skills, an ability to quickly acquire an understanding of the application domain, and the inter-personal skills to help build consensus between heterogeneous groups of stakeholders.

6. We have tried to avoid domain dependency in the document. The knowledge area document is really about identifying requirements engineering practice *and* identifying when the practice is and isn't appropriate. We recognise that desktop software products are different from reactor control systems and the document should be read in this light. Where we refer to particular tools, methods, notations, SPI models, etc. it does not imply our endorsement of them. They are merely used as examples.

7. 2. DEFINITION OF THE KNOWLEDGE AREA

8. This section provides an overview of requirements engineering in which:

9. - the notion of a 'requirement' is defined;

10. - motivations for systems are identified and their relationship to requirements is discussed;

11. - a generic process for analysis of requirements is described, followed by a discussion of why, in practice, organisations often deviate from this process; and

12. - the deliverables of the requirements engineering process and the need to manage requirements are described.

13. This overview is intended to provide a perspective or 'viewpoint' on the knowledge area that complements the one in section 4 - the knowledge area breakdown.

14. Readers who are familiar with requirements engineering concepts and terms are invited to skip to section 3.

15. 2.1 What is a requirement?

16. At its most basic, a requirement is a property that a system must exhibit in order for it to meet the system's motivating need. This may be to automate some part of a task of the people who will use the system, to support the business processes of the organisation that has commissioned the system, to control a device in which the software is to be embedded, and many more. The functioning of the users, or the business processes or the device will typically be complex. By extension, therefore, the requirements on the system will be a complex combination of requirements from different people at different levels of an organisation and from the environment in which the system must execute.

17. Requirements vary in intent and in the kinds of properties they represent. A distinction can be drawn between *product parameters* and *process parameters*. Product parameters are requirements on the system to be developed and can be further classified as:

18. - Functional requirements on the system such as formatting some text or modulating a signal. Functional requirements are sometimes known as capabilities.

19. - Non-functional requirements that act to constrain the solution. Non-functional requirements are sometimes known as constraints or quality requirements. They can be further classified according to whether they are (for example) performance requirements, maintainability requirements, safety requirements, reliability requirements, electro-magnetic compatibility requirements and many other types of requirements.

20. A process parameter is essentially a constraint on the development of the system (e.g. 'the software shall be written in Ada'). These are sometimes known as process requirements.

21. Non-functional requirements are particularly hard to handle and tend to vary from vaguely expressed goals to specific bounds on how the software must behave. Two examples of these might be: that the system must increase the call-center's throughput by 20%; and a reliability requirement that the system shall have a probability of generating a fatal error during any

hour of operation of less than 1 * 10^{-8}. The throughput requirement is at a very high level and will need to be elaborated into a number of specific functional requirements. The reliability requirement will tightly constrain the system architecture.

22. Many non-functional requirements are emergent properties. That is, requirements that can't be addressed by a single component, but which depend for their satisfaction on how all the system components inter-operate. The throughput requirement for a call-centre given above would, for example, depend upon how the telephone system, information system and the operators all interacted under actual operating conditions. Emergent properties are crucially dependent upon the system architecture.

23. An essential property of all requirements is that they should be verifiable. Unfortunately, non-functional requirements may be difficult to verify. For example, it is impossible to design a test that will demonstrate that the above reliability requirement has been satisfied. Instead, it will be necessary to construct simulations and perform statistical tests from which the system's probable reliability can be inferred. This will be very costly and illustrates the need to define non-functional requirements that are appropriate to the application domain yet not so stringent as to be beyond the bounds of the project budget.

24. Non-functional requirements should be quantified. If a non-functional requirement is only expressed qualitatively, it should be further analysed until it is possible to express it quantitatively. Non-functional requirements should never be expressed so vaguely as to be unverifiable ('the system shall be reliable', 'the user interface shall be user-friendly').

25. Stringent non-functional requirements often generate implicit process requirements. The choice of verification method is one example. Another might be the use of particularly rigorous analysis techniques (such as formal specification methods) to reduce systemic errors that can lead to inadequate reliability.

26. In a typical project there will be a large number of requirements derived from different sources and expressed at different levels of detail. In order to permit these to be referenced and managed, it is essential that each be assigned a unique identifier.

27. 2.2 System requirements and process drivers

28. The literature on requirements engineering sometimes calls system requirements user requirements. We prefer a restricted definition of the term user requirements in which they denote the requirements of the people who will be the system customers or end-users. System requirements, by contrast, are inclusive of user requirements, requirements of other stakeholders (such as regulatory authorities) and requirements that do not have an identifiable human source. Typical examples of system stakeholders include (but are not restricted to):

29. ◆ Users – the people who will operate the system. Users are often a heterogeneous group comprising people with different roles and requirements.

30. ◆ Customers – the people who have commissioned the system or who represent the system's target market.

31. ◆ Market analysts – a mass-market product will not have a commissioning customer so marketing people are often needed to establish what the market needs and to act as proxy customers.

32. ◆ Regulators – many application domains such as banking and public transport are regulated. Systems in these domains must comply with the requirements of the regulatory authorities.

33. ◆ System developers – these have a legitimate interest in profiting from developing the system. A common requirement is that costs be shared across product lines so one customer's requirements may be in conflict with the developer's wish to sell the product to other customers. For a mass market product, the developer will be the primary stakeholder since they wish to maintain the product in as large a market as possible for as long as possible.

34. In addition to these human sources of requirements, important system requirements often derive from other devices or systems in the environment which require some services of the system or act to constrain the system, or even from fundamental characteristics of the application domain. For example, a business system may be required to inter-operate with a legacy database and many military systems have to be tolerant of high levels of electro-magnetic radiation. We talk of 'eliciting' requirements but

in practice the requirements engineer discovers the requirements from a combination of human stakeholders, the system's environment, feasibility studies, market analyses, business plans, analyses of competing products and domain knowledge.

35. The elicitation and analysis of system requirements needs to be driven by the need to achieve of the overall project aims. To provide this focus, a business case should be made which clearly defines the benefits that the investment must deliver. These should act as a 'reality check' that can be applied to the system requirements to ensure that project focus does not drift. Where there is any doubt about the technical or financial viability of the project, a feasibility analysis should be conducted. This is designed to identify project risks and assess the extent to which they threaten the system's viability. Typical risks include the ability to satisfy non-functional requirements such as performance, or the availability of off-the-shelf components. In some specialised domains, it may be necessary to design simulations to generate data to enable an assessment of the project risks to be made. In domains such as public transport where safety is an issue, a hazard analysis should be conducted from which safety requirements can be identified.

36. 2.3 Requirements analysis in outline

37. Once the goals of the project have been established, the work of eliciting, analysing and validating the system requirements can commence. This is crucial to gaining a clear understanding of the problem for which the system is to provide a solution and its likely cost.

38. The requirements engineer must strive for completeness by ensuring that all the relevant sources of requirements are identified and consulted. It will be infeasible to consult everyone. There may be many of users of a large system, for example. However, representative examples of each class of system stakeholder should be identified and consulted. Although individual stakeholders will be authoritative about aspects of the system that represent their interests or expertise, the requirements engineer will be the only one with the ‘big picture’ and so the assurance of completeness rests entirely with them.

39. Elicitation of the stakeholders' requirements is rarely easy and the requirements engineer has to learn a range of techniques for helping people articulate how they do their jobs and what would help them do their jobs better. There are many social and political issues that can affect stakeholders' requirements and their ability or willingness to articulate them and it is necessary to be sensitive to them. In many cases, it is necessary to provide a contextual framework that serves to focus the consultation; to help the stakeholder identify what is possible and help the requirements engineer verify their understanding. Exposing the stakeholders to prototypes may help, and these don't necessarily have to be high fidelity. A series of rough sketches on a flip chart can sometimes serve the same purpose as a software prototype, whilst avoiding the pitfalls of distraction caused by cosmetic features of the software. Walking the stakeholder through a small number of scenarios representing sequences of events in the application domain can also help the stakeholder and requirements engineer to explore the key factors affecting the requirements.

40. Once identified, the system requirements have to be validated by the stakeholders and trade-offs negotiated before further resources are committed to the project. To enable validation, the system requirements are normally kept at a high level and expressed in terms of the application domain rather than in technical terms. Hence the system requirements for an Internet book store will be expressed in terms of books, authors, warehousing and credit card transactions, not in terms of the communication protocols, or key distribution algorithms that may form part of the solution. Too much technical detail at this stage obscures the essential characteristics of the system viewed from the perspective of its customer and users.

41. Not all of the system requirements will be satisfiable. Some may be technically infeasible, others may be too costly to implement and some will be mutually incompatible. The requirements engineer must analyse the requirements to understand their implications and how they interact. They must be prioritised and their costs estimated. The goal is to identify the scope of the system and a ‘baseline’ set of system requirements that is feasible and acceptable. This may necessitate helping stakeholders whose requirements conflict (with each other or with cost or other constraints) to negotiate acceptable trade-offs.

42. To help the analysis of the system requirements, conceptual models of the system are constructed. These aid understanding of the logical

partitioning of the system, its context in the operational environment and the data and control communications between the logical entities.

43. The system requirements must be analysed in the context of all the applicable constraints. Constraints come from many sources, such as the business environment, the customer's organisational structure and the system's operational environment. They include cost, technical (non-functional requirements), regulatory and other constraints. Hence, the requirements engineer's job is not restricted to eliciting stakeholders' requirements, but includes identifying the reasons why their requirements may be unrealisable.

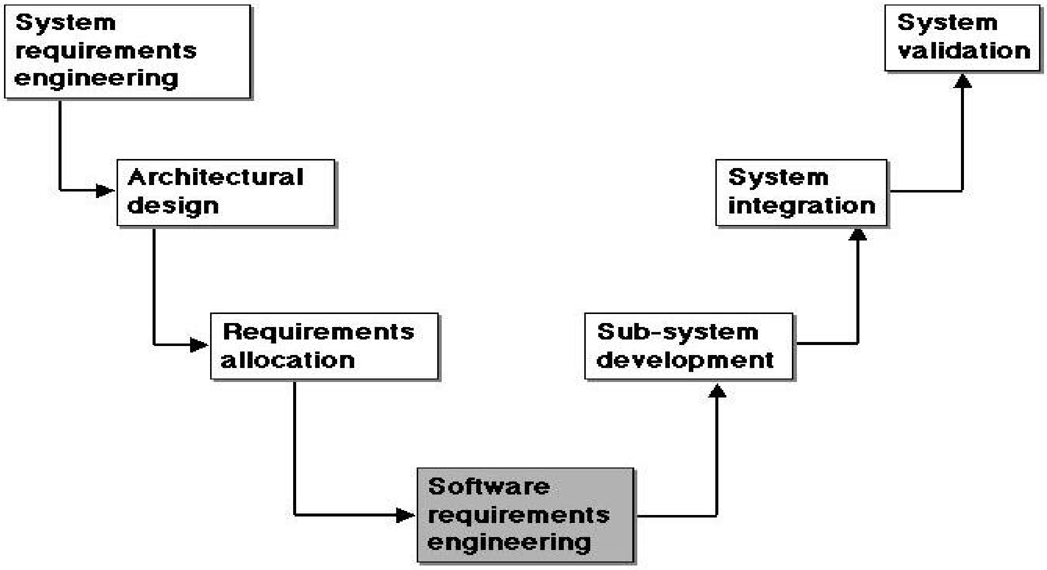
44. Unnecessary requirements should be excluded. The requirements engineer must avoid the common temptation of both users and developers to 'gold plate' systems. The essential principle is that the requirements should be *necessary and sufficient* – there should be nothing left out or anything that doesn't need to be included. The requirements engineer must also establish how implementation of the system requirements will be verified. Acceptance tests must be derived that will assure compliance with the requirements before delivery or release of the product.

45. Eventually, a complete and coherent set of system requirements will emerge as the result of the analysis process. At this point, the principal areas of functionality should be clear and the system can be partitioned into a set of subsystems or components to which responsibility for the satisfaction of subsets of the requirements are allocated. Where requirements are allocated to a software component, the requirements comprise the software requirements for that component.

46. This activity of partitioning and allocation is architectural design. Architectural design is a skill that is driven by many factors such as the recognition of reusable architectural 'patterns' or the existence of off-the shelf components. Derivation of the system architecture represents a major milestone in the project and it is crucial to get the architecture right because once defined, and resources are committed, the architecture is hard to change. In particular, the interaction of the system components crucially affects the extent to which the system will exhibit the desired emergent properties. At this point, the system requirements and system architecture are documented, reviewed and 'signed off' as the baseline for subsequent development, project planning and cost estimation.

47. Except in small-scale systems, it is generally infeasible for software developers to begin detailed design of system components from the system requirements document. The requirements allocated to components that are complex systems in themselves will need to undergo further cycles of analysis in order to add more detail, and to interpret the domain-oriented system requirements for developers who may lack sufficient knowledge of the application domain to interpret them correctly. Hence, a number of detailed technical requirements are typically derived from each high-level system requirement. It is crucial to record and maintain this derivation to enable the impact of any subsequent changes to the requirements to be assessed. This is called requirements tracing.

48. Refinement of the requirements and system architecture is where requirements engineering merges with software design. There is no clear-cut boundary but it is rare for requirements analysis to continue beyond 2 or 3 levels of architectural decomposition before responsibility is handed over to the design teams for the individual components. Figure 1 shows how software requirements engineering fits into the systems engineering process.



49. **Figure 1** The systems engineering process

	Activity	**Description**
50.	System requirements engineering	The requirements for the system as a whole are established. These will usually be expressed in a high-level fashion and written in natural language. Some detailed constraints may be included if these are critical for the success of the system.
51.	Architectural design	The system is decomposed into a set of independent sub-systems.
52.	Requirements allocation	The requirements are analysed and allocated to these sub-systems. At this stage, decisions may be made about whether requirements should be hardware or software requirements.
53.	Software requirements engineering	The high-level software requirements are decomposed into a more detailed set of requirements for the software components of the system
54.	Sub-system development	The hardware and the software subsystems are designed and implemented in parallel.
55.	*System integration*	The hardware and software subsystems are put together to complete the system
56.	*System validation*	The System is validated against its requirements.

57. 2.4 Requirements engineering in practice

58. While the general aims of the analysis process described above is fairly generic, it will not be appropriate in every case. There is often insufficient time, effort or freedom from implementation constraints to permit an orderly process such as that described in section 2.3. There is a general pressure in the software industry for ever shorter development cycles, and this is particularly pronounced in highly competitive market-driven sectors. Moreover, relatively few projects are 'green field'. Most are constrained in some way by their environment and many are upgrades to or revisions of existing systems where the system architecture is a given. In practice, therefore, it is almost always impractical to implement requirements engineering as a linear, deterministic process where system requirements are elicited from the stakeholders, baselined, allocated and handed over to the software development team. It is certainly a myth that the requirements are ever perfectly understood or perfectly specified.

59. Instead, requirements typically iterate toward a level of quality and detail that is *sufficient* to permit design and procurement decisions to me made. In some projects, this may result in the requirements being baselined before all their properties are fully understood. This is not desirable, but it is often a fact of life in the face of tight time pressure.

60. Even where more resources are allocated to requirements engineering, the level of analysis will seldom be uniformly applied. For example,

early on in the process experienced engineers are often able to identify where existing or off-the-shelf solutions can be adapted to the implementation of system components. The requirements allocated to these need not be elaborated further, while others, for which a solution is less obvious, may need to be subjected to further analysis. Critical requirements, such as those concerned with safety, must be analysed especially rigorously.

61. In almost all cases requirements understanding evolves in parallel with design and development, often leading to the revision of requirements late in the life-cycle. This is perhaps the most crucial point of understanding about requirements engineering - a significant proportion of the requirements *will* change. This is sometimes due to errors in the analysis, but it is frequently an inevitable consequence of change to the customer's business environment. It is important to recognise the inevitability of change and adopt measures to mitigate the effects of change. Change has to be managed by applying careful requirements tracing, impact analysis and version management. Hence, the requirements engineering process is not merely a front-end task to software development, but spans the whole development life-cycle. In a typical project the activities of the requirements engineer evolve over time from elicitation to change management.

62. 2.5 Products and deliverables

63. Good requirements engineering requires that the products of the process - the deliverables - are defined. The most fundamental of these in requirements engineering is the requirements document. This often comprises two separate documents:

64. ◆ A document that specifies the system requirements. This is sometimes known as the requirements definition document, user requirements document or, as defined by IEEE std 1362-1998, the concept of operations (ConOps) document. This document serves to define the high-level system requirements from the stakeholders' perspective(s). It also serves as a vehicle for validating the system requirements and, in certain types of project, may form the basis of an invitation to tender. Its typical readership includes representatives of the system stakeholders. It must be couched in terms of the customer's domain. In addition to a list of the system requirements, the requirements definition needs to include background information such as statements of the overall objectives for the system, a description of its target environment and a statement of the constraints and non-functional requirements on the system. It may include conceptual models designed to illustrate the system context, usage scenarios, the principal domain entities, and data, information and work flows.

65. ◆ A document that specifies the software requirements. This is sometimes known as the software requirements specification (SRS). The purpose and readership of the SRS is somewhat different than the requirements definition document. In crude terms, the SRS documents the detailed requirements derived from elaboration of the system requirements, and which have been allocated to software. The non-functional requirements in the requirements definition should have been elaborated and quantified. The principal readership of the SRS is technical and this can be reflected in the language and notations used to describe the requirements, and in the detail of models used to illustrate the system. For custom software, the SRS may form the basis of a contract between the developer and customer.

66. This is only a broad characterisation of the requirements document(s) that may be mandated by a particular requirements e ngineering process. The essential point is that some medium is needed for communicating the requirements engineer’s assessment of the system requirements to the stakeholders, and the software requirements to developers. The requirements document must be structured to make information easy to find and standards such as IEEE std 1362-1998 and IEEE std 830-1998 provide guidance on this. Such standards are intended to be generic and need to be tailored to the context in which they are used.

67. A requirements document should be easy to read because this affects the likelihood that the system will conform to the requirements. It should also be reasonably modular so that it is easy to maintain. The structure of the requirements document contributes to these properties but care must also be taken to describe the requirements as precisely as possible.

68. Requirements are usually written in natural language but in the SRS this may be

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

69. Even where formal notations are used, they need to be paraphrased by natural language descriptions. However, natural language has many serious shortcomings as a medium for description. Among the most serious are that it is ambiguous and hard to describe complex concepts precisely. Formal notations such as Z or CSP avoid the ambiguity problem because their syntax and semantics are formally defined. However, such notations are not expressive enough to adequately describe every system aspect. Natural language, by contrast, is extraordinarily rich and able to describe, however imperfectly, almost any concept or system property. A natural language is also likely to be the document author and readerships' only *lingua franca*. Because natural language is unavoidable, requirements engineers must be trained to use language simply, concisely and to avoid common causes of mistaken interpretation. These include:

70. ◆ long sentences with complex sub-clauses;

71. ◆ the use of terms with more than one plausible interpretation (ambiguity);

72. ◆ presenting several requirements as a single requirement;

73. ◆ inconsistency in the use of terms.

74. To counteract these problems, requirements descriptions often adopt a stylised form and use a restricted subset of a natural language. It is good practice, for example, to keep requirement descriptions short and to standardise on a small set of modal verbs to indicate relative priorities. Hence, for example, the use of 'shall' in the requirement 'The emergency breaks shall be applied to bring the train to a stop if the nose of the train passes a signal at DANGER' indicates a requirement that is mandatory.

75. Verification of the quality of the requirements documents(s) is an essential part of requirements validation. Hence, requirements validation is not merely about checking that the requirements engineer has understood the requirements. It is also about checking that the way the requirements have been documented conforms to company standards, and is understandable, consistent and complete. Formal notations offer the important advantage that they permit the last two properties to be proven. The document(s) should be subjected to review by different stakeholders including representatives of the customer and developer. Crucially, requirements documents must be placed under the same configuration management regime as the other deliverables of the development process.

76. The requirements document(s) are only the most visible manifestation of the requirements. They exclude information that is not required by the document readership. However this other information is needed in order to manage them. In particular, it is essential that requirements are traced. Tracing refers to the construction of a directed asynchronous graph (DAG) that records the derivation of requirements and provides audit trails of requirements. As a minimum, requirements need to be tracable backwards to their source (e.g. from a software requirement back to the system requirements from which it was elaborated), and forwards to the design or implementation artifacts that implement them (e.g. from a software requirement to the design document for a component that implements it). Tracing allows the requirements to be managed. In particular, it allows an impact analysis to be performed for a proposed change to one of the requirements.

77. Requirements tracing and the maintenance of requirements attributes has historically been grossly under-valued. Part of the reason for this is that it is an overhead. However, modern requirements management tools make this much less so. They typically comprise a database of requirements and a graphical user interface:

78. ◆ to store the requirement descriptions and attributes;

79. ◆ to allow the trace DAGs to be generated automatically;

80. ◆ to allow the propagation of requirements changes to be depicted graphically;

81. ◆ to generate reports on the status of requirements (such as whether they have been analysed, approved, implemented, etc.);

82. - to generate requirements documents that conform to selected standards;

83. - and to apply version management to the requirements.

84. It should be noted that not every organisation has a culture of documenting and managing requirements. It is common for dynamic start-up companies which are driven by a strong 'product vision' and limited resources to view requirements documentation as an unnecessary overhead. Inevitably, however, as these companies expand, as their customer base grows and as their product starts to evolve, they discover that they need to recover the requirements that motivated product features in order to assess the impact of proposed changes. It is true that requirements documentation and management is an overhead, but it is one that pays dividends in the longer term.

85. 3. SOFTWARE REQUIREMENTS KNOWLEDGE AREA BREAKDOWN

85. The knowledge area breakdown we have chosen is broadly compatible with the sections of ISO/IEC 12207-1995 that refer to requirements engineering activities. This standard views the software process at 3 different levels as primary, supporting and organisational life-cycle processes. In order to keep the breakdown simple, we conflate this structure into a single life-cycle process for requirements engineering. The separate topics that we identify include primary life-cycle process activities such as requirements elicitation and requirements analysis, along with requirements engineering-specific descriptions of management and, to a lesser degree, organisational processes. Hence, we identify requirements validation and requirements management as separate topics.

86. We are aware that a risk of this breakdown is that a waterfall-like process may be inferred. To guard against this, the first topic, the requirements engineering process, is designed to provide a high-level overview of requirements engineering by setting out the resources and constraints that requirements engineering operates under and which act to configure the requirements engineering process.

87. There are, of course, many other ways to structure the breakdown. For example, instead of a process-based structure, we could have used a product-based structure (system requirements, software requirements, prototypes, use-cases, etc.). We have chosen the process-based breakdown to reflect the fact that requirements engineering, if it is to be successful, must be considered as a process with complex, tightly coupled activities (both sequential and concurrent) rather than as a discrete, one-off activity at the outset of a software development project. The breakdown is compatible with that used by many of the works in the recommended reading list (Appendices B and C). See appendix A for an itemised rationale for the breakdown.

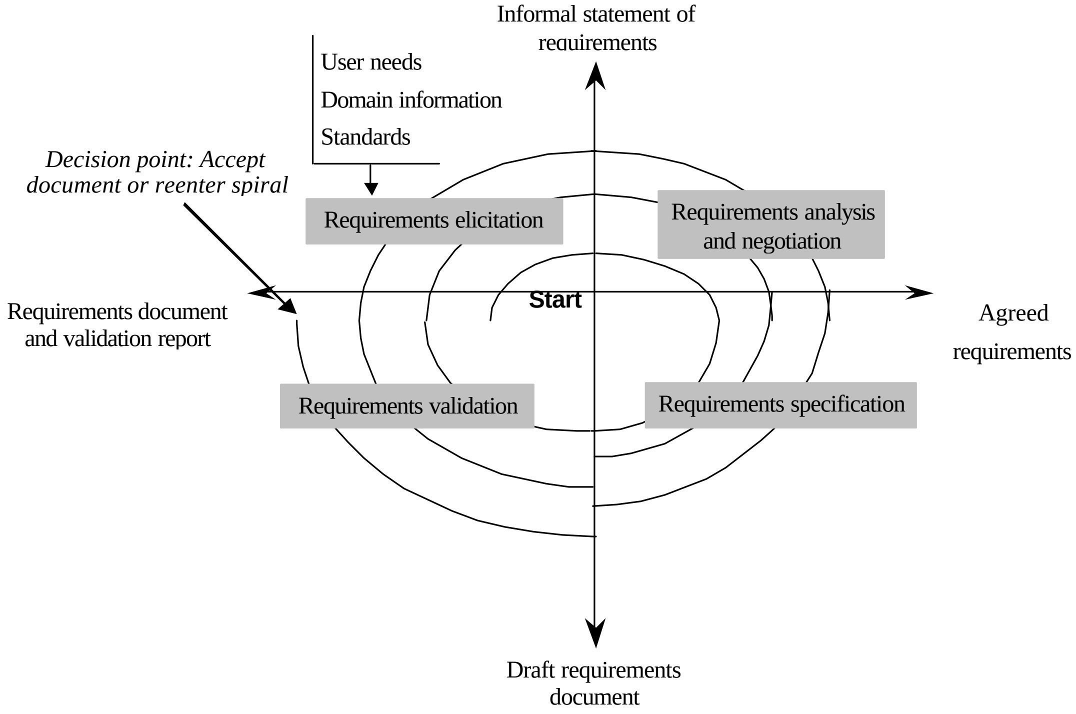
88. The breakdown comprises 6 topics as shown in table 1:

	Requirements engineering topics	**Subtopics**
89.	1. The requirement engineering process	Process models Process actors Process support and management Process quality and improvement
90.	2. Requirements elicitation	Requirements sources Elicitation techniques
91.	3. Requirement analysis	Requirements classification Conceptual modeling Architectural design and requirements allocation Requirements negotiation
92.	4. Requirements specification	The requirements definition document The software requirements specification (SRS) Document structure and standards Document quality
93.	5. Requirements validation	The conduct of requirements reviews Prototyping Model validation Acceptance tests

Requirements engineering topics	**Subtopics**
94. 6. Requirements management	Change management Requirements attributes Requirements tracing

95. **Table 1** Knowledge are breakdown

96. Figure 2 shows conceptually, how these activities comprise an iterative requirements engineering process. The different activities in requirements engineering are repeated until an acceptable requirements specification document is produced or until external factors such as schedule pressure or lack of resources cause the requirements engineering process to terminate. After a final requirements document has been produced, any further changes become part of the requirements management process.


Informal statement of requirements
User needs
Domain information
Standards
Decision point: Accept document or reenter spiral
Requirements elicitation
Requirements analysis and negotiation
Requirements document and validation report
Start
Agreed requirements
Requirements validation
Requirements specification
Draft requirements document

97. **Figure 2** A spiral model of the requirements engineering process

98. 3.1 The requirements engineering process

99. This section is concerned with introducing the requirements engineering process, orienting the remaining 5 topics and showing how requirements engineering dovetails with the overall software engineering process. This section also deals with contractual and project organisation issues. The project organisation issues in this section are described with reference to the early phase in the project concerned with bounding system requirements to ensure that an achievable project is defined. The topic is broken down into 5 subtopics.

100. 3.1.1 Process models

101. This subtopic is concerned with introducing a small number of generic process models. The purpose is to lead to an understanding that the requirements process:

102. - is not a discrete front-end activity of the software life-cycle but rather a process that is initiated at the beginning of a project but continues to operate throughout the life-cycle;

103. - the need to manage requirements under the same configuration management regime as other products of the development process;

104. - will need to be tailored to the organisation and project context.

105. In particular, the subtopic shows how the activities of elicitation, analysis, specification, validation and management are configured for different types of project and constraints. It includes an overview of activities provide input to the process such as marketing and feasibility studies.

106. 3.1.2 Process actors

107. This subtopic introduces the roles of the people who participate in the requirements engineering process. Requirements engineering is fundamentally interdisciplinary and the requirements engineer needs to mediate between the domains of the user and software engineering. There are often many people involved besides the requirements engineer, each of whom have a stake in the system. The stakeholders will vary across different projects but always includes users/operators and customer (who need not be the same). These need not be homogeneous groups because there may be many users and many customers, each with different concerns. There may also be other stakeholders who are external to the user's/customer's organisation, such as regulatory authorities, who's requirements need to be carefully analysed. The system/software developers are also stakeholders because the have a legitimate interest in profiting from the system. Again, these may be a heterogeneous group in which (for example) the system architect has different concerns from the system tester.

108. It will not be possible to perfectly satisfy the requirements of every stakeholder and the requirements engineer's job is to negotiate a compromise that is both acceptable to the principal stakeholders and within budgetary, technical, regulatory and other constraints. A prerequisite for this is that all the stakeholders are indentified, the nature of their 'stake' is analysed and their requirements are elicited.

109. 3.1.3 Process support and management

110. This subtopic introduces the project management resources required and consumed by the requirements engineering process. This topic merely sets the context for topic 4 (Initiation and scope definition) of the software management KA. It's principal purpose is to make the link from process activities identified in 3.1.1 to issues of cost, human resources, training and tools.

111. 3.1.4 Process quality and improvement

112. This subtopic is concerned with requirements engineering process quality assessment. Its purpose is to emphasize the key role requirements engineering plays in terms of the cost, timeliness and customer satisfaction of software products. It will help orient the requirements engineering process with quality standards and process improvement models for software and systems. This subtopic covers:

113. - requirements engineering coverage by process improvement standards and models;

114. - requirements engineering metrics and benchmarking;

115. - improvement planning and implementation;

	Links to common themes	
116.	Quality	The process quality and improvement subtopic is concerned with quality. It contains links to SPI standards such as the software and systems engineering CMMs, the forthcoming ISO/IEC 15504 (SPICE) and ISO 9001-3. Requirements engineering is at best peripheral to these and only work to address requirements engineering processes specifically, is the requirements engineering good practice guide (REGPG).
117.	Standards	SPI models/standards as above. In addition, the life-cycle software engineering standard ISO/IEC 12207-1995 describes software requirements engineering activities in the context of the primary, supporting and organisational life-cycle processes for software.
118.	Measurement	At the process level, requirements metrics tend to be relatively coarse-grained and concerned with (e.g.) counting numbers of requirements and numbers and effects of requirements changes. If these indicate room for improvement (as they inevitably will) it is possible to measure the extent and rigour with which requirements 'good practice' is used in a process. These measures can serve to highlight process weaknesses that should be the target improvement efforts.
119.	Tools	General project management tools. Refer to the software management KA.

120. 3.2 Requirements elicitation

121. This topic covers what is sometimes termed 'requirements capture', 'requirements discovery' or 'requirements acquisition'. It is concerned with where requirements come from and how they can be collected by the requirements engineer. Requirements elicitation is the first stage in building an understanding of the problem the software is required to solve. It is fundamentally a human activity and is where the stakeholders are identified and relationships established between the development team (usually in the form of the requirements engineer) and the customer. There are 2 main subtopics.

122. 3.2.1 Requirements sources

123. In a typical system, there will be many sources of requirements and it is essential that all potential sources are identified and evaluated for their impact on the system. This subtopic is designed to promote awareness of different requirements sources and frameworks for managing them. The main points covered are:

124. ◆ Goals. The term 'Goal' (sometimes called 'business concern' or 'critical success factor') refers to the overall, high-level objectives of the system. Goals provide the motivation for a system but are often vaguely formulated. Requirements engineers need to pay particular attention to assessing the impact and feasibility of the goals. A feasibility study is a relatively low-cost way of doing this.

125. ◆ Domain knowledge. The requirements engineer needs to acquire or to have available knowledge about the application domain. This enables them to infer tacit knowledge that the stakeholders don't articulate, inform the trade-offs that will be necessarybetween conflicting requirements and sometimes to act as a 'user' champion.

126. ◆ System stakeholders (see 3.1.2). Many systems have proven unsatisfactory because they have stressed the requirements for one group of stakeholders at the expense of others. Hence, systems are delivered that are hard to use or which subvert the cultural or political structures of the customer organisation. The requirements engineer to the need to identify, represent and manage the 'viewpoints' of many different types of stakeholder.

127. ◆ The operational environment. Requirements will be derived from the environment in which the software will execute. These may be, for example, timing constraints in a real-time system or interoperability constraints in an office environment. These must be actively sought because they can greatly affect system feasibility and cost.

128. ◆ The organizational environment. Many systems are required to support a business process and this may be conditioned by the structure, culture and internal politics of the organisation. The requirements engineer needs to be sensitive to these since, in general, new software systems should not force unplanned change to the business process.

129. 3.2.2 Elicitation techniques

130. When the requirements sources have been identified the requirements engineer can start eliciting requirements from them. This subtopic concentrates on techniques for getting human stakeholders to articulate their requirements. This is a very difficult area and the requirements engineer needs to be sensitized to the fact that (for example) users may have difficulty describing their tasks, may leave important information unstated, or may be unwilling or unable to cooperate. It is particularly important to understand that elicitation is not a passive activity and that even if cooperative and articulate stakeholders are available, the requirements engineer has to work hard to elicit the right information. A number of techniques will be covered but the principal ones are:

131. ◆ Interviews. Interviews are a 'traditional' means of eliciting requirements. It is important to understand the advantages and limitations of interviews and how they should be conducted.

132. ◆ Scenarios. Scenarios are valuable for providing context to the elicitation of users' requirements. They allow the requirements engineer to provide a framework for questions about users' tasks by permitting 'what if?' and 'how is this done?' questions to be asked. There is a link to 3.3.2. (conceptual modeling) because recent modeling notations have attempted to integrate scenario notations with object-oriented analysis techniques.

133. ◆ Prototypes. Prototypes are a valuable tool for clarifying unclear requirements. They

can act in a similar way to scenarios by providing a context within which users better understand what information they need to provide. There is a wide range of prototyping techniques, which range from paper mock-ups of screen designs to beta-test versions of software products. There is a strong overlap with the use of prototypes for requirements validation (3.5.2).

134. ◆ Facilitated meetings. The purpose of these is to try to achieve a summative effect whereby a group of people can bring more insight to their requirements than by working individually. They can brainstorm and refine ideas that may be difficult to surface using (e.g.) interviews. Another advantage is that conflicting requirements are surfaced early on in a way that lets the stakeholders recognise where there is conflict. At its best, this technique may result in a richer and more consistent set of requirements than might otherwise be achievable. However, meetings need to be handled carefully (hence the need for a facilitator) to prevent phenomena such as 'groupthink' or the requirements reflecting the concerns of a few vociferous (and perhaps senior) people to the detriment of others.

135. ◆ Observation. The importance of systems' context within the organizational environment has led to the adaptation of observational techniques for requirements elicitation whereby the requirements engineer learns about users' tasks by immersing themselves in the environment and observing how users interact with their systems and each other. These techniques are relatively new and expensive but are instructive because they illustrate that many user tasks and business processes are too subtle and complex for their actors to describe easily.

Links to common themes	
136. Quality	The quality of requirements elicitation has a direct effect on product quality. The critical issues are to recognise the relevant sources, to strive to avoid missing important requirements and to accurately report the requirements.
137. Standards	Only very general guidance is available for elicitation from current standards. These typically set out the goals of elicitation but have little to say on techniques.
138. Measurement	Very little work on metricating requirements elicitation has been carried out.
139. Tools	Elicitation is relatively poorly supported by tools. Some modern modeling tools support notations for scenarios. Several programming environments support prototyping but the applicability of these will depend on the application domain. A number of tools are becoming available that support the use of viewpoint analysis to manage requirements elicitation. These have had little impact to date.

140. 3.3 Requirements analysis

141. This subtopic is concerned with the process of analysing requirements to:

142. ◆ detect and resolve conflicts between requirements;

143. ◆ discover the bounds of the system and how it must interact with its environment;

144. ◆ elaborate system requirements to software requirements.

145. The traditional view of requirements analysis was to reduce it to conceptual modeling using one of a number of analysis methods such as SADT or OOA. While conceptual modeling is important, we include the classification of requirements to help inform trade-offs between requirements (requirements classification), and the process of establishing these trade-offs (requirements negotiation).

146. 3.3.1 Requirements classification

147. There is a strong overlap between requirements classification and requirements attributes (3.6.2). Requirements can be classified on a number of dimensions. Examples include:

148. ◆ Whether the requirement is functional or non-functional (see 2.1).

149. ◆ Whether the requirement is derived from one or more high-level requirements, an emergent property (see 2.4), or at a high level and imposed directly on the system by a stakeholder or some other source.

150. - Whether the requirement is on the product (functional or non-functional) or the process. Requirements on the process constrain, for example, the choice of contractor, the development practices to be adopted, and the standards to be adhered to.

151. - The requirement priority. In general, the higher the priority, the more essential the requirement is for meeting the overall goals of the system. Often classified on a fixed point scale such as *mandatory*, *highly desirable*, *desirable*, *optional*. In practice, priority often has to be balanced against cost of implementation.

152. - The scope of the requirement. Scope refers to the extent to which a requirement affects the system and system components. Some requirements, particularly certain non-functional ones, have a global scope in that their satisfaction cannot be allocated to a discrete component. Hence a requirement with global scope may strongly affect the system architecture and the design of many components, one with a narrow scope may offer a number of design choices with little impact on the satisfaction of other requirements.

153. - Volatility/stability. Some requirements will change during the life-cycle of the software and even during the development process itself. It is sometimes useful if some estimate of the likelihood of a requirement changing can be made. For example, in a banking application, requirements for functions to calculate and credit interest to customers' accounts are likely to be more stable than a requirement to support a particular kind of tax-free account. The former reflect a fundamental feature of the banking domain (that accounts can earn interest), while the latter may be rendered obsolete by a change to government legislation. Flagging requirements that may be volatile can help the software engineer establish a design that is more tolerant of change.

154. Other classifications may be appropriate, depending upon the development organization's normal practice and the application itself. Note that in all cases requirements must be unambiguously identified.

155. 3.3.2 Conceptual modeling

156. The development of models of the problem is fundamental to requirements analysis (see 2.4). The purpose is to aid understanding of the problem rather than to initiate design of the solution. Hence, conceptual models comprise models of entities from the problem domain configured to reflect their real-world relationships and dependencies.

157. There are several kinds of models that can be developed. These include data and control flows, state models, event traces, user interactions, object models and many others. The factors that influence the choice of model include:

158. - The nature of the problem. Some types of application demand that certain aspects be analysed particularly rigorously. For example, control flow and state models are likely to be more important for real-time systems than for an information system.

159. - The expertise of the requirements engineer. It is often more productive to adopt a modeling notation or method that the requirements engineer has experience with. However, it may be appropriate or necessary to adopt a notation that is better supported by tools, imposed as a process requirement (see 3.3.1), or simply ‘better’.

160. - The process requirements of the customer. Customers may impose a particular notation or method on the requirements engineer. This can conflict with the last factor.

161. - The availability of methods and tools. Notations or methods that are poorly supported by training and tools may not reach widespread acceptance even if they are suited to particular types of problem.

162. Note that in almost all cases, it is useful to start by building a model of the 'system boundary'. This is crucial to understanding the system's context in its operational environment and identify its interfaces to the environment.

163. The issue of modeling is tightly coupled with that of methods. For practical purposes, a method is a notation (or set of notations) supported by a process that guides the application of the notations. Methods and notations come and go in fashion. Object-oriented notations are currently in vogue (especially UML) but the issue of what is the 'best' notation is seldom clear. There is little empirical evidence to support claims for the superiority of one notation over another.

164. Formal modeling using notations based upon discrete mathematics and which are tractable to logical reasoning have made an impact in some specialized domains. These may be imposed by customers or standards or may offer compelling advantages to the analysis of certain critical functions or components.

165. This topic does not seek to 'teach' a particular modeling style or notation but rather to provide guidance on the purpose and intent of modeling.

166. 3.3.3 Architectural design and requirements allocation

167. At some point the architecture of the solution must be derived. Architectural design is the point at which requirements engineering overlaps with software or systems design and illustrates how impossible it is to cleanly decouple both tasks. In many cases, the requirements engineer acts as system architect because the process of analysing and elaborating the requirements demands that the subsystems and components that will be responsible for satisfying the requirements be identified. This is requirements allocation – the assignment of responsibility for satisfying requirements to subsystems and components.

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

169. Architectural design is closely identified with conceptual modeling and in many cases it is a natural progression to derive the solution architecture from the domain architecture. There is not always a simple one-to-one mapping from real-world domain entities to computational components, however, so architectural design is identified as a separate sub-topic. The requirements of notations and methods are broadly the same for conceptual modeling and architectural design.

170. 3.3.4 Requirements negotiation

171. Another name commonly used for this subtopic is 'conflict resolution'. It is concerned with resolving problems with requirements where conflicts occur; between two stakeholders' requiring mutually incompatible features, or between requirements and resources or between capabilities and constraints, for example. In most cases, it is unwi se for the requirements to make a unilateral decision so it is necessary to consult with the stakeholder(s) to reach a consensus on an appropriate trade-off. It is often important for contractual reasons that such decisions are traceable back to the customer. We have classified this as a requirements analysis topic because problems emerge as the result of analysis. However, a strong case can also be made for counting it as part of requirements validation.

	Links to common themes	
172.	Quality	The quality of the analysis directly affects product quality. In principle, the more rigorous the analysis, the more confidence can be attached to the software quality.
173.	Standards	Software engineering standards stress the need for analysis. Detailed guidance is provided only by de-facto modeling 'standards' (e.g. SADT or UML) which may not be completely domain independent.
174.	Measurement	Part of the purpose of analysis is to quantify required properties. This is particularly important for constraints such as reliability or safety requirements where suitable metrics need to be identified to allow the requirements to be quantified and verified.
175.	Tools	There are many tools that support conceptual modeling and a number of tools that support formal specification. There are a small number of tools that support conflict identification and requirements negotiation through the use of methods such as quality function deployment.

176. 3.4 Software requirements specification

177. This topic is concerned with the structure, quality and verification of the requirements document. This may take the form of two documents, or two parts of the same document with different readership and purposes (see 2.6): the requirements definition document and the software requirements specification. The topic stresses that documenting the requirements is the most fundamental precondition for successful requirements handling.

178. 3.4.1 The requirements definition document

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

180. 3.4.2 The software requirements specification (SRS)

181. The SRS serves an important role in software systems development. Its benefits include:

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

183. ◆ It forces a rigorous assessment of requirements before design can begin and reduces later redesign.

184. ◆ It provides a realistic basis for estimating product costs and schedules.

185. ◆ Organisations can use a SRS to develop their own validation and verification plans more productively.

186. ◆ Provides an informed a basis for transferring a software product to new users or new machines.

187. ◆ Focuses on product rather than project and therefore provides a basis for product enhancement

188. 3.4.3 Document structure and standards

189. This section describes the structure and content of a requirements document. It is also concerned with factors that influence how organisations interpret document standards to local circumstances. Several recommended guides and standards for SRS document exist. These include IEEE p123/D3 guide, IEEE Std. 1233 guide, IEEE std. 830-1998, ISO/IEC 12119-1994. IEEE std 1362-1998 concept of operations (ConOps) is a recent standard for a requirements definition document. Other guides and document template are also available.

190. 3.4.4 Document quality

191. This section is concerned with assessing the quality of an SRS. This is one area where metrics can be usefully employed in requirements engineering. There are tangible attributes that can be measured. Moreover, the quality of the requirements document can dramatically affect the quality of the product.

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

193. There is a strong overlap with 4.5.1 (the conduct of requirements reviews).

Links to common themes		
194.	Quality	The quality of the requirements documents dramatically affects the quality of the product.
195.	Standards	There are many of these. See 3.4.3.
196.	Measurement	Quality attributes of requirements documents can be identified and measured. See 3.4.4.
197.	Tools	Tool support for documentation exists in many forms from standard word processors to requirements management tools that may generate an SRS from their requirements database according to a standard template. Rudimentary quality checking tools are beginning to become commercially available, whilst more sophisticated ones are being piloted in some organisations.

198. 3.5 Requirements validation

199. It is normal for there to be one or more formally scheduled points in the requirements engineering process where the requirements are validated. The aim is to pick up any problems before resources are committed to addressing the requirements.

200. One of the key functions of requirements documents is the validation of their contents. Validation is concerned with checking the documents for omissions, conflicts and ambiguities and for ensuring that the requirements follow prescribed quality standards. The requirements should be necessary and sufficient and should be described in a way that leaves as little room as possible for misinterpretation. There are four important subtopics.

201. 3.5.1 The conduct of requirements reviews

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

203. Reviews may be constituted on completion of the system requirements definition document, the software requirements specification document, the baseline specification for a new release, etc.

204. 3.5.2 Prototyping

205. Prototyping is commonly employed for validating the requirements engineer's interpretation of the system requirements, as well as for eliciting new requirements. As with elicitation, there is a range of prototyping techniques and a number of points in the process when prototype validation may be appropriate. The advantage of prototypes is that they can make it easier to interpret the requirements engineer's assumptions and give useful feedback on why they are wrong. For example, the dynamic behaviour of a user interface can be better understood through an animated prototype than through textual description or graphical models. There are also disadvantages, however. These include the danger of users attention being distracted from the core underlying functionality by cosmetic issues or quality problems with the prototype. For this reason, several people recommend prototypes that avoid software – such as flip-chart-based mockups. Prototypes may be costly to develop although if they avoid the wastage of resources caused by trying to satisfy erroneous requirements, their cost can be more easily justified.

206. 3.5.3 Model validation

207. The quality of the models developed during analysis should be validated. For example, in object models, it is useful to perform a static analysis to verify that communication paths exist between objects that, in the stakeholders domain, exchange data. If formal specification notations are used, it is possible to use formal reasoning to prove properties of the specification (e.g. completeness).

208. *3.5.4 Acceptance tests*

209. An essential property of a system requirement is that it should be possible to verify that the finished product satisfies the requirement. Requirements that can't be verified are really just 'wishes'. An important task is therefore planning how to verify each requirement. In most cases, this is done by designing acceptance tests. One of the most important requirements quality attributes to be checked by requirements validation is the existence of adequate acceptance tests.

210. Identifying and designing acceptance test may be difficult for non-functional requirements (see 3.1). To be verifiable, they must first be analysed to the point where they can be expressed quantitatively.

	Links to common themes	
211.	Quality	Validation is all about quality - both the quality of the requirements and of the documentation.
212.	Standards	Software engineering life-cycle and documentation standards (e.g. IEEE std 830-1998) exist and are widely used in some domains to inform validation exercises.
213.	Measurement	Measurement is important for acceptance tests and definitions of how requirements are to be verified.
214.	Tools	Some limited tool support is available for model validation and theorem provers can assist developing proofs for formal models.

215. 3.6 Requirements management

216. Requirements management is an activity that should span the whole software life-cycle. It is fundamentally about change management and the maintenance of the requirements in a state that accurately mirrors the software to be, or that has been, built.

217. There are 3 subtopics concerned with requirements management.

218. *3.6.1 Change management*

219. Change management is central to the management of requirements. This subtopic is intended to describe the role of change management, the procedures that need to be in place and the analysis that should be applied to proposed changes. It will have strong links to the configuration management knowledge area.

220. *3.6.2 Requirements attributes*

221. Requirements should consist not only of a specification of what is required, but also of ancillary information that helps manage and interpret the requirements. This should include the various classification dimensions of the requirement (see 3.3.1) and the verification method or acceptance test plan. It may also include additional information such as a summary rationale for each requirement, the source of each requirement and a change history. The most fundamental requirements attribute, however, is an identifier that allows the requirements to be uniquely and unambiguously identified. A naming scheme for generating these IDs is an essential feature of a quality system for a requirements engineering process.

222. *3.6.3 Requirements tracing*

223. Requirements tracing is concerned with recovering the source of requirements and predicting the effects of requirements. Tracing is fundamental to performing impact analysis when requirements change. A requirement should be traceable backwards to the requirements and stakeholders that motivated it (from a software requirement back to the system requirement(s) that it helps satisfy, for example). Conversely, a requirement should be traceable forwards into requirements and design entities that satisfy it (for example, from a system requirement into the software requirements that have been elaborated from it and on into the code modules that implement it).

224. The requirements trace for a typical project will form a complex directed acyclic graph (DAG) of requirements. In the past, development organizations either had to write bespoke tools or manage it manually. This made tracing a short-term overhead on a project and vulnerable to expediency when resources were short. In most cases, this resulted in it either not being done at all or being performed poorly. The availability of modern requirements management tools has improved this situation and the importance of tracing (and requirements management in general) is starting to make an impact in software quality.

	Links to common themes	
225.	Quality	Requirements management is a level 2 key practice area in the software CMM and this has boosted recognition of its importance for quality.
226.	Standards	Software engineering life-cycle standards such as of ISO/IEC 12207-1995 exist and are widely used in some domains.
227.	Measurement	Mature organizations may measure the number of requirements changes and use quantitative measures of impact assessment.
228.	Tools	There are a number of requirements management tools on the market such as DOORS and RTM.

229. APPENDIX A – BREAKDOWN RATIONALE

230. Criteria are defined in Appendix A of the entire Guide.

231. Criterion (a): Number of topic breakdowns

232. One breakdown provided

233. Criterion (b): Reasonableness

234. The breakdown is reasonable in that it covers the areas discussed in most requirements engineering texts and standards. However requirements validation is normally combined with requirements verification.

235. Criterion (c): Generally accepted

236. The breakdowns are generally accepted in that they cover areas typically in texts and standards.

237. At level A.1 the breakdown is identical to that given in most requirements engineering texts, apart from process improvement. Requirements engineering process improvement is an important emerging area in requirements engineering. We believe this topic adds great value to any the discussion of the requirements engineering as its directly concerned with process quality assessment.

238. At level A.2 the breakdown is identical to that given in most requirements engineering texts. At level A.3 the breakdown is similar to that discussed in most texts. We have incorporated a reasonably detailed section on requirement characterization to take into account the most commonly discussed ways of characterizing requirements. A.4 the breakdown is similar to that discussed in most texts, apart from document quality assessment. We believe this an important aspect of the requirements specification document and deserves to be treated as a separate sub-section. In A.5 and A.6 the breakdown is similar to that discussed in most texts.

239. Criterion (d): No specific domains have been assumed

240. No specific domains have been assumed

Criterion (e): Compatible with various schools of though

241. Requirements engineering concept at the process level are general mature and stable.

242. Criterion (f): Compatible with industry, literature and standards

243. The breakdown used here has been derived from literature and relevant standards to reflect a consensus of opinion.

244. Criterion (g): As inclusive as possible

245. The inclusion of the requirements engineering process A.1 sets the context for all requirements engineering topics. This level is intended to capture the mature and stable concepts in requirements engineering. The subsequent levels all relate to level 1 but are general enough to allow more specific discussion or further breakdown.

246. Criterion (h): Themes of quality, tools, measurement and standards

247. The relationship of software requirements engineering product quality assurance, tools and standards is provided in the breakdown.

248. Criterion (i): 2 to 3 levels, 5 to 9 topics at the first level

249. The proposed breakdown satisfies this criterion.

250. Criterion (j): Topic names meaningful outside the guide

251. The topic names satisfy this criterion

252. Criterion (l): Version 0.1 of the description

253. Criterion (m): Text on the rationale underlying the proposed breakdowns

254. This document provides the rationale

255. Appendix B – Recommended References for Software Requirements

255. In Table B.1 shows the topic/reference matrix. The table is organized according to requirements engineering topics in section 3. A 'X' indicates that the topic is covered to a reasonable degree in the reference. A 'X' in appearing in main topic but not the sub-topic indicates that the main topic is reasonably covered (in general) but the sub-topic is not covered to any appreciable depth. This situation is quite common in most software engineering texts, where the subject of requirements engineering is viewed in the large context of software engineering.

TOPIC / REFERENCE	[Bry94]	[Dav93]	[Gog93]	[Kot98]	[Lou95]	[Pfl98]	[Ros98]	[Som96]	[Som97]	[Tha97]
256. **Requirements engineering process**		X		X				X	X	
Process models				X				X	X	
Process actors		X		X					X	
Process support									X	
Process improvement				X					X	
257. **Requirements elicitation**		X	X	X	X	X				
Requirements sources		X	X	X	X	X				
Elicitation techniques		X	X	X	X	X				
258. **Requirements analysis**		X		X				X		
Requirements classification		X		X				X		
Conceptual modeling		X		X				X		
Architectural design and requirements allocation		X						X		
Requirements negotiation				X						
259. **Requirement specification**	X	X		X		X	X	X		X
The requirements definition document	X	X		X			X	X		X
The software requirements specification (SRS)	X	X		X			X	X		X
Document structure	X	X		X			X			X
Document quality	X	X		X			X			
260. **Requirements validation**		X						X		X
The conduct of requirements reviews				X						X
Prototyping		X		X						X
Model validation		X		X						X
Acceptance tests		X								
261. **Requirements management**		X		X				X		
Change management				X						
Requirement attributes				X						
Requirements tracing				X						

262. Table B.1 Topics and their references

	Key	Reference
263.	[Bry94]	[Bryne 1994]
264.	[Dav93]	[Davis 1993]
265.	[Gog93]	[Goguen and Linde 1993]
266.	[Kot98]	[Kotonya and Sommerville 1998]
267.	[Lou95]	[Loucopulos and Karakostas 1995]
268.	[Pfl98]	[Pfleeger 1998]
269.	[Ros98]	[Rosenberg 1998]
270.	[Som96]	[Sommerville 1996]
271.	[Som97]	[Sommervelle and Sawyer 1997]
272.	[Tha97]	[Thayer and Dorfman 1997]

273. APPENDIX C1 – RECOMMENDED READING

274. [Bryne 1994]. Bryne, E., "IEEE Standard 830: Recommended Practice for Software Requirements Specification," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 58.

275. [Davis 1993]. Davis, A.M., Software Requirements: Objects, Functions and States. Prentice-Hall, 1993.

276. [Goguen and Linde 1993]. Goguen, J., and C. Linde, "Techniques for Requirements Elicitation," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 152-164.

277. [Kotonya and Sommerville 1998]. Kotonya, G., and I. Sommerville, Requirements Engineering: Processes and Techniques. John Wiley and Sons, 1998.

278. [Loucopulos and Karakostas 1995]. Loucopulos, P., and V. Karakostas, Systems Requirements Engineering. McGraw-Hill, pp. 1995.

279. [Pfleeger 1998]. Pfleeger, S.L., Software Engineering-Theory and Practice. Prentice-Hall, Chap. 4, 1998.

280. [Rosenberg 1998]. Rosenberg, L., T.F. Hammer and L.L. Huffman, "Requirements, testing and metrics, " 15th Annual Pacific Northwest Software Quality Conference, Utah, October 1998.

281. [Sommerville 1996]. Sommerville, I. Software Engineering (5th edition), Addison-Wesley, pp. 63-97,

282. 117-136, 1996.

283. [Sommerville 1997]. Sommerville, I., and P. Sawyer, Requirements engineering: A Good Practice Guide. John Wiley and Sons, Chap. 1-2, 1997

284. [Thayer and Dorfman 1997]. Thayer, R.H., and M. Dorfman, Software Requirements Engineering (2nd Ed). IEEE Computer Society Press, pp. 176-205, 389-404, 1997.

285. APPENDIX D – RECOMMENDED FURTHER READING

286. [Agarwal and Jones 1994]. Agarwal, N., and J. Jones, "Advancing System Engineering with a Requirements Problem. Reporting Process," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 959-964.

287. [Agusa 1984]. Agusa, K., et al., "A Verification Method for Formal Requirements Descriptions," Journal of Information Processing, 7, 4 (1984), pp. 223-229.

288. [Al-Saadoon 1995]. Al-Saadoon, O., et al., "AURA-CFG/E: An Object-Oriented Approach for Acquisition and Decomposition of DFDs from End Users," Seventh International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1995, pp. 1-7.

289. [Amber 1994]. Ambler, C., "Technology Transfer From the University Laboratory Point of View," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 146.

290. [Andews and Goeddel 1994]. Andrews, B., and W. Goeddel, "Using Rapid Prototypes for Early Requirements Validation," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 105-112.

291. [Andrews 1991]. Andrews, D., "JAD: A Crucial Dimension for Rapid Application Development," Journal of Systems Management, (March 1991), pp. 23-31.

292. [Andriole 1992]. Andriole, S., Rapid Application Prototyping, Wellesley, Massachusetts: QED, 1992.

293. [Andriole 1994]. Andriole, S., "Fast, Cheap Requirements: Prototype, or Else!" Manager Column, IEEE Software, 11, 2 (March 1994), pp. 85-87.

294. [Ardis 1997]. Ardis, M., "Formal Methods for Telecommunication System Requirements: A survey of Standardized Languages," Annals of Software Engineering, 3, N. Mead, ed., 1997.

295. [Ashworth 1988]. Ashworth, C. "Structured Systems Analysis and Design Method (SSADM)," Information and Software Technology, 30, 3 (April 1988), pp. 153-163.

296. [Astesiano and Reggio 1993]. Astesiano, and Reggio, "Specifying Reactive Systems By Abstract Events," IEEE International Workshop on Software Specification and Design, Los Alamitos, California: IEEE Computer Society Press, December 1993.

297. [Atkinson and Griswold 1996]. Atkinson, D., and W. Griswold, "The Design of Whole Program Analysis Tools," Eighteenth IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1996.

298. [Aue and Breu 1994]. Aue, A., and M. Breu, "Distributed Information Systems: An Advanced Methodology," IEEE Transactions on Software Engineering, 20, 8 (August 1994), pp. 594-605.

299. [Bally, et al. 1977]. Bally, L. et al., "A Prototype Approach to Information Systems Design and Development," Information and Management, 1, 1 (January 1977), pp. 21-26.

300. [Barroca and McDermid 1993]. Barroca, L., and J. McDermid, "Specification of Real-Time Systems -- A View-Oriented Approach," unknown, 1993.

301. [Barros 1993]. Barros, "Requirements Elicitation and Formalism Through External Design and Object-Oriented Specification," IEEE International Workshop on Software Specification and Design, Los Alamitos, California: IEEE Computer Society Press, December 1993.

302. [Belkhouche and Geraci 1994]. Belkhouche, B., and B. Geraci, "Ripple: A Formally Specified Prototyping System," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 150-153.

303. [Belscher 1995]. Belscher, R., "Evaluation of Real-Time Requirements by Simulation-Based Analysis," First IEEE International Conference on Engineering of Complex Computer Systems, Los Alamitos, California: IEEE Computer Society Press, November 1995.

304. [Ben-Abdallah, et al. 1997]. Ben-Abdallah, H., et al., "The Integrated Specification and Analysis of Functional, Temporal, and Resource Requirements," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

305. [Bentley 1992]. Bentley, R., T. Rodden, et. al., "Ethnographically informed Systems Design for Air Traffic Control," CSCW'92, Toronto, Canada, 1992.

306. [Bento 1994]. Bento, A., "Systems Analysis: A Decision Approach," Information and Management, 27, 3 (September 1994), pp. 185-193.

307. [Berdon and Davis 1995]. Berdon, J. D., and A. Davis, "Multiple-Viewpoint Based Method for Requirements Engineering," submitted to IEE Software Engineering Journal, December 1994.

308. [Berdon, et al. 1994]. Berdon, J. D., et al., "Inheritance and Adoption in Object-Oriented Systems," in preparation.

309. [Bernard and Price 1994]. Barnard, J. and A. Price, "Managing Code Inspection," IEEE Software 11, 2, 1994, pp. 59-69.

310. [Berzins and Luqi 1988]. Berzins, V., and Luqi, "Rapid Prototyping Real-Time Systems," IEEE Software, 5, 5 (September 1988), pp. 25-36.

311. [Berzins, et al. 1993]. Berzins, V., et al., "Using Transformations in Specification-Based Prototyping," IEEE Transactions on Software Engineering, 19, 5 (May 1993), pp. 436-452.

312. [Berzins, et al. 1997]. Berzins, V., et al., "A Requirements Evolution Model for Computer Aided Prototyping," Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1997, pp. 38-47.

313. [Bestavros 1991]. Bestavros, A., "Specification and Verification of Real-Time Embedded Systems Using Time-Constrained Reactive Automata," 1991 Real-Time Systems Symposium, Los Alamitos, California: IEEE Computer Society Press, 1991.

314. [Beyer and Holtzblatt 1995]. Beyer, H., and Holtzblatt, K., "Apprenticing with the Customer," Communications of the ACM, 38, 5 (May 1995), pp.45-52.

315. [Bischofberger and Pomberger 1992]. Bischofberger, W., and G. Pomberger, eds., Prototype-Oriented Software Development, Berlin, Germany: Springer Verlag, 1992.

316. [Blandford, et al. 1993]. Blandford, A., et al., "Integrating User Requirements and System Specification," in Computers, Communication and Usability: Design Issues, Research and Methods for Integrated Services, P. Byerly, et al., eds., New York, New York: Elsevier Science Publishers, 1993.

317. [Blyth, et al. 1993]. Blyth, A., et al., "ORDIT: A New Methodology to Assist in the Process of Eliciting and Modelling Organisational Requirements," Conference on Organisational Computing Systems, San Jose, California, November 1993. Also available as University of Newcastle Technical Report #coocs-93.ps, Newcastle, UK.

318. [Boehm 1976]. Boehm, B., "Software engineering," IEEE Transactions on Computers, 25, 12, 1976, pp. 1226-1241.

319. [Bolton, et al. 1986]. Bolton, D., et al., "Knowledge-Based Support for Requirements Engineering," Journal of Software Engineering and Knowledge Engineering, 2, 2 (1992), pp. 293-319.

320. [Booch 1994]. Booch, G., Object-Oriented Analysis and Design, Redwood City, California: Benjamin/Cummings, 1994.

321. [Borster and Janning 1992]. Borster, J., and T. Janning, "Traceability Between Requirements and Design: A Transformational Approach," IEEE International Conference on Computer Software and Applications, Los Alamitos, California: IEEE Computer Society Press, 1992.

322. [Bowen 1985]. Bowen, T., Specification of Software Quality Attributes, RADC Report #RADC-TR-85-37, Griffis Air Force Base, New York: Rome Air Development Center, February 1985.

323. [Brown, et al. 1994]. Brown, P., et al., "Improving the System Software Requirements Development Process," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 691-698.

324. [Brunet, et al. 1994]. Brunet, J., et al., "Applying Object Oriented Analysis on a Case Study," Information Systems, 19, 3 (1994), pp. 199-209.

325. [Bruno and Agarwal 1995]. Bruno, G., and R. Agarwal, "Validating Software Requirements Using Operational Models," Second Sympoium on Software Quality Techniques and Acquisition Criteria, Florence, Italy, May 1995.

326. [Bucci, et al. 1994]. Bucci, G., et al., "An Object-Oriented Dual Language for Specifying Reactive Systems," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 6-15.

327. [Burgess, et al. 1990]. Burgess, G., et al., "The Use of Causal Maps as a Requirements Analysis Tool: A Case Assessment of Research Propositions," in Human Factors in Information Systems Analysis and Design, A. Finkelstein, ed., Amsterdam: North-Holland-Elsevier Publ., 1990.

328. [Burns 1991]. Burns, C., "Parallel Proto: A Prototyping Tool for Analyzing & Validating Sequential and Parallel Processing Software Requirements," IEEE Second International Workshop on Rapid System Prototyping, Los Alamitos, California: IEEE Computer Society Press, June 1991, pp. 151-160.

329. [Bustard and Lundy 1995]. Bustard, D., and P. Lundy, "Enhancing Soft Systems Analysis with Formal Modeling," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

330. [Bustard and Winstanley 1994].Bustard, D., and A. Winstanley, "Making Changes to Formal Specifications: Requirements and an Example," IEEE Transactions on Software Engineering, 20, 8 (August 1994), pp. 562-568.

331. [Caspi and Halbwachs 1986]. Caspi, P., and N. Halbwachs, "A Functional Model for Describing and Reasoning About Time Behavior of Computing Systems," Acta Informatica, 22, 6 (March 1986), pp. 596-627.

332. [Castano and De Antonellis 1993]. Castano, S., and V. De Antonellis, "Reuse of Conceptual Requirements Specification," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 121-124.

333. [Castano and De Antonellis 1994]. Castano, S., and V. De Antonellis, "The F 3 Reuse Environment for Requirements Engineering," ACM Software Engineering Notes, 19, 3 (July 1994), pp. 62-65.

334. [Castano et al. 1994]. Castano, S., et al., "Reusability Based Comparison of Requirements Specification Methodologies," IFIP Conference on Methods and Associated Tools for the Information Systems Life Cycle, Amsterdam, The Netherlands: North-Holland, September 1994.

335. [Cerveny, et al. 1986]. Cerveny, R., et al., "The Application of Prototyping to Systems Development: A Rationale and Model," Journal of Management of Information Systems, 3, 2 (1986).

336. [Chambers and Manos 1992]. Chambers, G., and K. Manos, "Requirements: Their Origin, Format and Control," Second Annual International Symposium on Requirements Engineering, Seattle, Washington: National Council on Systems Engineering, July 1992.

337. [Chechik and Gannon 1994]. Chechik, M., and J. Gannon, "Automated Verification of Requirements Implementation," ACM Software Engineering Notes, Proceedings of the International Symposium on Software Testing and Analysis, Special Issue (October 1994), pp. 1-15.

338. [Checkland and Scholes 1990]. Checkland, P. and J. Scholes, Soft Systems Methodology in Action, Chichester: John Wiley and Sons, 1990.

339. [Cherry 1993]. Cherry, G., "Class/Object Stimulus-Response Machines," ACM Software Engineering Notes, 18 , 2 (April 1993), pp. 86-95.

340. [Chou and Chung 1994]. Chou, S., and S. Chung, "An OOA Model With System Function Specifications," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 16-23.

341. [Chudge and Fulton 1994]. Chudge, J., and D. Fulton, "Trust and Cooperation in System Development: Applying Responsibility Modeling to the Problem of Changing Requirements," Conference on Requirements Elicitation for Software-Based Systems, July 1994.

342. [Chung 1992]. Chung, L., J. Mylopoulos and B. Nixon, "Representing and using nonfunctional requirements -A process-oriented approach," IEEE Transactions on software engineering, 8, 6, (1992), pp. 483-497.

343. [Chung 1993]. Chung, L., "Dealing with Security Requirements During the Development of Information Systems" Fifth Conference on Advanced Information Systems Engineering, Paris, France, June 1993.

344. [Chung and Nixon 1995]. Chung, L., and B. Nixon, "Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach," Seventeenth IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

345. [Chung, et al. 1991]. Chung, L., et al., "From Information Systems Requirements to Design: A

Mapping Framework," Information Systems, 16, 4 (April 1991), pp. 429-461.

346. [Ciaccia, et al., 1995a]. Ciaccia, P., et al., "From Formal Requirements to Formal Design," Seventh International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1995, pp. 23-30.

347. [Ciancarini, et al. 1997]. Ciancarini, P., et al., "Engineering Formal Requirements: An Analysis and Testing Method for Z Documents," Annals of Software Engineering, 3, N. Mead, ed., 1997.

348. [Coleman and Baker 1997]. Coleman, D., and A. Baker, "Synthesizing Structured Analysis and Object-Based Formal Specifications," Annals of Software Engineering, 3, N. Mead, ed., 1997.

349. [Cooper and Swanson 1979]. Cooper, R., and E. Swanson, "Management Information Requirements Assessment: The State of the Art," Database, 11, 2 (February 1979), pp. 5-16.

350. [Crespo 1994]. Crespo, R., "We Need to Identify the Requirements of the Statements of Non-Functional Requirements," International Workshop on Requirements Engineering: Foundations of Software Quality, June 1994.

351. [Cucchiarelli, et al. 1994]. Cucchiarelli, A., et al., "Supporting User-Analyst Interaction in Functional Requirements Elicitation," First Asia-Pacific Software Engineering Conference, Los Alamitos, California: IEEE Computer Society, December 1994, pp. 114-123.

352. [Curran, et al. 1994]. Curran, P., et al., "BORIS-R Specification of the Requirements of a Large-Scale SoftwareIntensive System," Conference on Requirements Elicitation for Software-Based Systems, July 1994.

353. [Dankel, et al. 1992]. Dankel, D., et al., "A Model for Capturing Requirements," Fifth International Conference on Software Engineering and Its Applications, Nanterre, France: EC2, 1992.

354. [Dano et al. 1997]. Dano, B., et al., "Producing Object-Oriented Dynamic Specifications: An Approach Based on the Concept of 'Use Case'," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

355. [Dardenne et al. 1993]. Dardenne, A., et al., "Goal-Directed Requirements Acquisition," Science of Computer Programming, 20 (1993), pp. 3-50.

356. [Darimont and Souquieres 1997]. Darimont, R., and J. Souquieres, "Reusing Operational Requirements: A Process-Oriented Approach," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

357. [Dauphin, et al. 1993]. Dauphin, M., et al., "SPECS: An FDT Based Methodology for Development of Telecommunications Software," submitted to IEEE Software, May 1993.

358. [Davis 1995]. Davis, A., "Object Oriented Analysis to Object Oriented Design: An Easy Transformation?" Journal of Systems and Software, 30, 1 & 2, July-August 1995, pp. 151-159.

359. [Davis and Hsia 1994]. Davis, A., and P. Hsia, "Giving Voice to Requirements Engineering: Guest Editors' Introduction," IEEE Software, 11, 2 (March 1994), pp. 12-16.

360. [Davis and Sitaram 1993]. Davis, A., and P. Sitaram, "A Concurrent Model for Software Development," ACM Software Engineering Notes, (March 1994).

361. [Davis, et al. 1993]. Davis, A., et al., "Identifying and Measuring Quality in Software Requirements Specifications," IEEE-CS International Software Metrics Symposium, Los Alamitos, California: IEEE Computer Society Press, May 1993, pp. 141-152.

362. [De Antonellis and Vandoni 1993]. De Antonellis, V., and L. Vandoni, "Temporal Aspects in Reuse of Requirements Specifications," Fifth Conference on Advanced Information Systems Engineering, Paris, France, June 1993.

363. [De Lemos, et al. 1992a]. De Lemos, R., et al., "A Train Set as a Case Study for the Requirements Analysis of Safety-Critical Systems," The Computer Journal, 35, 1 (February 1992), pp. 30-40.

364. [DeFoe 1994]. DeFoe, J., "Requirements Engineering Technology in Industrial Education," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 145.

365. [DeFoe and McAuley 1994]. DeFoe, J., and J. McAuley, "Generating Operations Based Requirements," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 113-119.

366. [Delislie and Garlan 1990]. Delislie, N., and Garlan, D., "A Formal Specification of an Oscilloscope," IEEE Software, 7, 5 (1990) pp. 29-36.

367. [DeMarco 1997]. DeMarco, T., The Deadline, New York, New York: Dorset House, 1997.

368. [Demirors 1997]. Demirors, E., "A Blackboard Framework for Supporting Teams in Software Development," Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1997, pp. 232-239.

369. [Diepstraten 1995]. Diepstraten, M., "Command and Control System Requirements Analysis and System Requirements Specification for a Tactical System," First IEEE International Conference on Engineering of Complex Computer Systems, Los Alamitos, California: IEEE Computer Society Press, November 1995.

370. [Ding and Katayama 1993]. Ding, and Katayama, "Specifying Reactive Systems With Attributed Finite State Machines," IEEE International Workshop on Software Specification and Design, Los Alamitos, California: IEEE Computer Society Press, December 1993.

371. [Dobson and Strens 1993]. Dobson, J., and M. Strens, "How Responsibility Modelling Leads to Security Requirements," New Security Paradigms Workshop, Little Compton, Rhode Island, August 1993. Also available as University of Newcastle Technical Report #nspw.93.ps, Newcastle, UK.

372. [Dobson and Strens 1994] Dobson, J., and R. Strens, "Organizational Requirements Definition for Information Technology," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 158-165.

373. [Doke 1990]. Doke, E., "An Industry Survey of Emerging Prototyping Methodologies," Information and Management, 18, 4 (April 1990), pp. 169-176.

374. [Dowlatshashi 1994]. Dowlatshashi, J., "Rapid Prototyping Technique in Requirements Specification Phase of Software Development Life Cycle," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994.

375. [Drake and Tsai 1994]. Drake, J., and W. Tsai, "System Bounding Issues for Analysis," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 24-31.

376. [Dubois 1990]. Dubois, E., "Logical Support for Reasoning About the Specification and the Elaboration of Requirements," in Artificial Intelligence in Databases and Information Systems, R. Meersman, et al., eds., Oxford, U.K.: Elsevier Science Publishers, pp. 79-98.

377. [Dubois, et al. 1986]. Dubois, E., et al., "A Knowledge-Representation Language for Requirements Engineering," Proceedings of the IEEE, 74, 10 (October 1986), pp. 1431-1444.

378. [Duffy, et al. 1995]. Duffy, D., et al., "A Framework for Requirements Analysis Using Automated Reasoning," Seventh International Conference on Advanced Information Systems Engineering (CAiSE '95), Springer-Verlag, 1995.

379. [Easterbrook 1993]. Easterbrook, S., "Domain Modeling with Hierarchies of Alternative Viewpoints," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 65-72.

380. [Easterbrook and Nuseibeh 1995]. Easterbrook, S., and B. Nuseibeh, "Managing Inconsistencies in an Evolving Specification," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1995.

381. [Eckert 1994]. Eckert, G., "Types, Classes, and Collections in Object-Oriented Analysis," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 32-39.

382. [Edelweiss, et al. 1993]. Edelweiss, N., et al., "An Object-Oriented Temporal Model," Fifth Conference on Advanced Information Systems Engineering, Paris, France, June 1993.

383. [Edwards and White 1994]. Edwards, M., and S. White, "Requirements Capture Views," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 73-78.

384. [Edwards, et al 1995]. Edwards, M., et al., "RECAP: A Requirements Elicitation, Capture, and Analysis Process Prototype Tool for Large Complex Systems," First IEEE International Conference on Engineering of Complex Computer Systems, Los Alamitos, California: IEEE Computer Society Press, November 1995.

385. [Ege and Villalpando 1992]. Ege, A., and V. Villalpando, "SILK, An Advanced User Interface Builder and Prototyper," 2nd IEEE International Conference on Systems Integration, Los Alamitos, California: IEEE Computer Society Press, June 1992.

386. [El Emam and Madhavji 1995a]. El Emam, K., and N. Madhavji, "Requirements Engineering Practices in Information Systems Development: A Multiple Case Study," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

387. [Fairley and Thayer 1997]. Fairley, R., and R. Thayer, "The Concept of Operations: The Bridge From Operational Requirements to Technical Specifications," Annals of Software Engineering, 3, N. Mead, ed., 1997.

388. [Fairley, et al. 1994].Technical Specifications," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 40-47.

389. [Farbey 1990]. Farbey, B., "Software Quality Metrics: Considerations About Requirements and Requirements Specifications" Information and Software Technology, 32, 1 (January-February 1990), pp. 60-64; also in Software Engineering: A European Perspective, R. Thayer and A. McGettrick, eds., Los Alamitos, California: IEEE Computer Society Press, 1993, pp. 138-142.

390. [Faulk, et al. 1992]. Faulk, S., et al., "The Core Method for Real-Time Requirements," IEEE Software, 9, 5 (September 1992), pp. 22-33.

391. [Fayad, et al. 1993]. Fayad, M., et al., "Using the Shlaer-Mellor Object-Oriented Analysis Method.," IEEE Software, 10, 2 (March 1993), pp. 43-52.

392. [Feather and Fickas 1991]. Feather, M., and S. Fickas, "Coping with Requirements Freedom," International Workshop on Development of Intelligent Information Systems, Niagara-on-the-Lake, Canada, 1991, pp. 42-46.

393. [Fickas and Feather 1995]. Fickas, S., and M. Feather, "Requirements Monitoring in Dynamic Environments," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

394. [Fields, et al. 1995]. Fields, R., et al., "A Task-Centered Approach to Analyzing Human Error Tolerance Requirements," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

395. [Fiksel and Dunkle 1991]. Fiksel, J., "Principles of Requirements Management Automation," IEEE Reliability Society Leesburg Workshop on R&M CAE in Concurrent Engineering, Los Alamitos, California: IEEE Computer Society Press, October 1991.

396. [Finkelstein 1990]. Finkelstein, A., "Viewpoint Oriented Software Development," Third International Workshop on Software Engineering and Its Applications, Toulouse, France: EC2, 1990.

397. [Finkelstein 1991]. Finkelstein, A., "Tracing Back From Requirements," Colloquium on Tools and Techniques for Maintaining Traceability During Design, McGraw Hill, 1992.

398. [Finkelstein 1991b]. Finkelstein, A., "Tracing Back From Requirements," Tools and Techniques for Maintaining Traceability During Design, London, IEE Digest 1991/180, U.K.: IEE, 1991, pp. 7/1-7/2.

399. [Finkelstein and Goldsack 1991]. Finkelstein, A., and S. Goldsack, "Requirements Engineering for Real-Time Systems," IEE Software Engineering Journal, 6, 3 (March 1991), pp. 101-115.

400. [Finkelstein, et al. 1994]. Finkelstein, A., et al., "Inconsistency Handling in Multiperspective Specifications," IEEE Transactions on Software Engineering, 20, 8 (August 1994), pp. 569-578.

401. [Firesmith 1993]. Firesmith, D., Object-Oriented Requirement Analysis and Logical Design, New York, New York: Wiley, 1993.

402. [Flynn 1992]. Flynn, D., Information Systems Requirements: Determining and Analysis, London: IEE Press Digest 1991-190, December 1991.

403. [Forsgen and Rahkonen 1995]. Forsgen, P., and T. Rahkonen, "Specification of Customer and User Requirements in Industrial Control System Procurement Projects," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

404. [Fowler and Scott 1997]. Fowler, M. and Scott, K., UML Distilled: Applying the Standard Object Modelling Language. Reading, Masschusetts: Addison-Wesley, 1997.

405. [Fox and Smith 1994]. Fox, A., and H. Smith, "An Indirect Approach to Information Requirements Determination in the Development of Executive Information Systems," Conference

on Requirements Elicitation for Software-Based Systems, July 1994.

406. [Fraser, et al. 1991]. Fraser, M., et al., "Formal and Informal Requirements Specification Languages: Bridging the Gap," IEEE Transactions on Software Engineering, 17, 5 (May 1991), pp. 454-466.

407. [Freeman 1981]. Freeman, P., "Why Johnny Can't Analyze?," Systems Analysis and Design: A Foundation for the 1980's, W. Cotterman, et al., eds., Amsterdam, The Netherlands: North-Holland, 1981.

408. [Furbach 1993]. Furbach, U., "Formal Specification Methods for Reactive Systems," The Journal of Systems and Software, 21, 2 (May 1993), pp. 129-139.

409. [Gabrielian and Franklin 1988]. Gabrielian, A., and M. Franklin, "State-Based Specification of Real-Time Systems," 1988 Real-Time Systems Symposium, Los Alamitos, California: IEEE Computer Society Press, 1988.

410. [Galley and Smith 1993]. Galley, D., and J. Smith, "Overview of CORE Techniques" in Software Engineering: A European Perspective, R. Thayer and A. McGettrick, eds., Los Alamitos, California: IEEE Computer Society Press, 1993, pp. 97-104.

411. [Gamma 1995]. Gamma, E., Helm, R. et. al., Design Patterns: Elements of Reusable Object-oriented software. Reading, Massachusetts: Addison-Wesley, 1995.

412. [Garcia 1994]. Garcia, S., "ICRE Standard Panel: The SECMM Project," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 59.

413. [Ghajar-Dowlatshahi and Varnekar 1994]. Ghajar-Dowlatshahi, J., and A. Varnekar, "Rapid Prototyping in Requirements Specification Phase of Software Systems," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 135-140.

414. [Gibson 1995]. Gibson, M., "Domain Knowledge Reuse During Requirements Engineering," Seventh International Conference on Advanced Information Systems Engineering (CAiSE '95), Springer-Verlag, 1995.

415. [Goguen 1993]. Goguen, J., "Social Issues in Requirements Engineering," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 194-195.

416. [Goguen and Jirotka 1994]. Goguen, J., and M. Jirotka, eds., Requirements Engineering: Social and Technical Issues, Boston, Massachusetts: Academic Press, 1994.

417. [Goldin and Berry 1994]. Goldin, L., and D. Berry, "AbstFinder: A Prototype Abstraction Finder for Natural Language Text for Use in Requirements Elicitation: Design, Methodology and Evaluation," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 84-93.

418. [Goldman and Narayanaswamy 1992]. Goldman, N., and K. Narayanaswamy, "Software Evolution Through Iterative Prototyping," 14th IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1992.

419. [Gotel and Finkelstein 1995]. Gotel, O., and A. Finkelstein, "Contribution Structures," Second IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, March 1995.

420. [Gotel and Finkelstein 1997]. Gotel, O., and A. Finkelstein, "Extending Requirements Traceability: Lessons Learned from an Industrial Case Study," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

421. [Grady 1993]. Grady, J., "In Defense of Traditional System Requirements Analysis," Third Annual National Council on Systems Engineering International Symposium, Sunnyvale, California: NCOSE.

422. [Gray and Rao 1993]. Gray, E., and G. Rao, "Software Requirements Analysis and Specification in Europe: An Overview" in Software Engineering: A European Perspective, R. Thayer and A. McGettrick, eds., Los Alamitos, California: IEEE Computer Society Press, 1993, pp. 78-96.

423. [Gray and Thayer 1991]. Gray, E., and R. Thayer, "Requirements," in Aerospace Software Engineering: A Collection of Concepts, C. Anderson and M. Dorfman, eds., Washington, D.C.: AIAA, 1991, pp. 89-121.

424. [Gray, et al. 1988]. Gray, P., et al., "Dynamic Reconfigurability for Fast Prototyping of User Interfaces," Software Engineering Journal, 3, 6 (November 1988).

425. [Greenspan 1993]. Greenspan, S., "Panel on Recording Requirements Assumptions and Rationale," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 282.

426. [Greenspan and Feblowitz 1993]. Greenspan, S., and M. Feblowitz, "Requirements Engineering Using the SOS Paradigm," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 260-263.

427. [Greenspan, et al. 1994]. Greenspan, S., et al., " On Formal Requirements Modeling Languages: RML Revisited," Sixteenth International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, May 1994, pp. 135-147.

428. [Grosz 1992]. Grosz, G., "Building Information System Requirements Using Generic Structures," IEEE International Conference on Computer Software and Applications, Los Alamitos, California: IEEE Computer Society Press, 1992.

429. [Hadel and Lakey 1994]. Hadel, J., and P. Lakey, "A Customer-Oriented Approach to Optimizing Suballocations of System Requirements," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 29-37.

430. [Hagelstein 1988]. Hagelstein, J., "Declarative Approach to Information Systems Requirements," Knowledge Based Systems, 1, 4 (1988), pp. 211-220; ACM Software Engineering Notes, 16, 5 (December 1991), pp. 44-54.

431. [Hall 1990]. Hall, J. A., Using Z as a specification calculus for object-oriented systems. In: VDM and Z - Formal methods in Softwrae Development. Eds. D. Bjorner, C.A.R. Hoare and H. Langmaack. Heidelberg, Springer-Verlag, 1990, pp.290-318.

432. [Halligan 1993]. Halligan, R., "Requirements Metrics: The Basis of Informed Requirements Engineering Management," 1993 Complex Systems Engineering and Assessment Technology Workshop, Naval Surface Warfare Center, Dahlgren, Virginia, July 1993.

433. [Harel, et al. 1987]. Harel, D., et al., "On the Formal Semantics of Statecharts," Second IEEE Symposium on Logic in Computer Science, Los Alamitos, California: IEEE Computer Society Press, 1987.

434. [Harker 1991]. Harker, S., "Requirements Specification and the Role of Prototyping in Current Practice," in Taking Software Design Seriously, J. Karat, ed., Boston, Massachusetts: Academic Press, 1991.

435. [Harker, et al. 1993]. Harker, S., et al., "The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 266-272.

436. [Harris 1988]. Harris, D., "The Knowledge-Based Requirements Assistant," IEEE Expert, (1988).

437. [Harrison and Barnard 1993]. Harrison, M., and P. Barnard, "On Defining Requirements for Interaction," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 50-54.

438. [He and Yang 1992]. He, X., and C. Yang, "Structured Analysis Using Hierarchical Predicate Transition Nets," IEEE International Conference on Computer Software and Applications, Los Alamitos, California: IEEE Computer Society Press, 1992.

439. [Heerjee, et al. 1989]. Heerjee, K., et al., "Retrospective Software Specification" Information and Software Technology, 31, 6 (July/August 1989), pp. 324-332.

440. [Heimdahl 1996]. Heimdahl, M., "Errors Introduced during the TACS II Requirements Specification Effort: A Retrospective Case Study," Eighteenth IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1996.

441. [Heitmeyer and Labaw 1991]. Heitmeyer, C., and B. Labaw., "Requirements Specification of Hard Real-Time Systems: Experience with a Language and a Verifier," in Foundations of Real-Time Computing: Formal Specifications and Methods, van Tilborg, A, and G. Koob, eds., Norwell, Massachusetts: Kluwer Academic Publishers, 1991.

442. [Heitmeyer, et al. 1996]. Heitmeyer, C., et al., "Automated Consistency Checking Requirements Specifications," ACM Transactions on Software Engineering and Methodology, 5, 3 (July 1996), pp. 231-261.

443. [Heitz 1992]. Heitz, M., "Towards More Formal Developments Through the Integration of

BehaviorExpression Notations and Methods Within HOOD Developments," Fifth International Conference on Software Engineering and Its Applications, Nanterre, France: EC2, 1992.

444. [Henzinger, et al. 1991]. Henzinger, T., et al., "Timed Transition Systems," REX Workshop -- Real-Time Theory and Practice, 1991.

445. [Herbst 1995]. Herbst, H., "A Meta-Model for Business Rules in Systems Analysis," Seventh International Conference on Advanced Information Systems Engineering (CAiSE '95), Springer-Verlag, 1995.

446. [Hill 1991]. Hill, R., "Enabling Concurrent Engineering by Improving the Requirements Process" CALS and CE '91, Washington, D.C., June 1991.

447. [Hofmann and Holbein 1994]. Hofmann, H., and R. Holbein, "Reaching Out for Quality: Considering Security Requirements in the Design of Information Systems," International Workshop on Requirements Engineering: Foundations of Software Quality, June 1994.

448. [Holbrook 1990]. Holbrook, H., "A Scenario-Based Methodology for Conducting Requirements Elicitation," ACM Software Engineering Notes, 15, 1 (January 1990), pp. 95-104.

449. [Holtzblatt and Beyer 1995]. Holtzblatt, K., and H. Beyer, "Requirements Gathering: The Human Factor," Communications of the ACM, 38, 5 (May 1995), pp. 31-32.

450. [Houghton and Thompson 1994]. Houghton, P., and J. Thompson, "Using Enterprise Modeling to Elicit Requirements for Large Complex Systems," Conference on Requirements Elicitation for Software-Based Systems, July 1994.

451. [Hsia and Yaung 1988]. Hsia, P., and A. Yaung, "Screen-Based Scenario Generator: A Tool for Scenario-Based Prototyping," Hawaii International Conference on Systems Sciences, Los Alamitos, California: IEEE Computer Society Press, 1988, pp. 455-461.

452. [Hsia, et al. 1993]. Hsia, P., et al., "Status Report: Requirements Engineering," IEEE Software, 10, 6 (November. 1993), pp. 75-79.

453. [Hsia, et al. 1994]. Hsia, P., et al., "A Formal Approach to Scenario Analysis," IEEE Software, 11, 2 (March 1994).

454. [Hudak 1993]. Hudak, G., "Getting the Requirements for a Requirements Tool," Third Annual National Council on Systems Engineering International Symposium, Sunnyvale, California: NCOSE.

455. [Hudlicka 1996]. Hudlicka, E., "Requirements Elicitation with Indirect Knowledge Elicitation Techniques: Comparison of Three Methods," Second IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1996.

456. [Hughes, et al. 1994]. Hughes, K., et al., "A Taxonomy for Requirements Analysis Techniques," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 176-179.

457. [Hughes, et al. 1995]. Hughes, J., et al., "Presenting Ethnography in the Requirements Process," Second IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1995.

458. [Hunter and Nuseibeh 1997]. Hunter, A., and B. Nuseibeh, "Analyzing Inconsistent Specifications," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

459. [Hutt 1994]. Hutt, A., Object-Oriented Analysis and Design, New York, New York: Wiley, 1994.

460. [Ingram 1987]. Ingram, D., "Requirements Management is Key to Software Quality," CASE Outlook, 1, 5 (November 1987).

461. [Iris, et al. 1992]. Iris, J., et al., "Formalizing Requirements: The ARC2 Method," Fifth International Conference on Software Engineering and Its Applications, Nanterre, France: EC2, 1992.

462. [Ishihara, et al. 1993]. Ishihara, Y., et al., "A Translation Method From Natural Language Specifications into Formal Specifications Using Contextual Dependencies," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 232-239.

463. [Jackson 1995]. Jackson, M., Software Requirements and Specifications, Reading, Massachusetts: Addison Wesley, 1995.

464. [Jackson 1997]. Jackson, M., "The Meaning of Requirements," Annals of Software Engineering, 3, N. Mead, ed., 1997.

465. [Jackson and Zave 1993]. Jackson, M., and P. Zave, "Domain Descriptions," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 56-64.

466. [Jacobson, et al. 1993]. Jacobson, I., et al., Object-Oriented Software Engineering: A Use-Case Driven Approach, Reading, Massachusetts: Addison-Wesley, 1992.

467. [Jahanian and Mok 1986]. Jahanian, F., and A. Mok, "A Graph-Theoretic Approach for Timing Analysis in Real Time Logic," 1986 Real-Time Systems Symposium, Los Alamitos, California: IEEE Computer Society Press, 1986.

468. [Jahanian and Stuart 1988]. Jahanian, F., and D. Stuart, "A Method for Verifying Properties of Modechart Specifications," Ninth Real-Time Systems Symposium, Los Alamitos, California: IEEE Computer Society Press, 1988.

469. [Jahanian, et al. 1988]. Jahanian, F., et al., "Semantics of Modechart in Real Time Logic," 21st Hawaii International Conference on System Science, Los Alamitos, California: IEEE Computer Society Press, 1987.

470. [Jarke, et al. 1993a]. Jarke, M., et al., "Requirements Engineering: An Integrated View of Representation, Process, and Domain," 4th European Conference on Software Engineering, September 1993.

471. [Jeremaes, et al. 1986]. Jeremaes, P., et al., "A Modal (Action) Logic for Requirements Specification," in Software Engineering '86, P. Brown and D. Barnes, eds., Peter Peregrinus, 1986.

472. [Jirotka and Goguen 1994]. Jirotka, M., and J. Goguen, eds., Requirements Engineering: Social and Technical Issues, London, U.K.: Academic Press, 1994.

473. [Jirotka and Heath 1995] Jirotka, M., and C. Heath, "Ethnography by Video for Requirements Capture," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

474. [Johnson, et al. 1991]. Johnson, W., et al., "The KBSA Requirements/Specification Facet: ARIES," Sixth Annual Knowledge-Based Software Engineering Conference, Los Alamitos, California: IEEE Computer Society Press, September 1991.

475. [Johnson, et al. 1992]. Johnson, W., et al., "Representation and Presentation of Requirements Knowledge," IEEE Transactions on Software Engineering, 18, 10 (October 1992), pp. 853-69.

476. [Jones 1994]. Jones, L., "Practical Experiences in Automating Requirements in Elicitation: The Real Issues," Conference on Requirements Elicitation for Software-Based Systems, July 1994.

477. [Jones and Britton 1996]. Jones, S., and C. Britton, "Early Elicitation and Definition of Requirements for an Interactive Multimedia Information System," Second IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1996.

478. [Jones and Brooks 1994]. Jones, M., and L. Brooks, "Addressing Organizational Context in Requirements Analysis Using Cognitive Mapping," Conference on Requirements Elicitation for Software-Based Systems, July 1994.

479. [Kaindl 1993]. Kaindl, H., "Missing Link in Requirements Engineering," ACM Software Engineering Notes, 18 , 2 (April 1993), pp. 30-39.

480. [Kang and Ko 1995]. Kang, K., and G. Ko, "PARTS: A Temporal Logic-based Real-Time Software Specification and Verification Method Supporting Multiple Viewpoints," Seventeenth IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

481. [Kefer 1994]. Kefer, M., "Improving Requirements Processing, A Case Study," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 987-990.

482. [Kent, et al. 1993]. Kent, S., et al., "Formally Specifying Temporal Constraints and Error Recovery," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 208-215.

483. [Kesten and Pnueli 1991]. Kesten, Y., and A. Pnueli, "Timed and Hybrid Statecharts and Their Textual Representation," Formal Techniques in Real-Time and Fault Tolerant Systems, Berlin: Springer-Verlag, 1991, pp. 591-620.

484. [Kim and Chong 1996]. Kim, D.-H., and K. Chong, "A Method of Checking Errors and Consistency in the Process of Object-Oriented Analysis," 1996 Asia-Pacific Conference on Software Engineering, December 1996.

485. [Kirner 1993]. Kirner, T., "Analysis of Real-Time System Specification Methods," ACM Software Engineering Notes, 18 , 3 (July 1993), pp. A-50 -- A-53.

486. [Kirner and Davis 1995]. Kirner, T., and A. Davis, "Nonfunctional Requirements for Real-Time Systems," Advances in Computers, 1996.

487. [Kiskis and Shin 1994]. Kiskis, D., and K. Shin, "SWSL: A Synthetic Workload Specification Language for Real-Time Systems," IEEE Transactions on Software Engineering, 20, 10 (October 1994), pp. 798-811.

488. [Klein 1997]. Klein, M., "Handling Exceptions in Collaborative Requirements Acquisition," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

489. [Koch 1993]. Koch, G., "Process assessment: the 'BOOTSTRAP' approach," Information and Software Technology 35, 6/7 (1993), pp.387-403.

490. [Kosman 1997]. Kosman, R., "A Two-Step Methodology to Reduce Requirements Defects," Annals of Software Engineering, 3, N. Mead, ed., 1997.

491. [Kotonya and Sommerville 1992]. Kotonya, G., and I. Sommerville, "Viewpoints for Requirements Definition," Software Engineering Journal, 7, 6 (November 1992), pp. 375-387.

492. [Kovarik 1993]. Kovarik, V., "Automated Support for Managing System Requirements," Third Annual National Council on Systems Engineering International Symposium, Sunnyvale, California: NCOSE.

493. [Kramer, et al. 1988]. Kramer, J., et al., "Tool Support for Requirements Analysis," IEE Software Engineering Journal, 3 (May 1988), pp. 86-96.

494. [Kramer, et al. 1993]. Kramer, B., et al., "Computational Semantics of a Real-Time Prototyping Language," IEEE Transactions on Software Engineering, 19, 5 (May 1993), pp. 453-477.

495. [Krogstie, et al. 1995]. Krogstie, J., et al., "Towards a Deeper Understanding of Quality in Requirements Engineering," Seventh International Conference on Advanced Information Systems Engineering (CAiSE '95), Springer-Verlag, 1995.

496. [Kuwana and Herbsleb 1993]. Kuwana, E., and J. Herbsleb, "Representing Knowledge in Requirements Engineering: An Empirical Study of What Software Engineers Need to Know," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 273-276.

497. [Lalioti and Theodoulidis 1995]. Lalioti, V., and B. Theodoulidis, "Visual Scenarios for Validation of Requirements Specification," Seventh International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1995, pp. 114-116.

498. [Lam 1997]. Lam, W., "Achieving Requirements Reuse: A Domain-Specific Approach from Avionics," Journal of Systems and Software, submitted for review May 1996.

499. [Lamsweerde, et al. 1995]. Lamsweerde, A., et al., "Goal-Directed Elaboration of Requirements for a Meeting Scheduler," Second IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, March 1995.

500. [Landes 1994]. Landes, D., "Addressing Non-Functional Requirements in the Development of Knowledge-Based Systems," International Workshop on Requirements Engineering: Foundations of Software Quality, June 1994.

501. [Lano and Haughton 1994]. Lano, K., and H. Haughton, Object-Oriented Specification Case Studies, Englewood Cliffs, New Jersey: Prentice Hall, 1994.

502. [LaSala 1994]. LaSala, K., "Identifying Profiling System Requirements with Quality Function Deployment," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 249-253.

503. [Lea and Chung 1990]. Lea, R., and C. Chung, "Rapid Prototyping From Structured Analysis: Executable Specification Approach," Information and Software Technology, 32, 9 (September 1990), pp. 589-597.

504. [Lee 1993]. Lee, J., "Incrementality in Rationale Management," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, p. 283.

505. [Leffingwell 1994]. Leffingwell, D., "Object-Oriented Software Development and Medical Devices," Medical Device and Diagnostic Industry Magazine, 14, 11 (November 1994), pp. 80-88.

506. [Leffingwell 1997]. Leffingwell, D., "Calculating the Return on Investment from More Effective Requirements Management," American Programmer, 10, 4 (April 1997), pp. 13-16.

507. [Leite 1991]. Leite, J. C. P., and P. A. Freeman, "Requirements validation through viewpoint resolution," Transactions of Software Engineering, 12, 12, 1991, pp.1253-1269.

508. [Leite, et al. 1997]. Leite, J., et al., "Enhancing a Requirements Baseline with Scenarios," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

509. [Lerch, et al. 1997]. Lerch, F., et al., "Using Simulation-Based Experiments for Software Requirements Engineering," Annals of Software Engineering, 3, N. Mead, ed., 1997.

510. [Leveson and Harvey 1983]. Leveson, N. G. and Harvey, P. R., "Analyzing software safety," IEEE Transactions on Software Engineering, 9, 5, (1983), pp. 569- 579.

511. [Leveson, et al. 1994]. Leveson, N., et al., "Requirements Specification for Process-Control Systems," IEEE Transactions on Software Engineering, 20, 9 (September 1994), pp. 684-707.

512. [Li 1993]. Li, W., "Theory Revision for Requirements Capture," 4th International Joint Conference on the Theory and Practice of Software Development, Paris, France: AFCET, April 1993.

513. [Liang and Palmer 1994]. Liang, J., and J. Palmer, "A Pattern Matching and Clustering Based Approach for Supporting Requirements Transformation," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 180-183.

514. [Linton, et al. 1989]. Linton, M., et al., "Composing User Interfaces With Interviews," IEEE Computer, 22, 2 (February 1989).

515. [Liu 1993]. Liu, S., "A Formal Requirements Specification Method Based on Data Flow Analysis," The Journal of Systems and Software, 21, 2 (May 1993), pp. 141-149.

516. [Liu and Yen 1996]. Liu, F., and J. Yen, "An Analytic Framework for Specifying and Analyzing Imprecise Requirements," Eighteenth IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1996.

517. [Loucopoulos and Champion 1990]. Loucopoulos, P., and R. Champion, "Concept Definition and Analysis for Requirements Specification," IEE Software Engineering Journal, 2 (March 1990).

518. [Lubars, et al. 1992]. Lubars, M., et al, "Object Oriented Analysis for Evolving Systems," IEEE 14th International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, May 1992, pp. 173-185.

519. [Lubars, et al. 1993]. Lubars, M., et al, "A Review of the State of the Practice in Requirements Modeling," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 2-14.

520. [Lubars, et al. 1993a]. Lubars, M., et al, "Developing Initial OOA Models," 15th IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1993.

521. [Luff, et al. 1993]. Luff, P., et al, "Task and Social Interaction: The Relevance of Naturalist Analyses of Conduct for Requirements Engineering," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 187-190.

522. [Lukaszewski 1994]. Lukaszewski, M., "Applying Object-Oriented Methodology to Commercial Systems Engineering," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 867-872.

523. [Luqi 1993]. Luqi, "How to Use Prototyping for Requirements Engineering," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, p. 229.

524. [Lustman 1997]. Lustman, F., "A Formal Approach to Scenario Integration," Annals of Software Engineering, 3, N. Mead, ed., 1997.

525. [Lutz and Woodhouse 1996]. Lutz, R., and R. Woodhouse, "Contributions of SFMEA to Requirements Analysis," Second IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1996.

526. [Lutz and Woodhouse 1997]. Lutz,R., and R. Woodhouse, "Requirements Analysis Using Forward and Backward Search," Annals of Software Engineering, 3, N. Mead, ed., 1997.

527. [Macaulay 1993]. Macaulay, L., "Requirements Capture as a Cooperative Activity," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 174-181.

528. [Macaulay 1996]. Macaulay, L., Requirements Engineering, London, UK: Springer, 1996.

529. [Macfarlane and Reilly 1995]. Macfarlane, I., and I. Reilly, "Requirements Traceability in an Integrated Development Environment," Second IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, March 1995.

530. [Maiden and Rugg 1994]. Maiden, N., and G. Rugg, "Knowledge Acquisition Techniques in Requirements Engineering," Software Engineering Journal, 1995.

531. [Maiden and Rugg 1995]. Maiden, N., et al., "Computational Mechanisms for Distributed Requirements Engineering," Seventh International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1995, pp. 8-15.

532. [Maiden and Sutcliffe 1994]. Maiden, N., and A. Sutcliffe, "Requirements Critiquing Using Domain Abstractions," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 184-193.

533. [Manos 1993]. Manos, K., "Strategies for Preventing Future 'Requirements Creep,'" Third Annual National Council on Systems Engineering International Symposium, Sunnyvale, California: NCOSE, pp. 375-380.

534. [Mar 1994]. Mar, B., "Requirements for Development of Software Requirements," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 39-44.

535. [Marca and McGowan 1993]. Marca, D., and C. McGowan, "Specification Approaches Express Different World Hypotheses," IEEE International Workshop on Software Specification and Design, Los Alamitos, California: IEEE Computer Society Press, December 1993.

536. [Martin 1993]. Martin, J., "Managing Integrated Product Teams During the Requirements Definition Phase," Third International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, July 1993.

537. [Martinka 1995]. Martinka, J., "Functional Requirements for Client/Server Performance Modeling: An Implementation Using Discrete Event Simulation," First IEEE International Conference on Engineering of Complex Computer Systems, Los Alamitos, California: IEEE Computer Society Press, November 1995.

538. [Martin-Rubio and Martinez-Bejar 1997]. Martin-Rubio, F., and R. Martinez-Bejar, "A Mathematical Functions-Based for Analyzing Elicited Knowledge," Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1997, pp. 62-69.

539. [Massonet and van Lamsweerde 1997]. Massonet, P., and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

540. [Matthews and Ryan 1989]. Matthews, B., and K. Ryan, "Requirements Specification Using Conceptual Graphs," 2nd International CASE Conference, London, UK, 1989.

541. [Mays, et al. 1985]. Mays, R., et al., "PDM: A Requirements Methodology for Software System Enhancements," IBM Systems Journal, 24, 2 (February 1985), pp. 134-149.

542. [McFarland and Reilly 1995]. McFarland, I., and I. Reilly, "Requirements Traceability in an Integrated Development Environment," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

543. [McGowan 1994]. McGowan, C., "Requirements Engineering: A Different Analogy," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 147.

544. [Mead 1993]. Mead, N., "Mostly Theory at First Requirements Symposium," IEEE Software, 10, 2 (March 1993), p. 107.

545. [Mead 1994]. Mead, N., "The Role of Software Architecture in Requirements Engineering," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 242.

546. [Methlie 1980]. Methlie, L., "Systems Requirements Analysis -- Methods and Models," The Information Systems Environment, H.

Lucas, et al., eds., Amsterdam, The Netherlands: North-Holland, 1981.

547. [Meyer 1985]. Meyer, B., "On Formalism in Specification," IEEE Software, 2, 1 (January 1985), pp. 6-26.

548. [Meyers and White 1983]. Meyers, S., and S. White, Software Requirements Methodology and Tool Study: A-6E Technology Transfer, Grumman Aerospace Corporation Technical Report, prepared for Naval Weapons Center, Bethpage, New York, July 1983.

549. [Meziane and Vadera 1997]. Meziane, F., and S. Vadera, "Tools for Producing Formal Specifications: A View of Current Architectures and Future Directions," Annals of Software Engineering, 3, N. Mead, ed., 1997.

550. [Might 1993]. Might, R., "Requirements Analysis: Going From User Needs to Specific Performance Parameters," Third Annual National Council on Systems Engineering International Symposium, Sunnyvale, California: NCOSE.

551. [Miller and Sabor 1994]. Miller, L., and B. Sabor, "IEEE Draft Standard P1233: Guide for Developing System Requirements Specification," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 61.

552. [Milovanovic, et al. 1995]. Milovanovic, R., et al., "Organic Growth via Development: The Early Life Cycle," in Automated Systems Based on Human Skill, D. Brandt and T. Martin, eds., Berlin, Germany: Pergamon Press, 1995.

553. [Moreno 1997]. Moreno, A., "Object-Oriented Analysis From Textual Specifications ," Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1997, pp. 48-55.

554. [Morgan and Schahczenski 1994]. Morgan, N., and C. Schahczenski, "Transitioning to Rigorous Software Specification," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 110-117.

555. [Mostert and von Solms 1995]. Mostert, D., and S. von Solms, "A Technique to Include Computer Security, Safety, and Resilience Requirements as Part of the Requirements Specification," Journal of Systems and Software, 31, 1 (October 1995), pp. 45-53.

556. [Mullery 1979]. Mullery, G., "A method for controlled requirements specifications", 4th international Conference on Software Engineering, Munich, Germany, IEEE Computer Society Press, (1979), pp.126-135.

557. [Mumford 1985]. Mumford, E., "Defining System Requirements to Meet Business Needs: A Case Study Example," Computer Journal, 28, 2 (1985), pp. 97-104.

558. [Muntz and Lichota 1991]. Muntz, A., and R. Lichota, "A Requirements Specification Method for Real Time Systems," 1991 IEEE Real Time Systems Symposium, Los Alamitos, California: IEEE Computer Society Press, 1991, pp. 264-273.

559. [Mylopoulos, et al. 1992]. Mylopoulos, J., et al., "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," IEEE Transactions on Software Engineering, 18, 6 (June 1992), pp. 483-497.

560. [Mylopoulos, et al. 1995]. Mylopoulos, J., et al., "Multiple Viewpoints Analysis of Software Specification Process," submitted to IEEE Transactions on Software Engineering.

561. [Nakajima and Davis 1994]. Nakajima, T., and A. Davis, "Classifying Requirements for Improved SRS Reviews," International Workshop on Requirements Engineering: Foundations of Software Quality, June 1994.

562. [Nan and Buede 1994]. Nan, J., and D. Buede, "Incorporating Structured Modeling in Object-Oriented Analysis" Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994.

563. [Naumann, et al. 1980]. Naumann, J., et al., "Determining Information Requirements: A Contingency Method for Selection of Requirements Assurance Strategy," Journal of Systems and Software, 1, 4 (June 1980), pp. 273-281.

564. [Nellborn and Holm 1994]. Nellborn, C., and P. Holm, "Capturing Information Systems Requirements Through Enterprise and Speech Modeling," Sixth Conference on Advanced Information Systems Engineering, Utrecht, Netherlands, June 1994.

565. [Nishimura and Honiden 1992]. Nishimura, K., and S. Honiden, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach," submitted to IEEE Transactions on Software Engineering, December 1992.

566. [Nissen, et al. 1997]. Nissen, H., et al., "View-Directed Requirements Engineering: A

Framework and Metamodel," Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1997, pp. 366-373.

567. [Nixon 1993]. Nixon, B., "Dealing with Performance Requirements During the Development of Information Systems," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 42-49.

568. [Nuseibeh, et al. 1994]. Nuseibeh, B., et al., "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification," IEEE Transactions on Software Engineering, 20, 10 (October 1994), pp. 760-773.

569. [O'Brien 1996]. O'Brien, L., "From Use Case to Database: Implementing a Requirements Tracking System," Software Development, 4, 2 (February 1996), pp. 43-47.

570. [Ohnishi 1994]. Ohnishi, A., "A Visual Software Requirements Definition Method," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 194-201.

571. [Ohnishi 1994a]. Ohnishi, A., "Customizable Software Requirements Languages," Eighteenth International IEEE Conference on Computer Software and Applications, Los Alamitos, California: IEEE Computer Society, November 1994, pp. 5-10.

572. [Opdahl 1994]. Opdahl, A., "Requirements Engineering for Software Performance," International Workshop on Requirements Engineering: Foundations of Software Quality, June 1994.

573. [Ostroff and Wonham 1987]. Ostroff, J., and W. Wonham, "Modeling, Specifying, and Verifying Real-Time Embedded Computer Systems," 1987 Real-Time Systems Symposium, Los Alamitos, California: IEEE Computer Society Press, 1987.

574. [Ozcan and Siddiqi 1993]. Ozcan, M., and J. Siddiqi, "A Rapid Prototyping Environment for the Validation of Software Systems," 6th International Conference on Software Engineering and Its Applications, Los Alamitos, California: IEEE Computer Society Press, Paris, France: EC2, 1993.

575. [Ozcan and Siddiqi 1994]. Ozcan, M., and J. Siddiqi, "Validating and Evolving Software Requirements in a Systematic Framework," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 202-205.

576. [Palmer and Liang 1992]. Palmer, J., and J. Liang, "Indexing and Clustering of Software Requirements Specifications," Information and Decision Technologies, 18 (1992), pp. 283-299.

577. [Park, et al., 1995a]. Park, S., et al., "Text-Based Requirements Modeling Support System," Seventh International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1995, pp. 16-22.

578. [Paulk 1993]. Paulk, M.C., et. al., "Capability Maturity Model, Version 1.1," IEEE Software 10, 4, (1993), pp. 18-27.

579. [Paulk 1995]. Paulk, M. C., et. al., The Capability Maturity Model: Guidelines for Improving the Software Process. Reading, Massachusetts: Addison-Wesley, 1995.

580. [Pinheiro and Goguen]. Pinheiro,F., and J. Goguen, "An Object-Oriented Tool for Tracing Requirements," IEEE Software, 13, 2 (March 1996), pp. 52-64.

581. [Pinkerton and Fogle 1992]. Pinkerton, M., and F. Fogle, "Requirements Management/Traceability: A Case Study -- NASA's National Launch System," Second Annual International Symposium on Requirements Engineering, Seattle, Washington: National Council on Systems Engineering, July 1992.

582. [Piprani and Morris 1993]. Piprani, C., and Morris, "A Multi-Model Approach for Deriving Requirements Specifications for a Mega-Project," Fifth Conference on Advanced Information Systems Engineering, Paris, France, June 1993.

583. [Playle and Schroeder 1996]. Playle, G., and C. Schroeder, "Software Requirements Elicitation: Problems, Tools, and Techniques," Crosstalk: The Journal of Defense Software Engineering, 9, 12 (December 1996), pp. 19-24.

584. [Pliskin and Shoval 1987]. Pliskin, N., and P. Shoval, "End-User Prototyping: Sophisticated Users Supporting Systems Development," Database, 18, 4 (April 1987), pp. 7-17.

585. [Pohl, et al. 1994]. Pohl, K., et al., "Applying AI Techniques to Requirements Engineering: The NATURE Prototype," IEEE Workshop on

Research Issues in the Intersection Between Software Engineering and Artificial Intelligence, Los Alamitos, California: IEEE Computer Society Press, May 1994.

586. [Pohl, et al. 1995]. Pohl, K., et al., "Workshop Summary: First International Workshop on Requirements Engineering: Foundation of Software Quality," ACM Software Engineering Notes, 20, 1 (January 1995), pp. 39-46.

587. [Porter and Votta 1994]. Porter, A., and L. Votta, "An Experiment to Assess Different Defect Detection Methods for Software Requirements Inspections," Sixteenth International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, May 1994, pp. 103-112.

588. [Porter, et al. 1995]. Porter, A., et al., "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment," IEEE Transactions on Software Engineering, 21, 6 (June 1995), pp. 563-575.

589. [Poston 1996]. Poston, R., Automating Specification-Based Software Testing , Los Alamitos, California: IEEE Computer Society Press, 1996.

590. [Potts 1993]. Potts, C., "Panel: I Never Knew my Requirements were Object-Oriented Until I Talked to My Analyst," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, p. 226.

591. [Potts 1993a]. Potts, C., "Software Engineering Research Revisited," IEEE Software, 10, 5 (September 1993).

592. [Potts 1994]. Potts, C., "Three Architectures in Search of Requirements," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 243.

593. [Potts and Hsi 1997]. Potts, C., and I. Hsi, "Abstraction and Context in Requirements Engineering: Toward a Synthesis," Annals of Software Engineering, 3, N. Mead, ed., 1997.

594. [Potts and Newstetter 1997]. Potts, C., and W. Newstetter., "Naturalistic Inquiry and Requirements Engineering: Reconciling Their Theoretical Foundations," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

595. [Potts and Takahashi 1993]. Potts, C., and K. Takahashi, "An Active Hypertext Model for System Requirements," 7th International Workshop on Software Specification and Design, Los Alamitos, California: IEEE Computer Society Press, December 1993.

596. [Potts, et al. 1995] Potts, C., et al., "An Evaluation of Inquiry-Based Requirements Analysis for an Internet Server," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

597. [Quer and Olive 1993]. Quer, C. and A. Olive, "Object Interaction in Object-Oriented Deductive Conceptual Models," Fifth Conference on Advanced Information Systems Engineering, Paris, France, June 1993.

598. [Ramesh and Edwards 1993]. Ramesh, B., and M. Edwards, "Issues in the Development of a Requirements Traceability Model," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 256-259.

599. [Ramesh and Luqi 1993]. Ramesh, B., and Luqi, "Process Knowledge Based Rapid Prototyping for Requirements Engineering," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 248-255.

600. [Ramesh, et al. 1995]. Ramesh, B., et al., "Implementing Requirements Traceability: A Case Study," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

601. [Rauterberg and Strom 1994]. Rauterberg, M., and O. Strom, "About the Benefits of User-Oriented Requirements Engineering," International Workshop on Requirements Engineering: Foundations of Software Quality, June 1994.

602. [Ravn, et al. 1993]. Ravn, A., et al., "Specifying and Verifying Requirements of Real-Time Systems," IEEE Transactions on Software Engineering, 19, 1 (January 1993), pp. 41-55.

603. [Reed, et al. 1993]. Reed, M., et al., "Requirements Traceability," Third Annual National Council on Systems Engineering International Symposium, Sunnyvale, California: NCOSE.

604. [Regnell, et al. 1995]. Regnell, B., et al., "Improving the Use Case Driven Approach to Requirements Engineering," Second IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1995.

605. [Reizer, et al. 1994]. Reizer, N., et al., "Using Formal Methods for Requirements Specification of a Proposed POSIX Standard," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 118-125.

606. [Reubenstein 1994]. Reubenstein, H., "The Role of Software Architecture in Software Requirements Engineering," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 244.

607. [Robertson and Robertson 1994]. Robertson, J., and S. Robertson, Complete Systems Analysis, Vols. 1 and 2, Englewood Cliffs, New Jersey: Prentice Hall, 1994.

608. [Robinson 1988]. Robinson, W., "Integrating Multiple Specifications Using Domain Goals," Fifth International Workshop on Software Specification and Design, Los Alamitos, California: IEEE Computer Society Press, 1988, pp. 216-226.

609. [Robinson 1990]. Robinson, W., "A Multi-Agent View of Requirements," 12th International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1990.

610. [Robinson and Fickas 1994]. Robinson, W., and S. Fickas, "Supporting Multi-Perspective Requirements Engineering," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 206-215.

611. [Rolland 1994]. Rolland, C., "Modeling and Evolution of Artifacts," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 216-219.

612. [Rolland 1994a]. Rolland, C., "A Contextual Approach for the Requirements Engineering Process," 6th International Conference on Software Engineering and Knowledge Engineering, June 1994.

613. [Rolland and Prakash 1994]. Rolland, C., and N. Prakash, "Guiding the Requirements Engineering Process," First Asia-Pacific Conference on Software Engineering, December 1994.

614. [Rolland and Proix 1992]. Rolland, C., and C. Proix, "A Natural Language Approach to Requirements Engineering," 4th International CAiSE Conference, Manchester, UK, 1992.

615. [Ross 1985]. Ross, D., "Applications and extensions of SADT," IEEE Computer, 18, 4, (1985) pp.25-34.

616. [Rundlet and Miller 1994]. Rundlet, N., and W. Miller, "Requirements Management: DOORS to the Battlefield of the Future," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 65-72.

617. [Ryan 1993]. Ryan, K., "The Role of Natural Language in Requirements Engineering," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 240-242.

618. [Ryan and Matthews 1993]. Ryan, M., and B. Matthews, "Matching Conceptual Graphs to Requirements Re-use," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 112-120.

619. [Ryan and O'Beirne 1994]. Ryan, K., and A. O'Beirne, "An Experiment in Requirements Engineering Using Conceptual Graphs," Conference on Requirements Elicitation for Software-Based Systems, July 1994.

620. [Rzepka 1994]. Rzepka, W., "Technology Transfer at Rome Laboratory," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 148.

621. [Rzepka and Daley 1986]. Rzepka, W., and P. Daley, "A Prototyping Tool to Assist in Requirements Engineering," 19th Hawaii International Conference on Systems Science, Los Alamitos, California: IEEE Computer Society Press, January 1986.

622. [Rzepka, et al. 1993]. Rzepka, W., et al., "Requirements Engineering Technologies at Rome Laboratory," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 15-18.

623. [Saeed, et al. 1991]. Saeed, A., et al., "The Role of Formal Methods in the Requirements Analysis for Safety-Critical Systems: a Train Set Example," 21st Symposium on Fault-Tolerant Computing, June 1991, pp. 478-485.

624. [Saeed, et al. 1992]. Saeed, A., et al., An Approach to the Assessment of Requirements Specifications for Safety-Critical Systems,

Technical Report 381, Computing Laboratory, University of Newcastle on Tyne, 1992.

625. [Saeki, et al. 1996]. Saeki, M., et al., "Structuring Utterance Records of Requirements Elicitation Meetings Based on Speech Act Theory," Second IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1996.

626. [Sailor 1992]. Sailor, J., "Building and Managing the Requirements/Verification Database," Second Annual International Symposium on Requirements Engineering, Seattle, Washington: National Council on Systems Engineering, July 1992.

627. [Samson 1993]. Samson, D., "Knowledge-Based Test Planning: Framework for a Knowledge-Based System to Prepare a System Test Plan From System Requirements," The Journal of Systems and Software, 20, 2 (February 1993), pp. 115-124.

628. [Sateesh 1995]. Sateesh, T., "Making the Requirements of Process Controlled Systems," 28th Annual IEEE International Conference on System Sciences, Redondo Beach, CA: IEEE Computer Society Press, 1995.

629. [Scheurer and Volz 1994]. Scheurer, R., and M. Volz, "Capturing and Taming Derived Requirements," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 83-89.

630. [Schmitt 1993]. Schmitt, J., "Product Modeling for Requirements Engineering Process Modeling," in Information System Development Process, N. Prakash, et al., eds., New York, New York: Elsevier Science Publishers, 1993, pp. 231-245.

631. [Schneider, et al. 1992]. Schneider, G., et al., "An Experimental Study of Fault Detection in User Requirements Documents," ACM Transactions on Software Engineering and Methodology, 1, 2 (April 1992), pp. 188-204.

632. [Schoening 1994]. Schoening, W., "The Next Big Step in Systems Engineering Tools: Integrating Automated Requirements Tools with Computer Simulated Synthesis and Test," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 409-415.

633. [Selic et al. 1994]. Selic, B., et al., Real-Time Object-Oriented Modeling, New York, New York: Wiley, 1994.

634. [Sfigakis, et al. 1992]. Sfigakis, M., et al., "Mapping Structured Analysis Semantics to Hierarchical Object-Oriented Design," Fifth International Conference on Software Engineering and Its Applications, Nanterre, France: EC2, 1992.

635. [Shekaran 1994]. Shekaran, M., "The Role of Software Architecture in Requirements Engineering," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 245.

636. [Shim and Kim 1997]. Shim, Y., and H. Shim, et al., "Specification and Analysis of Security Requirements for Distributed Applications," Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, June 1997, pp. 374-381.

637. [Sibley, et al. 1993]. Sibley, E., et al., "The Role of Policy in Requirements Definition," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 277-280.

638. [Siddiqi, et al. 1994]. Siddiqi, J., et al., "Towards a System for the Construction, Clarification, Discovery, and Formalization of Requirements," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 230-238.

639. [Siddiqi, et al. 1997]. Siddiqi, J., et al., "Towards Quality Requirements Via Animated Formal Specifications," Annals of Software Engineering, 3, N. Mead, ed., 1997.

640. [Simon 1994]. Simon, B., "Requirements Management Implementation Roadblocks," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 79-82.

641. [Sindre and Opdahl 1993]. Sindre, G., and A. Opdahl, "Concepts for Real-World Modelling," Fifth Conference on Advanced Information Systems Engineering, Paris, France, June 1993.

642. [Si-Said, et al. 199]. Si-Said, S., et al., "Mentor: A Computer Aided Requirements Engineering Environment," Eighth Conference on Advanced Information Systems Engineering (CAiSE '96), Heraklion, Crete, Greece, May 1996,.

643. [Smith 1991]. Smith, M., Software Prototyping, New York, New York: McGraw Hill, 1991.

644. [Smith 1993]. Smith, T., "READS: A Requirements Engineering Tool," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 94-97.

645. [Smith, et al. 1992]. Smith, J., "Systems Engineering with the Right Requirements: An Approach for Assessing User Needs," Second Annual International Symposium on Requirements Engineering, Seattle, Washington: National Council on Systems Engineering, July 1992.

646. [Soares 1994]. Soares, J., "Underlying Concepts in Process Specification," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 48-52.

647. [Souquieres and Levy 1993]. Souquieres, J., and N. Levy, "Description of Specification Developments," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 216-223.

648. [Spanoudakis and Finkelstein 1997]. Spanoudakis, G., and A. Finkelstein, "Reconciling Requirements: A Method for Managing Interference, Inconsistency, and Conflict," Annals of Software Engineering, 3, N. Mead, ed., 1997.

649. [Stamper and Blackhouse 1988]. Stamper, R., and J Blackhouse, "MEASUR: Method for Eliciting Analyzing Specifying User Requirements," in Computerized Assistance During the Information Systems Life Cycle, T. Olle, et al., eds., North-Holland, 1988.

650. [Stephens and Bates 1990]. Stephens, M., and P. Bates, "Requirements Engineering by Prototyping: Experiences in Development of Estimating System," Information and Software Technology, 32, 4 (May 1990), pp. 253-7; also in Software Engineering: A European Perspective, R. Thayer and A. McGettrick, eds., Los Alamitos, California: IEEE Computer Society Press, 1993, pp. 105-111.

651. [Stevens 1994]. Stevens, R., "Structured Requirements," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 99-104.

652. [Stevens and Putlock 1997]. Stevens, R., and G. Putlock, "Improving the Industrial Application of Requirements Management," American Programmer, 10, 4 (April 1997), pp. 17-22.

653. [Stokes 1991]. Stokes, D., "Requirements Analysis," in Software Engineer's Reference Book, J. McDermid, ed., Boca Raton, Florida: CRC Press, 1991.

654. [Stuart and Clements 1991]. Stuart, D., and P. Clements, "Clairvoyance, Capricious Timing Faults, Causality, and Real-Time Specifications," 1991 Real-Time Systems Symposium, Los Alamitos, California: IEEE Computer Society Press, 1991.

655. [Suh, et al. 1992]. Suh, S., et al., "Requirements Specification for a Real-Time Embedded Expert System for Rapid Prototyping," Third International Workshop on Rapid System Prototyping, Los Alamitos, California: IEEE Computer Society Press, June 1992, pp. 172-180.

656. [Sutcliffe 1997]. Sutcliffe, A, "A Technique Combination Approach to Requirements Engineering," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, March 1997.

657. [Sutcliffe and Maiden 1990]. Sutcliffe, A., and N. Maiden, "How Specification Reuse can Support Requirements Analysis," Software Engineering '90, P. Hall, ed., Brighton, UK: Cambridge University Press, July 1990.

658. [Sutcliffe and Maiden 1993]. Sutcliffe, A., and N. Maiden, "Bridging the Requirements Gap: Policies, Goals, and Domains," IEEE International Workshop on Software Specification and Design, Los Alamitos, California: IEEE Computer Society Press, December 1993.

659. [Sutton, et al. 1991]. Sutton, S., et al., "Programming a Software Requirements-Specification Process," First International Conference on the Software Process, Los Alamitos, California: IEEE Computer Society Press, October 1991, pp. 68-89.

660. [Sweeney 1994]. Sweeney, T., "MIL-STD-499B: Systems Engineering," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 62.

661. [Takahashi, et al. 1996]. Takahashi, K., et al., "Hypermedia Support for Collaboration in Requirements Analysis," Second IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1996.

662. [Takeda, et al. 1993]. Takeda, N., et al., "Requirements Analysis by the KJ Editor,"

International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 98-101.

663. [Tamai and Irou 1993]. Tamai, T., and A. Irou, "Requirements and Design Change in Large-Scale Software Development: Analysis from the Viewpoint of Process Backtrack," 15th IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, May 1993, pp. 167-176.

664. [Tanik and Yeh 1989]. Tanik, M., and R. Yeh, "The Role of Rapid Prototyping in Software Development," IEEE Computer, 22, 5 (May 1989), pp. 9-10.

665. [Teng and Sethi 1990]. Teng, J., and V. Sethi, "A Comparison of Information Requirements Analysis Methods: An Experimental Study," Database, 20, 3(March 1990), pp. 27-39.

666. [Trienekens 1994]. Trienekens, J., "Quality Requirements Engineering: First Specification Then Realisation," International Workshop on Requirements Engineering: Foundations of Software Quality, June 1994.

667. [Tsai and Weigert 1993]. Tsai, J., and T. Weigert, Knowledge-Based Software Development for Real-Time Distributed Systems, World Scientific, 1993 .

668. [Tsai, et al. 1992]. Tsai, J., et al., "A Hybrid Knowledge Representation as a Basis of Requirements Specification and Specification Analysis" IEEE Transactions on Software Engineering, 18, 12 (December 1992), pp. 1076-1100.

669. [Tse and Ping 1991]. Tse, T., and L. Ping, "An Examination of Requirements Specification Languages," Computer Journal, 34 (April 1991), pp. 143-152.

670. [Valusek and Fryback 1992]. Valusek, J., and D. Fryback, "Information Requirements Determination: Obstacles Within, Among, and Between Participants," ACM End-User Computing Conference, 1985.

671. [van Lamsweerde, et al. 1995] van Lamsweerde, A., et al., "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

672. [van Schouwen 1992]. van Schouwen, A., The A-7 Requirements Model: R-Examination for Real-Time Systems and an Application to Monitoring Systems, McMaster University Telecommunications Research Institute of Ontario CRL Report #242, Hamilton, Ontario, Canada, February 1992.

673. [van Schouwen, et al. 1993]. van Schouwen, A., et al., "Documentation of Requirements for Computer Systems," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 198-207.

674. [Vassilou, et al. 1990]. Vassilou, Y., et al., "IRIS -- A Mapping Assistant for Generating Designs from Requirements," 2nd Nordic Conference on Advanced Information Systems Engineering (CAiSE '90), Stockholm, Sweden, pp. 307-338, 1990.

675. [Vessey and Conger 1994]. Vessey, I., and S. Conger, "Requirements Specification: Learning Object, Process, and Data Methodologies," Communications of the ACM, 37, 5 (May 1994), pp. 102-113.

676. [von der Beeck 1993]. Von der Beeck, M., "Integration of Structured Analysis and Times Statecharts for Real-Time and Currency Specification," Software Engineering - ESEC '93 Conference, Lecture Notes in Computer Science, DCCXVII, Berlin: Springer Verlag, pp. 313-328, 1993.

677. [von der Beeck 1994]. Von der Beeck, M., "Method Integration and Abstraction From Detailed Semantics to Improve Software Quality," International Workshop on Requirements Engineering: Foundations of Software Quality, June 1994.

678. [Wang and Chen 1993]. Wang, J., and H. Chen., "A Formal Technique to Analyze Real-Time Systems," IEEE International Conference on Computer Software and Applications, Los Alamitos, California: IEEE Computer Society Press, 1993.

679. [Wang, et al. 1992]. Wang, W., et al., "Scenario-Driven Requirements Analysis Method," 2nd IEEE International Conference on Systems Integration, Los Alamitos, California: IEEE Computer Society Press, June 1992, pp. 127-136.

680. [Weinberg 1995]. Weinberg, G., "Just Say No! Improving the Requirements Process, " American Programmer, October 1995.

681. [Welke 1977]. Welke, R., "Current Information Systems Analysis and Design Approaches: Framework, Overview, Comments, and Conclusions," Education and Large Information

Systems, R. Buckingham, ed., Amsterdam, The Netherlands: North-Holland, 1977.

682. [Weller 1993]. Weller, E.F., "Lessons from Three Years of Inspection Data," IEEE Software 10,5, (1993), pp. 38-45.

683. [White 1994]. White, S., "Traceability for Complex Systems Engineering," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 49-55.

684. [White 1994a]. White, S., "ECBS Task Force Standardization Efforts," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 63.

685. [White and Edwards 1995]. White, S., and M. Edwards, "A Requirements Taxonomy for Specifying Complex Systems," First IEEE International Conference on Engineering of Complex Computer Systems, Los Alamitos, California: IEEE Computer Society Press, November 1995.

686. [Whitten, et al. 1994]. Whitten, J., et al., Systems Analysis and Design Methods, Burr Ridge, Illinois: Irwin, 1994.

687. [Wieringa 1996]. Wieringa, R., Requirements Engineering: Frameworks for Understanding, New York: John Wiley & Sons, 1996.

688. [Wiley 1999]. Wiley, B., Essential System Requirements: A Practical Guide to Event-Driven Methods, Addison-Wesley, 1999.

689. [Wilson 1996]. Wilson, W., "Automated Analysis of Requirement Specifications," Fourteenth Annual Pacific Northwest Software Quality Conference, Los Alamitos, California: IEEE Computer Society Press, October 1996.

690. [Wood, et al. 1989]. Wood, W., et al., "Avionics Systems/Software Requirements Specification," Tenth Annual IEEE/AIAA Dayton Chapter Symposium, 1989, pp. 61-70.

691. [Wood, et al. 1994]. Wood, D., et al., "A Multimedia Approach to Requirements Capture and Modeling," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 53-56.

692. [Wood-Harper, et al. 1985]. Wood-Harper, T, et al., Information Systems Definition: The Multi-View Approach, London: Blackwell, 1985.

693. [Woods and Yang 1996]. Woods, S., and Q. Yang, "The Problem Understanding Problem: Analysis and a Heuristic Approach," Eighteenth IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1996.

694. [Wright, et al. 1994]. Wright, P., et al., "Deriving Human-Error Tolerance Requirements From Task Analysis," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 135-142.

695. [Wyder 1996]. Wyder, T., "Capturing Requirements With Use Cases," Software Development, 4, 2 (February 1996), pp. 36-40.

696. [Yamamoto, et al. 1994]. Yamamoto, J., et al., "Object-Oriented Analysis and Design Support System Using Algebraic Specification Techniques," First Asia-Pacific Conference on Software Engineering, December 1994.

697. [Yen and Tiao 1997]. Yen, J., and W. Tiao, "A Systematic Tradeoff Analysis for Conflicting Imprecise Requirements," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, March 1997.

698. [Yourdon 1994]. Yourdon, E., Object-Oriented Systems Design, Englewood Cliffs, New Jersey: Prentice-Hall, 1994.

699. [Yu 1993]. Yu, E., "Modeling Organizations for Information Systems Requirements Engineering," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 34-41.

700. [Yu 1997]. Yu, E., " Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, March 1997.

701. [Zave and Jackson 1996]. Zave, P., and M. Jackson, "Where Do Operations Come From? A Multiparadigm Specification Technique," IEEE Transactions on Software Engineering, 22,, 7 (July 1996), pp. 508-528.

702. [Zave and Jackson 1997]. Zave, P., and M. Jackson, "Requirements for Telecommunications Services: An Attack on Complexity," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, March 1997.

703. [Zowghi and Offen 1997]. Zowghi, D., and R. Offen, "A Logical Framework for Modeling and

Reasoning About the Evolution of Requirements," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, March 1997.

704. [Zucconi 1993]. Zucconi, L., "I Never Knew my Requirements were Object-Oriented Until I Talked to My Analyst," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, p. 230.

705. APPENDIX E – REFERENCES USED TO WRITE AND JUSTIFY THE DESCRIPTION

706. [Acosta 1994]. Acosta, R., et al., "A Case Study of Applying Rapid Prototyping Techniques in the Requirements Engineering Environment," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 66-73.

707. [Alford 1994]. Alford, M., "Attacking Requirements Complexity Using a Separation of Concerns," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 2-5.

708. [Alford 1994]. Alford, M., "Panel Session Issues in Requirements Engineering Technology Transfer: From Researcher to Entrepreneur," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 144.

709. [Anderson 1985]. Anderson, T., Software Requirements: Specification and Testing, Oxford, UK: Blackwell Publishing, 1985.

710. [Anderson and Durney 1993]. Anderson, J., and B. Durney, "Using Scenarios in Deficiency-Driven Requirements Engineering," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 134-141.

711. [Andriole 1992]. Andriole, S., "Storyboard Prototyping For Requirements Verification," Large Scale Systems, 12 (1987), pp. 231-247. 14.[Andriole 1992]

712. [Andriole 1995]. Andriole, S., "Interactive Collaborative Requirements Management," Software Development, (September 1995).

713. [Andriole 1996]. Andriole, S. J., Managing Systems Requirements: Methods, Tools and Cases. McGraw-Hill, 1996.

714. [Anton and Potts 1998]. Anton, A., and C. Potts, "The Use of Goals to Surface Requirements for Evolving Systems," Twentieth International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society, 1998.

715. [Ardis, et al. 1995]. Ardis, M., et al., "A Framework for Evaluating Specification Methods for Reactive Systems," Seventeenth IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

716. [Bickerton and Siddiqi 1993]. Bickerton, M., and J. Siddiqi, "The Classification of Requirements Engineering Methods," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 182-186.

717. [Blanchard and Fabrycky 1998]. Blanchard, B. and Fabrycky, W. J., Systems Engineering Analysis, Prentice Hall, 1998.

718. [Blum 1983]. Blum, B., "Still More About Prototyping," ACM Software Engineering Notes, 8, 3 (May 1983), pp. 9-11.

719. [Blum 1993]. Blum, B., "Representing Open Requirements with a Fragment-Based Specification," IEEE Transaction on Systems, Man and Cybernetics, 23, 3 (May-June 1993), pp. 724-736.

720. [Blyth, et al. 1993a]. Blyth, A., et al., "A Framework for Modelling Evolving Requirements," IEEE International Conference on Computer Software and Applications, Los Alamitos, California: IEEE Computer Society Press, 1993.

721. [Boehm 1994]. Boehm, B., P. Bose, et al., "Software Requirements as Negotiated Win Conditions," Proc. 1st International Conference on Requirements Engineering (ICRE), Colorado Springs, Co, USA, (1994), pp.74-83.

722. [Boehm, et al. 1995]. Boehm, B., et al., "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach," Seventeenth IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

723. [Brown and Cady 1993]. Brown, P., and K. Cady, "Functional Analysis vs. Object-Oriented Analysis: A View From the Trenches," Third International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, July 1993.

724. [Bryne 1994]. Bryne, E., "IEEE Standard 830: Recommended Practice for Software Requirements Specification," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 58.

725. [Burns and McDermid 1994]. Burns, A., and J. McDermid, "Real-Time Safety-Critical Systems: Analysis and Synthesis," IEE Software

Engineering Journal, 9, 6 (November 1994), pp. 267-281.

726. [Checkland and Scholes 1990]. Checkland, P., and J. Scholes, Soft Sysems Methodology in Action. John Wiley and Sons, 1990.

727. [Chung 1991a]. Chung, L., "Representation and Utilization of Nonfunctional Requirements for Information System Design," Third International Conference on Advanced Information Systems Engineering (CAiSE '90), Springer-Verlag, 1991, pp. 5-30.

728. [Chung 1999]. Chung, L., Nixon, B.A., Yu. E., Mylopoulos, J., Non-functional Requirements in Software Engineering, Kluwer Academic Publishers, 1999.

729. [Chung, et al. 1995]. Chung, L., et al., "Using Non-Functional Requirements to Systematically Support Change," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

730. [Connell and Shafer 1989]. Connell, J., and L. Shafer, Structured Rapid Prototyping, Englewood Cliffs, New Jersey, 1989.

731. [Coombes and McDermid 1994]. Coombes, A., and J. McDermid, "Using Quantitative Physics in Requirements Specification of Safety Critical Systems" Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence, Sorrento, Italy, May 1994.

732. [Costello and Liu 1995]. Costello, R., and D. Liu, "Metrics for Requirements Engineering," Journal of Systems and Software, 29, 1 (April 1995), pp. 39-63.

733. [Curtis 1994]. Curtis, A., "How to Do and Use Requirements Traceability Effectively," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 57-64.

734. [Davis 1993]. Davis, A.M., Software Requirements: Objects, Functions and States. Prentice-Hall, 1993.

735. [Davis 1995a]. Davis, A., 201 Principles of Software Development, New York, New York: McGraw Hill, 1995.

736. [Davis 1995b]. Davis, A., "Software Prototyping," in Advances in Computing, 40, M. Zelkowitz, ed., New York, New York: Academic Press, 1995.

737. [Davis, et al. 1997]. Davis, A., et al., "Elements Underlying Requirements Specification," Annals of Software Engineering, 3, N. Mead, ed., 1997.

738. [De Lemos, et al. 1992]. De Lemos, R., et al., "Analysis of Timeliness Requirements in Safety-Critical Systems," Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, Nijmegen, The Netherlands: Springer Verlag, January 1992, pp. 171-192.

739. [Dobson 1991]. Dobson, J., "A methodology for analysing human computer-related issues in secure systems," International Conference on Computer Security and Integrity in our Changing World, Espoo, Finland, (1991), pp. 151-170.

740. [Dobson, et al. 1992]. Dobson, J., et al., "The ORDIT Approach to Requirements Identification," IEEE International Conference on Computer Software and Applications, Los Alamitos, California: IEEE Computer Society Press, 1992, pp. 356-361.

741. [Dorfman and Thayer 1997]. Dorfman, M., and R. H. Thayer, Software Engineering. IEEE Computer Society Press, 1997.

742. [Easterbrook and Nuseibeh 1996]. Easterbrook, S., and B. Nuseibeh, "Using viewpoints for inconsistency management," Software Engineering Journal, 11, 1, 1996, pp.31-43.

743. [Ebert 1997]. Ebert, C., "Dealing with Non-Functional Requirements in Large Software Systems," Annals of Software Engineering, 3, N. Mead, ed., 1997.

744. [El Emam 1997]. EL Amam K., J. Drouin, et al., SPICE: The theory and Practice of Software Process Improvement and Capability Determination. IEEE Computer Society Press, 1997.

745. [El Emam and Madhavji 1995]. El Emam, K., and N. Madhavji, "Measuring the Success of Requirements Engineering," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

746. [Fagan 1986]. Fagan, M.E., "Advances in Software Inspection," IEEE Transactions on Software Engineering 12, 7, 1986, pp. 744-51.

747. [Feather 1991]. Feather, M., "Requirements Engineering: Getting Right from Wrong," Third European Software Engineering Conference, Springer Verlag, 1991.

748. [Fenton 1991]. Fenton, N. E., Software metrics: A rigorous approach. Chapman and Hall, 1991.

749. [Fiksel 1991]. Fiksel, J., "The Requirements Manager: A Tool for Coordination of Multiple Engineering Disciplines," CALS and CE '91, Washington, D.C., June 1991.

750. [Finkelstein 1992]. Finkelstein, A., Kramer, J., B. Nuseibeh and M. Goedicke, "Viewpoints: A framework for integrating multiple perspectives in systems development," International Journal of Software Engineering and Knowledge Engineering, 2, 10, (1992), pp.31-58.

751. [Garlan 1994]. Garlan, D., "The Role of Software Architecture in Requirements Engineering," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, p. 240.

752. [Gause and Weinberg 1989]. Gause, D.C., and G. M. Weinberg, Exploring Requirements : Quality Before Design, Dorset House, 1989.

753. [Gilb and Graham 1993]. Gilb, T., and D. Graham, Software Inspection. Wokingham: Addison-Wesley, 1993.

754. [Goguen and Linde 1993]. Goguen, J., and C. Linde, "Techniques for Requirements Elicitation," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 152-164.

755. [Gomaa 1995]. Gomaa, H., "Reusable Software Requirements and Architectures for Families of Systems," Journal of Systems and Software, 28, 3 (March 1995), pp. 189-202.

756. [Grady 1993a]. Grady, J., Systems Requirements Analysis, New York, New York: McGraw Hill, 1993.

757. [Graham 1998]. Graham, I., Requirements Engineering and Rapid Development : An Object-Oriented Approach, Addison Wesley, 1998.

758. [Hadden 1997]. Hadden, R., "Does Managing Requirements Pay Off?," American Programmer, 10, 4 (April 1997), pp. 10-12.

759. [Hall 1996]. Hall, A., "Using Formal Methods to Develop an ATC Information System," IEEE Software 13, 2, 1996, pp.66-76.

760. [Hansen, et al. 1991]. Hansen, K., et al., "Specifying and Verifying Requirements of Real-Time Systems," ACM SIGSOFT Conference on Software for Critical Systems, December 1991, pp. 44-54.

761. [Harel 1988]. Harel, D., "On Visual Formalisms," Communications of the ACM, 31, 5 (May 1988), pp. 8-20.

762. [Harel 1992]. Harel, D., "Biting the Silver Bullet: Towards a Brighter Future for System Development," IEEE Computer, 25, 1 (January 1992), pp. 8-20.

763. [Harel and Kahana 1992]. Harel, D., and C. Kahana, "On Statecharts with Overlapping," ACM Transactions on Software Engineering and Methodology, 1, 4 (October 1992), pp. 399-421.

764. [Harwell 1993]. Harwell, R., et al, "What is a Requirement," Proc 3rd Ann. Int'l Symp. Nat'l Council Systems Eng., (1993), pp.17-24.

765. [Heimdahl and Leveson 1995]. Heimdahl, M., and N. Leveson, "Completeness and Consistency Analysis of State-Based Requirements," Seventeenth IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

766. [Hofmann 1993]. Hofmann, H., Requirements Engineering: A Survey of Methods and Tools, Technical Report #TR-93.05, Institute for Informatics, Zurich, Switzerland: University of Zurich, 1993.

767. [Honour 1994]. Honour, E., "Requirements Management Cost/Benefit Selection Criteria," Fourth International Symposium on Systems Engineering, Sunnyvale, California: National Council on Systems Engineering, August 1994, pp. 149-156.

768. [Hooks and Stone 1992] Hooks, I., and D. Stone, "Requirements Management: A Case Study -- NASA's Assured Crew Return Vehicle," Second Annual International Symposium on Requirements Engineering, Seattle, Washington: National Council on Systems Engineering, July 1992.

769. [Hsia, et al. 1997]. Hsia, P. et al., "Software Requirements and Acceptance Testing," Annals of Software Engineering, 3, N. Mead, ed., 1997.

770. [Humphery 1988]. Humphery, W.S., "Characterizing the Software Process," IEEE Software 5, 2 (1988), pp. 73-79.

771. [Humphery 1989]. Humphery, W., Managing the Software Process, Reading, Massachusetts: Addison Wesley, 1989.

772. [Hutchings 1995]. Hutchings, A., and S. Knox, "Creating products customers demand," Communications of the ACM, 38, 5, (May 1995), pp. 72-80.

773. [IEEE Standards 1999]. IEEE Software Engineering Standards, Vol 1-4, IEEE, 1999.

774. [Ince 1994]. Ince, D., ISO 9001 and Software Quality Assurance. London: McGraw-Hill, 1994.

775. [Jackson and Zave 1995]. Jackson, M., and P. Zave, "Deriving Specifications from Requirements: An Example," Seventeenth IEEE International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

776. [Jarke and Pohl 1994]. Jarke, M., and K. Pohl, "Requirements Engineering in 2001: Virtually Managing a Changing Reality," IEE Software Engineering Journal, 9, 6 (November 1994), pp. 257-266.

777. [Jarke, et al. 1993]. Jarke, M., et al., "Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 19-31.

778. [Jenkins 1994]. Jenkins, M., "Requirements Capture," Conference on Requirements Elicitation for Software-Based Systems, July 1994.

779. [Jirotka 1991]. Jirotka, M., Ethnomethodology and Requirements Engineering, Centre for Requirements and Foundations Technical Report, Oxford, UK: Oxford University Computing Laboratory, 1991.

780. [Kotonya 1999]. Kotonya, G., "Practical Experience with Viewpoint-oriented Requirements Specification," Requirements Engineering, 4, 3, 1999, pp.115-133.

781. [Kotonya and Sommerville 1996]. Kotonya, G., and I. Sommerville, "Requirements Engineering with viewpoints," Software Engineering, 1, 11, 1996, pp.5-18.

782. [Kotonya and Sommerville 1998]. Kotonya, G., and I. Sommerville, Requirements Engineering: Processes and Techniques. John Wiley and Sons, 1998.

783. [Lam, et al. 1997a]. Lam, W., et al., "Ten Steps Towards Systematic Requirements Reuse," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

784. [Leveson 1986]. Leveson, N. G., "Software safety - why, what, and how," Computing surveys, 18, 2, (1986), pp. 125-163.

785. [Leveson 1995]. Leveson, N. G., Safeware: System Safety and Computers. Reading, Massachusetts: Addison-Wesley, 1995.

786. [Loucopulos and Karakostas 1995]. Loucopulos, P., and V. Karakostas, Systems Requirements Engineering. McGraw-Hill, 1995.

787. [Lutz 1993]. Lutz, R., "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 126-133.

788. [Lutz 1996]. Lutz, R., "Targeting Safety-Related Errors During Software Requirements Analysis," The Journal of Systems and Software, 34, 3 (September 1996), pp. 223-230.

789. [Maiden and Sutcliffe 1993]. Maiden, N., and A. Sutcliffe, "Requirements Engineering By Example: An Empirical Study," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 104-111.

790. [Maiden, et al., 1995] Maiden, N., et al., "How People Categorise Requirements for Reuse: A Natural Approach," Second International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 1995.

791. [Mazza 1996]. Mazza, C., J. Fairclough, B. Melton, D. DePablo, A. Scheffer, and R. Stevens, Software Engineering Standards, Prentice-Hall, 1996.

792. [Mazza 1996]. Mazza, C., J. Fairclough, B. Melton, D. DePablo, A. Scheffer, R. Stevens, M. Jones, G. Alvisi, Software Engineering Guides, Prentice-Hall, 1996.

793. [Modugno, et al. 1997]. Modugno, F., et al., "Integrating Safety Analysis of Requirements Specification," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1997.

794. [Morris, et al. 1994]. Morris, P., et al., "Requirements and Traceability," International Workshop on Requirements Engineering: Foundations of Software Quality, June 1994.

795. [Paulk 1996]. Paulk, M. C., C. V. Weber, et al., Capability Maturity Model: Guidelines for Improving the Software Process. Addison-Wesley, 1995.

796. [Pfleeger 1998]. Pfleeger, S.L., Software Engineering-Theory and Practice. Prentice-Hall, 1998.

797. [Pohl 1994]. Pohl, K., "The Three Dimensions of Requirements Engineering: A Framework and Its Applications," Information Systems 19, 3 (1994), pp. 243-258.

798. [Pohl 1999]. Pohl, K., Process-centered Requirements Engineering, Research Studies Press, 1999.

799. [Potts 1993]. Potts, C., "Choices and Assumptions in Requirements Definition," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, p. 285.

800. [Potts 1994]. Potts, C., K. Takahashi, et. al., "Inquiry-based Requirements Analysis," IEEE Software, 11, 2, 1994, pp. 21-32.

801. [Pressman 1997].Pressman, R.S. Software Engineering: A Practitioner's Approach (4 edition). McGraw-Hill, 1997.

802. [Ramesh et al. 1997]. Ramesh, B., et al., "Requirements Traceability: Theory and Practice," Annals of Software Engineering, 3, N. Mead, ed., 1997.

803. [Roberston and Robertson 1999]. Robertson, S., and J. Robertson, Mastering the Requirements Process, Addison Wesley, 1999.

804. [Rosenberg 1998]. Rosenberg, L., T.F. Hammer and L.L. Huffman, "Requirements, testing and metrics, " 15th Annual Pacific Northwest Software Quality Conference, Utah, October 1998.

805. [Rudd and Isense 1994]. Rudd, J., and S. Isense, "Twenty-two Tips for a Happier, Healthier Prototype," ACM Interactions, 1, 1, 1994.

806. [Rzepka 1992]. Rzepka, W., "A Requirements Engineering Testbed: Concept and Status," 2nd IEEE International Conference on Systems Integration, Los Alamitos, California: IEEE Computer Society Press, June 1992, pp. 118-126.

807. [SEI 1995]. A Systems Engineering Capability Model, Version 1.1, CMU/SEI95-MM-003, Software Engineering Institute, 1995.

808. [Siddiqi and Shekaran 1996]. Siddiqi, J., and M.C. Shekaran, "Requirements Engineering: The Emerging Wisdom," IEEE Software, pp.15-19, 1996.

809. [Sommerville 1996]. Sommerville, I. Software Engineering (5th edition), Addison-Wesley, pp. 63-97, 117-136, 1996.

810. [Sommerville and Sawyer 1997]. Sommerville, I., and P. Sawyer, "Viewpoints: Principles, Problems, and a Practical Approach to Requirements Engineering," Annals of Software Engineering, 3, N. Mead, ed., 1997.

811. [Sommerville, et al. 1993]. Sommerville, I., et al., "Integrating Ethnography into the Requirements Engineering Process," International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 165-173.

812. [Sommerville 1997]. Sommerville, I., and P. Sawyer, Requirements engineering: A Good Practice Guide. John Wiley and Sons, 1997

813. [Stevens 1998]. Stevens, R., P. Brook, K. Jackson and S. Arnold, Systems Engineering, Prentice Hall, 1998.

814. [Thayer and Dorfman 1990]. Thayer, R., and M. Dorfman, Standards, Guidelines and Examples on System and Software Requirements Engineering. IEEE Computer Society, 1990.

815. [Thayer and Dorfman 1997]. Thayer, R.H., and M. Dorfman, Software Requirements Engineering (2nd Ed). IEEE Computer Society Press, 1997.

816. [White 1993]. White, S., "Requirements Engineering in Systems Engineering Practice," IEEE International Symposium on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, January 1993, pp. 192-193.

817. [White 1994]. White, S., "Comparative Analysis of Embedded Computer System Requirements Methods," IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, April 1994, pp. 126-134.

CHAPTER 3
SOFTWARE DESIGN

Guy Tremblay
Département d'informatique
Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal, Québec, Canada, H3C 3P8
tremblay.guy@uqam.ca

TABLE OF CONTENTS

1. 1. INTRODUCTION

2. This document presents a description of the Software Design Knowledge Area for the Guide to the SWEBOK (Stone Man version). It has been developed in accordance with the "Knowledge Area Description Specifications for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge" (version 0.25, March 1999) and with the "Proposed changes to the KA description specifications for version 0.7" (December 1999 and January 2000). Various constraints had to be satisfied by the resulting Knowledge Area (KA) description to respect the above requirements. Among the major constraints were the followings: the KA description had to describe "generally accepted" knowledge not specific to any application domains or development methods; it had to suggest a list of "Proposed reference material" with a reasonably limited number of entries. As it will be seen, the first constraint led to the exclusion of certain topics which, at first, might seem to have been part of Software Design. As for the latter constraint, it led to some difficult choices regarding the selection of reference material, especially since the numerous reviewers of a previous version of this KA description, from which the precious feedback is acknowledged, suggested their own additions to this list of reference material.

3. 2. DEFINITION OF THE SOFTWARE DESIGN KNOWLEDGE AREA

4. Software design, from (software) requirements typically stated in terms relevant to the problem domain, produces a description of a solution that will solve the software-related aspects of the problem. Software design describes how the system is decomposed and organized into components and describes the interfaces between these components (architectural design). Software design also refines the description of these components into a level of detail suitable for allowing their construction (detailed design).

5. In a classical software development life cycle, e.g., ISO/IEC 12207 [ISO95b], software design fits between software requirements analysis and software coding and testing (software construction). Software design encompasses both software architectural design (sometime called top-level design) and software detailed design. Software design plays an important role in the development of a software system in that it allows the developer to produce a model, a blueprint of the solution to be implemented. Such a model can be analyzed and evaluated to determine if it will allow the various requirements to be fulfilled.

This model can also be used to plan the subsequent development activities, in addition to being used as input and starting point of the coding and testing activities.

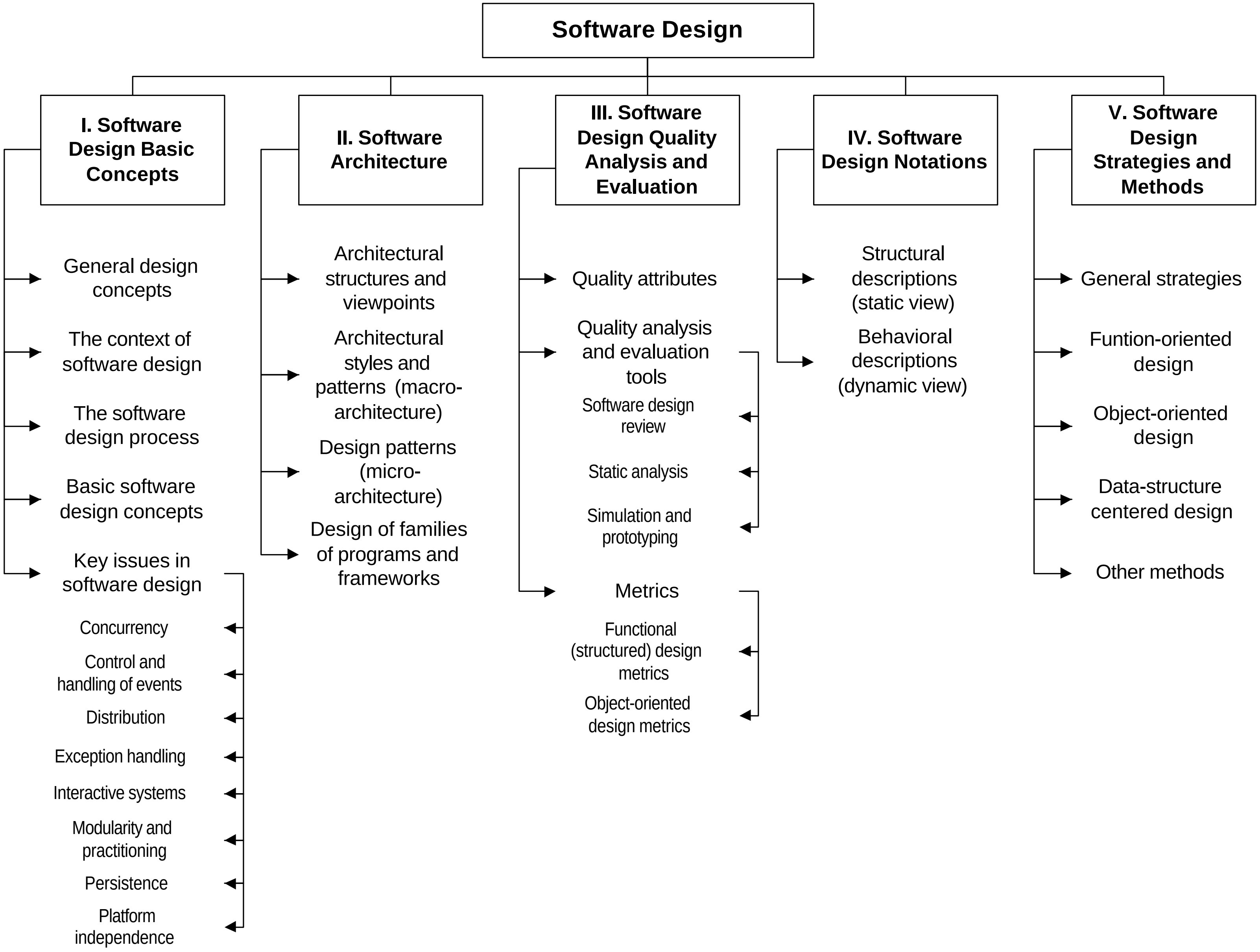
6. It is important to note that certain areas – for example, User Interface Design or Real-time Design – were specifically excluded from the Software Design KA (Guide to the SWEBOK – Straw Man Version), thus are not explicitly discussed in the proposed KA breakdown. However, it is clear that some of the topics included in the present Software Design KA description may also apply to these specialized areas. Finally, some additional "Design" topics were also excluded from the present description, as they were considered to be outside of "Software Design" in the sense mentioned above. Those various issues are discussed in more detail in the Breakdown Rationale section.

7. 3. BREAKDOWN AND DESCRIPTION OF TOPICS FOR THE SOFTWARE DESIGN KA

8. This section presents brief descriptions of each of the major topics of the Software Design Knowledge Area. These brief descriptions (section 3.2) should be sufficient to guide the reader, in section 6, to the appropriate reference material. But first (section 3.1), to give an overall picture of the Software Design KA, an outline of the KA breakdown together with an accompanying figure are presented.

9. 3.1 Breakdown outline

10. Figure 1 gives a graphical presentation of the top-level decomposition of the breakdown for the Software Design Knowledge Area. The detailed breakdown is presented in the following pages.


Software Design
I. Software Design Basic Concepts
II. Software Architecture
III. Software Design Quality Analysis and Evaluation
IV. Software Design Notations
V. Software Design Strategies and Methods
General design concepts
The context of software design
The software design process
Basic software design concepts
Key issues in software design
Concurrency
Control and handling of events
Distribution
Exception handling
Interactive systems
Modularity and practitioning
Persistence
Platform independence
Architectural structures and viewpoints
Architectural styles and patterns (macro-architecture)
Design patterns (micro-architecture)
Design of families of programs and frameworks
Quality attributes
Quality analysis and evaluation tools
Software design review
Static analysis
Simulation and prototyping
Metrics
Functional (structured) design metrics
Object-oriented design metrics
Structural descriptions (static view)
Behavioral descriptions (dynamic view)
General strategies
Funtion-oriented design
Object-oriented design
Data-structure centered design
Other methods

11. 3.2 Description of the Software Design breakdown topics

12. *I. Software Design Basic Concepts*

13. - General design concepts = Notions and concepts relevant to design in general: goals, constraints, alternatives, representations, and solutions. Design as wicked problem solving – no definitive solution, only good vs. bad solutions.

14. - The context of software design = The context (software development life cycle) in which software design fits: software requirements analysis vs. software design; software design vs. software construction; software design and testing. Traceability between the work products of the various phases.

15. - The software design process = The general process by which software is designed: Architectural and detailed design as the two classical phases of software design: whereas architectural design describes how the system is decomposed and organized into components, detailed design describes the specific behavior of these components. Another distinction is the one between software architecture and architectural design: whereas the goal of architectural design is to define the software architecture of a *specific* system, the process of defining a software architecture is considered more *generic.*

16. - Basic software design concepts = Key notions generally considered fundamental to software design, as they form kind of a foundation for understanding many of the proposed approaches to software design: abstraction, modularity (including notions like cohesion and coupling), encapsulation and information hiding, hierarchy, interface vs. implementation, separation of concerns, locality, etc.

17. - Key Issues in Software Design = The key issues which must be dealt with when designing a software system:

18. - Concurrency considerations: how to decompose the systems into processes, tasks and threads and deal with -related atomicity, synchronization and scheduling issues.

19. - Control issues and handling events: how to organize the flow of control, how to handle reactive and temporal events through various mechanisms, e.g., implicit invocation and call-backs, etc.

20. - Distribution: how the software is distributed on the hardware, the role of middleware when dealing with heterogeneous systems, etc.

21. - Handling of faults and exceptions: how to prevent and tolerate faults and deal with exceptional conditions.

22. - Interactive systems and dialogue independence: how to separate the details of the user-interface from the business logic. (Note: the details of User Interface design *per se* are not discussed in the current KA.)

23. - Modularity and partitioning: how to ensure the software is constructed in a modular way, in order to make it understandable and modifiable.

24. - Persistence: how long-lived data is to be handled, e.g., interface with the appropriate databases.

25. - Platform independence: how to ensure the software is relatively independent of the platform (hardware, OS, programming language) on which it will run.

26. *II. Software Architecture*

27. This section on software architecture includes topics dealing both with "generic" software architecture issues and the architectural design of a "specific" software system, as the frontier between the two is not always clear-cut and many of the topics mentioned below apply to both.

28. - Architectural structures and viewpoints: The different high-level facets of a software design that should be described and documented. For some authors, these views pertain to different issues associated with the design of software, for example, the logical view (satisfying the functional requirements) vs. the process view (concurrency issues) vs. the physical view (distribution issues) vs. the development view (how the design is implemented). Other authors use different terminologies, e.g., behavioral vs. functional vs. structural vs. data modeling views. The key idea is that a software design document

is multi-faceted, being made of relatively independent and orthogonal views.

29. ◆ Architectural styles and patterns (macro-architecture): The notion of architectural style – an architectural style is a paradigmatic architectural pattern that can be used to develop the high-level organization of a software system – is becoming an important notion of the field of software architecture. This section presents some of the major styles that have been identified by various authors. These styles are (tentatively) organized as follows:

30. - General structure (e.g., layers, pipes and filters, blackboards);

31. - Distributed systems (e.g., client-server, three-tiers, broker);

32. - Interactive systems (e.g., Model-View-Controller, Presentation-Abstraction-Control)

33. - Adaptable systems (e.g., micro-kernel, reflection);

34. - Other styles (e.g., batch, interpreters, process control, rule-based).

35. ◆ Design patterns (micro-architecture): In the last few years, the field of software design patterns has emerged as an important approach to describing, and thus reusing, design knowledge. Whereas architectural styles can be seen as patterns describing the high-level organization of software systems, its *macro*-architecture, other design patterns can be used to describe details at a lower-level, at a *micro*-architecture level. Such design patterns can (tentatively) be categorized as follows:

36. - Creational patterns: builder, factory, prototype, singleton, etc.

37. - Structural patterns: adapter, bridge, composite, decorator, facade, flyweight, proxy, etc.

38. - Behavioral patterns: command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor, etc.

39. ◆ Design of families of programs and frameworks: One approach to allow the reuse of software design is to design *families* of systems; this can be done by identifying exploitable commonalities among members of such families. Particularly in the field of OO programming, this has been made possible by the notion of framework: a framework is a partially complete software subsystem which can be extended by appropriately instantiating some specific plug-ins (also known as hot points).

40. *III. Software Design Quality Analysis and Evaluation*

41. ◆ Quality attributes: Various attributes are generally considered important for obtaining a design of good quality, e.g., various "ilities" (e.g., maintainability, testability, traceability, plus many others), various "nesses" (e.g., correctness, robustness), including "fitness of purpose". Because there are so many of them, no specific list is given here.

42. ◆ Quality analysis and evaluation tools: Conceptual or technical tools and techniques that can help ensure the quality of a design:

43. - Software design reviews: informal or semi-formal, often group-based, techniques to verify and ensure the quality of design documents, e.g., critical design reviews, active design reviews, inspections, scenario-based techniques.

44. - Static analysis: formal or semi-formal static (non-executable) analysis that can be used to evaluate a design, e.g., fault-tree analysis, dataflow anomaly analysis.

45. - Simulation and prototyping: dynamic techniques to evaluate a design, e.g., performance simulation, feasibility prototype.

46. ◆ Metrics: Formal metrics that can be used to estimate various aspects of the size, structure or quality of a design. Most such metrics generally depend on the approach used for producing the design:

47. - Functional (structured) design metrics: e.g., structural complexity, morphology metrics, etc.

48. - Object-oriented design metrics: weighted methods per class, depth of inheritance tree, etc.

49. *IV. Software Design Notations*

50. A large number of notations and languages exist to represent software design artifacts. Some are used mainly to describe the structural organization of a design, whereas others are used to represent the behavior of such software systems.

51. - Structural descriptions (static view): Notations, mostly graphical, that can be used to describe and represent the structural aspects (static view) of a software design, that is, to describe what the major components are and how they are interconnected. Such notations can be used to describe various views of a software design: the logical view (e.g., Architecture Description Languages (ADL), class and object diagrams, Entity-Relationship Diagrams (ERD), subsystems and packages), the process view (active objects and classes) or the physical view (e.g., deployment diagrams).

52. - Behavioral descriptions (dynamic view): Notations and languages used to describe the dynamic behavior of systems and components. These include various graphical notations (e.g., activity diagrams, Data Flow Diagrams (DFD), sequence diagrams, state transition diagrams) and various textual notations (e.g., formal specification languages, pseudo-code and Program Design Languages (PDL)).

53. *V. Software Design Strategies and Methods*

54. - General strategies: General strategies that can be used to design a system, e.g., divide-and-conquer, information hiding, use of heuristics, use of patterns and pattern languages, iterative and incremental approach to design, etc. Methods, in contrast with general strategies, are more specific in that they generally provide i) a set of notations to be used with the method; ii) a description of the process to be used when following the method; iii) a set of heuristics that provide guidance in using the method. A number of methods are described in the following paragraphs.

55. - Function-oriented (structured) design: One of the classical approach to software design, where the decomposition is centered around the identification of the major systems functions and their elaboration and refinement in a top-down manner. Structured design is generally used after structured analysis (viz., using DFDs and Entity-Relationship Diagrams (ERDs)) has been performed. Various strategies (e.g., transformation analysis, transaction analysis) and heuristics (fan-in/fan-out, scope of effect vs. scope of control, etc.) have been proposed to transform a DFD into a software architecture generally represented by a structure chart (identifying which modules uses/calls which other).

56. - Object-oriented design: This is probably the most (still?!) flourishing field of software design in the last 10-15 years, as numerous software design methods based on objects have been proposed. The field evolved from the early object-based design of the mid-1980's (noun = object; verb = method; adjective = attribute) through object-oriented design, where inheritance and polymorphism play a key role, and to the now emerging field of component-based design, where various meta-information can be defined and accessed (e.g., through reflection). Although object-oriented design's deep roots stem from the concept data abstraction, the notion of responsibility-driven design has also become an important approach to object-oriented design.

57. - Data-structure centered design: Although less popular in North America than in Europe, there has been some interesting work (e.g., M. Jackson, Warnier-Orr) on designing a program starting from the data structures it manipulates rather than from the function it performs. The structures of the input and output data are first described (e.g., using Jackson structure diagrams) and then the program is developed based on these data structure diagrams. Various heuristics have been proposed to deal with special cases, for example, when there is mismatch between the input and output structures.

58. - Other methods: Although software design based on functional decomposition or on object-oriented design are probably the most well-known approaches to software design, other interesting approaches, although probably less "mainstream", do exist, e.g., formal and rigorous methods (e.g., VDM and Cleanroom), knowledge-based approaches, transformational methods, etc.

59. 4. RATIONALE FOR THE BREAKDOWN OF TOPICS

60. The following section briefly goes through the various requirements described in the "Knowledge Area Description Specifications for the Stone Man Version of the Guide to the SWEBOK" (version 0.25) and describe how most of these

requirements are satisfied by the present KA description.

61. First and foremost, the breakdown of topics must describe "generally accepted" knowledge, that is, knowledge for which there is a "widespread consensus". Furthermore, and this is clearly where this becomes difficult, such knowledge must be "generally accepted" today and expected to be so in a 3 to 5 years timeframe. This explains why elements related with software architecture (e.g., "Software Architecture in Practice", Bass, Clements and Kazman, 1998; "Pattern-oriented software architecture", Buschmann et al., 1996), including notions related with architectural styles have been included, even though these are relatively recent topics that might not *yet* be generally accepted. Note that although "UML" (Unified Modeling Language) is not explicitly mentioned in the Design Notations section, many of its elements are indeed present, for example: class and object diagrams, collaboration diagrams, deployment diagrams, sequence diagrams, statecharts.

62. The need for the breakdown to be independent of specific application domains, life cycle models, technologies, development methods, etc., and to be compatible with the various schools (churches?) within software engineering, is particularly apparent within the "Software Design Strategies and Methods" section. In that section, numerous approaches and methods have been included and references given. This is also the case in the "Software Design Notations", which incorporates pointers to many of the existing notations and description techniques for software design artifacts. Although many of the design methods use specific design notations and description techniques, most of these notations are generally useful independently of the particular method that uses them. Note that this is also the approach used in many software engineering books, including the recent UML series of books by the three amigos, which describe "The Unified Modeling Language" apart from "The Unified Software Development Process".

63. The specifications document also specifically asked that the breakdown be as inclusive as possible and that it includes topics related with quality and measurements. Thus, a certain number of topics have been included in the list of topics even though they may not yet be fully considered as generally accepted. For example, although there are a number of books on metrics, design metrics *per se* is rarely discussed in detail and few "mainstream" software engineering books formally discuss this topic. But it is indeed discussed in some books and may become more mainstream in the coming years. Note that although those metrics can sometimes be categorized into high-level (architectural) design vs. component-level (detailed) design, the use of such metrics generally depend on the approach used for producing the design, for example, structured vs. object-oriented design. Thus, the metrics sub-topics have been divided into function- (structured-) vs. object-oriented design.

64. As required by the KA Description Specifications, the breakdown is at most three levels deep and use topic names which, after surveying the existing literature and having made a number of modifications suggested by the various reviewers, should be meaningful when cited outside Guide to the SWEBOK.

65. By contrast with the previous version (0.50) of the Software Design KA Description, and following suggestions made by a number of reviewers, the "Software Design Basic Concepts" section has been expanded to include topics related with design in general and topics introducing the context and process of software design. A totally new subsection has also been recently added: "Key Issues in Software Design". The reason for this new subsection is that a number of reviewers suggested that certain topics, not explicitly mentioned in the previous version, be added, e.g., concurrency and multi-threading, exception handling. Although some of these aspects are addressed by some of the existing design methods, it seemed appropriate that these key issues be explicitly identified and that more specific references be given for them, thus the addition of this new subsection. (Important note: this is a first attempt at such a description of this topic and the author of the Software Design KA Description would gladly welcome any suggestions that could improve and/or refine the content of this subsection.)

66. In the KA breakdown, as mentioned earlier, an explicit "Software Architecture" section has been included. Here, the notion of "architecture" is to be understood in the large sense of defining the structure, organization and interfaces of the components of a software system, by opposition to producing the "detailed design" of the specific components. This is what really is at the heart of Software Design. Thus, the "Software Architecture" section includes topics which pertain to the macro-architecture of a system – what is now becoming known as "Architecture" *per se,* including notions such as "architectural

styles" and "family of programs" – as well as topics related with the micro-architecture of the smaller subsystems – for example, lower-level design patterns. Although some of these topics are *relatively* new, they should become much more generally accepted within the 3-5 years timeframe expected from the Guide to the SWEBOK specifications. By contrast, note that no explicit "Detailed Design" section has been included: topics relevant to detailed design can implicitly be found in the "Software Design Notations" and "Software Design Strategies and Methods" sections, as well as in "The software design process" subsection.

67. The "Software Design Strategies and Methods" section has been divided, as is done in many books discussing software design, in a first section that presents general strategies, followed by subsequent sections that present the various classes of approaches (data-, function-, object-oriented or other approaches). For each of these approaches, numerous methods have been proposed and can be found in the software engineering literature. Because of the limit on the number of references, mostly general references have been given, which can then be used as starting point for more specific references. In the particular case of Object-Oriented Design (OOD), the Unified Software Development Process recently proposed by the UML group, which can be considered a kind of synthesis of many earlier well-known approaches (Booch, OMT, OOSE), was a must, even though it is quite recent (1999). For similar reasons, the "Software Design Notations" section mentions most of the elements that can be found in UML.

68. Another issue, alluded to in the introduction but worth explaining in more detail, is the exclusion of a number of topics which contain "Design" in their name and which, indeed, pertain to the development of software systems. Among these are the followings: User Interface Design, Real-time Design, Database Design, Participatory Design, Collaborative Design. The first two topics were specifically excluded, in the Straw Man document, from the Software Design KA. User Interface Design was considered to be a related discipline (see section 9: Relevant knowledge areas of related disciplines, both Computer Science and Cognitive Sciences) whereas Real-time Design was considered a specialized sub-field of software design, thus did not have to be addressed in this KA description. The third one, Database Design, can also be considered a relevant (specialized) knowledge area of a related discipline (Computer Science). Note that issues related with user-interfaces and databases still have to be dealt with during the software design process, which is why they are mentioned in the "Key Issues in Software Design" section. However, the specific tasks of designing the details of the user interface or database structure are not considered part of Software Design *per se*. As for the last two topics – Participatory and Collaborative Design –, they are more appropriately related with the Software Requirements KA, rather than Software Design. In the terminology of DeMarco (DeM99), these latter two topics belong more appropriately to I-Design (invention design, done by system analysts) rather than D-design (decomposition design, done by designers and coders) or FP-design (family pattern design, done by architecture groups). It is mainly D-design and FP-design, with a major emphasis on D-design, which can be considered as generally accepted knowledge related with Software Design.

69. Concerning the topic of standards, there seems to be few standards that directly pertain to the design task or work product *per se*. However, standards having some indirect relationships with various issues of Software Design do exist, e.g., OMG standards for UML or CORBA. Since the need for the explicit inclusion of standards in the KA breakdown has been put aside ("Proposed changes to the […] specifications […]", Dec. 1999), a few standards having a direct connection with the Software Design KA were included in the recommended reference material section. A number of standards related with design in a slightly more indirect fashion were also added to the list of further readings. Finally, additional standards having only an indirect yet not empty connection with Design were simply mentioned in the general References section. As for topics related with tools, they were excluded from the Software Design breakdown based also on the Dec. 1999 changes to the KA Description Specifications.

70. 5. MATRIX OF SOFTWARE DESIGN TOPICS VS. RECOMMENDED REFERENCE MATERIAL

72. The figure below presents a matrix showing the coverage of the topics of the Software Design KA by the various recommended reference material described in more detail in the following section. A number in an entry indicates a specific section

or chapter number. A "*" indicates a to the whole document, generally either a journal paper or a standard. An interval of the form "n1-n2" indicates a specific range of pages, whereas an interval of the form "n1:n2" indicates a range of sections. For Mar94, the letters refer to one of the encyclopedia's entry: "D" = Design; "DR" = Design Representation; "DD" = Design of Distributed systems".

73. Note: Except for the "Key Issues in Software Design" section, only the top two level of the breakdown have been indicated in the matrix. Otherwise, especially in the "Software Design Notations" subsections, this would have lead to very sparse lines (in an already quite sparse matrix).

	BCK98	BMR+96	BRJ99	Bud94	DT97	FW83	IEE98	ISO95b	JAI97	Mar94	Mey97	Pfl98	Pre97	SB93
74. **I. Software Design Basic Concepts**														
General design concepts				1										*
The context of software design								*		D		2.2	2.2 : 2.7	
The software design process	2.1, 2.3, 2.4			2	266-276	2-22	*	*		D				
Basic software design concepts	6.1	6.3					*		5.1, 5.2, 6.2			5.5	13.4:13.5, 23.2	
Key issues in software design														
Concurrency										DD	30		21.3	
Control and events	5.2										32.4, 32.5	5.3		
Distribution	8.3, 8.4	2.3								DD	30		28.1	
Exceptions											12	5.5		
Interaction independence	6.2	2.4									32.2			
Modularity and partitioning		6.3									3	5.5		
Persistence											31			
Platform independence		2.5									32.2			
75. **II. Software architecture**														
Architectural structures and viewpoints	2.5	6.1	31				*							
Architectural styles and patterns (macro-arch.)	5.1, 5.2, 5.4	1.1: 1.3, 6.2	28									5.3		
Design patterns (micro-arch.)	13.3	1.1: 1.3	28											
Families of programs and frameworks		6.2	28											

	BCK98	BMR+96	BRJ99	Bud94	DT97	FW83	IEE98	ISO95b	JAI97	Mar94	Mey97	Pfl98	Pre97	SB93
76. **III. Software design quality analysis and evaluation**														
Quality attributes	4.1	6.4		4.1: 4.3						D	3	5.5		
Quality analysis and evaluation	9.1, 9.2, 10.2, 10.3					542-576			5.5, 7.3			5.6, 5.7		
Metrics									5.6, 6.5, 7.4				18.4, 23.4,23.5	
77. **IV. Software design notations**														
Structural descriptions (static)	12.1, 12.2		4, 8, 11, 12, 14, 30, 31	6					5.3, 6.3	DR			12.3, 12.4	
Behavioral descriptions (dynamic)			18, 19, 24	6	181-192	485-490, 506-513			5.3, 7.2	DR	11		14.11 12.5	
78. **V. Software design strategies and methods**														
General strategies		5.1: 5.4		7.1, 8		304-320, 533-539				D		2.2		
Function-oriented design					170-180	328-352			5.4				13.5, 13.6, 14.3:14.5	
OO design					148-159, 160-169	420-436			6.4	D			19.2, 19.3, 21.1:21.3	
Data-oriented design						514-532				D				
Other methods				14	181-192	395-407, 461-468					11	2.2		

79. 6. RECOMMENDED REFERENCE MATERIAL FOR THE SOFTWARE DESIGN KA

80. In what follows, reference material for the various topics presented in the proposed breakdown of topics are suggested. Section 6.1 gives a brief presentation of each of the recommended reference. Then, in section 6.2, specific and detailed references are given for each of the major topics of the breakdown. Note that, for some topics, a number of global references are given for a *non-leaf* topic, rather a specific reference for each particular leaf topic. This seemed preferable because some of these topics were discussed in a number of interesting references.

81. Note that few references to existing standards have been included in this list, for the reasons explained earlier. Also note that almost *no* specific references have been given for the various design methods except very general ones. See the list of further readings in section 7 for more

precise and detailed references on such methods, especially for references to OO design methods.

82. 6.1 Brief description of the recommended references

83. [BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.*

84. A recent and major work on software architecture. It covers all the major topics associated with software architecture: what software architecture is, quality attributes, architectural styles, enabling concepts and techniques (called unit operations), architecture description languages, development of product lines, etc. Furthermore, it present a number of case studies illustrating major architectural concepts, including a chapter on CORBA and one on the WWW.

85. [BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture – A System of Patterns.*

86. According to the Software Design KA Description author's humble opinion, this is probably the best and clearest introduction to the notions of software architecture and patterns (both architectural and lower-level ones). Distinct chapters are dedicated to architectural patterns, design patterns and lower-level idioms. Another chapter discusses the relationships between patterns, software architecture, methods, frameworks, etc. This chapter also includes an interesting presentation of so-called "enabling techniques for software architecture", discussing many of the elements of the "Basic software design concepts" section, e.g., abstraction, encapsulation, information hiding, coupling and cohesion, etc.

87. [BRJ99] G. Booch, J. Rumbauch, and I. Jacobson. *The Unified Modeling Language User Guide.*

88. A comprehensive and thorough presentation of UML, which incorporates many of the notations mentioned in the "Software Design Notations" section.

89. [Bud94] D. Budgen. *Software Design.*

90. One of the few books known to the author – maybe the only one – which is neither a general software engineering textbook nor a book describing a specific software design method. This is probably the book that comes closest to the current Software Design KA description, as it discusses topics such as the followings: the nature of design; the software design process; design qualities; design viewpoints; design representations; design strategies and methods (including brief presentations of a number of such methods, e.g., JSP, SSASD, JSD, OOD, etc.) The only drawback might be its availability – at least in Canada, as the author of the Software Design KA description only managed to get hold of a copy of this book a few days before delivering the final version of the current KA description (version 0.70) – but it is worth reading.

91. [DT97] M. Dorfman and R.H. Thayer (eds.). *Software Engineering.*

92. This book contains a collection of papers on software engineering in general. Two chapters deal more specifically with software design. One of them contains a general introduction to software design, briefly presenting the software design process and the notions of software design methods and design viewpoints. The other chapter contains an introduction to object-oriented design and a comparison of some existing OO methods. The following articles are particularly interesting for Software Design:

93. - D. Budgen, Software Design: An Introduction, pp. 104-115.

94. - L.M. Northrop, Object-Oriented Development, pp. 148-159.

95. - A.G. Sutcliffe, Object-Oriented Systems Development: A Survey of Structured Methods, pp.160-169.

96. - C. Ashworth, Structured Systems Analysis and Design Method (SSADM), pp. 170-180.

97. - R. Vienneau, A Review of Formal Methods, pp. 181-192.

98. - J.D. Palmer, Traceability, pp. 266-276.

99. [FW83] P. Freeman and A.I. Wasserman. *Tutorial on Software Design Techniques*, fourth edition.

100. Although this is an old book, it is a very interesting one because it allows to better understand the evolution of the software design field. This book is a collection of papers where each paper presents a software design technique. The techniques range from basic strategies like stepwise refinement to, at the time, more refined method such as structured design à la Yourdon and Constantine. An historically important

reference. The following articles are particularly interesting for Software Design:

101. - P. Freeman, Fundamentals of Design, pp. 2-22.

102. - D.L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, pp. 304-309.

103. - D.L. Parnas, Designing Software for Ease of Extension and Contraction, pp. 310-320.

104. - W.P. Stevens, G.J. Myers and L.L. Constantine, Structured Design, pp. 328-352.

105. - G. Booch, Object-Oriented Design, pp. 420-436.

106. - S.H. Caine and E.K. Gordon, PDL – A Tool for Software Design, pp. 485-490.

107. - C.M. Yoder and M.L. Schrag, Nassi-Schneiderman Charts: An Alternative to Flowcharts for Design, pp. 506-513.

108. - M.A. Jackson, Constructive Methods of Program Design, pp. 514-532.

109. - N. Wirth, Program Development by Stepwise Refinement, pp. 533-539.

110. - P. Freeman, Toward Improved Review of Software Design, pp. 542-547.

111. - M.E. Fagan, Design and Code Inspections to Reduce Errors in Program Development, pp. 548-576.

112. [IEE98] IEEE Std 1016-1998. IEEE Recommended Practice for Software Design Descriptions.

113. This document describes the information content and recommended organization that should be used for software design descriptions. The attributes describing design entities are briefly described: identification, type, purpose, function, subordinates, dependencies, interfaces, resources, processing and data. How these different elements should be organized is then presented.

114. [ISO95b] ISO/IEC Std 12207. Information technology – Software life cycle processes.

115. A detailed description of the ISO/IEC-12207 life cycle model. Clearly shows where Software Design fits in the whole software development life cycle.

116. [Jal97] P. Jalote. *An integrated approach to software engineering, 2nd ed.*

117. A general software engineering textbook with a good coverage of software design, as three chapters discuss this topic: one on function-oriented design, one on object-oriented design, and the other on detailed design. Another interesting point is that all these chapters have a metrics section.

118. [Mar94] J.J. Marciniak. *Encyclopedia of Software Engineering.*

119. A general encyclopedia that contains (at least) three interesting articles discussing software design. The first one, "Design" (K. Shumate), is a general overview of design discussing alternative development processes (e.g., waterfall, spiral, prototyping), design methods (structured, data-centered, modular, object-oriented). Some issues related with concurrency are also mentioned. The second one discusses the "Design of distributed systems" (R.M. Adler): communication models, client-server and services models. The third one, "Design representation" (J. Ebert), presents a number of approaches to the representation of design. It is clearly not a detailed presentation of any method; however, it is interesting in that it tries to explicitly identify, for each such method, the kinds of components and connectors used within the representation.

120. [Mey97] B. Meyer. *Object-Oriented Software Construction (Second Edition).*

121. A detailed presentation of the Eiffel OO language and its associated Design-By-Contract approach, which is based on the use of formal assertions (pre/post-conditions, invariants, etc). It introduces the basic concepts of OO design, along with a discussion of many of the key issues associated with software design, e.g., user interface, exceptions, concurrency, persistence, etc.

122. [Pfl98] S.L. Pfleeger. *Software Engineering – Theory and Practice.*

123. A general software engineering book with one chapter devoted to design. Briefly presents and discusses some of the major architectural styles and strategies and some of the concepts associated with the issue of concurrency. Another section presents the notions of coupling and cohesion and also deals with the issue of exception handling. Techniques to improve and to evaluate a design are also presented: design by contract, prototyping, reviews. Although this chapter does not delve into any topic, it can be an interesting starting point for a number of issues not discussed in

some of the other general software engineering textbooks.

124. [Pre97] R.S. Pressman. *Software Engineering – A Practitioner's Approach (Fourth Edition).*

125. Probably the classic among all the general software engineering textbooks (4th edition!) It contains over 10 chapters that deal with notions associated with software design in one way or another. The basic concepts and the design methods are presented in two distinct chapters. Furthermore, the topics pertaining to the function-based (structured) approach are separated (part III) from those pertaining to the object-oriented approach (part IV). Independent chapters are also devoted to metrics applicable to each of those approaches, a specific section addressing the metrics specific to design. A chapter discusses formal methods and another presents the Cleanroom approach. Finally, another chapter discusses client-server systems and distribution issues.

126. [SB93] G. Smith and G. Browne. Conceptual foundations of design problem-solving.

127. An interesting paper that discusses what is design in general. More specifically, it presents the five basic concepts of design: goals, constraints, alternatives, representations, and solutions. The bibliography is a good starting point for obtaining additional references on design in general.

128. 6.2 Recommended references for each of the KA topic

129. Note: The numbers after the reference key indicate the appropriate chapter. In the case of Mar94, the appropriate entry of the encyclopedia is indicated as follows: "D" = Design; "DR" = Design Representation; "DD" = Design of Distributed systems". Note that, contrary, to the matrix presented in section 5, we have only indicated the appropriate chapter (or part) number, not the specific sections or pages.

130. I. Software Design Basic Concepts

131. General design concepts

132. [Bud94: 1][SB93]

133. The context of software design

134. [ISO95b][Mar94: D][Pfl98: 2][Pre97: 2]

135. The software design process

136. [BCK98: 2][DT97: 7][FW83: I][IEE98][ISO95b][Mar94]

137. Basic software design concepts

138. [BCK98: 6][BMR+96: 6][IEE98][Jal97: 5, 6][Pfl98: 5][Pre97: 13, 23]

139. Key Issues in Software Design

140. Concurrency considerations

141. [Mar94: DD][Mey97: 30][Pre97: 21]

142. Control and handling of events

143. [BCK98: 5][Mey97: 32][Pfl98: 5]

144. Distribution

145. [BCK98: 8][BMR+96: 2][Mar94: DD][Mey97: 30][Pre97: 28]

146. Exception handling

147. [Mey97: 12][Pfl98: 5]

148. Interactive systems and dialogue independence

149. [BCK98: 6][BMR+96: 2.4][Mey97: 32]

150. Modularity and partitioning

151. [BMR+96: 6][Mey97: 3][Pfl98: 5]

152. Persistence

153. [Mey97: 31]

154. Platform independence

155. [BMR+96: 2][Mey97: 32]

156. II. Software Architecture

157. Architectural structures and viewpoints

158. [BCK98: 2][BMR+96: 6][BRJ99: 31][IEE98]

159. Architectural styles and patterns (macro-architecture)

160. [BCK98: 5][BMR+96: 1, 6][BRJ99: 28][Pfl98: 5]

161. Design patterns (micro-architecture)

162. [BCK98: 13][BMR+96: 1][BRJ99: 28]

163. Design of families of programs and frameworks

164. [BMR+96: 6][BRJ99: 28]

165. III. Software Design Quality Analysis and Evaluation

166. Quality attributes

167. [BCK98: 4][BMR+96: 6][Mar94: D][Mey97: 3][Pfl98: 5]

168. Quality analysis and evaluation tools

169. [BCK98: 9-10][FW83: VIII][Jal97: 5, 7][Pfl98: 5]

170. Metrics

171. [Jal97: 5-7][Pre97: 18, 23]

172. *IV. Software Design Notations*

173. Structural descriptions (static view)

174. ADL (Architecture Description Languages)

175. [BCK98: 12]

176. Class and objects diagrams

177. [BRJ99: 8, 14][Jal97: 5,6]

178. CRC (Class-Responsibilities-Collaborators) Cards

179. [BRJ99: 4][BMR+96]

180. Deployment diagrams

181. [BRJ99: 30]

182. ERD (Entity-Relationship Diagrams)

183. [DT97: 4][Mar94: DR]

184. IDL (Interface Description Languages)

185. [BCK98: 8][BJR99: 11]

186. Jackson structure diagrams

187. [DT97: 4][Mar94: DR]

188. Structure charts

189. [DT97: 4-5][Jal97: 5][Mar94: DR][Pre97: 12, 14]

190. Subsystems (packages) diagrams

191. [BRJ99: 12, 31][DW99: 7]

192. Behavioral descriptions (dynamic view)

193. Activity diagrams

194. [BRJ99: 19]

195. Collaboration diagrams

196. [BRJ99: 18]

197. Data flow diagrams

198. [Jal97: 5, 7][Mar94: DR][Pre97: 14]

199. Decision tables and diagrams

200. [Pre97: 14]

201. Flowcharts and structured flowcharts

202. [FW83: VII][Mar94: DR][Pre97: 14]

203. Formal specification languages

204. [Bud94: 14][DT97: 5][Mey97: 11]

205. Pseudo-code and PDL (Program Design Language)

206. [FW83: VII][Jal97: 7][Pre97: 14]

207. Sequence diagrams

208. [BRJ99: 18]

209. State transition diagrams and statecharts

210. [BRJ99: 24][Mar94: DR][Jal97: 7]

211. *V. Software Design Strategies and Methods*

211. General strategies [Bud94: 8][Mar94: D]

212. Divide-and-conquer and stepwise refinement [FW83: VII]

213. Data abstraction and information hiding [FW83: V]

214. Iterative and incremental design [Pfl98: 2]

215. Heuristics-based design [Bud94: 7]

216. Pattern-based design and pattern languages [BMR+96: 5]

217. Function-oriented design

218. [DT97: 5][FW83: V][Jal97: 5][Pre97: 13-14]

219. Object-oriented design

220. [DT97: 5][FW83: VI][Jal97: 6][Mar94: D][Pre97: 19, 21]

221. Data-structure centered design

222. [FW83: III, VII][Mar94: D]

223. Other methods

224. Formal and rigorous methods [Bud94: 14][DT97: 5][Mey97: 11]

225. Transformational methods [Pfl98: 2]

226. 7. LIST OF FURTHER READINGS

227. The following section suggests a list of additional interesting reading material related with Software Design. A number of standards are mentioned; additional standards that may be pertinent or applicable to Software Design, although in a somewhat less direct way, are also mentioned, although not further described, in the general References section at the end of the document.

228. [Boo94] G. Booch. *Object Oriented Analysis and Design with Applications, 2nd ed.*

229. A classic in the field of OOD. The book introduces a number of notations that were to become part of UML (although sometimes with some slight modifications): class vs. objects diagrams, interaction diagrams, statecharts-like diagrams, module and deployment, process structure diagrams, etc. It also introduces a process to be used for OOA and OOD, both a higher-level (life cycle) process and a lower-level (micro-) process.

230. [Cro84] N. Cross (ed.). *Developments in Design Methodology.*

231. This book consists in a series of papers related to design in general, that is, design in other contexts than Software Design. Still, many

notions and principles discussed in some of these papers do apply to Software Design, e.g., the idea of design a wicked-problem solving.

232. [CY91] P. Coad and E. Yourdon. *Object-Oriented Design*.

233. This is yet another classic in the field of OOD – note that the second author is one of the father of classical Structured Design. An OOD model developed with their approach consists of the following four components, trying to separate how some of the key issues should be handled: problem domain, human interaction, task management and data management.

234. [DW99] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML – The Catalysis Approach.*

235. A thorough presentation of a specific OO approach with an emphasis on component design. The development of static, dynamic and interaction models is discussed. The notions of components and connectors are presented and illustrated with various approaches (Java Beans, COM, Corba); how to use such components in the development of frameworks is also discussed. Another chapter discusses various aspects of software architecture. The last chapter introduces a pattern system for dealing with both high-level and detailed design, the latter level touching on many key issues of design such as concurrent, distribution, middleware, dialogue independence, etc.

236. [FP97] N.E. Fenton and S.L. Pfleeger. *Software Metrics – A Rigorous & Practical Approach (Second Edition).*

237. This book contains a detailed presentation of numerous software metrics. Although the metrics are not necessarily presented based on the software development life cycle, many of those metrics, especially in chapter 7 and 8, are applicable to software design.

238. [GHJV95] E. Gamma et *al*. *Design Patterns – Elements of Reusable Object-Oriented Software*.

239. The seminal work on design patterns. A detailed catalogue of patterns related mostly with the micro-architecture level.

240. [Hut94] A.T.F. Hutt. *Object Analysis and Design – Description of Methods. Object Analysis and Design – Comparison of Methods.*

241. These two books describe (first book) and compare (second book), in a very outlined manner, a large number of OO analysis and design methods. Useful as a starting point for obtaining additional pointers and references to OOD methods, not so much as a detailed presentation of those methods.

242. [IEE90] IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology.

243. This standard is not specifically targeted to Software Design, which is why it has not been included in the recommended references. It describes and briefly explains many of the common terms used in the Software Engineering field, including many terms from Software Design.

244. [ISO91] ISO/IEC Std 9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use.

245. This standard describes six high-level characteristics that describe software quality: functionality, reliability, usability, efficiency, maintainability, portability.

246. [JBP+91] J. Rumbaugh et *al. Object-Oriented Modeling and Design.*

247. This book is another classic in the field of OOA and OOD. It was one of the first to clearly introduce the distinction between object, dynamic and functional modeling. However, contrary to [Boo94] whose emphasis is mostly on design, the emphasis here is slightly more on analysis, although a number of elements do apply to design too.

248. [JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.*

249. A detailed and thorough presentation of the Unified Software Development Process proposed by the Rational amigos. The notion of architecture plays a central role in this development process, the process being said to be architecture-centric. However, the associated notion of architecture is slightly different from the traditional purely design-based one: an architecture description is supposed to contain views not only from the design model but also from the use-case, deployment and implementation models. A whole chapter is devoted to the presentation of the iterative and incremental approach to software development. Another chapter is devoted to design *per se*, whose goal is to produce both the design model, which includes the logical (e.g., class diagrams, collaborations, etc.) and process (active

objects) views, and the deployment model (physical view).

250. [Kru95] P.B. Kruchten. The 4+1 view model of architecture.

251. A paper that explains in a clear and insightful way the importance of having multiple views to describe an architecture. Here, architecture is understood in the UML Process sense mentioned earlier, not in its strictly design-related way. The first four views discusses in the paper are the logical, process, development and physical views, whereas the fifth one (the "+1") is the use case view, which binds together the previous views. The views more intimately related with Software Design are the logical and process ones.

252. [McC93] S. McConnell. *Code Complete.*

253. Although this book is probably more closely related with Software Construction, it does contain a section on Software Design with a number of interesting chapters, e.g., "Characteristics of a High-Quality Routines", "Three out of Four Programmers Surveyed Prefer Modules", "High-Level Design in Construction". One of these chapters ("Characteristics […]") contains an interesting discussion on the use of assertions in the spirit of Meyer's Design-by-Contract; another chapter ("Three […]") discusses cohesion and coupling as well as information hiding; the other chapter ("High-Level […]") gives a brief introduction to some design methodologies (structured design, OOD).

254. [Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development.*

255. This book is particularly interesting for its discussion of framework design using what is called the "hot-spot driven" approach to the design of frameworks. The more specific topic of design patterns is better addressed in [BMR+96].

256. [Rie96] A.J. Riel. *Object-Oriented Design Heuristics.*

257. This book, targeted mainly towards OO design, presents a large number of heuristics that can be used in software design. Those heuristics address a wide range of issues, both at the architectural level and at the detailed design level.

258. [WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.*

259. Interesting as it introduced the notion of responsibility-driven design to OOD. Before that, OOD was often considered synonymous with data abstraction-based design. Although it is true that an object does encapsulate data and associated behavior, focusing strictly on this aspect may not lead, according to the responsibility-driven design approach, to the best design.

260. [Wie98] R. Wieringa. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques.

261. An interesting survey article that presents a wide range of notations and methods for specifying software systems and components. It also introduces an interesting framework for comparison based on the kinds of system properties to be specified: functions, behavior, communication or decomposition.

262. 8. REFERENCES

263. [BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* SEI Series in Software Engineering. Addison-Wesley, 1998.

264. [BDA+98] P. Bourque, R. Dupuis, A. Abran, J.W. Moore, L. Tripp, J. Shyne, B. Pflug, M. Maya, and G. Tremblay. Guide to the software engineering body of knowledge – a straw man version. Technical report, Dépt. d'Informatique, UQAM, Sept. 1998.

265. [BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture – A System of Patterns.* Wiley, West Sussex, England, 1996.

266. [Boo94] G. Booch. *Object Oriented Analysis and Design with Applications, 2nd ed.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

267. [BRJ99] G. Booch, J. Rumbauch, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, Reading, MA, 1999.

268. [Bud94] D. Budgen. *Software Design.* Addison-Wesley, Wokingham, England, 1994.

269. [Cop99] J. Coplien. *Multi-Paradigm Design for C++.* Addison-Wesley, 1999.

270. [Cro84] N. Cross (ed.). *Developments in Design Methodology.* John Wiley, 1984.

271. [CY91] P. Coad and E. Yourdon. *Object-Oriented Design.* Yourdon Press, 1991.

272. [DeM99] T. DeMarco. *The Paradox of Software Architecture and Design.* Stevens Prize Lecture, August 1999.

273. [DT97] M. Dorfman and R.H. Thayer. *Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA, 1997.

274. [DW99] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML – The Catalysis Approach.* Addison-Wesley, Reading, MA, 1999.

275. [FP97] N.E. Fenton and S.L. Pfleeger. *Software Metrics – A Rigorous & Practical Approach (Second Edition).* International Thomson Computer Press, 1997.

276. [FW83] P. Freeman and A.I. Wasserman. *Tutorial on Software Design Techniques*, fourth edition. IEEE Computer Society Press, Silver Spring, MD, 1983.

277. [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, Reading, MA, 1995.

278. [Hut94] A.T.F. Hutt. *Object Analysis and Design – Comparison of Methods. Object Analysis and Design – Description of Methods.* John Wiley & Sons, New York, 1994.

279. [IEE88] IEEE. IEEE Standard Dictionary of Measures to Produce Reliable Software. IEEE Std 982.1-1988, IEEE, 1988.

280. [IEE88b] IEEE. IEEE Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software. IEEE Std 982.2 -1988, IEEE, 1988.

281. [IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, IEEE, 1990.

282. [IEE98] IEEE. IEEE Recommended Practice for Software Design Descriptions. IEEE Std 1016-1998, IEEE, 1998.

283. [ISO91] ISO/IEC. Information technology – Software product evaluation – Quality characteristics and guidelines for their use. ISO/IEC Std 9126: 1991, ISO/IEC, 1991.

284. [ISO95] ISO/IEC. Open distributed processing – Reference model. ISO/IEC Std 10746: 1995, ISO/IEC, 1995.

285. [ISO95b] ISO/IEC. Information technology – Software life cycle processes. ISO/IEC Std 12207: 1995, ISO/IEC, 1995.

286. [Jal97] P. Jalote. *An Integrated Approach to Software Engineering, 2nd ed.* Springer, New York, NY, 1997.

287. [JBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

288. [JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, Reading, Ma, 1999.

289. [JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley, 1992.

290. [Kru95] P.B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

291. [Mar94] J.J. Marciniak. *Encyclopedia of Software Engineering.* John Wiley & Sons, Inc., New York, NY, 1994.

292. [McCr93] S. McConnell. *Code Complete.* Microsoft Press, Redmond, WA, 1993.

293. [Mey97] B. Meyer. *Object-Oriented Software Construction (Second Edition).* Prentice-Hall, Upper Saddle River, NJ, 1997.

294. [OMG98] OMG. The common object request broker: Architecture and specification. Technical Report Revision 2.2, Object Management Group, February 1998.

295. [OMG99] UML Revision Task Force. OMG Unified Modeling Language specification, v. 1.3. document ad/99-06-08, Object Management Group, June 1999.

296. [otSESC98] Architecture Working Group of the Software Engineering Standards Committee. Draft recommended practice for information technology – system design – architectural description. Technical Report IEEE P1471/D4.1, IEEE, New York, NY, December 1998.

297. [Pfl98] S.L. Pfleeger. *Software Engineering – Theory and Practice.* Prentice-Hall, Inc., 1998.

298. [Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley and ACM Press, 1995.

299. [Pre97] R.S. Pressman. *Software Engineering – A Practitioner's Approach (Fourth Edition).* McGraw-Hill, Inc., 1997.

300. [Rie96] A.J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, Reading, MA, 1996.

301. [SB93] G. Smith and G. Browne. Conceptual foundations of design problem-solving. *IEEE*

Trans. on Systems, Man, and Cybernetics, 23(5):1209–1219, 1993.

302. [WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

303. [Wie98] R. Wieringa. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Computing Surveys,* 30(4): 459–527, 1998.

CHAPTER 4
SOFTWARE CONSTRUCTION

Terry Bollinger
The MITRE Corporation

Philippe Gabrini and Louis Martin
Université du Québec à Montréal
{gabrini.philippe} {martin.louis}@uqam.ca

TABLE OF CONTENTS

1. 1. Software Construction

2. *Software construction* is the most fundamental act of software engineering: the construction of working, meaningful software through a combination of coding, self-validation, and self-testing (unit testing) by a programmer. Far from being a simple mechanistic “translation” of good design into working software, software construction burrows deeply into some of the most difficult issues of software engineering. It requires the establishment of a meaningful dialog between a person and a computer – a “communication of intent” that must reach from the slow and fallible human to a fast and unforgivingly literal computer. Such a dialog requires that the computer perform activities for which it is poorly suited, such as understanding implicit meanings and recognizing the presence of nonsensical or incomplete statements. On the human side, software construction requires that forgetful, sloppy, and unpredictable people train themselves to be precise and thorough to the point that, at the least, they do not appear to be completely insane from the viewpoint of a very literal computer. The relationship works only because each side possesses certain capabilities that the other lacks. In the symbiosis of disparate entities that is software construction, the computer provides astonishing reliability, retention, and (once the need has been explained) speed of performance. Meanwhile, the human side provides something utterly lacking on the part of the computer: Creativity and insight into how to solve new, difficult problems, plus the ability to express those solutions with sufficient precision to be meaningful to the computer. Perhaps the most remarkable aspect of software construction is that it is possible *at all*, given the strangeness of the symbiosis on which it is based.

3. 1.1 Software Construction and Software Design

4. Software construction is closely related to software design (see *Knowledge Area Description for Software Design*). *Software design* is a collection of skills and techniques for breaking up a large, complex problems into structured collections of smaller, easier-to-solve problems. Software design methods can be applied repeatedly until the resulting subproblems are small enough to be handled with confidence by a single developer. It is at this point – that is, when the design process has broken the larger problem up into easier-to-handle chunks – that *software construction* is generally understood to begin. This definition also implies the distinction that while software construction necessarily produces executable software, software design does not necessarily produce any executable products at all.

5. In practice, however, the boundary between design and construction is seldom this clearly defined. Firstly, software construction is greatly influenced by the scale or size of the software

product being constructed. Very small projects in which the design problems are already "construction size" may neither require nor need an explicit design phase, and very large projects may require a much more interactive relationship between design and construction as different prototyping alternatives are proposed, tested, and discarded or used. Secondly, many of the techniques of software design also apply to software construction, since dividing problems into smaller parts is just as much a part of construction as it is design. Thirdly, effective design techniques always contain some degree of guessing or approximation in how they define their subproblems. A few of the resulting approximations will turn out to be wrong, and will require corrective actions during software construction. These corrective actions are most easily taken if construction is capable of applying the same techniques. (While another seemingly obvious solution would be to remove guessing and approximation altogether from design methods, that would contradict the premise that the original problem was too large and complex to be solved in one step. Effective design techniques instead acknowledge risk, work to reduce it, and help make sure that effective alternatives will be available when some choices eventually prove wrong.)

6. Finally, there is a common misconception that software design solves all of the "hard" problems in software development, making software construction into little more than a mechanistic translation of software designs into final software. This is simply not the case. Design and construction both require sophisticated problem solving skills, although the two activities have somewhat different emphases. In design the emphasis is on how to divide up a complex problem effectively, while in construction the emphasis is on finding a complete and executable solution to a problem. When software construction techniques do become so well-defined that they can be applied mechanistically, the proper route for the software engineer is to automate those techniques and move on to new problems, ones whose answers are not so well defined. This trend toward automation of well-defined tasks began with the first assemblers and compilers, and it has continued unabated as new generations of tools and computers have made increasingly powerful levels of construction automation possible. Projects that do contain highly repetitive, mechanistic software construction steps should examine their designs, processes, and tools sets more closely for ways to automate such needlessly repetitive steps out of existence.

7. 1.2 The Role of Tools in Construction

8. In software engineering, a *tool* may be broadly defined as any hardware or software device that provides significant productivity or quality improvements to the overall development process (see *Knowledge Area Description for Engineering Tools and Methods*). This is a very inclusive definition, however, since it encompasses general-purpose hardware devices such as computers and peripherals that are part of an overall software-engineering environment. *Software construction tools* are a more specific category of tools that are both software-based and used primarily within the construction process. Common examples of software construction tools include compilers, version control systems, design tools, and documentation tools.

9. The best software construction tools bridge the gap between methodical computer efficiency and forgetful human creativity. Such tools allow creative minds to express their thoughts easily, but also enforce a level of rigor that keeps that same creativity from seriously damaging the overall construction process. Good tools also improve quality by keeping people from doing repetitive or precise work for which a computer is better suited.

10. 1.3 The Role of Integrated Self-Evaluation in Construction

10. Another important theme of software engineering is the *integrated self-evaluation* of processes. This concept encompasses such diverse activities as testing, software quality assurance, and metrics (see *Knowledge Area Description for Testing* and *Knowledge Area Description for Software Quality*). Integrated self-evaluation means that a process (in this case a development process) includes explicit continuous or periodic internal "self-checks" to ensure that it is still working correctly. These self-checks usually consist of evaluations of intermediate work products such as documents, designs, source code, or compiled modules, but they may also look at characteristics of the development process itself. Examples of product evaluations include design reviews, module compilations, and unit tests. An example of process-level self-evaluation would be periodic re-assessment of a

code library to ensure its accuracy, completeness, and self-consistency.

11. Integrated self-evaluation in software engineering parallels the concept of integrated self-test logic and built-in error recovery in complex integrated circuits. Such features were first added to integrated circuits when it was realized the circuits had become so complex that the assumption of perfect start-to-finish reliability was no longer tenable. Similarly, software engineering processes and products have become so complex that even the illusion that they can move from start (requirements) to finish (delivery) without incurring significant serious process errors along the way is no longer plausible – and probably never was even in the earliest days of computing. As with integrated circuits, the purpose of integrated self-checking in software processes is to ensure that they can operate for long periods without generating nonsensical answers.

12. Historically, software construction has tended to be one of the software engineering steps in which developers were particularly prone to omitting self-checks of the process. While nearly all developers practice some degree of informal self-evaluation when constructing software, it is all too common for authors to skip needed self-evaluation steps because they are too confident about the reliability and quality of their own software constructions. Nonetheless, a wide range of automated, semi-automated, and manual self-evaluation methods have been developed for use in the software construction phase.

13. The simplest and best-known form of software construction self-evaluation is the use of unit testing after completion of each well-defined software unit. Automated techniques such as compile-time checks and run-time checks help verify the basic integrity of software units, and manual techniques such as code reviews can be used to search for more abstract classes of errors. Tools for extracting metrics on code quality and structure can also be used during construction, although such measurement tools are more commonly applied during integration of large suites of software units. When collecting metrics, it is particularly important that there be a well-defined link between the metrics that are collected and the self-evaluation goal that is being pursued.

14. 1.4 The Role of Standards in Construction

15. All forms of successful communication require a common language. *Standards* are in many ways best understood as agreements by which both concepts and technologies can become part of the shared "language" of a broader community of users (see *Software Evolution and Management*). While it is possible in principle to do software construction without adhering to any standard beyond the design of the computer hardware, such an approach would be very slow, remarkably painful, and phenomenally expensive. It would, for example, require at least the creation of a new computer language, of operating system software, of tools to support development, and of hardware drivers for all devices. A much more practical approach is to choose as broad and enduring a set of standards as possible. Determining what this set should be can be a difficult task, but it is one that is almost always worth the trouble in the long term.

16. Software construction is particularly sensitive to the selection of standards, which directly affects such construction-critical issues as programming languages, databases, communication methods, platforms, and tools. Although such choices are often made before construction begins, it is important that the overall software development process take the needs of construction into account when standards are selected.

17. 1.5 The Spectrum of Construction Techniques

18. Software construction techniques can be broadly grouped in terms of how they fall between two endpoints: manual construction techniques, and automated construction techniques.

	Manual Construction	Goal: →	**Automated Construction**
19.	Usually procedural (i.e., order-dependent)	→	Often non-procedural (e.g., descriptive)
20.	Very large number of descriptive options	→	Limited number of descriptive options
21.	Emphasis on finding new problem solutions	→	Emphasis on reusing old problem solutions
22.	Process is defined by user (versus by tools)	→	Process is defined mostly by the tools used

	Manual Construction	Goal: →	Automated Construction
23.	Expensive, risky, and usable by few people	→	Low-cost, safe, and usable by many people
24.	More likely to be defined by a standard	→	More likely to be custom to application area

25. *Manual Construction*

26. *Manual construction* means solving complex problems in a language that a computer can execute. Practitioners of manual construction need a rich mix of skills that includes the ability to break complex problems down into smaller parts, a disciplined formal-proof-like approach to problem analysis, and the ability to "forecast" how constructions will change over time. Expert manual constructors thus need not only the skills of advanced logicians, but also the ability to apply those skills within a complex, changing environment such as a computer or network.

27. It would be easy to directly equate manual construction to coding in a procedural programming language, but it would also be an incomplete definition. An effective manual construction process should result in code that fully and correctly processes data for its entire problem space, anticipates and handles all plausible (and some implausible) classes of errors, runs efficiently, and is structured to be resilient and easy-to-change over time. An inadequate manual construction process will in contrast result in code like an amateurish painting, with critical details missing and the entire construction stitched together poorly.

28. *Automated Construction*

29. While no form of software construction can be fully automated, much or all of the overall coordination of the software construction process can be moved from people to the computer– that is, overall control of the construction process can be largely automated. *Automated construction* thus refers to software construction in which an automated tool or environment is primarily responsible for overall coordination of the software construction process. This removal of overall process control can have a large impact on the complexity of the software construction process, since it allows human contributions to be divided up into much smaller, less complex "chunks" that require fewer problem solving skills to solve. Automated construction is also reuse-intensive construction, since by limiting human options it allows the controlling software to make more effective use of its existing store of effective software problem solutions.

30. In its most extreme form, automated construction consists of little more than configuring a predefined set of options. For example, an accounting application for small businesses might lead users through a series of questions that will result in a customized installation of the application. When compared to using manual construction for the same type of problem, this form of automated construction "swallows" huge chunks of the overall software engineering process and replaces them with automated selections that are controlled by the computer. Toolkits provide a less extreme example in which developers still have a great deal of control over the construction process, but that process has been greatly constrained and simplified by the use of predefined components with well-defined relationships to each other.

31. Automated construction is necessarily tool-intensive construction, since the objective is to move as much of the overall software development process as possible away from the human developer and into automated processes. Automated construction tools tend to take the form of program generators and fully integrated environments that can more easily provide automated control of the construction process. To be effective in coordinating activities, automated construction tools also need to have easy, intuitive interfaces.

32. *Moving Towards Automation*

33. As indicated by the table, an important goal of software engineering is to move construction continually towards higher levels of automation. That is, when selection from a simple set of options is all that is really required to make software work for a business or system, then the goal of software engineers should continually be to make their systems come as close to that level of simplicity as possible. This not only makes software more accessible, but also makes it safer and more reliable by removing a plethora of needless opportunities for error.

34. The concept of moving towards higher levels of construction automation is so fundamental to good design that it permeates nearly every aspect

of software construction. When simple selections from a list of options will not suffice, software engineers often can still develop application specific tool kits (that is, sets of reusable parts designed to work with each other easily) to provide a somewhat lesser level of control. Even fully manual construction reflects the theme of automation, since many coding techniques and good programming practices are intended to make code modification easier and more automated. For example, even a concept as simple as assigning a value to a constant at the beginning of a software module reflects the automation theme, since such constants "automate" the appropriate insertion of new values for the constant in the event that changes to the program are necessary. Similarly, the concept of class inheritance in object-oriented programming helps automate and enforce the conveyance of appropriate sets of methods into new, closely related or derived classes of objects.

35. 1.6 Computer Languages

36. Since the fundamental task of software construction is to communicate intent unambiguously between two very different types of entities (people and computers), it is not too surprising that the interface between the two is most commonly expressed as languages. The resulting *computer languages*, such as Ada, Python, Fortran, C, C++, Java, and Perl, are close enough in form to human languages to allow some "borrowing" of innate skills of programmers in natural languages such as English or French. However, computer languages are also very literal from the perspective of natural languages, since no computer yet built has sufficient context and understanding of the natural world to recognize invalid language statements and constructions that would be caught immediately in a natural language context. As will be discussed below, computer languages can also borrow from other non-linguistic human skills such as spatial visualization.

37. Computer languages are often created in response to the needs of particular application fields, but the quest for more universal or encompassing programming language is ongoing. As in many relatively young disciplines, such quests for universality are as likely to lead to short-lived fads as they are to genuine insights into the fundamentals of software construction. For this very reason, it is important that software construction not be tied too greatly on any programming language or programming methodology.

38. 1.7 Construction Languages

39. *Construction languages* include all forms of communication by which a human can specify an executable problem solution to a computer. The simplest type of construction language is a configuration language, in which developers choose from a limited set of predefined options to create new or custom installations of software. The text-based configuration files used in both Windows and Unix operating systems are examples, and the menu-style selection lists of some program generators are another. Toolkit languages are used to build applications out of toolkits (integrated sets of application-specific reusable parts), and are more complex than configuration languages. Toolkit languages be explicitly defined as application programming languages (e.g., scripts), or may simply be implied by the collected set of interfaces of a toolkit. As described earlier, computer languages are the most flexible type of construction languages, but they also contain the least information about both application areas and development processes, and so require the most training and skill to use effectively.

40. 2. Styles of Construction

41. A good construction language moves detailed, repetitive, or memory-intensive construction tasks away from people and into the computer, where such tasks can be performed faster and more reliably. To accomplish this, construction languages must present and receive information in ways that are readily understandable to human senses and capabilities. This need to rely on human capabilities leads to three major styles of software construction interfaces:

42. **A. Linguistic**: Linguistic construction languages make statements of intent in the form of sentences that resemble natural languages such as French or English. In terms of human senses, linguistic constructions are generally conveyed visually as text, although they can (and are) also sometimes conveyed by sound. A major advantage of linguistic construction interfaces is that they are nearly universal among people. A disadvantage is the imprecision of ordinary languages such a English, which makes it hard for people to express needs clearly with sufficient precision when using linguistic interfaces to

computers. An example of this problem is the difficulty that most early students of computer science have learning the syntax of even fairly readable languages such as Pascal, Ada, or Python.

43. **B. Formal:** The precision and rigor of formal and logical reasoning make this style of human thought especially appropriate for conveying human intent accurately into computers, as well as for verifying the completeness and accuracy of a construction. Unfortunately, formal reasoning is not nearly as universal a skill as natural language, since it requires both innate skills that are not as universal as language skills, and also many years of training and practice to use efficiently and accurately. It can also be argued that certain aspects of good formal reasoning, such as the ability to realize all the implications of a new assertion on all parts of a system, cannot be learned by some people no matter how much training they receive. On the other hand, formal reasoning styles are often notorious for focusing on a problem so intently that all "complications" are discarded and only a very small, very pristine subset of the overall problem is actually addressed. This kind of excessively narrow focus at the expense of any complicating issues can be disastrous in software construction, since it can lead to software that is incapable of dealing with the unavoidable complexities of nearly any usable system.

44. **C. Visual:** Another very powerful and much more universal construction interface style is *visual*, in the sense of the ability to use the same very sophisticated and necessarily natural ability to "navigate" a complex three-dimensional world of images, as perceived primarily through the eye (but also through tactile senses). The visual interface is powerful not only as a way of organizing information for presentation to a human, but also as a way of conceiving and navigating the overall design of a complex software system. Visual methods are particularly important for systems that require many people to work on them – that is, for organizing a software design process – since they allow a natural way for people to "understand" how and where they must communicate with each other. Visual methods are also deeply important for single-person software construction methods, since they provide ways both to present options to people and to make key details of a large body of information "pop out" to the visual system.

45. Construction languages seldom rely solely on a single style of construction. Linguistic and formal style in particular are both heavily used in most traditional computer languages, and visual styles and models are a major part of how to make software constructions manageable and understandable in computer languages. Relatively new "visual" construction languages such as Visual Basic and Visual Java provide examples that intimately combine all three styles, with complex vi sual interfaces often constructed entirely through non-textual interactions with the software constructor. Data processing functionality behind the interfaces can then be constructed using more traditional linguistic and formal styles within the same construction language.

46. 3. Principles of Organization

47. In addition to the three basic human-oriented styles of interfacing to computers, there are four *principles of organization* that strongly affect the way software construction is performed. These principles are:

48. **Reduction of Complexity:** This principle of organization reflects the relatively limited ability of people to work with complex systems that have many parts or interactions. A major factor in how people convey intent to computers is the severely limited ability of people to "hold" complex structures and information in their working memory, especially over long periods of time. This need for simplicity in the human-to-computer interface leads to one of the strongest drivers in software construction: *reduction of complexity*. The need to reduce complexity applies to essentially every aspect of the software construction, and is particularly critical to the process of self-verification and self-testing of software constructions.

49. There are three main techniques for reducing complexity during software construction:

50. **Removal of Complexity:** Although trivial in concept, one obvious way to reduce complexity during software construction is to *remove* features or capabilities that are not absolutely required. This may or may not be the right way to handle a given situation, but certainly the general principle of parsimony – that is, of not

adding capabilities that clearly will never be needed when constructing software – is valid.

51. **Automation of Complexity:** A much more powerful technique for removal of complexity is to *automate* the handling of it. That is, a new construction language is created in which features that were previously time-consuming or error-prone for a human to perform are migrated over to the computer in the form of new software capabilities. The history of software is replete with examples of powerful software tools that raised the overall level of development capability of people by allowing them to address a new set of problems. Operating systems are one example of this principle, since they provide a rich construction language by which efficient use of underlying hardware resources can be greatly simplified. Visual construction languages similarly provide automation of visual aspects of software that otherwise could be very laborious to build.

52. **Localization of Complexity:** If complexity can neither be removed nor automated, the only remaining option is to *localize* complexity into small "units" or "modules" that are small enough for a person to understand in their entirety, and (perhaps more importantly) sufficiently *isolated* that meaningful assertions can be made about them. This might even lead to components that can be re-used. However, one must be careful, as arbitrarily dividing a very long sequence of code into small "modules" does not help, because the relationships between the modules become extremely complex and difficult to predict. Localization of complexity has a powerful impact on the design of computer languages, as demonstrated by the growth in popularity of object-oriented methods that seek to strictly limit the number of ways to interface to a software module, even though that might end up making components more dependent. Localization is also a key aspect of good design of the broader category of construction languages, since new feature that are too hard to find and use are unlikely to be effective as tools for construction. Classical design admonitions such as the goal of having "cohesion" within modules and to minimize "coupling" are also fundamentally localization of complexity techniques, since they strive to make the number and interaction of parts within a module easy for a person to understand.

53. **Anticipation of Diversity:** This principle has more to do with how people use software than with differences between computers and people. Its motive is simple: *There is no such thing as an unchanging software construction.* Any truly useful software construction will change in various ways over time, and the *anticipation* of what those changes will be turns out to be one of the fundamental drivers of nearly every aspect of software construction. Useful software constructions are unavoidably part of a changing external environment in which they perform useful tasks, and changes in that outside environment trickle in to impact the software constructions in diverse (and often unexpected) ways. In contrast, formal mathematical constructions and formulas can in some sense be stable or unchanging over time, since they represent abstract quantities and relationships that do not require direct "attachment" to a working, physical computational machine. For example, even the software implementations of "universal" mathematical functions must change over time due to external factors such as the need to port them to new machines, and the unavoidable issue of physical limitations on the accuracy of the software on a given machine.

54. Anticipation of the diversity of ways in which software will change over time is one of the more subtle principles of software construction, yet it is vitally important for the creation of software that can endure over time and add value to future endeavors. Since it includes the ability to anticipate changes due to design errors (bugs) in software, it is also a fundamental part of the ability to make software robust and error-free. Indeed, one handy definition of "aging" software is that it is software that no longer has the flexibility to accommodate bug fixes without breaking.

55. There are three main techniques for anticipating change during software construction:

56. **Generalization:** It is very common for software construction to focus first on highly specific problems with limited, rather specific solutions. This is common because the more general cases often simply are not obvious in the early stages of analysis. Generalization is the process of recognizing how a few specific problem cases fit together as part of some broader framework of problems, and thus can be solved by a single overarching software construction in place of several isolated ones. Generalization of functionality is a distinctly mathematical concept, and not too surprisingly the best generalizations that are developed are often expressed in the language of mathematics. Good design is equally an aspect of generalization,

however. For example, software constructions that use stacks to store data are almost always more generalized than similar solutions using fixed-sized arrays, since fixed sizes immediately place artificial (and usually unnecessary) constraints on the range of problem sizes that the construction can solve.

57. Generalization anticipates diversity because it creates solutions to entire classes of problems that may not have even been recognized as existing before. Thus just as Newton's general theory of gravity made a small number of formulas applicable to a much broader range of physics problems, a good generalization to a number of discrete software problems often can lead to the easy solution of many other development problems. For example, developing an easily customizable graphics user interface could solve a very broad range of development problems that otherwise would have required individual, labor-intensive development of independent solutions.

58. The greatest difficulty with generalization as a technique for anticipating diversity is that it depends very strongly on the ability of the individual developer to find generalizations that actually correspond to the eventual uses of the software. Developers may have no particular interest (or time) to develop the necessary generalizations under the schedule pressures of typical commercial projects. Even when the time needed is available, it is surprisingly easy to develop the wrong set of generalizations – that is, to create generalizations that make the software easier to change, but only in ways that prove not to correspond to what is really needed.

59. For these reasons, generalization is both safer and easier if it can be combined with the next technique of *experimentation.* Change experimentation makes generalization safer by capturing realistic data on which generalizations will be needed, and makes generalization easier by providing schedule-conscious projects with specific data on how generalizations can improve their products.

60. **Experimentation:** *Experimentation* means using early (sometimes very early) software constructions in as many different user contexts as possible, and as early in the development process as possible, for the explicit purpose of collecting data on how to generalize the construction. Experimentation effectively acknowledges the sizable difficulty of anticipating all the ways in which software constructions can change, and uses experimentation to fill the gap in knowledge.

61. Obviously, experimentation is a process-level technique rather than a code-level technique, since its goal is to collect data to help guide code-level processes such as generalization. This means that it is constrained by whether the overall development process allows it to be used at the construction level. Construction-level experimentation is most likely to be found in projects that have incorporated experimentation into their overall development process. The Internet-based open source development process that Linus Torvalds used to create the Linux operating system is an example of a process that both allowed and encouraged construction-level use of experimentation. In Torvalds' approach, individual code constructions were very quickly incorporated into an overall product and then redistributed via the Internet, sometimes on the same day. This encouraged further use, experimentation, and updates to the individual constructions.

62. **Localization:** *Localization* means keeping anticipated changes as localized in a software construction as possible. It is actually a special case of the earlier principle of *localization of complexity,* since change is a particularly difficult class of complexity. A software construction that can be changed in a common way by making only one change at one location within the construction thus demonstrates good *locality* for that particular class of modifications.

63. Localization is very common in software construction, and often is used intuitively as the "right way" to construct software. Objects are one example of a localization technique, since good object designs localize implementation changes to within the object. An even simpler example is using compile-time constants to reduce the number of locations in a program that must be changed manually should the constant change. Layered architectures such as those used in communication protocols are yet another example of localization, since good layer designs keep changes from crossing layers.

64. **Structuring for Validation:** No matter how carefully a person designs and implements software, the creative nature of non-trivial software construction (that is, of software that is not simply a re-implementation of previously solved problems) means that mistakes and omissions will occur. *Structuring for validation* means building software in such a fashion that

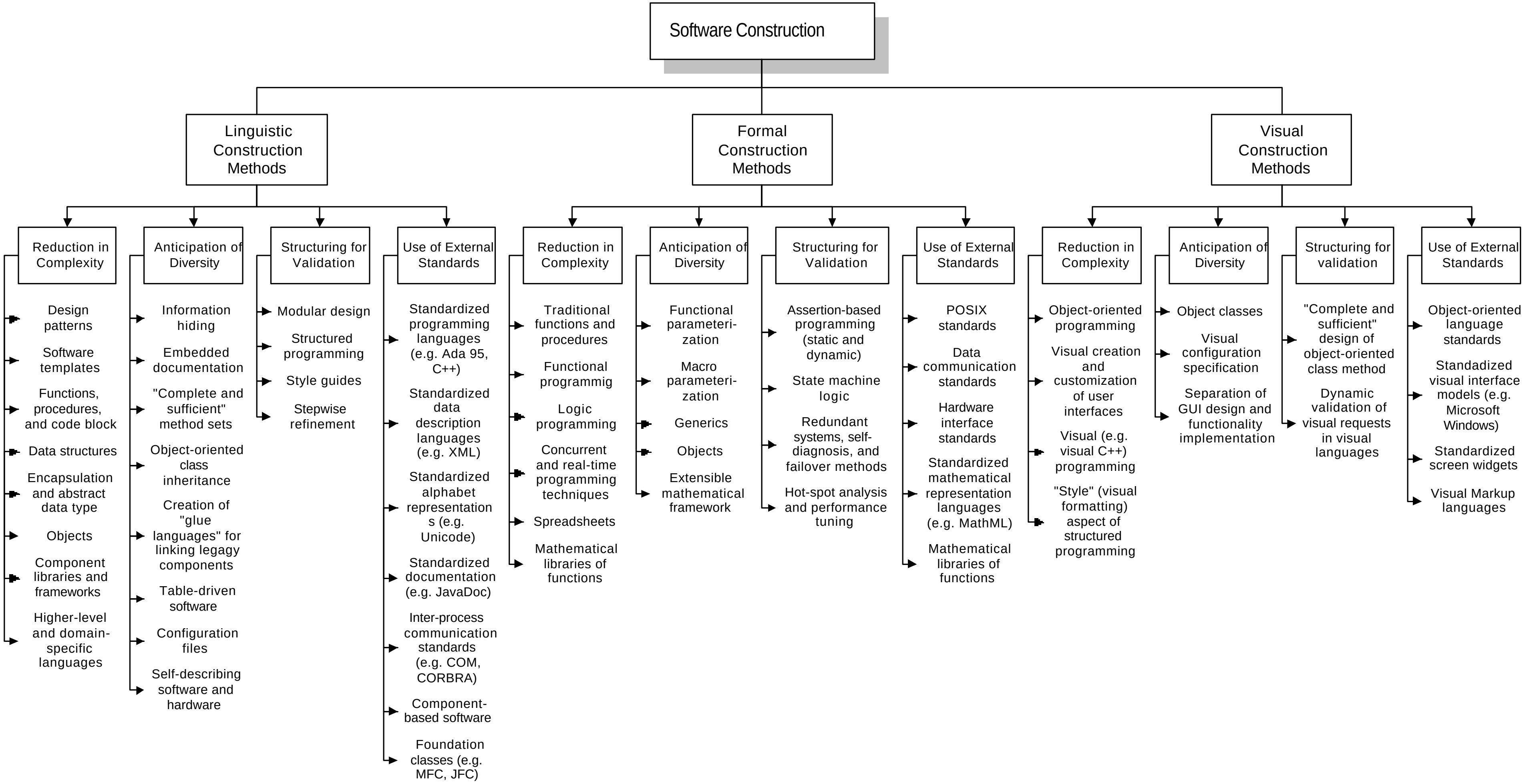
such errors and omissions can be ferreted out more easily during unit testing and subsequent testing activities. One of the single most important implications of structuring for validation is that software must generally be *modular* in at least one of its major representation spaces, such as in the overall layout of the displayed or printed text of a program. This modularity allows both improved analysis and thorough unit-level testing of such components before they are integrated into higher levels in which their errors may be more difficult to identify. As a principle of construction, structuring for validation generally goes hand-in-hand with anticipation of diversity, since any errors found as a result of validation represent an important type of "diversity" that will require software changes (bug fixes). It is not particularly difficult to write software that cannot really be validated no matter how much it is tested. This is because even moderately large "useful" software components frequently cover such a large range of outputs that exhaustive testing of all possible outputs would take eons with even the fastest computers. *Structuring for validation* thus becomes a fundamental constraint for producing software that can be shown to be acceptably reliable within a reasonable time frame. The concept of *unit testing* parallels structuring for validation, and is used in parallel with the construction process to help ensure that validation occurs before the overall structure gets "out of hand" and can no longer be readily validated.

65. **Use of External Standards:** A natural language that is spoken by one person would be of little value in communicating with the rest of the world. Similarly, a construction language that has meaning only within the software for which it was constructed can be a serious roadblock in the long-term use of that software. Such construction languages therefore should either conform to *external standards* such as those used for computer languages, or provide a sufficiently detailed internal "grammar" (e.g., documentation) by which the construction language can later be understood by others. The interplay between reusing external standards and creating new ones is a complex one, as it depends not only on the availability of such standards, but also on realistic assessments of the long-term viability of such external standards. With the advent of the Internet as a major force in software development and interaction, the importance of selecting and using appropriate external standards for how to construct software is more apparent than ever before. Software that must share data and even working modules with other software anywhere in the world obviously must "share" many of the same languages and methods as that other software. The result is that selection and use of external standards – that is, of standards such as language specifications and data formats that were not originated within a software effort – is becoming an even more fundamental constraint on software construction than it was in the past. It is a complex issue, however, because the selection of an external standard may need to depend on such difficult-to-predict issues as the long-term economic viability of a particular software company or organization that promotes that standard. Stability of the standard is especially important. Also, selecting one level of standardization often opens up an entire new set of standardization issues. An example of this is the data description language XML (eXtensible Markup Language). Selecting XML as an external standard answers many questions about how to describe data in an application, but it also opens up the issue of whether one of the growing numbers of customizations of XML to specific problem domains should also be used.

66. A Taxonomy of Software Construction Methods

67. Let us begin by stating that it is not possible to create a taxonomy of software construction methods that provides much insight into the relationships of software construction methods. The problem is that traditional taxonomies use exclusive tree structures to place each item in a unique position on the tree. However, techniques such as modularity are often so pervasive in their impacts on software construction that any attempt to force them into a single category of a taxonomic breakdown will result in a taxonomy that fails to explain the breadth of impact.

68. For this reason the taxonomy given here is more properly understood as a *taxonomy of principles* to which the impacts of individual construction methods can be mapped. In this taxonomy, an individual construction method may show up in many different locations in the taxonomy, rather than simply in one location. The number of locations in which a method shows up indicates its breadth of application, and thus an indication of its importance to software construction as a whole. Modularity is one example of a construction method that has such broad impacts.


Software Construction
Linguistic Construction Methods
Formal Construction Methods
Visual Construction Methods
Reduction in Complexity
Design patterns
Software templates
Functions, procedures, and code block
Data structures
Encapsulation and abstract data type
Objects
Component libraries and frameworks
Higher-level and domain-specific languages
Anticipation of Diversity
Information hiding
Embedded documentation
"Complete and sufficient" method sets
Object-oriented class inheritance
Creation of "glue languages" for linking legagy components
Table-driven software
Configuration files
Self-describing software and hardware
Structuring for Validation
Modular design
Structured programming
Style guides
Stepwise refinement
Use of External Standards
Standardized programming languages (e.g. Ada 95, C++)
Standardized data description languages (e.g. XML)
Standardized alphabet representations (e.g. Unicode)
Standardized documentation (e.g. JavaDoc)
Inter-process communication standards (e.g. COM, CORBRA)
Component-based software
Foundation classes (e.g. MFC, JFC)
Reduction in Complexity
Traditional functions and procedures
Functional programmig
Logic programming
Concurrent and real-time programming techniques
Spreadsheets
Mathematical libraries of functions
Anticipation of Diversity
Functional parameterization
Macro parameterization
Generics
Objects
Extensible mathematical framework
Structuring for Validation
Assertion-based programming (static and dynamic)
State machine logic
Redundant systems, self-diagnosis, and failover methods
Hot-spot analysis and performance tuning
Use of External Standards
POSIX standards
Data communication standards
Hardware interface standards
Standardized mathematical representation languages (e.g. MathML)
Mathematical libraries of functions
Reduction in Complexity
Object-oriented programming
Visual creation and customization of user interfaces
Visual (e.g. visual C++) programming
"Style" (visual formatting) aspect of structured programming
Anticipation of Diversity
Object classes
Visual configuration specification
Separation of GUI design and functionality implementation
Structuring for validation
"Complete and sufficient" design of object-oriented class method
Dynamic validation of visual requests in visual languages
Use of External Standards
Object-oriented language standards
Standadized visual interface models (e.g. Microsoft Windows)
Standardized screen widgets
Visual Markup languages

69. A. Linguistic Construction Methods

70. *Linguistic construction methods* are distinguished in particular by the use of word-like strings of text to represent complex software constructions, and the combination of such word-like strings into patterns that have a sentence-like syntax. Properly used, each such string should have a strong semantic connotation that provides an immediate intuitive understanding of what will happen when the underlying software construction is executed. For example, the term "search" has an immediate, readily understandable semantic meaning in English, yet the underlying software implementation of such a term in software can be very complex indeed. The most powerful linguistic construction methods allow users to focus almost entirely on the language-like meanings of such term, as opposed (for example) to frittering away mental efforts on examining minor variations of what "search" means in a particular context.

71. Linguistic construction methods are further characterized by similar use of other "natural" language skills such as using patterns of words to build sentences, paragraphs, or even entire chapters to express software design "thoughts." For example, a pattern such as "search table for out-of-range values" uses word-like text strings to imitate natural language verbs, nouns, prepositions, and adjectives. Just as having an underlying software structure that allows a more natural use of words reduces the number of issues that a user must address to create new software, an underlying software structure that also allows use of familiar higher-level patterns such as sentence further simplifies the expression process.

72. Finally, it should be noted that as the complexity of a software expression increases, linguistic construction methods begin to overlap unavoidably with visual methods that make it easier to locate and understand large sequences of statements. Thus just as most written versions of natural languages use visual clues such as spaces between words, paragraphs, and section headings to make text easier to "parse" visually, linguistic construction methods rely on methods such as precise indentation to convey structural information visually.

73. The use of linguistic construction methods is also limited by our inability to program computers to understand the levels of ambiguity typically found in natural languages, where many subtle issues of context and background can drastically influence interpretation. As a result, the linguistic model of construction usually begins to weaken at the more complex levels of construction that correspond to entire paragraphs and chapters of text.

74. 1. Reduction in Complexity (Linguistic)

75. The main technique for reducing complexity in linguistic construction is to make short, semantically "intuitive" text strings and patterns of text stand in for the much more complex underlying software that "implement" the intuitive meanings. Techniques that reduce complexity in linguistic construction include:

76. ◆ Design patterns

77. ◆ Software templates

78. ◆ Functions, procedures, and code blocks

79. ◆ Data structures

80. ◆ Encapsulation and abstract data types

81. ◆ Objects

82. ◆ Component libraries and frameworks

83. ◆ Higher-level and domain-specific languages

84. 2. Anticipation of Diversity (Linguistic)

85. Linguistic construction anticipates diversity both by permitting extensible definitions of "words," and also by supporting flexible "sentence structures" that allow many different types of intuitively understandable statements to be made with the available vocabulary. An excellent example of using linguistic construction to anticipate diversity is the use of human-readable configuration files to specify software or system settings.

86. ◆ Information hiding

87. ◆ Embedded documentation (commenting)

88. ◆ "Complete and sufficient" method sets

89. ◆ Object-oriented class inheritance

90. ◆ Creation of "glue languages" for linking legacy components

91. ◆ Table-driven software

92. - Configuration files

93. - Self-describing software and hardware (e.g., plug and play)

94. 3. Structuring for Validation (Linguistic)

95. Because natural language in general is too ambiguous to allow safe interpretation of completely free-form statements, structuring for validation shows up primarily as rules that at least partially constrain the free use of natural expressions in software. The objective is to make such constructions as "natural" sounding as possible, while not losing the structure and precision needed to ensure consistent interpretations of the source code by both human users and computers.

96. - Modular design

97. - Structured programming

98. - Style guides

99. - Stepwise refinement

100. 4. Use of External Standards (Linguistic)

101. Traditionally, standardization of programming languages was one of the first areas in which external standards appeared. The goal was (and is) to provide standard meanings and ways of using "words" in each standardized programming language, which makes it possible both for users to understand each other's software, and for the software to be interpreted consistently in diverse environments.

102. - Standardized programming languages (e.g., Ada 95, C++, etc.)

103. - Standardized data description languages (e.g., XML)

104. - Standardized alphabet representations (e.g., Unicode)

105. - Standardized documentation (e.g., JavaDoc)

106. - Inter-process communication standards (e.g., COM, CORBA)

107. - Component-based software

108. - Foundation classes (e.g., MFC, JFC)

109. B. Formal Construction Methods

110. *Formal construction methods* rely less on intuitive, everyday meanings of words and text strings, and more on definitions that are backed up by precise, unambiguous, and fully formal (or mathematical) definitions. Formal construction methods are at the heart of most forms of system programming, where precision, speed, and verifiability are more important than ease of mapping into ordinary language. Formal constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions. Functions are an obvious example of formal constructions, with their direct parallel to mathematical functions in both form and meaning.

111. Formal construction techniques also include the wide range of precisely defined methods for representing and implementing "unique" computer problems such as concurrent and multi-threaded programming, which are in effect classes of mathematical problems that have special meaning and utility within computers.

112. The importance of the formal style of programming cannot be understated. Just as the precision of mathematics is fundamental to disciplines such as physics and the hard science, the formal style of programming is fundamental to building up a reliable framework of software "results" that will endure over time. While the linguistic and visual styles work well for interfacing with people, these less precise styles can be unsuitable for building the interior of a software system for the same reason that stained glass should not be used to build the supporting arches of a cathedral. Formal construction provides a foundation that can eliminate entire classes of errors or omissions from ever occurring, whereas linguistic and visual construction methods are much more likely to focus on isolated instances of errors or omissions. Indeed, one very real danger in software quality assurance is to focus too much on capturing isolated errors occurring in the linguistic or visual modes of construction, while overlooking the much more grievous (but harder to identify and understand) errors that occur in the formal style of construction.

113. 1. Reduction in Complexity (Formal)

114. As is the case with linguistic construction methods, formal construction methods reduce complexity by representing complex software constructions as simple text strings. The main difference is that in this case the text strings follow the more precisely defined rules and syntax of formal notations, rather than the "fuzzier" rules of natural language. The reading, writing, and construction of such expressions requires generally more training, but once mastered, the use of formal

constructions tends to keep the ambiguity of what is being specified to an absolute minimum. However, as with linguistic construction, the quality of a formal construction is only as good as its underlying implementation. The advantage is that the precision of the formal definitions usually translates into a more precise specification for the software beneath it.

115. ◆ Traditional functions and procedures

116. ◆ Functional programming

117. ◆ Logic programming

118. ◆ Concurrent and real-time programming techniques

119. ◆ Spreadsheets

120. ◆ Mathematical libraries of functions

121. 2. Anticipation of Diversity (Formal)

122. Diversity in formal construction is handled in terms of precisely defined sets that can vary greatly in size. While mathematical formalizations are capable of very flexible representations of diversity, they require explicit anticipation and preparation for the full range of values that may be needed. A common problem in software construction is to use a formal technique – e.g., a fixed-length vector or array – when what is really needed to accommodate future diversity is a more generic solution that anticipates future growth – e.g., an indefinite variable-length vector. Since more generic solutions are often harder to implement and harder to make efficient, it is important when using formal construction techniques to try to anticipate the full range of future versions.

123. ◆ Functional parameterization

124. ◆ Macro parameterization

125. ◆ Generics

126. ◆ Objects

127. ◆ Extensible mathematical frameworks

128. 3. Structuring for Validation (Formal)

129. Since mathematics in general is oriented towards proof of hypothesis from a set of axioms, formal construction techniques provide a broad range of techniques to help validate the acceptability of a software unit. Such methods can also be used to "instrument" programs to look for failures based on sets of preconditions.

130. ◆ Assertion-based programming (static and dynamic)

131. ◆ State machine logic

132. ◆ Redundant systems, self-diagnosis, and failover methods

133. ◆ Hot-spot analysis and performance tuning

134. 4. Use of External Standards(Formal)

135. For formal construction techniques, external standards generally address ways to define precise interfaces and communication methods between software systems and the machines they reside on.

136. ◆ POSIX standards

137. ◆ Data communication standards

138. ◆ Hardware interface standards

139. ◆ Standardized mathematical representation languages (e.g., MathML)

140. ◆ Mathematical libraries of functions

141. C. Visual Construction Methods

142. Visual construction methods rely much less on the text-oriented constructions of both linguistic and formal construction, and instead rely on direct visual interpretation and placement of visual entities (e.g., "widgets") that represent the underlying software. Visual construction tends to be somewhat limited by the difficulty of making "complex" statements using only movement of visual entities on a display. However, it can also be a very powerful tool in cases where the primary programming task is simply to build and "tweak" a visual interface to a program whose detailed behavior was defined earlier.

143. Object-oriented languages are an interesting case. Although object-oriented languages use text and words to describe the detailed properties of objects, the style of reasoning that they encourage is highly visual. For example, experienced object-oriented programmers tend to view their designs literally as objects interacting in spaces of two or more dimensions, and a plethora of object-oriented design tools and techniques (e.g., Universal Modeling Language, or UML) actively encourage this highly visual style of reasoning.

144. However, object-oriented methods can also suffer from the lack of precision that is part of the more intuitive visual approach. For example, it is common for new – and sometimes not-so-new – programmers in object-oriented languages to define object classes that lack the formal precision that will allow them to work reliably

over user-time (that is, long-term system support) and user-space (e.g., relocation to new environments). The visual intuitions that object-oriented languages provide in such cases can be somewhat misleading, because they can make the real problem of how to define a class to be efficient and stable over user-time and user-space seem to be simpler than it really is. A complete object-oriented construction model therefore must explicitly identify the need for formal construction methods throughout the object design process. The alternative can be an object-based system design that, like a complex stained glass window, looks impressive but is too fragile to be used in any but the most carefully designed circumstances.

145. More explicitly visual programming methods such as those found in Visual C++ and Visual Basic reduce the problem of how to make precise visual statements by "instrumenting" screen objects with complex (and formally precise) objects that lie behind the screen representations. However, this is done at a substantial loss of generality when compared to using C++ with explicit training in both visual and formal construction, since the screen objects are much more tightly constrained in properties.

146. 1. Reduction in Complexity (Visual)

147. Especially when compared to the steps needed to build a graphical interface to a program using text-oriented linguistic or formal construction, visual construction can provide drastic reductions in the total effort required. It can also reduce complexity by providing a simple way to select between the elements of a small set of choices.

148. ◆ Object-oriented programming

149. ◆ Visual creation and customization of user interfaces

150. ◆ Visual (e.g., visual C++) programming

151. ◆ "Style" (visual formatting) aspects of structured programming

152. 2. Anticipation of Diversity (Visual)

153. Provided that the total sets of choices are not overly large, visual construction methods can provide a good way to configure or select options for software or a system. Visual construction methods are analogous to linguistic configuration files in this usage, since both provide easy ways to specify and interpret configuration information.

154. ◆ Object classes

155. ◆ Visual configuration specification

156. ◆ Separation of GUI design and functionality implementation

157. 3. Structuring for Validation (Visual)

158. Visual construction can provide immediate, active validation of requests and attempted configurations when the visual constructs are "instrumented" to look for invalid feature combinations and warn users immediately of what the problem is.

159. ◆ "Complete and sufficient" design of object-oriented class methods

160. ◆ Dynamic validation of visual requests in visual languages

161. 4. Use of External Standards (Visual)

162. Standards for visual interfaces greatly ease the total burden on users by providing familiar, easily understood "look and feel" interfaces for those users.

163. ◆ Object-oriented language standards

164. ◆ Standardized visual interface models (e.g., Microsoft Windows)

165. ◆ Standardized screen widgets

166. ◆ Visual Markup Languages

167. Recommended References

168. [BEN00] Bentley, Jon, *Programming Pearls (Second Edition).* Addison-Wesley, 2000. (Chapters 2, 3, 4, 11, 13 14)

169. [BOO94] Booch, Grady, and Bryan, Doug, *Software Engineering with Ada (Third edition).* Benjamin/Cummings, 1994. (Parts II, IV, V)[HOR99] Horrocks, Ian, *Constructing the User Interface with Statecharts.* Addison-Wesley, 1999. (Parts II, IV)

170. [KER99] Kernighan, Brian W., and Pike, Rob, *The Practice of Programming.* Addison-Wesley, 1999. (Chapters 1, 2, 3, 5, 6, 9)

171. [MAG93] Maguire, Steve, *Writing Solid Code.* Microsoft Press, 1993.

172. [McCO93] McConnell, Steve, *Code Complete.* Microsoft Press, 1993.

173. [MEY97] Meyer, Bertrand, *Object-Oriented Software Construction (Second Edition).* Prentice-Hall, 1997. (Chapters 6, 10, 11)

174. [SET96] Sethi, Ravi, *Programming Languages–Concepts & Constructs (Second Edition).* Addison-Wesley, 1996. (Parts II, III, IV, V)

175. [WAR99] Warren, Nigel, and Bishop, Philip, *Java in Practice – Design Styles and Idioms for Effective Java*. Addison-Wesley, 1999. (Chapters 1, 2, 3, 4, 5, 10)

176. Further Readings

177. [BAR98] Barker, Thomas T., Writing Software Documentation – A Task-Oriented Approach. Allyn & Bacon, 1998.

178. [FOW99] Fowler, Martin, Refactoring – Improving the Design of Existing Code. Addison-Wesley, 1999.

179. [GLA95] Glass, Robert L., Software Creativity. Prentice-Hall, 1995.

180. [HEN97] Henricson, Mats, and Nyquist, Erik, Industrial Strength C++. Prentice-Hall, 1997.

181. [HOR99] Horrocks, Ian, Constructing the User Interface with Statecharts. Addison-Wesley, 1999.

182. [HUM97] Humphrey, Watts S., Introduction to the Personal Software Process. Addison-Wesley, 1997.

183. [HUN00] Hunt, Andrew, and Thomas, David, The Pragmatic Programmer. Addison-Wesley, 2000.

184. [MAZ96] Mazza, C., et al., Software Engineering Guides. Prentice-Hall, 1996. (Part IV)

185. Standards Relevant to Software Constructions

186. IEEE Std 829-1983 (Reaff 1991), IEEE Standard for Software Test Documentation (ANSI)

187. IEEE Std 1008-1987 (Reaff 1993), IEEE Standard for Software Unit Testing (ANSI)

188. IEEE Std 1028-1988 (Reaff 1993), IEEE Standard for Software Reviews and Audits (ANSI)

189. IEEE Std 1063-1987 (Reaff 1993), IEEE Standard for Software User Documentation (ANSI)

190. IEEE Std 1219-1992, IEEE Standard for Software Maintenance (ANSI)

191. Matrix of Reference Material versus Topics

Topics	Proposed reference material
192. **Software Construction and Software Design**	[GLA95] Part III, IV [MAZ96] Part IV [McCO93] Chap. 1, 2, 3
193. **The Role of Tools in Construction**	[HUN00] Chap. 3 [MAG93] Chap. 4 [MAZ96] Part IV [McCO93] Chap. 20
194. **The Role of Integrated Self-Evaluation in Construction**	[HUM97] [MAG93] Chap. 8 [McCO93] Chap. 31, 32, 33
195. **The Role of Standards in Construction**	[IEEE]
196. **The Spectrum of Construction Techniques**	[HUN00] Chap. 3
197. **Computer Languages**	[SET96]
198. **Construction Languages**	[HUN00] Chap. 3 [SET96]
199. **A. Linguistic Construction Methods**	[SET96] Part II
200. 1. Reduction in Complexity (Linguistic)	[BEN00] Chap. 2, 3 [KER99] Chap. 2, 3 [McCO93] Chap. 4 to 19
201. 2. Anticipation of Diversity (Linguistic)	[BOO94] Part VI [McCO93] Chap. 30
202. 3. Structuring for Validation (Linguistic)	[BEN00] Chap. 4 [KER99] Chap. 1, 5, 6 [MAG93] Chap. 2, 5, 7 [McCO93] Chap. 23, 24, 25, 26
203. 4. Use of External Standards (Linguistic)	http://www.xml.org/ http://www.omg.org/corba/beginners.html
204. **B. Formal Construction Methods**	[SET96] Part IV and V
205. 1. Reduction in Complexity (Formal)	[BOO94] Part II and V [MAG93] Chap. 6 [MEY97] Chap. 6, 10
206. 2. Anticipation of Diversity (Formal)	[BEN00] Chap. 11, 13, 14 [KER99] Chap. 2, 9
207. 3. Structuring for Validation (Formal)	[MAG93] Chap. 3 [MEY97] Chap. 6, 11
208. 4. Use of External Standards (Formal)	Object Constraint Language: http://www.omg.org/uml/
209. **C. Visual Construction Methods**	[SET96] Part III
210. 1. Reduction in Complexity (Visual)	[HOR99] Part II [WAR99] Chap. 1, 2, 3, 4, 5, 10
211. 2. Anticipation of Diversity (Visual)	[WAR99] Chap. 1, 2, 3, 4, 5, 10
212. 3. Structuring for Validation (Visual)	[HOR99] Part IV [MEY97] Chap. 11
213. 4. Use of External Standards (Visual)	http://www.omg.org/uml/

CHAPTER 5
SOFTWARE TESTING

A. Bertolino
Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche
Pisa Research Area
Via Alfieri, 1, 56010 S. Giuliano Terme - PISA (Italy)
bertolino@iei.pi.cnr.it

TABLE OF CONTENTS

1. 1. INTRODUCTION

2. Testing is an important, mandatory part of software development, for improving and evaluating product quality.

3. In the *Software Quality* (SQ) Knowledge Area of the Guide to the SWEBOK, activities and techniques for quality analysis are categorized into*: static techniques* (no code execution), and *dynamic techniques* (code execution). Both categories are useful. Although this chapter focuses on testing, that is dynamic (see Sect. 2), static techniques are as important for the purpose of building quality in a software product. Static techniques are covered into the SQ Knowledge Area description.

4. In the years, the view of Software Testing has evolved towards a more constructive attitude. Testing is no longer seen as an activity that starts only after the coding phase is complete, with the limited purpose of detecting failures. Software testing is nowadays seen as an activity that encompasses the whole development process, and is an important part itself of the actual product construction. Indeed, planning for testing should start since the early stages of requirement analysis, and test plans and procedures must be systematically and continuously refined as the development proceeds. These activities of planning and designing tests constitute themselves a useful input to designers for highlighting potential weaknesses.

5. As more extensively discussed in the SQ Knowledge Area, the right attitude towards quality is one of prevention: it is obviously much better to avoid problems, rather than repairing them. Testing must be seen as a means primarily for demonstrating that the prevention has been effective, but also for identifying anomalies in those cases in which, for some reason, it has been not. Finally, it is worth recognizing that even after a good testing, the software could still contain faults. The remedy to system failures that are experienced after delivery is provided by (corrective) maintenance actions. Maintenance topics are covered into the *Software Maintenance* chapter of the Guide to the SWEBOK.

6. 2. DEFINITION OF THE SOFTWARE TESTING KNOWLEDGE AREA

7. Software testing consists of the <u>dynamic</u> verification of the behavior of a program on a <u>finite</u> set of test cases, suitably <u>selected</u> from the usually infinite executions domain, against the specified <u>expected</u> behavior.

8. In the above definition, and in the following as well, underlined words correspond to key issues in identifying the Knowledge Area of Software Testing. In particular:

9. - dynamic: this term means *testing always implies executing the program on valued inputs*. Static analysis techniques, such as peer review and inspection (that sometimes are improperly referred to as "static testing"), are not considered as part of this Knowledge Area (nor is program execution on symbolic inputs, or symbolic evaluation);

10. - finite: clearly the tests are not selected literally from an infinite set of possible tests, but a very large finite one (the set of all bit strings shorter than some length). Nonetheless, for even simple programs, so many test cases are theoretically possible that exhaustive testing could require even years to execute. This is why in practice the number of tests can generally be considered infinite. However, t he number of executions which can realistically be observed in testing must obviously be finite, and -more than this- it must be manageable. Indeed, testing always implies a *trade-off* between limited resources and schedules, and inherently unlimited test requirements: this conflict points to well known problems of testing, both technical in nature (criteria for deciding test adequacy) and managerial in nature (estimating the effort to put in testing);

11. - selected: the many proposed *test techniques* essentially differ in how they select the (finite) test set, and testers must be aware that different selection criteria may yield largely different effectiveness. The problem of identifying the most suitable selection criterion under given conditions is still under research;

12. - expected: it must be possible to decide whether the observed outcomes of program execution are acceptable or not, otherwise the testing effort would be useless. The observed behavior may be checked against user's expectations (commonly referred to as testing for *validation*) or against a functional specification (testing for *verification*). The test pass/fail decision is referred to as the *oracle problem*, which can be addressed with different approaches.

13. 2.1 Conceptual Structure of the Breakdown

14. Software testing is usually performed at different levels along the development process. That is to say, the object of the test can vary: a whole program, part of it (functionally or structurally related), a single module.

15. The testing is conducted in view of a specific purpose (test objective), which is stated more or less explicitly, and with varying degrees of precision. Stating the objective in precise, quantitative terms allows for establishing control over the test process.

16. One of testing aims is to expose failures (as many as possible), and many popular test techniques have been developed for this objective. These techniques variously attempt to "break" the program, by running identified classes of (deemed equivalent) executions: the leading principle underlying such techniques is being as much systematic as possible in identifying a representative set of program behaviors (generally in the form of subclasses of the input domain). However, a comprehensive view of the Knowledge Area of Software Testing as a means for quality must include other as important objectives for testing, e.g., reliability measurement, usability evaluation, contractor's acceptance, for which different approaches would be taken. Note that the test objective varies with the test object, i.e., in general *different purposes are addressed at the different levels of testing*.

17. The test objective determines how the test set is identified, both with regard to 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?*- (although usually the "*for achieving the stated objective*" part is left implicit and only the first part of the two italicized questions above is posed). Criteria for addressing the first question are referred to as *test adequacy criteria*, while for the second as *test selection criteria*.

18. Sometimes, it can happen that confusion is made between test objectives and techniques. For instance, branch coverage is a popular test technique. Achieving a specified branch coverage measure should not be considered *per se* as the objective of testing: it is a means to improve the chances of finding failures (by systematically exercising every program branch out of a decision point). To avoid such misunderstandings, a clear distinction should be made between test measures which evaluate the thoroughness of the test set, like measures of coverage, and those which instead provide an evaluation of the program under test, based on the observed test outputs, like reliability.

19. Testing concepts, strategies, techniques and metrics need to be integrated into a defined and controlled process, which is run by people. The test process supports testing activities and provide guidance to testing teams, from test planning to test outputs evaluation, in such a way as to provide justified assurance that the test objectives are met cost-effectively.

20. Software testing is a very expensive and labor-intensive part of development. For this reason, tools are instrumental to support test activities. Moreover, in order to enhance cost-effectiveness ratio, a key issue has always been pushing test automation as much as possible.

21. 2.2 Overview

22. Following the above-presented conceptual scheme, this description is organized as follows.

23. Part A deals with *Testing Basic Concepts and Definitions*. It covers the basic definitions within the Software Testing field, as well as an introduction to the terminology. In the same part, the scope of the Knowledge Area is laid down, also in relation with other activities.

24. Part B deals with *Test Levels*. It consists of two (orthogonal) subsections: B.1 lists the levels in which the testing of large software systems is traditionally subdivided. In B.2 testing for specific conditions or properties is instead considered, and is referred to as "Objectives of testing". Clearly not all types of testing apply to every system, nor has every possible type been listed, but those most generally applied.

25. As said, several *Test Techniques* have been developed in the last two decades according to various criteria, and new ones are still proposed. "Generally accepted" techniques are covered in Part C.

26. *Test-related Measures* are dealt in Part D.

27. Finally, issues relative to *Managing the Test Process* are covered in Part E.

28. Existing tools and concepts related to supporting and automating the activities into the test process are not addressed here. They are covered within the Knowledge Area description of Software Engineering Methods and Tools in this Guide.

29. 3. Breakdown of Topics for Software testing

30. This section gives the list of topics identified for the Software Testing Knowledge Area, with succinct descriptions and references. Two levels of references are provided with topics: the core reference material within brackets, and additional references within parentheses. The core references have been *reasonably* limited, according to the guideline that they should consist of the study material for a *software engineering licensing exam that a graduate would pass after completing four years of work experience*. In particular, the core reference material for Software Testing has been identified into selected book chapters (for instance, Chapter 1 of reference Be is denoted as Be:c1), or, in some cases, sections (for instance, Section 1.4 of Chapter 1 of Be is denoted as Be:c1s1.4). The Further Readings list includes several refereed journal or conference papers and relevant Standards, for a deeper study of the pointed arguments.

31. The breakdown is also visually described by the following tables (note that two decompositions are proposed for the level 1 topic of Testing Techniques)

Table 1: Level 1 Topics for Software Testing

32.	**Software Testing**	**A. Testing Basic Concepts and Definitions**
33.		**B. Test Levels**
34.		**C. Test Techniques**
35.		**D. Test related measures**
36.		**E. Managing the Test Process**

Table 1-A: Decomposition for Testing Basic Concepts and Definitions

A. Testing Basic Concepts and Definitions	*A.1 Testing-related terminology*	Definitions of testing and related terminology
		Faults vs. Failures
	A.2 Theoretical foundations	Test selection criteria/Test adequacy criteria (or stopping rules)
		Testing effectiveness/Objectives for testing
		Testing for defect removal
		The oracle problem
		Theoretical and practical limitations of testing
		The problem of infeasible paths
		Testability
		Relationships of testing to other activities

Table 1-B: Decomposition for Test Levels

B. Test Levels	*B.1 The object of the test*	Unit testing
		Integration testing
		System testing
	B.2 Objectives of testing	Acceptance/qualification testing
		Installation testing
		Alpha and Beta testing
		Conformance testing/ Functional testing/ Correctness testing
		Reliability achievement and evaluation by testing
		Regression testing
		Performance testing
		Stress testing
		Back-to-back testing
		Recovery testing
		Configuration testing
		Usability testing

Table 1-C': Decomposition for Test Techniques (criterion "base on which tests are generated")

C. Test Techniques	*C1.1 Based on tester's intuition*	Ad hoc
	C1.2 Specification-based	Equivalence partitioning
		Boundary-value analysis
		Decision table
		Finite-state machine-based
		Testing from formal specifications
		Random testing
	C1.3 Code-based	Reference models for code-based testing (flow graph, call graph)
		Control flow-based criteria
		Data flow-based criteria

Table 1-C': Decomposition for Test Techniques (criterion “base on which tests are generated”)

	C1.4 Fault-based	Error guessing
		Mutation testing
	C1.5 Usage-based	Operational profile
		SRET
	C1.6 Based on nature of application	Object-oriented testing
		Component-based testing
		GUI testing
		Testing of concurrent programs
		Protocol conformance testing
		Testing of distributed systems
		Testing of real-time systems
		Testing of scientific software
	C3 Selecting and combining techniques	Functional and structural
		Coverage and operational/Saturation effect

Table 1-C”: Additional decomposition for Test Techniques
(criterion “ignorance or knowledge of implementation”)

C. Test Techniques	*C2.1 Black-box techniques*	Equivalence partitioning
		Boundary-value analysis
		Decision table
		Finite-state machine-based
		Testing from formal specifications
		Error guessing
		Random testing
		Operational profile
		SRET
	C2.2 White-box techniques	Reference models for code-based testing (flow graph, call graph)
		Control flow-based criteria
		Data flow-based criteria
		Mutation testing

103. Table 1-D: Decomposition for Test Related Measures

104.	**D. Test Related Measures**	*D.1 Evaluation of the program under test*	Program measurements to aid in planning and designing testing
105.			Types, classification and statistics of faults
106.			Remaining number of defects/Fault density
107.			Life test, reliability evaluation
108.			Reliability growth models
109.		*D.2 Evaluation of the tests performed*	Coverage/thoroughness measures
110.			Fault seeding
111.			Mutation score
112.			Comparison and relative effectiveness of different techniques

113. Table 1-E: Decomposition for Managing the Test Process

114.	**E. Managing the Test Process**	*E.1 Management concerns*	Attitudes/Egoless programming
115.			Test process
116.			Test documentation
117.			Internal vs. independent test team
118.			Cost/effort estimation and other process metrics
119.			Test reuse
120.		*E.2 Test activities*	Planning
121.			Test case generation
122.			Test environment development
123.			Execution
124.			Test results evaluation
125.			Trouble reporting/Test log
126.			Defect tracking

127. A. Testing Basic Concepts and Definitions

128. A.1 Testing-related terminology

129. ◆ Definitions of testing and related terminology [Be:c1; Jo:c1,2,3,4; Ly:c2s2.2] (610)

130. A comprehensive introduction to the Knowledge Area of Software Testing is provided by the core references. Moreover, the IEEE Standard Glossary of Software Engineering Terminology (610) defines terms for the whole field of software engineering, including testing-related terms.

131. ◆ Faults vs. Failures [Ly:c2s2.2; Jo:c1; Pe:c1; Pf:c7] (FH+; Mo; ZH+:s3.5; 610; 982.2:fig3.1.1-1; 982.2:fig6.1-1)

132. Many terms are used in the software literature to speak of malfunctioning, notably *fault*, *failure*, *error*, and several others. Often these terms are used interchangeably. However, in some cases they are given a more precise meaning (unfortunately, not in consistent ways between different sources), in order to identify the subsequent steps of the cause-effect chain that originates somewhere, e.g., in the head of a designer, and eventually leads to the system's user observing an undesired effect. This terminology is precisely defined in the IEEE Std 610.12-1990, Standard Glossary of Software Engineering Terminology (610) and is also

discussed in more depth in the SQ Knowledge Area. What is essential in order to discuss Software Testing, as a minimum, is to clearly distinguish between the *cause* for a malfunctioning, for which the term *fault* is used here, and an undesired effect observed in the system delivered service, that will be called a *failure*. Testing can reveal failures, but then to remove them it is the faults that must be fixed.

133. However, it should be recognized that not always the cause of a failure can be univocally identified, i.e., no theoretical criteria exists to uniquely say what the fault was that caused a failure. One may choose to say the fault was "what was changed", but other things could have been changed just as well. This is why some authors instead of faults prefer to speak in terms of *failure-causing inputs* (FH+), i.e., those sets of inputs that when executed cause a failure.

134. A.2 Theoretical foundations

135. ◆ Test selection criteria/Test adequacy criteria (or stopping rules) [Pf:c7s7.3; ZH+:s1.1] (We-b; WW+; ZH+)

136. A test criterion is a means of deciding which a suitable set of test cases should be. A criterion can be used for selecting the test cases, or for checking if a selected test suite is adequate, i.e., to decide if the testing can be stopped. In mathematical terminology it would be a decision predicate defined on triples (P, S, T), where P is a program, S is the specification (intended here to mean in general sense any relevant source of information for testing) and T is a test set.

137. ◆ Testing effectiveness/Objectives for testing [Be:c1s1.4; Pe:c21] (FH+)

138. Testing amounts at observing a sample of program executions. The selection of the sample can be guided by different objectives: it is only in light of the objective pursued that the effectiveness of the test set can be evaluated. This important issue is discussed at some length in the references provided.

139. ◆ Testing for defect identification [Be:c1; KF+:c1]

140. In testing for defect identification a successful test is one that causes the system to fail. This is quite different from testing to demonstrate that the software meets its specification, or other desired properties, whereby testing is successful if no (important) failures are observed.

141. ◆ The oracle problem [Be:c1] (We-a; BS)

142. An oracle is any (human or mechanical) agent that decides whether a program behaved correctly on a given test, and produces accordingly a verdict of "pass" or "fail". There exist many different kinds of oracles; oracle automation still poses several open problems.

143. ◆ Theoretical and practical limitations of testing [KF+:c2] (Ho)

144. Testing theory warns against putting a not justified level of confidence on series of passed tests. Unfortunately, most established results of testing theory are negative ones, i.e., they state what testing can never achieve (as opposed to what it actually achieved). The most famous quotation in this regard is Dijkstra aphorism that "program testing can be used to show the *presence* of bugs, but never to show their absence". The obvious reason is that complete testing is not feasible in real systems. Because of this, testing must be driven based on risk, i.e., testing can also be seen as a risk management strategy.

145. ◆ The problem of infeasible paths [Be:c3]

146. Infeasible paths, i.e., control flow paths which cannot be exercised by any input data, are a significant problem in path-oriented testing, and particularly in the automated derivation of test inputs for code-based testing techniques.

147. ◆ Testability [Be:c3,c13] (BM; BS; VM)

148. The term of software testability has been recently introduced in the literature with two related, but different meanings: on the one hand as the degree to which it is easy for a system to fulfill a given test coverage criterion, as in (BM); on the other hand, as the likelihood (possibly measured statistically) that the system exposes a failure under testing, *if* it is faulty, as in (VM, BS). Both meanings are important.

149. A.3 Relationships of testing to other activities

150. Here the relation between the Software Testing and other related activities of software engineering is considered. Software Testing is related to, but different from, static analysis techniques, proofs of correctness, debugging and programming. On the other side, it is informative to consider testing from the point of view of software quality analysts, users of CMM and Cleanroom processes, and of certifiers. A non-exhaustive list of interesting

151. ◆ Testing vs. Static Analysis Techniques [Be:c1; Pe:c17p359-360] (1008:p19)

152. ◆ Testing vs. Correctness Proofs [Be:c1s5; Pf:c7]

153. ◆ Testing vs. Debugging [Be:c1s2.1] (1008:p19)

154. ◆ Testing vs. Programming [Be:c1s2.3]

155. ◆ Testing within SQA (see the SQ Knowledge Area in this Guide to the SWEBOK)

156. ◆ Testing within CMM (Po:p117-123)

157. ◆ Testing within Cleanroom [Pf:c8s8.9]

158. ◆ Testing and Certification (WK+)

159. B. Test Levels

160. B.1 The object of the test

161. Testing of large software systems usually involves more steps [Be:c1; Jo:c12; Pf:c7]

162. ◆ Unit testing [Be:c1; Pe:c17; Pf:c7s7.3] (1008)

163. Unit testing verifies the functioning in isolation of software pieces that are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of tightly related units. A test unit is defined more precisely in the IEEE Standard for Software Unit Testing [1008], that also describes an integrated approach to systematic and documented unit testing. Clearly, unit testing starts after a clean compile.

164. ◆ Integration testing [Jo:c12,13; Pf:c7s7.4]

165. Integration testing is the process of verifying the interaction between system components (possibly already tested in isolation). Systematic, incremental integration testing strategies, such as top-down or bottom-up, are to be preferred to putting all units together at once, that is pictorially said "big-bang" testing.

166. ◆ System testing [Jo:c14; Pf:c8]

167. System testing is concerned with the behavior of a whole system, and at this level the main goal is not to find functional failures (most of them should have been already found at finer levels of testing), but rather to demonstrate performance in general. External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level.

168. There are many system properties one may want to verify by testing, including conformance, reliability, usability among others. These are discussed below under part "Objectives of testing".

169. B.2 Objectives of Testing [Pe:c8; Pf:c8s8.3]

170. Testing of a software system (or subsystem) can be aimed at verifying different properties. Test cases can be designed to check that the functional specifications are correctly implemented, which is variously referred to in the literature as conformance testing, "correctness" testing, functional testing. However several other non-functional properties need to be tested as well. References cited above give essentially a collection of the potential different purposes. The topics separately listed below (with the same or additional references) are those most often cited in the literature.

171. Note that some kinds of testing are more appropriate for custom made packages, e.g. installation testing, while others for generic products, e.g. beta testing.

172. ◆ Acceptance/qualification testing [Pe:c10; Pf:c8s8.5] (12207:s5.3.9)

173. Acceptance testing checks the system behavior against the customer's requirements (the "contract"), and is usually conducted by or with the customer.

174. ◆ Installation testing [Pe:c9; Pf:c8s8.6]

175. After completion of system and acceptance testing, the system is verified upon installation in the target environment, i.e., system testing is conducted according to the hardware configuration requirements. Installation procedures are also verified.

176. ◆ Alpha and Beta testing [KF+:c13]

177. Before releasing the system, sometimes it is given in use to a small representative set of potential users, in-house (alpha testing) or external (beta testing), who report to the developer potential experienced problems with use of the product. Alpha and beta use is uncontrolled, i.e., the testing does not refer to a test plan.

178. B3. Conformance testing/Functional testing/Correctness testing [KF+:c7; Pe:c8] (WK+)

179. Conformance testing is aimed at verifying whether the observed behavior of the tested system conforms to its specification.

180. ◆ Reliability achievement and evaluation by testing [Pf:c8s.8.4; Ly:c7] (Ha; Musa and Ackermann in Po:p146-154)

181. By testing failures can be detected. If the faults that are the cause of the identified failures are efficaciously removed, the software will be more reliable. In this sense, testing is a means to improve reliability. On the other hand, by randomly generating test cases accordingly to the operational profile, statistical measures of reliability can be derived. Using reliability growth models, both objectives can be pursued together (see also part D.1).

182. ◆ Regression testing [KF+:c7; Pe:c11,c12; Pf:c8s8.1] (RH)

183. According to (610), regression testing is the "selective retesting of a system or component to verify that modifications have not caused unintended effects [...]". Regression testing can be conducted at each of the test levels in B.1. [Be] defines it as any repetition of tests intended to show that the software's behavior is unchanged except insofar as required.

184. ◆ Performance testing [Pe:c17; Pf:c8s8.3] (WK+)

185. This is specifically aimed at verifying that the system meets the specified performance requirements, e.g., capacity and response time. A specific kind of performance testing is volume testing (Pe:p185, p487; Pf:p349), in which internal program or system limitations are proved.

186. ◆ Stress testing [Pe:c17; Pf:c8s8.3]

187. Stress testing exercises a system at the maximum design load as well as beyond it.

188. ◆ Back-to-back testing

189. A same test set is presented to two implemented versions of a system, and the results are compared with each other.

190. ◆ Recovery testing [Pe:c17; Pf:c8s8.3]

191. It is aimed at verifying system restart capabilities after a "disaster".

192. ◆ Configuration testing [KF+:c8; Pf:c8s8.3]

193. In those cases in which a system is built to serve different users, configuration testing analyzes the system under the various specified configurations.

194. ◆ Usability testing [Pe:c8; Pf:c8s8.3]

195. It evaluates the ease of using and learning the system by the end users.

196. C. Test Techniques

197. In this section, two alternative classifications of test techniques are proposed. It is arduous to find a homogeneous criterion for classifying all techniques, as there exist many and very disparate.

198. The first classification, from C1.1 to C1.6, is based on how tests are generated, i.e., respectively from: tester's intuition and expertise, the specifications, the code structure, the (real or artificial) faults to be discovered, the field usage or finally the nature of application, which in some case can require knowledge of specific test problems and of specific test techniques.

199. The second classification is the classical distinction of test techniques between *black-box* and *white-box* (pictorial terms derived from the world of integrated circuit testing). Test techniques are here classified according to whether the tests rely on information about how the software has been designed and coded (white-box, somewhere also said glass-box), or instead only rely on the input/output behavior, without no assumption about what happens in between the “pins” of the system (black box). Clearly this second classification is more coarse than the first one, and it does not allow us to categorize the techniques specialized on the nature of application (section C1.6) nor ad hoc approaches, because these can be either black-box or white-box.

200. A final section, *C3,* deals with combined use of more techniques.

201. C1: CLASSIFICATION “base on which tests are generated”

202. C1.1 Based on tester's intuition [KF+:c1]

203. Perhaps the most widely practiced technique remains *ad hoc testing*: test cases are derived relying on the tester skill and intuition (“exploratory” testing), and on his/her experience with similar programs. While a more systematic approach is advised, this remains very useful to identify special tests, not easily "captured" by formalized techniques.

204. C1.2 Specification-based

205. ◆ Equivalence partitioning [Jo:c6; KF+:c7]

206. The input domain is subdivided into a collection of subsets, or "equivalent classes", which are deemed equivalent according to a specified relation, and a representative set of tests (sometimes even one) is taken from within each class.

207. ◆ Boundary-value analysis [Jo:c5; KF+:c7]

208. Test cases are chosen on and near the boundaries of the input domain of variables, with the underlying rationale that many defects tend to concentrate near the extreme values of inputs. A simple, and often worth, extension of this technique is *Robustness Testing*, whereby test cases are also chosen outside the domain, in fact to test program robustness to unexpected, erroneous inputs.

209. ◆ Decision table [Be:c10s3] (Jo:c7)

210. Decision tables represent logical relationships between conditions (roughly, inputs) and actions (roughly, outputs). Test cases are systematically derived by considering every possible combination of conditions and actions. A related techniques is *Cause-effect graphing* [Pf:c8].

211. ◆ Finite-state machine-based [Be:c11; Jo:c4s4.3.2]

212. By modeling a program as a finite state machine, tests can be selected in order to cover states and transitions on it, applying different techniques. This technique is suitable for transaction-processing, reactive, embedded and real-time systems.

213. ◆ Testing from formal specifications [ZH+:s2.2] (BG+; DF; HP)

214. Giving the specifications in a formal language (i.e., one with precisely defined syntax and semantics) allows for automatic derivation of functional test cases from the specifications, and at the same time provides a reference output, an oracle, for checking test results. Methods for deriving test cases from model-based (DF, HP) or algebraic specifications (BG+) are distinguished.

215. ◆ Random testing [Be:c13; KF+:c7]

216. Tests are generated purely random (not to be confused with statistical testing from the operational profile, where the random generation is biased towards reproducing field usage, see C1.5). Actually, therefore, it is difficult to categorize this technique under the scheme of "base on which tests are generated". It is put under the Specification-based entry, as at least which is the input domain must be known, to be able to pick random points within it.

217. *C1.3 Code-based*

218. ◆ Reference models for code-based testing (flowgraph, call graph) [Be:c3; Jo:c4].

219. In code-based testing techniques, the control structure of a program is graphically represented using a flowgraph, i.e., a directed graph whose nodes and arcs correspond to program elements. For instance, nodes may represent statements or uninterrupted sequences of statements, and arcs the transfer of control between nodes.

220. ◆ Control flow-based criteria [Be:c3; Jo:c9] (ZH+:s2.1.1)

221. Control flow-based coverage criteria aim at covering all the statements or the blocks in a program, or proper combinations of them. Several coverage criteria have been proposed (like Decision/Condition Coverage), in the attempt to get good approximations for the exhaustive coverage of all control flow paths, that is unfeasible for all but trivial programs.

222. ◆ Data flow-based criteria [Be:c5] (Jo:c10; ZH+:s2.1.2)

223. In data flow-based testing, the control flowgraph is annotated with information about how the program variables are defined and used. Different criteria exercise with varying degrees of precision how a value assigned to a variable is used along different control flow paths. A reference notion is a definition-use pair, which is a triple (d,u,V) such that: V is a variable, d is a node in which V is defined, and u is a node in which V is used; and such that there exists a path between d and u in which the definition of V in d is used in u.

224. *C1.4 Fault-based* (Mo)

225. With different degrees of formalization, fault based testing techniques devise test cases specifically aimed at revealing categories of likely or pre-defined faults.

226. ◆ Error guessing [KF+:c7]

227. In error guessing, test cases are ad hoc designed by testers trying to figure out those, which could be the most plausible faults in the given program. A good source of information is the history of faults discovered in earlier projects, as well as tester's expertise.

228. ◆ Mutation testing [Pe:c17; ZH+:s3.2-s3.3]

229. Originally conceived as a technique to evaluate a test set (see D.2.2), mutation testing is also a testing criterion in itself: either tests are randomly generated until enough mutants are killed or tests are specifically designed to kill (survived) mutants. In the latter case, mutation testing can also be categorized as a code-based technique. The underlying assumption of mutation testing, the coupling effect, is that by

looking for simple syntactic faults, also more complex, (i.e., real) faults will be found.

230. *C1.5 Usage-based*

231. ◆ Operational profile [Jo:c14s14.7.2; Ly:c5; Pf:c8]

232. In testing for reliability evaluation, the test environment must reproduce as closely as possible the product use in operation. In fact, from the observed test results one wants to infer the future reliability in operation. To do this, inputs are assigned a probability distribution, or profile, according to their occurrence in actual operation.

233. ◆ (Musa's) SRET [Ly:c6]

234. Software Reliability Engineered Testing (SRET) is a testing methodology encompassing the whole development process, whereby testing is "designed and guided by reliability objectives and expected relative usage and criticality of different functions in the field".

235. *C1.6 Based on nature of application*

236. The above techniques apply to all types of software, and their classification is based on how test cases are derived. However, for some kinds of applications some additional know-how is required for test derivation. Here below a list of few "specialized" testing techniques is provided, based on the nature of the application under test.

237. ◆ Object-oriented testing [Jo:c15; Pf:c7s7.5] (Bi)

238. ◆ Component-based testing

239. ◆ GUI testing (OA+)

240. 1. Testing of concurrent programs (CT)

241. 2. Protocol conformance testing (Sidhu and Leung in Po:p102-115; BP)

242. ◆ Testing of distributed systems

243. ◆ Testing of real-time systems (Sc)

244. ◆ Testing of scientific software

245. C2: CLASSIFICATION “ignorance or knowledge of implementation”

246. *C2.1 Black-box techniques*

247. ◆ Equivalence partitioning [Jo:c6; KF+:c7]

248. ◆ Boundary-value analysis [Jo:c5; KF+:c7]

249. ◆ Decision table [Be:c10s3] (Jo:c7)

250. ◆ Finite-state machine-based [Be:c11; Jo:c4s4.3.2]

251. ◆ Testing from formal specifications [ZH+:s2.2] (BG+; DF; HP)

252. ◆ Error guessing [KF+:c7]

253. ◆ Random testing [Be:c13; KF+:c7]

254. ◆ Operational profile [Jo:c14s14.7.2; Ly:c5; Pf:c8]

255. ◆ (Musa's) SRET [Ly:c6]

256. *C2.2 White-box techniques*

257. ◆ Reference models for code-based testing (flowgraph, call graph) [Be:c3; Jo:c4].

258. ◆ Control flow-based criteria [Be:c3; Jo:c9] (ZH+:s2.1.1)

259. ◆ Data flow-based criteria [Be:c5] (Jo:c10; ZH+:s2.1.2)

260. ◆ Mutation testing [Pe:c17; ZH+:s3.2-s3.3]

261. *C3 Selecting and combining techniques*

262. ◆ Functional and structural [Be:c1s.2.2; Jo:c1, c11s11.3; Pe:c17] (Po:p3-4; Po: Appendix 2)

263. Functional and structural approaches to test selection are not to be seen as alternative, but rather as complementary: in fact, they use different sources of information and highlight different kinds of problems. They should be used in combination, compatibly with budget availability.

264. ◆ Coverage and operational/Saturation effect (Ha; Ly:p541-547; Ze)

265. This topic discusses the differences and complementarity of deterministic and statistical approaches to test case selection.

266. D. Test related measures

267. Measurement is instrumental to quality analysis. Indeed, product evaluation is effective only when based on quantitative measures. This section specifically focuses on measures that are obtained from data collected by testing. A wider coverage of the topic of quality measurement, including fundamentals, metrics and techniques for measurement, is provided in the SQ Knowledge Area of the Guide to the SWEBOK. A comprehensive reference is provided by the IEEE Std. 982.2 "Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software". It has been originally conceived as a guide to using the companion standard 982.1, that is the Dictionary. However, the guide is also a valid and very useful reference

by itself, for selection and application of measures in a project.

268. Test related measures can be divided into two classes: those relative to evaluating the program under test, and those relative to evaluating the test set. The first class, for instance, includes measures that count and predict either faults (e.g., fault density) or failures (e.g., reliability). The second class instead evaluates the test suites against selected test criteria; notably, this is what is usually done by measuring the code coverage achieved by the executed tests. Measures relative to the test process for management purposes are instead considered in part E.

269. *D.1 Evaluation of the program under test* (982.2)

270. ◆ Program measurements to aid in planning and designing testing. [Be:c7s4.2; Jo:c9] (982.2:sA16, BMa)

271. Measures based on program size (e.g., SLOC, function points) or on program structure (e.g., complexity) is useful information to guide the testing. These are also covered in the SQ Knowledge Area.

272. ◆ Types, classification and statistics of faults [Be:c2; Jo:c1; Pf:c7] (1044, 1044.1; Be: Appendix; Ly:c9; KF+:c4, Appendix A)

273. The testing literature is rich of classifications and taxonomies of faults. Testing allows for discovering defects. To make testing more effective it is important to know which types of faults could be found in the application under test, and the relative frequency with which these faults have occurred in the past. This information can be very useful to make quality predictions as well as for process improvement. The topic "Defect Characterization" is also covered more deeply in the SQ Knowledge Area. An IEEE standard on how to classify software "anomalies" (1044) exists, with a relative guide (1044.1) to implement it. An important property for fault classification is orthogonality, i.e., ensuring that each fault can be univocally identified as belonging to one class.

274. ◆ Remaining number of defects/Fault density [Pe:c20] (982.2:sA1; Ly:c9)

275. In common industrial practice a product under test is assessed by counting and classifying the discovered faults by their types (see also A1). For each fault class, fault density is measured by the ratio between the number of faults found and the size of the program.

276. ◆ Life test, reliability evaluation [Pf:c8] (Musa and Ackermann in Po:p146-154)

277. A statistical estimate of software reliability, that can be obtained by operational testing (see in B.2), can be used to evaluate a product and decide if testing can be stopped.

278. ◆ Reliability growth models [Ly:c7; Pf:c8] (Ly:c3, c4)

279. Reliability growth models provide a prediction of reliability based on the failures observed under operational testing. They assume in general that the faults that caused the observed failures are fixed (although some models also accept imperfect fixes) and thus, on average, the product reliability exhibits an increasing trend. There exist now tens of published models, laid down on some common assumptions as well as on differing ones. Notably, the models are divided into *failures-count* and *time-between-failures* models.

280. *D.2 Evaluation of the tests performed*

281. ◆ Coverage/thoroughness measures [Jo:c9; Pf:c7] (982.2:sA5-sA6)

282. Several test adequacy criteria require the test cases to systematically exercise a set of elements identified in the program or in the specification (see Part C). To evaluate the thoroughness of the executed tests, testers can monitor the elements covered, so that they can dynamically measure the ratio (often expressed as a fraction of 100%) between covered elements and the total number. For example, one can measure the percentage of covered branches in the program flowgraph, or of exercised functional requirements among those listed in the specification document. Code-based adequacy criteria require appropriate instrumentation of the program under test.

283.◆ Fault seeding [Pf:c7] (ZH+:s3.1)

284. Some faults are artificially introduced into the program before test. By monitoring then which and how many of the artificial faults are discovered by the executed tests, this technique allows for measuring testing effectiveness, and for estimating how many (original) faults remain.

285. ◆ Mutation score [ZH+:s3.2-s3.3]

286. A mutant is a slightly modified version of the program under test, differing from it by a small, syntactic change. Every test case exercises both the original and all generated mutants: for the technique to be effective, a high number of mutants must be automatically derived in systematic way. If a test case is successful in

identifying the difference between the program and a mutant, the latter is said to be killed. Strong and weak mutation techniques have been developed.

287. ◆ Comparison and relative effectiveness of different techniques [Jo:c8,c11; Pe:c17; ZH+:s5] (FW; Weyuker in Po p64-72; FH+)

288. Several studies have been recently conducted to compare the relative effectiveness of different test techniques. It is important to be precise relative to the property against which the techniques are being assessed, i.e., what "effectiveness" is exactly meant for. Possible interpretations are how many tests are needed to find the first failure, or the ratio of the number of faults found by the testing to all the faults found during and after the testing, or how much reliability is improved. Analytical and empirical comparisons between different techniques have been conducted according to each of the above specified notions of "effectiveness".

289. E. Managing the Test Process

290. E.1 Management concerns

291. ◆ Attitudes/Egoless programming [Be:c13s3.2; Pf:c7]

292. A very important component of successful testing is a positive and collaborative attitude towards testing activities. Managers should revert a negative vision of testers as the destroyers of developers' work and as heavy budget consumers. On the contrary, they should foster a common culture towards software quality, by which early failure discover is an objective for all involved people, and not only of testers.

293. ◆ Test process [Be:c13; Pe:c1,c2,c3,c4; Pf:c8] (Po:p10-11; Po:Appendix 1; 12207:s5.3.9;s5.4.2;s6.4;s6.5)

294. A process is defined as "a set of interrelated activities, which transform inputs into outputs"[12207]. Test activities conducted at different levels (see B.1) must be organized, together with people, tools, policies, measurements, into a well defined process, which is integral part to the life cycle. In the IEEE/EIA Standard 12207.0 testing is not described as a stand alone process, but principles for testing activities are included along with the five primary life cycle processes, as well as along with the supporting process.

295. ◆ Test documentation and workproducts [Be:c13s5; KF+:c12; Pe:c19; Pf:c8s8.8] (829)

296. Documentation is an integral part of the formalization of the test process. As The IEEE standard for Software Test Documentation [829] provides a good description of test documents and of their relationship with one another and with the testing process. Test documents includes, among others, Test Plan, Test Design Specification, Test Procedure Specification, Test Case Specification, Test Log and Test Incident or Trouble Report. These documents should be produced and continually updated, at the same standards as other types of documentation in development. Unfortunately, this is not yet common practice.

297. The object of testing, with specified version and identified hw/sw requirements before testing can begin, is documented as the *test item.*

298. ◆ Internal vs. independent test team [Be:c13s2.2-2.3; KF+:c15; Pe:c4; Pf:c8]

299. Formalization of the test process requires formalizing the test team organization as well. The test team can be composed of members internal to the project team, or of external members, in the latter case bringing in an unbiased, independent perspective, or finally of both internal and external members. The decision will be determined by considerations of costs, schedule and application criticality.

300. ◆ Cost/effort estimation and other process metrics [Pe:c4, c21] (Pe:Appendix B; Po:p139-145; 982.2:sA8-sA9)

301. In addition to those discussed in Part D, several metrics relative to the resources spent on testing, as well as to the relative effectiveness in fault finding of the different test phases, are used by managers to control and improve the test process. Evaluation of test phase reports is often combined with root cause analysis to evaluate test process effectiveness in finding faults as early as possible. Moreover, the resources that are worth spending in testing should be commensurate to the use/criticality of the application: the techniques listed in part C have different costs, and yield different levels of confidence in product reliability. “Good enough” testing should be planned.

302. ◆ Test Reuse [Be:c13s5]

303. E.2 Test Activities

304. ◆ Planning [KF+:c12; Pe:c19; Pf:c7s7.6] (829:s4; 1008:s1, s2, s3)

305. - Test case generation [KF+:c7] (Po:c2; 1008:s4, s5)

306. - Test environment development [KF+:c11]

307. - Execution [Be:c13; KF+:c11] (1008:s6, s7;)

308. - Test results evaluation [Pe:c20,c21] (Po:p18-20; Po:p131-138)

309. - Trouble reporting/Test log [KF+:c5; Pe:c20] (829:s9-s10)

310. - Defect tracking [KF+:c6]

311. 4. BREAKDOWN RATIONALE

312. The conceptual scheme followed in decomposing the Software Testing Knowledge Area is described in Section 2.1. Level 1 topics include five entries, labeled from A to E, that correspond to the fundamental and complementary concerns forming the Software Testing knowledge: Basic Concepts and Definitions, Levels, Techniques, Measures, and Process. There is not a standard way to decompose the Software Testing Knowledge Area, each book on Software Testing would structure its table of contents in different ways. However any thorough book on Software Testing would cover these five topics. A sixth level 1 topic would be Test Tools. These are not covered here, but in a specific section of the *Software Engineering Methods and Tools* chapter of the Guide to the SWEBOK.

313. The breakdown is three levels deep. The second level is for making the decomposition more understandable. The selection of level 3 topics, that are the subjects of study, has been quite difficult. This description is expected to be as inclusive as possible (too many topics are deemed better than having relevant topics missing). On the other side, the proposed breakdown should be compatible with breakdowns generally found in industry, in literature and in standards, and the selected topics should be "generally accepted" knowledge. Finding a breakdown of topics that is "generally accepted" by all different communities of potential users of the Guide to the SWEBOK is challenging for Software Testing, because there still exists a wide gap between the literature on Software Testing and current industrial test practice. There are topics that have been taking a relevant position in the academic literature for many years now, but are not generally used in industry, for example data-flow based or mutation testing. The position taken in writing this document has been to include any relevant topics in the literature, even those that are likely not considered so relevant by practitioners at the current time. The proposed breakdown of topics for Software Testing is thus considered as an inclusive list, from which each stakeholder can pick according to his/her needs.

314. However, under the precise definition for "generally accepted" adopted in the Guide to the SWEBOK, i.e., *knowledge to be included in the study material of a software engineering with four years of work experience*, some of the included topics (like the examples above) would be lightly covered in a curriculum of a software engineering with four years of experience. The ratings in the Bloom's taxonomy of topics in an Appendix of the entire Guide reflect this guideline, and the core References have been selected accordingly, i.e., they provide reading material for the topics according to this precise meaning of "generally accepted". Advanced topics are more deeply covered in the Further Reading list.

315. Finally, the reader should understand the high difficulty of being selective in limiting topics and references to a reasonable amount. As spelled out in the specifications for the Stone Man Version of the Guide to the SWEBOK, *the breakdowns of topics are expected to be "reasonable", not "perfect"*, and definitely they are to be seen as documents undergoing continuous improvement.

316. 5. MATRIX OF TOPICS VS. REFERENCE MATERIAL

317.	**A. Testing Basic Concepts and Definitions**	**[Be]**	**[Jo]**	**[Ly]**	**[KF+]**	**[Pe]**	**[Pf]**	**[ZH+]**
318.	Definitions of testing and related terminology	C1	C1,2,3,4	C2S2.2				
319.	Faults vs. Failures		C1	C2S2.2		C1	C7	
320.	Test selection criteria/Test adequacy criteria (or stopping rules)						C7S7.3	S1.1
321.	Testing effectiveness/Objectives for testing	C1S1.4				C21		
322.	Testing for defect identification	C1			C1			
323.	The oracle problem	C1						
324.	Theoretical and practical limitations of testing				C2			
325.	The problem of infeasible paths	C3						
326.	Testability	C3,13						
327.	Testing vs. Static Analysis Techniques	C1				C17		
328.	Testing vs. Correctness Proofs	C1S5					C7	
329.	Testing vs. Debugging	C1S2.1						
330.	Testing vs. Programming	C1S2.3						
331.	Testing within SQA							
332.	Testing within CMM							
333.	Testing within Cleanroom						C8S8.9	
334.	Testing and Certification							

335.	**B. Test Levels**	**[Be]**	**[Jo]**	**[Ly]**	**[KF+]**	**[Pe]**	**[Pf]**
336.	Unit testing	C1				C17	C7S7.3
337.	Integration testing		C12,13				C7S7.4
338.	System testing		C14				C8
339.	Acceptance/qualification testing					C10	C8S8.5
340.	Installation testing					C9	C8S8.6
341.	Alpha and Beta testing				C13		
342.	Conformance testing/ Functional testing/ Correctness testing				C7	C8	
343.	Reliability achievement and evaluation by testing			C7			C8S8.4
345.	Regression testing				C7	C11,12	C8S8.1
346.	Performance testing					C17	C8S8.3
347.	Stress testing					C17	C8S8.3
348.	Back-to-back testing						
349.	Recovery testing					C17	C8S8.3
350.	Configuration testing				C8		C8S8.3
351.	Usability testing					C8	C8S8.3

352.	**C. Test Techniques**	**[Be]**	**[Jo]**	**[Ly]**	**[KF+]**	**[Pe]**	**[Pf]**	**[ZH+]**
353.	Ad hoc				C1			
354.	Equivalence partitioning		C6		C7			
355.	Boundary-value analysis		C5		C7			
356.	Decision table	C10S3						
357.	Finite-state machine-based	C11	C4S4.3.2					
358.	Testing from formal specifications							S2.2
359.	Random testing	C13			C7			
360.	Reference models for code-based testing (flow graph, call graph)	C3	C4					
361.	Control flow-based criteria	C3	C9				C7	
362.	Data flow-based criteria	C5						
363.	Error guessing						C7	
364.	Mutation testing					C17		S3.2, 3.3
365.	Operational profile		C14S14.7.2	C5			C8	
366.	SRET			C6				
367.	Object-oriented testing		C15				C7S7.5	
368.	Component-based testing							
369.	GUI testing							
370.	Testing of concurrent programs							
371.	Protocol conformance testing							
372.	Testing of distributed systems							
373.	Testing of real-time systems							
374.	Testing of scientific software							
375.	Functional and structural	C1S2.2	C1,11S11.3			C17		
376.	Coverage and operational/Saturation effect							

377.	**D. Test Related Measures**	**[Be]**	**[Jo]**	**[Ly]**	**[KF+]**	**[Pe]**	**[Pf]**	**[ZH+]**
378.	Program measurements to aid in planning and designing testing.	C7S4.2	C9					
379.	Types, classification and statistics of faults	C2	C1				C7	
380.	Remaining number of defects/Fault density					C20		
381.	Life test, reliability evaluation						C8	
382.	Reliability growth models			C7			C8	
383.	Coverage/thoroughness measures		C9				C7	
384.	Fault seeding						C7	
385.	Mutation score							S3.2, 3.3
386.	Comparison and relative effectiveness of different techniques		C8,11			C17		S5

387.	**E. Managing the Test Process**	**[Be]**	**[Jo]**	**[Ly]**	**[KF+]**	**[Pe]**	**[Pf]**
388.	Attitudes/Egoless programming	C13S3.2					C7
389.	Test process	C13				C1,2,3,4	C8
390.	Test documentation and workproducts	C13S5			C12	C19	C8S8.8
391.	Internal vs. independent test team	C13S2.2,2.3			C15	C4	C8

387. E. Managing the Test Process	[Be]	[Jo]	[Ly]	[KF+]	[Pe]	[Pf]
392. Cost/effort estimation and other process metrics					C4,21	
393. Test reuse	C13					
394. Planning				C12	C19	C7S7.6
395. Test case generation				C7		
396. Test environment development				C11		
397. Execution	C13			C11		
398. Test results evaluation					C20,21	
399. Trouble reporting/Test log				C5	C20	
400. Defect tracking				C6		

401. 6. CORE REFERENCES FOR SOFTWARE TESTING

402. [Be] Beizer, B. *Software Testing Techniques* 2nd Edition. Van Nostrand Reinhold, 1990. [Chapters 1, 2, 3, 5, 7s4, 10s3, 11, 13]

403. [Jo] Jorgensen, P.C., *Software Testing A Craftsman's Approach*, CRC Press, 1995. [Chapters 1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15]

404. [KF+] Kaner, C., Falk, J., and Nguyen, H. Q., *Testing Computer Software*, 2nd Edition, Wiley, 1999. [Chapters 1, 2, 5, 6, 7, 8, 11, 12, 13, 15]

405. [Ly] Lyu, M.R. (Ed.), *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996. [Chapters 2s2.2, 5, 6, 7]

406. [Pe] Perry, W. *Effective Methods for Software Testing*, Wiley, 1995. [Chapters 1, 2, 3, 4, 9, 10, 11, 12, 17, 19, 20, 21]

407. [Pf] Pfleeger, S.L. *Software Engineering Theory and Practice*, Prentice Hall, 1998. [Chapters 7, 8]

408. [ZH+] Zhu, H., Hall, P.A.V., and May, J.H.R. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29, 4 (Dec. 1997) 366-427. [Sections 1, 2.2, 3.2, 3.3,

409. 7. LIST OF FURTHER READINGS

410. Books

411. [Be] Beizer, B. *Software Testing Techniques* 2nd Edition. Van Nostrand Reinhold, 1990.

412. [Jo] Jorgensen, P.C., *Software Testing A Craftsman's Approach*, CRC Press, 1995.

413. [KF+] Kaner, C., Falk, J., and Nguyen, H. Q., *Testing Computer Software*, 2nd Edition, Wiley, 1999.

414. [Ly] Lyu, M.R. (Ed.), *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996.

415. [Pe] Perry, W. *Effective Methods for Software Testing*, Wiley, 1995.

416. [Po] Poston, R.M. *Automating Specification-based Software Testing*, IEEE, 1996.

417. Survey Papers

418. [Bi] Binder, R.V. Testing Object-Oriented Software: a Survey. *Software Testing Verification and Reliability*, 6, 3/4 (Sept-Dec. 1996) 125-252.

419. [ZH+] Zhu, H., Hall, P.A.V., and May, J.H.R. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29, 4 (Dec. 1997) 366-427.

420. Specific Papers

421. [BG+] Bernot, G., Gaudel, M.C., and Marre, B. Software Testing Based On Formal Specifications: a Theory and a Tool. *Software Engineering Journal* (Nov. 1991) 387-405.

422. [BM] Bache, R., and Müllerburg, M. Measures of Testability as a Basis for Quality Assurance. *Software Engineering Journal*, 5 (March 1990) 86-92.

423. [Bma] Bertolino, A., Marrè, M. “How many paths are needed for branch testing?”, *The Journal of Systems and Software*, Vol. 35, No. 2, 1996, pp.95-106.

424. [BP] Bochmann, G.V., and Petrenko, A. Protocol Testing: Review of Methods and Relevance for Software Testing. *ACM Proc. Int. Symposium on Sw Testing and Analysis (ISSTA' 94),* (Seattle, Washington, USA, August 1994) 109-124.

425. [BS] Bertolino, A., and Strigini, L. On the Use of Testability Measures for Dependability Assessment. *IEEE Transactions on Software Engineering*, 22, 2 (Feb. 1996) 97-108.

426. [CT] Carver, R.H., and Tai, K.C., Replay and testing for concurrent programs. *IEEE Software* (March 1991) 66-74

427. [DF] Dick, J., and Faivre, A. Automating The Generation and Sequencing of Test Cases From Model-Based Specifications. *FME'93: Industrial-Strenght Formal Method*, LNCS 670, Springer Verlag, 1993, 268-284.

428. [FH+] Frankl, P., Hamlet, D., Littlewood B., and Strigini, L. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering,* 24, 8, (August 1998), 586-601.

429. [FW] Frankl, P., and Weyuker, E. A formal analysis of the fault detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19, 3, (March 1993), 202-

430. [Ha] Hamlet, D. Are we testing for true reliability? *IEEE Software* (July 1992) 21-27.

431. [Ho] Howden, W.E., Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering,* 2, 3, (Sept. 1976) 208-215

432. [HP] Horcher, H., and Peleska, J. Using Formal Specifications to Support Software Testing. *Software Quality Journal*, 4 (1995) 309-327.

433. [Mo] Morell, L.J. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering* 16, 8 (August 1990), 844-857.

434. [MZ] Mitchell, B., and Zeil, S.J. A Reliability Model Combining Representative and Directed Testing. *ACM/IEEE Proc. Int. Conf. Sw Engineering ICSE 18* (Berlin, Germany, March 1996) 506-514.

435. [OA+] Ostrand, T., Anodide, A., Foster, H., and Goradia, T. A Visual Test Development Environment for GUI Systems. *ACM Proc. Int. Symposium on Sw Testing and Analysis (ISSTA' 98),* (Clearwater Beach, Florida, USA, March 1998) 82-92.

436. [OB] Ostrand, T.J., and Balcer, M. J. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of ACM*, 31, 3 (June 1988), 676-686.

437. [RH] Rothermel, G., and Harrold, M.J., Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22, 8 (Aug. 1996) 529-

438. [Sc] Schütz, W. Fundamental Issues in Testing Distributed Real-Time Systems. *Real-Time Systems Journal*. 7, 2, (Sept. 1994) 129-157.

439. [VM] Voas, J.M., and Miller, K.W. Software Testability: The New Verification. *IEEE Software*, (May 1995) 17-28.

440. [We-a] Weyuker, E.J. On Testing Non-testable Programs. *The Computer Journal*, 25, 4, (1982) 465-470

441. [We-b] Weyuker, E.J. Assessing Test Data Adequacy through Program Inference. *ACM Trans. on Programming Languages and Systems*, 5, 4, (October 1983) 641-655

442. [WK+] Wakid, S.A., Kuhn D.R., and Wallace, D.R. Toward Credible IT Testing and Certification, *IEEE Software*, (August 1999) 39-47.

443. [WW+] Weyuker, E.J., Weiss, S.N, and Hamlet, D. Comparison of Program Test Strategies in *Proc. Symposium on Testing, Analysis and Verification TAV 4* (Victoria, British Columbia, October 1991), ACM Press, 1-10.

444. Standards

445. [610] IEEE Std 610.12-1990, Standard Glossary of Software Engineering Terminology.

446. [829] IEEE Std 829-1998, Standard for Software Test Documentation.

447. [982.2] IEEE Std 982.2-1998, Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.

448. [1008] IEEE Std 1008-1987 (R 1993), Standard for Software Unit Testing.

449. [1044] IEEE Std 1044-1993, Standard Classification for Software Anomalies.

450. [1044.1] IEEE Std 1044.1-1995, Guide to Classification for Software Anomalies.

451. [12207] IEEE/EIA 12207.0-1996, Industry Implementation of Int. Std. ISO/IEC 12207:1995, Standard for Information Technology-Software Life cycle processes

CHAPTER 6
SOFTWARE MAINTENANCE

Thomas M. Pigoski
Technical Software Services (TECHSOFT), Inc.
31 West Garden Street, Suite 100
Pensacola, Florida 32501
USA
+1 850 469 0086
tmpigoski@techsoft.com

TABLE OF CONTENTS

1. Acronyms

2.	CASE	Computer Aided Software Engineering
3.	CM	Configuration Management
4.	CMM	Capability Maturity Model
5.	ICSM	International Conference on Software Maintenance
6.	SCM	Software Configuration Management

7. 1. INTRODUCTION

8. Software maintenance is part of the software engineering life cycle and is a misunderstood area of software engineering. Although systems have been maintained for years, relatively little is written about software maintenance. Funding for research is essentially non-existent and thus the academic researchers publish very little about software maintenance. Practitioners publish even less because of corporate fear of giving away the "competitive edge." Whereas they are many book devoted to software engineering, there are very few books written exclusively about maintenance.

9. Schneidewind [31] stressed the need for standardization of maintenance and, as a result, the IEEE Computer Society Software Engineering Standards Subcommittee published the "IEEE Standard for Software Maintenance" [14] in 1993. Later in 1995 the International Organization for Standards (ISO), developed an international standard for software life-cycle processes, ISO/IEC 12207 [15], which included a maintenance process. ISO/IEC 14764 [16], the ISO/IEC Standard for Software Maintenance, elaborates the maintenance process of ISO/IEC 12207 [15].

10. Software engineering is the application of engineering to software. The classic life-cycle paradigm for software engineering includes: system engineering, analysis, design, code, testing, and maintenance. This paper addresses the maintenance portion of software engineering and the software life-cycle.

11. This paper presents an overview of the Knowledge Area of software maintenance. Brief descriptions of the topics are provided so that

the reader can select the appropriate reference material according to his/her needs.

12. 2. DEFINITION OF KNOWLEDGE AREA

13. This section provides a definition of the Software Maintenance Knowledge Area. Definitions are derived from appropriate standards and current usage.

14. Software maintenance is defined in the IEEE Standard for Software Maintenance, IEEE 1219 [14], as the modification of a software product after delivery to correct faults, to improve performance, or to adapt the product to a modified environment. It does, however, address maintenance activities prior to delivery of the software product but only in an information annex of the standard. Sommerville [33] states that maintenance means evolution.

15. The ISO/IEC 12207 Standard for Life Cycle Processes [15], essentially depicts maintenance as one of the primary life cycle processes and describes maintenance as the process of a software product undergoing "modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify existing software product while preserving its integrity." [15] Of note is that ISO/IEC 12207 describes an activity called "Process Implementation." That activity establishes the maintenance plan and procedures that are later used during the maintenance process.

16. ISO/IEC 14764 [16], the International Standard for Software Maintenance, defines software maintenance in the same terms as ISO/IEC 12207 and places emphasis on the predelivery aspects of maintenance, e.g., planning.

17. A current definition generally accepted by software researchers and practitioners, is as follows:

18. SOFTWARE MAINTENANCE: The totality of activities required to provide cost-effective support to a software system. Activities are performed during the predelivery stage as well as the postdelivery stage. Predelivery activities include planning for postdelivery operations, supportability, and logistics determination. postdelivery activities include software modification, training, and operating a help desk [28].

19. A similar definition is used by the Research Institute in Software Evolution, formerly named the Centre for Software Maintenance.

20. A maintainer is defined by ISO/IEC 12207 as an organization that performs maintenance activities [15].

21. ISO/IEC 12207 identifies the primary activities of software maintenance as: process implementation; problem and modification analysis; modification implementation; maintenance review/acceptance; migration; and retirement. These activities are discussed in a later section. They are further defined by the tasks in the standard.

22. 3. BREAKDOWN OF TOPICS FOR SOFTWARE MAINTENANCE

23. The breakdown of topics for software maintenance is a decomposition of software engineering topics that are "generally accepted" in the software maintenance community. They are general in nature and are not tied to any particular domain, model, or business needs. The presented topics can be used by small and medium sized organizations, as well as by larger software organizations. Organizations should use those topics that are appropriate for their unique situations. The topics are consistent with what is found in current software engineering literature and standards. The common themes of quality, measurement, tools, and standards are included in the breakdown of topics. The breakdown of topics is provided in this section.

24. The breakdown of topics, along with a brief description of each, is provided in this section. Key references are provided. Table 2.1 describes the breakdown.

25. TABLE 2-1. SUMMARY OF THE SOFTWARE MAINTENANCE BREAKDOWN

26.	**SUMMARY OF THE SOFTWARE MAINTENANCE BREAKDOWN**
27.	**Introduction to Software Maintenance**
28.	*Need for Maintenance*
29.	*Categories of Maintenance*
30.	*Maintenance Activities*
31.	*Unique Activities*
32.	*Supporting Activities*
33.	Configuration Management
34.	Quality
35.	*Maintenance Planning Activity*

26.	**SUMMARY OF THE SOFTWARE MAINTENANCE BREAKDOWN**
36.	**Maintenance Process**
37.	*Maintenance Process Models*
38.	**Organization Aspect of Maintenance**
39.	*The Maintainer*
40.	*Outsourcing*
41.	*Organizational Structure*
42.	**Problems of Software Maintenance**
43.	*Technical*
44.	Limited Understanding
45.	Testing
46.	Impact Analysis
47.	Maintainability
48.	*Management*
49.	Alignment with organizational issues
50.	Staffing
51.	Process issues
52.	**Maintenance Cost and Maintenance Cost Estimation**
53.	*Cost*
54.	*Cost estimation*
55.	*Parametric models*
56.	*Experience*
57.	**Software Maintenance Measurements**
58.	*Establishing a Metrics Program*
59.	*Specific Measures*
60.	**Techniques for Maintenance**
61.	*Program Comprehension*
62.	*Re-engineering*
63.	*Reverse Engineering*
64.	*Impact Analysis*
65.	**Resources**

66. Introduction to Software Maintenance

67. The area of software maintenance and evolution of systems was first addressed by Lehman in 1969. His research led to an investigation of the evolution of OS/360 [19] and continues today on the Feedback, Evolution, and Software Technology (FEAST) research at Imperial College, England.

68. Over a period of twenty years, that research led to the formulation of eight Laws of Evolution [20]. Simply put, Lehman stated that maintenance is really evolutionary developments and that maintenance decisions are aided by understanding what happens to systems (and software) over time. Others state that maintenance is really continued development, except that there is an extra input (or constraint) – the existing software system.

69. Lehman's Laws of Evolution are generally accepted by the software engineering community and these clearly depict what happens over time. Key points from Lehman include that large systems are never complete and continue to evolve. As they evolve, they grow more complex unless some action is taken to reduce the complexity. As systems demonstrate regular behavior and trends, these can be measured and predicted. Pfleeger [25], Sommerville [33], and Arthur [3] have excellent discussions regarding software evolution.

70. A common perception of maintenance is that it is merely fixing bugs. However, studies over the years have indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions [33] [29] [28]. This perception is perpetuated by users submitting problem reports that in reality are major enhancements to the system. This "lumping of enhancement requests with problems" contributes to some of the misconceptions regarding maintenance. Software evolves over its life cycle, as evidenced by the fact that over 80% of the effort after initial delivery goes to implement non-corrective actions. Thus, maintenance is similar to software development. There is, however, another input or constraint – the existing system.

71. The focus of software development is to solve problems through producing code. The generated code implements stated requirements and should operate correctly. Maintenance is different than development [25]. Maintainers look back at development products and also the present by working with users and operators. Maintainers also look forward to anticipate problems and to consider functional changes. Pfleeger [25] states that maintenance has a broader scope, with more to track and control. Thus, configuration management is an important aspect of software evolution and maintenance.

72. Maintenance, however, must learn from the development effort. For the maintenance effort to succeed there should be contact with the developers and early involvement is encouraged. Maintenance must take the products of the development, e.g., code, documentation, and evolve/maintain them over the life cycle.

73. Need For Maintenance

74. Maintenance is needed to ensure that the system continues to satisfy user requirements. The system changes due to corrective and non-corrective software actions. Maintenance must be performed in order to:

75. - Correct errors.

76. - Correct design flaws.

77. - Interface with other systems that are new or changed.

78. - Make enhancements.

79. - Make necessary changes to the system.

80. - Make changes in files or databases.

81. - Improve the design.

82. - Convert programs so that different hardware, software, system features, and telecommunications facilities can be used.

83. The four major aspects that evolution and maintenance focus on are [25]:

84. - Maintaining control over the system's day-to-day functions.

85. - Maintaining control over system modification.

86. - Perfecting existing acceptable functions.

87. - Preventing system performance from degrading to unacceptable levels.

88. Accordingly, software must evolve and be maintained.

89. Categories of maintenance

90. Lehman developed the concept of software evolution. E. B. Swanson of UCLA was one of the first to examine what really happens in evolution and maintenance, using empirical data from industry maintainers. Swanson believed that, by studying the maintenance phase of the life cycle, a better understanding of the maintenance phase would result. Swanson was able to create three different categories of maintenance. These are reflected in software maintenance standards such as, IEEE 1219 [14] and ISO/IEC 14764 [16], as well as numerous texts. Swanson's categories of maintenance and his definitions are as follows:

91. - Corrective maintenance. Reactive modification of a software product performed after delivery to correct discovered faults.

92. - Adaptive maintenance. Modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment.

93. - Perfective maintenance. Modification of a software product after delivery to improve performance or maintainability.

94. The ISO Standard on Software Maintenance [16] refers to Adaptive and Perfective maintenance as enhancements. Another type of maintenance, preventive maintenance, is defined in the IEEE Standard on Software Maintenance [14] and the ISO Standard on Software [16]. Preventive maintenance is defined as maintenance performed for the purpose of preventing problems before they occur. This type of maintenance could easily fit under corrective maintenance but the international community, and in particular those who are concerned about safety, classify preventive as a separate type of maintenance.

95. Of note is that Pfleeger [25], Sommerville [33], and others address that the corrective portion of maintenance is only about 20% of the total maintenance effort. The remaining 80% is for enhancements, i.e., the adaptive and perfective categories of maintenance. This further substantiates Lehman's Laws of Evolution.

96. Maintenance Activities

97. Maintenance activities are similar to those of software development. Maintainers perform analysis, design, coding, testing, and documenting. Maintainers must track requirements just as they do in development. However, for software maintenance, the activities involve processes unique to maintenance.

98. Unique Activities

99. Maintainers must possess an intimate knowledge of the code's structure and content [25]. Unlike software development, maintainers must perform impact analysis. Analysis is performed in order to determine the cost of making a change. The change request, sometimes called a modification request and often called a problem report, must first be analyzed and translated into software terms [11]. The maintainer then identifies the affected components. Several potential solutions are provided and then a recommendation is made as to the best course of action.

100. *Supporting Activities*

101. Maintainers must also perform supporting activities such as configuration management (CM), verification and validation, quality assurance, reviews, audits, operating a help desk, and conducting user training. The IEEE Standard for Software Maintenance, IEEE 1219 [14], describes CM as a critical element of the maintenance process. CM procedures should provide for the verification, validation, and certification of each step required to identify, authorize, implement, and release the software product. Training of maintainers, a supporting process, is also a needed activity [28] [33] [24]. Maintenance also includes activities such as planning, migration, and retiring of systems [14] [28] [16] [15].

102. Configuration Management. It is not sufficient to simply track modification requests or problem reports. The software product and any changes made to it must be controlled. This control is established by implementing and enforcing an approved software configuration management process (SCM). The SCM process is implemented by developing and following a CM Plan and operating procedures.

103. Quality. Quality should be built into the software maintenance processes. The complexity of the software should be reduced to improve the quality of the software product. Software inspections should be used to improve quality. Quality of Service Agreements should be used to aid in quality improvement.

104. *Maintenance Planning Activity*

105. An important activity for software maintenance is planning. Whereas developments typically can last for 1-2 years, the operation and maintenance phase typically lasts for many years. Maintenance is performed during the operation and maintenance phase [25]. Maintenance planning should begin with the decision to develop a new system. A concept and then a maintenance plan should be developed. The concept for maintenance should address:

106. ◆ The scope of software maintenance.

107. ◆ The tailoring of the postdelivery process.

108. ◆ The designation of who will provide maintenance.

109. ◆ An estimate of life cycle costs.

110. Once the maintenance concept is determined, the next step is to develop the maintenance plan. The maintenance plan should be prepared during software development and should specify how users will request modifications or report problems. Maintenance planning [28] is addressed in IEEE 1219 [14] and ISO/IEC 14764 [16]. ISO/IEC 14764 [16] provides guidelines for a maintenance plan.

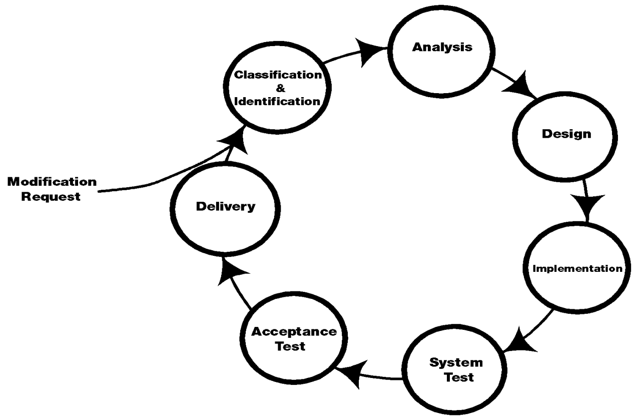
111. Maintenance Process

112. The need for software processes is well documented. The Software Engineering Institute’s Software Capability Maturity Model (CMM) provides a means to measure levels of maturity. Of importance, is that there is a direct correlation between levels of maturity and cost savings. The higher the level of maturity, the greater the cost savings. The CMM applies equally to maintenance and maintainers should have a documented maintenance process

113. *Maintenance Process Models*

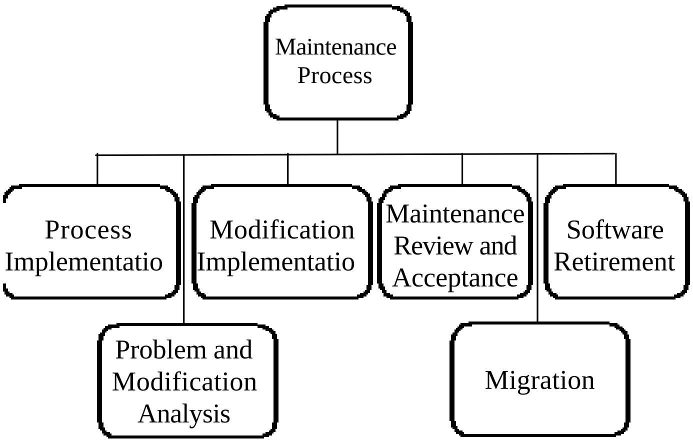
114. Process models provide needed operations and detailed inputs/outputs to those operations. Maintenance process models are provided in the software maintenance standards, IEEE 1219 [14] and ISO/IEC 14764 [16].

115. The maintenance process model described in IEEE 1219 [14], the Standard for Software Maintenance, starts the software maintenance effort during the post-delivery stage and discusses items such as planning for maintenance and metrics outside the process model. That process model with the IEEE maintenance phases is depicted in Figure 3.1.



116. **Figure 3.1: The IEEE Maintenance Process**

117. ISO/IEC 14764 [16] is an elaboration of the maintenance process of ISO/IEC 12207 [15]. The activities of the maintenance process are similar although they are aggregated a little differently. The maintenance process activities developed by ISO/IEC are shown in Figure 3.2.


Maintenance Process
Process Implementatio
Modification Implementatio
Maintenance Review and Acceptance
Software Retirement
Problem and Modification Analysis
Migration

118. **Figure 3.2: IEEE Maintenance Process Activities**

119. Each of these primary software maintenance activities is further broken down into tasks.

120. Process Implementation tasks are:

121. ◆ Maintenance planning and procedures.

122. ◆ Procedures for Modification Requests.

123. ◆ Interface with CM.

124. Problem and Modification tasks are:

125. ◆ Perform initial analysis.

126. ◆ Verify the problem.

127. ◆ Develop options for implementing the modification.

128. ◆ Document the results.

129. ◆ Obtain approval for modification option.

130. Modification Implementation tasks are:

131. ◆ Perform detailed analysis.

132. ◆ Develop, code, and test the modification.

133. Maintenance Review/Acceptance tasks are:

134. ◆ Conduct reviews.

135. ◆ Obtain approval for modification.

136. Migration tasks are:

137. ◆ Ensure that migration is in accordance with the Standard.

138. ◆ Develop a migration plan.

139. ◆ Notify users of migration plans.

140. ◆ Conduct parallel operations.

141. ◆ Notify user that migration has started.

142. ◆ Conduct a post-operation review.

143. ◆ Ensure that old data is accessible.

144. Software Retirement tasks are:

145. ◆ Develop a retirement plan.

146. ◆ Notify users of retirement plans.

147. ◆ Conduct parallel operations.

148. ◆ Notify user that retirement has started.

149. ◆ Ensure that old data is accessible.

150. Takang and Grubb [35] provide a history of maintenance process models leading up to the development of the IEEE and ISO/IEC process models. A good overview of a generic maintenance process is given by Sommerville [33].

151. Organization Aspect of Maintenance

152. The team that develops the software is not always used to maintain the system once it is operational.

153. The Maintainer

154. Often, a separate team (or maintainer) is employed to ensure that the system runs properly and evolves to satisfy changing needs of the users. There are many pros and cons to having the original developer or a separate team

maintain the software [25] [28] [24]. That decision should be made on a case-by-case basis.

155. *Outsourcing*

156. Outsourcing of maintenance is becoming a major industry. Large corporations are outsourcing entire operations, including software maintenance. Dorfman and Thayer [11] provide some guidance in the area of outsourcing maintenance.

157. *Organizational Structure*

158. Based on the fact there are almost as many organizational structures as there are software maintenance organizations, an organizational structure for maintenance is best developed on a case-by-case basis. What is important is the delegation or designation of maintenance responsibility to a group [28], regardless of the organizational structure.

159. Problems of Software Maintenance

160. It is important to understand that software evolution and maintenance provides unique technical and management problems for software engineers. Trying to find a defect in a 500K line of code system that the maintainer did not develop is a challenge for the maintainer. Similarly, competing with software developers for resources is a constant battle. The following discusses some of the technical and management problems relating to software evolution and maintenance.

161. *Technical*

162. Limited Understanding [25]. Several studies indicate that some 40% to 60% of the maintenance effort is devoted to understanding the software to be modified. Thus, the topic of program comprehension is one of extreme interest to maintainers. It is often difficult to trace the evolution of the software through its versions, changes are not documented, and the developers are usually not around to explain the code. Thus, maintainers have a limited understanding of the software and must learn the software on their own.

163. Testing. The cost of repeating full testing on a major piece of software can be significant in terms of time and money. Thus, determining a sub-sets of tests to perform in order to verify changes are a constant challenge to maintainers [11]. Finding time to test is often difficult [25].

164. Impact Analysis. The software and the organization must both undergo impact analysis. Critical skills and processes are needed for this area. Impact analysis is necessary for risk abatement.

165. Maintainability. The IEEE Computer Society defines maintainability as the ease with which software can be maintained, enhanced, adapted, or corrected to satisfy specified requirements. Maintainability features must be incorporated into the software development effort to reduce life-cycle costs. If this is done, the quality of evolution and maintenance of the code can improve. Maintainability is often a problem in maintenance because maintainability is not incorporated into the software development process, documentation is non-existent, and program comprehension is difficult. Means to improve maintainability, and thereby constrain life-cycle costs, is to define coding standards, documentation standards, and standard test tools in the software development phase of the life-cycle.

166. Management

167. Alignment with organizational issues. Dorfman and Thayer [11] relate that return on investment is not clear with maintenance. Thus, there is a constant struggle to obtain resources.

168. Staffing. Maintenance personnel often are viewed as second class citizens [25] and morale suffers [11]. Maintenance is not viewed as glamorous work. Deklava provides a list of staffing related problems based on survey data [10].

169. Process issues. Maintenance requires several activities that are not found in software development, e.g. , help desk support. These present challenges to management [11].

170. Maintenance Cost and Maintenance Cost Estimation

171. Maintenance costs are high due to all the problems of maintaining a system [25]. Software engineers must understand the different categories of maintenance, previously discussed, in order to address the cost of maintenance. For planning purposes, estimating

costs is an important aspect of software maintenance.

172. *Cost*

173. Maintenance now consumes a major share of the life cycle costs. Prior to the mid-1980s, the majority of costs went to development. Since that time, maintenance consumes the majority of life-cycle costs. Understanding the categories of maintenance helps to understand why maintenance is so costly. Also understanding the factors that influence the maintainability of a system can help to contain costs. Pfleeger [25] and Sommerville [33] address some of the technical and non-technical factors affecting maintenance.

174. Impact analysis identifies all systems and system products affected by a change request and develops an estimate of the resources needed to accomplish the change [3]. It is performed after a change request enters the configuration management process. It is used in concert with the cost estimation techniques discussed below.

175. *Cost estimation*

176. Maintenance cost estimates are affected by many technical and non-technical factors. Primary approaches to cost estimating include use of parametric models and experience. Most often a combination of these is used to estimate costs.

177. *Parametric models*

178. The most significant and authoritative work in the area of parametric models for estimating was performed by Boehm [5]. His COCOMO model, derived from COnstructive COst MOdel, puts the software life cycle and the quantitative life-cycle relationships into a hierarchy of software cost-estimation models [25] [33] [28]. Of significance is that data from past projects is needed in order to use the models. Jones [18] discusses all aspects of estimating costs including function points, and provides a detailed chapter on maintenance estimating.

179. *Experience*

180. Experience should be used to augment data from parametric models. Sound judgement, reason, a work breakdown structure, educated guesses, and use of empirical/historical data are several approaches. Clearly the best approach to maintenance estimation is to use empirical data and experience. That data should be provided as a result of a metrics program. In practice, cost estimation relies much more on experience than parametric models.

181. Software Maintenance Measurements

182. Software life cycle costs are growing and a strategy for maintenance is needed. Software measurement or software metrics need to be a part of that strategy. Software measurement is the result of a software measurement process. Software metrics are often synonymous with software measurement. Grady and Caswell [12] discuss establishing a corporate-wide metrics program. Software metrics are vital for software process improvement but the process must be measurable.

183. Takang and Grubb [35] state that measurement is undertaken for evaluation, control, assessment, improvement, and prediction. A program must be established with specific goals in mind.

184. *Establishing a metrics program*

185. Successful implementation strategies were used at Hewlett-Packard [12] and at the NASA/Software Engineering Laboratory [8]. Common to many approaches is to use the Goal, Question, Metric (GQM) paradigm put forth by Basili [34]. This approach states that a metrics program would consist of: identifying organizational goals; defining the questions relevant to the goals; and then selecting measures that answer the questions.

186. The *IEEE Standard For a Software Quality Metrics Methodology, ANSI/IEEE 1061-1998,* [1] provides a methodology for establishing quality requirements and identifying, implementing, analyzing and validating process and product software quality metrics. The methodology applies to all software at all phases of any software life cycle and is a valuable resource for software evolution and maintenance.

187. There are two primary lessons learned from practitioners about metrics programs. The first is to focus on a few key characteristics. The second is not to measure everything. Most organizations collect too much. Thus, a good approach is to evolve a metrics program and to use the GQM paradigm.

188. *Specific Measures*

189. There are metrics that are common to all efforts and the Software Engineering Institute (SEI) identified these as: size; effort; schedule; and quality [28]. Those metrics are a good starting point for a maintainer.

190. Takang and Grubb [35] group metrics into areas of: size; complexity; quality; understandability; maintainability; and cost estimation.

191. Documentation regarding specific metrics to use in maintenance is not often published. Typically generic software engineering metrics are used and the maintainer determines which ones are appropriate for their organization. IEEE 1219 [14] provides suggested metrics for software programs. Stark, et al [34] provides a suggested list of maintenance metrics used at NASA's Mission Operations Directorate. That list includes:

192. ◆ Software size

193. ◆ Software staffing

194. ◆ Maintenance request processing

195. ◆ Software enhancement processing

196. ◆ Computer resource scheduling

197. ◆ Fault density

198. ◆ Software volatility

199. ◆ Discrepancy report open duration

200. ◆ Break/fix ration

201. ◆ Software reliability

202. ◆ Design complexity

203. ◆ Fault type distribution

204. *Techniques for Maintenance*

205. Effective software maintenance is performed using techniques specific to maintenance. The following provides some of the best practice techniques used by maintainers.

206. *Program Comprehension*

207. Studies indicate that 40% to 60% of a maintenance programmer's time is spent trying to understand the code. Time is spent in reading and comprehending programs in order to implement changes. Browsers are a key tool in program comprehension. Based on the importance of this subtopic, an annual IEEE workshop is now held to address program comprehension [11]. Additional research and experience papers regarding comprehension are found in the annual proceedings of the IEEE Computer Society's International Conference on Software Maintenance (ICSM). Takang and Grubb [35] provide a detailed chapter on comprehension.

208. *Re-engineering*

209. Re-engineering is the examination and alteration of the subject system to reconstitute it in a new form, and the subsequent implementation of the new form. Dorfman and Thayer [11] state that re-engineering is the most radical (and expensive) form of alteration. Others believe that re-engineering can be used for minor changes. Re-engineering is often not undertaken to improve maintainability but is used to replace aging legacy systems. Arnold [2] provides a comprehensive compendium of topics, e.g., concepts, tools and techniques, case studies, and risks and benefits associated with re-engineering.

210. *Reverse engineering*

211. Reverse engineering is the process of analyzing a subject system to identify the system's components and their inter-relationships and to create representations of the system in another form or at higher levels of abstraction. Reverse engineering is passive, it does not change the system, or result in a new one. A simple reverse engineering effort may merely produce call graphs and control flow graphs from source code. One type of reverse engineering is redocumentation. Another type is design recovery [11].

212. *Impact Analysis*

213. Impact analysis identifies all systems and system products affected by a change request and develops an estimate of the resources needed to accomplish the change [3]. It is performed after a change request enters the configuration management process. Arthur [3] states that the objectives of impact analysis are:

214. ◆ Determine the scope of a change in order to plan and implement work.

215. ◆ Develop accurate estimates of resources needed to perform the work.

216. ◆ Analyze the cost/benefits of the requested change.

217. ◆ Communicate to others the complexity of a given change.

218. *Resources*

219. Beside the references listed in this paper, there are other resources available to learn more about software maintenance. The IEEE Computer Society sponsors the annual *International Conference on Software Maintenance (ICSM).* That conference started in 1983 and continues today. ICSM provides a Proceedings, which incorporates numerous research and practical industry papers concerning evolution and maintenance topics. Other venues, which address these topics, include:

220. - The Workshop on Software Change and Evolution (SCE).

221. - The International Workshop on the Principles of Software Evolution (IWPSE).

222. - Manny Lehman's work on the FEAST project at the Imperial College in England continues to provide valuable research into software evolution.

223. - The Research Institute for Software Evolution (RISE) at the University of Durham, England, concentrates its research on software maintenance and evolution.

224. The *Journal of Software Maintenance*, published by John Wiley & Sons, also is an excellent resource.

225. Rationale for the breakdown

226. The breakdown of topics for software maintenance is a decomposition of software engineering topics that are "generally accepted" in the software maintenance community. They are general in nature. There is agreement in the literature and in the standards on the topics.

227. A detailed discussion of the rationale for the proposed breakdown, keyed to the SWEBOK development criteria, is given in Appendix B. The following is a narrative description of the rationale for the breakdown.

228. The Introduction to Software Maintenance was selected as the initial topic in order to introduce the topic. The subtopics are needed to emphasis why there is a need for maintenance. Categories are critical to understand the underlying meaning of maintenance. All pertinent texts use a similar introduction.

229. The Maintenance Activities topic is needed to differentiate maintenance from development and to show the relationship to other software engineering activities. Maintenance Process is needed to provide the current references and standards needed to implement the maintenance process.

230. Every organization is concerned with who will perform maintenance. The Organizational Aspect of Maintenance provides some options. There is always a discussion that maintenance is hard. The topic on the Problems of Software Maintenance was chosen to ensure that the software engineers fully comprehended these problems.

231. Every software maintenance reference discusses the fact that maintenance consumes a large portion of the life cycle costs. The topic on Cost and Cost Estimation was provided to ensure that the readers select references to help with this difficult task.

232. The Software Maintenance Measurements topic is one that is not addressed very well in the literature. Most maintenance books barely touch on the topic. Measurement information is most often found in generalized measurement books. This topic was chosen to highlight the need for unique maintenance metrics and to provide specify maintenance measurement references.

233. The Techniques topic was provided to introduce some of the generally accepted techniques used in maintenance operations.

234. Finally, there are other resources besides textbooks and periodicals that are useful to software engineers who wish to learn more about software maintenance. The Resources topic was provided to list these additional resources.

235. Coverage of the software breakdown topics by the recommended references

236. The cross-reference is shown in Appendix A.

237. 4. RECOMMENDED REFERENCES FOR SOFTWARE MAINTENANCE

238. The following set of references provides the best reading material to acquire knowledge on specific topics identified in the breakdown. They were chosen to provide coverage of all aspects of software maintenance. Priority was given to standards, maintenance specific

publications, and then general software engineering publications.

239. References

240. [1] ANSI/IEEE STD 1061. *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Computer Society Press, 1998, pp. 3-13.

241. [2] R. S. Arnold. *Software Reengineering*. IEEE Computer Society, 1992, pp. 3-22

242. [3] L. J. Arthur. *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons, 1988, pp. 1-6, 39-57.

243. [5] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981, pp. 534-553.

244. [8] D. N. Card and R. L. Glass, *Measuring Software Design Quality*, Prentice Hall, 1990, pp. 15-22.

245. [10] S. M. Dekleva. Delphi Study of Software Maintenance Problems. *Proceedings of the International Conference on Software Maintenance*, 1992, pp. 10-17.

246. [11] M. Dorfman and R. H. Thayer. *Software Engineering*. IEEE Computer Society Press, 1997, pp. 289-307.

247. [12] R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a Company-wide Program*. Prentice-Hall, 1987, Chapter 1.

248. [14] *IEEE STD 1219: Standard for Software Maintenance*, 1993, pp. 1-17.

249. [15] *ISO/IEC 12207: Information Technology-Software Life Cycle Processes*, 1995, pp. 6-9.

250. [16] *ISO/IEC 14764: Software Engineering-Software Maintenance*, 2000, pp. 1-35.

251. [18] T. C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998, pp. 595-636.

252. [19] M. M. Lehman and L. A. Belady, *Program Evolution – Processes of Software Change*, Academic Press Inc. (London) Ltd., 1985.

253. [20] M . M Lehman, Laws of Software Evolution Revisited, EWSPT96, October 1996, LNCS 1149, Springer Verlag, 1997, pp 108-124, pp. 108-124.

254. [24] G. Parikh. *Handbook of Software Maintenance*. John Wiley & Sons, 1986, pp. 361, 126-129.

255. [25] S. L. Pfleeger. *Software Engineering—Theory and Practice*. Prentice Hall, 1998, pp. 420-422, 422-423, 424, 425, 424-431, 427-436.

256. [28] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing your Software Investment*. Wiley, 1997, pp. 20-27, 29-36, 89-99, 92-93, 103-106, 223-225, 309-322.

257. [29] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, fourth edition, 1997, pp. 762-763

258. [31] N. F. Schneidewind. *The State of Software Maintenance*. *Proceedings of the IEEE*, 77(4), 1987, pp. 618-624.

259. [33] I. Sommerville. *Software Engineering*. McGraw-Hill, fifth edition, 1996, pp. 121-124, 660-661, 662-663, 664-666, 666-670.

260. [34] G. E. Stark, L. C. Kern, and C. V. Vowell. *A Software Metric Set for Program Maintenance Management*. Journal of Systems and Software, 1994.

261. [35] A. Takang and P. Grubb. *Software Maintenance Concepts and Practice*. International Thomson Computer Press, 1997, 117-126, 117-130, 155-156.

262. 5. LIST OF FURTHER READINGS

263. A list of additional readings, called Further Readings, is provided to provide additional reference material for the Knowledge Area of Software Maintenance. These references also contain generally accepted knowledge.

264. References

265. [1] ANSI/IEEE STD 1061. *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Computer Society Press, 1998.

266. [2] R. S. Arnold. *Software Reengineering*. IEEE Computer Society, 1992.

267. [3] L. J. Arthur. *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons, 1988.

268. [4] V. R. Basili, "Quantitative Evaluation of Software Methodology," *Proceedings First Pan-Pacific Computer Conference,* September 1985.

269. [5] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

270. [6] C. Boldyreff, E. Burd, R. Hather, R. Mortimer, M. Munro, and E. Younger, "The AMES Approach to Application Understanding: A Case Study," *Proceedings of the*

International Conference on Software Maintenance-1995, IEEE Computer Society Press, Los Alamitos, CA, 1995.

271. [7] M.A. Capretz and M. Munro, "Software Configuration Management Issues in the Maintenance of Existing Systems," *Journal of Software Maintenance*, Vol 6, No.2, 1994.

272. [8] D. N. Card and R. L. Glass, *Measuring Software Design Quality*, Prentice Hall, 1990.

273. [9] J. Cardow, "You Can't Teach Software Maintenance!," *Proceedings of the Sixth Annual Meeting and Conference of the Software Management Association*, 1992.

274. [10] S. M. Dekleva. Delphi Study of Software Maintenance Problems. *Proceedings of the International Conference on Software Maintenance,* 1992.

275. [11] M. Dorfman and R. H. Thayer. *Software Engineering*. IEEE Computer Society Press, 1997.

276. [12] R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a Company-wide Program.* Prentice-Hall, 1987.

277. [13] R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

278. [14] *IEEE STD 1219: Standard for Software Maintenance,* 1993.

279. [15] *ISO/IEC 12207: Information Technology-Software Life Cycle Processes, 1995.*

280. [16] *ISO/IEC 14764: Software Engineering-Software Maintenance,* 2000.

281. [17] ISO/IEC TR 15271, *Information Technology - Guide for ISO/IEC 12207, (Software Life Cycle Process)*

282. [18] T. C. Jones. *Estimating Software Costs.* McGraw-Hill, 1998.

283. [19] M. M. Lehman and L. A. Belady, *Program Evolution – Processes of Software Change*, Academic Press Inc. (London) Ltd., 1985.

284. [20] M. M Lehman, Laws of Software Evolution Revisited, EWSPT96, October 1996, LNCS 1149, Springer Verlag, 1997.

285. [21] T.M. Khoshgoftaar, R.M. Szabo, and J.M. Voas, "Detecting Program Module with Low Testability," *Proceedings of the International Conference on Software Maintenance-1995*, IEEE Computer Society Press, Los Alamitos, CA, 1995.

286. [22] P.W. Oman, J. Hagemeister, and D. Ash, *A Definition and Taxonomy for Software Maintainability*, University of Idaho, Software Engineering Test Lab, Technical Report, 91-08 TR, November 1991.

287. [23] P. Oman and J. Hagemeister, "Metrics for Assessing Software System Maintainability," *Proceedings of the International Conference on Software Maintenance-1992*, IEEE Computer Society Press, Los Alamitos, CA, 1992.

288. [24] G. Parikh. *Handbook of Software Maintenance*. John Wiley & Sons, 1986.

289. [25] S. L. Pfleeger. *Software Engineering—Theory and Practice.* Prentice Hall, 1998.

290. [26] T.M. Pigoski, "Maintainable Software: Why You Want It and How to Get It," *Proceedings of the Third Software Engineering Research Forum-November 1993*, University of West Florida Press, Pensacola, FL, 1993.

291. [27] T.M. Pigoski. "Software Maintenance," *Encyclopedia of Software Engineering,* John Wiley & Sons, New York, NY, 1994.

292. [28] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing your Software Investment.* Wiley, 1997.

293. [29] R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, fourth edition, 1997.

294. [30] S. R. Schach, *Classical and Object-Oriented Software Engineering With UML and C++*, McGraw-Hill, 1999

295. [31] N. F. Schneidewind. *The State of Software Maintenance*. *Proceedings of the IEEE*, 1987.

296. [32] S. L. Schneberger, *Client/Server Software Maintenance*, McGraw-Hill, 1997.

297. [33] I. Sommerville. *Software Engineering.* McGraw-Hill, fifth edition, 1996.

298. [34] G. E. Stark, L. C. Kern, and C. V. Vowell. *A Software Metric Set for Program Maintenance Management*. Journal of Systems and Software, 1994.

299. [35] A. Takang and P. Grubb. *Software Maintenance Concepts and Practice.* International Thomson Computer Press, 1997.

300. [36] J.D. Vallett, S.E. Condon, L. Briand, Y.M. Kim and V.R. Basili, "Building on

Experience Factory for Maintenance," *Proceedings of the Software Engineering Workshop*, Software Engineering Laboratory, 1994.

301. 6. References used to write and justify the description for Software Maintenance

302. The following set of references was chosen to provide coverage of all aspects of software evolution and maintenance. Priority was given to standards, maintenance specific publications, and then general software engineering publications.

303. References

304. [1] ANSI/IEEE STD 1061. *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Computer Society Press, 1998.

305. [2] R. S. Arnold. *Software Reengineering*. IEEE Computer Society, 1992.

306. [3] L. J. Arthur. *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons, 1988.

307. [4] V. R. Basili, "Quantitative Evaluation of Software Methodology," *Proceedings First Pan-Pacific Computer Conference,* September 1985.

308. [5] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

309. [6] C. Boldyreff, E. Burd, R. Hather, R. Mortimer, M. Munro, and E. Younger, "The AMES Approach to Application Understanding: A Case Study," *Proceedings of the International Conference on Software Maintenance-1995*, IEEE Computer Society Press, Los Alamitos, CA, 1995.

310. [7] M.A. Capretz and M. Munro, "Software Configuration Management Issues in the Maintenance of Existing Systems," *Journal of Software Maintenance,* Vol 6, No.2, 1994.

311. [8] D. N. Card and R. L. Glass, *Measuring Software Design Quality*, Prentice Hall, 1990.

312. [9] J. Cardow, "You Can't Teach Software Maintenance!," *Proceedings of the Sixth Annual Meeting and Conference of the Software Management Association*, 1992.

313. [10] S. M. Dekleva. Delphi Study of Software Maintenance Problems. *Proceedings of the International Conference on Software Maintenance,* 1992.

314. [11] M. Dorfman and R. H. Thayer. *Software Engineering*. IEEE Computer Society Press, 1997.

315. [12] R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a Company-wide Program.* Prentice-Hall, 1987.

316. [13] R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

317. [14] *IEEE STD 1219: Standard for Software Maintenance*, 1993.

318. [15] *ISO/IEC 12207: Information Technology-Software Life Cycle Processes, 1995.*

319. [16] *ISO/IEC 14764: Software Engineering-Software Maintenance,* 2000.

320. [17] ISO/IEC TR 15271, *Information Technology - Guide for ISO/IEC 12207, (Software Life Cycle Process).*

321. [18] T. C. Jones. *Estimating Software Costs.* McGraw-Hill, 1998.

322. [19] M . M Lehman, Laws of Software Evolution Revisited, EWSPT96, October 1996, LNCS 1149, Springer Verlag, 1997.

323. [20] M. M. Lehman and L. A. Belady, *Program Evolution – Processes of Software Change*, Academic Press Inc. (London) Ltd., 1985.

324. [21] T.M. Khoshgoftaar, R.M. Szabo, and J.M. Voas, "Detecting Program Module with Low Testability," *Proceedings of the International Conference on Software Maintenance-1995*, IEEE Computer Society Press, Los Alamitos, CA, 1995.

325. [22] P.W. Oman, J. Hagemeister, and D. Ash, *A Definition and Taxonomy for Software Maintainability*, University of Idaho, Software Engineering Test Lab, Technical Report, 91-08 TR, November 1991.

326. [23] P. Oman and J. Hagemeister, "Metrics for Assessing Software System Maintainability," *Proceedings of the International Conference on Software Maintenance-1992,* IEEE Computer Society Press, Los Alamitos, CA, 1992.

327. [24] G. Parikh. *Handbook of Software Maintenance*. John Wiley & Sons, 1986.

328. [25] S. L. Pfleeger. *Software Engineering—Theory and Practice.* Prentice Hall, 1998.

329. [26] T.M. Pigoski, “Maintainable Software: Why You Want It and How to Get It,” *Proceedings of the Third Software Engineering Research Forum-November 1993*, University of West Florida Press, Pensacola, FL, 1993.

330. [27] T.M. Pigoski. “Software Maintenance,” *Encyclopedia of Software Engineering,* John Wiley & Sons, New York, NY, 1994.

331. [28] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing your Software Investment.* Wiley, 1997.

332. [29] R. S. Pressman. *Software Engineering: A Practitioner’s Approach.* McGraw-Hill, fourth edition, 1997.

333. [30] S. R. Schach, *Classical and Object-Oriented Software Engineering With UML and C++,* McGraw-Hill, 1999

334. [31] N. F. Schneidewind. *The State of Software Maintenance*. *Proceedings of the IEEE*, 1987.

335. [32] S. L. Schneberger, *Client/Server Software Maintenance,* McGraw-Hill, 1997.

336. [33] I. Sommerville. *Software Engineering.* McGraw-Hill, fifth edition, 1996.

337. [34] G. E. Stark, L. C. Kern, and C. V. Vowell. *A Software Metric Set for Program Maintenance Management*. Journal of Systems and Software, 1994.

338. [35] A. Takang and P. Grubb. *Software Maintenance Concepts and Practice.* International Thomson Computer Press, 1997.

339. [36] J.D. Vallett, S.E. Condon, L. Briand, Y.M. Kim and V.R. Basili, “Building on Experience Factory for Maintenance,” *Proceedings of the Software Engineering Workshop,* Software Engineering Laboratory, 1994.

340. Appendix A – Coverage of the breakdown topics by the recommended references

TOPIC	REFERENCE																					
	AI 98	Arn 92	Art 88	Boe 81	CG 90	Dek 92	DT 97	GC 87	IEEE 1219	ISO 12207	ISO 14764	Jon 98	LB 85	Leh 97	Par 86	Pfl 98	Pig 97	Pre 97	Sch 87	Som 96	SKV 94	TG 97
	[1]	[2]	[3]	[5]	[8]	[10]	[11]	[12]	[14]	[15]	[16]	[18]	[19]	[20]	[24]	[25]	[28]	[29]	[31]	[33]	[34]	[35]
Introduction to Software Maintenance			X	X			X		X	X	X	X	X	X	X	X	X	X	X	X	X	X
Need for Maintenance										X			X	X		X	X			X		X
Categories of Maintenance			X													X	X			X		
Maintenance Activities			X				X			X		X	X	X		X	X					
Unique Activities			X						X	X	X					X	X					
Supporting Activities	X		X				X		X	X	X				X	X	X	X		X		X
Configuration Management			X				X		X	X						X	X	X		X		X
Quality	X		X				X		X	X						X		X		X		X
Maintenance Planning Activity									X	X	X					X	X					
Maintenance Process			X				X		X	X	X					X	X		X	X		X
Maintenance Process Models							X		X	X	X						X					X
Organization Aspect of Maintenance							X				X				X	X	X	X				X
The Maintainer							X				X				X		X	X				X
Outsourcing							X				X						X					
Organizational Structure																	X					
Problems of Software Maintenance																X	X					X
Technical							X									X						X
Limited Understanding		X														X				X		X
Testing			X				X		X													
Impact Analysis							X		X							X						
Maintainability	X										X					X	X			X		
Management							X									X						X
Alignment with organizational issues							X									X		X		X		
Staffing						X									X		X			X		X
Process issues							X															
Maintenance Cost and Maintenance Cost Estimation			X	X					X		X				X	X	X					
Cost			X	X												X	X	X		X		
Cost estimation				X								X				X	X	X		X		

TOPIC	REFERENCE																					
	AI 98	Arn 92	Art 88	Boe 81	CG 90	Dek 92	DT 97	GC 87	IEEE 1219	ISO 12207	ISO 14764	Jon 98	LB 85	Leh 97	Par 86	Pfl 98	Pig 97	Pre 97	Sch 87	Som 96	SKV 94	TG 97
	[1]	[2]	[3]	[5]	[8]	[10]	[11]	[12]	[14]	[15]	[16]	[18]	[19]	[20]	[24]	[25]	[28]	[29]	[31]	[33]	[34]	[35]
Parametric models	X			X								X				X	X	X		X		
Experience				X	X				X								X					
Software Maintenance Measurements	X							X			X					X	X			X	X	X
Establishing a Metrics Program					X			X									X				X	
Specific Measures	X				X											X	X	X		X	X	X
Techniques for Maintenance		X										X										X
Program Comprehension		X					X										X					X
Re-engineering		X	X				X		X			X						X		X		X
Reverse Engineering		X					X		X			X						X				X
Impact Analysis			X													X						
Resources			X													X	X	X		X		X

341. APPENDIX B – BREAKDOWN RATIONALE

342. Please note that criteria are defined in Appendix A of entire Guide

343. Criterion (a): Number of topic breakdowns

344. One breakdown is provided.

345. Criterion (b): Reasonableness

346. The breakdowns are reasonable in that they cover the areas typically discussed in texts and standards, although there is less discussion regarding the pre-maintenance activities, e.g., planning. Other topics such as metrics are also often not addressed although they are getting more attention now.

347. Criterion (c): Generally Accepted

348. The breakdowns are generally accepted in that they cover the areas typically discussed in texts and standards.

349. Criterion (d): No specific Application Domains

350. No specific application domains are assumed.

351. Criterion (e): Compatibility with Various Schools of Thought

352. Software maintenance concepts are stable and mature.

353. Criterion (f): Compatible with Industry, Literature, and Standards

354. The breakdown was derived from the literature and key standards reflecting consensus opinion. The extent to which industry implements the software maintenance concepts in the literature and in standards varies by company and project.

355. Criterion (g): As Inclusive as Possible

356. The primary topics are addressed within the page constraints of the document.

357. Criterion (h): Themes of Quality, Measurement, and Standards

358. Quality, Measurement and standards are discussed.

359. Criterion (i): 2 to 3 levels, 5 to 9 topics at the first level

360. The proposed breakdown satisfies this criterion.

361. Criterion (j): Topic Names Meaningful Outside the Guide

362. Wording is meaningful. Version 0.5 review indicates that the wording id meaningful.

363. Criterion (l): Topics only sufficiently described to allow reader to select appropriate material

364. A tutorial on maintenance was not provided. Generally accepted concepts were introduced with appropriate references for additional reading were provided.

365. Criterion (m): Text on the Rationale Underlying the Proposed Breakdowns

366. The Introduction to Software Maintenance was selected as the initial topic in order to introduce the topic. The subtopics are needed to emphasis why there is a need for maintenance. Categories are critical to understand the underlying meaning of maintenance. All pertinent texts use a similar introduction.

367. The Maintenance Activities topic is needed to differentiate maintenance from development and to show the relationship to other software engineering activities. Maintenance Process is needed to provide the current references and standards needed to implement the maintenance process.

368. Every organization is concerned with who will perform maintenance. The Organizational Aspect of Maintenance provides some options. There is always a discussion that maintenance is hard. The topic on the Problems of Software Maintenance was chosen to ensure that the software engineers fully comprehended these problems.

369. Every software maintenance reference discusses the fact that maintenance consumes a large portion of the life cycle costs. The topic on Cost and Cost Estimation was provided to ensure that the readers select references to help with this difficult task.

370. The Software Maintenance Measurements topic is one that is not addressed very well in the literature. Most maintenance books barely touch on the topic. Measurement information is most often found in generalized measurement books. This topic was chosen to highlight the need for unique maintenance metrics and to provide specify maintenance measurement references.

371. The Techniques topic was provided to introduce some of the generally accepted techniques used in maintenance operations.

372. Finally, there are other resources besides textbooks and periodicals that are useful to software engineers who wish to learn more about software maintenance. This topic is provided to list these additional resources.

CHAPTER 7
SOFTWARE CONFIGURATION MANAGEMENT

John A. Scott and David Nisse
Lawrence Livermore National Laboratory
7000 East Avenue
P.O. Box 808, L-632
Livermore, CA 94550, USA
(925) 423-7655
scott7@llnl.gov

TABLE OF CONTENTS

1. INTRODUCTION

2. This paper presents an overview of the knowledge area of software configuration management for the Guide to the Software Engineering Body of Knowledge (SWEBOK) project. A breakdown of topics is presented for the knowledge area along with a succinct description of each topic. References are given to materials that provide more in-depth coverage of the key areas of software configuration management. Important knowledge areas of related disciplines are also identified.

3. Acronyms

4.	CCB	Configuration Control Board
5.	CM	Configuration Management
6.	FCA	Functional Configuration Audit
7.	PCA	Physical Configuration Audit
8.	SCI	Software Configuration Item
9.	SCR	Software Change Request
10.	SCM	Software Configuration Management
11.	SCMP	Software Configuration Management Plan
12.	SCSA	Software Configuration Status Accounting
13.	SDD	Software Design Description
14.	SQA	Software Quality Assurance
15.	SRS	Software Requirements Specification

16. DEFINITION OF THE SCM KNOWLEDGE AREA

17. A system can be defined as a collection of components organized to accomplish a specific function or set of functions [IEEE 610]. The configuration of a system is the function and/or physical characteristics of hardware, firmware, software or a combination thereof as set forth in technical documentation and achieved in a product [Buckley]. 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 accomplish 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 [Bersoff, (3)]. CM is formally defined [IEEE 610] as:

18.
> "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."

19. 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.

20. This paper presents a breakdown of the key software configuration management (SCM) concepts along with a succinct description of each concept. The concepts are generally accepted in that they cover the areas typically addressed in texts and standards. The descriptions cover the primary activities of SCM and are only intended to be sufficient for allowing the reader to select appropriate reference material according to the reader’s needs. The SCM activities are: the management of the software configuration management process, software configuration identification, software configuration control, software configuration status accounting, software configuration auditing, and software release management and delivery.

21. Figure 1 shows a stylized representation of these activities.

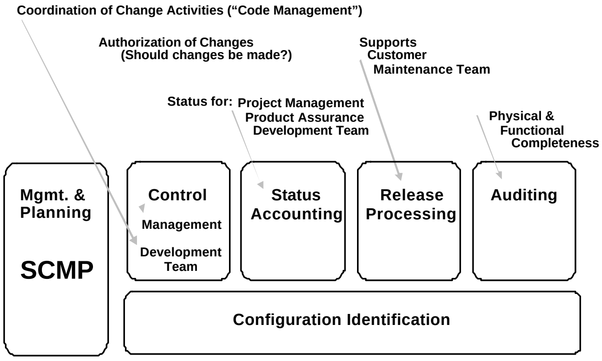

Coordination of Change Activities (“Code Management”)
Authorization of Changes (Should changes be made?)
Supports Customer Maintenance Team
Status for: Project Management Product Assurance Development Team
Physical & Functional Completeness
Mgmt. & Planning
SCMP
Control
Management
Development Team
Status Accounting
Release Processing
Auditing
Configuration Identification

Figure 1. SCM Activities

22. Following the breakdown, key references for SCM are listed along with a cross-reference of topics that each listed reference covers. Finally, topics in related disciplines that are important to SCM are identified.

23. BREAKDOWN OF TOPICS FOR SCM

24. Breakdown of Topics

25. An outline of the breakdown of topics is shown below. The following sections provide a brief description of each topic. The breakdown covers the concepts and activities of SCM. The variety of SCM tools and tool systems now available, as well as the variety of characteristics of the projects to which they are applied, may make the implementation of these concepts and the nature of the activities appear quite different from project to project. However, the underlying concepts and types of activities still apply.

26. I. Management of the SCM Process
27. A. Organizational Context for SCM
28. B. Constraints and Guidance for SCM
29. C. Planning for SCM
30. 1. SCM Organization and Responsibilities
31. 2. SCM Resources and Schedules
32. 3. Tool Selection and Implementation
33. 4. Vendor/Subcontractor Control
34. 5. Interface Control
35. D. Software Configuration Management Plan
36. E. Surveillance of Software Configuration Management
37. 1. SCM Metrics and Measurement
38. 2. In-Process Audits of SCM
39. II. Software Configuration Identification
40. A. Identifying Items to be Controlled
41. 1. Software Configuration
42. 2. Software Configuration Item
43. 3. Software Configuration Item Relationships
44. 4. Software Versions
45. 5. Baseline
46. 6. Acquiring Software Configuration Items
47. B. Software Library
48. III. Software Configuration Control
49. A. Requesting, Evaluating and Approving Software Changes
50. 1. Software Configuration Control Board
51. 2. Software Change Request Process
52. B. Implementing Software Changes
53. C. Deviations & Waivers
54. IV. Software Configuration Status Accounting
55. A. Software Configuration Status Information
56. B. Software Configuration Status Reporting
57. V. Software Configuration Auditing
58. A. Software Functional Configuration Audit
59. B. Software Physical Configuration Audit
60. C. In-process Audits of a Software Baseline
61. VI. Software Release Management and Delivery
62. A. Software Building
63. B. Software Release Management

64. *I. Management of the SCM Process*

65. Software configuration management is a supporting software life cycle process that benefits project and line management, development and maintenance activities, assurance activities, and the customers and users of the end product. From a management perspective, SCM controls the evolution of a

product by identifying its elements, managing and controlling change, and verifying, recording and reporting on configuration information. From the developer's perspective, SCM facilitates the 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 upon, the design and implementation of the SCM process.

66. I.A Organizational Context for SCM

67. To plan an SCM process for a project, it is necessary to understand the organizational structure and the relationships among organizational elements. SCM interacts with several other activities or organizational elements.

68. SCM, like other processes such as software quality assurance and software verification and validation, is categorized as a supporting life cycle process [ISO/IEC 12207]. The organizational elements responsible for these processes may be structured in various ways. Although the responsibility for performing certain SCM tasks might be assigned to other organizations, such as the development organization, the overall responsibility for SCM typically rests with a distinct organizational element or designated individual.

69. 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. Buckley [5] describes SCM within this context.

70. SCM is closely related to the software quality assurance (SQA) activity. The goals of SQA can be characterized [Humphrey] as monitoring the software and its development process, ensuring compliance with standards and procedures, and ensuring that product, process, and standards defects are visible to management. SCM activities are closely related to these SQA goals and, in some project contexts, e.g. see [IEEE 730], specific SQA requirements prescribe certain SCM activities.

71. SCM might also interface with an organization's quality assurance activity on issues such as records management and non-conforming 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 non-conforming items is usually the responsibility of the quality assurance activity, however, SCM might assist with tracking and reporting on software items that fall in this category.

72. Perhaps the closest relationship is with the software development and maintenance organizations. The environment for engineering software includes such things as the:

73. ◆ software life cycle model and its resulting plans and schedules,

74. ◆ project strategies such as concurrent or distributed development activities,

75. ◆ software reuse processes,

76. ◆ development and target platforms, and

77. ◆ software development tools.

78. This environment is also the environment within which many of the software configuration control tasks are conducted. Frequently, the same tools support development, maintenance and SCM purposes.

79. I.B Constraints and Guidance for SCM

80. 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 configuration management. When software products to be developed have the potential to affect the public safety, external regulatory bodies may impose constraints. For example, see [USNRC]. Finally, the particular software life cycle model chosen for a software project and the tools selected to implement the software affect the design and implementation of the SCM process [Bersoff, (4)].

81. Guidance for designing and implementing an SCM process can also be obtained from 'best practice' as reflected in standards and process improvement or process assessment models such as the Software Engineering Institute's Capability Maturity Model [Paulk] or the ISO SPICE project [El Emam]. 'Best practice' is also reflected in the standards on software engineering issued by the various standards

organizations. Moore [31] provides a roadmap to these organizations and their standards.

82. *I.C Planning for SCM*

83. 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 (e.g., size and criticality). 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 a Software Configuration Management Plan (SCMP). The SCMP is typically subject to SQA review and audit.

84. *I.C.1 SCM Organization and Responsibilities*

85. To prevent confusion about who will perform given SCM activities or tasks, organizations 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 organizational element. The overall authority for SCM should also be identified, although this might be accomplished in the project management or quality assurance planning.

86. *I.C.2 SCM Resources and Schedules*

87. The planning for SCM identifies the staff and tools involved in carrying out SCM activities and tasks. It addresses schedule questions by establishing necessary sequences of SCM tasks and identifying their relationships to the project schedules and milestones. Any training requirements necessary for implementing the plans are also specified.

88. *I.C.3 Tool Selection and Implementation*

89. Different types of tool capabilities, and procedures for their use, support the SCM activities. Depending on the situation, these tool capabilities can be made available with some combination of manual tools, automated tools providing a single SCM capability, automated tools integrating a range of SCM (and, perhaps other) capabilities, or integrated tool environments that serve the needs of multiple participants in the software development process (e.g., SCM, development, V&V). Automated tool support becomes increasingly important, and increasingly difficult to establish, as projects grow in size and as project environments get more complex. These tool capabilities provide support for:

90. ◆ the SCM Library,

91. ◆ the software change request and approval procedures,

92. ◆ code and change management tasks,

93. ◆ reporting software configuration status and collecting SCM metrics,

94. ◆ software auditing,

95. ◆ performing software builds, and

96. ◆ managing and tracking software releases and their distribution.

97. The use of tools in these areas increases the potential for obtaining product and process measurements to be used for project management and process improvement purposes. Royce [37] describes seven core metrics of value in managing software processes. Information available from the various SCM tools relates to Royce’s Work and Progress management indicator and to his quality indicators of Change Traffic and Stability, Breakage and Modularity, Rework and Adaptability, and MTBF(mean time between failures) and Maturity. Reporting on these indicators can be organized in various ways, such as by software configuration item or by type of change requested. Details on specific goals and metrics for software processes are described in [Grady].

98. Figure 2 shows a representative mapping of tool capabilities and procedures to the SCM Activities.

99. In this example, code management systems support the operation of software libraries by controlling access to library elements, coordinating the activities of multiple users, and helping to enforce operating procedures. Other tools support the process of building software and release documentation from the software elements contained in the libraries. Tools for managing software change requests support the change control procedures applied to controlled software items. Other tools can provide database management and reporting capabilities for management, development, and quality assurance activities. As mentioned above, the capabilities of several tool types might be

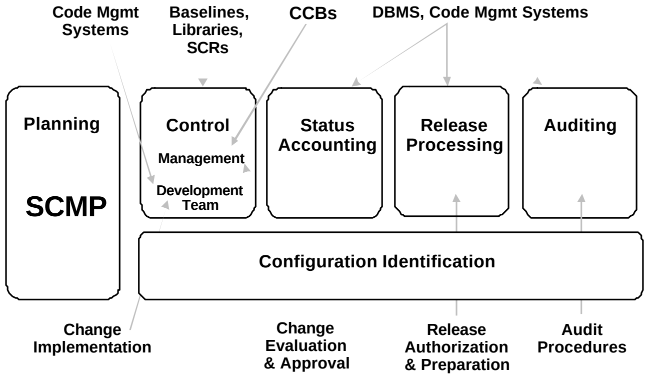


Figure 2. Characterization of SCM Tools and Related Procedures

integrated into SCM systems, which, in turn, are closely coupled to software development and maintenance activities.

100. The planning activity assesses the SCM tool needs for a given project within the context of the software engineering environment to be used and selects the tools to be used for SCM. The planning considers issues that might arise in the implementation of these tools, particularly if some form of culture change is necessary. An overview of SCM systems and selection considerations is given in [Dart, (7)], a recent case study on selecting an SCM system is given in [Midha], and [Hoek] provides a current web-based resource listing web links to various SCM tools.

101. *I.C.4 Vendor/Subcontractor Control*

102. A software project might acquire or make use of purchased software products, such as compilers. The planning for SCM considers if and how these items will be taken under configuration control (e.g., integrated into the project libraries) and how changes or updates will be evaluated and managed.

103. Similar considerations apply to subcontracted software. In this case, the SCM requirements to be imposed on the subcontractor's SCM process as part of the subcontract and the means for monitoring compliance also need to be established. The latter includes consideration of what SCM information must be available for effective compliance monitoring.

104. *I.C.5 Interface Control*

105. When a software item will interface with another software or hardware item, a change to either item can affect the other. The 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 and 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. A discussion of the performance of interface control activities is given in [Berlack].

106. *I.D Software Configuration Management Plan*

107. The results of SCM planning for a given project are recorded in a Software Configuration Management Plan (SCMP). The SCMP is a 'living document' that serves as a reference for the SCM process. It is maintained (i.e., updated and approved) as necessary during the software life cycle. In implementing the plans contained in the SCMP, it may be necessary to develop a number of more detailed, subordinate procedures that define how specific requirements will be carried out during day-to-day activities.

108. Guidance for 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 [IEEE 828]. 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:

109. 1. Introduction (purpose, scope, terms used)

110. 2. SCM Management (organization, responsibilities, authorities, applicable policies, directives, and procedures)

111. 3. CM Activities (configuration identification, configuration control, etc.)

112. 4. CM Schedules (coordination with other project activities)

113. 5. CM Resources (tools, physical, and human resources)

114. 6. CMP Maintenance

115. *I.E Surveillance of Software Configuration Management*

116. After the SCM process has been implemented, some degree of surveillance may be conducted 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. This could involve an SCM authority ensuring that the defined SCM tasks are performed correctly by those with the assigned responsibility. The software quality assurance

authority, as part of a compliance auditing activity, might also perform this surveillance.

117. The use of integrated SCM tools that have capabilities for process control can make the surveillance task easier. Some tools facilitate process compliance while providing flexibility for the developer to adapt procedures. Other tools enforce process, leaving the developer less flexibility.

118. I.E.1 SCM Metrics and Measurement

119. SCM metrics 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. Quantitative measurements against SCM process metrics 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.

120. The 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 processing time required for various types of changes would be useful in an evaluation of the criteria for determining what levels of authority are optimal for certain types of changes.

121. 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.

122. I.E.2 In-process Audits of SCM

123. Audits can be carried out during the development 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 auditing function.

124. II. Software Configuration Identification

125. The software configuration identification activity 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.

126. II.A Identifying Items to be Controlled

127. A first step in controlling change is to identify 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 the baselines to be used, along with the procedure for a baseline's acquisition of the items.

128. II.A.1 Software Configuration

129. A software configuration is the set of functional and physical characteristics of software as set forth in the technical documentation or achieved in a product [IEEE 1042]. It can be viewed as a part of an overall system configuration.

130. II.A.2 Software Configuration Item

131. A software configuration item (SCI) is an aggregation of software that is designated for configuration management and is treated as a single entity in the SCM process [IEEE 1042]. A variety of items, in addition to the code itself, are typically controlled by SCM. Software items with potential to become SCIs include plans, specifications, testing materials, software tools, source and executable code, code libraries, data and data dictionaries, and documentation for installation, maintenance, operations and software use.

132. Selecting SCIs is an important process that must achieve a balance between providing adequate visibility for project control purposes and providing a manageable number of controlled items. A list of criteria for SCI selection is given in [Berlack].

133. II.A.3 Software Configuration Item Relationships

134. The 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. The design of the identification scheme for these items should consider the need to map the identified items to the software structure as well as the need to support the evolution of the software items and their relationships.

135. *II.A.4 Software Versions*

136. Software items evolve as a software project proceeds. A *version* of a software item is a particular identified and specified item. It can be thought of as a state of an evolving item [Conradi]. A *revision* is a new version of an item that is intended to replace the old version of the item. A *variant* is a new version of an item that will be added to the configuration without replacing the old version. The management of software versions in various software engineering environments is a current research topic; see [Conradi], [Estublier], and [Sommerville, (39)].

137. *II.A.5 Baseline*

138. A software baseline is a set of software items formally designated and fixed at a specific time during the software life cycle. The term is also used to refer to a particular version of a software item that has been agreed upon. 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.

139. Commonly used baselines are the 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. The product baseline corresponds to the completed software product delivered for system integration. The baselines to be used for a given project, along with their associated levels of authority needed for change approval, are typically identified in the SCMP.

140. *II.A.6 Acquiring Software Configuration Items*

141. Software configuration items are placed under SCM control at different times; i.e. 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 3 characterizes the growth of baselined items as the life cycle proceeds. This figure is based on a 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 III.A.

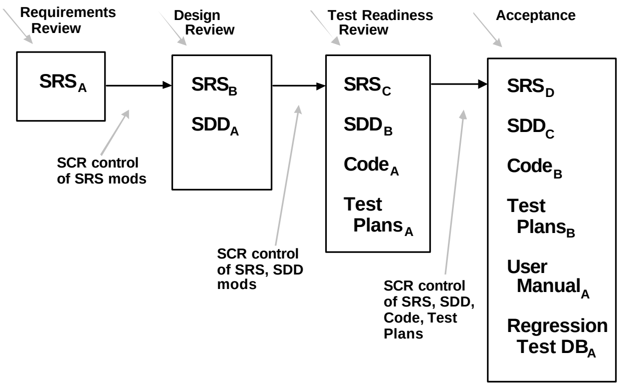


Figure 3. Acquisition of Items

142. Following the acquisition of an SCI, changes to the item must be formally approved as appropriate for the SCI and the baseline involved. Following the approval, the item is incorporated into the software baseline according to the appropriate procedure.

143. *II.B Software Library*

144. A software library is a controlled collection of software and related documentation designed to aid in software development, use, and maintenance [IEEE 610]. It is also instrumental in software release and delivery activities. Several types of libraries might be used, each corresponding to a particular level of maturity of the software item. For example a working library could support coding, whereas 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. A model of a software library is described in [Berlack].

145. 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.

146. These libraries are also an important source of information for measurements of work and progress.

147. III. Software Configuration Control

148. 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 and waivers from project requirements. Information derived from these activities is useful in measuring change traffic, breakage, and aspects of rework.

149. III.A. Requesting, Evaluating and Approving Software Changes

150. The first step in managing changes to controlled items is determining what changes to make. A software change request (SCR) process (see Figure 4) provides formal procedures for submitting and recording change requests, evaluating the potential cost and impact of a proposed change, and accepting, modifying or rejecting the proposed change. Requests for changes to software configuration items may be originated by anyone at any point in the software life cycle. One source of change requests is the initiation of corrective action in response to problem reports. Regardless of the source, the type of change (e.g. defect or enhancement) is usually recorded on the 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 modifications that would be necessary should the change request be accepted. A good understanding of the relationships among software 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 or reject the proposed change.

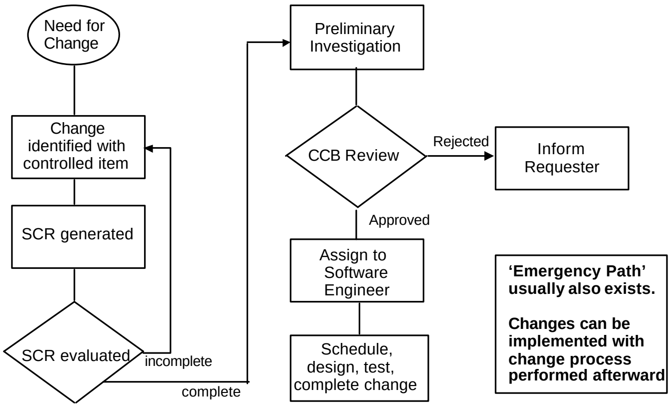

Need for Change
Change identified with controlled item
SCR generated
SCR evaluated
incomplete
complete
Preliminary Investigation
CCB Review
Rejected
Inform Requester
Approved
Assign to Software Engineer
Schedule, design, test, complete change
'Emergency Path' usually also exists.
Changes can be implemented with change process performed afterward

Figure 4. Flow of a Change Control Process

151. III.A.1. Software Configuration Control Board

152. 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 actually may reside with the responsible leader or an assigned individual rather than a multi-person 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 (e.g., impact on budget and schedule), or the 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 SQA audit or review.

153. III.A.2 Software Change Request Process

154. The software change request process requires the use of supporting tools and procedures ranging from paper forms and a documented procedure to an electronic tool 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. Change process descriptions and supporting forms (information) are given in a variety of references, e.g. [Berlack] and [IEEE 1042]. Typically, change management tools are tailored to local processes and tool suites and are often locally developed. The current trend is towards integration of these kinds of tools within a suite referred to as a software engineering environment.

155. III.B. Implementing Software Changes

156. Approved change requests are implemented according to the defined software procedures. Since a number of approved change requests might be implemented simultaneously, it is necessary to provide a means for tracking which change requests are incorporated into particular software versions and baselines. As part of the closure of the change process, completed

changes may undergo configuration audits and SQA verification. This includes ensuring that only approved changes were made. The change request process described above will typically document the SCM and other approval information for the change.

157. The actual implementation of a change is supported by the library tool capabilities that provide version management and code repository support. At a minimum, these tools provide source file check-in/out and associated version control. More powerful tools can support parallel development and geographically distributed environments. These tools may be manifested as separate specialized applications under control of an independent SCM group. They may also appear as an integrated part of the software development environment. Finally, they may be as elementary as the rudimentary change control systems provided with many operating systems, such as UNIX.

158. III.C. Deviations and Waivers

159. The constraints imposed on a software development 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 an authorization to depart from a provision prior to the development of the item. A waiver is an authorization to use an item, following its development, that departs from the provision in some way. In these cases, a formal process is used for gaining approval for deviations to, or waivers of, the provisions.

160. IV. Software Configuration Status Accounting

161. Software configuration status accounting (SCSA) is the recording and reporting of information needed for effective management of the software configuration. The design of the SCSA capability can be viewed from an information systems perspective, utilizing accepted information systems design techniques.

162. IV.A. Software Configuration Status Information

163. 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. A partial list of important data elements is given in [Berlack].

164. Some form of automated tool support is necessary to accomplish the SCSA data collection and reporting tasks. This could be a database capability, such as a relational or object-oriented database management system. This could be a stand-alone tool or a capability of a larger, integrated tool environment.

165. IV.B. Software Configuration Status Reporting

166. Reported information can be used by various organizational and project elements, including the development team, the maintenance team, project management, and quality assurance activities. Reporting can take the form of ad hoc queries to answer specific questions or the periodic production of pre-designed reports. Some information produced by the status accounting activity during the course of the life cycle might become quality assurance records.

167. In addition to reporting the current status of the configuration, the information obtained by SCSA can serve as a basis for various measurements of interest to management, development, and SCM. Examples include the number of change requests per SCI and the average time needed to implement a change request.

168. V. Software Configuration Auditing

169. A software audit is an activity performed to independently evaluate the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures [IEEE 1028]. 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. Guidance for conducting software audits is available in various references, such as [Berlack], [Buckley], and [IEEE 1028].

170. The software configuration auditing activity 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 (e.g., 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. Buckley [5] contrasts the purposes of the FCA and PCA in hardware versus software contexts and recommends careful evaluation of the need for the software FCA and PCA before performing them.

171. V.A. Software Functional Configuration Audit

172. 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 is a key input to this audit.

173. V.B. Software Physical Configuration Audit

174. The purpose of the software PCA is to ensure that the design and reference documentation is consistent with the as-built software product.

175. V.C. In-process Audits of a Software Baseline

176. As mentioned above, audits can be carried out during the development processto 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 was consistent with specification or to ensure that evolving documentation was staying consistent with the developing baseline item.

177. VI. Software Release Management and Delivery

178. The term "release" is used in this context to refer 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.

179. VI.A. Software Building

180. Software building is the activity of combining the correct versions of software 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 is delivered to the system building activity. Build instructions ensure that the proper build steps are taken and 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, or additional release purposes.

181. Software is built using particular versions of supporting tools, such as compilers. It might be necessary to rebuild an exact copy of a previously built software item. In this case, the supporting tools need to be under SCM control to ensure availability of the correct versions of the tools.

182. 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 large projects with parallel development or distributed development environments, this tool capability is necessary. Most software development environments provide this capability and it is usually referred to as the "make" facility (as in UNIX). These tools vary in complexity from requiring the engineer to learn a specialized scripting language to graphics-oriented approaches that hide much of the complexity of an "intelligent" build facility.

183. The build process and products are often subject to SQA verification.

184. VI.B Software Release Management

185. Software release management encompasses the identification, packaging and delivery of the elements of a product, for example, the executable, documentation, release notes, and configuration data. Given that product changes can be occurring on a continuing basis, one issue 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 [Sommerville, (38)]. The packaging task must identify which product items are to be delivered and select the correct variants of those

items, given the intended application of the product. The set of information documenting the physical contents of a release is known as a version description document and may exist in hardcopy or electronic form. The release notes typically describe new capabilities, known problems, and platform requirements necessary for proper product operation. The package to be released also contains loading or upgrading instructions. The latter can be complicated by the fact that some current users might have versions that are several releases old. Finally, in some cases, the release management activity might be required to track the distribution of the product to various customers. An example would be a case where the supplier was required to notify a customer of newly reported problems.

186. 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 var ious target platforms and on various customer environments.

187. Rationale for the Breakdown

188. One of the primary goals of the Guide to the SWEBOK is to arrive at a breakdown that is 'generally accepted'. Consequently, the breakdown of SCM topics was developed largely by attempting to synthesize the topics covered in the literature and in recognized standards, which tend to reflect consensus opinion. The topic on Software Release Management and Delivery is an exception since it has not commonly been broken out separately in the past. The precedent for this was set by the ISO/IEC 12207 standard [23], which identifies a 'Release Management and Delivery' activity.

189. There is widespread agreement in the literature on the SCM activity areas and their key concepts. However, there continues to be active research on implementation aspects of SCM. Examples are found in ICSE workshops on SCM such as [Estublier] and [Sommerville, (39)].

190. The hierarchy of topics chosen for the breakdown presented in this paper is expected to evolve as the Guide to the SWEBOK review processes proceed. A detailed discussion of the rationale for the proposed breakdown, keyed to the Guide to the SWEBOK development criteria, is given in Appendix B.

191. RECOMMENDED REFERENCES FOR SCM

192. Cross Reference Matrix

193. Table 1, in Appendix A, provides a cross reference between the recommended references and the topics of the breakdown. Note that, where a recommended reference is also shown in the Further Reading section, the cross reference reflects the full text rather than just the specific passage referenced in the Recommended References.

194. Recommended References

195. Specific recommendations are made here to provide additional information on the topics of the SCM breakdown.

196. W.A. Babich, *Software Configuration Management*, Coordination for Team Productivity [1] Pages 20-43 address the basics of code management.

197. H.R. Berlack, *Software Configuration Management* [2] See pages 101-175 on configuration identification, configuration control and configuration status accounting, and pages 202-206 on libraries.

198. F.J. Buckley, *Implementing Configuration Management: Hardware, Software, and Firmware* [5] See pages 10-19 on organizational context, pages 21-38 on CM planning, and 228-250 on CM auditing.

199. R. Conradi and B. Westfechtel, "*Version Models for Software Configuration Management*" [6] An in-depth article on version models used in software configuration management. It defines fundamental concepts and provides a detailed view of versioning paradigms. The versioning characteristics of various SCM systems are discussed.

200. S.A. Dart, *Spectrum of Functionality in Configuration Management Systems* [7] This report covers features of various CM systems and the scope of issues concerning users of CM systems. As of this writing, the report can be found on the Internet at: http://www.sei.cmu.edu/about/website/search.html

201. *Hoek, "Configuration Management Yellow Pages," [13]* This web page provides a current compilation of SCM resources.

http://www.cs.colorado.edu/users/andre/configuration_management.html

202. IEEE/EIA Std 12207.0-1996*, Software Life Cycle Processes, [20]* and IEEE/EIA Std 12207.1-1996*, Software Life Cycle Processes - Life Cycle Data, [21]* These standards provide the ISO/IEC view of software processes along with specific information on life cycle data keyed to software engineering standards of other standards bodies.

203. IEEE Std.828-1990, *IEEE Standard for Software Configuration Management Plans* [17] and IEEE Std.1042-1987, *IEEE Guide to Software Configuration Management* [19] These standards focus on SCM activities by specifying requirements and guidance for preparing the SCMP. These standards reflect commonly accepted practice for software configuration management.

204. A.K. Midha, "*Software Configuration Management for the 21st Century*" [30] This article discusses the characteristics of SCM systems, assessment of SCM needs in a particular environment, and the issue of selecting and implementing an SCM system. It is a current case study on this issue.

205. J.W. Moore, *Software Engineering Standards, A User's Road Map* [31] Pages 118-119 cover SCM and pages 194-223 cover the perspective of the 12207 standards.

206. M.C. Paulk, et al., *Key Practices of the Capability Maturity Model* [32] Pages 180-191 cover the SCM key process area of the SEI CMM.

207. R.S. Pressman, *Software Engineering: A Practitioner's Approach* [36] Pages 209-226 address SCM in the context of a textbook on software engineering.

208. Walker Royce, *Software Project Management, A United Framework [37]* Pages 188-202 and 283-298 cover metrics of interest to software project management that are closely related to SCM.

209. I. Sommerville, *Software Engineering* [38] Pages 675-696 cover SCM with an emphasis on software building and release management.

210. Further Reading

211. The following set of references was chosen to provide coverage of all aspects of SCM, from various perspectives and to varying levels of detail. The author and title are cited; the complete reference is given in the References section. Some items overlap with those in the Recommended References since they cover the full texts rather than specific passages.

212. W.A. Babich, *Software Configuration Management*, Coordination for Team Productivity [1] This text is focused on code management issues from the perspective of the development team.

213. H.R. Berlack, *Software Configuration Management* [2] This textbook provides detailed, comprehensive coverage of the concepts of software configuration management. This is one of the more recent texts with this focus.

214. F.J. Buckley, *Implementing Configuration Management: Hardware, Software, and Firmware* [5] This text presents an integrated view of configuration management for projects in which software, hardware and firmware are involved. It is a recent text that provides a view of software configuration management from a systems perspective.

215. J. Estublier, *Software Configuration Management*, ICSE SCM-4 and SCM-5 Workshops Selected Papers [10] These workshop proceedings are representative of current experience and research on SCM. This reference is included with the intention of directing the reader to the whole class of conference and workshop proceedings.

216. *The suite of IEEE/EIA and ISO/IEC 12207 standards*, [20]-[24] These standards cover software life cycle processes and address SCM in that context. These standards reflect commonly accepted practices for software life cycle processes. Note - the developing ISO/IEC TR 15504 (SPICE99) expands on SCM within the context of the ISO/IEC 12207 standard.

217. IEEE Std.1042-1987, *IEEE Guide to Software Configuration Management* [19] This standard provides guidance, keyed to IEEE 828, for preparing the SCMP.

218. J.W. Moore, *Software Engineering Standards, A User's Road Map* [31] This text provides a comprehensive view of current standards and standards activities in the area of software engineering.

219. M.C. Paulk, et al., *Key Practices of the Capability Maturity Model* [32] This report describes the key practices that could be evaluated in assessing software process maturity. Therefore, the section on SCM key practices provides a view of SCM from a software process assessment perspective.

220. R.S. Pressman, *Software Engineering: A Practitioner's Approach* [36] This reference and the Sommerville reference address SCM in the context of a textbook on software engineering.

221. I. Sommerville, *Software Engineering* [38] This reference and the Pressman reference address SCM in the context of a textbook on software engineering.

222. J.P. Vincent, et al., *Software Quality Assurance* [41] In this text, SCM is described from the perspective of a complete set of assurance processes for a software development project.

223. D. Whitgift, *Methods and Tools for Software Configuration Management* [43] This text covers the concepts and principles of SCM. It provides detailed information on the practical questions of implementing and using tools. This text is out of print but still available in libraries.

224. REFERENCES

225. These references were used in preparing this paper; the recommended references for SCM are listed in Section 3.1.

226. [1] W.A. Babich, Software Configuration Management: Coordination for Team Productivity, Addison-Wesley, Reading, Massachusetts, 1986.

227. [2] H.R. Berlack, Software Configuration Management, John Wiley & Sons, New York, 1992.

228. [3] E.H. Bersoff, "Elements of Software Configuration Management", Software Engineering, M. Dorfman and R.H. Thayer ed., IEEE Computer Society Press, Los Alamitos, CA, 1997.

229. [4] E.H. Bersoff and A.M. Davis, "Impacts of Life Cycle Models on Software Configuration Management", Communications of the ACM, Vol. 34, no 8, August 1991, pp104-118.

230. [5] F.J. Buckley, Implementing Configuration Management: Hardware, Software, and Firmware, Second Edition, IEEE Computer Society Press, Los Alamitos, CA, 1996.

231. [6] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," ACM Computing Surveys, Vol. 30, no 2, June 1998, pp. 232-282.

232. [7] S.A. Dart, Spectrum of Functionality in Configuration Management Systems, Technical Report CMU/SEI-90-TR-11, Software Engineering Institute, Carnegie Mellon University, 1990.

233. [8] S.A. Dart, "Concepts in Configuration Management Systems," Proceedings of the Third International Workshop on Software Configuration Management, ACM Press, New York, 1991, pp1-18.

234. [9] Khaled El Emam, et al., SPICE, The Theory and Practice of Software Process Improvement and Capability Determination, IEEE Computer Society, Los Alamitos, CA, 1998.

235. [10] J. Estublier, Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops Selected Papers, Springer-Verlag, Berlin, 1995.

236. [11] P.H. Feiler, Configuration Management Models in Commercial Environments, Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon University, 1991.

237. [12] R.B. Grady, Practical Software Metrics for Project Management and Process Improvement, Prentice-Hall, Englewook Cliffs, NJ, 1992.

238. [13] A. Hoek, "Configuration Management Yellow Pages," http://www.cs.colorado.edu/users/andre/configuration_management.html

239. [14] W.S. Humphrey, Managing the Software Process, Addison-Wesley, Reading, MA, 1989.

240. [15] IEEE Std.610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE, Piscataway, NJ, 1990.

241. [16] IEEE Std.730-1998, IEEE Standard for Software Quality Assurance Plans, IEEE, Piscataway, NJ, 1998.

242. [17] IEEE Std.828-1998, IEEE Standard for Software Configuration Management Plans, IEEE, Piscataway, NJ, 1998.

243. [18] IEEE Std.1028-1997, IEEE Standard for Software Reviews, IEEE, Piscataway, NJ, 1997.

244. [19] IEEE Std.1042-1987, IEEE Guide to Software Configuration Management, IEEE, Piscataway, NJ, 1987.

245. [20] IEEE/EIA Std 12207.0-1996, Software Life Cycle Processes, IEEE, Piscataway, NJ, 1996.

246. [21] IEEE/EIA Std 12207.1-1996, Guide for Software Life Cycle Processes – Life Cycle Data, IEEE, Piscataway, NJ, 1996.

247. [22] IEEE/EIA Std 12207.2-1996, Guide for Software Life Cycle Processes – Implementation Considerations, IEEE, Piscataway, NJ, 1996.

248. [23] ISO/IEC 12207:1995(E), Information Technology - Software Life Cycle Processes, ISO/IEC, Geneve, Switzerland, 1995.

249. [24] ISO/IEC TR 15846:1998, Information Technology - Software Life Cycle Processes - Configuration Management, ISO/IEC, Geneve, Switzerland, 1998.

250. [25] ISO/DIS 9004-7 (now ISO 10007), Quality Management and Quality System Elements, Guidelines for Configuration Management, International Organization for Standardization, Geneve, Switzerland, 1993.

251. [26] P. Jalote, An Integrated Approach to Software Engineering, Springer-Verlag, New York, 1997

252. [27] John J. Marciniak and Donald J. Reifer, Software Acquisition Management, Managing the Acquisition of Custom Software Systems, John Wiley & Sons, 1990.

253. [28] J.J. Marciniak, "Reviews and Audits," Software Engineering, M. Dorfman and R.H. Thayer ed., IEEE Computer Society Press, Los Alamitos, CA, 1997.

254. [29] K. Meiser, "Software Configuration Management Terminology," Crosstalk, 1995, http://www.stsc.hill.af.mil/crosstalk/1995/jan/terms.html, February 1999.

255. [30] A.K. Midha, "Software Configuration Management for the 21st Century," Bell Labs Technical Journal, Winter 1997.

256. [31] J.W. Moore, Software Engineering Standards, A User's Roadmap, IEEE Computer Society, Los Alamitos, CA, 1998.

257. [32] M.C. Paulk, et al., Key Practices of the Capability Maturity Model, Version 1.1, Technical Report CMU/SEI-93-TR-025, Software Engineering Institute, Carnegie Mellon University, 1993

258. [33] M.C. Paulk, et al., The Capability Maturity Model, Guidelines for Improving the Software Process, Addison-Wesley, Reading, Massachusetts, 1995.

259. [34] S.L. Pfleeger, Software Engineering: Theory and Practice, Prentice Hall, Upper Saddle River, NJ, 1998

260. [35] R.K. Port, "Software Configuration Management Technology Report, September 1994", http://www.stsc.hill.af.mil/cm/REPORT.html, February 1999.

261. [36] R.S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, New York, 1997.

262. [37] Walker Royce, Software Project Management, A United Framework, Addison-Wesley, Reading, Massachusetts, 1998.

263. 38] I. Sommerville, Software Engineering, Fifth Edition, Addison-Wesley, Reading, Massachusetts, 1995.

264. [39] I. Sommerville, Software Configuration Management, ICSE SCM-6 Workshop, Selected Papers, Springer-Verlag, Berlin, 1996.

265. [40] USNRC Regulatory Guide 1.169, Configuration Management Plans for Digital Computer Software Used in Safety Systems of Nuclear Power Plants, U.S. Nuclear Regulatory Commission, Washington DC, 1997.

266. [41] J.P. Vincent, et al., Software Quality Assurance, Prentice-Hall, Englewood Cliffs, NJ, 1988.

267. [42] W.G. Vincenti, What Engineers Know and How They Know It, The Johns Hopkins University Press, Baltimore, MD, 1990.

268. [43] D. Whitgift, Methods and Tools for Software Configuration Management, John Wiley & Sons, Chichester, England, 1991.

269. APPENDIX A. CROSS REFERENCE TABLE

270. Table 1. Coverage of the Breakdown Topics by the Recommended References

	Babich	Berlack	Buckley	Conradi	Dart	Hoek	IEEE 828	IEEE/EIA 12207	Midha	Moore	Paulk	Pressman	Royce	Sommerville
I. Management														
I.A. Org. Context		X	X		X		X							X
I.B Constraints		X					X			X		X		X
I.C Planning		X	X		X		X							X
I.C.1 Org. & Resp.		X	X				X	X						
I.C.2 Resources & Sched.		X	X				X	X						X
I.C.3 Tool Selection		X	X	X	X	X	X		X			X		X
I.C.4 Vendor Control		X	X				X							X
I.C.5 Interface Control		X	X				X							
I.D SCM Plan		X	X				X	X			X			X
I.E Surveillance			X				X				X			
I.E.1 Metrics/Meas.											X		X	
I.E.2 In-Process Audit			X				X				X			
II. SW Config Identification														
II.A Identifying Items		X	X				X	X			X			
II.A.1 SW Configuration		X	X				X					X		
II.A.2 SW Config. Item		X	X	X			X				X	X		X
II.A.3 SCI Relationships		X		X			X					X		
II.A.4 Software Versions	X			X								X		
II.A.5 Baselines	X	X	X				X	X				X		
II.A.6 Acquiring SCIs			X				X							
II.B Software Library	X	X	X	X			X		X		X	X		
(SCM Library Tool)	X			X	X	X			X			X		X

271. Table 1. Coverage of the Breakdown Topics by the Recommended References (cont.)

	Babich	Berlack	Buckley	Conradi	Dart	Hoek	IEEE 828	IEEE/EIA 12207	Midha	Moore	Paulk	Pressman	Royce	Sommerville
III. SW Configuration Control					X		X	X	X		X	X		X
III.A Requesting Changes		X	X				X	X	X		X	X		X
III.A.1 SW CCB		X	X				X				X	X		X
(Change Mgmt Tool)					X				X					X
III.A.2 SW Change Process		X	X				X					X		X
III.B Implementing Changes			X				X	X	X		X	X		X
(Change Cntl Tool)	X			X	X	X			X					X
III.C Deviations & Waivers		X	X											
IV. SW Config Status Acctg														
IV.A. Status Information		X	X				X	X			X			
(CSA Tool)														
IV.B Status Reporting		X	X				X	X			X	X		
V. SW Configuration Audit		X	X		X		X	X			X	X		
V.A Functional Config Audit		X	X				X	X						
V.B Physical Config Audit		X	X				X	X						
V.C In-Process Audit			X				X				X			
VI. SW Release Mgmt & Del								X						X
VI.A SW System Building									X					X
(SW Build Tools)	X			X		X								X
VI.B SW Release Mgmt											X			X
(SW Release Tool)						X								

272. APPENDIX B. RATIONALE DETAILS

273. Criteria are defined in Appendix A of the entire Guide.

274. *Criterion (a): Number of topic breakdowns*

275. One breakdown is provided.

276. *Criterion (b): Reasonableness*

277. The breakdowns are reasonable in that they cover the areas typically discussed in texts and standards, although there is somewhat less discussion of release management as a separate topic. In response to comments on version 0.5 of the paper, the tool discussion under 'Planning for SCM' has been expanded. The various tool subheadings used throughout the text have been removed (so they do not appear as topics), however, the supporting text has been retained and incorporated into the next higher level topics.

278. *Criterion (c): Generally Accepted*

279. The breakdowns are generally accepted in that they cover the areas typically discussed in texts and standards.

280. At level 1, the breakdown is identical to that given in IEC 12207 (Section 6.2) except that the term "Management of the Software Configuration Management Process" was used instead of "Process Implementation" and the term "Software Configuration Auditing" was used instead of "Configuration Evaluation." The typical texts discuss Software Configuration Management Planning (our topic A.3); We have expanded this to a "management of the process" concept in order to capture related ideas expressed in many of the references that we have used. These ideas are captured in topics A.1 (organizational context), A.2 (constraints and guidance), and A.4 (surveillance of the SCM process). A similar comparison can also be made to [Buckley] except for the addition of "Software Release Management and Delivery."

281. We have chosen to include the word "Software" as a prefix to most of the configuration topics to distinguish the topics from hardware CM or system level CM activities. We would reserve "Configuration Management" for system purposes and then use HCM and SCM for hardware and software respectively.

282. The topic A.1, "Software Configuration Management Organizational Context," covers key topics addressed in multiple texts and articles and it appears within the level 1 headings consistently with the placement used in the references. This new term on organizational context was included as a placeholder for capturing three concepts found in the references. First, [Buckley] discusses SCM in the overall context of a project with hardware, software, and firmware elements. We believe that this is a link to a related discipline of system engineering. (This is similar to what IEEE 828 discusses under the heading of "Interface Control"). Second, SCM is one of the product assurance processes supporting a project, or in IEC 12207 terminology, one of the supporting lifecycle processes. The processes are closely related and, therefore, interfaces to them should be considered in planning for SCM. Finally, some of the tools for implementing SCM might be the same tools used by the developers. Therefore, in planning SCM, there should be awareness that the implementation of SCM is strongly affected by the environment chosen for the development activities.

283. The inclusion of the topic "Release Management and Delivery" is somewhat controversial since the majority of texts on software configuration management devote little or no attention to the topic. We believe that most writers assume the library function of configuration identification would support release management and delivery but, perhaps, assume that these activities are the responsibility of project or line management. The IEC 12207 standard, however, has established this as a required area for SCM. Since this has occurred and since this topic should be recognized somewhere in the overall description of software activities, "Release Management and Delivery" has been included.

284. *Criterion (d): No Specific Application Domains*

285. No specific application domains have been assumed.

286. *Criterion (e): Compatible with Various Schools of Thought*

287. SCM concepts are fairly stable and mature.

288. *Criterion (f): Compatible with Industry, Literature, and Standards*

289. The breakdown was derived from the literature and from key standards reflecting consensus

opinion. The extent to which industry implements the SCM concepts in the literature and in standards varies by company and project.

290. *Criterion (g): As Inclusive as Possible*

291. The inclusion of the level 1 topic on management of SCM expands the planning concept into a larger area that can cover all management-related topics, such as surveillance of the SCM process. For each level 1 topic, the level 2 topics categorize the main areas in various references' discussions of the level 1 topic. These are intended to be general enough to allow an open-ended set of subordinate level 3 topics on specific issues. The level 3 topics cover specifics found in the literature but are not intended to provide an exhaustive breakdown of the level 2 topic.

292. *Criterion (h): Themes of Quality, Tools, Measurement, and Standards*

293. The relationship of SCM to product assurance is provided for in the breakdowns. The description will also convey the role of SCM in achieving a consistent, verified, and validated product.

294. A number of level 3 topics were included throughout the breakdown in order to call attention to the types of tool capabilities that are needed for efficient work within the areas described by particular level 1 and level 2 topics. These are intended to address capabilities, not specific tools; i.e. one tool may perform several of the capabilities described. These topics may not be significant enough to stand alone; if not, we would combine the discussion and place it in the management section or include the discussion in the higher level topic. One or more references on the subject of tool selection will be listed.

295. A similar approach was taken toward the use of measures.

296. Standards are explicitly included in the breakdowns.

297. *Criterion (i): 2 to 3 levels, 5 to 9 topics at the first level*

298. The proposed breakdown satisfies this criterion.

299. *Criterion (j): Topic Names Meaningful Outside the Guide*

300. For the most part, we believe this is the case. Some terms, such a "Baselines" or "Physical Configuration Audit" require some explanation but they are obviously the terms to use since appear throughout the literature.

301. *Criterion (l): Topics only sufficiently described to allow reader to select appropriate material*

302. We believe this has been accomplished. We have not attempted to provide a tutorial on SCM.

303. *Criterion (m): Text on the Rationale Underlying the Proposed Breakdowns*

304. This document provides the rationale.

CHAPTER 8
SOFTWARE ENGINEERING MANAGEMENT

Stephen G. MacDonell and Andrew R. Gray
University of Otago, Dunedin, New Zealand
+64 3 479 8135 (phone) +64 3 479 8311 (fax)
stevemac@infoscience.otago.ac.nz

TABLE OF CONTENTS

1. 1. INTRODUCTION

2. This is the current draft (version 0.7) of the Knowledge Area description for Software Engineering Management. The primary goals of this draft are to:

3. 1. define the Software Engineering Management Knowledge Area,

4. 2. present two alternative breakdowns of the knowledge area in hierarchical topic frameworks,

5. 3. provide the topic-reference matrix,

6. 4. list the three classes of references (recommended, further readings, and those used in preparing this document).

7. A draft glossary (without definitions) is included. We have found considerable differences in definitions amongst the reviewers and feel that such a glossary, either for this document or all Knowledge Area documents, is essential.

8. 2. DEFINITION OF KNOWLEDGE AREA

9. The *Software Engineering Management* Knowledge Area addresses the management of software development projects and the measurement and modeling of such projects. While measurement is an important aspect of all Guide to the SWEBOK Knowledge Areas, it is here that the topic is most focused, particularly with regard to issues involved in model development and testing.

10. There is considerable overlap with other Knowledge Areas, and reading the following Knowledge Area documents along side this one may be useful. Material is not duplicated here that is covered in these separate documents. Of course all Knowledge Area documents share some commonalties with this one, these are simply those with more obvious and extensive overlap.

11. *Software Quality*, as quality is constantly a goal of management and involves many activities that must be managed.

12. *Software Testing*, where this is a managed phase in the development process and with regard to quality.

13. *Software Engineering Process*, where these activities must be managed.

14. As alluded to above, the *Software Engineering Management* knowledge area consists of both the measurement/metrics and management process sub-areas. Whilst these two topics are often regarded (and generally taught) as being separate, and indeed they do possess many mutually unique aspects, their close relationship has led to their combined treatment here as part of the Guide to the SWEBOK. In essence, management without measurement-qualitative and quantitative -suggests a lack of rigor, and measurement without management suggests a lack of purpose or context. In the same way,

however, management and measurement without expert knowledge is equally ineffectual so we must be careful to avoid overemphasizing the quantitative aspects of *Software Engineering Management*. Effective management requires a combination of both numbers and stories.

15. The following working definitions are used in this document.

16. *Measurement/metrics* refers to the assignment of values and labels to aspects of software development (products, processes, and resources as defined by [Fenton and Pfleeger, 1997]) and the models that may be derived therefrom whether these models are developed using statistical, expert knowledge, or other techniques.

17. *Management process* refers to the activities that are undertaken in order to ensure that the software development process is performed in a manner consistent with the organization's policies, goals, and requirements.

18. The management process sub-area makes (in theory at least) extensive use of the measurement/metrics sub-area-ideally this exchange between the two sub-areas occurs continuously throughout the software development processes.

19. 3. BREAKDOWN OF TOPICS

20. It is immediately apparent that there are several different ways of looking at the breakdown of topics in this Knowledge Area, and between ourselves and reviewer comments we have selected just two: a life-cycle approach and a topic-based approach. Each is discussed in this section in turn and the following section discusses the justification of each. In both cases the management and measurement sub-topics are separated which will no doubt please many of the reviewers whilst not troubling those happy with the combination of these in the one Knowledge Area.

21. In many ways these two breakdowns complement each other, providing different perspectives on the same ideas which may be beneficial to students and practitioners alike. The latter topic-based breakdown may be especially useful for those who disagree with the topics included and wish to produce more focused courses, for example, simply covering software project management in a minimalist fashion without dealing with measurement and metric issues or more general management topics. It may also prove to be more suitable for smaller organizations who wish to concentrate on particular aspects of the breakdown as opposed to the approach in its entirety.

22. 3.1 Life-cycle breakdown

23. 1. Measurement

24. 1. Determining the goals of a measurement program

25. 1. Organizational objectives (broad issues)

26. 2. Software process improvement goals (specific issues)

27. 3. Determining specific measurement goals

28. 2. Measuring software and its development

29. 1. Size measurement (for example, lines of code)

30. 2. Complexity measurement

31. 3. Performance measurement

32. 4. Resource measurement

33. 3. Selection of measurements

34. 1. The Goal/Question/Metric approach (as an example)

35. 2. Other metric frameworks (such as Practical Software Measurement (PSM))

36. 3. Measurement validity (scales)

37. 4. Collection of data (ongoing)

38. 1. Survey techniques and questionnaire design

39. 2. Automated and manual data collection

40. 5. Software metric models

41. 1. Model building, calibration and evaluation

42. 2. Implementation, interpretation and refinement of models

43. 3. Existing models (examples as case studies)

44. 2. Organizational management and coordination

45. 1. Portfolio management

46. 1. Strategy development and coordination

47. 2. General investment management techniques

48. 3. Project selection

49. 4. Portfolio construction (risk minimization and value maximization)

50. 2. Acquisition decisions and management
51. 1. Vendor management
52. 2. Subcontract management
53. 3. Policy management
54. 1. Standards
55. 2. Means of policy development
56. 3. Policy dissemination and enforcement
57. 4. Personnel management (ongoing)
58. 1. Hiring and firing
59. 2. Training and motivation
60. 3. Directing personnel career development
61. 4. Team structures
62. 5. Communication (ongoing)
63. 1. Meeting procedures
64. 2. Written presentations
65. 3. Oral presentations
66. 4. Negotiation
67. 3. Initiation and scope definition
68. 1. Collection and negotiation of requirements
69. 1. Requirements analysis management
70. 2. Use cases (as an example)
71. 2. Proposal construction
72. 3. Feasibility analysis (ongoing)
73. 1. Technical feasibility
74. 2. Financial feasibility
75. 3. Social/political feasibility
76. 4. Process for the revision of requirements
77. 5. Iterative development (ongoing)
78. 1. Low fidelity prototyping (as an example)
79. 2. Prototype evolution
80. 4. Planning
81. 1. Risk management (ongoing)
82. 1. Risk analysis
83. 2. Critical risk assessment
84. 3. Techniques for modeling risk
85. 4. Contingency planning
86. 5. Project abandonment policies
87. 2. Process planning
88. 1. Life-cycle approach
89. 2. Methodologies
90. 3. Standards
91. 4. Planning techniques
92. 1. GANTT
93. 2. PERT
94. 3. Tools for supporting planning
95. 3. Determine deliverables
96. 4. Quality management (ongoing)
97. 1. Defining quality
98. 2. Quality control and assurance
99. 5. Schedule and cost estimation
100. 1. Effort estimation
101. 2. Task dependencies
102. 3. Duration estimation
103. 6. Resource allocation
104. 1. Equipment and facilities
105. 2. People
106. 7. Task and responsibility allocation
107. 8. Implementing a metrics process
108. 5. Enactment
109. 1. Implementation of plan
110. 2. Monitor process
111. 1. Reporting
112. 2. Variance analysis
113. 3. Control process
114. 1. Change control
115. 2. Configuration management
116. 3. Scenario analysis
117. 4. Feedback
118. 1. Reporting
119. 2. Problem detection
120. 3. Crisis identification
121. 6. Review and evaluation
122. 1. Determining satisfaction of requirements
123. 1. User review
124. 2. Verification
125. 3. Validation
126. 2. Reviewing and evaluating performance
127. 1. Personnel performance
128. 2. Tool and technique evaluation
129. 3. Process assessment
130. 7. Project close out (closure)

131. 1. Determining closure

132. 2. Archival activities

133. 1. Measurement database

134. 2. Organizational learning-lessons learned

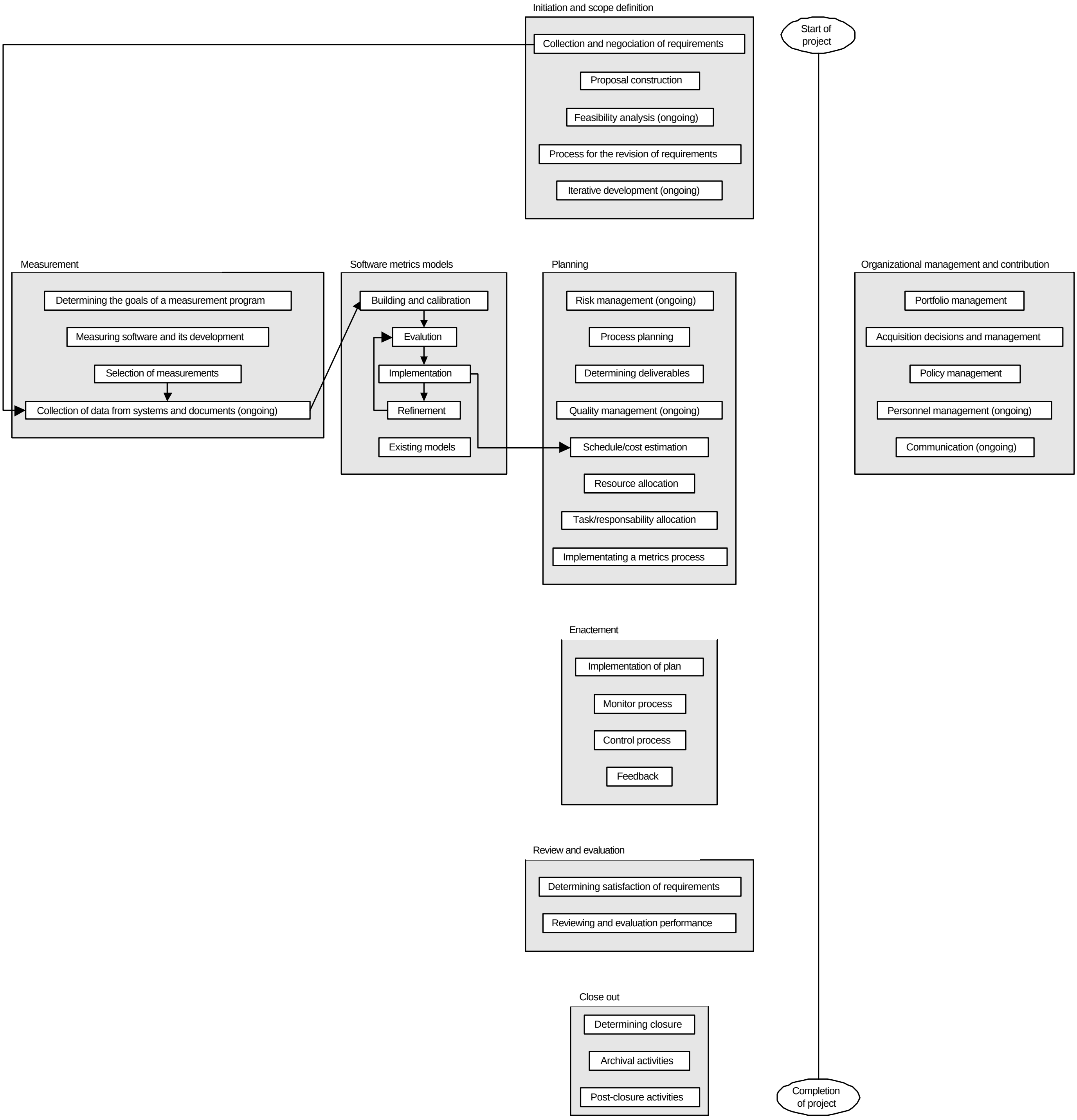
135. 3. Duration of retention

136. 8. Post-closure activities

137. 1. Maintenance

138. 2. System retirement

139. The topics are not listed strictly in temporal order since there are in fact three somewhat distinct processes being performed here, namely measurement/metrics, coordination, and the management process. Figure 1 shows this more clearly. We have decided to treat the first process as the actual activity of developing and releasing models, and the second and third as the usage of those pre-existing models in coordination and management activities. This is discussed in more detail later in the document.


Initiation and scope definition
Collection and negociation of requirements
Proposal construction
Feasibility analysis (ongoing)
Process for the revision of requirements
Iterative development (ongoing)
Start of project
Measurement
Determining the goals of a measurement program
Measuring software and its development
Selection of measurements
Collection of data from systems and documents (ongoing)
Software metrics models
Building and calibration
Evalution
Implementation
Refinement
Existing models
Planning
Risk management (ongoing)
Process planning
Determining deliverables
Quality management (ongoing)
Schedule/cost estimation
Resource allocation
Task/responsability allocation
Implementating a metrics process
Organizational management and contribution
Portfolio management
Acquisition decisions and management
Policy management
Personnel management (ongoing)
Communication (ongoing)
Enactement
Implementation of plan
Monitor process
Control process
Feedback
Review and evaluation
Determining satisfaction of requirements
Reviewing and evaluation performance
Close out
Determining closure
Archival activities
Post-closure activities
Completion of project

140. Figure 1: Software engineering management flowchart

141. 3.2 Topic-based breakdown

142. This is a more recently created outline, containing the same topics as the life-cycle breakdown, but organized according to what we see as common themes. This remains quite similar to the life-cycle breakdown since obviously life-cycle stages have some inherent cohesion.

143. 1. Mathematical, statistical, and model building topics

144. 1. Measuring software and its development

145. 1. Size measurement (for example, lines of code)

146. 2. Complexity measurement

147. 3. Performance measurement

148. 4. Resource measurement

149. 2. Selection of measurements

150. 1. The Goal/Question/Metric approach (as an example)

151. 2. Other metric frameworks (such as Practical Software Measurement (PSM))

152. 3. Measurement validity (scales)

153. 3. Collection of data (ongoing)

154. 1. Survey techniques and questionnaire design

155. 2. Automated and manual data collection

156. 4. Software metric models

157. 1. Model building, calibration and evaluation

158. 2. Implementation, interpretation and refinement of models

159. 3. Existing models (examples as case studies)

160. 5. Schedule and cost estimation

161. 1. Effort estimation

162. 2. Task dependencies

163. 3. Duration estimation

164. 6. Implementing a metrics process

165. 2. Software engineering management topics

166. 1. Determining the goals of a software measurement program

167. 1. Organizational objectives (broad issues)

168. 2. Software process improvement goals (specific issues)

169. 3. Determining specific measurement goals

170. 2. Collection and negotiation of requirements

171. 1. Requirements analysis management

172. 2. Use cases (as an example)

173. 3. Proposal construction

174. 4. Feasibility analysis (ongoing)

175. 1. Technical feasibility

176. 2. Financial feasibility

177. 3. Social/political feasibility

178. 5. Process for the revision of requirements

179. 6. Iterative development (ongoing)

180. 1. Low fidelity prototyping (as an example)

181. 2. Prototype evolution

182. 7. Process planning

183. 1. Life-cycle approach

184. 2. Methodologies

185. 3. Standards

186. 8. Determine deliverables

187. 9. Control process

188. 1. Change control

189. 2. Configuration management

190. 3. Scenario analysis

191. 10. Determining satisfaction of requirements

192. 1. User review

193. 2. Verification

194. 3. Validation

195. 11. Post-closure activities

196. 1. Maintenance

197. 2. System retirement

198. 3. Management topics

199. 1. Portfolio management

200. 1. Strategy development and coordination

201. 2. General investment management techniques

202. 3. Project selection

203. 4. Portfolio construction (risk minimization and value maximization)

204. 2. Acquisition decisions and management

205. 1. Vendor management

206. 2. Subcontract management

207. 3. Policy management

208. 1. Standards

209. 2. Means of policy development

210. 3. Policy dissemination and enforcement

211. 4. Personnel management (ongoing)

212. 1. Hiring and firing

213. 2. Training and motivation

214. 3. Directing personnel career development

215. 4. Team structures

216. 5. Communication (ongoing)

217. 1. Meeting procedures

218. 2. Written presentations

219. 3. Oral presentations

220. 4. Negotiation

221. 6. Risk management (ongoing)

222. 1. Risk analysis

223. 2. Critical risk assessment

224. 3. Techniques for modeling risk

225. 4. Contingency planning

226. 5. Project abandonment policies

227. 7. Planning techniques

228. 1. GANTT

229. 2. PERT

230. 3. Tools for supporting planning

231. 8. Quality management (ongoing)

232. 1. Defining quality

233. 2. Quality control and assurance

234. 9. Resource allocation

235. 1. Equipment and facilities

236. 2. People

237. 10. Task and responsibility allocation

238. 11. Implementation of plan

239. 12. Monitor process

240. 1. Reporting

241. 2. Variance analysis

242. 13. Feedback

243. 1. Reporting

244. 2. Problem detection

245. 3. Crisis identification

246. 14. Reviewing and evaluating performance

247. 1. Personnel performance

248. 2. Tool and technique evaluation

249. 3. Process assessment

250. 15. Determining closure

251. 16. Archival activities

252. 1. Measurement database

253. 2. Organizational learning-lessons learned

254. 3. Duration of retention

255. 4. BREAKDOWN RATIONALE

256. 4.1 Life-cycle breakdown

257. It is important to note that we have not based this breakdown (or the topic-based breakdown) on existing breakdowns *per se*. While these have provided inspiration, we have aimed for consistency and completeness rather then picking our favorite hierarchy of topics.

258. This outline is, as we have said, very much a "life-cycle" based breakdown. Topics tend to appear in the same order as their associated activities are enacted in a software development project-with the obvious exceptions of the Organizational management and coordination topics and the measurement/metrics sub-area which encompass the entire process. Many of these stages are also iterative, especially planning and development when prototyping. Any ongoing activities, such as risk management and quality management, are indicated as such (although this obviously depends on the specific development and management processes used).

259. In several places quite specific techniques are listed, such as Function Point Analysis and the Goal/Question/Metric approach. This generally indicates that the technique is suggested as being a good tutorial/case-study example of the overall concept, rather than a crucial topic to be mastered. Other specific techniques could be used to replace these if desired.

260. Within the measurement/metrics sub-area five main subtopics are addressed: measurement program goals, fundamental measurement, measurement selection, data collection and model development and use. The first four subtopics are primarily concerned with the actual theory and purpose behind measurement and address issues such as measurement scales and measure selection (such as by GQM). The collection of measures is included as an issue to be addressed here. This involves both technical issues (automated extraction) and human issues (questionnaire design, responses to measurements being taken). The fifth subtopic

(software metric models) is concerned with the task of building models using both data and knowledge. Such models need to be evaluated (for example, by testing their performance on holdout samples) to ensure that their levels of accuracy are sufficient and that their limitations are known. The refinement of models, which could take place during or after projects are completed is another activity here. The implementation of metric models is more management-oriented since the use of such models has an influential effect on the *subject's* (for want of a better word) behavior. (Note: We have continued to use the common terminology (in software engineering circles) of *software metrics* here, rather than limiting ourselves to measurement. We recognize that this could lead to some confusion with engineers familiar with the empirical model-building process from another discipline, necessitating careful wording. The alternative of using more standard terminology however, whilst well intentioned, would make less obvious the connection between this work and many excellent papers and books (including Fenton and Pfleeger's seminal work [Fenton and Pfleeger, 1997]). On the other hand Zuse's excellent book [Zuse, 1997] does include "measurement" in the title rather than "metrics". Here it seems that the best solution is to use both sets of expressions in a somewhat interchangeable manner so that practitioners are familiar with both.)

261. In the management process sub-area the notion of management "in the large" is considered in the coordination topic, addressing issues including portfolio development and management, project selection and system acquisition, the development and implementation of policies, personnel management, and communication. The remaining topics then correspond (roughly) to stages in the project development life cycle. First is the initiation and scope-definition topic, which covers the management of the requirements, gathering process and the specification of procedures and methods for their revision. Feasibility analysis is included as part of this topic even though this is an ongoing activity. Here the focus is on high-level feasibility, as in "is it possible". Feasibility may well be determined by reference to some formal model. Planning is the next set of activities for a software-engineering manager. Management of risk is included here, as is planning for the process(es) used. Ongoing quality management is begun at this point. The tasks of schedule and cost estimation also fall within this topic. Given schedule estimates it is possible to perform resource then task allocation. Responsibilities need to be allocated and quality control procedures implemented. The outcome of this stage would be a series of plans. These plans are then put into action in the enactment topic. The project must then be monitored for deviations and corrective actions may be taken. Change control and configuration management are important activities at this stage in the process. The timeliness and format of reports is also important if feedback is to be successful. The review topic involves determining that the requirements have indeed been satisfied by the system. Performance assessment, of individuals, tools, techniques and processes is necessary for performance improvement and as part of the organization's learning process. Finally, the project needs to be closed and all useful information securely recorded. These archival activities are often neglected in both practice and education so we would like to emphasize their necessity for supporting a measurement program.

262. The above breakdown of topics is based on a division into measurement/metrics and management processes. The former refers to the actual creation of models, which can then be used as part of the latter. These activities may be performed by the same person, but they could then be seen to be "wearing different hats."

263. 4.2 Topic-based breakdown

264. This contains the same topics as the life-cycle breakdown, but organizes them according to three broad topic areas: mathematical, statistical, and model building topics; software engineering management topics; and management topics. This breakdown may be more useful for partial or more specific courses, etc.

265. The same justifications for the topics are used for the life-cycle approach also apply here.

266. 5. Matrix of topics vs. Reference material

267. The level of granularity used in Table 1 is a mixture of second and third level topics, depending on the specificity of the topic in question. The topics are in the order given in the life-cycle breakdown.

	Topic	Reference (sections and pages)
268.	Determining the goals of a measurement program	3.2, 83-95; 13.1-13.6, 464-483; 14.1-14.4, 487-514 [Fenton and Pfleeger, 1997]
269.	Size measurement	7.1-7.4, 244-267 [Fenton and Pfleeger, 1997]
270.	Complexity measurement	7.5, 267-275 [Fenton and Pfleeger, 1997] 8.2.2.1- 8.2.2.3, 293-296 [Fenton and Pfleeger, 1997]
271.	Performance measurement	7.5, 267-275 [Fenton and Pfleeger, 1997]
272.	Resource measurement	3.1.3, 82-83 [Fenton and Pfleeger, 1997] 15.3, 529- 531 [Fenton and Pfleeger, 1997]
273.	Goal/Question/Metric	S3.2, 83-95 [Fenton and Pfleeger, 1997]
274.	Measurement validity (scales)	2.7-2.8, 42-55 [Zuse, 1997]
275.	Survey techniques and questionnaire design	4.1, 118-125 [Fenton and Pfleeger, 1997]
276.	Data collection	1.3.3, 16-17 [Fenton and Pfleeger, 1997] 5.3-5.5, 169-180 [Fenton and Pfleeger, 1997] 30.5.1, 626-627 [Sommerville, 1996]
277.	Model building and calibration	6.2-6.3, 190-215 [Fenton and Pfleeger, 1997]3.3, 98-113 [Pfleeger, 1998]
278.	Model evaluation	3.3, 98-113 [Pfleeger, 1998]
279.	Implementation of models	4.6, 95-97 [Pressman, 1997]
280.	Interpretation of models	6.2-6.3, 190-215 [Fenton and Pfleeger, 1997]3.3, 98-113 [Pfleeger, 1998]
281.	Function Point Analysis	4.3.2-4.3.3, 85-90; 4.4, 90-92; 5.7.1, 120-121 [Pressman, 1997]
282.	COCOMO	5.7.1-5.7.2, 120-124 [Pressman, 1997]
283.	Portfolio management	**Still seeking an appropriate reference**
284.	Vendor management	1.4, 14-15 [Pfleeger, 1998]
285.	Subcontract management	1.4, 14-15 [Pfleeger, 1998]
286.	Policy management	2.3-2.4, 58-69 [Pfleeger, 1998]
287.	Personnel management	[Weihrich] [Thayer] [Zwacki] 3.2, 59-66 [Pressman, 1997] 3.2, 89-98 [Pfleeger, 1998]
288.	Communication	[Weihrich] [Thayer]
289.	Requirements analysis	[Faulk] 3.2, 59-66 [Pressman, 1997]
290.	Use cases	20.4.1, 592-594 [Pressman, 1997]
291.	Proposal construction	3.1-3.2, 47-51 [Sommerville, 1996]
292.	Feasibility analysis	4.1, 67-68 [Sommerville, 1996] 10.6, 250-259 [Pressman, 1997]
293.	Portfolio management	**Still seeking an appropriate reference**
294.	Revision of requirements	4.2-4.4, 68-75 [Sommerville, 1996]
295.	Prototyping	8.1-8.3, 140-153 [Sommerville, 1996]
296.	Risk management	6.1-6.8, 133-150 [Pressman, 1997] [Thayer and Fairley] 3.4, 113-117 [Pfleeger, 1998]
297.	Process planning	2.2-2.11, 26-49; 7.3-7.8, 160-175 [Pressman, 1997]
298.	Determining deliverables	3.3, 51-52 [Sommerville, 1996]3.1, 76-88 [Pfleeger, 1998]
299.	Quality management	8.1-8.10, 180-203 [Pressman, 1997]30.1-30.6, 615-634 [Sommerville, 1996] [Dunn]
300.	Schedule and cost estimation	12.3-12.4, 435-448 [Fenton and Pfleeger, 1997] [Brooks] [Heemstra]
301.	Resource allocation	5.4, 108-111 [Pressman, 1997] 3.4, 52-57 [Sommerville, 1996]
302.	Task and responsibility allocation	[Weihrich] [Thayer]
303.	Implementing a metrics program	4.6, 95-97 [Pressman, 1997]14.1-14.4, 487-514 [Fenton and Pfleeger, 1997]
304.	Revision of requirements	4.2-4.4, 68-75 [Sommerville, 1996]
305.	Implementing plans	7.8, 174-175 [Pressman, 1997] 3.5, 118-119 [Pfleeger, 1998] 3.2, 48-51 [Sommerville, 1996]
306.	Process monitoring	31.2-31.3, 641-647 [Sommerville, 1996]

	Topic	Reference (sections and pages)
307.	Change control	9.5, 220-223 [Pressman, 1997]
308.	Configuration management	9.19.4, 210-220 [Pressman, 1997]
309.	Scenario analysis	**Still seeking an appropriate reference**
310.	Feedback	[Weihrich] [Thayer]
311.	Determining satisfaction of requirements	4.9, 174-178 [Pfleeger, 1998]
312.	Reviewing and evaluating performance	8.5, 190-194 [Pressman, 1997] [Marciniak]
313.	Determining closure	4.9, 174-178 [Pfleeger, 1998]
314.	Archival activities	**Still seeking an appropriate reference**
315.	Implementing plans	7.8, 174-175 [Pressman, 1997] 3.5, 118-119 [Pfleeger, 1998] 3.2, 48-51 [Sommerville, 1996]
316.	Maintenance	32,1-32.5, 662-672 [Sommerville, 1996] [Bennett]
317.	System retirement	2.3.8, 36 [Sommerville, 1996]

318. Table 1: Topics and their references

319. [Thayer and Thayer] is an excellent glossary of project management terminology and can be added to this list as a general reference.

321. 6. RECOMMENDED REFERENCES

322. The Topic-Reference matrix as shown in Section 5 requires the following references to be included in the Guide to the SWEBOK.

323. [Fenton and Pfleeger, 1997] 16-17, 82-95, 118-125, 169-180, 190-215, 244-267, 293-296, 435-448, 464-483, 487-514, 529-531 **Total: 155 pages**

324. [Dorfman and Thayer, 1997]13-22, 82-103, 256-265, 289-303, 374-386 **Total: 70 pages**

325. [Pfleeger, 1998]14-15, 58-69, 76-119, 174-178 **Total: 63 pages**

326. [Pressman, 1997] 26-49, 59-66, 85-92, 95-97, 108-111, 120-124, 133-150, 160-175, 210-223, 250-259, 592-594 **Total: 113 pages**

327. [Reifer, 1997] 292-293 **Total: 2 pages**

328. [Sommerville, 1996] 36, 47-57, 67-75, 140-153, 615-634, 641-647, 662-672 **Total: 73 pages**

329. [Thayer, 1997] 4-13, 72-104, 195-202, 433-440, 506-529 **Total: 83 pages**

330. [Zuse, 1997] 42-55 **Total: 14 pages**

331. This totals **573 pages** (assuming that part pages count as wholes) with three topics yet to be referenced. There does not appear to be any easy way to reduce this much further without overly reducing the topics or their coverage.

332. 7. LIST OF FURTHER READINGS

333. The following texts (which include all of the required references) are suggested as useful sources of information about this Knowledge Area.

334. [Dorfman and Thayer, 1997] **531 pages**

335. [Fenton and Pfleeger, 1997] **638 pages**

336. [Karolak] **171 pages**

337. [McConell, 1996] **647 pages**

338. [McConell, 1997] **250 pages**

339. [Moore, 1998] **296 pages**

340. [Pfleeger, 1998] **576 pages**

341. [Pressman, 1997] **852 pages**

342. [Reifer, 1997] **652 pages**

343. [Sommerville, 1996] **742 pages**

344. [Thayer, 1997] **531 pages**

345. [Zuse, 1997] **755 pages**

346. These total **6641 pages** (before subtracting the above-cited pages and without accounting for the duplicated papers in the three collections). With these adjustments the total page count should be around 5000 pages.

347. 8. REFERENCES USED TO WRITE AND JUSTIFY THE DESCRIPTION

348. [Duncan, 1996]

349. [Vincenti, 1990]

350. 9. GLOSSARY

351. The terms below have not been defined in this version, they are provided to indicate some terms that we have found to require entries.

352. Control:
353. Coordination:
354. Life-cycle:
355. Measurement:
356. Metric:
357. Model:
358. Monitoring:
359. Plan:
360. Planning:
361. Policy:
362. Portfolio:
363. Portfolio management:
364. Process:
365. Quality:
366. Quality assurance:
367. Quality control:
368. Requirements:
369. Resource:
370. Risk assessment:
371. Risk management:
372. Stakeholder:
373. Standard:
374. Users:

375. 10. REFERENCES

376. [Bennett] **Keith H. Bennett**. Software maintenance: a tutorial. Pages 289-303. In [Dorfman and Thayer, 1997].

377. [Brooks] **Frederick P. Brooks**. No silver bullet: essence and accidents of software engineering. Pages 13-22. In [Dorfman and Thayer, 1997].

378. [Dorfman and Thayer, 1997] **Merlin Dorfman and Richard H. Thayer**. 1997. *Software engineering*. Ed. Merlin Dorfman and Richard H. Thayer. IEEE Computer Society.

379. [Duncan, 1996] **W.R. Duncan**. 1996. *A guide to the project management body of knowledge*.

380. [Dunn] **Robert H. Dunn**. Software Quality Assurance: A Management Perspective . Pages 433-440. In [Thayer, 1997].

381. [Faulk] **Stuart R. Faulk**. Software Requirements: A Tutorial. Pages 82-103. In [[Dorfman and Thayer, 1997].

382. [Fenton and Pfleeger, 1997] **Norman E. Fenton and Shari Lawrence Pfleeger**. 1997. *Software metrics: a rigorous practical approach.* PWS Publishing Company.

383. [Heemstra] **F.J. Heemstra**. Software cost estimation. Pages 374-386. In [Dorfman and Thayer, 1997].

384. [Karolak] **Dale Walter Karolak**. 1996. *Software engineering risk management.* IEEE Computer Society.

385. [Marciniak] **John J. Marciniak**. Reviews and audits. Pages 256-265. In [Dorfman and Thayer, 1997].

386. [McConell, 1996] **Steve C McConell**. 1996. *Rapid Development: Taming Wild Software Schedules.* Microsoft Press.

387. [McConell, 1997] **Steve C McConell**. 1997. *Software Project Survival Guide.* Microsoft Press.

388. [Moore, 1998] **James W. Moore**. 1998. *Software engineering standards: a user's road map.* IEEE Computer Society.

389. [Pfleeger, 1998] **Shari Lawrence Pfleeger**. 1998. *Software engineering: theory and practice.* Prentice Hall.

390. [Pressman, 1997] **Roger S. Pressman**. 1997. *Software engineering: a practitioner's approach.* McGraw-Hill.

391. [Reifer, 1997] **Donald J. Reifer**. 1997. *Software management*, 5th edition. IEEE Computer Society.

392. [Sommerville, 1996] **Ian Sommerville**. 1996. *Software engineering.* Addison-Wesley.

393. [Thayer] **Richard H. Thayer**. Software Engineering Project Management . Pages 72-104. In [Thayer, 1997].

394. [Thayer and Fairley] **Richard H. Thayer and Richard E. Fairley**. Software Risk Management. Pages 195-202. In [Thayer, 1997].

395. [Thayer and Thayer] **Richard H. Thayer and Mildred C. Thayer**. Software Engineering Project Management Glossary . Pages 506-529. In [Thayer, 1997].

396. [Thayer, 1997] **Richard H. Thayer**. 1997. *Software engineering project management.* Ed. Richard H. Thayer. IEEE Computer Society.

397. [Vincenti, 1990] **W.G. Vincenti**. 1990. *What engineers know and how they know it-analytical studies from aeronautical history*. John Hopkins.

398. [Weihrich] **Heinz Weihrich**. Management: Science, Theory, and Practice . Pages 4-13. In [Thayer, 1997].

399. [Zuse, 1997] **Horst Zuse**. 1997. *A framework of software measurement.* Walter de Gruyter.

400. [Zwacki] **Robert A. Zawacki**. How to pick eagles. Pages 292-293. In [Reifer, 1997].

CHAPTER 9
SOFTWARE ENGINEERING PROCESS

Khaled El Emam
NRC, Canada

TABLE OF CONTENTS

1. 1. INTRODUCTION

2. The software engineering process area has witnessed dramatic growth over the last decade. This was partly fueled by a recognition by major acquirers of systems where software is a major component that process issues can have an important impact on the ability of their suppliers to deliver. Therefore, they encouraged a focus on the software process as a way to remedy this. Furthermore, the academic community has pursued an active research agenda in developing new tools and techniques to support software processes, and also empirically studying software processes and their improvement. It should also be recognized that other disciplines have been studying software processes for many years, namely, the Management Information Systems community, albeit they used a different terminology. With the publication of a few success stories, industrial adoption of software process technology has also been growing. Therefore, there is in fact an extensive body of knowledge on the software engineering process.

3. This document presents a description of the knowledge area of software engineering process for the Guide to the Software Engineering Body of Knowledge (SWEBOK) project. The intention is to provide a coherent framework where the different types of knowledge can be organized, and key references identified. A breakdown of topics is presented for the knowledge area along with a succinct description of each topic. References are given to materials that provide more in-depth coverage of the important areas of software process. Where available, web addresses where cited material can be downloaded have been added.

4. 1.1 Acronyms

5.	CBA IPI	CMM Based Appraisal for Internal Process Improvement
6.	CMM	Capability Maturity Model
7.	EF	Experience Factory
8.	G/Q/Q	Goal/Question/Metric
9.	HRM	Human Resources Management
10.	IDEAL	Initiating-Diagnosing-Establishing-Acting-Leveraging (model)
11.	MIS	Management Information Systems
12.	PDCA	Plan-Do-Check-Act (cycle)
13.	QIP	Quality Improvement Paradigm
14.	ROI	Return on Investment
15.	SCE	Software Capability Evaluation
16.	SEPG	Software Engineering Process Group
17.	SW-CMM	Capability Maturity Model for Software

18. 2. DEFINITION

19. The software engineering process Knowledge Area (KA) can be examined at two levels. The first level encompasses the technical and managerial activities that are performed during software development, maintenance, acquisition, and retirement. The second is the meta-level, which is concerned with the definition, implementation, measurement, management, change and improvement of the software

processes. The latter we will term *software process engineering.*

20. The first level is covered by the other KA's of the Guide to the Software Engineering Body of Knowledge. This knowledge area is concerned with the second: software process engineering.

21. It is important to orient the readers and reviewers by making the following clarification. This KA description has been developed with the following example uses in mind:

22. - If one were to write a book on the topic of software process engineering, this KA description would identify the chapters and provide the initial references for writing the chapters. The KA description is not the book.

23. - If one were to prepare a certification exam that includes software process engineering, this KA description would identify the sections of the exam and provide the initial references for writing the questions. The KA description by itself will not be the source of questions.

24. - If one were to prepare a course on software process engineering, this KA description would identify the sections of the course and the course material, and identify the initial references to use as the basis for developing the course material. The KA description is not the course material by itself.

25. 2.1 Scope

26. The scope of the KA is defined to exclude the following:

27. - Human resources management (as embodied in the People CMM 309 for example)

28. - Systems engineering processes

29. The reason for this exclusion is that, while important topics in themselves, they are outside the direct scope of software process engineering. However, where relevant, interfaces to HRM and systems engineering will be addressed.

30. 2.2 Currency of Material

31. The software process engineering discipline is rapidly changing, with new paradigms and new models. The breakdown and references included here are pertinent at the time of writing. An attempt has been made to focus on concepts to shield the knowledge area description from changes in the field, but of course this cannot be 100% successful, and therefore the material here must be evolved over time. A good example is the on-going CMM Integration effort and the Team Software Process effort 342, both of which are likely to have a considerable influence on the software process community once widely disseminated, and would therefore have to be accommodated in the knowledge area description.

32. 2.3 Structure of the KA

33. To structure this KA in a way that is directly related to practice, we have defined a generic process model for software process engineering. This model identifies the activities that are performed in a process engineering context. The topics are mapped to these activities. The advantage of such a structure is that one can see, in practice, where each of the topics is relevant, and provides an overall rationale for the topics. This generic model is based on the PDCA cycle, which should be familiar to many readers.

34. 3. BREAKDOWN OF TOPICS

35. Below is the overall breakdown of the topics in this knowledge area. Further explanations are provided in the subsequent sections.

36. Basic Concepts and Definitions
37. Themes
38. Terminology
39. Process Infrastructure
40. The Experience Factory
41. The Software Engineering Process Group
42. Process Measurement
43. Methodology in Process Measurement
44. Process Measurement Paradigms
45. Analytic Paradigm
46. Benchmarking Paradigm
47. Process Definition

48. Types of Process Definitions

49. Life Cycle Models

50. Software Life Cycle Models

51. Notations for Process Definitions

52. Process Definition Methods

53. Automation

54. Qualitative Process Analysis

55. Process Definition Review

56. Root Cause Analysis

57. Process Implementation and Change

58. Paradigms for Process Implementation and Change

59. Guidelines for Process Implementation and Change

60. Evaluating the Outcome of Process Implementation and Change

61. 3.1 Basic Concepts and Definitions

62. 3.1.1 Themes

63. Dowson 313 notes that “All process work is ultimately directed at ‘software process assessment and improvement’”. This means that the objective is to implement new or better processes in actual practices, be they individual, project or organizational practices.

64. We describe the main topics in the software process engineering (i.e., the meta-level that has been alluded to earlier) area in terms of a cycle of process change, based loosely on the commonly known PDCA (plan-do-check-act) cycle. This cycle highlights that individual process engineering topics are part of a larger process to improve practice, and that process evaluation and feedback is an important element of process engineering.

65. Software process engineering consists of four activities as illustrated in the model in Figure 1. The activities are sequenced in an iterative cycle allowing for continuous feedback and improvement of the software process.

66. The “Establish Process Infrastructure” activity consists of establishing commitment to process implementation and change, and putting in place an appropriate infrastructure (resources and responsibilities) to make it happen.

67. The activities “Analyze Process” and “Implement and Change Process” are the core ones in process engineering, in that they are essential for any long-lasting benefit from process engineering to accrue. In “Analyze Process” the objective is to understand the current business objectives and process needs of the organization[1], identify its strengths and weaknesses, and make a plan for process implementation and change. In “Implement and Change Process”, the objective is to execute the plan, deploy new processes (which may involve, for example, the deployment of tools and training of staff), and/or change existing processes.

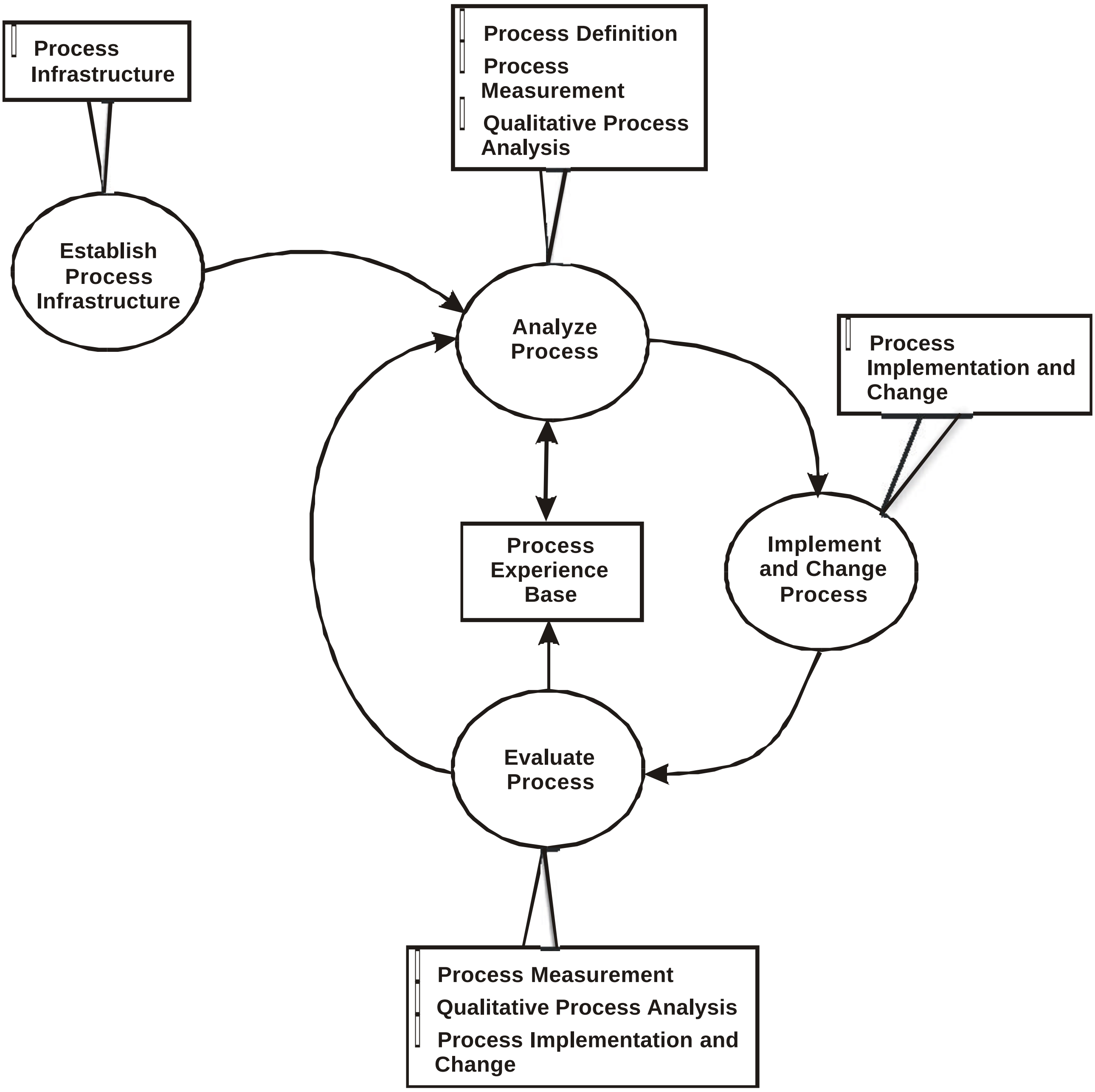
68. The fourth activity, “Evaluate Process” is concerned with finding out how well the implementation and change went; whether the expected benefits materialized. This is then used as input for subsequent cycles.

69. At the centre of the cycle is the “Process Experience Base”. This is intended to capture lessons from past iterations of the cycle (e.g., previous evaluations, process definitions, and plans). Evaluation lessons can be qualitative or quantitative. No assumptions are made about the nature or technology of this “Process Experience Base”, only that it be a persistent storage. It is expected that during subsequent iterations of the cycle, previous experiences will be adapted and reused.

70. With this cycle as a framework, it is possible to map the topics in this knowledge area to the specific activities where they would be most relevant. This mapping is also shown in Figure 1.

71. It should be noted that this cycle is not intended to imply that software process engineering is relevant to only large organizations. To the contrary, process-related activities can, and have been, performed successfully by small organizations, teams, and individuals. The way the activities defined in the cycle are performed would be different depending on the context. Where it is relevant, we will present examples of approaches for small organizations.

[1] The term “organization” is meant in a loose sense here. It could be a project, a team, or even an individual.


Process Infrastructure
Process Definition
Process Measurement
Qualitative Process Analysis
Establish Process Infrastructure
Analyze Process
Process Implementation and Change
Process Experience Base
Implement and Change Process
Evaluate Process
Process Measurement
Qualitative Process Analysis
Process Implementation and Change

72. Figure 1: A model of the software process engineering cycl e, and the relationship of its activities to the KA topics.

73. The topics in this KA are as follows:

74. *Process Infrastructure:* This is concerned with putting in place an infrastructure for software process engineering.

75. *Process Measurement:* This is concerned with quantitative techniques to diagnose software processes; to identify strengths and weaknesses. This can be performed to initiate process implementation and change, and afterwards to evaluate the consequences of process implementation and change.

76. *Process Definition:* This is concerned with defining processes in the form of models, plus the automated support that is available for the modeling task, and for enacting the models during the software process.

77. *Qualitative Process Analysis:* This is concerned with qualitative techniques to analyze software processes, to identify strengths and weaknesses. This can be performed to initiate process implementation and change, and afterwards to evaluate the consequences of process implementation and change.

78. *Process Implementation and Change:* This is concerned with deploying processes for the first time and with changing existing process. This topic focuses on organizational change. It describes the paradigms, infrastructure, and critical success factors necessary for successful process implementation and change. Within the scope of this topic, we also present some conceptual issues about the evaluation of process change.

79. The main, generally accepted, themes in the software engineering process field have been described by Dowson in 313. His themes are a subset of the topics that we cover in this KA. Below are Dowson's themes:

80. - Process definition: covered in topic 0 of this KA breakdown

81. - Process assessment: covered in topic 0 of this KA breakdown

82. - Process improvement: covered in topics 0 and 0 of this KA breakdown

83. - Process support: covered in topic 0 of this KA breakdown

84. We also add one theme in this KA description, namely the qualitative process analysis (covered in topic 0).

85. *3.1.2 Terminology*

86. There is no single universal source of terminology for the software engineering process field, but good sources that define important terms are 326363, and the vocabulary (Part 9) in the ISO/IEC 15504 documents 351.

87. 3.2 Process Infrastructure

88. At the initiation of process engineering, it is necessary to have an appropriate infrastructure in place. This includes having the resources (competent staff and funding), as well as the assignment of responsibilities. This is an indication of management commitment to the process engineering effort. Various committees may have to be established, such as a steering committee to oversee the process engineering effort.

89. It is widely recognized that a team separate from the developers/maintainers must be set up and tasked with process analysis, implementation and change 296. The main reason for this is that the priority of the developers/maintainers is to produce systems or releases, and therefore process engineering activities will not receive as much attention as they deserve or need. In a small organization, outside help (e.g., consultants) may be required to assist in making up a process team.

90. Two types of infrastructure are embodied in the concepts of the Experience Factory 289290 and the Software Engineering Process Group 329. The IDEAL handbook 366 provides a good description of infrastructure for process improvement in general.

91. *3.2.1 The Experience Factory*

92. The EF is different from the project organization which focuses on the development and maintenance of applications. Their relationship is depicted in Figure 2.

93. The concept of the EF is intended to institutionalize the collective learning of an organization by developing, updating, and delivering to the project organization *experience packages* (e.g., guide books, models, and training courses). The project organization offers to the experience factory their products, the plans used in their development, and the data gathered during development and operation. Examples of experience packages include:

94. - resource models and baselines (e.g., local cost models, resource allocation models)

95. - change and defect baselines and models (e.g., defect prediction models, types of defects expected for the application)

96. - project models and baselines (e.g., actual vs. expected product size)

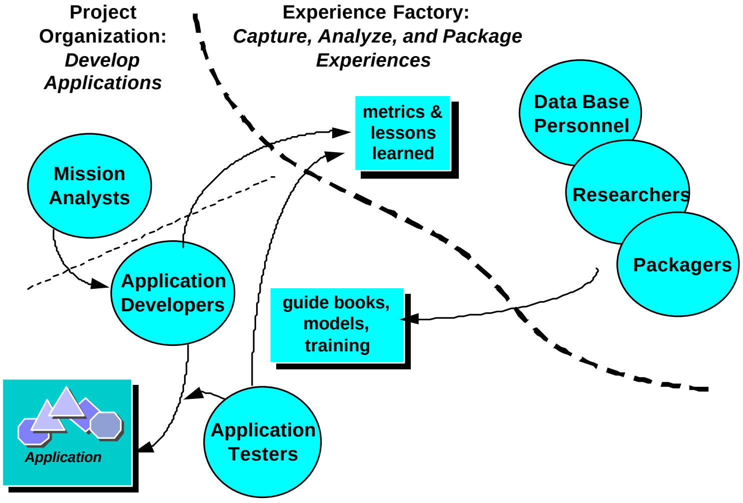
97. - process definitions and models (e.g., process models for Cleanroom, Ada waterfall model)

98. - method and technique evaluations (e.g., best method for finding interface faults)

99. - products and product parts (e.g., Ada generics for simulation of satellite orbits)

100. - quality models (e.g., reliability models, defect slippage models, ease of change models), and

101. - lessons learned (e.g., risks associated with an Ada development).


Project Organization: *Develop Applications*
Experience Factory: *Capture, Analyze, and Package Experiences*
metrics & lessons learned
Data Base Personnel
Researchers
Packagers
Mission Analysts
Application Developers
guide books, models, training
Application
Application Testers

102. Figure 2: The relationship between the Experience Factory and the project organization as implemented at the Software Engineering Laboratory at NASA/GSFC. This diagram is reused here from 291 with permission of the authors.

103. *3.2.2 The Software Engineering Process Group*

104. The SEPG is intended to be the central focus for process improvement within an organization. According to 388, the analysts within the EF are comparable to the SEPG. Therefore, the SEPG can in principle fit within the EF.

105. The SEPG typically has the following ongoing activities:

106. - Obtains and maintains the support of all levels of management

107. - Facilitates software process assessments (see below)

108. - Works with line managers whose projects are affected by changes in software engineering practice

109. - Maintains collaborative working relationships with software engineers

110. - Arranges for any training or continuing education related to process implementation and change

111. - Tracks, monitors, and reports on the status of particular improvement efforts

112. - Facilitates the creation and maintenance of process definitions

113. - Maintains a process database

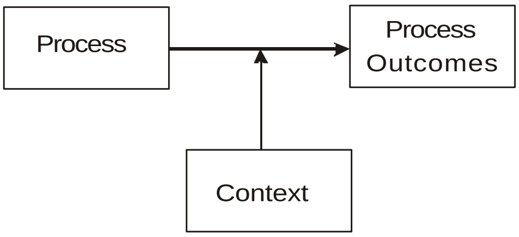
114. - Provides process consultation to development projects and management

115. Fowler and Rifkin 329 suggest the establishment of a steering committee consisting of line and supervisory management. This would allow management to guide process implementation and change, and also provides them with visibility. Furthermore, technical working groups may be established to focus on specific issues, such as selecting a new design method to setting up a measurement program.

116. 3.3 Process Measurement

117. Process measurement, as used here, means that quantitative information about the process is collected, analyzed, and interpreted. Measurement is used to identify the strengths and weaknesses of processes, and to evaluate processes after they have been implemented and/or changed (e.g., evaluate the ROI from implementing a new process).[2]

118. The assumption upon which most process engineering work is premised can be depicted by the path diagram in Figure 3. Here, we assume that the process has an impact on process outcomes. Process outcomes could be, for example, product quality (faults per KLOC), maintainability (effort to make a certain type of change), productivity (LOC per person month), time-to-market, the extent of process variation, or customer satisfaction (as measured through a customer survey). This relationship depends on the particular context (e.g., size of the organization, or size of the project).



119. Figure 3: Path diagram showing the relationship between process and outcomes (results).

120. Not every process will have a positive impact on outcomes. For example, the introduction of software inspections may reduce testing effort and cost, but may increase interval time if each inspection introduces large delays due to the scheduling of inspection meetings. Therefore, it is preferred to use multiple process outcome measures that are important for the organization's business.

121. In general, we are not really interested in the process itself, rather we are most concerned about the process outcomes. However, in order to achieve the process outcomes that we desire (e.g., better quality, better maintainability, greater customer satisfaction) we have to implement the appropriate process.

122. Of course, it is not only process that has an impact on outcomes, other factors such as the capability of the staff and the tools that are used play an important role. But here we focus only on the process as an antecedent.

123. One can measure the quality of the software process itself, or the process outcomes. The methodology in Section 3.3.1 is applicable to both. We will focus in Section 3.3.2 on process measurement since the measurement of process outcomes is more general and applicable in other knowledge areas.

124. 3.3.1 Methodology in Process Measurement

125. A guide for measurement using the G/Q/M method is provided in 391, and the "Practical Software Measurement" guidebook provides another good overview of measurement 374. A good practical text on establishing and operating a measurement program has been produced by the Software Engineering Laboratory 389. This also discusses the cost of measurement. Texts that present experiences in implementing measurement in software organizations include 356371380. An emerging international standard that defines a generic measurement process is also available (ISO/IEC CD 15939: *Information Technology – Software Measurement Process*) 352.

126. Two important issues in the measurement of software engineering processes are reliability and validity. Reliability becomes important when there is subjective measurement, for example, when assessors assign scores to a particular process. There are different types of validity that ought to be demonstrated for a software process measure, but the most critical one is predictive validity. This is concerned with the relationship between the process measure and the process outcome. A discussion of both of these and different methods for achieving them can be found in 319334. An IEEE Standard describes a methodology for validating metrics (*IEEE Standard for a Software Quality Metrics Methodology*. IEEE Std 1061-1998) 346.

127. An overview of existing evidence on reliability of software process assessments can be found in 324, and for predictive validity in 334357322.

[2] Process measurement may serve other purposes as well. For example, process measurement is useful for managing a software project. Some of these are covered in the Project Management and other KA's. Here we focus on process measurement for the purpose of process implementation and change.

128. *3.3.2 Process Measurement Paradigms*

129. Two general paradigms that are useful for characterizing the type of process measurement that can be performed have been described by Card 301. The distinction made by Card is a useful conceptual one. Although, there may be overlaps in practice.

130. The first is the analytic paradigm. This is characterized as relying on *"quantitative evidence to determine where improvements are needed and whether an improvement initiative has been successful".*[3] The second, the benchmarking paradigm, *"depends on identifying an 'excellent' organization in a field and documenting its practices and tools".* Benchmarking assumes that if a less-proficient organization adopts the practices of the excellent organization, it will also become excellent. Of course, both paradigms can be followed at the same time, since they are based on different types of information.

131. The analytic paradigm is exemplified by the Quality Improvement Paradigm (QIP) consisting of a cycle of understanding, assessing, and packaging 388. The benchmarking paradigm is exemplified by the software process assessment work (see below).

132. We use these paradigms as general titles to distinguish between different types of measurement.

133. *3.3.2.1 Analytic Paradigm*[4]

134. ◆ Experimental and Observational Studies

135. Experimentation involves setting up controlled or quasi experiments in the organization to evaluate processes 367. Usually, one would compare a new process with the current process to determine whether the former has better process outcomes. Correlational (nonexperimental) studies can also provide useful feedback for identifying process improvements (e.g., 283).

136. ◆ Process Simulation

137. The process simulation approach can be used to *predict* process outcomes if the current process is changed in a certain way 382. Initial data about the performance of the current process needs to be collected, however, as a basis for the simulation.

138. ◆ Orthogonal Defect Classification

139. Orthogonal Defect Classification is a technique that can be used to link faults found with potential causes. It relies on a mapping between fault types and fault triggers 302303. There exists an IEEE Standard on the classification of faults (or anomalies) that may also be useful in this context (*IEEE Standard for the Classification of Software Anomalies.* IEEE Std 1044-1993) 347.

140. ◆ Statistical Process Control

141. Placing the software process under statistical process control, through the use of control charts and their interpretations, is an effective way to identify stability, or otherwise, in the process 328.

142. ◆ The Personal Software Process

143. This defines a series of improvements to an individual’s development practices in a specified order 340. It is ‘bottom-up’ in the sense that it stipulates personal data collection and improvements based on the data interpretations.

144. *3.3.2.2 Benchmarking Paradigm*

145. This paradigm involves measuring the capability/maturity of an organization’s processes. A general introductory overview of the benchmarking paradigm and its application is provided in 398.

146. ◆ Process assessment models

147. *Architectures of assessment models*

148. There are two general architectures for an assessment model that make different assumptions about the order in which processes must be measured: the continuous and the staged architectures 375. At this point it is not possible to make a recommendation as to which approach is better than another. They have considerable differences. An organization should evaluate them to see which are most pertinent to their needs and objectives when selecting a model.

149. *Assessment models*

150. The most commonly used assessment model in the software community is the

[3] Although qualitative evidence also can play an important role. In such a case, see Section 0 on qualitative process analysis.

[4] These are intended as examples of the analytic paradigm, and reflect what is currently done in practice. Whether a specific organization uses all of these techniaues will depend, at least partially, on its maturity.

SW-CMM 387. It is also important to recognize that ISO/IEC 15504 is an emerging international standard on software process assessments 321351. It defines an exemplar assessment model and conformance requirements on other assessment models. ISO 9001 is also a common model that has been applied by software organizations 396. Other notable examples of assessment models are Trillium 300, Bootstrap 394, and the requirements engineering capability model 393. There are also maturity models for other software processes available, such as for testing 298299, a measurement maturity model 297, and a maintenance maturity model 314 (although, there have been many more capability/maturity models that have been defined, for example, for design, documentation, and formal methods, to name a few). A maturity model for systems engineering has also been developed, which would be useful where a project or organization is involved in the development and maintenance of systems including software 317. A voiced concern has been the applicability of assessment models to small organizations. This is addressed in 355385, where assessments models tailored to small organizations are presented.

151. ◆ Process assessment methods

152. Purpose

153. In order to perform an assessment, a specific assessment method needs to be followed. In addition to producing a quantitative score that characterizes the capability of the process (or maturity of the organization), an important purpose of an assessment is to create a climate for change within the organization 316. In fact, it has been argued that the latter is the most important purpose of doing an assessment 315.

154. Assessment methods

155. The most well known method that has a reasonable amount of publicly available documentation is the CBA IPI 316. Many other methods are refinements of this for particular contexts. Another well known method for supplier selection is the SCE 287. Requirements on methods that reflect what are believed to be good assessment practices are provided in 365351.

156. 3.4 Process Definition

157. Software engineering processes are defined for a number of reasons, including: facilitating human understanding and communication, supporting process improvement, supporting process management, providing automated process guidance, and providing automated execution support 308339327. The types of process definitions required will depend, at least partially, on the reason.

158. It should be noted also that the context of the project and organization will determine the type of process definition that is most important. Important variables to consider include the nature of the work (e.g., maintenance or development), the application domain, the structure of the delivery process (e.g., waterfall, incremental, evolutionary), and the maturity of the organization.

159. There are different approaches that can be used to define and document the process. Under this topic the approaches that have been presented in the literature are covered, although at this time there is no data on the extent to which these are used in practice.

160. 3.4.1 Types of Process Definitions

161. Processes can be defined at different levels of abstraction (e.g., generic definitions vs. tailored definitions, descriptive vs. prescriptive vs. proscriptive). The differentiation amongst these has been described in 364340376.

162. Orthogonal to the levels above, there are also types of process definitions. For example, a process definition can be a procedure, a policy, or a standard.

163. 3.4.2 Life Cycle Models

164. These models serve as a high level definition of the activities that occur during development. They are not detailed definitions, but only the high level activities and their interrelationships. The common ones are: the waterfall model, throwaway prototyping model, evolutionary prototyping model, incremental/iterative development, spiral model, reusable software model, and automated software synthesis. (see 292307354376378). Comparisons of these models are provided in 307310, and a method for selection amongst many of them in 284.

165. *3.4.3 Software Life Cycle Process Models*

166. Definitions of life cycle process models tend to be more detailed than life cycle models. Another difference being that life cycle process models do not attempt to order their processes in time. Therefore, in principle, the life cycle processes can be arranged to fit any of the life cycle models. The two main references in this area are ISO/IEC 12207: *Information Technology – Software Life Cycle Processes* 350 and ISO/IEC TR 15504: *Information Technology – Software Process Assessment* 351321. Extensive guidance material for the application of the former has been produced by the IEEE (*Guide for Information Technology - Software Life Cycle Processes - Life cycle data*, IEEE Std 12207.1-1998, and *Guide for Information Technology - Software Life Cycle Processes– Implementation. Considerations*. IEEE Std 12207.2-1998) 348349. The latter defines a two dimensional model with one dimension being processes, and the second a measurement scale to evaluate the capability of the processes. In principle, ISO/IEC 12207 would serve as the process dimension of ISO/IEC 15504.

167. The IEEE standard on developing life cycle processes also provides a list of processes and activities for development and maintenance (*IEEE Standard for Developing Software Life Cycle Processes*, IEEE Std 1074-1991) 344, and provides examples of mapping them to life cycle models. A standard that focuses on maintenance processes is also available from the IEEE (*IEEE Standard for Software Maintenance*, IEEE Std 1219-1992) 345.

168. *3.4.4 Notations for Process Definitions*

169. Different elements of a process can be defined, for example, activities, products (artifacts), and resources 339. Detailed frameworks that structure the types of information required to define processes are described in 369285.

170. There are a large number of notations that have been used to define processes. They differ in the types of information defined in the above frameworks that they capture. A text that describes different notations is 390.

171. Because there is no data on which of these was found to be most useful or easiest to use under which conditions, we cover what seemingly are popular approaches in practice: data flow diagrams 330, in terms of process purpose and outcomes 351, as a list of processes decomposed in constituent activities and tasks defined in natural language 350, Statecharts 358382 (also see 336 for a comprehensive description of Statecharts), ETVX 381, Actor-Dependency modeling 294397, SADT notation 368, Petri nets 286, IDEF0 390, rule-based 288, and System Dynamics 282. Other process programming languages have been devised, and these are described in 308327339.

172. *3.4.5 Process Definition Methods*

173. These methods specify the activities that must be performed in order to define a process model. These may include eliciting information from developers to build a descriptive process definition from scratch, and to tailoring an existing standard or commercial process. In general, there is a strong similarity amongst them in that they tend to follow a traditional software development life cycle: 369368293294359.

174. *3.4.6 Automation*

175. Automated tools either support the execution of the process definitions, or they provide guidance to humans performing the defined processes. In cases where process analysis is performed, some tools allow different types of simulations (e.g., discrete event simulation).

176. There exist tools that support each of the above process definition notations. Furthermore, these tools can execute the process definitions to provide automated support to the actual processes, or to fully automate them in some instances. An overview of process modeling tools can be found in 327, and of process-centered environments in 332333.

177. Recent work on the application of the www to the provision of real-time process guidance is described in 360.

178. 3.5 Qualitative Process Analysis

179. The objective of qualitative process analysis is to identify the strengths and weaknesses of the software process. It can be performed as a diagnoses before implementing or changing a process. It could also be performed after a process is implemented or changed to determine whether the change has had the desired effect.

180. Below we present two techniques for qualitative analysis that have been used in practice. Although it is plausible that new techniques would emerge in the future.

181. *3.5.1 Process Definition Review*

182. Qualitative evaluation means reviewing a process definition (either a descriptive or a prescriptive one, or both), and identifying deficiencies and potential process improvements. Typical examples are presented in 286358. An easily operational way to analyze a process is to compare it to an existing standard (national, international, or profesisonal body), such as ISO/IEC 12207 350.

183. With this approach, one does not collect quantitative data on the process. Or if quantitative data is collected, it plays a supportive role. The individuals performing the analysis of the process definition use their knowledge and capabilities to decide what process changes would potentially lead to desirable process outcomes.

184. *3.5.2 Root Cause Analysis*

185. Another common qualitative technique that is used in practice is a "Root Cause Analysis". This involves tracing back from detected problems (e.g., faults) to identify the process causes, with the aim of changing the process to avoid the problems in the future. Examples of this for different types of processes are described in 293320306373.

186. With this approach, one starts from the process outcomes, and traces back along the path in Figure 3 to identify the process causes of the undesirable outcomes. The Orthogonal Defect Classification technique described in Section 3.3.2.1 can be considered a more formalized approach to root cause analysis using quantitative information.

187. 3.6 Process Implementation and Change

188. This topic describes the situation when processes are deployed for the first time (e.g., introducing an inspection process within a project or a complete methodology, such as Fusion 305 or the Unified Process 353), and when current processes are changed (e.g., introducing a tool, or optimizing a procedure).[5] In both instances, existing practices have to be modified. If the modifications are deep, then changes in the organizational culture may be necessary.

189. *3.6.1 Paradigms for Process Implementation and Change*

190. Two general paradigms that have emerged for driving process implementation and change are the Quality Improvement Paradigm 388 and the IDEAL model 366. The two paradigms are compared in 388. A concrete instantiation of the QIP is described in 296.

191. *3.6.2 Guidelines for Process Implementation and Change*

192. Process implementation and change is an instance of organizational change. Most successful organizational change efforts treat the change as a project in its own right, with appropriate plans, monitoring, and review.

193. Guidelines about process implementation and change within software engineering organizations, including action planning, training, management sponsorship and commitment, and the selection of pilot projects, and that cover both the transition of processes and tools, are given in 395311361379392385.An empirical study evaluating success factors for process change is reported in 323. Grady describes the process improvement experiences at HP, with some general guidance on implementing organizational change 335.

194. The role of change agents in this activity should not be underestimated. Without the enthusiasm, influence, credibility, and persistence of a change agent, organizational change has little chance of succeeding. This is further discussed in 343.

195. Process implementation and change can also be seen as an instance of consulting (either internal or external). A suggested text, and classic, on consulting is that of Schein 386.

196. One can also view organizational change from the perspective of technology transfer. The classic text on the stages of technology transfer is that by Rogers384. Software engineering articles that discuss technology transfer, and the characteristics of recipients of new technology (which could include process related technologies) are 377383.

197. *3.6.3 Evaluating the Outcome of Process Implementation and Change*

198. Evaluation of process implementation and change outcomes can be qualitative or quantitative. The topics above on qualitative analysis and measurement are relevant when

[5] This can also be termed "process evolution".

evaluating implementation and change since they describe the techniques. Below we present some conceptual issues that become important when evaluating the outcome of implementation and change.

199. There are two ways that one can approach evaluation of process implementation and change. One can evaluate it in terms of changes to the process itself, or in terms of changes to the process outcomes (for example, measuring the Return on Investment from making the change). This issue is concerned with the distinction between cause and effect (as depicted in the path diagram in Figure 3), and is discussed in 296.

200. Sometimes people have very high expectations about what can be achieved in studies that evaluate the costs and benefits of process implementation and change. A pragmatic look at what can be achieved from such evaluation studies is given in 338.

201. Overviews of how to evaluate process change, and examples of studies that do so can be found in 322334357362361367.

202. 4. KEY REFERENCES

203. The following are the key references that are recommended for this knowledge area. The mapping to the topics is given in Section 5.

204. K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

205. This IEEE edited book provides detailed chapters on the software process assessment and improvement area. It could serve as a general reference for this knowledge area, however, specifically chapters 1, 7, and 11 cover quite a bit of ground in a succinct manner.

206. K. El Emam, J-N Drouin, W. Melo: *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*. IEEE CS Press, 1998.

207. This IEEE edited book describes the emerging ISO/IEC 15504 international standard and its rationale. Chapter 3 provides a description of the overall architecture of the standard, which has since then been adopted in other assessment models.

208. S-L. Pfleeger: *Software Engineering: Theory and Practice*. Prentice-Hall, 1998.

209. This general software engineering reference has a good chapter, chapter 2, that discusses many issues related to the process modeling area.

210. Fuggetta and A. Wolf: *Software Process*, John Wiley & Sons, 1996.

211. This edited book provides a good overview of the process area, and covers modeling as well as assessment and improvement. Chapters 1 and 2 are reviews of modeling techniques and tools, and chapter 4 gives a good overview of the human and organizational issues that arise during process implementation and change.

212. R. Messnarz and C. Tully (eds.): *Better Software Practice for Business Benefit: Principles and Experiences*, IEEE CS Press, 1999.

213. This IEEE edited book provides a comprehensive perspective on process assessment and improvement efforts in Europe. Chapter 7 is a review of the costs and benefits of process improvement, with many references to prior work. Chapter 16 describes factors that affect the success of process improvement.

214. J. Moore: *Software Engineering Standards: A User's Road Map*. IEEE CS Press, 1998.

215. This IEEE book provides a comprehensive framework and guidance on software engineering standards. Chapter 13 is the process standards chapter.

216. N. Madhavji: "The Process Cycle". In *Software Engineering Journal*, 6(5):234-242, 1991.

217. This article provides an overview of different types of process definitions and relates them within an organizational context.

218. M. Dowson: "Software Process Themes and Issues". In *Proceedings of the 2^nd^ International Conference on the Software Process*, pages 54-62, 1993.

219. This article provides an overview of the main themes in the software process area. Although not recent, most of the issues raised are still valid today.

220. P. Feiler and W. Humphrey: "Software Process Development and Enactment: Concepts and Definitions". In *Proceedings of the Second International Conference on the Software Process*, pages 28-40, 1993.

221. This article was one of the first attempts to define terminology in the software process area. Most of its terms are commonly used nowadays.

222. L. Briand, C. Differding, and H. D. Rombach: "Practical Guidelines for Measurement-Based

Process Improvement". In *Software Process Improvement and Practice*, 2:253-280, 1996.

223. This article provides a pragmatic look at using measurement in the context of process improvement, and discusses most of the issues related to setting up a measurement program.

224. Software Engineering Laboratory: *Software Process Improvement Guidebook*. NASA/GSFC, Technical Report SEL-95-102, April 1996. (available from http://sel.gsfc.nasa.gov/doc-st/docs/95-102.pdf)

225. This is a standard reference on the concepts of the QIP and EF.

226. P. Fowler and S. Rifkin: *Software Engineering Process Group Guide*. Software Engineering Institute, Technical Report CMU/SEI-90-TR-24, 1990. (available from http://www.sei.cmu.edu)

227. This is the standard reference on setting up and running an SEPG.

228. M. Dorfmann and R. Thayer (eds.): *Software Engineering*, IEEE CS Press, 1997.

229. Chapter 11 of this IEEE volume gives a good overview of contemporary life cycle models.

230. K. El Emam and D. Goldenson: "An Empirical Review of Software Process Assessments". In *Advances in Computers*, 2000.

231. This chapter provides the most up-to-date review of evidence supporting process assessment and improvement, as well as a historical perspective on some of the early MIS work.

232. 5. KEY REFERENCES VS. TOPICS MAPPING

233. Below are the matrices linking the topics to key references. In an attempt to limit the number of references and the total number of pages, as requested, some relevant articles are not included in this matrix. The reference list below provides a more comprehensive coverage.

234. In the cells, where there is a tick indicates that the whole reference (or most of it) is relevant. Otherwise, specific chapter numbers are provided in the cell.

		Elements 318	SPICE 321	Pfleeger 376	Fuggetta 331	Messnarz 370	Moore 372	Madhavji 364	Dowson 313
235.	Basic Concepts and Definitions								
236.	Themes								√
237.	Terminology								
238.	Process Infrastructure								
239.	The Experience Factory								
240.	The Software Engineering Process								
241.	Process Measurement								
242.	Methodology in Process Measurement								
243.	Process Measurement Paradigms	Ch. 1, 7	Ch. 3						
244.	Process Definition								
245.	Types of Process							√	
246.	Life Cycle Models			Ch. 2					
247.	Software Life Cycle Process Models						Ch. 13		
248.	Notations for Process Definitions				Ch. 1				
249.	Process Definition Methods	Ch. 7							
250.	Automation			Ch. 2	Ch. 2				

	Elements	SPICE	Pfleeger	Fuggetta	Messnarz	Moore	Madhavji	Dowson
	318	321	376	331	370	372	364	313
251. Qualitative Process Analysis								
252. Process Definition Review	Ch. 7							
253. Root Cause Analysis	Ch. 7							
254. Process Implementation and Change								
255. Paradigms for Process Implementation and Change	Ch. 1, 7							
256. Guidelines for Process Implementation and Change	Ch. 11			Ch. 4	Ch. 16			
257. Evaluating the Outcome of Process Implementation and Change					Ch. 7			

		Feiler & Humphrey 326	Briand et al. 295	SEL 388	SEPG 329	Dorfmann & Thayer 312	El Emam & Goldenson 325
258.	Basic Concepts and Definitions						
259.	Themes						
260.	Terminology	√					
261.	Process Infrastructure						
262.	The Experience Factory			√			
263.	The Software Engineering Process Group				√		
264.	Process Measurement						
265.	Methodology in Process Measurement		√				√
266.	Process Measurement Paradigms		√				
267.	Process Definition						
268.	Types of Process Definitions						
269.	Life Cycle Models					Ch. 11	
270.	Software Life Cycle Process Models						
271.	Notations for Process Definitions						
272.	Process Definition Methods						
273.	Automation						

	Feiler & Humphrey 326	Briand et al. 295	SEL 388	SEPG 329	Dorfmann & Thayer 312	El Emam & Goldenson 325
274. Qualitative Process Analysis						
275. Process Definition Review		√				
276. Root Cause Analysis		√				
277. Process Implementation and change						
278. Paradigms for Process Implementation and Change			√	√		
279. Guidelines for Process Implementation and Change			√	√		√
280. Evaluating the Outcome of Process Implementation and Change			√			√

281. 6. GENERAL REFERENCES

282. [1]T. Abdel-Hamid and S. Madnick: *Software Project Dynamics: An Integrated Approach*. Prentice-Hall, 1991.

283. [2] W. Agresti: "The Role of Design and Analysis in Process Improvement". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

284. [3] L. Alexander and A. Davis: "Criteria for Selecting Software Process Models". In *Proceedings of COMPSAC'91*, pages 521-528, 1991.

285. [4] J. Armitage and M. Kellner: "A Conceptual Schema for Process Definitions and Models". In *Proceedings of the Third International Conference on the Software Process*, pages 153-165, 1994.

286. [5] S. Bandinalli, A. Fuggetta, L. Lavazza, M. Loi, and G. Picco: "Modeling and Improving an Industrial Software Process". In *IEEE Transactions on Software Engineering*, 21(5):440-454, 1995.

287. [6] R. Barbour:*Software Capability Evaluation– Version 3.0 : Implementation Guide for Supplier Selection.* Software Engineering Institute, Technical Report CMU/SEI-95-TR012, 1996. (available from http://www.sei.cmu.edu).

288. [7] N. Barghouti, D. Rosenblum, D. Belanger, and C. Alliegro: "Two Case Studies in Modeling Real, Corporate Processes". In*Software Process – Improvement and Practice*, Pilot Issue, 17-32, 1995.

289. [8] V. Basili, G. Caldiera, and G. Cantone: "A Reference Architecture for the Component Factory". In *ACM Transactions on Software Engineering and Methodology*, 1(1):53-80.1992.

290. [9] V. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page, and S. Waligora: "The Software Engineering Laboratory – An Operational Software Experience Factory". In *Proceedings of the International Conference on Software Engineering*, pages 370-381, 1992.

291. [10] V. Basili, S. Condon, K. El Emam, B. Hendrick, and W. Melo: "Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components". In *Proceedings of the 19th International Conference on Software Engineering*, pages 282-291, 1997.

292. [11] B. Boehm: "A Spiral Model of Software Development and Enhancement". In *Computer*, 21(5):61-72, 1988.

293. [12] L. Briand, V. Basili, Y. Kim, and D. Squire: "A Change Analysis Process to Characterize Software Maintenance Projects". In*Proceedings of the International Conference on Software Maintenance*, 1994.

294. [13] L. Briand, W. Melo, C. Seaman, and V. Basili: "Characterizing and Assessing a Large-Scale Software Maintenance Organization". In *Proceedings of the 17th International Conference on Software Engineering*, 1995.

295. [14] L. Briand, C. Differding, and H. D. Rombach: "Practical Guidelines for Measurement-Based Process Improvement". In *Software Process Improvement and Practice*, 2:253-280, 1996.

296. [15] L. Briand, K. El Emam, and W. Melo: "An Inductive Method for Software Process Improvement: Concrete Steps and Guidelines". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

297. [16] F. Budlong and J. Peterson: "Software Metrics Capability Evaluation Guide". The Software Technology Support Center, Ogden Air Logistics Center, Hill Air Force Base, 1995.

298. [17] I. Burnstein, T. Suwannasart, and C. Carlson: "Developing a Testing Maturity Model: Part I". In *Crosstalk*, pages 21-24, August 1996.

299. [18] I. Burnstein, T. Suwannasart, and C. Carlson: "Developing a Testing Maturity Model: Part II". In *Crosstalk*, pages 1926-24, September 1996.

300. [19] F. Coallier, J. Mayrand, and B. Lague: "Risk Management in Software Product Procurement". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

301. [20] D. Card: "Understanding Process Improvement". In *IEEE Software*, pages 102-103, July 1991.

302. [21] R. Chillarege, I. Bhandhari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong: "Orthogonal Defect Classification – A Concept for In-Process Measurement". In *IEEE Transactions on Software Engineering*, 18(11):943-956, 1992.

303. [22] R. Chillarege: "Orthogonal Defect Classification". In M. Lyu (ed.): *Handbook of*

Software Reliability Engineering, IEEE CS Press, 1996.

304. [23] A. Christie: *Software Process Automation: The Technology and its Adoption*. Springer Verlag, 1995.

305. [24] D. Coleman, P. Arnold, S. Godoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes: *Object-Oriented Development: The Fusion Method*. Englewood Cliffs, NJ:Prentice Hall, 1994.

306. [25] J. Collofello and B. Gosalia: "An Application of Causal Analysis to the Software Production Process". In *Software Practice and Experience*, 23(10):1095-1105, 1993.

307. [26] E. Comer: "Alternative Software Life Cycle Models". In M. Dorfmann and R. Thayer (eds.): *Software Engineering*, IEEE CS Press, 1997.

308. [27] B. Curtis, M. Kellner, and J. Over: "Process Modeling". In *Communications of the ACM*, 35(9):75-90, 1992.

309. [28] B. Curtis, W. Hefley, S. Miller, and M. Konrad: "The People Capability Maturity Model for Improving the Software Workforce". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

310. [29] A. Davis, E. Bersoff, and E. Comer: "A Strategy for Comparing Alternative Software Development Life Cycle Models". In *IEEE Transactions on Software Engineering*, 14(10):1453-1461, 1988.

311. [30] R. Dion: "Starting the Climb Towards the CMM Level 2 Plateau". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

312. [31] M. Dorfmann and R. Thayer (eds.): *Software Engineering*, IEEE CS Press, 1997.

313. [32] M. Dowson: "Software Process Themes and Issues". In *Proceedings of the 2nd International Conference on the Software Process*, pages 54-62, 1993.

314. [33] D. Drew: "Tailoring the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) to a Software Sustaining Engineering Organization". In *Proceedings of the International Conference on Software Maintenance*, pages 137-144, 1992.

315. [34] K. Dymond: "Essence and Accidents in SEI-Style Assessments or 'Maybe this Time the Voice of the Engineer Will be Heard'". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

316. [35] D. Dunnaway and S. Masters: *CMM-Based Appraisal for Internal Process Improvement (CBA IPI): Method Description*. Software Engineering Institute, Technical Report CMU/SEI-96-TR-007, 1996. (available from http://www.sei.cmu.edu)

317. [36] EIA: *EIA/IS 731 Systems Engineering Capability Model*. (available from http://www.geia.org/eoc/G47/index.html)

318. [37] K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

319. [39] K. El Emam and D. R. Goldenson: "SPICE: An Empiricist's Perspective". In *Proceedings of the Second IEEE International Software Engineering Standards Symposium*, pages 84-97, August 1995.

320. [39] K. El Emam, D. Holtje, and N. Madhavji: "Causal Analysis of the Requirements Change Process for a Large System". In *Proceedings of the International Conference on Software Maintenance*, pages 214-221, 1997.

321. [40] K. El Emam, J-N Drouin, W. Melo: *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*. IEEE CS Press, 1998.

322. [41] K. El Emam and L. Briand: "Costs and Benefits of Software Process Improvement". In R. Messnarz and C. Tully (eds.): *Better Software Practice for Business Benefit: Principles and Experiences*, IEEE CS Press, 1999.

323. [42] K. El Emam, B. Smith, P. Fusaro: "Success Factors and Barriers for Software Process Improvement: An Empirical Study". In R. Messnarz and C. Tully (eds.): *Better Software Practice for Business Benefit: Principles and Experiences*, IEEE CS Press, 1999.

324. [43] K. El Emam: "Benchmarking Kappa: Interrater Agreement in Software Process Assessments". In *Empirical Software Engineering: An International Journal*, 4(2):113-133, 1999.

325. [44] K. El Emam and D. Goldenson: "An Empirical Review of Software Process Assessments". In *Advances in Computers*, 2000.

326. [45] P. Feiler and W. Humphrey: "Software Process Development and Enactment: Concepts and Definitions". In *Proceedings of the Second International Conference on the Software Process*, pages 28-40, 1993.

327. [46] A. Finkelstein, J. Kramer, and B. Nuseibeh (eds.): *Software Process Modeling and Technology*. Research Studies Press Ltd., 1994.

328. [47] W. Florac and A. Carleton: *Measuring the Software Process: Statistical Process Control for Software Process Improvement*. Addison Wesley, 1999.

329. [48] P. Fowler and S. Rifkin: *Software Engineering Process Group Guide*. Software Engineering Institute, Technical Report CMU/SEI-90-TR-24, 1990. (available from http://www.sei.cmu.edu)

330. [49] D. Frailey: "Defining a Corporate-Wide Software Process". In *Proceedings of the 1st International Conference on the Software Process*, pages 113-121, 1991.

331. [50] A. Fuggetta and A. Wolf: *Software Process*, John Wiley & Sons, 1996.

332. [51] P. Garg and M. Jazayeri: *Process-Centered Software Engineering Environments*. IEEE CS Press, 1995.

333. [52] P. Garg and M. Jazayeri: "Process-Centered Software Engineering Environments: A Grand Tour". In A. Fuggetta and A. Wolf: *Software Process*, John Wiley & Sons, 1996.

334. [53] D. Goldenson, K. El Emam, J. Herbsleb, and C. Deephouse: "Empirical Studies of Software Process Assessment Methods". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

335. [54] R. Grady: *Successful Software Process Improvement*. Prentice Hall, 1997.

336. [55] D. Harel and M. Politi: *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, 1998.

337. [56] J. Henry and B. Blasewitz: "Process Definition: Theory and Reality". In *IEEE Software*, page 105, November 1992.

338. [57] J. Herbsleb: "Hard Problems and Hard Science: On the Practical Limits of Experimentation". In *IEEE TCSE Software Process Newsletter*, No. 11, pages 18-21, 1998. (available from http://www.seg.iit.nrc.ca/SPN)

339. [58] K. Huff: "Software Process Modeling". In A. Fuggetta and A. Wolf: *Software Process*, John Wiley & Sons, 1996.

340. [59] W. Humphrey: *Managing the Software Process*. Addison Wesley, 1989.

341. [60] W. Humphrey: *A Discipline for Software Engineering*. Addison Wesley, 1995.

342. [61] W. Humphrey: *An Introduction to the Team Software Process*. Addison-Wesley, 1999.

343. [62] D. Hutton: *The Change Agent's Handbook: A Survival Guide for Quality Improvement Champions*. Irwin, 1994.

344. [63] IEEE: *IEEE Standard for Developing Software Life Cycle Processes*. IEEE Std 1074-1991.

345. [64] IEEE: *IEEE Standard for Software Maintenance*, IEEE Std 1219-1992.

346. [65] IEEE: *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Std 1061-1998.

347. [66] IEEE: *IEEE Standard for the Classification of Software Anomalies*. IEEE Std 1044-1993.

348. [67] IEEE: *Guide for Information Technology - Software Life Cycle Processes - Life cycle data*. IEEE Std 12207.1-1998.

349. [68] IEEE: *Guide for Information Technology - Software Life Cycle Processes– Implementation. Considerations*. IEEE Std 12207.2-1998.

350. [69] ISO/IEC 12207: *Information Technology – Software Life Cycle Processes*. 1995.

351. [70] ISO/IEC TR 15504: *Information Technology – Software Process Assessment*, 1998. (parts 1-9; part 5 was published in 1999). Available from http://www.seg.iit.nrc.ca/spice.

352. [71] ISO/IEC CD 15939: *Information Technology – Software Measurement Process*, 2000.

353. [72] I. Jacobson, G. Booch, and J. Rumbaugh: *The Unified Software Development Process*. Addison-Wesley, 1998.

354. [73] P. Jalote: *An Integrated Approach to Software Engineering*. Springer, 1997.

355. [74] D. Johnson and J. Brodman: "Tailoring the CMM for Small Businesses, Small Organizations, and Small Projects". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

356. [75] C. Jones: *Applied Software Measurement*. McGraw-Hill, 1994.

357. [76] C. Jones: "The Economics of Software Process Improvements". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

358. [77] M. Kellner and G. Hansen: "Software Process Modeling: A Case Study". In

Proceedings of the 22nd International Conference on the System Sciences, 1989.

359. [78] M. Kellner, L. Briand, and J. Over: "A Method for Designing, Defining, and Evolving Software Processes". In *Proceedings of the 4th International Conference on the Software Process*, pages 37-48, 1996.

360. [79]M. Kellner, U. Becker-Kornstaedt, W. Riddle, J. Tomal, and M. Verlage: "Process Guides: Effective Guidance for Process Participants". In *Proceedings of the 5th International Conference on the Software Process*, pages 11-25, 1998.

361. [80] B. Kitchenham: "Selecting Projects for Technology Evaluation". In *IEEE TCSE Software Process Newsletter*, No. 11, pages 3 -6, 1998. (available from http://www.seg.iit.nrc.ca/SPN)

362. [81] H. Krasner: "The Payoff for Software Process Improvement: What it is and How to Get it". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

363. [82] J. Lonchamp: "A Structured Conceptual and Terminological Framework for Software Process Engineering". In *Proceedings of the Second International Conference on the Software Process*, pages 41-53, 1993.

364. [83] N. Madhavji: "The Process Cycle". In *Software Engineering Journal*, 6(5):234-242, 1991.

365. [84] S. Masters and C. Bothwell: *CMM Appraisal Framework – Version 1.0*. Software Engineering Institute, Technical Report CMU/SEI-TR-95-001, 1995. (available from http://www.sei.cmu.edu)

366. [85] B. McFeeley: *IDEAL: A User's Guide for Software Process Improvement*. Software Engineering Institute, Handbook CMU/SEI-96-HB-001, 1996. (available from http://www.sei.cmu.edu)

367. [86] F. McGarry, R. Pajerski, G. Page, S. Waligora, V. Basili, and M. Zelkowitz: *Software Process Improvement in the NASA Software Engineering Laboratory.* Software Engineering Institute, Technical Report CMU/SEI-94-TR-22, 1994.

368. [87] C. McGowan and S. Bohner: "Model Based Process Assessments". In *Proceedings of the International Conference on Software Engineering*, pages 202-211, 1993.

369. [88] N. Madhavji, D. Hoeltje, W. Hong, T. Bruckhaus: "Elicit: A Method for Eliciting Process Models". In *Proceedings of the Third International Conference on the Software Process*, pages 111-122, 1994.

370. [89] R. Messnarz and C. Tully (eds.): *Better Software Practice for Business Benefit: Principles and Experiences*, IEEE CS Press, 1999.

371. [90] K. Moller and D. Paulish:*Software Metrics*. Chapman & Hall, 1993.

372. [91] J. Moore: *Software Engineering Standards: A User's Road Map*. IEEE CS Press, 1998.

373. [92] T. Nakajo and H. Kume: "A Case History Analysis of Software Error Cause-Effect Relationship". In *IEEE Transactions on Software Engineering*, 17(8), 1991.

374. [93] Office of the Under Secretary of Defense for Acquisitions and Technology:*Practical Software Measurement: A Foundation for Objective Project Management*, 1998 (available from http://www.psmsc.com).

375. [94] M. Paulk and M. Konrad: "Measuring Process Capability Versus Organizational Process Maturity". In *Proceedings of the 4th International Conference on Software Quality*, 1994.

376. [95] S-L. Pfleeger: *Software Engineering: Theory and Practice*. Prentice-Hall, 1998.

377. [96] S-L Pfleeger: "Understanding and Improving Technology Transfer in Software Engineering". In *The Journal of Systems and Software*, 47:111-124, 1999.

378. [97] R. Pressman: *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1997.

379. [98] J. Puffer: "Action Planning". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

380. [99] L. Putnam and W. Myers: *Measures for Excellence: Reliable Software on Time, Within Budget.* Yourdon Press, 1992.

381. [100] R. Radice, N. Roth, A. O'Hara Jr., and W. Ciarfella: "A Programming Process Architecture". In *IBM Systems Journal*, 24(2):79-90, 1985.

382. [101] D. Raffo and M. Kellner: "Modeling Software Processes Quantitatively and Evaluating the Performance of Process Alternatives". In K. El Emam and N Madhavji

(eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

383. [102] S. Raghavan and D. Chand: "Diffusing Software-Engineering Methods". In *IEEE Software*, pages 81-90, July 1989.

384. [103] E. Rogers: *Diffusion of Innovations*. Free Pressm 1983.

385. [104] M. Sanders (ed.): *The SPIRE Handbook: Better, Faster, Cheaper Software Development in Small Organisations*. Published by the European Comission, 1998.

386. [105] E. Schein: *Process Consultation Revisited: Building the Helping Relationship*. Addison-Wesley, 1999.

387. [106] Software Engineering Institute: *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison Wesley, 1995.

388. [107] Software Engineering Laboratory: *Software Process Improvement Guidebook*. NASA/GSFC, Technical Report SEL-95-102, April 1996. (available from http://sel.gsfc.nasa.gov/doc-st/docs/95-102.pdf)

389. [108] Software Engineering Laboratory: *Software Measurement Guidebook*. NASA/GSFC, Technical Report SEL-94-002, July 1994.

390. [109] Software Productivity Consortium: *Process Definition and Modeling Guidebook*. SPC-92041-CMC, 1992.

391. [110] R. van Solingen and E. Berghout: *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. McGraw Hill, 1999.

392. [111] I. Sommerville and T. Rodden: "Human, Social and Organisational Influences on the Software Process". In A. Fuggetta and A. Wolf: *Software Process*, John Wiley & Sons, 1996.

393. [112] I. Sommerville and P. Sawyer: *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, 1997.

394. [113] H. Steinen: "Software Process Assessment and Improvement: 5 Years of Experiences with Bootstrap". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

395. [114] K. Wiegers: *Creating a Software Engineering Culture*. Dorset house, 1996.

396. [115] S. Weissfelner: "ISO 9001 for Software Organizations". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

397. [116] E. Yu and J. Mylopolous: "Understanding 'Why' in Software Process Modeling, Analysis, and Design". In *Proceedings of the 16th International Conference on Software Engineering*, 1994.

398. [117] S. Zahran: *Software Process Improvement: Practical Guidelines for Business Success*. Addison Wesley, 1998.

CHAPTER 10
SOFTWARE ENGINEERING TOOLS AND METHODS

David Carrington
Department of Computer Science and Electrical Engineering
The University of Queensland
Brisbane, Qld 4072
Australia
+61 7 3365 3310
davec@csee.uq.edu.au

TABLE OF CONTENTS

1. 1. INTRODUCTION

2. This document provides an initial breakdown of topics within the Software Engineering Infrastructure Knowledge Area as defined by the document “Approved Baseline for a List of Knowledge Areas for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge”. Earlier versions of this Knowledge Area included material on integration and reuse, but this has been removed. Consequently the Knowledge Area has been renamed from “Software Engineering Infrastructure” to “Software Engineering Tools and Methods”.

3. The five texts [DT97, Moo98, Pfl98, Pre97, and Som96] have been supplemented by Tucker [Tuc96], who provides nine chapters on software engineering topics. In particular, Chapter 112, “Software Tools and Environments” by Steven Reiss [Rei96] was particularly helpful for this Knowledge Area. Specialized references have been identified for particular topics, e.g., Object-oriented development.

4. 2. DEFINITION OF KNOWLEDGE AREA

5. The Software Engineering Tools and Methods Knowledge Area includes both the development methods and the software development environments knowledge areas identified in the Straw Man version of the guide.

6. Development methods impose structure on the software development activity with the goal of making the activity systematic and ultimately more likely to be successful. Methods usually provide a notation and vocabulary, procedures for performing identifiable tasks and guidelines for checking both the process and the product. Development methods vary widely in scope, from a single life cycle phase to the complete life cycle. The emphasis in this Knowledge Area is on methods that encompass multiple lifecycle phases since phase-specific methods are likely to be covered in other Knowledge Areas.

7. Software development environments are the computer-based tools that are intended to assist the software development process. Tools allow repetitive, well-defined actions to be automated, thus reducing the cognitive load on the software engineer. The engineer is then free to concentrate on the creative aspects of the process. Tools are often designed to support particular methods, reducing any administrative load associated with applying the method manually. Like methods, they are intended to make development more systematic, and they vary in scope from supporting individual tasks to encompassing the complete life cycle.

8. 3. BREAKDOWN OF TOPICS

9. This section contains a top-level breakdown of topics in the Software Engineering Tools and Methods Knowledge Area.

10. I. Software Tools

11. A. Software Requirements Tools

12. B. Software Design Tools

13. C. Software Construction Tools

14. 1. program editors

15. 2. compilers

16. 3. debuggers

17. D. Software Testing Tools

18. 1. test generators

19. 2. test execution frameworks

20. 3. test evaluation tools

21. 4. test management tools

22. E. Software Maintenance Tools

23. 1. comprehension tools

24. 2. reverse engineering tools

25. 3. re-engineering tools

26. 4. traceability tools

27. F. Software Engineering Process Tools

28. 1. integrated CASE environments

29. 2. process-centered software engineering environments

30. 3. process modeling tools

31. G. Software Quality Tools

32. 1. inspection tools

33. 2 static analysis tools

34. 3. performance analysis tools

35. H. Software Configuration Management Tools

36. 1. version management tools

37. 2. release and build tools

38. I. Software Engineering Management Tools

39. 1. project planning and tracking tools

40. 2. risk analysis and risk management tools

41. 3. measurement tools

42. 4. defect, enhancement, issue and problem tracking tools

43. J. Infrastructure support tools

44. 1. interpersonal communication tools

45. 2. information retrieval tools

46. 3. system administration and support tools

47. K. Miscellaneous

48. 1. tool integration techniques

49. 2. meta tools

50. 3. tool evaluation

51. II. Software Development Methods

52. A. Heuristic methods

53. 1. ad-hoc (unstructured) methods

54. 2. structured methods

55. 3. data-oriented methods

56. 4. object-oriented methods

57. 5. domain-specific methods:

58. B. Formal methods

59. 1. specification languages & notations

60. 2. refinement

61. 3. verification

62. C. Prototyping methods

63. 1. styles

64. 2. prototyping target

65. 3. evaluation techniques

66. D. Miscellaneous

67. 1. method evaluation

68. Software Tools

69. The top-level partitioning of the Software Tools section uses the same structure as the Stone Man Version of the Guide to the Software Engineering Body of Knowledge. The first five subsections correspond to the five Knowledge Areas (Requirements, Design, Construction, Testing, and Maintenance) that correspond to a phase of a software lifecycle, so these sections provide a location for phase-specific tools. The next four subsections correspond to the remaining Knowledge Areas (Process, Quality, Configuration Management and Management), and provide a location for phase-independent tools that are associated with activities described in these Knowledge Areas. Two additional subsections are provided: one for infrastructure support tools that do not fit in any of earlier sections, and a Miscellaneous subsection for topics, such as tool integration techniques, that are potentially applicable to all classes of tools. Because software engineering tools evolve rapidly and continuously, the hierarchy and

description avoids discussing particular tools as far as possible.

70. *Software Requirements Tools*

71. Tools used for eliciting, recording, analysing and validating software requirements belong in this section.

72. *Software Design Tools*

73. This section covers tools for creating and checking software designs. There is a variety of such tools, with much of this variety being a consequence of the diversity of design notations and methods.

74. *Software Construction Tools*

75. Program editors are tools used for creation and modification of programs (and possibly associated documents). These tools can be general-purpose text or document editors, or they can be specialized for a target language. Editing refers to human-controlled development tools whereas compilers are generally not interactive. Some environments provide both interactive editing and compilation via one interface. The compilers topic also covers pre-processors, linkers/loaders, and code generators. Debugging tools have been made a separate topic since they support the construction process but are different from program editors or compilers.

76. *Software Testing Tools*

77. Testing tools can be categorized according to where in the testing process they are used. Test generators assist the development of test cases. Test execution frameworks enable the execution of test cases in a controlled environment where the behavior of the object under test is observed. Test evaluation tools support the assessment of the results of test execution, helping todetermine whether the observed behavior conforms to the expected behavior. Test management tools provide support for the testing process.

78. *Software Maintenance Tools*

79. The first topic in this section concerns tools to assist human comprehension of programs. Example tools include visualization tools such as animators and program slicers. The next topic is reverse engineering tools that assist the process of working backwards from an existing product to create artefacts such as design and specification descriptions. Re-engineering tools extend this approach by applying transformations to generate a new product from an old one. Such tools allow translation of a program to a new programming language, or a database to a new format. Traceability tools have been included in this section since a major goal of traceability is to facilitate maintenance.

80. *Software Engineering Process Tools*

81. Computer-aided software engineering tools or environments that cover multiple phases of the software development lifecycle have been incorporated in this section. Such tools perform multiple functions and hence potentially interact with the software process that is being enacted. The second topic covers those environments that explicitly incorporate software process information and that guide and monitor the user according to a defined process. The third topic covers tools to model and investigate software processes.

82. *Software Quality Tools*

83. The first topic in this section covers tools to support reviews and inspections. The second topic deals with tools that analyse software artefacts, such as syntactic and semantic analysers, and data, control flow and dependency analysers. Such tools are intended for checking software artefacts for conformance or for verifying desired properties. The third topic deals with analysis of dynamic behaviour or performance.

84. *Software Configuration Management Tools*

85. Tools for configuration management have been categorized as either related to version management or to software release and build management.

86. *Software Engineering Management Tools*

87. Management tools have been subdivided into four categories: project planning and tracking, risk analysis and risk management, measurement, and tools for tracking defects, enhancements, issues and problems.

88. *Infrastructure support tools*

89. This section covers tools that provide interpersonal communication, information retrieval, and system administration and support.

These tools, such as e-mail, databases, web browsers and file backup tools, are generally not specific to a particular lifecycle stage, nor to a particular development method.

90. *Miscellaneous*

91. This section covers tool integration techniques, meta-tools and tool evaluation. Tool integration is important for making individual tools cooperate. The kinds of tool integration are platform, presentation, process, data, and control [Sommeville, Section 25.2]. Meta-tools generate other tools; compiler-compilers are the classic example. Because of the continuous evolution of software engineering tools, tool evaluation is an important topic.

92. Software Development Methods

93. This section is divided into four subsections: *heuristic methods* dealing with informal approaches, *formal methods* dealing with mathematically based approaches, *prototyping methods* dealing with software development approaches based on various forms of prototyping, and *miscellaneous*. The first three subsections are not disjoint; rather they represent distinct concerns. For example, an object-oriented method may incorporate formal techniques and rely on prototyping for verification and validation. Like software engineering tools, methodologies evolve continuously. Consequently, the Knowledge Area description avoids naming particular methodologies as far as possible.

94. *Heuristic methods*

95. This subsection contains five categories: *ad-hoc, structured, data-oriented, object-oriented and domain-specific.* The domain-specific category includes specialized methods such as real-time development methods.

96. *Formal methods*

97. This subsection deals with mathematically based development methods and is subdivided by different aspects of formal methods. Topic 1 is the specification notation or language used. Specification languages are commonly classified as model-oriented, property-oriented or behavior-oriented. Topic 2 deals with how the method refines (or transforms) the specification into a form that is closer to the desired final form of an executable program. Topic 3 covers the verification properties that are specific to the formal approach and covers both theorem proving and model checking.

98. *Prototyping methods*

99. The third subsection covers methods involving software prototyping and is subdivided into prototyping styles, targets and evaluation techniques. The topic of prototyping styles identifies the different approaches: throwaway, evolutionary and the executable specification. Example targets of a prototyping method may be requirements, architectural design or the user interface.

100. *Miscellaneous*

101. The final subsection is intended to cover topics not covered elsewhere. The only topic identified so far is method evaluation.

102. Links to common themes

103. *Quality*

104. Development methods are intended to provide guidance to software developers, primarily with the goal of making it easier to produce a high quality product. Different methods emphasize different software qualities. Software tools also contribute to quality by automating activities thus assisting the software developer.

105. *Standards*

106. Software engineering standards represent the collected wisdom and conventions of the software engineering community. As methods mature and gain widespread use, standardization provides a way to codify the knowledge. No standards for software development methodologies have been identified for this document although individual methods are standardized. For software tools, the relevant IEEE standards are:

107. ◆ Trial-Use Standard Reference Model for Computing System Tool Interconnections, IEEE Std 1175-1992

108. ◆ IEEE Recommended Practice for the Evaluation and Selection of CASE Tools, IEEE Std 1209-1992 (ISO/IEC 14102)

109. ◆ IEEE Recommended Practice for the Adoption of CASE Tools, IEEE Std 1348-1995 (ISO/IEC 14471).

110. Two relevant ECMA standards are:

111. - ECMA TR/55 Reference Model for Frameworks of Software Engineering Environments, 3rd edition, June 1993,

112. - ECMA TR/69 Reference Model for Project Support Environments, December 1994.

113. *Measurement*

114. Specific development methods often incorporate particular measurements. Tools can assist software developers perform measurement activities and this is a specific category of management tools.

115. 4. BREAKDOWN RATIONALE

116. The Stone Man Version of the Guide to the Software Engineering Body of Knowledge conforms at least partially with the partitioning of the software life cycle in the ISO/IEC 12207 Standard [ISO95]. Some Knowledge Areas, such as this one, are intended to cover knowledge that applies to multiple phases of the life cycle. One approach to partitioning topics in this Knowledge Area would be to use the software life cycle phases. For example, software methods and tools could be classified according to the phase with which they are associated. This approach was not seen as effective. If software engineering infrastructure could be cleanly partitioned by life cycle phase, it would suggest that this Knowledge Area could be eliminated by allocating each part to the corresponding life cycle Knowledge Area, e.g., infrastructure for software design to the Software Design Knowledge Area. Such an approach would fail to identify the commonality of, and interrelationships between, both methods and tools in different life cycle phases. However since tools are a common theme to most Knowledge Areas, several reviewers of Version 0.5 of this Knowledge Area suggested that a breakdown based on Knowledge Area for tools would be helpful. This suggestion was endorsed by the Industry Advisory Board.

117. There are many links between methods and tools, and one possible structure would seek to exploit these links. However because the relationship is not a simple "one-to-one" mapping, this structure has not been used to organize topics in this Knowledge Area. This does mean that these links are not explicitly identified.

118. Some topics in this Knowledge Area do not have corresponding reference materials identified in the matrices in Appendix 2. There are two possible conclusions: either the topic area is not relevant to this Knowledge Area, or additional reference material needs to be identified. Feedback from reviewers will be helpful to resolve this issue.

119. 5. MATRIX OF TOPICS VS REFERENCE MATERIAL

120. The matrices in the Appendix indicate for each topic sources of information within the selected references (see Section 2).

121. 6. RECOMMENDED REFERENCES

122. This section briefly describes each of the recommended references.

123. [CW96] Edmund M. Clarke et al. Formal Methods: State of the Art and Future Directions.

124. This tutorial on formal methods explains techniques for formal specification, model checking and theorem proving, and describes some successful case studies and tools.

125. [DT97] Merlin Dorfman and Richard H. Thayer (eds.). Software Engineering.

126. This tutorial volume contains a collection of papers organized into chapters. The following papers are referenced (section numbers have been added to reference individual papers more conveniently in the matrices in the Appendix):

127. Chapter 4: Software Requirements Engineering and Software Design

128. 4.1 Software Requirements: A Tutorial, Stuart Faulk

129. 4.2 Software Design: An Introduction, David Budgen

130. Chapter 5: Software Development Methodologies

131. 5.1 Object-oriented Development, Linda M. Northrup

132. 5.2 Object-oriented Systems Development: Survey of Structured Methods, A.G. Sutcliffe

133. 5.4 A Review of Formal Methods, Robert Vienneau

134. Chapter 7: Software Validation, Verification and Testing

135. 7.4 Traceability, James D. Palmer

136. Chapter 12 Software Technology

137. 12.2 Prototyping: Alternate Systems Development Methodology, J.M. Carey

138. 12.3 A Classification of CASE Technology, Alfonso Fuggetta

139. [Pfl98] S.L. Pfleeger. Software Engineering — Theory and Practice.

140. This text is structured according to the phases of a life cycle so that discussion of methods and tools is distributed throughout the book.

141. [Pre97] R.S. Pressman. Software Engineering — A Practitioner's Approach (4th Ed.)

142. Chapter 29 covers "Computer-Aided Software Engineering" including a taxonomy of case tools (29.3). There is not much detail about any particular class of tool but it does illustrate the wide range of software engineering tools. The strength of this book is its description of methods with chapters 10-23 covering heuristic methods, chapters 24 and 25 covering formal methods. Section 11.4 describes prototyping methods and tools.

143. [Rei96] Steven P. Reiss. Software Tools and Environments

144. This chapter from [Tuc96] provides an overview of software tools. The emphasis is on programming tools rather than tools for analysis and design although CASE tools are mentioned briefly.

145. [Som96] Ian Sommerville. Software Engineering (5th Ed.)

146. Chapters 25, 26 and 27 introduce computer-aided software engineering with the emphasis being on tool integration and large-scale environments. Static analysis tools are covered in Section 24.3. Chapter 9, 10 and 11 introduce formal methods with formal verification being described in Section 24.2 and the Cleanroom method in Section 24.4. Prototyping is discussed in Chapter 8.

147. [Was96] Anthony I. Wasserman. Towards a Discipline of Software Engineering

148. This general article discusses the role of both methods and tools in software engineering. Although brief, the paper integrates the major themes of the discipline.

149. 7. LIST OF FURTHER READINGS

150. *A commentary on the additional reference material listed in the bibliography is to be added in this section.*

151. 8. ACKNOWLEDGMENTS

152. This document was developed from the jump-start document written by Tuong Vinh Ho. Phil Cook, Andrew Coyle, Ian MacColl and Jim Welsh helped identify relevant information sources and offered helpful advice. Their contribution is gratefully acknowledged. The two reviewers for version 0.1, Don Bagert and Jorge Diaz-Herrera, provided useful advice about topics and structure (see Appendix 3). Valuable feedback from numerous reviewers of version 0.5 has been incorporated into the document. A separate document detailing this feedback and its disposition is available on the SWEBOK website (www.swebok.org).

153. 9. REFERENCES

154. Edward V. Berard. Essays on Object-oriented software Engineering. Prentice-Hall, 1993.

155. Edmund M. Clarke, Jeanette M. Wing et al. Formal Methods: State of the Art and Future Directions. ACM Computer Surveys, 28(4):626-643, 1996.

156. Derek Coleman et al. Object-Oriented Development: The Fusion Method. Prentice Hall, 1994.

157. Dan Craigen, Susan Gerhart and Ted Ralston. Formal Methods Reality Check: Industrial Usage, IEEE Transactions on Software Engineering, 21(2):90-98, February 1995.

158. Merlin Dorfman and Richard H. Thayer, Editors. Software Engineering. IEEE Computer Society, 1997.

159. ECMA. TR/55 Reference Model for Frameworks of Software Engineering Environments, 3rd edition, June 1993.

160. ECMA TR/69 Reference Model for Project Support Environments, December 1994.

161. Pankaj K. Garg and Mehdi Jazayeri. Process-Centered Software Engineering Environments, IEEE Computer Society, 1996.

162. IEEE. Trial-Use Standard Reference Model for Computing System Tool Interconnections, IEEE Std 1175-1992.

163. IEEE. Recommended Practice for the Evaluation and Selection of CASE Tools, IEEE Std 1209-1992 (ISO/IEC 14102, 1995).

164. IEEE Recommended Practice for the Adoption of CASE Tools, IEEE Std 1348-1995 (ISO/IEC 14471).

165. ISO/IEC Standard for Information Technology —Software Life Cycle Processes, ISO/IEC 12207 (IEEE/EIA 12207.0-1996), 1995.

166. Stan Jarzabek and Riri Huang. The Case for User-Centered CASE Tools, Communications of the ACM, 41(8):93-99, August 1998.

167. B. Kitchenham, L. Pickard, and S.L. Pfleeger. Case Studies for Method and Tool Evaluation, IEEE Software, 12(4):52-62, July 1995.

168. Bertrand Meyer. Object-oriented Software Construction (2nd Ed.). Prentice Hall, 1997.

169. James W. Moore. Software Engineering Standards: A User's Road Map. IEEE Computer Society, 1998.

170. Vicky Mosley. How to Assess Tools Efficiently and Quantitatively, IEEE Software, 9(3):29-32, May 1992.

171. H.A. Muller, R.J. Norman and J. Slonim (eds.). Computer Aided Software Engineering, Kluwer, 1996. (A special issue of Automated Software Engineering, 3(3/4), 1996).

172. Shari Lawrence Pfleeger. Software Engineering: Theory and Practice. Prentice Hall, 1998.

173. R.M. Poston. Automating specification-based Software Testing. IEEE, 1996.

174. Roger S. Pressman. Software Engineering: A Practitioner's Approach. 4th edition, McGraw-Hill, 1997.

175. Steven P. Reiss. Software Tools and Environments, Ch. 112, pages 2419-2439. In Tucker [Tuc96], 1996.

176. C. Rich and R.C. Waters. Knowledge Intensive Software Engineering Tools, IEEE Transactions on Knowledge and Data Engineering, 4(5):424-430, October 1992.

177. Ian Sommerville. Software Engineering. 5th edition, Addison-Wesley, 1996.

178. Xiping Song and Leon J. Osterweil. Towards Objective, Systematic Design-Method Comparisons, IEEE Software, 9(3):43-53, May 1992.

179. Allen B. Tucker, Jr., Editor-in-chief. The Computer Science and Engineering Handbook. CRC Press, 1996.

180. Walter G. Vincenti. What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. John Hopkins University Press, 1990.

181. Anthony I. Wasserman. Toward a Discipline of Software Engineering, IEEE Software, 13(6): 23-31, November 1996.

182. APPENDIX: TOPIC VS REFERENCE MATERIAL MATRICES

I. Software Tools	CW96	DT97	Pfl98	Pre97	Rei96	Som96	Was96	Other
A. Software Requirements Tools		4.1 pp.98-100 12.3		11.4.2, 29.3		26.2		
B. Software Design Tools		12.3		29.3		26.2		
C. Software Construction Tools		12.3		29.3	112.2	26.1		
1. program editors								
2. compilers								
3. debuggers								
D. Software Testing Tools		12.3	7.7, 8.7	29.3	112.3	26.3		
1. test generators								
2. test execution frameworks								
3. test evaluation tools								
4. test management								
E. Software Maintenance Tools		12.3	10.5	29.3				
1. comprehension tools					112.5			
2. Reverse engineering tools								
3. Re-engineering tools								
4. traceability tools		7.4 pp.273-4						
F. Software Engineering Process Tools		12.3				25, 26, 27		
1. integrated CASE environments				29	112.3, 112.4			
2. Process-centered software engineering environments				29.6	112.5			
3. Process modeling tools			2.3, 2.4					
G. Software Quality Tools		12.3						
1. inspection tools								
2. static analysis tools	✓		7.7	29.3	112.5	24.3		
3. performance analysis tools					112.5			
H. Software Configuration Management Tools		12.3	10.5		112.3			
1. version management tools				29				
2. release and build tools				29.3				
I. Software Engineering Management Tools		12.3						
1. project planning and tracking tools				29.3				
2. risk analysis and management tools								
3. measurement tools				29.3				
4. defect, enhancement, issue and problem tracking tools				29.3				
J. Infrastructure Support Tools		12.3						
1. interpersonal communication tools				29.3				
2. information retrieval tools				29.3				
3. system administration and support tools				29.3				
K. Miscellaneous		12.3						
1. tool integration techniques			1.8 (p.35)		112.4		✓	
2. meta tools								
3. tool evaluation			8.10 (p.388)					

II. Development Methods	**CW96**	**DT97**	**Pfl98**	**Pre98**	**Som96**	**Was96**	**Other**
A. Heuristic Methods				10-23		✓	
1. ad-hoc methods							
2. structured methods		4.2, 5.2	4.5	10-18	15		
3 data-oriented methods		4.2, 5.2		12.8			
4 object-oriented methods		5.1, 5.2	4.4, 7.5	19-23	6.3, 14		
5 domain-specific methods				15	16		
B. Formal Methods		5.4		24, 25	9-11, 24.4		
1. specification languages	✓		4.5	24.4			
2. refinement				25.3			
3. verification/proving properties	✓		5.7, 7.3		24.2		
C. Prototyping Methods				2.5	8	✓	
1. styles		12.2	4.6, 5.6	11.4			
2. prototyping targets		12.2					
3. evaluation techniques							
D. Miscellaneous							
1. Method evaluation							

CHAPTER 11
SOFTWARE QUALITY

Dolores Wallace and Larry Reeker
National Institute of Standards and Technology
Gaithersburg, Maryland 20899 USA
{Dolores.Wallace, Larry.Reeker}@NIST.gov

TABLE OF CONTENTS

1. 1. INTRODUCTION: DEFINING THE KNOWLEDGE AREA

2. Software Quality Assurance (SQA) and Verification and Validation (V&V) are the processes of the Knowledge Area on Softwa re Quality. The scope of this Knowledge Area is the quality of the product being produced by the Software Engineer, where the term “product” means any artifact that is the output of any process used to build the final software product. Examples of a product include, but are not limited to, an entire system specification, a software requirements specification for a software component of a system, a design module, code, test documentation, or reports from quality analysis tasks. While most treatments of quality are described in terms of the final system’s performance, sound engineering practice requires that intermediate products relevant to quality be checked throughout the development and maintenance process.

3. Because of the pervasiveness of quality considerations in software, there is a large body of literature on the subject, and the authors have had to make difficult choices. It is necessary to limit the number of specific references to make the SWEBOK maximally useful as a distillation of the knowledge of the field, so the basic set of *Core References* is included for the topics covered herein. Other authors may have chosen different or additional references, but these cover the points that are most essential. A set of *Additional Readings* includes some additional books and articles that the authors wish to call to the attention of the reader. In addition, the remainder of the books and articles from which the core references have been specified might be useful to the reader. Even the extended reading set, though, does not cover everything that might be found useful to a person interested in Software Quality, and new material appears regularly.

4. The reader will notice many pointers to other knowledge areas (KAs) in the SWEBOK. This is again an expression of the ubiquity of software quality concerns within the field of Software Engineering. There may be some duplication of material between this knowledge area and the other KAs, but the pointers are intended to minimize such duplication.

5. 2. TOPIC BREAKDOWN FOR SOFTWARE QUALITY

6. The quality of a given product is sometimes defined as "the totality of characteristics [of the product] that bear on its ability to satisfy stated or implied needs" [1]. Quality is software is sometimes also defined as “the efficient, effective, and comfortable use by a given set of users for a set of purposes under specified conditions”. These two definitions can be much the same if the requirements are properly elicited, but both of them require some way of communicating to the engineer what will constitute quality for the given system. In this chapter, therefore, the first topic is the meaning of quality and some of the product characteristics that relate to it. The Knowledge Area on Software Requirements deals with how these qualities will be elicited and expressed.

7. Sections on the processes SQA and V&V that focus on software quality follow the discussion on software quality concepts. These quality-focused processes help to ensure better software. They also provide information needed to improve the quality of the entire software and maintenance processes. The knowledge areas Software Engineering Process and Software Engineering Management, discuss quality programs for the organization developing software systems, which use the results of SQA and V&V for improving the quality of the process.

8. Engineering for quality requires the measurement of quality in a concrete way, so this knowledge area contains a section on measurement as applied to SQA and V&V. Other processes for assuring software product quality are discussed in other parts of the SWEBOK. One of these, singled out in SWEBOK as a separate knowledge area within the software life cycle, Software Testing, is also used in both SQA and V&V. Another process fundamental to the software development and maintenance and also important to software quality is Software Configuration Management.

	SOFTWARE QUALITY KNOWLEDGE AREA
9.	1. Introduction: Defining the Knowledge Area
10.	2. Topic Breakdown for Software Quality
11.	3. Software Quality Concepts
12.	3.1 Measuring the Value of Quality
13.	3.2 ISO 9126 Quality Description
14.	3.3 Dependability
15.	3.4 Special Types of Systems and Quality Needs
16.	3.5 Quality Attributes for Engineering Process
17.	4. Defining SQA and V&V
18.	5. Planning for SQA and V&V
19.	5.1 The SQA Plan
20.	5.2 The V&V Plan
21.	6. Activities and Techniques for SQA and V&V
22.	6.1 Static Techniques
23.	6.1.1 Audits, Reviews, and Inspections
24.	6.1.2 Analytic Techniques
25.	6.2 Dynamic Techniques
26.	7. Measurement Applied to SQA and V&V
27.	7.1 Fundamentals of Measurement
28.	7.2 Metrics
29.	7.3 Measurement Techniques
30.	7.4 Defect Characterization
31.	7.5 Additional uses of SQA and V&V data
32.	8. References
33.	8.1 Topics and References
34.	8.2 Reference Lists

35. 3. SOFTWARE QUALITY CONCEPTS

36. What is software quality, and why is it so important that it is pervasive in the Software Engineering Body of Knowledge? Within an information system, software is a tool, and tools have to be selected for quality and for appropriateness. That is the role of requirements. But software is more than a tool. It dictates the performance of the system, and it is therefore important to the system quality. Much thought must therefore go into the value to place on each quality desired and on the overall quality of the information system. This section discusses the value and the attributes of quality.

37. The notion of “quality” is not as simple as it may seem. For any engineered product, there are many *desired qualities* relevant to a particular project, to be discussed and determined at the time that the product requirements are determined. Qualities may be present or absent, or may be matters of degree, with tradeoffs among them, with practicality and cost as major considerations. The software engineer has a responsibility to elicit the system’s quality requirements that may not be explicit at the outset and to discuss their importance and the difficulty of attaining them. All processes associated with software quality (e.g. building,

[1] From *Quality—Vocabulary*, (ISO 8402: 1986, note 1).

checking, improving quality) will be designed with these in mind and carry costs based on the design. Thus, it is important to have in mind some of the possible attributes of quality.

38. Various researchers have produced models (usually taxonomic) of software quality characteristics or attributes that can be useful for discussing, planning, and rating the quality of software products. The models often include metrics to "measure" the degree of each quality attribute the product attains. Usually these metrics may be applied at any of the product levels. They are not always direct measures of the quality characteristics of the finished product, but may be relevant to the achievement of overall quality. Some of the classical thinking in this area is found in McCall, Boehm [Boe78], and others and is discussed in the texts of Pressman [Pr], Pfleeger [Pf] and Kan [Kan94]. Each model may have a different set of attributes at the highest level of the taxonomy, and selection of and definitions for the attributes at all levels may differ. The important point is that the system software requirements define the quality requirements and the definitions of the attributes for them.

39. 3.1 Measuring the Value of Quality

40. A motivation behind a software project is a determination that it has a value, and this value may or not be quantified as a cost, but the customer will have some maximum cost in mind. Within that cost, the customer expects to attain the basic purpose of the software and may have some expectation of the necessary quality, or may not have thought through the quality issues or cost. The software engineer, in discussing software quality attributes and the processes necessary to assure them, should keep in mind the value of each one. Is it merely an adornment or it essential to the system? If it is somewhere in between, as almost everything is, it is a matter of making the customer fully aware of both costs and benefits. There is no definite rule for how this is done, but it is good for the software engineer to have some notion of how to go about this process. A discussion of measuring cost and value of quality requirements can be found in [Wei93], Chapter 8, pp118-134] and [Jon91], Chapter 5.

41. 3.2 ISO 9126 Quality Description

42. Terminology for quality attributes differs from one model to another; each model may have different numbers of hierarchical levels and a different total number of attributes. One attempt to standardize terminology in an inclusive model resulted in ISO 9126 (*Information Technology-Software Product Quality,* Part 1: Quality Model, 1998), of which a synopsis is included in this KA as Table 1. ISO 9126 is concerned primarily with the definition of quality characteristics in the final product. ISO 9126 sets out six quality characteristics, each very broad in nature. They are divided into 21 sub-characteristics. In the 1998 revision, "compliance" to application-specific requirements is included as a sub-characteristic of each characteristic.

43. Some terms for characteristics and their attributes are used differently in the other models mentioned above, but ISO 9126 has taken the various sets and arrangements of quality characteristics and has reached consensus for that model. Other models may have different definitions for the same attribute. A software engineer understands the underlying meanings of quality characteristics regardless of their names, as well as their value to the system under development or maintenance.

44. 3.3 Dependability

45. For systems whose failure may have extremely severe consequences, dependability of the overall system (hardware, software, and humans) is the main goal in addition to the realization of basic functionality. Software dependability is the subject of IEC 50-191 and the IEC 300 series of standards. Some types of systems (e.g., radar control, defense communications, medical devices) have particular needs for high dependability, including such attributes as fault tolerance, safety, security, usability. Reliability is a criterion under dependability and also is found among the ISO/IEC 9126 (Table 1). Reliability is defined similarly, but not identically, in the two places. In Moore's treatment [M], Kiang's factors are used as shown in the following list, with the exception of the term Trustability from Laprie.

46. - Availability: The product's readiness for use on demand

47. - Reliability: The longevity of product performance

48. - Maintainability: The ease of maintenance and upgrade

49. - Maintenance support: Continuing support to achieve availability performance objectives

50. - Trustability: System's ability to provide users with information about service correctness.

51. There is a large body of literature for systems that must be highly dependable ("high confidence" or "high integrity systems"). Terminology from systems that do not include software have been imported for discussing threats or hazards, risks, system integrity, and related concepts, and may be found in the references cited for this section.

52. 3.4 Special Types of Systems and Quality Needs

53. As implied above, there are many particular qualities of software that may or may not fit under ISO 9126. Particular classes of application systems may have other quality attributes to be judged. This is clearly an open-ended set, but the following are examples:

54. - Intelligent and Knowledge Based Systems – "Anytime" property (guarantees best answer that can be obtained within a given time if called upon for an answer in that amount of time), Explanation Capability (explains reasoning process in getting an answer).

55. - Human Interface and Interaction Systems – Adaptivity (to user's traits, interests), Intelligent Help, Display Salience.

56. - Information Systems – Ease of query, High recall (obtaining most relevant information), High Precision (not returning irrelevant information).

57. 3.5 Quality Attributes for Engineering Process

57. Other considerations of software systems are known to affect the software engineering process while the system is being built and during its future evolution or modification, and these can be considered elements of product quality. These software qualities include, but are not limited to:

58. - Code and object reusability

59. - Traceability of requirements from code and test documentation and to code and test documentation from requirements

60. - Modularity of code and independence of modules.

61. These software quality attributes and their subjective or objective measurement are important in the development process, particularly in large software projects. They can also be important in maintenance (if code is traceable to requirements – and vice/versa, then modification for new requirements is facilitated). They can improve the quality of the process and of future products (code that is designed to be reusable, if it functions well, avoids rewriting which could introduce defects).

Table 1. Software Quality Characteristics and Attributes – ISO 9126-1998 View

Characteristics & Subcharacteristics	**Short Description of the Characteristics and Subcharacteristics**
Functionality	**Characteristics relating to achievement of the basic purpose for which the software is being engineered**
. Suitability	The presence and appropriateness of a set of functions for specified tasks
. Accuracy	The provision of right or agreed results or effects
. Interoperability	Software's ability to interact with specified systems
. Security	Ability to prevent unauthorized access, whether accidental or deliberate, to programs and data.
. Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Reliability	**Characteristics relating to capability of software to maintain its level of performance under stated conditions for a stated period of time**
. Maturity	Attributes of software that bear on the frequency of failure by faults in the software
. Fault tolerance	Ability to maintain a specified level of performance in cases of software faults or unexpected inputs
. Recoverability	Capability and effort needed to reestablish level of performance and recover affected data after possible failure
. Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Usability	**Characteristics relating to the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users**
. Understandability	The effort required for a user to recognize the logical concept and its applicability
. Learnability	The effort required for a user to learn its application, operation, input, and output
. Operability	The ease of operation and control by users
. Attractiveness	The capability of the software to be attractive to the user
. Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Efficiency	**Characteristic related to the relationship between the level of performance of the software and the amount of resources used, under stated conditions**
. Time behavior	The speed of response and processing times and throughput rates inperforming its function
. Resource utilization	The amount of resources used and the duration of such use in performing its function
. Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Maintainability	**Characteristics related effort needed to make modifications, including corrections, improvements or adaptation of software to changes in environment, requirements and functional specifications**
. Analyzability	The effort needed for diagnosis of deficiencies or causes of failures, or for identification parts to be modified
. Changeability	The effort needed for modification fault removal or for environmental change
. Stability	The risk of unexpected effect of modifications
. Testability	The effort needed for validating the modified software
. Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Portability	**Characteristics related to the ability to transfer the software from one organization or hardware or software environment to another**
. Adaptability	The opportunity for its adaptation to different specified environments
. Installability	The effort needed to install the software in a specified environment
. Co-existence	The capability of a software product to co-exist with other independent software in common environment
. Replaceability	The opportunity and effort of using it in the place of other software in a particular environment
. Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols

97. 4. DEFINING SQA AND V&V

98. The KA on Software Requirements describes how the requirements and their individual features are defined, prioritized and documented and how the quality of that documentation can be measured. The set of requirements has a direct effect on the quality of other products, down to the delivered software. While the software engineering process builds quality into software products and prevents defects, the software engineering process also employs supporting processes to examine and assure software products for quality. The software engineering process and the many standards and models for Software Engineering Process are discussed in that KA of the SWEBOK. The supporting processes conduct activities to ensure that the software engineering process required by the project is followed. This section of the Software Quality KA addresses two of those supporting processes, SQA and V&V, which examine software through its development and maintenance. These processes detect defects and provide visibility to the management in determining how well the software carries out the documented requirements.

99. SQA and V&V provide management with visibility into the quality of the products at each stage in their development or maintenance. The visibility comes from the data and measurements produced through the performance of tasks to assess (examine and measure) the quality of the outputs of the software development and maintenance processes while they are developed.

100. The SQA process provides assurance that the software products and processes in the project life cycle conform to their specified requirements and adhere to their established plans. The SQA process is a planned systematic set of activities to help build quality into software from the beginning, that is, by ensuring that the problem is clearly and adequately stated and that the solution's requirements are properly defined and expressed. Then SQA retains the quality throughout the development and maintenance of the product by execution of a variety of activities. The SQA role with respect to process is to ensure that planned processes are appropriate and have been implemented according to their plans and that relevant measurements about processes are provided to the appropriate organization. Process and process improvement are discussed in both the Software Engineering Management and Software Engineering Process KAs.

101. The V&V process determines whether products of a given development or maintenance activity conform to the requirements of that activity and those imposed by previous activities, and whether the final software product (through its evolution) satisfies its intended use and user needs. Verification ensures that the product is built correctly, that is, verification determines that software products of an activity fulfill requirements imposed on them in the previous activities. Validation ensures that the right product is built, that is, the final product fulfills its specific intended use. The activities of validation begin early in the development or maintenance process, as do those of verification. V&V provides an examination of every product relative both to its immediate predecessor and to the system requirements it must satisfy.

102. Sometimes the terms SQA and V&V are associated with organizations rather than processes. SQA often is the name of a unit within an organization and sometimes an independent organization is contracted to conduct V&V. Testing may occur in BOTH SQA and V&V and is discussed in this KA in relation to those processes. Details on testing are found in the KA on Software Testing. The purpose of this KA is not to define organizations but rather the disciplines of SQA and V&V. Some discussion on organizational issues appears in [Hum98], and the IEEE Std. 1012.

103. First, to re-emphasize, many SQA and V&V evaluation techniques may be employed by the software engineers who are building the product. Second, the techniques may be conducted in varying degrees of independence from the development organization. Finally, the integrity level of the product may drive the degree of independence and the selection of techniques.

104. 5. PLANNING FOR SQA AND V&V

105. Planning for software quality involves planning, or defining, the required product along with its quality attributes and the processes to achieve the required product. Planning of these processes is discussed in other KAs: Software Engineering Management, Software Engineering Design, and Software Engineering Methods and Tools. These topics are different from planning the SQA and V&V processes. The SQA and V&V processes assess the implementation of those plans, that is,

how well software products satisfy customer requirements, provide value to the customers and users, and meet the quality requirements specified in the system requirements.

106. System requirements vary among systems as do the activities selected from the disciplines of SQA and V&V. Various factors influence planning, management and selection of activities and techniques. Some of these factors include, but are not limited to:

107. 1. the environment of the system in which the software will reside;

108. 2. system and software requirements;

109. 3. the commercial or standard components to be used in the system;

110. 4. the specific software standards used in developing the software;

111. 5. the software standards used for quality;

112. 6. the methods and software tools to be used for development and maintenance and for quality evaluation and improvement;

113. 7. the budget, staff, project organization, plans and schedule (size is inherently included) of all the processes;

114. 8. the intended users and use of the system, and

115. 9. the integrity level of the system.

116. Information from these factors influences how the SQA and V&V processes are planned organized, and documented, and the selection of specific SQA and V&V activities and needed resources as well as resources that impose bounds on the efforts. One factor, the integrity level of the system, needs some explanation. This level is determined from the possible consequences of failure of the system and the probability of failure. For software systems where safety or security is important, techniques such as hazard analysis for safety or threat analysis for security may be used in the planning process to help identify where potential trouble spots may be. This information would help in planning the activities. Failure history of similar systems may help in identifying which activities will be most useful in detecting faults and assessing quality.

117. 5.1 The SQA Plan

118. The SQA plan defines the processes and procedures that will be used to ensure that software developed for a specific product meets its requirements and is of the highest quality possible within project constraints. This plan may be governed by software quality assurance standards, life cycle standards, quality management standards and models, company policies and procedures for quality and quality improvement, and the management, development and maintenance plans for the software. Standards and models such as ISO9000, CMM, Baldrige, SPICE, TickIT influence the SQA plan and are addressed in Software Engineering Process.

119. The SQA plan defines the activities and tasks to be conducted, their management, and their schedule in relation to those in the software management, development or maintenance plans. The SQA plan may encompass Software Configuration Management and V&V or may call for separate plans for either of those processes. The SQA plan identifies documents, standards, practices, and conventions that govern the project and how they will be checked and monitored to ensure adequacy or compliance. The SQA plan identifies metrics, statistical techniques, procedures for problem reporting and corrective action, resources such as tools, techniques and methodologies, security for physical media, training, and SQA documentation to be retained. The SQA plan addresses assurance of any other type of function addressed in the software plans, such as supplier software to the project or commercial off-the-shelf software (COTS), installation, and service after delivery of the system.

120. 5.2 The V&V Plan

121. The V&V plan is the instrument to explain the requirements and management of V&V and the role of each technique in satisfying the objectives of V&V. An understanding of the different types of purposes of each verification and validation activity will help in planning carefully the techniques and resources needed to achieve their purposes. Verification activities examine a specific product, that is, output of a process, and provide objective evidence that specified requirements have been fulfilled. The “specified requirements” refer to the requirements of the examined product, relative to the product from which it is derived. For example, code is examined relative to requirements of a design description, or the software requirements are examined relative to system requirements.

122. Validation examines a specific product to provide objective evidence that the particular requirements for a specific intended use are

fulfilled. The validation confirms that the product traces back to the software system requirements and satisfies them. This includes planning for system test more or less in parallel with the system and software requirements process. This aspect of validation often serves as part of a requirements verification activity. While some communities separate completely verification from validation, the activities of each actually service the other.

123. V&V activities are exercised at every step of the life cycle, often on the same product, possibly using the same techniques in some instances. The difference is in the technique's objectives for that product, and the supporting inputs to that technique. Sequentially, verification and validation will provide evidence from requirements to the final system, a step at a time. This process holds true for any life cycle model, gradually iterating or incrementing through the development. The process holds in maintenance also.

124. The plan for V&V addresses the management, communication, policies and procedures of the V&V activities and their iteration, evaluation of methods and tools for the V&V activities, defect reports, and documentation requirements. The plan describes V&V activities, techniques and tools used to achieve the goals of those activities.

125. The V&V process may be conducted in various organizational arrangements. First, to re-emphasize, many V&V techniques may be employed by the software engineers who are building the product. Second, the V&V process may be conducted in varying degrees of independence from the development organization. Finally, the integrity level of the product may drive the degree of independence.

126. 6. ACTIVITIES AND TECHNIQUES FOR SQA AND V&V

127. The SQA and V&V processes consist of activities to indicate how software plans (e.g., management, development, configuration management) are being implemented and how well the evolving and final products are meeting their specified requirements. When these resources are formally organized, results from these activities are collected into reports for management before corrective actions are taken. The management of SQA and V&V are tasked with ensuring the quality of these reports, that is, that the results are accurate.

128. Specific techniques to support the activities software engineers perform to assure quality may depend upon their personal role (e.g., programmer, quality assurance staff) and project organization (e.g., test group, independent V&V). To build or analyze for quality, the software engineer understands development standards and methods and the genesis of other resources on the project (e.g., components, automated tool support) and how they will be used. The software engineer performing quality analysis activities is aware of and understands considerations affecting quality assurance: standards for software quality assurance, V&V, testing, the various resources that influence the product, techniques, and measurement (e.g., what to measure and how to evaluate the product from the measurements).

129. The SQA and V&V activities consist of many techniques; some may directly find defects and others may indicate where further examination may be valuable. These may be referred to as direct-defect finding and supporting techniques. Some often serve as both, such as people-intensive techniques like reviews, audits, and inspection and some static techniques like complexity analysis and control flow analysis. The SQA and V&V techniques can be categorized as two types: static and dynamic. Static techniques do not involve the execution of code, whereas dynamic techniques do. Static techniques involve examination of the documentation (e.g., requirements specification, design, plans, code, test documentation) by individuals or groups of individuals and sometimes with the aid of automated tools. Often, people tend to think of testing as the only dynamic technique, but simulation is an example of another one. Sometimes static techniques are used to support dynamic techniques, and vice-versa. An individual, perhaps with the use of a software tool, may perform some techniques; in others, several people are required to conduct the technique. Such techniques are "people-intensive". Depending on project size, others, such as testing, may involve many people, but are not people-intensive in the sense described here.

130. Static and dynamic techniques are used in either SQA or V&V. Their selection, specific objectives and organization depend on project and product requirements. Discussion in the following sections and the tables in the appendices provide only highlights about the various techniques; they are not inclusive. There

are too many techniques to define in this document but the lists and references provide a flavor of SQA and V&V techniques and will yield to the serious software engineer insights for selecting techniques and for pursuing additional reading about techniques.

131. 6.1 Static Techniques

132. Static techniques involve examination of the project's documentation, software and other information about the software products without executing them. The techniques may include activities that require two or more people ("people intensive") or analytic activities conducted by individuals, with or without the assistance of automated tools. These support both SQA and V&V processes and their specific implementation can serve the purpose of SQA, verification, or validation, at every stage of development or maintenance.

133. 6.1.1 Audits, Reviews, and Inspections

134. The setting for audits, reviews, inspections, and other people-intensive techniques may vary. The setting may be a formal meeting, an informal gathering, or a desk-check situation, but always two or more people are involved. Preparation ahead of time may be necessary. Resources in addition to the items under examination may include checklists and results from analytic techniques and testing. Another technique that may be included in this group is the walkthrough. These are activities are discussed throughout the IEEE Std. 1028 on reviews and audits, [Fre82], [Hor96], and [Jon91], [Rak97].

135. Reviews that specifically fall under the SQA process are technical reviews, that is, on technical products. However, the SQA organization may be asked to conduct management reviews as well. Persons involved in the reviews are usually a leader, a recorder, technical staff, and in the management review, management staff.

136. Management reviews determine adequacy of and monitor progress or inconsistencies against plans and schedules and requirements. These reviews may be exercised on products such as audit reports, progress reports, V&V reports and plans of many types including risk management, project management, software configuration management, software safety, risk management plans and risk assessment reports and others.

137. Technical reviews examine products such as software requirement specifications, software design documents, test documentation, user documentation, installation procedures but the coverage of the material may vary with purpose of the review. The subject of the review is not necessarily the completed product, but may be a portion at any stage of its development or maintenance. For example, a subset of the software requirements may be reviewed for a particular set of functionality, or several design modules may be reviewed, or separate reviews may be conducted for each category of test for each of its associated documents (plans, designs, cases and procedures, reports).

138. An audit is an independent evaluation of conformance of software products and processes to applicable regulations, standards, plans, and procedures. Audits may examine plans like recovery, SQA, and maintenance, design documentation. The audit is a formally organized activity, with participants having specific roles, such as lead auditor, other auditors, a recorder, an initiator, and a representative of the audited organization. While for reviews and audits there may be many formal names such as those identified in the IEEE Std. 1028, the important point is that they can occur on almost any product at any stage of the development or maintenance process.

139. Software inspections generally involve the author of a product, while reviews likely do not. Other persons include a reader, and the inspectors. The inspector team may consist of different expertise, such as domain expertise, or design method expertise, or language expertise, etc. Inspections are usually conducted on a relatively small section of the product. Often the inspection team may have had a few hours to prepare, perhaps by applying an analytic technique to a small section of the product, or to the entire product with a focus only on one aspect, e.g., interfaces. A checklist, with questions germane to the issues of interest, is a common tool used in inspections. Inspection sessions last a couple hours, whereas reviews and audits are usually broader in scope and take longer.

140. The walkthrough is similar to an inspection, but is conducted by only members of the development group, who examine a specific part of a product. With the exception of the walkthrough – primarily an assurance technique used only by the developer, these people-intensive techniques are traditionally considered to be SQA techniques, but may be performed by others. The technical objectives may also change,

depending on who performs them and whether they are conducted as verification or as validation activities. Often, when V&V is an organization, it may be asked to support these techniques, either by previous examination of the products or by attending the sessions to conduct the activities.

141. *6.1.2 Analytic Techniques*

142. An individual generally applies analytic techniques. Sometimes several people may be assigned the technique, but each applies it to different parts of the product. Some are tool-driven; others are primarily manual. With the References (Section 7.1) there are tables of techniques according to their primary purpose. However, many techniques listed as support may find some defects directly but are typically used as support to other techniques. Some howe ver are listed in both categories because they are used either way. The support group of techniques also includes various assessments as part of overall quality analysis.

143. Each type of analysis has a specific purpose and not all are going to be applied to every project. An example of a support technique is complexity analysis, useful for determining that the design or code may be too complex to develop correctly, to test or maintain; the results of a complexity analysis may be used in developing test cases. Some listed under direct defect finding, such as control flow analysis, may also be used as support to another activity. For a software system with many algorithms, then algorithm analysis is important, especially when an incorrect algorithm could cause a catastrophic result. There are too many analytic techniques to define in this document but the lists and references provide a flavor of software analysis and will yield to the serious software engineer insights for selecting techniques and for pursuing additional reading about techniques.

144. A class of analytic techniques that is gaining greater acceptance is the use of formal methods to verify software requirements and designs. Proof of correctness may also be applied to different parts of programs. Their acceptance to date has mostly been in verification of crucial parts of critical systems, such as specific security and safety requirements [NAS97].

145. 6.2 Dynamic Techniques

146. Different kinds of dynamic techniques are performed throughout the development and maintenance of software systems. Generally, these are testing techniques but techniques such as simulation and symbolic execution may be considered dynamic. Code reading is considered a static technique but experienced software engineers may execute the code as they read through it. In this sense, code reading may also fit under dynamic. This discrepancy in categorizing indicates that people with different roles in the organization may consider and apply these techniques differently.

147. Some testing may fall under the development process, the SQA process, or V&V, again depending on project organization. The discipline of V&V encompasses testing and requires activities for testing at the very beginning of the project. Because both the SQA and V&V plans address testing, this section includes some commentary about testing. The knowledge area on Software Testing provides discussion and technical references to theory, techniques for testing, and automation. Supporting techniques for testing fall under test management, planning and documentation. V&V testing generally includes component or module, integration, system, and acceptance testing. V&V testing may include test of commercial off-the-shelf software (COTS) and evaluation of tools to be used in the project

148. The assurance processes of SQA and V&V examine every output relative to the software requirement specification to ensure the output's traceability, consistency, completeness, correctness, and performance. This confirmation also includes exercising the output s of the development and maintenance processes, that is, the analysis consists of validating the code by testing to many objectives and strategies, and collecting, analyzing and measuring the results. SQA ensures that appropriate types of tests are planned, developed, and implemented, and V&V develops test plans, strategies, cases and procedures.

149. Two types of testing fall under SQA and V&V because of their responsibility for quality of materials used in the project:

150. ◆ Evaluation and test of tools to be used on the project

151. ◆ Conformance test (or review of conformance test) of components and COTS products to be used in the product.

152. The SWEBOK knowledge area on Software Testing addresses special purpose testing. Many of these types are also considered and performed

during planning for SQA or V&V testing. Occasionally the V&V process may be asked to perform these other testing activities according to the project's organization. Sometimes an independent V&V organization may be asked to monitor the test process and sometimes to witness the actual execution, to ensure that it is conducted in accordance with specified procedures. And, sometimes, V&V may be called on to evaluate the testing itself: adequacy of plans and procedures, and adequacy and accuracy of results.

153. Another type of testing that may fall under a V&V organization is third party testing. The third party is not the developer or in any way 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. Discussion on third party testing appears in the July/August 1999 *IEEE Software* special issue on software certification.

154. 7. MEASUREMENT APPLIED TO SQA AND V&V

155. SQA and V&V discover information about the quality of the software system at all stages of its development and maintenance and provide visibility into the software development and maintenance processes. Some of this information is about defects, where "defect" refers to errors, faults, and failures. Different cultures and standards may differ somewhat in their meaning for these same terms. Partial definitions taken from the IEEE Std 610.12-1990 ("IEEE Standard Glossary of Software Engineering Terminology") are these:

156. - Error: "A difference…between a computed result and the correct result"

157. - Fault: "An incorrect step, process, or data definition in a computer program"

158. - Failure: "The [incorrect] result of a fault"

159. - Mistake: "A human action that produces an incorrect result".

160. Mistakes (as defined above) are the subject of the quality improvement process, which is covered in the Knowledge Area Software Engineering Process. Failures found in testing as the result of software faults are included as defects in the discussion of this section.. "Failure" is the term used in reliability models, in which these models are built from failure data collected during system testing or from systems in service. These models are generally used to predict failure and to assist decisions on when to stop testing.

161. Many SQA and V&V techniques find inadequacies and defects, but information about these findings may be lost unless it is recorded. For some techniques (e.g., reviews, audits, inspections), recorders are usually present to record issues, decisions, and information about inadequacies and defects. When automated tools are used, the tool output may provide the defect information. For others And even for output of tools, data about defects are collected and recorded on some "trouble report" form and may further be entered into some type of database, either manually or automatically from an analysis tool. Reports about the defects are provided to the software management and development organizations.

162. One probable action resulting from SQA and V&V reports is to remove the defects from the product under examination. Other actions enable achieving full value from the findings of the SQA and V&V activities. These actions include analyzing and summarizing the findings with use of measurement techniques to improve the product and the process ands to track the defects and their removal. Process improvement is primarily discussed in Software Engineering Process, but some supporting information will be addressed here.

163. 7.1 Fundamentals of Measurement

164. Theories of measurement establish the foundation on which meaningful measurements can be made. Measuring implies classification and numbers, and various scales apply to different types of data. The four scales for measurement include nominal scale or a classification into exhaustive and mutually exclusive categories (e.g., boys, girls), ordinal scale (comparison in order, e.g., small, medium, large), interval scale (exact differences between two measurement points, e.g., addition and subtraction apply), and ratio scale (an absolute point can be located in the interval scale, and division, multiplication, addition and subtraction apply). An example in software is the number of defects. In module 1, there may be 10 defects per function point of SLOC, in module 2, 15 and in module 3, 20. The difference between module 1 and 2 is 5 and module 3 has twice as many defects as module 1. Theories of measurement and scales are discussed in [Kan94], pp. 54-82.

165. Measurement for measurement's sake does not help define the quality. Instead, the software engineer needs to define specific questions about the product, and hence the objectives to be met to answer those questions. Only then can specific measures be selected. Basili's paradigm on Goal-Question-Metric has been used since the early 80's and is used as a basis for some software measurement programs [Bas]. Another approach is "Plan-Do-Check-Act" discussed in Rakitin. Others are discussed in the references on software measurement. The point is that there has to be a reason for collecting data, that is, there is a question to be answered. Measurement programs are not arbitrary, but require planning and setting objectives according to some formalized procedure, as do other software engineering processes.

166. Other important measurement practices deal with experimentation and data collection. Experimentation is useful in determining the value of a development, maintenance, or assurance technique and results may be used to predict where faults may occur. Data collection is non-trivial and often too many types of data are collected. Instead, it is important to decide what is the purpose, that is, what question is to be answered from the data, then decide what data is needed to answer the question and then to collect only that data. While a measurement program has costs in time and money, it may result in savings. Methods exist to help estimate the costs of a measurement program. Discussion on the following key topics for measurement planning are found in ([Bas84], [Kan94], [Pr], [Pf], [Rak97], [Zel98]:

167. ◆ Experimentation

168. ◆ Selection of approach for measurement

169. ◆ Methods

170. ◆ Costing

171. ◆ Data Collection process.

172. 7.2 Metrics

173. Measurement models and frameworks for software quality enable the software engineer to establish specific product measures as part of the product concept. Models and frameworks for software quality are discussed in [Kan94], [Pf], and [Pr].

174. Data can be collected on various characteristics of software products. Many of the metrics are related to the quality characteristics defined in Section 2 of this Knowledge Area. Much of the data can be collected as results of the static techniques previously discussed and from various testing activities (see Software Testing Knowledge Area). The types of metrics for which data are collected fall into these categories and are discussed in [Jon91], [Lyu96], [Pf], [Pr], [Lyu96], and [Wei93]:

175. ◆ Quality characteristics measures

176. ◆ Reliability models & measures

177. ◆ Defect features (e.g., counts, density)

178. ◆ Customer satisfaction

179. ◆ Product features (e.g., size including SLOC, function points, number of requirements)

180. ◆ Structure metrics (e.g., modularity, complexity, control flow)

181. ◆ Object-oriented metrics.

182. 7.3 Measurement Techniques

183. While the metrics for quality characteristics and product features may be useful in themselves (for example, the number of defective requirements or the proportion of requirements that are defective), mathematical and graphical techniques can be applied to aid in interpretation of the metrics. These fit into the following categories and are discussed in [Fen97], [Jon91], [Kan94], [Lyu96] and [Mus98].

184. ◆ Statistically based (e.g., Pareto analysis, run charts, scatter plots, normal distribution)

185. ◆ Statistical tests (e.g., binomial test; chi-squared test)

186. ◆ Trend analysis

187. ◆ Prediction, e.g., reliability models.

188. The statistically 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 that the decision makers can use to focus resources where they appear most needed. Results from trend analysis may indicate whether a schedule may be slipped, such as in testing, or may indicate that certain classes of faults will gain in intensity unless some corrective action is taken in development. And the predictive techniques assist in planning test time and predicting failure. However, generally the decisions to be made from these techniques are not part of SQA and V&V. More discussion on these should appear in "Software Engineering Process" and "Software Engineering Management".

189. 7.4 Defect Characterization

190. SQA and V&V processes discover defects. Characterizing those defects enables understanding of the product, facilitates corrections to the process or the product, and informs the project management or customer of the status of the process or product. Many defect (fault) taxonomies exist and while attempts have been made to get consensus on a fault and failure taxonomy, the literature indicates that quite a few are in use (IEEE Std. 1044, [Bei90], [Chi92], [Gra92]).

191. As new design methodologies and languages evolve, along with advances in overall application technologies, new classes of defects appear, or, the connection to previously defined classes requires much effort to realize. When tracking defects, the software engineer is interested not only in the count of defects, but the types. Without some classification, information will not really be useful in identifying the underlying causes of the defects because no one will be able to group specific types of problems and make determinations about them. The point, again, as in selecting a measurement approach with quality characteristics, metrics and measurement techniques, is to establish a defect taxonomy that is meaningful to the organization and software system.

192. The above references as well as [Kan94], [Fen95] and [Pf], and [Jon89] all provide discussions on analyzing defects, that is, measuring their occurrences and then applying statistical methods to understand the types of defects that occur most frequently, that is, where do mistakes occur (their density), to understand the trends and how well detection techniques are working, and, how well the development and maintenance processes are doing.[2] Measuring test coverage helps to estimate how much test effort remains and to predict possible remaining defects. From these measurement methods, one can develop defect profiles for a specific application domain. Then, for the next software system within that organization, the profiles can be used to guide the SQA and V&V processes, that is, to expend the effort where the problems are likeliest 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.

193. The following topics are useful for establishing measurement approaches for the software products:

194. - Defect classification and descriptions

195. - Defect analysis

196. - Measuring adequacy of the SQA and V&V activities

197. - Test coverage

198. - Benchmarks, profiles, baselines, defect densities.

199. 7.5 Additional uses of SQA and V&V data

200. The measurement section of this KA on SQA and V&V touches only minimally on measurement, for measurement is a major topic itself. The purpose here is only to provide some insight on how the SQA and V&V processes use measurement directly to support achieving their goals. There are a few more topics which measurement of results from SQA and V&V may support. These include some assistance in deciding when to stop testing. Reliability models and benchmarks, both using fault and failure data, are useful for this objective. Again, finding a defect, or perhaps trends among the defects, may help to locate the source of the problem.

201. The cost of SQA and V&V processes is almost always an issue raised in deciding how to organize a project. Often generic models of cost, based on when the defect is found and how much effort it takes to fix the defect relative to finding the defect earlier, are used. Data within an organization from that organization's projects may give a better picture of cost for that organization. Discussion on this topic may be found in [Rak97], pp. 39-50.

202. Finally, the SQA and V&V reports themselves provide valuable information not only to these processes but to all the other software engineering processes for use in determining how to improve them. Discussions on these topics are found in [McC93] and IEEE Std. 1012.

[2] Discussion on using data from SQA and V&V to improve development and maintenance processes appears in Software Engineering Management and Software Engineering Process.

203 8. REFERENCES

204. 8.1 References Keyed to Text Topics

205.	**Software Quality Concepts**	[Boe78]	[D]	[Fen97]	[Kia95]	[Lap91]	[Lew92]	[Lyu96]	[M]	[Mus98]	[Pf]	[Pr]	[Rak97]	[S]	[Wal96]	[Wei96]
206.	Value of Quality	X									X					X
207.	Functionality												X			
208.	Reliability							X	X	X	X	X	X	X	X	
209.	Efficiency							X	X			X				
210.	Usability			X			X		X		X	X	X	X		
211.	Maintainability			X	X		X		X		X	X		X		
212.	Portability								X		X	X	X	X		
213.	Dependability			X		X		X	X	X	X	X		X	X	
214.	Other Qualities		X				X		X		X	X		X	X	

215.	**Definition & Planning for Quality**	[Gra92]	[Hor96]	[Kaz99]	[Lew92]	[Lyu96]	[McC93]	[M]	[Mus98]	[Pf]	[Pr]	[Rak97]	[Sch98]	[S]	[Wal89]	[Wal96]
216.	Overall							X		X	X			X		
217.	SQA		X		X		X	X		X	X	X	X	X		
218.	VV				X			X		X	X	X	X	X	X	X
219.	Independent V&V				X					X	X	X		X	X	X
220.	Hazard, threat anal.							X		X	X			X		X
221.	Risk assessment	X	X		X	X		X	X					X		
222.	Performance analysis			X						X	X					

223.	**Techniques Requiring Two or More People**	[Ack97]	[Ebe94]	[Fre82]	[Gra92]	[Hor96]	[Lew92]	[McC96]	[Pf]	[Pr]	[Rak97]	[Sch98]	[S]	[Wal89]	[Wal96]
224.	Audit			X		X	X				X			X	
225.	Inspection	X	X	X	X	X		X	X	X	X	X	X	X	X
226.	Review			X		X	X	X	X	X			X	X	X
227.	Walkthrough			X		X		X	X	X			X	X	X

228. **Support to Other Techniques**	[Bei90]	[Con86]	[Fri95]	[Het84]	[Lev95]	[Lew92]	[Lyu96]	[Mus98]	[Pf]	[Pr]	[Rak97]	[Rub94]	[S]	[Fri95]	[Wal89]	[Wal96]
229. Change Impact Anal.							X		X	X			X			
230. Checklists				X		X					X	X				
231. Complexity Analysis	X	X					X	X		X						
232. Coverage Analysis	X						X	X								
233. Consistency Analysis									X	X		X				
234. Criticality Analysis					X	X				X						X
235. Hazard Analysis			X		X					X	X		X			
236. Sensitivity Analysis			X											X		
237. Slicing	X													X		X
238. Test documents	X	X					X	X							X	X
239. Tool evaluation						X	X								X	
240. Traceability Analysis						X	X		X				X		X	X
241. Threat Analysis			X				X		X	X			X			

242. **Testing Special to SQA or V&V**	[Fri95]	[Lev95]	[Lyu96]	[Mus98]	[Pf]	[Pr]	[Rak97]	[Rub94]	[Sch98]	[S]	[Voa99]	[Wak99]	[Wal89]
243. Conformance Test.	X											X	
244. Configuration Test.						X							
245. Certification Testing			X	X		X	X		X		X	X	
246. Reliability Testing	X	X	X	X					X				
247. Safety Testing	X		X	X					X				
248. Security Testing					X								
249. Statistical Testing			X	X	X	X			X	X			
250. Usability Testing					X			X					
251. Test Monitoring													X
252. Test Witnessing													X

253.	Defect Finding **Techniques**	[Bei90]	[Fen96]	[Fri95]	Herzel	[Hor96]	[Ipp95]	[Lev95]	[Lew92]	[Lyu96]	Moore	[Mus98]	[Pf]	[Pr]	[Rak97]	[Rub94]	[Sch98]	[S]	[Wak99]	[Wal89]
254.	Algorithm Analysis		X		X				X										X	X
255.	Boundary Value Anal.			X									X	X				X	X	X
256.	Change Impact Anal.									X			X	X	X		X	X		
257.	Checklists					X									X					
258.	Consistency Analysis							X						X			X			
259.	Control Flow Analysis	X	X						X	X			X	X					X	X
260.	Database Analysis	X	X	X					X							X			X	X
261.	Data Flow Analysis	X	X	X					X	X	X					X			X	X
262.	Distrib. Arch. Assess.													X						
263.	Evaluation of Docts.: Concept, Reqmts.			X					X	X						X			X	X
264.	Evaluation of Docts.: Design, Code, Test			X					X	X						X			X	
265.	Evaluation of Doc.: User, Installation			X					X	X						X			X	
266.	Event Tree Analysis			X																X
267.	Fault Tree Analysis			X			X			X	X					X				
268.	Graphical Analysis	X	X									X							X	
269.	Hazard Analysis		X	X			X	X		X	X					X				
270.	Interface Analysis	X		X		X			X	X						X				X
271.	Formal Proofs			X						X	X					X				X
272.	Mutation Analysis			X						X									X	X
273.	Perform. Monitoring									X										X
274.	Prototyping			X						X	X					X				X
275.	Reading			X																X
276.	Regression Analysis			X		X			X	X									X	X
277.	Simulation			X																X
278.	Sizing & Timing Anal.			X					X	X	X								X	X
279.	Threat Analysis									X	X					X				

280.	**Measurement in Software Quality Analysis**	[Bas84]	[Bei90]	[Con86]	[Chi96]	[Fen95]	[Fen97]	[Fri95]	[Gra92]	[Het84]	[Hor96]	[Jon91]	[Kan94]	[Mus89]	[Lew92]	[Lyu96]	[Mus98]	[Pen92]	[Pf]	[Pr]	[McC93]	[Rak97]	[Sch98]	[S]	[Wak99]	[Wei93]	[Zel98]
281.	Benchmarks, profiles, etc.		X				X												X		X		X				
282.	Company Metric Progs.					X	X		X				X						X	X							
283.	Costing		X	X					X		X				X		X		X		X	X	X	X		X	
284.	Customer satisfaction											X	X							X							
285.	Data Collection process	X		X		X	X		X			X															
286.	Debugging		X	X			X									X					X			X			
287.	Defect Analysis		X	X	X			X	X	X	X		X			X	X	X	X	X	X						
288.	Defect Classif. and Descr.		X		X		X	X	X		X		X		X	X	X		X	X							
289.	Defect Features			X	X		X		X		X		X			X						X					
290.	Example of applied GQM						X		X																		
291.	Experimentation:			X	X	X	X										X										X
292.	Framework					X	X																				
293.	GQM	X				X	X		X				X											X			
294.	Methods			X		X			X				X			X	X			X							
295.	Metrics			X			X		X		X		X			X		X	X	X		X	X	X			
296.	Models					X	X									X	X										
297.	Prediction						X						X			X	X					X					
298.	Prod. features: O/O Metr.																						X				
299.	Prod. Features: Structure			X		X	X		X							X						X					
300.	Product features: Size			X			X		X				X			X											
301.	Quality Attributes														X				X	X				X			
302.	Quality Character. Meas.						X								X			X				X					
303.	Reliab. Models & Meas.			X			X	X					X			X	X				X	X					
304.	Scales			X		X	X						X														
305.	SQA & V&V reports *										X						X					X			X		
306.	Statistical tests			X			X									X		X	X					X			
307.	Statistical Analysis & measurement			X		X	X				X		X			X	X	X				X					
308.	Test coverage																X					X					
309.	Theory			X		X	X						X														
310.	Trend analysis															X											
311.	When to stop testing*							X						X			X										

312.*See also [Musa89]

313.	**Standards**	Quality Requirements & planning	Reviews/ Audits	SQA/ V&V planning	Safety/ security analysis, tests	Documentation of quality analysis	Measurement
314.	ISO 9000	X	X			X	X
315.	ISO 9126	X					
316.	IEC 61508	X					X
317.	ISO/IEC 14598				X	X	X
318.	ISO/IEC 15026	X					
319.	ISO FDIS 15408	X			X		
320.	FIPS 140-1	X			X		
321.	IEEE 730		X	X		X	
322.	IEEE 1008			X			
323.	IEEE 1012		X	X	X	X	
324.	IEEE 1028		X				
325.	IEEE 1228				X		
326.	IEEE 829					X	
327.	IEEE 982.1,.2						X
328.	IEEE 1044						X
329.	IEEE 1061						X

330. 8.2 Reference Lists

331. *8.2.1 Basic SWEBOK References*

332. Dorfman, M., and R. H. Thayer, *Software Engineering*. IEEE Computer Society Press, 1997. **[D]**

333. Moore, J. W.,*Software Engineering Standards: A User's Road Map*. IEEE Computer Society Press, 1998. **[M]**

334. Pfleeger, S. L., *Software Engineering – Theory and Practice*. Prentice Hall, 1998. **[Pf]**

335. Pressman, R. S., *Software Engineering: A Practitioner's Approach* (4th edition). McGraw-Hill, 1997. **[Pr]**

336. Sommerville, I., *Software Engineering* (5th edition). Addison-Wesley, 1996. **[S]**

337. Vincenti, W. G., What Engineers Know and How They Know It – Analytical Studies form Aeronautical History. Baltimore and London: John Hopkins, 1990. **[V]**

338. *8.2.2 Core References*

339. (N.B. Some of these will be removed in the next version, but we are checking to make sure that every topic is adequately covered before we do so. See also the note at 8.2.3)

340. Abran, A.; Robillard, P.N. , Function Points Analysis: An Empirical Study of its Measurement Processes, in IEEE Transactions on Software Engineering, vol. 22, 1996, pp. 895-909. [Abr96].

341. Ackerman, Frank A., "Software Inspections and the Cost Effective Production of Reliable Software," in [D] pp. 235-255. [Ack97]

342. Basili, Victor R. and David M. Weiss, A Methodology for Collecting Valid Software Engineering Data, IEEE Transactions on Software Engineering, pp. 728-738, November 1984. [Bas84]

343. Beizer, Boris, *Software Testing Techniques,* International Thomson Press, 1990. [Bei90]

344. Boehm, B.W. et al., *Characteristics of Software Quality"*, TRW series on Software Technologies, Vol. 1, North Holland, 1978. [Boe78]

345. Chilllarege, Ram, Chap. 9, pp359-400, in [Lyu96]. [Chi96]

346. Conte, S.D., et al, *Software Engineering Metrics and Models,* The Benjamin / Cummings Publishing Company, Inc., 1986. [Con86]

347. Ebenau, Robert G., and Susan Strauss, *Software Inspection Process*, McGraw-Hill, 1994. [Ebe94]

348. Fenton, Norman E., *Software Metrics,* International Thomson Computer Press, 1995. [Fen95]

349. Fenton, Norman E., and Shari Lawrence Pfleeger, Software Metrics, International Thomson Computer Press, 1997. [Fen97]

350. Freedman, Daniel P., and Gerald M. Weinberg, Handbook of Walkthroughs, Inspections, and Technical Reviews, Little, Brown and Company, 1982. [Fre82]

351. Friedman, Michael A., and Jeffrey M. Voas, *Software Assessment*, John Wiley & Sons, Inc., 1995. [Fri95]

352. Grady, Robert B, *Practical Software Metrics for project Management and Process Management,* Prentice Hall, Englewood Cliffs, NJ 07632, 1992. [Gra92]

353. Hetzel, William, *The Complete Guide to Software Testing*, QED Information Sciences, Inc., 1984, pp177-197. [Het84]

354. Horch, John W., *Practical Guide to Software Quality Management,* Artech-House Publishers, 1996. [Hor96]

355. Humphrey, Watts S., Managing the Software Process, Addison Wesley, 1989 Chapters 8, 10, 16. [Hum89]

356. Ince, Darrel, *ISO 9001 and Software Quality Assurance*, McGraw-Hill, 1994. [Inc94]

357. Ippolito, Laura M. and Dolores R. Wallace, NISTIR 5589, A Study on Hazard Analysis in High Integrity Software Standards and Guidelines,@ U.S. Department. of Commerce, Technology Administration, National Institute of Standards and Tech., Jan 1995. http://hissa.nist.gov/HAZARD/ [Ipp95]

358. Jones, Capers, Applied Software Measurement, McGraw-Hill, Inc., 1991; (Chap. 4: Mechanics of Measurement; Chapter 5: User Satisfaction). [Jon91]

359. Kan, Stephen, H., *Metrics and Models in Software Quality Engineering,* Addison-Wesley Publishing Co., 1994. [Kan94]

360. Kazman, R., M. Barbacci, M. Klein, S. J. Carriere, S. G. Woods, Experience with Performing Architecture Tradeoff Analysis, *Proceedings of ICSE 21,* (Los Angeles, CA), IEEE Computer Society, May 1999, 54-63. [Kaz99]

361. Kiang, David, Harmonization of International Software Standards on Integrity and Dependability, *Proc. IEEE International Software Engineering Standards Symposium*, IEEE Computer Society Press, Los Alamitos, CA, 1995, pp. 98-104. [Kia95]

362. Laprie, J.C., *Dependability: Basic Concepts and Terminology, IFIP WG 10.4,* Springer-Verlag, New York 1991. [Lap91]

363. Leveson, Nancy, *SAFEWARE: system safety and requirements,* Addison-Wesley, 1995. [Lev95]

364. Lewis, Robert O., *Independent Verification and Validation*, John Wiley & Sons, Inc., 1992. [Lew92]

365. Lyu , Michael R., *Handbook of Software Reliability Engineering*, McGraw Hill, 1996. [Lyu96]

366. McConnell, Steven C., *Code Complete: a practical handbook of software construction,* Microsoft Press, 1993. [McC93]

367. Musa, John D., and A. Frank Ackerman, "Quantifying Software Validation: When to stop testing?" *IEEE Software*, May 1989, 31-38. [Mus89]

368. Musa, John, *Software Reliability Engineering,* McGraw Hill, 1998. [Mus98]

369. NASA, *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion,* [NASA-GB-001-97], 1997, http://eis.jpl.nasa.gov/quality/Formal_Methods/. [NAS97]

370. Palmer, James D., "Traceability," In: [Dorf], pp. 266-276. [Pal97]

371. Peng, Wendy W. and Dolores R. Wallace, "Software Error Analysis," NIST SP 500-209, National Institute of Standards and Technology, Gaithersburg, MD 20899, December 1992.] http://hissa.nist.gov/SWERROR/. [Pen92]

372. Rakitin, Steven R., *Software Verification and Validation, A Practitioner's Guide,* Artech House, Inc., 1997. [Rak97]

373. Rosenberg, Linda, Applying and Interpreting Object-Oriented Metrics, Software Tech. Conf. 1998,

http://satc.gsfc.nasa.gov/support/index.html. [Ros98]

374. Rubin, Jeffrey, *Handbook of Usability Testing*, JohnWiley & Sons, 1994. [Rub94]

375. Schulmeyer, Gordon C., and James I. McManus, *Handbook of Software Quality Assurance*, Third Edition, Prentice Hall, NJ, 1998. [Sch98]

376. Voas, Jeffrey, "Certifying Software For High Assurance Environments, " *IEEE Software*, July-August, 1999, pp. 48-54. [Voa99]

377. Wakid, Shukri, D. Richard Kuhn, and Dolores R. Wallace, "Software Measurement: Testing and Certification," *IEEE Software*, July-August 1999, 39-47. [Wak99]

378. Wallace, Dolores R., and Roger U. Fujii, "Software Verification and Validation: An Overview," *IEEE Software*, May 1989, 10-17 . [Wal89]

379. Wallace, Dolores R., Laura Ippolito, and Barbara Cuthill, Reference Information for the Software Verification and Validation Process,@ NIST SP 500-234, NIST, Gaithersburg, MD 20899, April, 1996. http://hissa.nist.gov/VV234/. [Wal96]

380. Weinberg, Gerald M., Quality Software Management, Vol 2: First-Order Measurement, Dorset House, 1993. (Ch. 8, Measuring Cost and Value). [Wei93]

381. Zelkowitz, Marvin V. and Dolores R. Wallace, Experimental Models for Validating Computer Technology, *Computer*, Vol. 31 No.5, 1998 pp.23-31. [Zel98]

382. 8.2.3 Additional Readings

383. (Note: A portion of the sources now in 8.2.2 will be included here, but we are still checking carefully to be sure that we do not remove anything vital. We will also add other references that were not included in 8.2.2, and pages not selected in references of 8.2.2 from the sources therein.)

384. 8.2.4 Relevant Standards

385. FIPS 140-1, 1994, Security Requirements for Cryptographic Modules

386. IEC 61508 Functional Safety - Safety -related Systems Parts 1,2

387. IEEE 610.12-1990, Standard Glossary of Software Engineering Terminology

388. IEEE 730-1998 Software Quality Assurance Plans

389. IEEE 829 -1998 Software Test Documentation

390. IEEE Std 982.1 and 982.2 Standard Dictionary of Measures to Produce Reliable Software

391. IEEE 1008-1987 Software Unit Test

392. IEEE 1012-1998 Software Verification and Validation

393. IEEE 1028 -1997 Software Reviews

394. IEEE 1044 -1993 Standard Classification for Software Anomalies

395. IEEE Std 1061-1992 Standard for A Software Quality Metrics Methodology

396. IEEE Std 1228-1994 Software Safety Plans

397. ISO 8402-1986 Quality - Vocabulary

398. ISO 9000-1994 Quality Management and Quality Assurance Standards

399. ISO 9001-1994 Quality Systems

400. ISOIEC 9126-1999: Software Product Quality

401. ISO 12207 Software Life Cycle Processes 1995

402. ISO/IEC 14598-1998: Software Product Evaluation

403. ISO/IEC 15026:1998, Information technology -- System and software integrity levels.

404. The Common Criteria for Information Technology Security Evaluation (CC) VERSION 2.0 / ISO FDIS 15408

APPENDIX A

KNOWLEDGE AREA DESCRIPTION SPECIFICATIONS FOR THE STONE MAN VERSION OF THE GUIDE TO THE SOFTWARE ENGINEERING BODY OF KNOWLEDGE

Pierre Bourque, Robert Dupuis and Alain Abran
Université du Québec à Montréal

James W. Moore
The MITRE Corporation

Leonard Tripp
1999 President IEEE Computer Society

1. INTRODUCTION[1]

2. This document presents a third interim version (version 0.7) of the specifications provided by the Editorial Team to the Knowledge Area Specialist regarding the Knowledge Area Descriptions of the Guide to the Software Engineering Body of Knowledge (Stone Man Version). The Editorial Team definitely views the development of these specifications as an iterative process and strongly encourages comments, suggested improvements and feedback on these specifications from all involved.

3. This set of specifications may of course be improved through feedback obtained from the next review cycle of the Guide scheduled for this spring.

4. This document begins by presenting specifications on the contents of the Knowledge Area Description. Criteria and requirements are defined for proposed breakdowns of topics, for the rationale underlying these breakdowns and the succinct description of topics, for the rating of these topics according to Bloom's taxonomy, for selecting reference materials, and for identifying relevant Knowledge Areas of Related Disciplines. Important input documents are also identified and their role within the project is explained. Non-content issues such as submission format and style guidelines are also discussed in the document.

5. CONTENT GUIDELINES

6. The following guidelines are presented in a schematic form in the figure found below. While all components are part of the Knowledge Area Description, it must be made very clear that some components are essential, while other are not. The breakdown(s) of topics, the selected reference material and the matrix of reference material versus topics are essential. Without them there is no Knowledge Area Description. The other components could be produced by other means if, for whatever reason, the Specialist cannot provide them within the given timeframe and should not be viewed as major stumbling blocks.

7. Criteria and requirements for proposing the breakdown(s) of topics within a Knowledge Area

8. The following requirements and criteria should be used when proposing a breakdown of topics within a given Knowledge Area:

9. a) Knowledge Area Specialists are expected to propose one or possibly two complementary breakdowns that are specific to their Knowledge Area. The topics found in all breakdowns within a given Knowledge Area must be identical.

[1] Text in bold indicates changes between version 0.25 of this document and version 0.7.

10. b) These breakdowns of topics are expected to be "reasonable", not "perfect". The Guide to the Software Engineering Body of Knowledge is definitely viewed as a multi-phase effort and many iterations within each phase as well as multiple phases will be necessary to continuously improve these breakdowns. At least for the Stone Man version, "soundness and reasonableness" are being sought after, not "perfection".

11. c) The proposed breakdown of topics within a Knowledge Area must decompose the subset of the Software Engineering Body of Knowledge that is "generally accepted". See section found below for a more detailed discussion on this.

12. d) The proposed breakdown of topics within a Knowledge Area 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.

13. e) The proposed breakdown of topics must, as much as possible, be compatible with the various schools of thought within software engineering.

14. f) The proposed breakdown of topics within Knowledge Areas must be compatible with the breakdown of software engineering generally found in industry and in the software engineering literature and standards.

15. g) The proposed breakdown of topics is expected to be as inclusive as possible. It is deemed better to suggest too many topics and have them be abandoned later than the reverse.

16. h) The Knowledge Area Specialist are expected to adopt 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 quality (in general) and measurement.

17. Please note that the issue of how to properly handle these "cross-running" or "orthogonal topics" and whether or not they should be handled in a different manner has not been completely resolved yet.

18. i) The proposed breakdowns 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 Knowledge Area, Knowledge Area Specialists are expected to propose a reasonable and manageable number of topics per Knowledge Area. Emphasis should also be put on the selection of the topics themselves rather than on their organization in an appropriate hierarchy.

19. j) Proposed topic names must be significant enough to be meaningful even when cited outside the Guide to the Software Engineering Body of Knowledge.

20. k) Knowledge Area Specialists are also expected to propose a breakdown of topics based on the categories of engineering design knowledge defined in Chapter 7 of Vincenti's book. This exercise should be regarded by the Knowledge Area specialists as a tool for viewing the proposed topics in an alternate manner and for linking software engineering itself to engineering in general. Please note that effort should not be spent on this classification at the expense of the three essential components of the Knowledge Area Description. **(Please note that the classification of the topics as per the categories of engineering design knowledge has been produced but will be published at a latter date in a separate working document. Please contact the editorial team for more information).**

21. Criteria and requirements for describing topics and for describing the rationale underlying the proposed breakdown(s) within the Knowledge Area

22. a) Topics need only to be sufficiently described so the reader can select the

appropriate reference material according to his/her needs.

23. b) Knowledge Area Specialists are expected to provide a text describing the rationale underlying the proposed breakdown(s).

24. Criteria and requirements for rating topics according to Bloom's taxonomy

25. a) Knowledge Area Specialists are expected to provide an Appendix that states for each topic at which level of Bloom's taxonomy a "graduate plus four years experience" should "master" this topic. This is seen by the Editorial Team as a tool for the Knowledge Area Specialists to ensure that the proposed material meets the criteria of being "generally accepted". Additionally, the Editorial Team views this as a means of ensuring that the Guide to the Software Engineering Body of Knowledge is properly suited for the educators that will design curricula and/or teaching material based on the Guide and licensing/certification officials defining exam contents and criteria.

26. Please note that these appendices will all be combined together and published as an Appendix to the Guide to the Software Engineering Body of Knowledge.

27. Criteria and Requirements for selecting Reference Material

28. a) Specific reference material must be identified for each topic. Each reference material can of course cover multiple topics.

29. b) Proposed Reference Material can be book chapters, refereed journal papers, refereed conference papers or refereed technical or industrial reports or any other type of recognized artifact such as web documents. They must be generally available and must not be confidential in nature. **Please be as precise as possible by identifying what specific chapter or section is relevant.**

30. c) Proposed Reference Material must be in English.

31. d) A reasonable amount of reference material must be selected for each Knowledge Area. The following guidelines should be used in determining how much is reasonable:

32. ◆ If the 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 Knowledge Area description), an average target for the number of pages would be 500. 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 material.

33. ◆ The amount of reference material would be reasonable if it consisted of the study material on this Knowledge Area of a software engineering licensing exam that a graduate would pass after completing four years of work experience.

34. ◆ The Guide to the Software Engineering Body of Knowledge is intended by definition to be selective in its choice of topics and associated reference material The list of reference material for each Knowledge Area should be viewed and will be presented as an "informed and reasonable selection" rather than as a definitive list.

35. ◆ The classification of topics according to Bloom's taxonomy should be used to allot the appropriate amount and level of depth of the reference material selected for each topic.

36. ◆ Additional reference material can be included in a "Further Readings" list. These further readings still must be related to the topics in the breakdown. They must also discuss generally accepted knowledge. However, the further readings material will not be made available on the web nor should there be a matrix between the reference material listed in Further Readings and the individual topics.

37. e) If deemed feasible and cost-effective by the IEEE Computer Society, selected reference material will be published on the Guide to the Software Engineering Body of

Knowledge web site. To facilitate this task, preference should be given to reference material for which the copyrights already belong to the IEEE Computer Society or the ACM. This should however not be seen as a constraint or an obligation.

38. f) A matrix of reference material versus topics must be provided.

39. Criteria and Requirements for identifying Knowledge Areas of the Related Disciplines

40. a) Knowledge Area Specialists are expected to identify in a separate section which Knowledge Areas of the Related Disciplines that are sufficiently relevant to the Software Engineering Knowledge Area that has been assigned to them be expected knowledge by a graduate plus four years of experience.

41. This information will be particularly useful to and will engage much dialogue between the Guide to the Software Engineering Body of Knowledge initiative and our sister initiatives responsible for defining a common software engineering curricula and standard performance norms for software engineers.

42. The list of Knowledge Areas of Related Disciplines can be found in the Proposed Baseline List of Related Disciplines. If deemed necessary and if accompanied by a justification, Knowledge Area Specialists can also propose additional Related Disciplines not already included or identified in the Proposed Baseline List of Related Disciplines.

43. Common Table of Contents

44. a) Knowledge Area descriptions should use the following table of contents:

45. ◆ Table of contents

46. ◆ Introduction

47. ◆ Definition of the Knowledge Area

48. ◆ Breakdown of topics of the Knowledge Area (for clarity purposes, we believe this section should be placed in front and not in an appendix at the end of the document. Also, it should be accompanied by a figure describing the breakdown)

49. ◆ Breakdown rationale

50. ◆ Matrix of topics vs. Reference material

51. ◆ Recommended references for the Knowledge Area being described (please do not mix them with references used to write the Knowledge Area description)

52. ◆ List of Further Readings

53. ◆ References used to write and justify the Knowledge Area description.

54. What do we mean by “generally accepted knowledge”?

55. 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 Guide to the Software Engineering Body of Knowledge seeks to identify and describe that subset of the body of knowledge that is generally accepted or, in other words, the core body of knowledge. To better illustrate what “generally accepted knowledge” is relative to other types of knowledge, Figure 1 proposes a draft three-category schema for classifying knowledge.

56. The Project Management Institute in its Guide to the Project Management Body of Knowledge[2] defines “generally accepted” knowledge for project management in the following manner:

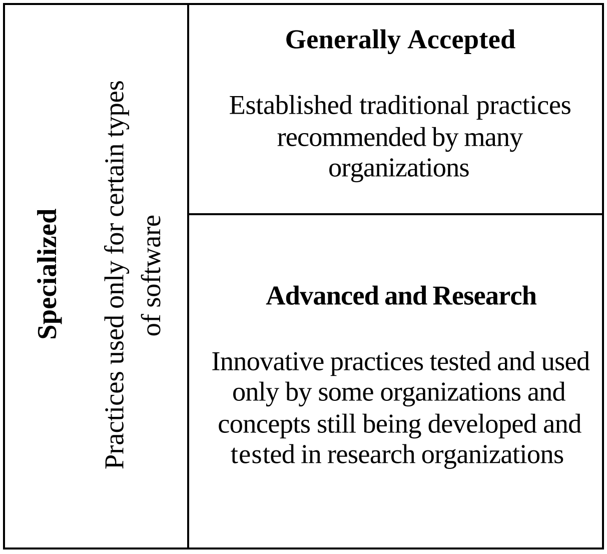
57. ‘“Generally accepted” means that the knowledge and practices described are applicable to most projects most of the time, and that there is widespread consensus about their value and usefulness. “Generally accepted” does not mean that the knowledge and practices described are or should be applied uniformly on all projects; the project management team is always responsible for determining what is appropriate for any given project.’

[2] See [1] W. R. Duncan, “A Guide to the Project Management Body of Knowledge,” Project Management Institute, Upper Darby, PA 1996. Can be downloaded from www.pmi.org

58. The Guide to the Project Management Body of Knowledge is now an IEEE Standard.

59. At the Mont-Tremblant kick off meeting, the Industrial Advisory Board better defined “generally accepted” as knowledge to be included in the study material of a software engineering licensing exam that a graduate would pass after completing four years of work experience. These two definitions should be seen as complementary.

60. Knowledge Area Specialists are also expected to be somewhat forward looking in their interpretation by taking into consideration not only what is “generally accepted” today and but what they expect will be “generally accepted” in a 3 to 5 years timeframe.



Specialized

Practices used only for certain types of software

Generally Accepted

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

61. Figure 1 Categories of knowledge

62. Length of Knowledge Area Description

1. 63. Knowledge Area Descriptions are currently expected to be roughly in the 10 pages range using the format of the International Conference on Software Engineering format as defined below. This includes text, references, appendices and tables etc. This, of course, does not include the reference materials themselves. This limit should, however, not be seen as a constraint or an obligation.

64. Role of Editorial Team

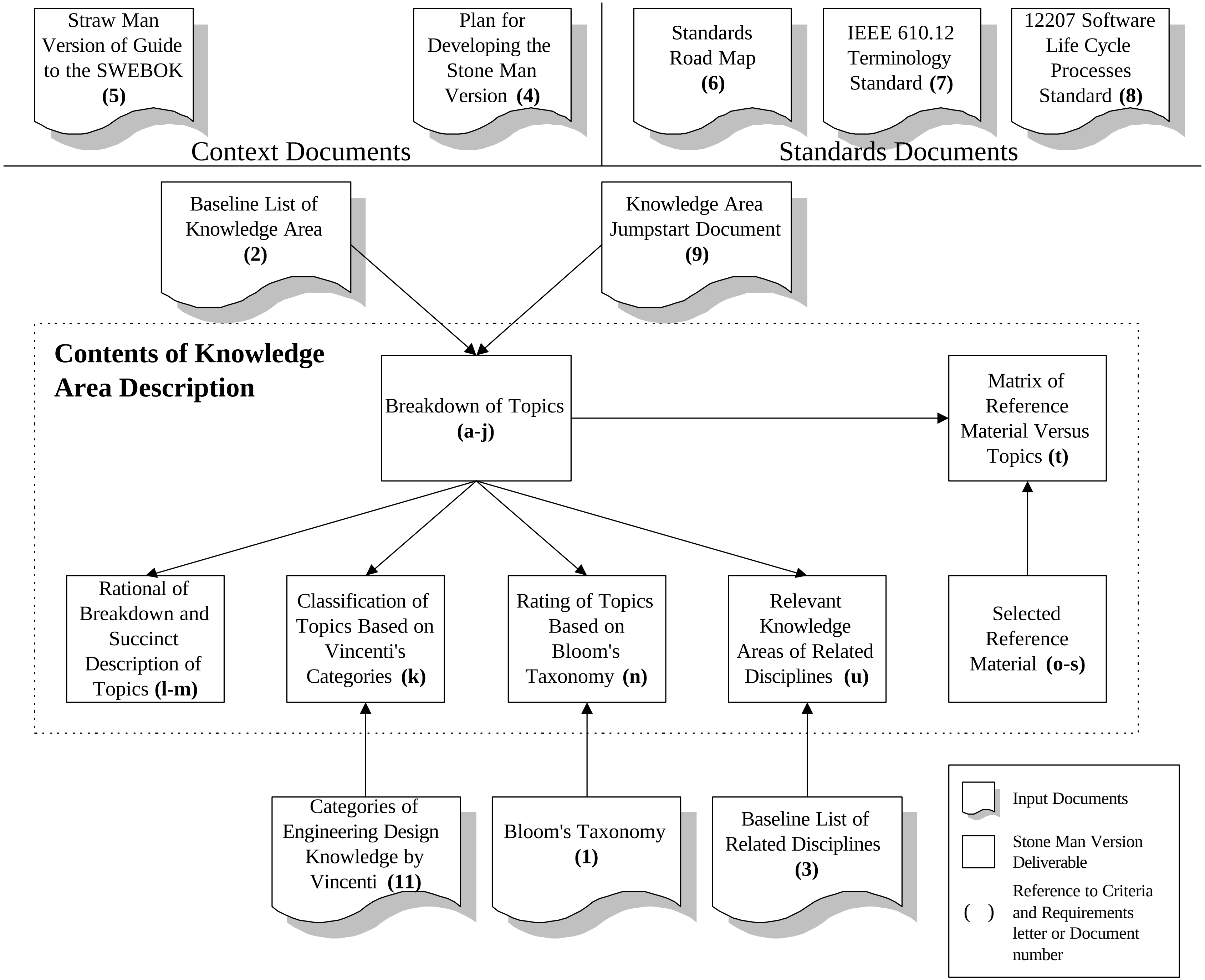
65. Alain Abran and James W. Moore are the Executive Editors and are responsible for maintaining good relations with the IEEE CS, the ACM, the Industrial Advisory Board and the Panel of Experts as well as for the overall strategy, approach, organization and funding of the project.

66. Pierre Bourque and Robert Dupuis are the Editors and are responsible for the coordination, operation and logistics of this project. More specifically, the Editors are responsible for developing the project plan, the Knowledge Area description specification and for coordinating Knowledge Area Specialists and their contribution, for recruiting the reviewers and the review captains as well as coordinating the various review cycles.

67. The Editors are therefore responsible for the coherence of the entire Guide and for identifying and establishing links between the Knowledge Areas. The resolution of gaps and overlaps between Knowledge Areas will be negotiated by the Editors and the Knowledge Area Specialists themselves.

68. Summary

2. The following figure presents in a schematic form the Knowledge Area Description Specifications


Straw Man Version of Guide to the SWEBOK (5)
Plan for Developing the Stone Man Version (4)
Context Documents
Standards Road Map (6)
IEEE 610.12 Terminology Standard (7)
12207 Software Life Cycle Processes Standard (8)
Standards Documents
Baseline List of Knowledge Area (2)
Knowledge Area Jumpstart Document (9)
Contents of Knowledge Area Description
Breakdown of Topics (a-j)
Matrix of Reference Material Versus Topics (t)
Rational of Breakdown and Succinct Description of Topics (l-m)
Classification of Topics Based on Vincenti's Categories (k)
Rating of Topics Based on Bloom's Taxonomy (n)
Relevant Knowledge Areas of Related Disciplines (u)
Selected Reference Material (o-s)
Categories of Engineering Design Knowledge by Vincenti (11)
Bloom's Taxonomy (1)
Baseline List of Related Disciplines (3)
Input Documents
Stone Man Version Deliverable
( ) Reference to Criteria and Requirements letter or Document number

70. IMPORTANT RELATED DOCUMENTS (IN ALPHABETICAL ORDER OF FIRST AUTHOR)

71. 1. Bloom *et al.*, Bloom's Taxonomy of the Cognitive Domain

72. Please refer to http://www.valdosta.peachnet.edu/~whuitt/psy702/cogsys/bloom.html for a description of this hierarchy of educational objectives.

68. 2. P. Bourque, R. Dupuis, A. Abran, J. W. Moore, L. Tripp, D. Frailey, A Baseline List of Knowledge Areas for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge, Université du Québec à Montréal, Montréal, February 1999.

69. Based on the Straw Man version, on the discussions held and the expectations stated at the kick off meeting of the Industrial Advisory Board, on other body of knowledge proposals, and on criteria defined in this document, this document proposes a baseline list of ten Knowledge Areas for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge. This baseline may of course evolve as work progresses and issues are identified during the course of the project.

70. This document is available at www.swebok.org.

71. 3. P. Bourque, R. Dupuis, A. Abran, J. W. Moore, L. Tripp. A Proposed Baseline List of Related Disciplines for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge, Université du Québec à Montréal, Montréal, February 1999.

72. Based on the Straw Man version, on the discussions held and the expectations stated at the kick off meeting of the Industrial Advisory Board and on subsequent work, this document proposes a baseline list of Related Disciplines and Knowledge Areas within these Related Disciplines. This document has been submitted to and discussed with the Industrial Advisory Board and a recognized list of Knowledge Areas still has to be identified for certain Related Disciplines. Knowledge Area Specialists will be informed of the evolution of this document.

73. The current version is available at www.swebok.org

74. 4. P. Bourque, R. Dupuis, A. Abran, J. W. Moore, L. Tripp, D. Frailey, Approved Plan, Stone Man Version of the Guide to the Software Engineering Body of Knowledge, Université du Québec à Montréal, Montréal, February 1999.

75. This report describes the project objectives, deliverables and underlying principles. The intended audience of the Guide is identified. The responsibilities of the various contributors are defined and an outline of the schedule is traced. This documents defines notably the review process that will be used to develop the Stone Man version. This plan has been approved by the Industrial Advisory Board.

76. This document is available at www.swebok.org

77. 5. P. Bourque, R. Dupuis, A. Abran, J. W. Moore, L. Tripp, K. Shyne, B. Pflug, M. Maya, and G. Tremblay, Guide to the Software Engineering Body of Knowledge - A Straw Man Version, Université du Québec à Montréal, Montréal, Technical Report, September 1998.

78. This report is the basis for the entire project. It defines general project strategy, rationale and underlying principles and proposes an initial list of Knowledge Areas and Related Disciplines.

79. This report is available at www.swebok.org.

80. 6. J. W. Moore, Software Engineering Standards, A User's Road Map. Los Alamitos: IEEE Computer Society Press, 1998.

81. 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 you to key relationships between standards.

82. Even though the Guide to the Software Engineering Body of Knowledge is not a software engineering standards development project per se, special care will be taken throughout the project regarding the compatibility of the Guide with the current IEEE and ISO Software Engineering Standards Collection.

83. 7. IEEE Standard Glossary of Software Engineering Terminology, IEEE, Piscataway, NJ std 610.12-1990, 1990.

84. The hierarchy of references for terminology is Merriam Webster's Collegiate Dictionary (10th Edition), IEEE Standard 610.12 and new proposed definitions if required.

85. 8. Information Technology – Software Life Cycle Processes, International Standard, Technical ISO/IEC 12207:1995(E), 1995.

86. This standard is considered the key standard regarding the definition of life cycle process and has been adopted by the two main standardization bodies in software engineering: ISO/IEC JTC1 SC7 and the IEEE Computer Society Software Engineering Standards Committee. It also has been designated as the pivotal standard around which the Software Engineering Standards Committee (SESC) is currently harmonizing its entire collection of standards. This standard was a key input to the Straw Man version.

87. Even though we do not intend that the Guide to the Software Engineering Body of Knowledge be fully 12207-compliant, this standard remains a key input to the Stone Man version and special care will be taken throughout the project regarding the compatibility of the Guide with the 12207 standard.

88. 9. Knowledge Area Jumpstart Documents

89. A “jumpstart document” has already been provided to all Knowledge Area Specialists. These “jumpstart documents” propose a breakdown of topics for each Knowledge Area based on the analysis of the four most widely sold generic software engineering textbooks. As implied by their title, they have been prepared as an enabler for the Knowledge Area Specialist and the Knowledge Area Specialist are not of course constrained to the proposed list of topics nor to the proposed breakdown in these “jumpstart documents”.

90. 10. Merriam Webster's Collegiate Dictionary (10th Edition).

91. See note for IEEE 610.12 Standard.

92. 11. W. G. Vincenti, What Engineers Know and How They Know It - Analytical Studies from Aeronautical History. Baltimore and London: Johns Hopkins, 1990.

93. The categories of engineering design knowledge defined in Chapter 7 (The Anatomy of Engineering Design Knowledge) of this book were used as a framework for organizing topics in the various Knowledge Area “jumpstart documents “ and are imposed as decomposition framework in the Knowledge Area Descriptions because:

94. - they are based on a detailed historical analysis of an established branch of engineering: aeronautical engineering. A breakdown of software engineering topics based on these categories is therefore seen as an important mechanism for linking software engineering with engineering at large and the more established engineering disciplines;

95. - they are viewed by Vincenti as applicable to all branches of engineering;

96. - gaps in the software engineering body of knowledge within certain categories as well as efforts to reduce these gaps over time will be made apparent;

97. - due to generic nature of the categories, knowledge within each knowledge area could evolve and progress significantly while the framework itself would remain stable;

98. AUTHORSHIP OF KNOWLEDGE AREA DESCRIPTION

99. The Editorial Team will submit a proposal to the project’s Industrial Advisory Board to have

Knowledge Area Specialists recognized as authors of the Knowledge Area description.

100. STYLE AND TECHNICAL GUIDELINES

101. Knowledge Area Descriptions should conform to the International Conference on Software Engineering Proceedings format (templates are available at http://sunset.usc.edu/icse99/cfp /technical_papers.html).

102. Knowledge Area Descriptions are expected to follow the IEEE Computer Society Style Guide. See http://computer.org/author/style/cs-style.htm

103. Microsoft Word 97 is the preferred submission format. Please contact the Editorial Team if this is not feasible for you.

104. Other Detailed Guidelines:

105. When referencing the guide, we recommend that you use the full title "Guide to the SWEBOK" instead of only "SWEBOK."

106. For the purpose of simplicity, we recommend that Knowledge Area Specialists avoid footnotes. Instead, they should try to include their content in the main text.

107. We recommend to use in the text explicit references to standards, as opposed to simply inserting numbers referencing items in the bibliography. We believe it would allow to better expose the reader to the source and scope of a standard.

108. 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.

109. Make sure you use current information about references (versions, titles, etc.)

110. To make sure that some information contained in the Guide to the SWEBOK does not become rapidly obsolete, please avoid directly naming tools and products. Instead, try to name their functions. The list of tools and products can always be put in an appendix.

111. You are expected to spell out all acronyms used and to use all appropriate copyrights, service marks, etc.

112. The Knowledge Area Descriptions should always be written in third person.

113. EDITING (TO BE CONFIRMED)

114. Knowledge Area Descriptions will be edited by IEEE Computer Society staff editors. Editing includes copy editing (grammar, punctuation, and capitalization), style editing (conformance to the Computer Society magazines' house style), and content editing (flow, meaning, clarity, directness, and organization). The final editing will be a collaborative process in which IEEE Computer Society staff editors and the authors work together to achieve a concise, well-worded, and useful a Knowledge Area Description.

115. RELEASE OF COPYRIGHT

116. All intellectual properties associated with the Guide to the Software Engineering Body of Knowledge will remain with the IEEE Computer Society. Knowledge Area Specialists will be asked to sign a copyright release form.

117. It is also understood that the Guide to the Software Engineering Body of Knowledge will be put in the public domain by the IEEE Computer Society, free of charge through web technology, or other means.

118. For more information, See http://computer.org/ copyright.htm

APPENDIX B

A LIST OF RELATED DISCIPLINES FOR THE STONE MAN VERSION OF THE GUIDE TO THE SWEBOK

1. In order to circumscribe software engineering, it is necessary to identify the other disciplines with which SE shares a common boundary. These disciplines are called Related Disciplines. In this regard, the mandate of the Guide to the SWEBOK project is to Identify other disciplines that contain knowledge areas that are important to a software engineer. The list of such Knowledge areas would be useful to attain the fifth objective of the project: Provide a foundation for curriculum development and individual certification and licensing material.

2. Therefore, this appendix identifies:

3. ◆ a list of Related Disciplines, based on the Strawman Guide, on the discussions of the Industrial Advisory Board at the Industrial Advisory Board kick-off meeting in Mont-Tremblant (Canada) and on subsequent work and discussions;

4. ◆ a list of knowledge areas for these Related Disciplines, based on as authoritative a source as found.

5. These lists were to be as large as possible because we considered it easier to eliminate topics than adding them further on in the process.

6. The SWEBOK KA Specialists were asked to identify from these lists the Knowledge Areas of the Related Disciplines that are sufficiently relevant to the Software Engineering KA that has been assigned to them to be expected knowledge from a graduate with four years of experience. If deemed necessary and if accompanied by a justification, Knowledge Area Specialists could also propose additional Related Disciplines not already. These choices are presented in Appendix D. The level and extent of knowledge that a software engineer should posses within these knowledge areas is not specified at this point. This will be done by other projects according to their needs.

7. LIST OF RELATED DISCIPLINES AND SOURCES OF KNOWLEDGE AREAS.

8. Computer Science

9. ◆ It was agreed in Mont-Tremblant that the reference for this Related Discipline would be obtained through an initiative called the IEEE Computer Society and ACM Joint Task Force on "Year 2001 Model Curricula for Computing: CC-2001". To ensure proper coordination with this initiative, Carl Chang, Joint Task Force Co-Chair is a member of the Industrial Advisory Board and was present in Mont-Tremblant. Appendix B.1 lists the preliminary Knowledge Areas of Computer Science as determined by the CC-2001 group.

10. Mathematics

11. ◆ It was agreed in Mont-Tremblant that the Computing Curricula 2001 initiative would be the "conduit" to mathematics. So far, we have not received such a list of Knowledge Areas (Knowledge Units in the CC-2001 vocabulary), for Mathematics but it is expected that CC-2001 will provide it. In the mean time, the project refers to the list defined by the Computing Curriculum 1991[1] initiative and found in Appendix B.2.

12. Project Management

13. ◆ The reference for this Related Discipline is "A Guide to the Project Management Body of Knowledge"[2] published by the Project Management Institute. This document is currently being adopted as an

[1] See http://computer.org/educate/cc1991/

[2] See www.pmi.org to download this report.

IEEE software engineering standard. The list of Knowledge Areas for project management can be found in Appendix B.3.

14. Computer Engineering

15. A list of Knowledge Areas for Computer Engineering and found in Appendix B.4 was compiled from the integration of:

16. - The syllabus for the British licensing exam for the field of Computer Systems Engineering[3].

17. - The Principles and Practice of Engineering Examination - Guide for Writers and Reviewers in Electrical Engineering of the National Council of Examiners for Engineering and Surveying (USA). An appendix listed Computer Engineering Knowledge Areas for which questions should be put to the candidates.

18. - The Computer Engineering undergraduate program at the Milwaukee School of Engineering[4]. This program is considered to be a typical example of an American accredited program by the director of the Computer Engineering and Computer Science Department at MSOE.

19. Systems Engineering

20. Appendix B.5 contains a proposed list of Knowledge Areas for Systems Engineering. The list was compiled from:

21. - The EIA 632 and IEEE 1220 (Trial-Use) standards;

22. - the Andriole and Freeman paper[5];

23. - the material available on the INCOSE (International Council on Systems Engineering) website[6];

24. - a curriculum for a graduate degree in Systems Engineering at the University of Maryland[7];

25. Three experts in the field were also consulted, John Harauz, from Ontario Hydro, John Kellogg from Lockheed Martin, and Claude Laporte consultant, previously with the Armed Forces of Canada and Oerlikon Aerospace.

26. Management and Management Science

27. No definitive source has been identified so far for a list of Management and Management Science Knowledge Areas relevant to software engineering. A list was therefore compiled from

28. - the Technology Management Handbook[8] which contains many relevant chapters;

29. - the Engineering Handbook[9] which contains a section on Engineering Economics and Management covering many of the relevant topics;

30. - an article by Henri Barki and Suzanne "Rivard titled A Keyword Classification Scheme for IS Research Literature: An Update"[10].

31. The proposed list of knowledge areas for Management and Management Science can be found in Appendix B.6.

32. Cognitive Sciences and Human Factors

33. Appendix B.7 contains a list of proposed Knowledge Areas for Cognitive Sciences and Human Factors. The was compiled from the list of courses offered at the John Hopkins University Department of Cognitive Sciences[11] and from the ACM SIGCHI Curricula for Human-Computer Interaction[12].

34. The list was then refined by three experts in the field: two from UQAM and W. W. McMillan, from Eastern Michigan University. They were asked to indicate which of these topics should be known by a software engineer. The topics that were rejected by two of the three respondents were removed from the original list.

[3] See http://www.engc.org.uk

[4] See http://www.msoe.edu/eecs/ce/index.htm

[5] Stephen J. Andriole and Peter A. Freeman, *Software systems engineering: the case for a new discipline*, System Engineering Journal, Vol. 8, No. 3, May 1993, pp. 165-179.

[6] See www.incose.org

[7] See http://www.isr.umd.edu/ISR/education/msse/

[8] See CRC Press

[9] See Crc Press

[10] See MIS Quaterly, June 1993, pp. 209-226

[11] See http://www.cogsci.jhu.edu/

[12] See TABLE 1. Content of HCI athttp://www.acm.org/sigchi/cdg/cdg2.html

35. APPENDIX B.1. KNOWLEDGE AREAS OF COMPUTER SCIENCE.

36. 0. [MP] Mathematics and Physical Sciences

37. 1. [FO] Foundations

38. Complexity analysis

39. Complexity classes

40. Computability and undecidability

41. Discrete mathematics (logic, combinatorics, probability)

42. Proof techniques

43. Automata (regular expressions, context-free grammars, FSMs/PDAs/TMs)

44. Formal specifications

45. Program semantics

46. 2. [AL] Algorithms and Data Structures

47. Basic data structures

48. Abstract data types

49. Sorting and searching

50. parallel and distributed algorithms

51. 3. [AR] Computer Architecture

52. Digital logic

53. Digital systems

54. Machine level representation of data

55. Number representations

56. Assembly level machine organization

57. Memory system organization and architecture

58. Interfacing and communication

59. Alternative architectures

60. Digital signal processing

61. Performance

62. 4. [IS] Intelligence Systems (IS)

63. Artificial intelligence

64. Robotics

65. Agents

66. Pattern Recognition

67. Soft computing (neural networks, genetic algorithms, fuzzy logic)

68. 5. [IM] Information Management

69. Database models

70. Search Engines

71. Data mining/warehousing

72. Digital libraries

73. Transaction processing

74. Data compression

75. 6. [CI] Computing at the Interface

76. Human-computer interaction (usability design, human factors)

77. Graphics

78. Vision

79. Visualization

80. Multimedia

81. PDAs and other new hardware

82. User-level application generators

83. 7. [OS] Operating Systems

84. Tasks, processes and threads

85. Process coordination and synchronization

86. Scheduling and dispatching

87. Physical and virtual memory organizations

88. File systems

89. Networking fundamentals (protocols, RPC, sockets)

90. Security

91. Protection

92. Distributed systems

93. Real-time computing

94. Embedded systems

95. Mobile computing infrastructure

96. 8. [PF] Programming Fundamentals and Skills

97. Introduction to programming languages

98. Recursive algorithms/programming

99. Programming paradigms

100. Program-solving strategies

101. Compilers/translation

102. Code Generation

103. 9. [SE] Software Engineering

104. Software Engineering will not be a related discipline to Software Engineering

105. This focus group will be coordinated with the SWEBOK project in order to avoid double definitions of the field.

106. 10. [NC] Net-centric Computing

107. Computer-supported cooperative work

108. Collaboration Technology

109. Distributed objects computing (DOC/CORBA/DCOM/JVM)

110. E-Commerce

111. Enterprise computing

112. Network-level security

113. 11. [CN] Computational Science

114. Numerical analysis

115. Scientific computing

116. Parallel algorithms

117. Supercomputing

118. Modeling and simulation

119. 12. [SP] Social, Ethical, Legal and Professional Issues

120. Historical and social context of computing

121. Philosophical ethics

122. Intellectual property

123. Copyrights, patents, and trade secrets

124. Risks and liabilities

125. Responsibilities of computing professionals

126. Computer crime

127. Appendix B.2. Knowledge Areas of Mathematics

128. **Discrete Mathematics**: sets, functions, elementary propositional and predicate logic, Boolean algebra, elementary graph theory, matrices, proof techniques (including induction and contradiction), combinatorics, probability, and random numbers.

129. **Calculus**: differential and integral calculus, including sequences and series and an introduction to differential equations.

130. **Probability**: discrete and continuous, including combinatorics and elementary statistics.

131. **Linear Algebra**: elementary, including matrices, vectors, and linear transformations.

132. **Mathematical Logic**: propositional and functional calculi, completeness, validity, proof, and decision

133. Appendix B.3. Knowledge Areas of Project Management

134. The list of Knowledge Areas defined by the Project Management Institute for project management is:

135. ◆ Project Integration Management

136. ◆ Project Scope Management

137. ◆ Project Time Management

138. ◆ Project Cost Management

139. ◆ Project Quality Management

140. ◆ Project Human Resource Management

141. ◆ Project Communications Management

142. ◆ Project Risk Management

143. ◆ Project Procurement Management

144. Appendix B.4. Knowledge Areas of Computer Engineering.

145. Digital Data Manipulation

146. Processor Design

147. Digital Systems Design

148. Computer Organization

149. Storage Devices and Systems

150. Peripherals and Communication

151. High Performance Systems

152. System Design

153. Measurement and Instrumentation

154. Codes and Standards

155. Circuit Theory

156. Electronics

157. Controls

158. Combinational and Sequential Logic

159. Embedded Systems Software

160. Engineering Systems Analysis with Numerical Methods

161. Computer Modeling and Simulation

162. APPENDIX B.5. KNOWLEDGE AREAS OF SYSTEMS ENGINEERING

163. PROCESS
164. Need Analysis
165. Behavioral Analysis
166. Enterprise Analysis
167. Prototyping
168. Project Planning
169. Acquisition
170. Requirements Definition
171. System definition
172. Specification trees
173. System breakdown structure
174. Design
175. Effectiveness Analysis
176. Component specification
177. Integration
178. Maintenance & Operations
179. Configuration Management
180. Documentation
181. Systems Quality Analysis and Management
182. Systems V & V
183. System Evaluation
184. Systems Lifecycle Cost Estimation
185. Design of Human-Machine Systems
186. Fractals and self-similarities
187. ESSENTIAL FUNCTIONAL PROCESSES: (IEEE 1220)
188. Development
189. Manufacturing
190. Test
191. Distribution
192. Operations
193. Support
194. Training
195. Disposal
196. TECHNIQUES & TOOLS (IEEE 1220)
197. Metrics
198. Privacy
199. Process Improvement
200. Reliability
201. Safety
202. Security
203. Vocabulary
204. Effectiveness Assessment

205. APPENDIX B.6. KNOWLEDGE AREAS OF MANAGEMENT AND MANAGEMENT SCIENCE

206. BUSINESS STRATEGY
207. FINANCE
208. EXTERNAL ENVIRONMENT
209. Economic Environment
210. Legal Environment
211. Regulation processes
212. ORGANIZATIONAL ENVIRONMENT
213. Organizational Characteristics
214. Organizational Functions
215. Organizational Dynamics
216. INFORMATION SYSTEMS MANAGEMENT
217. Data Resource Management
218. Information Resource Management
219. Personnel Resource Management
220. IS Staffing
221. INNOVATION AND CHANGE
222. ACCOUNTING
223. TRAINING
224. MANAGEMENT SCIENCE
225. Models
226. Financial Models
227. Planning Models
228. Optimization
229. Optimization methods
230. Heuristics
231. Linear Programming
232. Goal Programming
233. Mathematical Programming
234. Statistics
235. Simulation

236. APPENDIX B.7. KNOWLEDGE AREAS OF COGNITIVE SCIENCES AND HUMAN FACTORS

237. **Cognition**

238. Cognitive AI I: Reasoning

239. Machine Learning and Grammar Induction

240. Formal Methods in Cognitive Science: Language

241. Formal Methods in Cognitive Science: Reasoning

242. Formal Methods in Cognitive Science: Cognitive Architecture

243. Cognitive AI II: Learning

244. Foundations of Cognitive Science

245. Information Extraction from Speech and Text

246. Lexical Processing

247. Computational Language Acquisition

248. The Nature of HCI

249. (Meta-)Models of HCI

250. Use and Context of Computers

251. Human Social Organization and Work

252. Application Areas

253. Human-Machine Fit and Adaptation

254. Human Characteristics

255. Human Information Processing

256. Language, Communication, Interaction

257. Ergonomics

258. Computer System and Interface Architecture

259. Input and Output Devices

260. Dialogue Techniques

261. Dialogue Genre

262. Computer Graphics

263. Dialogue Architecture

264. Development Process

265. Design Approaches

266. Implementation Techniques

267. Evaluation Techniques

268. Example Systems and Case Studies

APPENDIX C

CLASSIFICATION OF TOPICS ACCORDING TO BLOOM'S TAXONOMY

1. INTRODUCTION

2. Bloom's taxonomy is the best known and most widely used classification of cognitive educational goals. In order to help all audiences in that field who wish to use the Guide as a tool in designing course material, programs or accreditation criteria, the project was mandated to provide a first draft evaluation of the topics included in the Knowledge Areas breakdowns according Bloom's Taxonomy. This should only be seen as a jump-start document to be further developed by other steps in other, related projects.

3. Knowledge Area Specialists were asked to provide an Appendix that states for each topic at which level of Bloom's taxonomy a "graduate plus four years experience" should "master" this topic. The resulting table could also be used by the specialists themselves as a guide to choose the amount and level of reference material appropriate for each topic.

4. This appendix contains, for each Knowledge Area[1], a table identifying the topics and the associated Bloom's taxonomy level of understanding on each topic for a graduate with four years experience. The levels of understanding from lower to higher are: knowledge, comprehension, application, analysis, synthesis, and evaluation. The version used can be found at http://www.valdosta.peachnet.edu/~whuitt/psy702/cogsys/bloom.html

5. SOFTWARE REQUIREMENTS

	TOPIC	Bloom Level
6.	**Requirements engineering process**	
7.	Process models	Knowledge
8.	Process actors	Knowledge
9.	Process support	Knowledge
10.	Process quality and improvement	Knowledge
11.	**Requirements elicitation**	
12.	Requirements sources	Comprehension
13.	Elicitation techniques	Application
14.	**Requirements analysis**	
15.	Requirements classification	Comprehension
16.	Conceptual modeling	Comprehension
17.	Architectural design and requirements allocation	Analysis
18.	Requirements negotiation	Analysis
19.	**Requirement specification**	
20.	The requirements definition document	Application
21.	The software requirements specification (SRS)	Application
22.	Document structure	Application
23.	Document quality	Analysis
24.	**Requirements validation**	
25.	The conduct of requirements reviews	Analysis
26.	Prototyping	Application
27.	Model validation	Analysis
28.	Acceptance tests	Application
29.	**Requirements management**	
30.	Change management	Analysis
31.	Requirement attributes	Comprehension
32.	Requirements tracing	Comprehension

[1] Please note that the rating for the *Software Construction* Knowledge Area is still missing.

SOFTWARE DESIGN

Software Design Topic	**Know-ledge**	**Compre hension**	**Appli-cation**	**Analy -sis**	**Syn-thesis**	**Eva-luation**
I. Software Design Basic Concepts						
General design concepts		X				
The context of software design		X				
The software design process				X		X
Basic software design concepts				X		
Key issues in software design		X	X			
II. Software Architecture						
Architectural structures and viewpoints			X			
Architectural styles and patterns (macro-architecture)				X		X
Design patterns (micro-architecture)				X		X
Design of families of programs and frameworks			X			
III. Software Design Quality Analysis and Evaluation						
Quality attributes				X		
Quality analysis and evaluation tools			X	X		
Metrics			X	X		
IV. Software Design Notations						
Structural descriptions (static view)			X	X		
Behavioral descriptions (dynamic view)			X	X		
V. Software Design Strategies and Methods						
General strategies			X			
Function-oriented design			X			
Object-oriented design				X		X
Data -structure centered design		X				
Other methods		X	X			

Note: As mentioned in the URL used as reference for “Bloom’s et al.’s Taxonomy of the Cognitive Domain”, Evaluation has been considered to be at the same level as Synthesis, but using different cognitive processes.

59. SOFTWARE TESTING

	Topic	Bloom's level
60.	**A. Testing Basic Concepts and definitions**	
61.	Definitions of testing and related terminology	Analysis
62.	Faults vs. failures	Analysis
63.	Test selection criteria/Test adequacy criteria (or stopping rules)	Application
64.	Testing effectiveness/Objectives for testing	Comprehension
65.	Testing for defect identification	Comprehension
66.	The oracle problem	Comprehension
67.	Theoretical and practical limitations of testing	Application
68.	The problem of infeasible paths	Comprehension
69.	Testability	Comprehension
70.	Testing vs. Static Analysis Techniques	Application
71.	Testing vs. Correctness Proofs	Knowledge
72.	Testing vs. Debugging	Comprehension
73.	Testing vs. Programming	Application
74.	Testing within SQA	Application
75.	Testing within CMM	Knowledge
76.	Testing within Cleanroom	Knowledge
77.	Testing and Certification	Comprehension
78.	**B. Test Levels**	
79.	Unit testing	Application
80.	Integration testing	Application
81.	System testing	Application
82.	Acceptance/qualification testing	Application
83.	Installation testing	Application
84.	Alpha and Beta testing	Application
85.	Conformance testing/Functional testing/Correctness testing	Application
86.	Reliability achievement and evaluation by testing	Comprehension
87.	Regression testing	Application
88.	Performance testing	Comprehension
89.	Stress testing	Comprehension
90.	Back-to-back testing	Knowledge
91.	Recovery testing	Comprehension
92.	Configuration testing	Comprehension
93.	Usability testing	Comprehension
94.	**C. Test Techniques**	
95.	Ad hoc	Synthesis
96.	Equivalence partitioning	Application
97.	Boundary-value analysis	Application
98.	Decision table	Knowledge
99.	Finite-state machine-based	Knowledge
100.	Testing from formal specifications	Knowledge
101.	Random testing	Application
102.	Reference models for code-based testing (flow graph, call graph)	Application
103.	Control flow-based criteria	Application
104.	Data flow-based criteria	Comprehension

	Topic	Bloom’s level
105.	Error guessing	Application
106.	Mutation testing	Knowledge
107.	Operational profile	Comprehension
108.	SRET	Knowledge
109.	Object-oriented testing	Application
110.	Component-based testing	Comprehension
111.	GUI testing	Knowledge
112.	Testing of concurrent programs	Knowledge
113.	Protocol conformance testing	Knowledge
114.	Testing of distributed systems	Knowledge
115.	Testing of real-time systems	Knowledge
116.	Testing of scientific software	Knowledge
117.	Functional and structural	Synthesis
118.	Coverage and operational/Saturation effect	Knowledge
119.	**D. Test related measures**	
120.	Program measurements to aid in planning and designing testing.	Synthesis
121.	Types, classification and statistics of faults	Application
122.	Remaining number of defects/Fault density	Application
123.	Life test, reliability evaluation	Comprehension
124.	Reliability growth models	Knowledge
125.	Coverage/thoroughness measures	Application
126.	Fault seeding	Knowledge
127.	Mutation score	Knowledge
128	Comparison and relative effectiveness of different techniques	Comprehension
129.	**E. Managing the Test Process**	
130.	Attitudes/Egoless programming	Application
131.	Test process	Synthesis
132.	Test documentation and workproducts	Synthesis
133.	Internal vs. independent test team	Comprehension
134.	Cost/effort estimation and other process metrics	Application
135.	Test reuse	Application
136.	Planning	Application
137.	Test case generation	Application
138.	Test environment development	Application
139.	Execution	Application
140.	Test results evaluation	Application
141.	Trouble reporting/Test log	Application
142.	Defect tracking	Application

143. SOFTWARE MAINTENANCE

	TOPIC	BLOOM LEVEL
144.	**Introduction to Software Maintenance**	Comprehension
145.	*Need for Maintenance*	Comprehension
146.	*Categories of Maintenance*	Comprehension
147.	**Maintenance Activities**	Comprehension
148.	*Unique Activities*	Comprehension
149.	*Supporting Activities*	Comprehension
150.	Configuration Management	Comprehension
151.	Quality	Comprehension
152.	*Maintenance Planning Activity*	Comprehension
153.	**Maintenance Process**	Synthesis
154.	*Maintenance Process Models*	Synthesis
155.	**Organization Aspect of Maintenance**	Comprehension
156.	*The Maintainer*	Comprehension
157.	*Outsourcing*	Comprehension
158.	*Organizational Structure*	Comprehension
159.	**Problems of Software Maintenance**	Comprehension
160.	*Technical*	Comprehension
161.	Limited Understanding	Comprehension
162.	Testing	Comprehension
163.	Impact Analysis	Comprehension
164.	Maintainability	Comprehension
165.	*Management*	Comprehension
166.	Alignment with organizational issues	Comprehension
167.	Staffing	Comprehension
168.	Process issues	Synthesis
169.	**Maintenance cost and Maintenance Cost Estimation**	Comprehension
170.	*Cost*	Comprehension
171.	*Cost estimation*	Comprehension
172.	*Parametric models*	Comprehension
173.	*Experience*	Comprehension
174.	**Software Maintenance Measurements**	Synthesis
175.	*Establishing a Metrics Program*	Comprehension
176.	*Specific Measures*	Synthesis
177.	**Techniques for Maintenance-**	Synthesis
178.	*Program Comprehension*	Synthesis
179.	*Re-engineering*	Synthesis
180.	*Reverse Engineering*	Synthesis
181.	*Impact Analysis*	Synthesis
182.	**Resources**	Comprehension

183. SOFTWARE CONFIGURATION MANAGEMENT

SCM TOPIC	Bloom Level
184. I. Management of the SCM Process	Knowledge
185. A. Organizational Context for SCM	Knowledge
186. B. Constraints and Guidance for	Knowledge
187. C. Planning for SCM	Knowledge
188. 1. SCM Organization and	Knowledge
189. 2. SCM Resources and	Comprehension
190. 3. Tool Selection and	Knowledge
191. 4. Vendor/Subcontractor	Knowledge
192. 5. Interface Control	Comprehension
193. D. Software Configuration	Knowledge
194. E. Surveillance of SCM	Comprehension
195. 1. SCM Metrics and	Comprehension
196. 2. In-Process Audits of	Knowledge
197. II. Software Configuration Identification	Comprehension
198. A. Identifying Items to be	Comprehension
199. 1. Software	Comprehension
200. 2. Software	Comprehension
201. 3. Software configuration	Comprehension
202. 4. Software Versions	Comprehension
203. 5. Baselines	Comprehension
204. 6. Acquiring Software	Knowledge
205. B. SCM Library	Comprehension
206. III. Software Configuration Control	Application
207. A. Requesting, Evaluating, and	Application
208. 1. Software	Application
209. 2. Software Change	Application
210. B. Implementing Software Changes	Application
211. C. Deviations & Waivers	Comprehension
212. IV. Software Configuration Status Accounting	Comprehension
213. A. Software Configuration Status	Comprehension
214. B. Software Configur ation Status	Comprehension
215. V. Software Configuration Auditing	Knowledge
216. A. Software Functional	Knowledge
217. B. Software Physical Configuration	Knowledge
218. C. In-process Audits of a Software	Knowledge
219. VI. Software Release Management & Delivery	Comprehension
220. A. Software Building	Comprehension
221. B. Software Release Management	Comprehension

222. SOFTWARE ENGINEERING MANAGEMENT

Topic	Level
Determining the goals of a measurement program	Synthesis
Size measurement	Analysis
Complexity measurement	Analysis
Performance measurement	Analysis
Resource measurement	Analysis
Goal/Question/Metric	Application
Measurement validity (scales)	Comprehension
Survey techniques and questionnaire design	Knowledge
Data collection	Knowledge
Model building and calibration	Evaluation
Model evaluation	Synthesis
Implementation of models	Analysis
Interpretation of models	Analysis
Function Point Analysis	Application
COCOMO	Application
Portfolio management	Comprehension
Vendor management	Application
Subcontract management	Knowledge
Policy management	Comprehension
Personnel management	Analysis
Communication	Analysis
Requirements analysis	Comprehension
Use cases	Comprehension
Proposal construction	Application
Feasibility analysis	Application
Revision of requirements	Comprehension
Prototyping	Comprehension
Risk management	Synthesis
Process planning	Analysis
Determining deliverables Quality management	Comprehension Synthesis
Schedule and cost estimation	Analysis
Resource allocation	Application
Task and responsibility allocation	Application
Implementing a metrics program	Analysis
Implementing plans	Application
Process monitoring	Application
Change control	Comprehension
Configuration management	Comprehension
Scenario analysis	Comprehension
Feedback and reporting	Application
Determining satisfaction of requirements	Comprehension
Reviewing and evaluating performance	Application
Determining closure	Application
Archival activities	Comprehension
Maintenance	Comprehension
System retirement	Comprehension

223. SOFTWARE ENGINEERING PROCESS

Topic	Bloom Level
Basic Concepts and Definitions	
Themes	Comprehension
Terminology	Knowledge
Process Infrastructure	
The Experience Factory	Comprehension
The Software Engineering Process Group	Comprehension
Process Measurement	
Methodology in Process Measurement	Comprehension
Process Measurement Paradigms	Comprehension
Analytic Paradigm	Comprehension
Benchmarking Paradigm	Comprehension
Process Definition	
Types of Process Definitions	Application
Life Cycle Models	Application
Software Life Cycle Models	Application
Notations for Process Definitions	Application
Process Definition Methods	Application
Automation	Knowledge
Qualitative Process Analysis	
Process Definition Review	Comprehension
Root Cause Analysis	Comprehension
Process Implementation and Change	
Paradigms for Process Implementation and Change	Comprehension
Guidelines for Process Implementation and Change	Comprehension
Evaluating the Outcome of Process Implementation and Change	Comprehension

224. SOFTWARE ENGINEERING TOOLS AND METHODS

Topics	Bloom level
I. Software Tools	
A. Software Requirements Tools	application
B. Software Design Tools	application
C. Software Construction Tools	
1. program editors	application
2. compilers	application
3. debuggers	application
D. Software Testing Tools	
1. test generators	comprehension
2. test execution frameworks	application
3. test evaluation tools	application
4. test management	comprehension
E. Software Maintenance Tools	
1. comprehension tools	application
2. Reverse engineering tools	knowledge
3. Re-engineering tools	knowledge
4. traceability tools	knowledge
F. Software Engineering Process Tools	
1. integrated CASE environments	application
2. Process-centered software engineering environments	comprehension
3. Process modeling tools	knowledge
G. Software Quality Analysis Tools	
1. inspection tools	comprehension
2. static analysis tools	application
3. performance analysis tools	comprehension
H. Software Configuration Management Tools	
1. version management tools	application
2. release and build tools	application
I. Software Engineering Management Tools	
1. project planning and tracking tools	application
2. risk analysis and management tools	comprehension
3. measurement tools	application
4. defect, enhancement, issue and problem tracking tools	application
J. Infrastructure Support Tools	
1. interpersonal communication tools	application
2. information retrieval tools	application
3. system administration and support tools	comprehension
K. Miscellaneous	

1. tool integration techniques	knowledge
2. meta tools	comprehension
3. tool evaluation	application
II. Development Methods	
A. Heuristic Methods	
1. ad-hoc methods	application
2. structured methods	application
3. data-oriented methods	application
4. object-oriented methods	application
5. domain-specific methods	knowledge
B. Formal Methods	
1. specification languages	comprehension
2. refinement	knowledge
3. verification/proving properties	comprehension
C. Prototyping Methods	
1. styles	comprehension
2. prototyping targets	application
3. evaluation techniques	comprehension
D. Miscellaneous	
1. Method evaluation	application

225. SOFTWARE QUALITY

226. All software engineers are responsible for the quality of the products they build. We consider that the knowledge requirements for topics in Software Quality Analysis vary depending on the role of the software engineer. We use the roles of programmer, SQA/VV specialist, and project manager. The programmer will design and build the system, possibly be involved in inspections and reviews, analyze his work products statically, and possibly perform unit test. This person may turn over the products to others who will conduct integration and higher levels of testing, and may be asked to submit data on development tasks, but will not conduct analyses on faults or on measurements. The SQA/VV specialist will plan and implement the processes for software quality analysis, verification, and validation. The project manager of the development project will use the information from the software quality analysis processes to make decisions. Of course, in a small project, the software engineer may have to assume all of these roles, in which case, the highest of the three is appropriate.

Software Quality Topic (Numbered as to Section in this KA)	Bloom Level*, By Job Responsibility		
	Programmer	***SQA/VV Spec.***	***Project Manager***
227. 3. Software Quality Concepts			
228. 3.1 Measuring the Value of Quality	Comprehension	Comprehension	Analysis
229. 3.2 ISO 9126 Quality Description	Comprehension	Comprehension	Comprehension
230. 3.3 Dependability	Comprehension	Comprehension	Comprehension
231. 3.4 Special Types of Systems and Quality Needs	Comprehension	Comprehension	Comprehension
232. 3.5 Quality Attributes for Engineering Process	Comprehension	Comprehension	Comprehension
233. 4. Defining SQA and V&V	Comprehension	Comprehension	Comprehension
234. 5. Planning for SQA and V&V			
235. 5.1 The SQA Plan	Application	Synthesis	Evaluation
236. 5.2 The V&V Plan	Application	Synthesis	Evaluation
237. 6. Activities and Techniques for SQA and V&V			
238. 6.1 Static Techniques			
239. 6.1.1 Audits, Reviews, and Inspections	Application	Evaluation	Analysis
240. 6.1.2 Analytic Techniques	Application	Evaluation	Analysis
241. 6.2 Dynamic Techniques	Application	Evaluation	Analysis
242. 7. Measurement Applied to SQA and V&V			
243. 7.1 Fundamentals of Measurement	Application	Evaluation	Analysis
244. 7.2 Metrics	Application	Evaluation	Analysis
245. 7.3 Measurement Techniques	Application	Evaluation	Analysis
246. 7.4 Defect Characterization	Application	Evaluation	Analysis
247. 7.5 Additional uses of SQA and V&V data	Application	Evaluation	Analysis

248. *The levels, in ascending order: Knowledge, Comprehension, Application, Analysis, Synthesis, Evaluation.

APPENDIX D

IDENTIFICATION OF RELEVANT KNOWLEDGE AREAS OF RELATED DISCIPLINES

1. INTRODUCTION

2. Each SWEBOK KA description identified relevant KAs from Related Disciplines. Although these KAs are merely identified without additional description or references, they should aid curriculum developers. This Appendix must be viewed as a jumpstart document and as aid to curriculum developers rather than as a definitive list of relevant Knowledge Areas of Related Disciplines.

3. Relevant Knowledge Areas of Computer Science

		SR[1]	SD	SC[2]	ST	SM	SCM	SEM	SEP	SETM	SQ
4.	**Mathematics and Physical Sciences**										
5.	**Foundations**		X		X	X				X	
6.	**Algorithms and Data Structures**		X		X	X				X	
7.	**Computer Architecture**		X		X	X					
8.	**Intelligence Systems** (IS)					X			X		
9.	**Information Management**		X			X	X			X	
10.	**Computing at the Interface**		X		X	X				X	
11.	**Operating Systems**		X		X	X	X				
12.	**Programming Fundamentals and Skills**		X		X	X	X	X		X	
13.	**Net-centric Computing**		X		X	X	X			X	
14.	**Computational Science**					X			X		
15.	**Social, Ethical, Legal and Professional Issues**		X		X	X		X			

16. Relevant Knowledge Areas of Mathematics

		SR	SD	SC	ST	SM	SCM	SEM	SEP	SETM	SQ
17.	**Discrete Mathematics**		X		X	X	X			X	
18.	**Calculus**										
19.	**Probability**		X		X	X		X	X	X	
20.	**Linear Algebra**					X					
21.	**Mathematical Logic**		X		X	X				X	

22. SR : Software Requirements

23. SD: Software Design

24. SC: Software Construction

25. ST: Software Testing

26. SM: Software Maintenance

27. SCM: Software Configuration Management

28. SEM: Software Engineering Management

[1] Relevant Knowledge Areas of Related Disciplines will be identified in version 0.9 of the Guide.

[2] Relevant Knowledge Areas of Related Disciplines will be identified in version 0.9 of the Guide.

SEP:	Software Engineering Process
SETM:	Software Engineering Tools and Methods
SQ:	Software Quality

Relevant Knowledge Areas of Project Management

	SR	SD	SC	ST	SM	SCM	SEM	SEP	SETM	SQ
Project Integration Management		X		X		X	X	X	X	
Project Scope Management		X					X	X	X	
Project Time Management		X		X			X	X	x	
Project Cost Management		X		X			X	X	X	
Project Quality Management		X		X		X	X	X	X	
Project Human Resource Management				X			X	X	X	
Project Communications Management				X			X	X	X	
Project Risk Management		X		X		X	X	X	X	
Project Procurement Management							X	X	X	

Relevant Knowledge Areas of Computer Engineering

	SR	SD	SC	ST	SM	SCM	SEM	SEP	SETM	SQ
Digital Data Manipulation										
Processor Design										
Digital Systems Design										
Computer Organization				X						
Storage Devices and Systems										
Peripherals and Communication				X						
High Performance Systems										
System Design		X		X						
Measurement and Instrumentation				X						
Codes and Standards				X						
Circuit Theory										
Electronics										
Controls										
Combinational and Sequential Logic										
Embedded Systems Software				X						
Engineering Systems Analysis with Numerical Methods										
Computer Modeling and Simulation				X						

60. Relevant Knowledge Areas of Systems Engineering

		SR	SD	SC	ST	SM	SCM	SEM	SEP	SETM	SQ
61.	**Process**		X		X	X	X	X	X	X	
62.	Need Analysis							X			
63.	Behavioral Analysis		X					X			
64.	Enterprise Analysis							X			
65.	Prototyping		X		X			X			
66.	Project Planning				X		X	X			
67.	Acquisition				X			X			
68.	Requirements Definition				X		X	X			
69.	System definition				X		X	X			
70.	Specification trees							X			
71.	System breakdown structure		X					X			
72.	Design		X		X			X			
73.	Effectiveness Analysis				X			X			
74.	Component specification		X		X			X			
75.	Integration				X		X	X			
76.	Maintenance & Operations				X	X	X	X			
77.	Configuration Management				X	X	X	X			
78.	Documentation				X	X	X	X			
79.	Systems Quality Analysis and Management				X		X	X			
80.	Systems V & V				X		X	X			
81.	System Evaluation				X		X	X			
82.	Systems Lifecycle Cost Estimation				X			X			
83.	Design of Human-Machine Systems				X			X			
84.	Fractals and self-similarities										
85.	**Essential Functional Processes: (IEEE 1220)**				X	X	X	X	X		
86.	Development				X	X	X	X			
87.	Manufacturing										
88.	Test				X	X	X	X			
89.	Distribution				X	X	X	X			
90.	Operations				X	X	X	X			
91.	Support				X	X	X	X			
92.	Training				X	X		X			
93.	Disposal				X	X		X			
94.	**Techniques & Tools (IEEE 1220)**				X	X	X	X	X	X	
95.	Metrics				X	X	X	X			
96.	Privacy				X	X		X			
97.	Process Improvement				X	X		X			
98.	Reliability				X	X	X	X			
99.	Safety				X	X		X			
100.	Security				X	X	X	X			
101.	Vocabulary				X	X		X			
102.	Effectiveness Assessment				X						

103. Relevant Knowledge Areas of Management and Management Science

104. MANAGEMENT

	SR	SD	SC	ST	SM	SCM	SEM	SEP	SETM	SQ
105. **Business Strategy**							X	X		
106. **Finance**										
107. **External Environment**				X			X	X		
108. Economic Environment				X			X			
109. Legal Environment				X			X			
110. Regulation processes				X			X			
111. **Organizational environment**				X	X		X	X		
112. Organizational Characteristics				X	X	X	X			
113. Organizational Functions				X	X	X	X			
114. Organizational Dynamics				X	X	X	X			
115. **Information Systems Management**				X	X		X	X		
116. Data Resource Management				X	X	X	X			
117. Information Resource Management				X			X			
118. Personnel Resource Management				X			X			
119. IS Staffing				X	X		X			
120. **Innovation and change**				X			X	X		
121. **Accounting**				X						
122. **Training**				X	X		X	X		

123.. MANAGEMENT SCIENCE

	SR	SD	SC	ST	SM	SCM	SEM	SEP	SETM	SQ
124. **Models**							X			
125. Financial Models							X			
126. Planning Models							X			
127. **Optimization**					X		X			
128. Optimization methods							X			
129. Heuristics							X			
130. Linear Programming					X		X			
131. Goal Programming							X			
132. Mathematical Programming					X		X			
133. **Statistics**					X		X	X		
134. **Simulation**					X		X	x		

135. Relevant Knowledge Areas of Cognitive Sciences and Human Factors

		SR	SD	SC	ST	SM	SCM	SEM	SEP	SETM	SQ
136.	Cognition							X			
137.	Cognitive AI I: Reasoning										
138.	Machine Learning and Grammar Induction										
139.	Formal Methods in Cognitive Science: Language										
140.	Formal Methods in Cognitive Science: Reasoning										
141.	Formal Methods in Cognitive Science: Cognitive Architecture										
142.	Cognitive AI II: Learning										
143.	Foundations of Cognitive Science										
144.	Information Extraction from Speech and Text										
145.	Lexical Processing										
146.	Computational Language Acquisition										
147.	**The Nature of HCI**				X			X			
148.	(Meta-)Models of HCI				X			X			
149.	**Use and Context of Computers**				X			X		X	
150.	Human Social Organization and Work				X			X			
151.	Application Areas				X			X			
152.	Human-Machine Fit and Adaptation				X			X			
153.	**Human Characteristics**				X			X			
154.	Human Information Processing				X			X			
155.	Language, Communication, Interaction				X			X			
156.	Ergonomics				X			X			
157.	**Computer System and Interface Architecture**		X		X					X	
158.	Input and Output Devices				X						
159.	Dialogue Techniques										
160.	Dialogue Genre										
161.	Computer Graphics				X						
162.	Dialogue Architecture		X								
163.	**Development Process**		x		X			X		X	
164.	Design Approaches		X		X			X			
165.	Implementation Techniques				X			X			
166.	Evaluation Techniques		X		X			X			
167.	Example Systems and Case Studies				X			X			

APPENDIX E

CHANGES BETWEEN VERSION 0.5 AND VERSION 0.7 OF THE GUIDE

1. This document lists the high-level or major changes that were incorporated in version 0.7 of the Stoneman Guide to the Software Engineering Body of Knowledge from version 0.5. These changes are based on the detailed analysis by the editorial team of reviewer feedback gathered on version 0.5 of the Guide. Feedback from the Knowledge Area Specialist updating the Knowledge Area Descriptions was also considered when making these changes.

2. Additionally to what is listed below, a total of about five thousand different comments were received, compiled, disposed of by the Knowledge Area Specialists in the Knowledge Area Descriptions, and included in a database - which can be accessed at www.swebok.org.

3. The major changes listed below were all approved by the project's Industry Advisory Board.

	Major Change	Reason for Change
4.	Impose a page limit on the cited reference material for each Knowledge Area. (Criteria R in Appendix A presents the details of how this page limit is stated)	The amount of reference material currently cited was too large to be manageable, published on the Web and eventually taught in a reasonable timeframe. This is notably due to the fact that many of these references are entire books.
5.	Modify the Criteria R of KA Descriptions Specifications (see Appendix A). This criteria was initially stated as : "The Knowledge Area Specialist are expected to adopt 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: • quality (in general), • measurement • tools • standards." So that: ◆ all topics related to tools are included in the Software Engineering Methods and Tools Knowledge Area. ◆ Standards are included in the cited reference material not in the breakdowns of topics. ◆ Software Quality Analysis has appropriate links to other Knowledge Areas	Reviewer feedback indicated that these four common themes were unevenly discussed in the Knowledge Area descriptions.

	Major Change	**Reason for Change**
6.	Remove "Component Integration" from the current version of the Guide. It was previously included in "Software Engineering Infrastructure"	The Knowledge Area Specialist indicates that the request by the Industrial Advisory Board to include component integration (standard designs, integration and reuse) in this Knowledge Area is difficult to achieve. Few links were identified between these topics and the other two major components of this Knowledge Area (Methods and Tools). Reviewers generally agreed with the weak fit of "component integration" in this Knowledge Area. The editorial team concluded that though there is a strong industry need for this type of knowledge, there is not yet sufficient consensus on what portion of it is generally accepted.
7.	Rename the Knowledge Area "Software Engineering Infrastructure" as "Software Engineering Methods and Tools."	The editorial team recommended this change due to the change proposed above and to the fact that reviewer feedback indicates varying interpretations of what "software engineering infrastructure" means.
8.	The taxonomy of tools in Software Engineering Methods and Tools should be broken down as per the list of Knowledge Areas o the Stoneman Guide.	As stated above, the editorial team recommended that "tools" be dropped from the "common themes" discussed in all Knowledge Areas and that all topics related to tools be concentrated in this Knowledge Area. However, the editorial team also recommended that the distribution of topics related to tools in the various Knowledge Area breakdowns be reconsidered for the Iron Man version. Breaking down the topics related to tools as per the list of Knowledge Areas facilitates this redistribution. The editorial team also considers that the decision of whether or not « software engineering methods and tools » should remain as a distinct Knowledge Area should be reevaluated in the Iron Man phase.
9.	Rename the "Software Evolution and Maintenance" Knowledge Area to "Software Maintenance"	Current standards adopt the term "software maintenance": IEEE 1219, ISO/IEC 14764 and ISO/IEC 12207 The Knowledge Area Specialist states himself that it is "common practice" to refer to this as "software maintenance".
10.	Rename the "Software Requirements Analysis" Knowledge Area to "Software Requirements"	This recommendation is based on the statement found below by the Knowledge Area Specialist and on the fact that reviewer feedback did not oppose this statement. The Knowledge Area Specialists write that: "The knowledge area was originally proposed as 'Software Requirements Analysis'. However, as a term to denote the whole process of acquiring and handling of software requirements, 'Requirements Analysis' has been largely superceded by 'Requirements Engineering'. We therefore use 'Requirements Engineering' to denote the knowledge area and 'Requirements Analysis' as one of the activities that comprise 'Software Requirements Engineering."
11.	Rename the "Software Quality Analysis" Knowledge Area to "Software Quality"	To remove the duplication of having SQA listed at the first and second levels of the breakdown.