Guide to the Software Engineering Body of Knowledge

A Stone Man Version

(Version 0.5)

SWEBOK

October 1999

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

Corporate support by:

Project managed by:

Co-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

Project Champion:

Leonard L. Tripp, IEEE Computer Society

Table of Contents

INTRODUCTORY TEXT FROM THE EDITORIAL TEAM

The IEEE Computer Society and the Association for Computing Machinery are working on a joint project to develop a guide to the Software Engineering Body Of Knowledge (SWEBOK). This is the current draft (version 0.5 completed in September 1999) of the Stoneman version of the Guide[1]. Articulating a body of knowledge is an essential step toward developing a profession because it represents a broad consensus regarding the contents of the discipline. Without such a consensus, there is no way to validate a licensing examination, set a curriculum to prepare individuals for the examination, or formulate criteria for accrediting the curriculum.

The project team is currently working on an update to this draft version of the Guide based on the results of the second review cycle. Early in 2000, major professional societies and the software engineering community will be invited to participate in the third review cycle. The completed Stoneman guide will then be made available on the Web in Spring 2000.

OBJECTIVES AND AUDIENCE

The SWEBOK project objectives are:

1. Characterize the contents of the software engineering discipline.
2. Provide topical access to the Software Engineering Body of Knowledge.
3. Promote a consistent view of software engineering worldwide.
4. 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.
5. Provide a foundation for curriculum development and individual certification material.

The product of the SWEBOK project will not be the Body of Knowledge itself, but rather a guide to it. The knowledge already exists; the goal is to gain consensus on the core subset of knowledge characterizing the software engineering discipline.

To achieve this goal, the project 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 aims to serve practicing software engineers and the officials responsible for making public policy regarding licensing and professional guidelines. In addition, professional societies and educators defining the

[1] All final and intermediate products of the SWEBOK project including the contents of this paper can be downloaded without any charge from www.swebok.org

certification rules, accreditation policies for university curricula, and guidelines for professional practice will benefit from SWEBOK, as well as students learning software engineering.

THE GUIDE

The project comprises three phases: Strawman, Stoneman, and Ironman:

- The 1998 Strawman Guide, completed within nine months of project initiation, served as a model for organizing the SWEBOK project.
- Spring 2000 will see the completion of the Stoneman version.
- The Ironman phase will continue for two or three years after the completion of the Stoneman version. Following the principles of the Stoneman phase, Ironman will benefit from more in-depth analyses, a broader review process, and the experience gained from trial usage**.**

The SWEBOK Guide will organize the body of knowledge into several Knowledge Areas. In its current draft, the Stoneman version of the Guide identifies 10 Knowledge Areas (see Table 1) which form the chapters of this Guide. They are printed in alphabetical order in this paper copy. Table 1 also indicates the domain experts who are responsible for writing the chapters.

TABLE 1. THE SWEBOK KNOWLEDGE AREAS AND THEIR CORRESPONDING SPECIALISTS.

Knowledge Area	**Specialists**
Software configuration management	John A. Scott and David Nisse, Lawrence Livermore Laboratory, US
Software construction	Terry Bollinger, The MITRE Corporation, US
Software design	Guy Tremblay, Université du Québec à Montréal, Canada
Software engineering infrastructure	David Carrington, The University of Queensland, Australia
Software engineering management	Stephen G. MacDonell and Andrew R. Gray, University of Otago, New Zealand
Software engineering process	Khaled El Emam, National Research Council, Canada
Software evolution and maintenance	Thomas M. Pigoski, TECHSOFT, US
Software quality analysis	Dolores Wallace and Larry Reeker, National Institute of Standards and Technology, US
Software requirements analysis	Pete Sawyer and Gerald Kotonya, Lancaster University, UK
Software testing	Antionia Bertolino, Consiglio Nazionale delle Ricerche, Italy

REVIEWS

The development of the Stoneman version includes three public review cycles. The first review cycle focused on the soundness and 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.

The second review cycle was organized around the guidelines originally given to the

Knowledge Area specialists or chapter authors. These guidelines can also be found at the back of this paper copy. A considerably larger group of professionals, organized into review viewpoints, answered a detailed questionnaire for each Knowledge Area 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. The results of this review cycle, completed in October 1999, are also available on the project's Web site. Knowledge Area specialists will document how reviewer feedback was resolved in the Knowledge Area descriptions or chapters.

The focus of the third review cycle review to be conducted on the entire Guide will be on the correctness and utility of the Guide. This review cycle, scheduled to start in January 2000, will be completed by individuals and organizations representing a cross-section of potential interest groups. Hundreds of professionals have already been recruited to review the entire Guide, and we're soliciting more to fulfill our coverage objectives.

How to Contribute to the Project

Those interested in participating in the third review cycle of the Guide to the Software Engineering Body of Knowledge (Stoneman version) can volunteer by signing up at the project's Web site, http://www.swebok.org. The transition from Stoneman to Ironman will be based primarily on feedback received from trial applications of the Stoneman guide. Those interested in performing trial applications are invited to contact the Editorial team.

Project Champion:	Leonard Tripp, 1999 President, l.tripp@computer.org
Project Coexecutive Editors:	Alain Abran, Université du Québec à Montréal, Abran.alain@uqam.ca James W. Moore, The MITRE Corporation James.W.Moore@ieee.org
Project Editors:	Pierre Bourque, Université du Québec à Montréal, Bourque.pierre@uqam.ca Robert Dupuis, Université du Québec à Montréal Dupuis.robert@uqam.ca

Software Configuration Management

Draft Version 0.5

John A. Scott
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

ABSTRACT

This paper presents an overview of the knowledge area of software configuration management for 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.

Keywords
Software configuration management, software configuration identification, software configuration control, software configuration status accounting, software configuration auditing, software release management.

Acronyms

CCB	Change Control Board
CM	Configuration Management
FCA	Functional Configuration Audit
PCA	Physical Configuration Audit
SCI	Software Configuration Item
SCR	Software Change Request
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SCSA	Software Configuration Status Accounting
SDD	Software Design Description
SRS	Software Requirements Specification

INTRODUCTION

A system can be defined as a collection of components organized to accomplish a specific function or set of functions [11]. 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 [5]. 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 discrete 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 [3]. CM is formally defined [11] as:

> "A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements."

The concepts of configuration management apply to all items to be controlled although there are some differences in implementation between hardware configuration management and software configuration management.

This paper presents a breakdown of the key software configuration management (SCM) concepts along with a succinct description of each concept. In keeping with the directions of the Software Engineering Body of Knowledge project, the following sections briefly describe the primary activities of SCM. These activities (see Fig.1) 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. In the discussion of these SCM activities, the commonly recognized topics are identified and summarized.

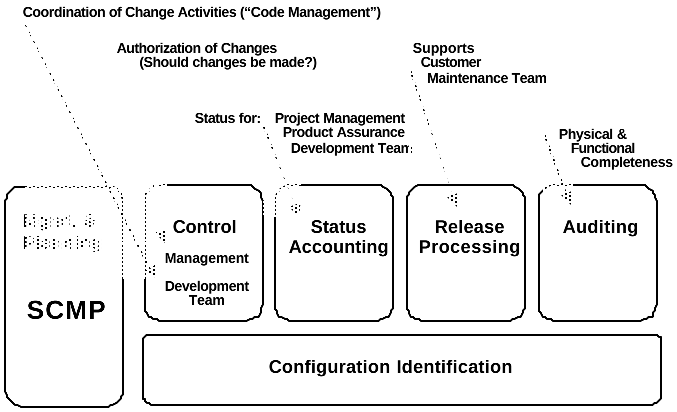

Coordination of Change Activities (“Code Management”)
Authorization of Changes (Should changes be made?)
Supports Customer Maintenance Team
Status for: Project Management Product Assurance Development Team
Physical & Functional Completeness
Mgmt. & Planning
SCMP
Control
Management
Development Team
Status Accounting
Release Processing
Auditing
Configuration Identification

Figure 1. SCM Activities

Following the breakdown are the key references for SCM along with a cross-reference of topics that each listed reference covers. Finally, topics in related disciplines that are important to SCM are identified.

BREAKDOWN OF TOPICS FOR SOFTWARE CONFIGURATION MANAGEMENT

2.1 Rationale for the Breakdown

One of the primary goals of the SWEBOK project 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 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, which identifies a ‘Release Management and Delivery’ activity.

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 [10] and [34].

The hierarchy of topics chosen for the breakdown presented in this paper is expected to evolve as the SWEBOK project review processes proceed. A detailed discussion of the rationale for the proposed breakdown, keyed to SWEBOK development criteria, is given in Appendix B.

2.2 Breakdown of Topics

The breakdown of topics, along with a brief description of each, is provided in this section. A table summarizing the breakdown hierarchy is given in Table 1 of Appendix A.

I. Management of the SCM Process

Software configuration management is a supporting software life cycle process that benefits project and line management, development and assurance activities, and the customers and users of the end product. A successful SCM implementation must be carefully planned and managed. This requires an understanding of the organizational context for, and the constraints placed upon, the design and implementation of the SCM process.

I.A Organizational Context for SCM

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.

SCM, like other processes such as software quality assurance and software verification and validation, is categorized as a supporting life cycle process [18]. 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 should rest with a distinct organizational element.

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 both must be consistent with system level CM. Buckley [5] describes SCM within this context. SCM might also be related to an organization’s records management activity since some items under SCM control might also be project records subject to provisions of the organization’s quality assurance program.

Perhaps the closest relationship is with the software development organization. The environment for developing software, including such things as the software life cycle model and its resulting schedules, the development and target platforms, and the software development tools, is also the environment within which many of the software configuration control tasks are conducted. Frequently, the same tool systems support both development and SCM purposes.

I.B Constraints and Guidance for SCM

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 controlled as software configuration items. When software products to be developed have the potential to affect the public safety, external regulatory bodies may impose constraints. For example, see [35]. Finally, the particular software life cycle model chosen for a software development project and the development tools selected for its implementation affect the design and implementation of the SCM process [4].

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 [28] or the ISO SPICE guidance [9]. ‘Best practice’ is also reflected in the standards on software engineering issued by the various standards organizations. Moore [26] provides a roadmap to these organizations and their standards.

I.C Planning for SCM

The planning of a 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).

I.C.1 SCM Organization and Responsibilities

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.

I.C.2 SCM Resources and Schedules

The planning for SCM identifies the staff and tools involved in carrying out SCM tasks. It addresses schedule questions by establishing necessary sequences of SCM tasks and identifying the relationships of the tasks to the project development schedules and milestones. Any training requirements necessary for implementing the plans are also specified.

I.C.3 Tool Selection and Implementation

Different types of tools, and procedures for their use, are used to support the SCM activities (see Fig. 2). For example, code management tools are needed to support the operation of software development libraries. These tools control access to library elements, coordinate the activities of multiple users, and help 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 can support the change control procedures applied to controlled software items. Other tools can provide database management and reporting capabilities for management, development, and assurance activities. The capabilities of several tool types might be integrated into SCM systems, which, in turn, are closely coupled to software development activities.
The planning activity assesses the SCM tool needs for a given project within the context of the development 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 [7] and a recent case study on selecting an SCM system is given in [25].

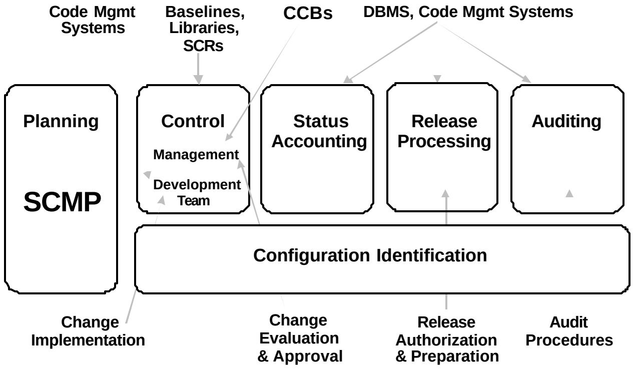


Figure 2. Characterization of SCM Tools and Related Procedures

I.C.4 Vendor/Subcontractor Control

A software development 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.

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.

I.C.5 Interface Control

When a software item developed by a project will interface with another item outside the scope of the software effort, 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.

I.D Software Configuration Management Plan

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.

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 [13]. This reference provides requirements for the information that should be contained in an SCMP. It defines and describes six categories of SCM information that should be included in an SCMP:

1. Introduction (purpose, scope, terms used)
2. SCM Management (organization, responsibilities, applicable policies, directives, and procedures)
3. SCM Activities (configuration identification, configuration control, etc.)
4. SCM Schedules (coordination with other project activities)
5. SCM Resources (tools, physical, and human resources)
6. SCM Plan Maintenance

I.E Surveillance of Software Configuration Management

After the SCM process has been implemented, some degree of surveillance should be conducted to assure that the provisions of the SCMP are properly carried out. This could involve an SCM authority assuring 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.

I.E.1 SCM Metrics

A related goal in monitoring the SCM process is to discover opportunities for process improvement. The software libraries and the various SCM tools 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.

II. Software Configuration Identification

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.

II.A Identifying Items To Be Controlled

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 their relationships, and identifying the baselines to be used, along with the procedure for a baseline’s acquisition of the items.

II.A.1 Software Configuration

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 [12]. It can be viewed as a part of an overall system configuration.

II.A.2 Software Configuration Item

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 [12]. Software items with potential to become SCIs include plans, specifications, test materials, software tools, source and executable code, code libraries, data and data dictionaries, and documentation for maintenance, operations and software use.

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 [2].

II.A.3 Software Configuration Item Relationships

The structural relationships among the selected SCIs, and their constituent parts, affect other SCM tasks and activities, 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 and the need to support the evolution of the software items.

II.A.4 Software Versions

Software items evolve as a software project proceeds. A *version* of a software item is a particular identified and documented item. It can be thought of as a state of an evolving item [6]. A *revision* is a new version intended to replace the old version. 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 development environments is a current research topic [6], [10], and [34].

II.A.5 Baseline

A 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.

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.

II.A.6 Acquiring Software Configuration Items

Different software configuration items are taken 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.

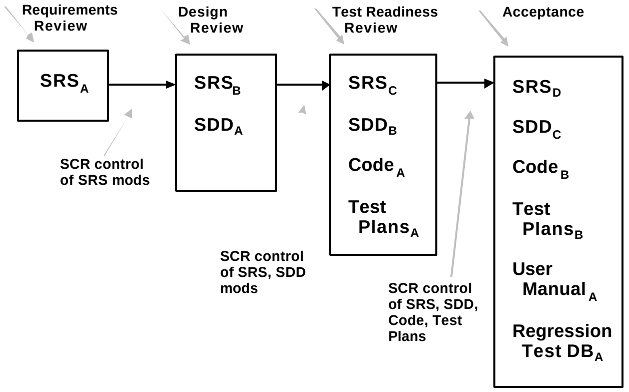


Figure 3. Acquisition of Items

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 baseline according to the appropriate procedure.

II.B Software Library

A software library is a controlled collection of software and related documentation designed to aid in software development, use, and maintenance [12]. 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. The appropriate level of SCM control (associated baseline and level of authority for change) is associated with each library. A model of a software library is described in [2].

II.B.1 SCM Library Tool

The tool(s) used for each library must support the SCM control needs for that library. At the working library level, this is a code management capability serving both developers and SCM. It is focused on managing the versions of software items while supporting the activities of multiple developers. At higher levels, access control is more restricted and SCM is the primary user.

III. Software Configuration Control

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.

III.A. Requesting, Evaluating, and Approving Software Changes

The first step in managing changes to controlled items is determining what changes to make. A software change request process (see Fig. 4) provides formal procedures for submitting and recording change requests, evaluating the potential cost and impact of a proposed change, and accepting or rejecting the proposed change. Requests for changes to software configuration items may be originated by anyone associated with the item at any point in the software life cycle. Once received, a technical evaluation 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 or reject the proposed change.

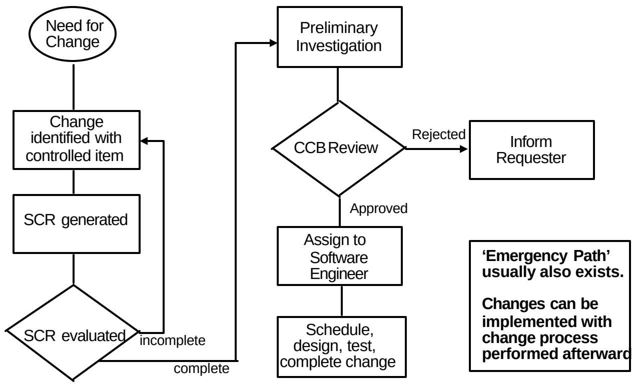

Need for Change
Change identified with controlled item
SCR generated
SCR evaluated
incomplete
complete
Preliminary Investigation
CCB Review
Rejected
Inform Requester
Approved
Assign to Software Engineer
Schedule, design, test, complete change
'Emergency Path' usually also exists.
Changes can be implemented with change process performed afterward

Figure 4. Flow of a Change Control Process

III.A.1. Software Configuration Control Board

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 multiperson board. There can be multiple levels of change authority depending on a variety of criteria, such as the criticality of the item involved, the nature of the change (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. When the scope of authority of a CCB is strictly software, it is known as a software configuration control board (SCCB).

III.A.2 Software Change Management Tool

The software change request process requires the use of supporting tools that can range 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. Change process descriptions and supporting forms (information) are given in a variety of references, e.g. [2] and [14]. 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 development environment (SDE).

III.B. Implementing Software Changes

Approved change requests are implemented according to the defined software development 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. As part of the closure of the change process, completed changes may undergo configuration audits. The change management tool described above will typically reflect the SCM and other approval information for the change.

III.B.1. Software Change Control Tool

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 checkin/out and associated version control. These tools may be manifested as separate entities under control of an independent SCM group, such as Platinum's Harvest or Lucent's Sablime. They may also appear as an integrated part of the SDE, such as, Rational's ClearCase or MicroSoft's SourceSafe products. Finally, they may be as elementary as the original (and still widely used) version of SCCS, which is part of Unix.

III.C. Deviations and Waivers

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.

IV. Software Configuration Status Accounting

Software configuration status accounting (SCSA) is the recording and reporting of information needed for effective management of the software configuration. This can be viewed as an information systems activity and the SCSA capability should be designed as such.

IV.A. Software Configuration Status Information

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 is needed to support the SCM process and to meet the configuration status reporting needs of management, development, 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 [2].

IV.A.1. Software Configuration Status Accounting Tools

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.

IV.B. Software Configuration Status Reporting

Reported information can be used by various organizational and project elements, including the development team, the maintenance team, project management, and company 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.

IV.B.1. Metrics (Measures)

In addition to reporting the current status of the configuration, the information contained in the SCSA database 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.

V. Software Configuration Auditing

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: 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.

V.A. Software Functional Configuration Audit

The purpose of the software FCA is to determine that the performance of 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.

V.B. Software Physical Configuration Audit

The purpose of the software PCA is to determine if the design and reference documentation is consistent with the as-built software product.

V.C. Software In-process Audit

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.

VI. Software Release Management and Delivery

The term “release” is used in this context to refer to the distribution of a software configuration item outside the development organization. This includes 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.

VI.A. Software Building

Software building is the activity of combining the correct versions of software elements, using the appropriate configuration data, into an executable program for delivery to a customer. 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.

Software is built using particular versions of supporting tools, such as compilers. It might be necessary to rebuild an exact copy of a previously delivered software item. In this case, the supporting tools need to be under SCM control to ensure their availability.

VI.A.1 Software Building Tools

A tool capability is needed 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. 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.

VI.B Software Release Management

Software release management encompasses the packaging of the elements of a product for delivery, for example, the executable, documentation, release notes, and configuration data. It may also be necessary to track the distribution of the product to various customers.

VI.B.1 Software Release Management Tool

A tool capability is needed for tracking the full set of elements in a particular release. If necessary, this could also include information on various target platforms and on various customers.

RECOMMENDED REFERENCES FOR SOFTWARE CONFIGURATION MANAGEMENT

3.1 Recommendations

The following set of references was chosen to provide coverage of all aspects of software configuration management from various perspectives and to varying levels of detail. The author and title are cited; the complete reference is given in the References section.

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.

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.

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.

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.

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.

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.

The suite of IEEE/EIA and ISO/IEC 12207 standards, [15]-[18]
These standards cover software life cycle processes and address software configuration management in that context. These standards reflect commonly accepted practice for software life cycle processes.

IEEE Std.828-1990, IEEE Standard for Software Configuration Management Plans [13] and IEEE Std.1042-1987, IEEE Guide to Software Configuration Management [14]
These standards focus on software configuration management activities by specifying requirements and guidance for preparing the SCMP. These standards reflect commonly accepted practice for software configuration management.

A.K. Midha, "Software Configuration Mangement for the 21st Century" [25]
This article discusses the characteristics of SCM systems, assessment of SCM needs in a particular environment, and issue of selecting and implementing an SCM system. It is a current case study on this issue.

J.W. Moore, Software Engineering Standards, A User's Road Map [26]
This text provides a comprehensive view of current standards and standards activities in the area of software engineering.

M.C. Paulk, et al., Key Practices of the Capability Maturity Model [27]
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.

R.S. Pressman, Software Engineering: A Practioner's Approach [31]
This reference addresses SCM in the context of a textbook on software engineering.

I. Sommerville, Software Engineering [33]
This reference addresses SCM in the context of a textbook on software engineering.

J.P. Vincent, et al., Software Quality Assurance [36]
In this text, SCM is described from the perspective of a complete set of assurance processes for a software development project.

D. Whitgift, Methods and Tools for Software Configuration Management [38]
This text covers the concepts and principles of SCM. It is provides detailed information on the practical questions of implementing and using tools. This text is out of print but still available in libraries.

3.2 Coverage of the SCM Breakdown Topics by the Recommended References

This cross-reference is shown in Table 2 of Appendix A.

KNOWLEDGE AREAS OF RELATED DISCIPLINES

The following knowledge areas of related disciplines are important to SCM.

4.1 Computer Science

A. Information Management – Database Models
B. Operating Systems – File Systems, Protection, Security
C. Programming Fundamentals and Skills – Compilers, Code Generation

4.2 Project Management

A. Project Integration Management
B. Project Quality Management
C. Project Risk Management

4.3 Computer Engineering

A. Storage Devices and Systems

4.4 Systems Engineering

A. Process – Requirements Definition, System Definition, Integration, Maintenance & Operations, Configuration Management, Documentation, Systems Quality Analysis and Management, Systems V&V, Systems Evaluation

B. Essential Functional Processes – Development, Test, Distribution, Operations, Support

C. Techniques and Tools – Metrics, Reliability, Security

4.5 Management and Management Science

A. Organizational Environment – Characteristics, Functions, and Dynamics

B. Information Systems Management – Data Resource Management

4.6 Cognitive Science and Human Factors

A. Development Process – Design Approaches, Implementation Techniques, Evaluation Techniques

REFERENCES

1. W.A. Babich, Software Configuration Management: Coordination for Team Productivity, Addison-Wesley, Reading, Massachusetts, 1986.

2. H.R. Berlack, Software Configuration Management, John Wiley & Sons, New York, 1992.

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.

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.

5. F.J. Buckley, Implementing Configuration Management: Hardware, Software, and Firmware, Second Edition, IEEE Computer Society Press, Los Alamitos, CA, 1996.

6. R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," ACM Computing Surveys, Vol. 30, No. 2, June 1998, pp. 232-282.

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.

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.

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.

10. J. Estublier, Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops Selected Papers, Springer-Verlag, Berlin, 1995.

11. P.H. Feiler, Configuration Management Models in Commercial Environments, Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon University, 1991.

12. IEEE Std.610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE, Piscataway, NJ, 1990.

13. IEEE Std.828-1990, IEEE Standard for Software Configuration Management Plans, IEEE, Piscataway, NJ, 1990.

14. IEEE Std.1042-1987, IEEE Guide to Software Configuration Management, IEEE, Piscataway, NJ, 1990.

15. IEEE/EIA Std 12207.0-1996, Software Life Cycle Processes, IEEE, Piscataway, NJ, 1996.

16. IEEE/EIA Std 12207.1-1996, Guide for Software Life Cycle Processes – Life Cycle Data, IEEE, Piscataway, NJ, 1996.

17. IEEE/EIA Std 12207.1-1996, Guide for Software Life Cycle Processes – Implementation Considerations, IEEE, Piscataway, NJ, 1996.

18. ISO/IEC 12207:1995(E), Information Technology - Software Life Cycle Processes, ISO/IEC, Geneve, Switzerland, 1995.

19. ISO/IEC 12207:1995(E) (Committee Draft), Information Technology - Software Life Cycle Processes: Part 2: Configuration Management for Software, ISO/IEC, Geneve, Switzerland, 1995.

20. ISO/DIS 9004-7 (now ISO 10007), Quality Management and Quality System Elements, Guidelines for Configuration Management, International Organization for Standardization, Geneve, Switzerland, 1993.

21. P. Jalote, An Integrated Approach to Software Engineering, Springer-Verlag, New York, 1997

22. John J. Marciniak and Donald J. Reifer, Software Acquisition Management, Managing the Acquisition of Custom Software Systems, John Wiley & Sons, 1990.

23. J.J. Marciniak, "Reviews and Audits," Software Engineering, M. Dorfman and R.H. Thayer ed., IEEE Computer Society Press, Los Alamitos, CA, 1997.

24. K. Meiser, "Software Configuration Management Terminology," Crosstalk, 1995,

http://www.stsc.hill.af.mil/crosstalk/1995/jan/terms.html, February 1999.

25. A.K. Midha, "Software Configuration Management for the 21st Century," Bell Labs Technical Journal, Winter 1997.

26. J.W. Moore, Software Engineering Standards, A User's Roadmap, IEEE Computer Society, Los Alamitos, CA, 1998.

27. 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

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

29. S.L. Pfleeger, Software Engineering: Theory and Practice, Prentice Hall, Upper Saddle River, NJ, 1998

30. R.K. Port, "Software Configuration Management Technology Report, September 1994, " http://www.stsc.hill.af.mil/cm/REPORT.html, February 1999.

31. R.S. Pressman, Software Engineering: A Practioner's Approach, McGraw-Hill, New York, 1997.

32. Walker Royce, Software Project Management, A United Framework, Addison-Wesley, Reading, Massachusetts, 1998.

33. I. Sommerville, Software Engineering, Fifth Edition, Addison-Wesley, Reading, Massachusetts, 1995.

34. I. Sommerville, Software Configuration Management, ICSE SCM-6 Workshop, Selected Papers, Springer-Verlag, Berlin, 1996.

35. 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.

36. J.P. Vincent, et al., Software Quality Assurance, Prentice-Hall, Englewood Cliffs, NJ, 1988.

37. W.G. Vincenti, What Engineers Know and How They Know It, The Johns Hopkins University Press, Baltimore, MD, 1990.

38. D. Whitgift, Methods and Tools for Software Configuration Management, John Wiley & Sons, Chichester, England, 1991.

APPENDIX A

Table 1. Summary of the SCM Breakdown

- I. Management of the SCM Process
  - A. Organizational Context for SCM
  - B. Constraints and Guidance for SCM
  - C. Planning for SCM
    - 1. SCM Organization and Responsibilities
    - 2. SCM Resources and Schedules
    - 3. Tool Selection and Implementation
    - 4. Vendor/Subcontractor Control
    - 5. Interface Control
  - D. Software Configuration Management Plan
  - E. Surveillance of SCM
    - 1. SCM Metrics
- II. Software Configuration Identification
  - A. Identifying Items to be controlled
    - 1. Software Configuration
    - 2. Software Configuration Item
    - 3. Software Configuration Item Relationships
    - 4. Software Versions
    - 5. Baseline
    - 6. Acquiring Software Configuration Items
  - B. Software Library
    - 1. SCM Library Tool
- III. Software Configuration Control
  - A. Requesting, Evaluating, and Approving Software Changes
    - 1. Software Configuration Control Board
    - 2. Software Change Management Tool
  - B. Implementing Software Changes
    - 1. Software Change Control Tool
  - C. Deviations & Waivers
- IV. Software Configuration Status Accounting
  - A. Software Configuration Status Information
    - 1. Software Configuration Status Accounting tools
  - B. Software Configuration Status Reporting
    - 1. Metrics (Measures)
- V. Software Configuration Auditing
  - A. Software Functional Configuration Audit
  - B. Software Physical Configuration Audit
  - C. Software In-process Audit
- VI. Software Release Management and Delivery
  - A. Software Building
    - 1. Software Building Tools
  - B. Software Release Management
    - 1. Software Release Management Tool

Table 2. Coverage of the Breakdown Topics by the Recommended References

	Babich	Berlack	Buckley	Conradi	Dart	Estublier (SCM 4/5)	IEEE 828/1042	ISO/IEC 12207	Midha	Moore	Paulk	Pressman	Sommerville	Vincent	Whitgift
I. Management		X			X		X					X			X
I.A. Org. Context		X	X		X		X				x		X		X
I.B Constraints		X					X			X		X	X		X
I.C Planning		X	X		X		X						X		X
I.C.1 Org. & Resp.		X	X				X	X							
I.C.2 Resources & Sched.		X	X				X	X			X				X
I.C.3 Tool Selection		X	X	X	X	X	X		X			X	X		X
I.C.4 Vendor Control		X	X				X						X		X
I.C.5 Interface Control		X	X				X								
I.D SCM Plan		X	X				X	X			X		X		X
I.E Surveillance			X				X				X				
I.E.1 Metrics											X				
II. SW Config Identification		X	X		X		X	X			X		X	X	X
II.A Identifying Items		X	X				X	X			X				
II.A.1 SW Configuration		X	X				X					X			X
II.A.2 SW Config. Item		X	X	X			X				X	X	X	X	X
II.A.3 SCI Relationships		X		X			X					X			X
II.A.4 Software Versions	X			X		X						X			
II.A.5 Baselines	X	X	X				X	X				X		X	
II.A.6 Acquiring SCIs			X				X								
II. Software Library	X	X	X	X			X		X		X	X		X	X
II.B.1 SCM Library Tool	X			X	X	X			X			X	X		X

Table 2. Coverage of the Breakdown Topics by the Recommended References (cont.)

	Babich	Berlack	Buckley	Conradi	Dart	Estublier (SCM 4/5)	IEEE 828/1042	ISO/IEC 12207	Midha	Moore	Paulk	Pressman	Sommerville	Vincent	Whitgift
III. SW Configuration Control		X	X		X		X	X	X		X	X	X	X	X
III.A Requesting Changes		X	X				X	X	X		X	X	X	X	X
III.A.1 SW CCB		X	X				X				X	X	X	X	X
III.A.2 Change Mgmt Tool					X				X				X		X
III.B Implementing Changes			X				X	X	X		X	X	X	X	X
III.B.1 Change Cntl Tool	X			X	X	X			X				X		X
III.C Deviations & Waivers		X	X												
IV. SW Config Status Acctg		X	X		X		X	X			X			X	
IV.A. Status Information		X	X				X	X			X			X	
IV.A. CSA Tool															
IV.B Status Reporting		X	X				X	X			X	X		X	
IV.B Metrics															
V. SW Configuration Audit		X	X		X		X	X			X	X		X	
V.A Functional Config Audit		X	X				X	X						X	
V.B Physical Config Audit		X	X				X	X						X	
V.C In-Process Audit			X				X				X			X	
VI. SW Release Mgmt & Del								X					X		X
VI.A SW System Building						X			X				X		X
VI.A.1 SW Build Tools	X			X									X		X
VI.B SW Release Mgmt											X		X		X
VI.B.1 SW Release Tool													X		

APPENDIX B. BREAKDOWN RATIONALE

Criterion (a): Number of topic breakdowns

One breakdown is provided.

Criterion (b): Reasonableness

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. The topics dealing with types of tools are appropriate but may not have enough supporting material to stand by themselves. In this case, the discussion would be incorporated into the higher level topic or into the tool discussion in the management of SCM section.

Criterion (c): Generally Accepted

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

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 (my 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."

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.

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.

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.

Criterion (d): No Specific Application Domains

No specific application domains have been assumed.

Criterion (e): Compatible with Various Schools of Thought

SCM concepts are fairly stable and mature.

Criterion (f): Compatible with Industry, Literature, and Standards

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.

Criterion (g): As Inclusive as Possible

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.

Criterion (h): Themes of Quality, Tools, Measurement, and Standards

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.

A number of level 3 topics were included throughout the breakdown in order to call attention to the types of tool capabilities that 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.

A similar approach was taken toward the use of measures.

Standards are explicitly included in the breakdowns.

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

The proposed breakdown satisfies this criterion.

Criterion (j): Topic Names Meaningful Outside the Guide

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.

Criterion (k): Vincenti Breakdown

TBD

Criterion (l): Version 0.1 of the Description

TBD

Criterion (m): Text on the Rationale Underlying the Proposed Breakdowns

This document provides the rationale.

APPENDIX C. BLOOM'S TAXONOMY

This appendix contains a table identifying the SCM topic areas and the associated Bloom's taxonomy level of understanding on each topic. The levels of understanding from lower to higher are: knowledge, comprehension, application, analysis, synthesis, and evaluation.

SCM TOPIC	Bloom Level
I. Management of the SCM Process	Knowledge
A. Organizational Context for SCM	Knowledge
B. Constraints and Guidance for SCM	Knowledge
C. Planning for SCM	Knowledge
1. SCM Organization and Responsibilities	Knowledge
2. SCM Resources and Schedules	Comprehension
3. Tool Selection and Implementation	Knowledge
4. Vendor/Subcontractor Control	Knowledge
5. Interface Control	Comprehension
D. Software Configuration Management Plan	Knowledge
E. Surveillance of SCM	Comprehension
1. SCM Metrics	Comprehension
II. Software Configuration Identification	Comprehension
A. Identifying Items to be controlled	Comprehension
1. Software Configuration	Comprehension
2. Software Configuration Item	Comprehension
3. Software configuration item relationships	Comprehension
4. Software Versions	Comprehension
5. Baselines	Comprehension
6. Acquiring Software Configuration Items	Knowledge
B. SCM Library	Comprehension
1. SCM Library Tool (Version Management)	Comprehension
III. Software Configuration Control	Application
A. Requesting, Evaluating, and Approving Software	Application
1. Software Configuration Control Board	Application
2. Software Change Management Tool	Application
B. Implementing Software Changes	Application
1. Software Change Control Tool	Application
C. Deviations & Waivers	Comprehension
IV. Software Configuration Status Accounting	Comprehension
A. Software Configuration Status Information	Comprehension
1. Software Configuration Status Accounting	Knowledge
B. Software Configuration Status Reporting	Comprehension
1. Metrics (Measures)	Comprehension
V. Software Configuration Auditing	Knowledge
A. Software Functional Configuration Audit	Knowledge
B. Software Physical Configuration Audit	Knowledge
C. Software In-process Audit	Knowledge
VI. Software Release Management & Delivery	Comprehension
A. Software Building	Comprehension
1. Software Building Tools	Application
B. Software Release Management	Comprehension
1. Software Release Management Tool	Comprehension

APPENDIX D. VINCENTI CATEGORIZATION

TBD

Software Construction (Version 0.5)

Terry Bollinger

Software Construction

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 at all 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.

Software Construction and Software Design

To fully understand the role of software construction in software engineering, it is important to understand its relationship to the related engineering concept of *software design.* The relationship is closer than it might seem: Software design is simply what happens when software construction is “parceled out” to more than one developer, usually for reasons of scale or schedule constraints. Since humans are notoriously bad at communicating with each other efficiently and without quarrelling, this act of parceling out construction tasks requires the use of procedural and automated enforcement to ensure that programmers “behave” over the course of completing their tasks. The act of parceling out software construction to create a multi-person software design activity also unavoidably entails a massive drop in productivity, since it is far more difficult for such a group to remain “sane” from the perspective of a literal computer that must run their integrated results.

In practice, nearly all of the skills required in “software design” are also a key part of software construction, as demonstrated by the accomplishments of many individual programmers who have created powerful and well-designed software by directly applying design and construction methods. The idea that all of the hard problems are solved during “design” can lead directly to the common but highly inaccurate impression that software construction is nothing more than the “mechanistic” translation of software designs into actual software. If this were actually the case, the commercial market would long ago have moved entirely to abstract design tools that did in fact automate the final coding steps out of existence. While progress has indeed been made on this front for some specialized styles of software construction, the actual consequence of such automation is simply to move programming-style activities up to a new and more powerful level of capability. Furthermore, the idea that significant levels of repetitive mechanical translation by people should be tolerated in any strong software engineering process is inherently absurd, as it attempts to make costly and error-prone humans perform the very types of tasks for which the computer is vastly better suited. Sadly, project politics also play in some perceptions of the relative importance and complexity of the design and construction efforts. While it is not at all uncommon for experienced programmers to massively repair, replace, or expand a documented design as the "details" of the implementation become more apparent, it is seldom to their benefit politically to shout too loudly about the naivete or inaccuracy of original design. As a result, the software design summary reports and metrics of projects seldom reflect the exact dynamics of what happened between design and construction, and in more than a few cases they may be vastly and unfairly biased towards touting the results of the "design" phase over construction.

The Spectrum of Construction Techniques

Software construction techniques can be broadly grouped in terms of how they fall between two endpoints: computer-intensive construction techniques, and human-intensive construction ones.

Computer-Intensive Construction	←	**Human-Intensive Construction**
Task: Configure limited sets of options	←	*Task:* Configure nearly unlimited sets of options
Description: Most knowledge is "in" the computer	←	*Description:* Most knowledge is "in" the programmer
Goal: Move tasks into this category	←	*Goal:* Move tasks out of this category

Computer-Intensive Construction

In the simplest forms of software construction the computer has been previously programmed to take care of the vast majority of algorithmic steps, and the human then takes the simple role of "configuring" software to the special needs of an organization of people and other systems. While this type of construction is far less taxing on the human constructors, it still requires skills such as communication with an organization that are far beyond the capabilities of an unassisted computer, and thus is unavoidably an example of communicating intent to a computer – that is, of software construction. More important, however, is that this simple style of construction should in many cases be the goal to which all construction methods should aspire, since it is vastly more reliable and efficient than nearly any other type of communication-of-intent interface to a computer. Once implemented, configuration-style software construction "swallows" huge chunks of the overall software engineering process and replaces them with a more localized process that is controlled by the computer. It also allows the overall software engineering process to refocus on new and more complex problems that have not yet been so thoroughly automated.

Human-Intensive Construction

At the other extreme of software construction is the development of new and creative algorithms and software mechanisms, in which the programmer must wrestle with unsolved problems or the first-time automation of highly complex processes. One example of this type of programming would be creating a driver for an entirely new class of software peripheral. Human-intensive software construction requires the use of people who are capable of “thinking like computers,” and thus are able to provide the computer with the kind of expertise that makes the computer appear to be “smarter” than it was before. This end of the construction spectrum is close to the traditional concept of “coding,” although in terms of its burden on the programmer it bears very little resemblance to the idea that programming is simply “translating” higher-level designs into code. On the contrary, this end of the software construction spectrum requires a rich mix of skills including creative problem analysis, disciplined formal-proof-like logical and mathematical expression of solutions, and the ability to “forecast” how the resulting software construction will change over time. The best constructors in this domain not only have much the same skills as advanced logicians, but also the ability to use those skills in a much more difficult environment.

As indicated by the left arrows and goals in the table, all software construction methods should as a general goal work to move tasks away from the human-intensive side and into the computer-intensive side. The rationale for this is straightforward: People are too expensive and error-prone to waste in re-solving the same problems repeatedly. However, moving construction processes to the left can be difficult to accomplish in practice, especially if large numbers of developers are involved or the overall development process does not recognize the need for automation. Even so, the concept of moving towards higher levels of automated, computer-intensive software construction is sufficiently fundamental to good design that it permeates nearly every aspect of the software construction problem. For example, even as simple a concept 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.

Computer Languages

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 would be expressed in the form of 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 nit-picky 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 with a natural language.

Construction Languages

Construction languages are a broader set of languages that include not only computer languages such as C and Java, but also pre-constructed sets of parts and terms that have been designed for use together in constructing new software for some well-defined set of problems. Thus the example of sets of simple software configuration options mentioned earlier can also be understood as constituting a small but well-defined construction language for the problem domain of how to configure that software. The concept of construction languages is useful for emphasizing issues such as the need to transition over time from human-intensive computer languages to computer-intensive ones in which the required construction languages are simpler and more problem-focused. In practice the creation and use of specialized construction languages usually begins very early in the construction process, even for one-person efforts. This occurs in part because the creation of simpler, more automated sets of components is an important tool for controlling complexity during the construction process.

Styles of Construction

A good construction language moves detailed, repetitive, or memory-intensive construction tasks away from people and into the computer, where they 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:

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 their imprecision.

B. **Mathematical:** The precision and rigor of mathematical 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, mathematical 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 mathematical 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, mathematical 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.

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.

Construction languages seldom rely solely on a single style of construction. Linguistic and mathematical 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 visual 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 mathematical styles within the same construction language.

Principles of Organization

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:

1. **Reduction of Complexity:** This principle of organization reflects the relatively limited (when compared to computers) ability of people to work with complex systems with many parts or interactions.

2. **Anticipation of Diversity:** The motive behind this principle 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.

3. **Structuring for Validation** No matter how carefully a person designs and implements software, the creative nature of non-trivial software design (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 requires will require software changes (bug fixes).

4. 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.

These principles of organization in software construction are discussed in more detail below.

Reduction in Complexity

Another 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. A human-to-computer construction that does not shield people from the nearly unlimited complexity and retention possible within a computer can easily overload the people working with it, potentially leading to serious inefficiencies and a proliferation of errors during the construction process. 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.

There are three main techniques for reducing complexity during software construction:

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.

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 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 features or 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.

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. (A contrasting example: 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. 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.

Anticipation of Diversity

This principle has more to do with how people use software than with differences between computers and people. As stated earlier, the main idea is quite simple: *There is no such thing as an unchanging 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, mathematical constructions and formulae 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.

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.

Structuring for Validation

It is not particularly difficult to write software that cannot really be validated not 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 millennia or longer 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. One of the most important implications of this principle is that software must be *modular* in some fashion that allows its behavior to be thoroughly analyzed and validated through testing before it becomes so complex that such validation is no longer feasible. 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.

Use of 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. 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.

A Taxonomy of Software Construction Methods

The following taxonomy uses the above breakdowns of the software construction problem to suggest categories of tools and techniques needed for effective and well-rounded software engineering during the construction phase. Rather than being delayed until the construction phase itself, decisions about what styles of construction and specific methods will be used should be made early in the engineering process, since they will affect other phases as well.

A. Linguistic Construction Methods

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.

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.

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.

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.

1. *Reduction in Complexity (Linguistic)*

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:

Functions, procedures, and code blocks

Software templates

Design patterns

Component libraries and frameworks

Interrupt and event-driven programming

Data structures

Encapsulation and abstract data types

Higher-level and domain-specific languages

2. *Anticipation of Diversity (Linguistic)*

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.

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

Self-describing software and hardware

3. *Structuring for Validation (Linguistic)*

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 the free use of 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.

Modular design

Structured programming

Style guides

Unit testing

4. *Use of External Standards (Linguistic)*

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.

Standardized programming languages

Standardized data description languages (e.g., XML)

Standardized alphabet representations (e.g., Unicode)

Inter-process communication standards (e.g., COM, CORBA)

Component-based software

B. Mathematical Construction Methods

Mathematical 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 (mathematical) definitions. Mathematical 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. Mathematical constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions. Functions are an obvious example of mathematical constructions, with their direct parallel to mathematical functions in both form and meaning.

Mathematical 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.

The importance of the mathematical style of programming cannot be understated. Just as the precision of mathematics is fundamental to disciplines such as physics and the hard science, the mathematical 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. Mathematical 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 mathematical style of construction.

1. *Reduction in Complexity (Mathematical)*

   As is the case with linguistic construction methods, mathematical 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 mathematical notations, rather than the “fuzzier” rules of natural language. Reading and writing such expressions and constructs them generally more training, but once mastered, the use of mathematical constructions tends to keep the ambiguity of what is being specified to an absolute minimum. However, as with linguistic construction, the quality of a mathematical construction is only as good as its underlying implementation. The advantage is that the precision of the mathematical definitions usually translates into a more precise specification for the software beneath it.

   Traditional functions and procedures

   Functional programming

   Logic programming

   Concurrent and real-time programming techniques

   Spreadsheets

   Mathematical libraries of functions

2. *Anticipation of Diversity (Mathematical)*

   Diversity in mathematical 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 mathematical 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 indefinitely variable-length vector. Since more generic solutions are often harder to implement and harder to make efficient, it is important when using mathematical construction techniques to try to anticipate the full range of future versions.

Functional parameterization

Macro parameterization

Extensible mathematical frameworks

3. *Structuring for Validation (Mathematical)*

Since mathematics in general is oriented towards proof of hypothesis from a set of axioms, mathematical 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.

Assertion-based programming (static and dynamic)

State machine logic

Redundant systems, self-diagnosis, and failover methods

Hot-spot analysis and performance tuning

4. *Use of External Standards*

For mathematical construction techniques, external standards generally address ways to define precise interfaces and communication methods between software systems and the machines they reside on.

POSIX standards

Data communication standards

Hardware interface standards

Standardized mathematical representation languages (e.g., MathML)

Mathematical libraries of functions

C. Visual Construction Methods

Visual construction methods rely much less on the text-oriented constructions of both linguistic and mathematical 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.

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 Mapping Language, or UML) actively encourage this highly visual style of reasoning.

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 mathematical 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 mathematical 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.

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 mathematically 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 mathematical construction, since the screen objects are much more tightly constrained in properties.

1. *Reduction in Complexity (Visual)*

   Especially when compared to the steps needed to build a visual interface using text-oriented linguistic or mathematical construction, visual construction can provide drastic reductions in the total effort required to building a working graphical interface to a program. It can also reduce complexity by providing a simple way to select between the elements of a small set of choices.

   Object-oriented programming

   Visual creation and customization of user interfaces

   Visual (e.g., visual C++) programming

   "Style" aspects of structured programming

2. *Anticipation of Diversity (Visual)*

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.

Object classes

Visual configuration specification

Separation of GUI design and functionality implementation

3. *Structuring for Validation (Visual)*

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.

"Complete and sufficient" design of object-oriented class methods

Dynamic validation of visual requests in visual languages

4. *Use of External Standards (Visual)*

Standards for visual interfaces greatly ease the total burden on users by providing familiar, easily understood "look and feel" interfaces for those users.

Object-oriented language standards

Standardized visual interface models (e.g., Microsoft Windows)

Standardized screen widgets

Visual Markup Languages

References

1. Bentley, Jon, *Programming Pearls.* Addison-Wesley, 1989.

2. McConnell, Steve, *Code Complete.* Microsoft Press, 1993.

3. Henricson, Mats, and Nyquist, Erik, *Industrial Strength C++*. Prentice-Hall, 1997.

4. Brooks, Frederick P. Jr., *The Mythical Man-Month.* Addison-Wesley, 1995.

Very important note from the Knowledge Area specialist and the editors. Suggestions for book and article references are very much welcome for this Knowledge Area Description. Here is a reminder of the specifications regarding references:

1.1 Criteria and Requirements for selecting Reference Material

a) Specific reference material must be identified for each topic. Each piece of reference material can, of course, cover multiple topics.

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. They must be generally available and cannot be confidential in nature.

c) Proposed Reference Materials must be in English.

d) A maximum of 15 Reference Materials can be recommended for each Knowledge Area.

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 to the ACM. This should not, however, be seen as a constraint or an obligation.

f) A matrix of reference material versus topics must be provided.

In order to facilitate your contribution to this section, here is table containing the breakdown of topics suggested above by the Knowledge Area Specialist. Please make your suggestions of reference material which cover one or more of the following topics, while respecting the above specifications.

We do not expect you to make reference material suggestions for each topic, especially if you don’t agree with the breakdown or some part of it. If you suggest other topics, please include relevant material suggestions for these new topics.

Topics	Proposed reference material
A. Linguistic Construction Methods	
1. Reduction in Complexity (Linguistic)	
2. Anticipation of Diversity (Linguistic)	
3. Structuring for Validation (Linguistic)	
4. Use of External Standards (Linguistic)	
B. Mathematical Construction Methods	
1. Reduction in Complexity (Mathematical)	
2. Anticipation of Diversity (Mathematical)	
3. Structuring for Validation (Mathematical)	
4. Use of External Standards (Mathematical)	
C. Visual Construction Methods	
1. Reduction in Complexity (Visual)	
2. Anticipation of Diversity (Visual)	
3. Structuring for Validation (Visual)	
4. Use of External Standards (Visual)	

Knowledge Area Description for Design (version 0.5)

Guy Tremblay
Dépt. d'Informatique, UQAM
Montréal, Qué.

Overview and rationale

As demanded in the specifications document, I have tried to be as inclusive as possible and, also, to include topics related with quality, measurements and standards. Thus, certain topics have been included in the list of topics even though they may not yet be fully considered as generally accepted (e.g., design metrics, which are formally discussed in a single software engineering book). As for the topic of standards, it was not clear how to include it, especially considering the relatively few number of such standards. Thus, a special "Existing standards" topic section has been created.

Some topics have also been included because it was asked to include topics which might be considered generally accepted within a 3-5 years time frame. In this spirit, I have included the following topics: i) elements related with software architecture ("Software Architecture", Shaw and Garlan, 1996; "Software Architecture in Practice", Bass, Clements and Kazman, 1998), including notions related with architectural styles; ii) the approach to OOD recently presented by the UML group ("The Unified Software Development Process", Jacobson, Booch and Rumbaugh, 1999). Note that although UML is not explicitly mentioned in the design notations, many of its elements are indeed present, for example: class diagrams, collaboration diagrams, deployment diagrams, object diagrams, sequence diagrams, statecharts. Note that no special section explicitly related with tools has been included as it was not clear what should be included in relation with design.

After receiving the comments from the reviewers of version 0.1, a number of modifications have been done and a single breakdown of topics has been selected. This breakdown takes the approach taken in a number of software engineering books, where the notations are presented apart from the strategies and methods that use those notations. Also, in the current breakdown of topics, the concepts related with the topics of architecture and standards have been given explicit sections, whereas the topics related with quality and metrics are grouped in a single section (because metrics can bee seen as tools to evaluate the quality of a design). Finally, the section on strategies and methods has been divided, as done in many references, in a first section that presents general strategies followed by subsequent sections that present the various classes of approaches (data- vs. function- vs. object-oriented). Note that although numerous approaches to OOD have been presented in the literature, we mention only a few, as the one recently proposed by the UML group can be considered a fusion of many existing approaches (Booch, Objectory, OMT).

Before presenting the proposed detailed breakdown of topics, let us present an overview of the major categories accompanied by a brief description.

Basic concepts and principles

The basic concepts and principles which are generally considered fundamental to software design. These are key notions that often reappear, in various forms, in the subsequent topics. They are like a foundation for understanding design.

Design quality and metrics

Quality attributes

The attributes that are generally considered important for obtaining a design of good quality. These include both attributes related with the external behavior of the software (e.g., availability, usability) or related with the quality of the internal structure of the system (e.g., maintainability, simplicity).

Quality assurance

General QA techniques that can be used to ensure the good quality of a design.

Metrics

Formal metrics that can be used to estimate various aspects of the quality of a design. Although some metrics are general, some depend on the approach used for producing the design, for example, structured vs. object-oriented design.

Software architecture

Structures and viewpoints

Various authors have different views of the differents high-level facets of a design that should be described and documented. Some of these views are presented.

Architectural styles

The notion of architectural style -- a paradigmatic architectural pattern that can be used to obtain a high-level organization of software -- has become an important notion of the field of software architecture. This section presents some of the major styles which have been identified by various authors.

Architectural descriptions

The techniques and/or notations that can be used to describe a high-level architecture, either formally (e.g., formal architecture description language) or informally (e.g., patterns). Note that these techniques generally differ from the ones used in a specific architectural design document.

Object-Oriented Frameworks: Description of the notion of framework, which has become important in the context of OO programming.

Design notations

Architectural design

Various notations that can be used to document a specific architectural design.

Detailed design

Various notations that can be used to produce the detailed design of a system.

Design strategies and methods

General strategies

General classes of strategies that can be used to create a design for a system.

Data-structure centered design

Although less popular in North America than in Europe, the work on data-structured design (mostly M. Jackson’s work) seems worthy of mention.

Function-oriented design

The classical approach of function/structured design (à la Yourdon/Constantine).

Object-oriented design

Some of the strategies and methods that have been proposed for OO design, including the recently proposed Unified Software Development Process approach.

Detailed breakdown of topics

Basic concepts and principles

Abstraction

Cohesion and coupling

Decomposition

Encapsulation

Information hiding

Interface vs. Implementation

Modularity

Partitioning

Refinement

Separation of concerns

Structure

Design quality and metrics

Quality attributes
- Availability
- Correctness
- Efficiency
- Integrability
- Interoperability
- Maintainability
- Modifiability
- Portability
- Reliability
- Reusability
- Security
- Simplicity
- Testability
- Usability

Quality assurance
- Preliminary design reviews and/or design walkthroughs
- Critical design reviews
- Program design reviews

Metrics
- Graph/cyclomatic complexity metrics
- Structured design metrics
- Object-oriented design metrics

Software architecture

Structures and viewpoints
- Behavioural vs. functional vs. structural vs. data modeling
- Conceptual vs. module vs. code vs. architecture
- Logical vs. process vs. physical vs. development

Architectural styles
- Batch systems
- Blackboards
- Data abstraction and OO organization
- Distributed systems (e.g., CORBA, DCOM)
- Event-based systems
- Interactive systems
- Interpreters
- Layered systems
- Pipes and filters
- Process control
- Repositories
- Rule-based systems

Architectural descriptions
- Module Interconnection Languages
- Architecture description languages
- Patterns

Object-Oriented Frameworks

Design notations

Architectural design
- Class diagrams
- CRC (Class-Responsibilities-Collaborators) Cards
- Deployment and processes diagrams
- Module diagrams
- Structure charts
- Subsystems diagrams

Detailed design
- Collaboration diagrams
- Decision tables and diagrams
- Flowcharts and structured (box) flowcharts
- HIPO diagrams (Hierarchical Input-Process-Output)
- Jackson structure diagrams
- Pseudo-code and PDL (Program Design Language)
- Sequence diagrams
- State transition diagrams and statecharts

Design strategies and methods

General strategies
- Data-oriented decomposition
- Functional decomposition
- Information-hiding
- Object-oriented design
- Stepwise refinement

Data-structure centered design
- Jackson Structured Programming (JSP)
- Jackson System Development (JSD)

Function-oriented design
- SSADM
- Structured design

Object-oriented design
- Coad and Yourdon method
- Fusion
- Meyer’s Design By Contract
- Shlaer/Mellor Method
- The Unified Software Development Process’ approach to design
- Wirfs-Brock et al.’s Responsibility-driven design

Existing standards

IEEE Standard Glossary of Software Engineering Terminology

IEEE Recommended Practice for Ada as a Program Design Language

IEEE Recommended Practice for Software Design Descriptions

IEEE Guide to Software Design Descriptions

Proposed reference material

In what follows, we suggest reference material for the various topics presented in the proposed breakdown of topics. 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.

Also note that the number of references is slightly above the suggested 15 references indicated in the specification document for the knowledge area descriptions. In our case, this number is both only slightly above (17) and largerly above. Exactly 17 references have been indicated, of which four (4) are references to existing IEEE Standards. These latter were included for completeness but not all of them are directly relevant (e.g., IEE87a, IEE91). However, if complete and detailed references had been given, the number of references would have been *largely* above the suggested 15: in 3 cases, the indicated references are to publications from the IEEE Computer Society Press which are in fact collections of papers previously published elsewhere. These references are used mainly for the topics related with the various design notations and methods.

<u>Basic concepts and principles</u>

[BMR+96, chap. 6]
[Jal97, chap. 5]
[Pfl98, chap. 5]
[Pre92, chaps. 13 and 23]
[YC79]

<u>Design quality and metrics</u>

Quality attributes
[BCK98, chap. 4]
[BMR+96, chap. 6]
[Jal97, chap. 5]
[IEE91]

Quality assurance
[BCK98, chap. 10]
[Jal97, chap. 7]
[Pfl98, chap. 5]

Metrics
[Jal97, chaps. 5, 6 and 7]
[Pre98, chaps. 18 and 23]

<u>Software architecture</u>

Structures and viewpoints
[BCK98, chap. 2]
[BMR+96, chap. 6]
[Pfl98, chap. 5]

Architectural styles
[BCK98, chap. 2]
[BMR_96, chap. 2]
[Pfl98, chap. 5]
[SG96, chaps. 2 and 4]

Architectural descriptions
Module Interconnection Languages [FW83, part V][SG96, chap. 7]
Architecture description languages [BCK98, chap. 12][SG96, chaps. 6 and 7]
Patterns [BCK98, chap. 12][BMR+96][Pre92, chap. 21]

Object-Oriented Frameworks [BMR+96, chap. 6]

Design notations

Architectural design
[DT97, chap. 4]
[Jal97, chaps. 5 and 6]
[JBR99, chap. 9]
[Pre92, chap. 14]
[TM93, chap. 5]
[WBWW90]
[YC79]

Detailed design
[DT97, chap. 4]
[FW, parts III and VII]
[JBR99, chap. 9]
[Pre92, chap. 14]

Design strategies and methods

General strategies
Data-oriented decomposition [FW83, part V]
Functional decomposition [FW83, part V][Pre92, chaps. 13 and 14]
Information-hiding [FW83, part V][Pre92, chap. 13]
Object-oriented design [FW83, part VI][Pre92, chaps. 19 and 21]
Stepwise refinement [FW83, part VII]

Data-structure centered design
Jackson Structured Programming (JSP) [DT97, chap. 4][FW83, part VII][TM93, chap. 5]
Jackson System Development (JSD) [DT97, chap. 4][FW83, part III][TM93, chap. 5]

Function-oriented design
SSADM [TM93, chap. 5]
Structured design [Jal97, chap. 5][Pre92, chap. 14][YC79]

Object-oriented design [Hut94][Jal97, chap. 6][Pre92, chap. 21]
Coad and Yourdon method [Pre92, chap. 21][Hut94]
Fusion [Hut94]
Meyer’s Design By Contract [Hut94][Pfl98, chap. 5]
Shlaer/Mellor Method [Hut94]
The Unified Software Development Process’ approach to design [JBR99]
Wirfs-Brock et al.’s Responsibility-driven design [Hut94][Pre92, chap. 21][WBWW90]

Existing standards

IEEE Standard Glossary of Software Engineering Terminology [IEE91]

IEEE Recommended Practice for Ada as a Program Design Language [IEE87a]

IEEE Recommended Practice for Software Design Descriptions [IEE87b]

IEEE Guide to Software Design Descriptions [IEE93a]

Matrix of topics vs. reference material

Note: A number in the matrix entry indicates the appropriate chapter number. A "*" indicates a general reference (no specific chapter).

	BCK 98	BMR +96	DT 97	FW 83	Hut 94	Jal 97	JBR99	Pfl 98	Pre 92	SG 96	TM 93	WBWW90	YC 79
Basic conc. and princ.		**6**				**5,6**		**5**	**13,23**				
Abstraction		6				5		5	13,23				
Cohes. and coupl.		6				5,6		5	13				
Decomposition								5					
Encapsulation		6				6			23				
Info. hiding		6				6		5	13,23				
Interf. vs. Impl.		6											
Modularity		6				5		5	13				
Partitioning						5			13				
Refinement						5			13				
Sep. of concerns		6											
Structure									13				
Design qual. and m.													
Quality attr.	4	6				5				7			
Availability	4												
Correctness						5							
Efficiency	4	6				5							
Integrability	4												
Interoperability		6											
Maintainability													
Modifiability	4	6											
Portability	4												
Reliability		6											
Reusability	4	6								7			
Security	4												
Simplicity						5							
Testability	4	6											
Usability	4												
Quality assurance	10					5		5					
PDR and/or DW	10					7		5					
CDR						7		5					
Program DR								5					
Metrics						5-7			18, 23				
Graph metrics						7			18				
Struct. des. m.						5			18				
OO des. m.						6			23				

Software architecture													
Structures and viewpoints	2	6	4					5					
Beh./func./str./data	2		4										
Conc./mod./code./ arch.	2	6						5					
Log./proc./phys./dev.	2	6											
Architectural styles	6, 8	2						5		2,4			
Batch systems										4			
Blackboards		2											
Data abs. and OO										2			
Distr. syst.	8	2						5					
Event syst.								5		2			
Interactive syst.	6	2											
Interpreters								5		2			
Layered syst.		2						5		2,4			
Pipes and filters		2						5		2			
Proc. control								5		2			
Reposit.								5		2,4			
Rule syst.										2			
Archi. descr.	12	1		V					21	6,7			
MIL				V						7			
Arch. descr. lang.	12									6,7			
Patterns	12	1							21				
OO Frameworks		6											
Des. notations													
Arch. design			4		12	5,6	9		14		5	*	*
Class diagr.						6	9						
CRC Cards					21							*	
Deploy. diagr.							9						
Module diagr.							9						
Struct. charts			4			5			14		5		*
Subsyst. diagr.							9						

Detailed design			4	III, VII		7	9		14				
Collab. diagr.							9						
Decision diagr									14				
Flowcharts				VII					14				
HIPO diagr.				III									
Jackson struct. diagr.			4										
PDL				VII		7			14				
Sequence diagr.							9						
STD			4			7	9						
Design strategies and methods													
General strategies			4	V-VII									
Data-oriented dec.			4	V									
Funct. dec.				V									
Info.-hiding				V									
OO design				VI									
Stepw. ref.				VII									
Data-str. design			4	III, VII							5		
JSP			4	VII							5		
JSD			4	III							5		
Funct.-or. design						5			14		5		*
SSADM											5		
Struct. design						5			14				*
OO design					4,7,19, 21		9	5				*	
Coad and Yourd.					4								
Fusion					7								
D BC								5					
Shlaer/Mellor					19								
UML Proc.							9						
Resp.-driven					21							*	

References

[BCK98]
L. Bass, P. Clements, and R. Kazman.
Software Architecture in Practice.
SEI Series in Software Engineering. Addison-Wesley, 1998.

[BMR+96]
F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal.
A System of Patterns.
Wiley, West Sussex, England, 1996.

[DT97]
M. Dorfman and R.H. Thayer.
Software Engineering.
IEEE Computer Society Press, Los Alamitos, CA, 1997.

[FW83]
P. Freeman and A.I. Wasserman.
Tutorial on Software Design Techniques, fourth edition.
IEEE Computer Society Press, Silver Spring, Md, 1983.

[Hut94]
A.T.F. Hutt.
Object Analysis and Design --- Description of Methods.
John Wiley & Sons, New York, 1994.

[IEE87a]
An american national standard --- IEEE recommended practice for Ada as a program design language.
IEEE Std 990-1987, IEEE, New York, NY, 1987.

[IEE87b]
An american national standard --- IEEE recommended practice for software design descriptions.
IEEE Std 1016-1987, IEEE, New York, NY, 1987.

[IEE91]
IEEE standard glossary of Software Engineering Terminology.
IEEE Std 610.12-1990, IEEE, New York, NY, 1993.

[IEE93a]
IEEE guide to software design descriptions.
IEEE Std 1016.1-1993, IEEE, New York, NY, 1993.

[Jal97]
P. Jalote.
An integrated approach to software engineering, 2nd ed.
Springer, New York, NY, 1997.

[JBR99]
I. Jacobson, G. Booch, and J. Rumbaugh.
The Unified Software Development Process.
Addison-Wesley, Reading, Ma, 1999.

[Pfl98]
S.L. Pfleeger.
Software Engineering --- Theory and Practice.
Prentice-Hall, Inc., 1998.

[Pre92]
R.S. Pressman.
Software Engineering --- A Practitioner's Approach (Third Edition).
McGraw-Hill, Inc., 1992.

[SG96]
M. Shaw and D. Garlan.
Software architecture: Pespectives on an emerging discipline.
Prentice Hall, Englewood Cliff, NJ, 1996.

[TM93]
R.H. Thayer and A.D. McGettrick.
Software Engineering: A European Perspective.
IEEE Computer Society Press, Los Alamitos, CA, 1993.

[WBWW90]
R. Wirfs-Brock, B. Wilkerson, and L. Wiener.
Designing Object-Oriented Software.
Prentice-Hall, Prentice-Hall, NJ, 1990.

[YC79]
E. Yourdon and L. Constantine.
Structured Design.
Yourdon Press, Englewood Cliffs, NJ, 1979.

SWEBOK Knowledge Area Description for Software Engineering Infrastructure (version 0.5)

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

1 INTRODUCTION

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". This Knowledge Area contains the topics of development methods and software development environments. The Industry Advisory Board in Mont-Tremblant identified "standard designs", "integration" and "reuse" as potential additions to this Knowledge Area. Development of this document has been guided by the document "Knowledge Area Description Specifications for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge (version 0.2)".

2 SOURCES USED

The five standard 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. Specialised references have been identified for particular topics, e.g., Szyperski [Szy97] for component-based software. *Note: citations will use the letters plus year digits form in the initial versions of this document while the list of references is volatile. Ultimately reference numbers will be used to conform to the IEEE style.*

3 AUTHOR REFLECTIONS

Scope of Knowledge Area

The request by the Industry Advisory Board that the topics "standard designs", "integration" and "reuse" be added to this Knowledge Area has proven difficult to achieve in a coherent fashion. Few links have been identified between these topics and the original topics associated with this Knowledge Area. Appendix 1 of the "Approved Baseline" document suggests that an additional Knowledge Area covering the life cycle operation of integration would be an alternative location for these topics. Initial development of the Software Engineering Infrastructure Knowledge Area supports this proposal. Such a Knowledge Area also strengthens the conformance to the structure identified in the ISO/IEC 12207 standard.

Approaches to Knowledge Area Breakdown

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. 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.

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 organise topics in this Knowledge Area. This does mean that these links are not explicitly identified.

Some topics in this Knowledge Area do not have corresponding reference materials identified in the matrices in Appendix 2. There are two possible conclusions: 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.

4 DEFINITION OF KNOWLEDGE AREA

The Software Engineering Infrastructure Knowledge Area includes both the development methods and the software development environments knowledge areas identified in the Straw Man version of the guide.

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.

Software development environments are the computer-based tools that are intended to assist the software development 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.

The emergence of software components as a viable approach to software development represents a maturing of the discipline to overcome the "not invented here" syndrome. The point is often made that more mature engineering disciplines rely on component technologies [Szy97]. Using components affects both methods and tools but the extent of this effect is currently difficult to quantify.

5 OUTLINE OF KNOWLEDGE AREA

This section contains a "first-cut" breakdown of topics in the Software Engineering Infrastructure Knowledge Area.

The Development Methods section has been divided into three subsections: *heuristic methods* dealing with informal approaches, *formal methods* dealing with mathematically based approaches, and *prototyping methods* dealing with software development approaches based on various forms of prototyping. The 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.

The top-level partitioning of the Software Tools section is based initially on part of the ISO/IEC 12207 Standard by separating out development and maintenance, supporting activities and management tools. The remaining categories cover integrated tool sets (also known as software engineering environments), and tool assessment techniques. This section is broader than the preceding methods section as it covers the full spectrum of tools.

The term "software document" refers generically to any product of a software life-cycle task. Creation and editing refers to human-controlled development whereas translation refers to machine-controlled development.

The Component Integration section dealing with components and integration has been partitioned into topics dealing with individual components, reference models that describe how components can be combined, and the more general topic of reuse.

I. Development Methods

- A. Heuristic methods
  1. structured methods
  2. data-oriented methods
  3. object-oriented methods
  4. real-time methods
- B. Formal methods
  1. specification languages: *model-oriented, property-oriented, behaviour-oriented, visual, executable*
  2. refinement (reification)
  3. verification/proving properties: *theorem proving, model checking*
- C. Prototyping methods
  1. styles: *throwaway, evolutionary/iterative, executable specification*
  2. prototyping target: *requirements, design, user interface*
  3. evaluation techniques

II. Software Tools

- A. Development & maintenance tools
  1. creation & editing of software documents
  2. translation of software documents: *preprocessors, compilers, linkers/loaders, GUI generators, document generators*
  3. analysis of software documents: *syntactic/semantic; data, control flow and dependency analysers*
  4. comprehension tools: *debuggers, static and dynamic visualisation, program slicing*
  5. reverse and re-engineering tools
- B. Supporting activities tools
  1. configuration management tools: *system builders, version managers*
  2. review tools
  3. verification & validation tools: a*nimation, prototyping and testing tools*
- C. Management tools
  1. measurement tools
  2. planning and estimation tools
  3. communication tools (groupware)
- D. Workbenches: integrated CASE tools and Software Engineering Environments
  1. storage repositories (software engineering databases)
  2. integration techniques: *platform, presentation, process, data, control*
  3. process tools
  4. meta-tools
- E. Tool assessment techniques

III. Component Integration

A. Component definition
  1. interface specifications
  2. protocol specifications
  3. off-the-shelf components

B. Reference models
  1. patterns
  2. frameworks
  3. standard architectures: c*lient-server, middleware products*
  4. semantic interoperability

C. Reuse
  1. types of reuse: *code, design, requirements*
  2. re-engineering
  3. reuse repositories
  4. cost/benefit analysis

6 RELEVANT STANDARDS

No standards for software development methodologies have been identified although individual methods are sometimes standardised.

For software tools, the relevant standards are:

- Trial-Use Standard Reference Model for Computing System Tool Interconnections, IEEE Std 1175-1992
- IEEE Recommended Practice for the Evaluation and Selection of CASE Tools, IEEE Std 1209-1992 (ISO/IEC 14102)
- IEEE Recommended Practice for the Adoption of CASE Tools, IEEE Std 1348-1995 (ISO/IEC 14471)

For reuse, the relevant standards are:

- IEEE Standard for Information Technology — Software Reuse—Data Model for Reuse Library Interoperability: Basic Interoperability Data Model (BIDM), IEEE Std 1420.1-1995
- Supplement to IEEE Standard for Information Technology —Software Reuse—Data Model for Reuse Library Interoperability: Asset Certification Framework, IEEE Std 1420.1a-1996
- Guide for Information Technology—Software Reuse—Concept of Operations for Interoperating Reuse Libraries, IEEE Std 1430-1996

7 CLASSIFICATION OF KNOWLEDGE AREA

In this section, the topics of this Knowledge Area will be classified according to the categories of engineering design knowledge defined by Vincenti [Vin90, Chapter 7].

To be completed.

8 RELATED DISCIPLINES

Mastery of the Software Engineering Infrastructure Knowledge Area is considered to require knowledge of the following Knowledge Areas of Related Disciplines:

- Computer Science
- Mathematics
- Project Management

To be completed.

The full list of Knowledge Areas of Related Disciplines can be found in the document "A Proposed Baseline for a List of Related Disciplines".

ACKNOWLEDGEMENTS

This document was developed from the jump-start document written by Tuong Vinh Ho. Phil Cook, Andrew Coyle, Ian MacColl and Jim Welsh helped to identify relevant information sources and offered helpful advice. Their contribution is gratefully acknowledged. The two referees for version 0.1, Don Bagert and Jorge Diaz-Herrera, provided useful advice about topics and structure (see Appendix 3).

REFERENCES

1. Alan W. Brown and Kurt C. Wallnau. Engineering of Component-Based Systems. In *Component-Based Software Engineering*, Alan W. Brown (Editor), IEEE Computer Society Press, pages 7-15, 1996.
2. 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.
3. Merlin Dorfman and Richard H. Thayer, Editors. *Software Engineering*. IEEE Computer Society, 1997.
4. James W. Moore. S*oftware Engineering Standards: A User's Road Map*. IEEE Computer Society, 1998.
5. Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, 1998.
6. Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. 4th edition, McGraw-Hill, 1997.
7. Steven P. Reiss. Software Tools and Environments, Ch. 112, pages 2419-2439. In Tucker [Tuc96], 1996.
8. Ian Sommerville. *Software Engineering*. 5th edition, Addison-Wesley, 1996.
9. Clemens Szyperski. *Component Software: Beyond Object-oriented Programming.* Addison-Wesley, 1997.
10. Allen B. Tucker, Jr., Editor-in-chief. *The Computer Science and Engineering Handbook*. CRC Press, 1996.
11. Walter G. Vincenti. *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*. John Hopkins University Press, 1990.

APPENDIX 1: BLOOM’S TAXONOMY ANALYSIS

This appendix will (when complete) identify for each topic in this Knowledge Area the level at which a “graduate plus four years experience” should “master this topic.

To be completed.

APPENDIX 2: REFERENCE MATERIAL MATRICES

The following matrices indicate for each topic sources of information within the selected references (see Section 2). The tutorial volume [DT97] contains a collection of papers organised into chapters. The following papers are referenced (section numbers have been added to reference individual papers more conveniently):

Chapter 5: Software Development Methodologies

5.1 Object-oriented Development, Linda M. Northrup

5.2 Object-oriented Systems Development: Survey of Structured Methods, A.G. Sutcliffe

5.4 A Review of Formal Methods, Robert Vienneau

Chapter 12 Software Technology

12.1 The Re-engineering and Reuse of Software, Patrick A.V. Hall and Lingzi Jin

12.2 Prototyping: Alternate Systems Development Methodology, J.M. Carey

12.3 A Classification of CASE Technology, Alfonso Fuggetta

The text by Pfleeger [Pfl98] is structured according to the phases of a life cycle so that discussion of methods and tools is distributed throughout the book.

Note: formatting these matrices within the two-column format is excessively constraining. Landscape mode may provide the maximum flexibility.

I. Development Methods	**BW96**	**CW96**	**DT97**	**Moo98**	**Pfl98**	**Pre98**	**Rei96**	**Som96**	**Szy97**
A. Heuristic Methods						10-23			
1. structured methods			5.2		4.5,	10-18		15	
2. data-oriented methods			5.2			12.8			
3. object-oriented methods			5.1, 5.2		4.4, 7.5	19-23		6.3, 14	
4. real-time methods						15		16	
B. Formal Methods		✓	5.4			24, 25		9-11	
1. specification languages		✓			4.5	24.4			
2. refinement						25.3			
3. verification/proving properties		✓			5.7, 7.3			24.2	
C. Prototyping Methods			12.2		4.6, 5.6	2.5, 11.4		8	
1. styles			12.2		4.6	11.4			
2. prototyping target			12.2						
3. evaluation techniques									

II. Software Tools	**BW96**	**CW96**	**DT97**	**Moo98**	**Pfl98**	**Pre98**	**Rei96**	**Som96**	**Szy97**
A. Development & maintenance tools			12.3		10.5	29	112.2		
1. creation & editing			12.3				112.2		
2. translation tools			12.3		10.5		112.2		
3. analysis tools		✓	12.3		7.7, 8.7		112.5	24.3	
4. comprehension tools			12.3				112.5		
5. reverse & re-engineering tools			12.3						
B. Supporting Activity tools			12.3				112.3		
1. configuration management			12.3		10.5			33.3, 33.4	
2. review tools			12.3						
3. verification & validation tools		✓	12.3		7.7, 8.7		112.3		
C. Management tools			12.3						
1. measurement tools			12.3						
2. planning & estimation tools			12.3						
3. communication (groupware) tools			12.3						
D. Workbenches and CASE tools			12.3	11	1.8	29	112.3, 112.4	25-27	
1. storage repositories			12.3			29.6			
2. integration techniques			12.3	11	1.8	29.5		27.1	
3. process tools			12.3		2.3, 2.4		112.4		
4. meta-tools			12.3					26.4	
F. Tool assessment techniques				11	8.10				

III. Component Integration	BW96	CW96	DT97	Moo98	Pfl98	Pre98	Rei96	Som96	Szy97
A. Component definition	✓		12.1			26.4			4, 5, 22
1. interface specifications						26.5			5
2. protocol specifications									
3. off-the-shelf components	✓					26.5			
B. Reference models	✓								9, 20, 21
1. patterns									9
2. frameworks									21
3. standard architectures	✓					28		13	20
4. semantic interoperability	✓								9
C. Reuse			12.1	11	11.4	26, 27		20	
1. types of reuse			12.1		11.4	26.2		20	
2. re-engineering			12.1		10.6	27		34	
3. reuse repositories			12.1		11.4	26.5			
4. cost/benefit analysis			12.1		11.4	26.6			

APPENDIX 3: FEEDBACK ON VERSION 0.1

The following comments were received as feedback on version 0.1 of this document and were handled as described below in italics. For subsequent reviews, it may be more appropriate to create a separate document for the feedback comments and their disposition.

Don Bagert:

Here are my comments concerning Infrastructure v0.1:

1. Concerning "Development Methods", section 1 (object-oriented methods) should include a subsection on patterns. Even though patterns are called "Design Patterns", the "Design Methods" subsection is not sufficient to include patterns, which are more of an architectural technique.

   Patterns have been included in the Component Integration section. The restructuring of the Development Methods section reduces the significance of any addition to this section.

2. Once again concerning "Development Methods", section 4 (structured methods") should include a subsection on structured programming. (Note: Section 1 includes a subsection on object-oriented programming; either structured and OO programming should be both in or both out; I suggest that they are both in.)

   The restructuring of the Development Methods section reduces the significance of this suggestion although the words "structured programming" have been added.

3. For section 8 (Related Disciplines), I would include Computer Science, Mathematics, and possibly Project Management.

   This suggestion has been incorporated in this version.

Jorge L. Diaz Herrera:

The comments below correspond to Carrington's document "SWEBOK Knowledge Area Description for Software Engineering Infrastructure (version 0.1).

Development methods

1. I suggest reorganising the main partitioning of Development Methods into three main categories related by a single aspect (type of method) as follows:

   1) Heuristic methods
      -- make Object-oriented and Structure methods subcategories of this partition
      -- add structured programming, integration and testing, and metrics to Structured methods category (just like for OO)
      -- what about data-oriented methods?
      -- what about real-time methods?

   2) Formal methods

   3) Prototyping methods
      -- add "iii) operational Specifications" to a) styles

   This restructuring suggestion has been adopted.

Software Development Tools

2. could this area be renamed simply "Software Tools" (to avoid redundancy below)?

   Incorporated

3. under 1.b, add "iv) GUI generators" and "v) document generators"

   Incorporated

4. under 3.c.iii, add "(#) Dependency analysers" (use or static structure)

   Incorporated

5. delete 3.d "Documentation tools" since this is included in 1.b above; document generation to conform to a standard is more part of "translation of software documents" than to "supporting tools."

   Incorporated

Component Integration

This area seems to require more work. A clearer definition is lacking. E.g., the whole category of reuse needs to be developed. Numerous topics related to reuse are missing, although they may be treated elsewhere in the SWEBOK; these include product-line practices, domain engineering, domain analysis, development-for-reuse (generic designs) vs. development-with-reuse (product configuration). Also component integration approaches (like top-down vs. bottom-up) may fit in this category too.

Additional work is definitely required in this section. The issue of whether this section belongs in this Knowledge Area is still to be resolved.

Here are some specific comments on this area.

6. not sure if "1.b Off-the-shelf components" belong here.

   Left in for now

7. add "1.c protocol specifications" (counterpart to interface specifications"

   Incorporated

8. change "2) Standard Designs" to "2) Reference Models"

   Incorporated

9. Add "2.d Patterns"

   Incorporated

10. In category 3, be consistent "3) reuse vs. b) re-use"

    Fixed

11. not sure we need “3.a types of reuse …”

    Left in for now

12. in 3.b, “libraries” seems too restrictive, I suggest “repositories”

    Incorporated

SWEBOK: Software engineering management knowledge area description version 0.5

Stephen G. MacDonell and Andrew R. Gray
Software Metrics Research Laboratory
University of Otago, Dunedin, New Zealand
+64 3 479 8135 (phone) +64 3 479 8311 (fax)
{stevemac}{agray}@infoscience.otago.ac.nz

1 Introduction

This is the current draft (version 0.5) of the knowledge area description for software engineering management. The relevant jump-start document [Gra99] was quite detailed in certain respects and some of its material has been incorporated here (with suitable modifications and extensions). Refinements from version 0.1 of this document reflect the more detailed presentation required and some much appreciated feedback from reviewers. It is apparent that there are several different viewpoints as to the breakdown of this topic and the reviewers' alternative perspectives have been very helpful in improving this document – both in terms of clarity and content.

The primary goal of this draft is to present the breakdown of the knowledge area into topics, list reference material, and to provide the topic-reference matrix. Additionally, an initial sketch of Bloom's taxonomy for certification courses is included. The tables showing Vincenti's classifications for the topics are still being revised from those shown in the jump-start document and version 0.1 of the draft, as is the table of standards and references.

Much of this material is therefore still incomplete, as is to be expected with a draft document at this stage of its life. A considerable amount of additional material is included to assist the authors and reviewers to work towards the final version. The eventually completed document is expected to be much closer to the ten page suggested guideline from [BDA$^+$99] once this material is removed.

The software engineering management knowledge area consists of both the measurement/metrics and management process sub-areas (as defined below based on the jump-start document). 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 necessitates the combined treatment taken here as part of the SWEBOK. In essence, management without measurement – qualitative or quantitative – suggests a lack of rigor and measurement without management suggests a lack of purpose or context

The following working definitions are based on those from the software engineering management jump-start document [Gra99].

> Measurement/metrics refers to the assignment of values and labels to aspects of software development (products, processes, and resources) and the models that may be derived therefrom.
>
> 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.
>
> The management process area makes (in theory at least) extensive use of the measurement/metrics area – ideally this exchange between the two processes occurs continuously throughout the development process.

As is apparent from these definitions, the software engineering management knowledge area overlaps with several other knowledge areas, most notably, software construction, software testing, configuration management, and software quality analysis. This overlap is explored a little more fully in Appendix E.

2 References used

All of the references from the jump-start document are used here, namely [DT97, FP97, Moo98, Pfl98, Pre97b, Som96][1]. These form a core of texts that provides a solid understanding of most of the area and are all expected to be updated with sufficient regularity to ensure timely and complete coverage. The latter point is of greatest importance for the management process area which encompasses many disparate topics.

[1] With respect to the [DT97] text, only a subset of the papers were used in the jump-start document. They are [Boe97a, Boe97b, Bro97a, Bro97b, BB97, Car97, Com97, DeM97, Fai97, Gib97, Hee97, Hur97, NB97, Pau97, Pre97a, Tha97a, Tha97c].

An additional reference added is a basic statistics text to complement and extend the material covered in [FP97] – which is by necessity limited in treatment. The text we have chosen is [MM98] although any general introductory statistics book would suffice equally well here. However this text is very highly regarded for introductory statistics courses and should be fairly accessible to software engineers in general. As well as having excellent presentation it contains a plentiful source of sample applications that should be useful in software engineering management practice.

Other references added include [Rei97], [Tha97b], [Kar96], and [Zus97]. [Rei97] contains extensive supplementary material on the management process topics presented in the breakdown, whilst [Zus97] provides more comprehensive theoretical coverage of the measurement/metrics area. [Tha97b] complements the collections of papers found in [Rei97] and [DT97], although there is some overlap. In order to provide a more general and comprehensive text on risk management we have also included [Kar96]. As a group these supplementary references provide broader coverage of the issues associated with the *management* of software engineering as compared to the set of core texts that are more closely focused on the *process* of software engineering.

3 Outline of knowledge area

The outline of the knowledge area topics is shown firstly as the draft from the jump start document, and then in a revised form (to illustrate its evolution) . We are still preparing another breakdown (as referred to in version 0.1 of the draft) based on topics rather than the life-cycle approach taken here. We envisage that both breakdowns will be useful (and complementary) in education and practice.

Original outline

The jump start document for this knowledge area identified the following topics before classifying them (using slightly different terminology) using Vincenti's [Vin90] classification scheme [Gra99]. See Appendix B or the original jump-start document for this classification.

1. Measurement
   - (a) Selection of measurements
   - (b) Collection of data
2. Software metric models
   - (a) Model building and calibration
   - (b) Model evaluation
   - (c) Implementation of models
   - (d) Refinement of models
3. Existing metric models
   - (a) Function Point Analysis
   - (b) COCOMO
4. Initiation and scope definition
   - (a) Collection of requirements
   - (b) Feasibility analysis
   - (c) Process for the revision of requirements
5. Planning
   - (a) Schedule and cost estimation
   - (b) Risk assessment
   - (c) Resource allocation
   - (d) Task and responsibility allocation
   - (e) Quality control
6. Execution

    (a) Implementation of plan
    (b) Monitor process (including reporting)
    (c) Control process
    (d) Feedback

7. Review and evaluation
    (a) Determining satisfaction of requirements
    (b) Reviewing and evaluating performance

8. Closure
    (a) Determining closure
    (b) Archival activities

As can be seen in this breakdown the topics divide into two sections. Measurement/metrics covers the first three headings, whilst management process concerns the following five.

Revised outline number one
The changes that have been made through versions 0.1 to 0.5 of the draft are as follows:

- The three major topics relating to measurement have now been "collapsed" into one so that its relative importance in the breakdown is reflected more realistically.
- We have augmented the topic area of coordination, to now consist of portfolio management, policy management, personnel management, and general management activities. Communication is a vital part of all management endeavors – irrespective of the domain of application. These are all higher-level issues that were omitted in the jump-start which focused more on individual projects (although it did make mention of portfolio management). As such we assume that this set of tasks is ongoing throughout all software processes. However, the focus is (necessarily) on the management of software development projects and related activities rather than general management.
- Feasibility analysis is now considered to be more of an on-going activity and has been added to the planning topic as well as still being part of initiation and scope definition. This differs from risk assessment and management in that feasibility analysis deals with the question "should we continue with the project?" Furthermore feasibility can be broken down into two types – technical and economic, the former dealing with "can it be done?" and the latter with "can we afford to do it?". Risk assessment and management on the other hand deal with the uncertainties that occur as part of the project with the intention that the project will continue. This can include allowances for delays, alternative paths to a goal, contingency plan development, and such like.
- Change control and configuration management are now mentioned as part of controlling the development process under the execution topic. This further stresses the interrelationship between the two knowledge areas.
- In some cases, third-level subtopics have been added to clarify which aspects of second-level subtopics are considered important here. These are not intended to be comprehensive, but rather are indicative of the important subtopics that we consider should be covered.

The current outline is very much a "life-cycle" based breakdown. Topics tend to appear in each of the two threads in the same order as their associated activities are enacted in a software development project – with the obvious exception of the "coordination" topic.

In several places specific techniques are listed. This generally indicates that the technique is suggested as being a good tutorial/case-study example of the overall concept. Other specifics could be used to replace these if desired.

The current outline is proposed as follows.

1. Measurement
    (a) The goal of a measurement program

      i. Organizational objectives
      ii. Software process improvement goals
      iii. Determining measurement goals
   (b) Selection of measurements
      i. The Goal/Question/Metric approach (as an example)
      ii. Other metric frameworks (such as Practical Software Measurement (PSM) and GQM++)
      iii. Measurement validity
   (c) Collection of data
      i. Survey techniques and questionnaire design
      ii. Automated and manual data extraction
      iii. Costing data collection
   (d) Software metric models
      i. Model building, calibration and evaluation
      ii. Implementation and refinement of models
      iii. Existing models (examples as case-studies)

2. Coordination
   (a) Portfolio management
      i. Project selection
      ii. Portfolio construction (risk minimization)
   (b) Policy management
      i. Means of policy development
      ii. Policy dissemination and enforcement
   (c) Personnel management
      i. Hiring and staffing
      ii. Directing personnel
      iii. Team structures
   (d) Communication
   (e) General management issues

3. Initiation and scope definition
   (a) Collection of requirements
   (b) Feasibility analysis
      i. Technical feasibility
      ii. Investment analysis
   (c) Process for the revision of requirements

4. Planning
   (a) Schedule and cost estimation
      i. Effort estimation
      ii. Task dependencies
      iii. Duration estimation
   (b) Risk assessment
      i. Critical risk analysis
      ii. Techniques for modeling risk
      iii. Contingency planning

   (c) Resource allocation
   (d) Task and responsibility allocation
   (e) Quality control
   (f) Feasibility analysis
      i. Investment analysis
      ii. Project abandonment policies

5. Execution
   (a) Implementation of plan
   (b) Monitor process
      i. Reporting
      ii. Variance analysis
   (c) Control process
      i. Change control
      ii. Configuration management
      iii. Scenario analysis
   (d) Feedback

6. Review and evaluation
   (a) Determining satisfaction of requirements
   (b) Reviewing and evaluating performance
      i. Personnel performance
      ii. Tool and technique evaluation
      iii. Process assessment

7. Closure
   (a) Determining closure
   (b) Archival activities
      i. Measurement database

These topics are not listed in temporal order since there are in fact two distinct processes being performed here as was mentioned with respect to the previous iteration, namely, measurement/metrics and management process. Figure 1 shows this more clearly. We have decided to treat the former as the actual activity of developing and releasing of models, and the latter as the usage of those pre-existing models. This is discussed in more detail later in the document.

4 Descriptions of topics

The overall topic is, in this breakdown, divided into the measurement/metrics and management process sub-areas.

Within the measurement/metrics topic area four main subtopics are addressed: measurement program goals, measurement selection, data collection and model development. The first three 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 fourth subtopic (model development) 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 hold-out samples) to ensure that their levels of accuracy are both 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

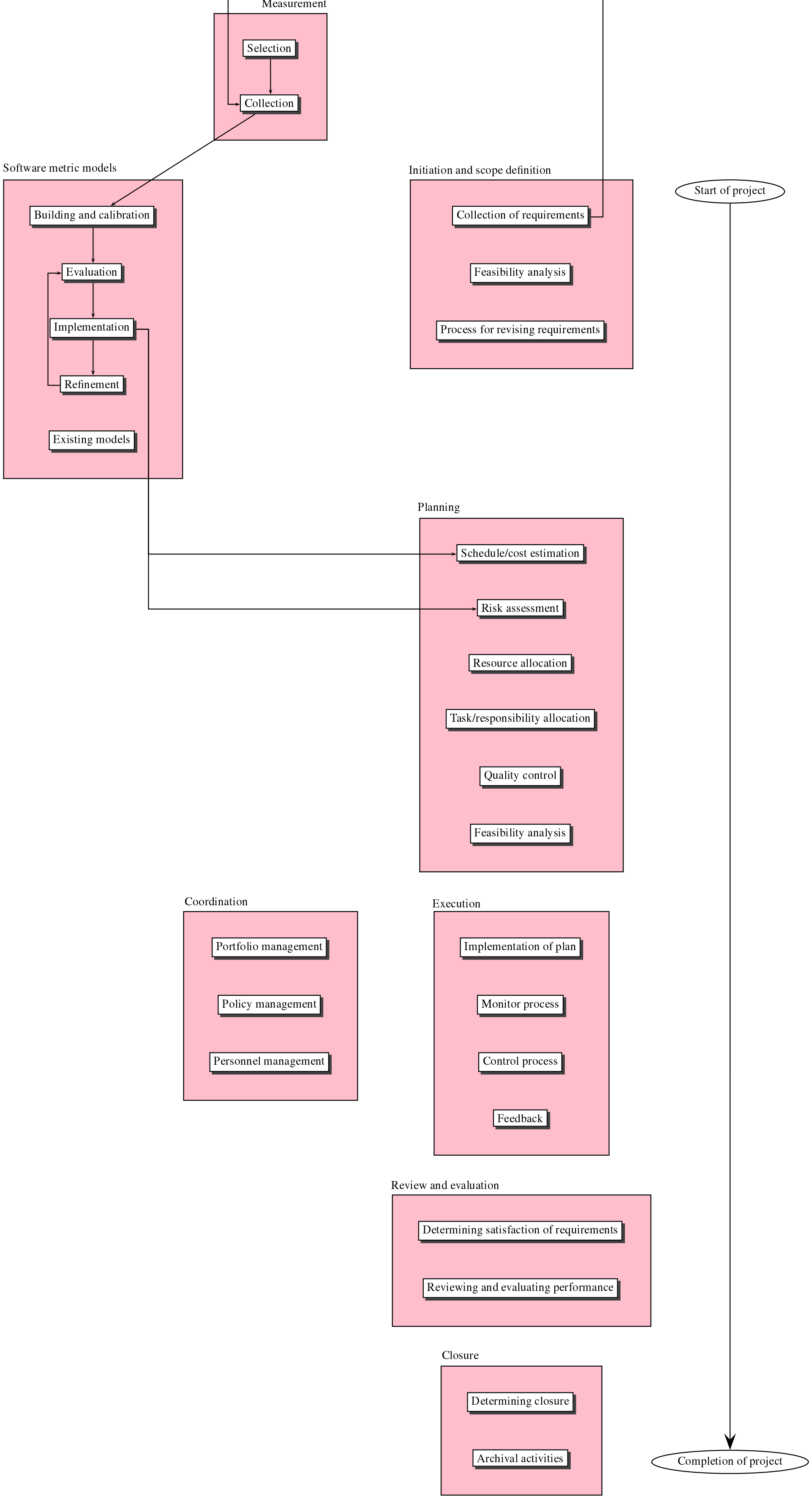


Figure 1: Measurement/metrics and management process flowchart (draft)

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). On the other hand Zuse's excellent book 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.)

In the management process sub-area the notion of management "in the large" is considered in the coordination topic, addressing issues including project selection, the development and implementation of standards, project staffing, and team development. The remaining topics then correspond 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 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". It is during this sub-task that the communication between the two sub-areas begins. Feasibility may well be determined by reference to some formal model.

Planning is the next set of activities for a software engineering manager. This continues the interaction between the two processes. Schedule and cost estimation are perhaps the most vital tasks, although without some form of risk analysis estimates can be quite worthless. Again, feasibility analysis needs to be performed, this time from the perspective of the project. Given schedule estimates it is possible to perform 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 execution 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 development 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.

5 Justification of revised outline

The above breakdown of topics is based on a division into measurement/metrics and management process. 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."

The division of topics within measurement/metrics consists of the preliminary activities (measure selection, preferably via a controlled process such as GQM, and then data collection for calibration). This then leads to model development. Here the actual models are created, implemented and refined.

Apart from the "meta-project management" issues considered in the coordination topic, the management process section follows very much a life-cycle approach, with topics covering each stage in a project from its initiation though to archival activities. Since management involves all other activities, directly or indirectly, there is considerable overlap with other knowledge areas.

6 Matrix of material versus topics

In Table 1 a '■' indicates that the topic is covered to a reasonable degree. The threshold of *reasonableness* is subjective.

Topic	[DT97][2]	[FP97]	[Kar96]	[MM98]	[Pfl98]	[Pre97b]	[Rei97]	[Som96]	[Tha97b]	[Zus97]
Archival activities						■		■		
Collection of data		■		■						■
Collection of requirements	■				■	■		■	■	
Communication	■	■	■		■	■	■	■	■	
Control process	■				■	■	■	■	■	
Determining closure						■		■		
Determining satisfaction of requirements	■				■	■		■	■	
Feasibility analysis						■		■		
Feedback						■	■	■	■	
General management issues	■	■	■		■	■	■	■	■	
Goal of measurement	■	■	■		■	■		■	■	■
Implementation of plan	■				■	■	■	■	■	
Monitor process	■				■	■	■	■	■	
Personnel management			■		■	■	■	■	■	
Policy management					■	■	■	■	■	
Portfolio management										
Process for the revision of requirements	■				■	■		■	■	
Quality control	■					■	■	■	■	
Resource allocation					■	■		■	■	
Reviewing and evaluating performance		■			■	■	■	■	■	■
Risk assessment	■		■		■	■	■	■	■	■
Schedule and cost estimation	■	■	■		■	■	■	■	■	■
Selection of measurements		■				■		■	■	■
Software metric models	■	■	■	■	■	■	■	■	■	■
Task and responsibility allocation					■	■		■		

Table 1: Topics and their references

A Assessing the breakdown using the criteria

The following subsections each discuss how the proposed draft of the KA meets the criteria given in the guide to producing this document [BDA$^+$99]. In cases where the relevant material has yet to be written this is noted as “TO DO”.

One or two breakdowns with identical topics

A single breakdown of topics is shown at present along with the draft version from the jump start document.

Soundness and reasonableness

The primary references and secondary sources were examined quite thoroughly in order to list all main topics. The division of the management process into life-cycle based topics seems both plausible and useful in terms of educational presentation.

Generally acceptable

In our view the material in this knowledge area description meets the criterion of being generally acceptable in terms of being “applicable to most projects, most of the time” and having “widespread consensus about their value and usefulness” [Dun96]. These topics are those that receive the greatest coverage in both the original texts and additional materials suggested herein.

Similarly, the Industrial Advisory Board definition of “study material of a software engineering licensing exam that a graduate would pass after completing four years of work experience” appears to be met. However, in this case the specific responsibilities of the graduate will obvious influence what areas they have the opportunity to gain experience in. Project management is often a more senior position and as such graduates with four years of experience may not have had significant experience in managing, at least with large-scale, projects.

The importance of measurement and better management practices is widely acknowledged and so its importance can only increase in the coming three to five years. Measurement has become one of the cornerstones of organizational maturity.

The only material that we feel *could* be seen to too specialized is that of software metric modeling using statistics. However, the level required here is roughly equivalent to a solid introductory course and should not be overly demanding. The understanding is not merely required for the development of models, but also for understanding existing measures and models and their limitations.

Avoid presuming specific features of applicability

The measurement/metric activities of model development, implementation and revision are seen as essential to any managed software development process. Similarly, the management activities are mostly generic to management in general.

Compatible with various schools of thought within software engineering

Excluding debate on measurement theoretic issues there is little intense debate in the measurement/metrics field.

There is nothing that appears to be controversial in the management process sub-area.

Compatible with breakdown in industry, literatures, and standards

TO DO

Inclusive as opposed to limited coverage

We have included some areas, such as statistical model building and assessment, that may be considered too specialized by some.

Inclusion of themes (quality, measurement, tools, and standards)

Quality is an important aspect of management and this is noted in Appendix E.

The second of these, measurement, is obviously included as one of the sub-areas.

Tools TO DO

Standards are listed in Appendix C.

Depth and node density

The suggested guidelines have been met here.

Meaningful topic names

Apart from the issue relating to the use of the term “metrics” (as discussed above) we feel that the names are indeed meaningful.

Breakdown based on Vincenti’s classification scheme

See Appendix B for a draft breakdown from the original jump-start document. The new version, using the new references and new topic areas, is currently being prepared for the next version of this document.

Brevity of topic descriptions
The descriptions seem adequately brief and to the point.

Rationale of breakdown
We believe that the breakdowns are both rational and suitable for education. The section above on justification for the breakdown provides more information relating to this criterion.

Bloom's taxonomy
A draft version is shown in Appendix D. This will be revised and justification added in the next version of this document.

Specific reference material
Additional reference material for more specialized topics not covered adequately in the primary reference material has been added.

Proposed reference material (publicly available)
All material is publicly available.

Language of material must be English
Yes.

Maximum number of reference materials is 15
We are aiming to adhere to this limit.

Preference to IEEE or ACM copyrighted material
This is evident in the selection of reference material, especially the collections of papers.

Matrix of reference material versus topics
A draft of this is presented in 6.

Licensing exam material
TO DO

B Vincenti's categories

This is the version of the classification found in the jump-start documentation [Gra99]. A newer version with the additional references and the new topics added here is being prepared for the next version of this document.

The following Tables 2, 3, 4, 5, 6, and 7 list the main topics in the software engineering management Knowledge Area under each of Vincenti's categories [Vin90]. The relevant literature for each topic is also indicated. These categories, as noted by Vincenti, are neither mutually exclusive nor exhaustive. With the former point in mind, many topics appear in more than one table. The area of software engineering management does not, in the authors' opinion, fit easily into Vincenti's categories.

In these tables a ■ indicates that the topic is covered to a reasonable degree. The threshold of *reasonableness* is of course subjective.

Fundamental Design Concepts	[DT97]	[FP97]	[Pfl98]	[Pre97b]	[Som96]
Project planning	■		■	■	■
Project management	■		■	■	■
Quality assurance	■			■	■
Characteristics of projects	■	■	■	■	■
Risk assessment and management	■		■	■	■
Software metric models	■	■	■	■	■

Table 2: Vincenti's categories – fundamental design concepts

C Standards

Table 8 is a possibly incomplete list of standards that are listed and/or discussed within the reference publications. This is the same as was provided in the jump-start document [Gra99]. The next step will be to reference the newer texts and map topics to standards.

D Bloom's taxonomy

Criteria and Specifications	[DT97]	[FP97]	[Pfl98]	[Pre97b]	[Som96]
Project proposal				■	■
System requirements	■		■	■	■
System characteristics		■	■	■	■
User characteristics			■	■	

Table 3: Vincenti's categories – criteria and specifications

Theoretical Tools	[DT97]	[FP97]	[Pfl98]	[Pre97b]	[Som96]
COCOMO	■	■	■	■	■
Function Point Analysis	■	■		■	■
Goal/Question/Metric Framework	■	■			■
Project schedule	■	■	■	■	■
Software metric models	■	■	■	■	■
Statistical techniques for model building		■			
Statistical techniques for analyzing experimental data		■			

Table 4: Vincenti's categories – theoretical tools

Quantitative Data	[DT97]	[FP97]	[Pfl98]	[Pre97b]	[Som96]
Complexity measurements	■	■		■	■
Cost estimates	■	■	■	■	■
Developer productivity	■	■	■	■	■
Effort estimates	■	■	■	■	■
Metric model assessment		■	■		
Quality measurements	■	■		■	■
Reliability measurements	■	■	■	■	■
Size measurements	■	■	■	■	■
Software standards	■		■	■	■
System performance requirements	■		■	■	■

Table 5: Vincenti's categories – quantitative data

Practical Considerations	[DT97]	[FP97]	[Pfl98]	[Pre97b]	[Som96]
Expert opinion for metrics estimation	■	■	■		■
Process maturity	■	■	■	■	■

Table 6: Vincenti's categories – practical considerations

Design Instrumentalities	[DT97]	[FP97]	[Pfl98]	[Pre97b]	[Som96]
Project management	■	■	■	■	■
Software metric model building	■	■	■	■	■

Table 7: Vincenti's categories – design instrumentalities

Table 9 shows the level of mastery that we feel a "graduate plus four years experience" should possess for each topic. This is still a very rough draft, but indicates our approach to this task.

E Related knowledge areas

Some parts of these topic areas overlap with other Knowledge Areas, most notably software construction (which must be compatible vis-á-vis software engineering management), software testing (which should involve software metrics in some capacity), and software quality analysis (which should also make use of software metrics and management-initiated reviews).

F Related disciplines

In terms of related disciplines (which are defined in terms of the authors' own experience) the most important are as follows. Again, this is an updated version of material from the jump-start document [Gra99].

Computer science

Without some understanding of the fundamental concepts underlying the activity of software development some areas of software engineering management would be inaccessible or prohibitively buried in computer science terminology and concepts. For example, many software metric models require some understanding of data structures as independent variables. Similarly, feasibility analysis requires a basic understanding of computer science in terms of hardware performance, storage devices, etcetera. The history of computing is also useful when projects require some judgements on likely changes to technologies in the industry.

Project management

This is obviously a major component of the software engineering management Knowledge Area.

Management

Many principles in managing software engineering projects are common to generic management processes. These include personnel issues, planning techniques and tools, budgeting, and project selection methods (both under limited resources and between mutually exclusive alternatives).

Cognitive science

This discipline is mainly invoked here as part of quality assurance in terms of usability. Other aspects making use of this discipline include motivation, task allocation, and training issues.

Management sciences

Many management science topics can be readily used without adaptation in a software engineering management context. This especially includes the use of modeling techniques.

REFERENCES

[BB97] David Budgen and Pearl Brereton. *A software engineering bibliography*, pages 503–508. In Dorfman and Thayer [DT97], 1997.

[BDA+99] Pierre Bourque, Robert Dupuis, Alain Abran, James W. Moore, and Leonard Tripp. Knowledge area description specifications for the stone man version of the guide to the software engineering body of knowledge version 0.2. Submitted to the Industrial Advisory Board for SWEBOK, March 1999.

[Boe97a] Barry W. Boehm. *Industrial software metrics top 10 list*, page 494. In Dorfman and Thayer [DT97], 1997.

[Boe97b] Barry W. Boehm. *A spiral model of software development and enhancement*, pages 415–426. In Dorfman and Thayer [DT97], 1997.

[Bro97a] Frederick P. Brooks. *The mythical man-month*, pages 350–357. In Dorfman and Thayer [DT97], 1997.

[Bro97b] Frederick P. Brooks. *No silver bullet: essence and accidents of software engineering*, pages 13–22. In Dorfman and Thayer [DT97], 1997.

[Car97] J. M. Carey. *Prototyping: alternative systems development methodology*, pages 461–468. In Dorfman and Thayer [DT97], 1997.

[Com97] Edward R. Comer. *Alternative software life cycle models*, pages 404–414. In Dorfman and Thayer [DT97], 1997.

[DeM97] Tom DeMarco. *Why does software cost so much?*, pages 372–373. In Dorfman and Thayer [DT97], 1997.

[DT97] Merlin Dorfman and Richard H. Thayer, editors. *Software engineering*. IEEE Computer Society, 1997.

[Dun96] W.R. Duncan. A guide to the project management body of knowledge. Project Management Institute, 1996.

[Fai97] Richard Fairly. *Risk management for software development*, pages 387–400. In Dorfman and Thayer [DT97], 1997.

[FP97] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics: a rigorous & practical approach*. PWS Publishing Company, second edition, 1997.

[Gib97] Wayt Gibbs. *Software's chronic crisis*, pages 4–12. In Dorfman and Thayer [DT97], 1997.

[Gra99] Andrew R. Gray. Swebok knowledge area jump-start document for software engineering management. Unpublished report for SWEBOK, February 1999.

[Hee97] F. J. Heemstra. *Software cost estimation*, pages 374–386. In Dorfman and Thayer [DT97], 1997.

[Hur97] Patricia W. Hurst. *Software quality assurance: a survey of an emerging view*, pages 308–319. In Dorfman and Thayer [DT97], 1997.

[Kar96] Dale Walter Karolak. *Software engineering risk management*. IEEE Computer Society, 1996.

[MM98] David S. Moore and George P. McCabe. *Introduction to the practice of statistics*. W H Freeman & Co., third edition, 1998.

[Moo98] James W. Moore. *Software engineering standards: a user's road map*. IEEE Computer Society, 1998.

[NB97] Ronald E. Nusenoff and Dennis C. Bunde. *A guidebook and a spreadsheet tool for a corporate metrics program*, pages 483–493. In Dorfman and Thayer [DT97], 1997.

[Pau97] Mark C. Paulk. *The Capability Maturity Model for software*, pages 427–438. In Dorfman and Thayer [DT97], 1997.

[Pfl98] Shari Lawrence Pfleeger. *Software engineering: theory and practice*. Prentice Hall, 1998.

[Pre97a] Roger S. Pressman. *Software engineering*, pages 57–74. In Dorfman and Thayer [DT97], 1997.

[Pre97b] Roger S. Pressman. *Software engineering: a practitioner's approach*. McGraw-Hill, fourth edition, 1997.

[Rei97] Donald J. Reifer, editor. *Software management*. IEEE Computer Society, fifth edition, 1997.

[Som96] Ian Sommerville. *Software engineering*. Addison-Wesley, 1996.

[Tha97a] Richard H. Thayer. *Software engineering project management*, pages 358–371. In Dorfman and Thayer [DT97], 1997.

[Tha97b] Richard H. Thayer, editor. *Software engineering project management*. IEEE Computer Society, 1997.

[Tha97c] Richard H. Thayer. *Software engineering standards*, pages 509–523. In Dorfman and Thayer [DT97], 1997.

[Vin90] W.G. Vincenti. *What engineers know and how they know it - analytical studies from aeronautical history*. John Hopkins, 1990.

[Zus97] Horst Zuse. *A framework of software measurement*. Walter de Gruyter, 1997.

Standard	[DT97][3]	[FP97]	[Moo98]	[Pfl98]	[Pre97b]	[Som96]
AFNOR						
Z67-101-1FD	■					
ASTM						
E 622-94	■					
E 792-87	■					
E E1113-86	■					
DOD						
AFSCP 800-43	■					
AFSCP 800-45	■					
MIL STD 1521B	■					
EIA						
CRB 1-89	■					
DMG-1-86	■					
DMG-2-89	■					
ESA						
PSS-05-08	■					
GER MOD						
PROSIS	■					
IEEE Std.						
982.1	■		■			
982.2	■		■			
1044			■			
1044.1			■			
1045	■		■			
1058.1	■		■			
1061	■		■			
1209	■					
1220	■					
ISO						
9000 series		■		■	■	■
9000-3		■		■	■	■
9001		■		■	■	■
9126		■		■		
ISO/IEC DIS						
14143.1			■			
14756			■			
JPL						
D-4011	■					
NATO						
NAT-PRC-1	■					
NAT-STAN-7	■					
PMI						
1996			■			

One of the papers in [DT97], namely [Tha97c] is in fact a comprehensive list of software engineering standards. All standards indicated as being mentioned in this volume came from this one paper in §4.14 (Management Standards).

Table 8: Standards relevant to software engineering management

Topic	Level
Archival activities	Application
Collection of data	Analysis
Collection of requirements	Analysis
Communication	Synthesis
Control process	Evaluation
Determining closure	Application
Determining satisfaction of requirements	Analysis
Feasibility analysis	Synthesis
Feedback	Synthesis
General management issues	Synthesis
Goal of measurement	Analysis
Implementation of plan	Synthesis
Monitor process	Analysis
Personnel management	Synthesis
Policy management	Synthesis
Portfolio management	Analysis
Process for the revision of requirements	Analysis
Quality control	Evaluation
Resource allocation	Application
Reviewing and evaluating performance	Synthesis
Risk assessment	Synthesis
Schedule and cost estimation	Evaluation
Selection of measurements	Analysis
Software metric models	Analysis
Task and responsibility allocation	Analysis

Table 9: Bloom’s taxonomy and mastery levels for the software engineering management topics

Description of The SWEBOK Knowledge Area Software Engineering Process (Version 0.5)

Khaled El Emam , NRC, Canada

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 first level is covered by the other KA's of the SWEBOK. This knowledge area is concerned with the second, meta, level of software process.

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:

- If one were to write a book on the topic of software engineering process, this KA description would identify the chapters and provide the initial references for writing the chapters. The KA description is not the book.
- If one were to prepare a certification exam on software engineering process, 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.
- If one were to prepare a course on the software engineering process, 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.

Scope

The scope of the KA is defined to exclude the following:

- Human resources management (as embodied in the People CMM [21] for example)
- Systems engineering processes

The reason for this exclusion is that, while important topics in themselves, they are outside the direct scope of software engineering process. However, where relevant, interfaces to HRM and systems engineering will be addressed.

Currency of Material

The software engineering process area 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, both of which are likely to have a considerable influence on the software engineering process community once completed, and would therefore have to be accommodated in the knowledge area description.

Structure of the Document

The following is the proposed breakdown of the topics in the Software Engineering Process Knowledge Area:

1. “Basic Concepts and Definitions”: which establishes the themes and terminology of the KA.
2. “Process Definition”: which describes the purpose and methods for defining software processes, as well as existing software process definitions and automated support.
3. “Process Evaluation”: which describes the approaches for the qualitative and quantitative analysis of software processes.
4. “Process Implementation and Change”: which describes the paradigms, infrastructure, and critical success factors necessary for successful process implementation and change.

1. Basic Concepts and Definitions

- **Themes**

  The main themes in the software engineering process field have been described by Dowson in [25]. His themes are a basis for the manner in which this KA has been structured, except that our scope is slightly larger. Below are Dowson's themes:

  - Process definition: covered in topic 2 of this KA breakdown
  - Process assessment: covered in topic 3 of this KA breakdown
  - Process improvement: covered in topic 4 of this KA breakdown
  - Process support: covered in topic 2 of this KA breakdown

  We also add two themes in this KA description, namely the analytic paradigm to process evaluation (covered in topic 3) and process measurement (covered in topic 3).

- **Terminology**:

  There is no single universal source of terminology for the software engineering process field, but good sources that define important terms are [35][59], and the vocabulary (Part 9) in the ISO/IEC 15504 documents [50].

2. Process Definition

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 [20][46][36]. The types of process definitions required will depend, at least partially, on the reason.

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[1] on the extent to which these are used in practice.

- **Types of process definitions**

  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 [60][47][71].

- **Life cycle models**

  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 [9][19][51][71]). Comparisons of these models are provided in [19][22], and a method for selection amongst many of them in [3].

- **Life cycle process models**

  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 [49] and ISO/IEC 15504 [50][31]. The latter defines a two dimensional model with one dimension being processes, and the second a

[1] There are many opinions, of course.

measurement scale to evaluate the capability of the processes. In principle, ISO/IEC 12207 would serve as the process dimension of ISO/IEC 15504.

- **Notations for process definitions**

  Different elements of a process can be defined, for example, activities, products (artifacts), and resources [46]. Detailed frameworks that structure the types of information required to define processes are described in [65][4].

  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.

  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 [38], in terms of process purpose and outcomes [50], as a list of processes decomposed in constituent activities and tasks defined in natural language [49], Statecharts [54][75] (also see [43]), ETVX [74], Actor-Dependency modeling [11][85], SADT notation [64], Petri nets [5], rule-based [7], and System Dynamics [1]. Other process programming languages have been devised, and these are described in [20][36][46].

- **Process definition methods**

  These methods specify the activities that must be performed in order to define a process model. In general, there is a strong similarity amongst them in that they follow a traditional software development life cycle: [65][64][10][11][55].

- **Automation**

  Automated tools either support the execution of the process definitions, or they provide guidance to humans performing the defined processes.

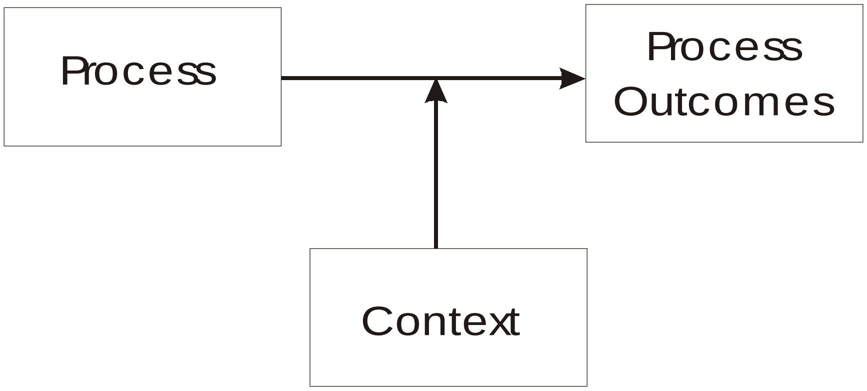
  There exist tools that support all of the above process definition notations. Furthermore, these tools can execute the process definitions to provide automated support to the actual processes, or to fully automate them in some instances. An overview of process modeling tools can be found in [36], and of process-centered environments in [40][41].

  Recent work on the application of the www to the provision of real-time process guidance is described in [56].

3. Process Evaluation

Process evaluation can consist of qualitative analysis of the process or its definition, or a quantitative analysis of data collected about the process. There are a multitude of reasons why one would wish to evaluate the software engineering process. Some of these are covered in the Project Management and other KA’s. Here we focus on process evaluation mainly for the purpose of process improvement.

The assumption upon which most process evaluation work is premised can be depicted by the following path diagram. Here, we assume that the process has an impact on process outcomes. Process outcomes could be, for example, product quality, maintainability, productivity, or customer satisfaction. This relationship depends on the particular context (e.g., size of the organization, or size of the project).



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.

In general, we are not really interested in the process itself, 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.

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.

Measurement plays an important role in process evaluation. Therefore, knowledge of process measurement is considered to be important.

- **Methodology in process measurement**

  A guide for measurement using the G/Q/M method is provided in [79], and the “Practical Software Measurement” guidebook provides another good overview of measurement [69].

  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 [29][42].

  An overview of existing evidence on reliability of software process assessments can be found in [34], and for predictive validity in [42][53][32].

Two general paradigms that are useful for characterizing the type of process evaluation that can be performed have been described by Card [14].

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"*.[2] 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 can provide different sources of information to guide improvement efforts.

[2] Although qualitative evidence also can play an important role.

The analytic paradigm is exemplified by the Quality Improvement Paradigm (QIP) consisting of a cycle of understanding, assessing, and packaging [78]. The benchmarking paradigm is exemplified by the software process assessment work (see below).

We use these paradigms as general titles to distinguish between different types of evaluation.

- **Analytic Paradigm**
    - Qualitative Evaluation

      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 [5][54].

      With this approach, one does not collect data on process outcomes. 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.
    - Root-Cause Analysis

      This involves tracing back from detected problems (e.g., defects) 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 [10][30][18][68].

      With this approach, one starts from the process outcomes, and traces back along the path in the above figure to identify the process causes of the undesirable outcomes.
    - Process Simulation

      The process simulation approach can be used to *predict* process outcomes if the current process is changed in a certain way [75]. Initial data about the performance of the current process needs to be collected, however, as a basis for the simulation.
    - Orthogonal Defect Classification

      Orthogonal Defect Classification is a technique that can be used to link defects found with potential causes. It relies on a mapping between defect types and defect triggers [15][16].
    - Experimental and Observational Studies

      Experimentation involves setting up controlled or quasi experiments in the organization to evaluate processes [63]. 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., [2]).
    - The Personal Software Process

      This defines a series of improvements to an individual's development practices in a specified order [47]. It is 'bottom-up' in the sense that it stipulates personal data collection and improvements based on the data interpretations.
- **Benchmarking Paradigm**

  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 [86]
    - Process assessment models

      Architectures of assessment models

      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 [70].

      Assessment models

The most commonly used assessment model in the software community is the SW-CMM [77]. It is also important to recognize that ISO/IEC 15504 is an emerging international standard on software process assessments [31][50]. 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 [84]. Other notable examples of assessment models are Trillium [13], Bootstrap [82], and the requirements engineering capability model [81] (although, there have been many more capability/maturity models that have been defined, for example, for testing, design, documentation, and formal methods, to name a few). A voiced concern has been the applicability of assessment models to small organizations. This is addressed in [52][76], where assessments models tailored to small organizations are presented.

- Process assessment methods

  Purpose

  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 [27]. In fact, it has been argued that the latter is the most important purpose of doing an assessment [26].

  Assessment methods

  The most well known method that has a reasonable amount of publicly available documentation is the CBA IPI [27]. Many other methods are refinements of this for particular contexts. Another well known method for supplier selection is the SCE [6]. Requirements on methods that reflect what are believed to be good assessment practices are provided in [61][50].

4. Process Implementation and Change

- **Paradigms for process implementation and change**

  Two general paradigms that have emerged for driving process change are the Quality Improvement Paradigm [78] and the IDEAL model [62]. The two paradigms are compared in [78]. A concrete instantiation of the QIP is described in [12].

- **Infrastructure**

  An important element of successful process implementation and change is an appropriate infrastructure, including the requisite roles and responsibilities. Two types of infrastructure are embodied in the concepts of the Experience Factory [8] and the Software Engineering Process Group [37].

- **Guidelines for process implementation and change**

  Process implementation and change is an instance of organizational change. 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 [83][23][57][73][80][76]. An empirical study evaluating success factors for process change is reported in [33].

- **Evaluating process implementation and change**

  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. This issue is concerned with the distinction between cause and effect (as depicted in the diagram above), and is discussed in [12].

  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 [45].

Evaluation implies measurement. See the sub-topic above on methodology in process measurement.

Overviews of how to evaluate process change, and examples of studies that do so can be found in [32][42][53][58][57][63].

Key References vs. Topics Mapping

Below are the matrices linking the topics to key references. In an attempt to limit the number of references, as requested, some relevant articles are not included in this matrix. The reference list above provides a more comprehensive coverage.

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 [28]	SPICE [31]	Pfleeger [71]	CMM [77]	Pressman [72]	Jalote [51]	Fuggetta [39]	Messnarz [66]	Dorfmann [24]	Humphrey [47]
Basic Concepts										
Themes										
Terminology										√
Process Definition			Ch. 2							
Types of Defs.			Ch. 2							√
Life Cycle			Ch. 2		Ch. 2	Ch. 2			Ch. 11	
Life Cycle Process		Ch. 3								
Notations	Ch. 16		Ch. 2				Ch. 1			√
Methods	Ch. 7									
Automation			Ch. 2				Ch. 2			
Process Evaluation										
Analytic	Ch. 7, 14		Ch. 7							
Benchmarking	Ch. 1, 2, 3, 4, 5, 6	√		√						
Methodology	Ch. 8, 10							Ch. 7		
Process Imp. & Change										
Paradigms	Ch. 7									
Infrastructure										
Guideline	Ch. 11, 13, 15						Ch. 4	Ch. 16		
Evaluations	Ch. 7, 8, 9, 10							Ch. 7		

	12207 [49]	15504 [50]	Moore [67]	Finkel. [36]	SEL [78]	SEPG [37]	IDEAL [62]	Wiegers [83]	Madhavji [60]	Dowson [25]
Basic Concepts										
Themes										√
Terminology		√								
Process Definition				√						
Types of Defs.									√	
Life Cycle										
Life Cycle Process	√	√	Ch. 13							
Notations	√	√		√						
Methods										
Automation				√						
Process Evaluation										
Analytic					√					
Benchmarking		√								
Methodology										
Process Imp. & Change							√	√		
Paradigms					√		√			
Infrastructure					√	√				
Guideline								√		
Evaluations					√					

References

[1] T. Abdel-Hamid and S. Madnick: *Software Project Dynamics: An Integrated Approach*. Prentice-Hall, 1991.

[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.

[3] L. Alexander and A. Davis: "Criteria for Selecting Software Process Models". In *Proceedings of COMPSAC'91*, pages 521-528, 1991.

[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.

[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.

[6] R. Barbour: *Software Capability Evaluation – Version 3.0 : Implementation Guide for Supplier Selection*. Software Engineering Institute, Technical Report CMU/SEI-95-TR012, 1996.

[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.

[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.

[9] B. Boehm: "A Spiral Model of Software Development and Enhancement". In *Computer*, 21(5):61-72, 1988.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] D. Card: "Understanding Process Improvement". In *IEEE Software*, pages 102-103, July 1991.

[15] 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.

[16] R. Chillarege: "Orthogonal Defect Classification". In M. Lyu (ed.): *Handbook of Software Reliability Engineering*, IEEE CS Press, 1996.

[17] A. Christie: *Software Process Automation: The Technology and its Adoption*. Springer Verlag, 1995.

[18] 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.

[19] E. Comer: "Alternative Software Life Cycle Models". In M. Dorfmann and R. Thayer (eds.): *Software Engineering*, IEEE CS Press, 1997.

[20] B. Curtis, M. Kellner, and J. Over: "Process Modeling". In *Communications of the ACM*, 35(9):75-90, 1992.

[21] 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.

[22] 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.

[23] 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.

[24] M. Dorfmann and R. Thayer (eds.): *Software Engineering*, IEEE CS Press, 1997.

[25] M. Dowson: "Software Process Themes and Issues". In *Proceedings of the 2nd International Conference on the Software Process*, pages 54-62, 1993.

[26] 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.

[27] 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.

[28] K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

[29] 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.

[30] 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.

[31] K. El Emam, J-N Drouin, W. Melo: *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*. IEEE CS Press, 1998.

[32] 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.

[33] 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.

[34] K. El Emam: "Benchmarking Kappa: Interrater Agreement in Software Process Assessments". In *Empirical Software Engineering: An International Journal*, 4(2), 1999.

[35] 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.

[36] A. Finkelstein, J. Kramer, and B. Nuseibeh (eds.): *Software Process Modeling and Technology*. Research Studies Press Ltd., 1994.

[37] P. Fowler and S. Rifkin: *Software Engineering Process Group Guide*. Software Engineering Institute, Technical Report CMU/SEI-90-TR-24, 1990.

[38] D. Frailey: "Defining a Corporate-Wide Software Process". In *Proceedings of the 1st International Conference on the Software Process*, pages 113-121, 1991.

[39] A. Fuggetta and A. Wolf: *Software Process*, John Wiley & Sons, 1996.

[40] P. Garg and M. Jazayeri: *Process-Centered Software Engineering Environments*. IEEE CS Press, 1995.

[41] 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.

[42] 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.

[43] D. Harel and M. Politi: *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, 1998.

[44] J. Henry and B. Blasewitz: "Process Definition: Theory and Reality". In *IEEE Software*, page 105, November 1992.

[45] 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.

[46] K. Huff: "Software Process Modeling". In A. Fuggetta and A. Wolf: *Software Process*, John Wiley & Sons, 1996.

[47] W. Humphrey: *Managing the Software Process*. Addison Wesley, 1989.

[48] W. Humphrey: *A Discipline for Software Engineering*. Addison Wesley, 1995.

[49] ISO/IEC 12207: *Information Technology – Software Life Cycle Processes*. 1995.

[50] ISO/IEC TR 15504: *Information Technology – Software Process Assessment*, 1998. (parts 1-9; part 5 was published in 1999). Available from http://www.iese.fhg.de/SPICE.

[51] P. Jalote: *An Integrated Approach to Software Engineering*. Springer, 1997.

[52] 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.

[53] 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.

[54] M. Kellner and G. Hansen: "Software Process Modeling: A Case Study". In *Proceedings of the 22nd International Conference on the System Sciences*, 1989.

[55] 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.

[56] 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.

[57] B. Kitchenham: "Selecting Projects for Technology Evaluation". In *IEEE TCSE Software Process Newsletter*, No. 11, pages 3-6, 1998.

[58] 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.

[59] 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.

[60] N. Madhavji: "The Process Cycle". In *Software Engineering Journal*, 6(5):234-242, 1991.

[61] S. Masters and C. Bothwell: *CMM Appraisal Framework – Version 1.0*. Software Engineering Institute, Technical Report CMU/SEI-TR-95-001, 1995.

[62] B. McFeeley: *IDEAL: A User's Guide for Software Process Improvement*. Software Engineering Institute, Handbook CMU/SEI-96-HB-001, 1996.

[63] 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.

[64] C. McGowan and S. Bohner: "Model Based Process Assessments". In *Proceedings of the International Conference on Software Engineering*, pages 202-211, 1993.

[65] 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.

[66] R. Messnarz and C. Tully (eds.): *Better Software Practice for Business Benefit: Principles and Experiences*, IEEE CS Press, 1999.

[67] J. Moore: *Software Engineering Standards: A User's Road Map*. IEEE CS Press, 1998.

[68] T. Nakajo and H. Kume: "A Case History Analysis of Software Error Cause-Effect Relationship". In *IEEE Transactions on Software Engineering*, 17(8), 1991.

[69] 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).

[70] M. Paulk and M. Konrad: "Measuring Process Capability Versus Organizational Process Maturity". In *Proceedings of the 4th International Conference on Software Quality*, 1994.

[71] S-L. Pfleeger: *Software Engineering: Theory and Practice*. Prentice-Hall, 1998.

[72] R. Pressman: *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1997.

[73] J. Puffer: "Action Planning". In K. El Emam and N. Madhavji (eds.): *Elements of Software Process Assessment and Improvement*, IEEE CS Press, 1999.

[74] R. Radice, N. Roth, A. O'Hara Jr., and W. Ciarfella: "A Programming Process Architecture". In *IBM Systems Journal*, 24(2):79-90, 1985.

[75] 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.

[76] M. Sanders (ed.): *The SPIRE Handbook: Better, Faster, Cheaper Software Development in Small Organisations*. Published by the European Comission, 1998.

[77] Software Engineering Institute: *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison Wesley, 1995.

[78] Software Engineering Laboratory: *Software Process Improvement Guidebook*. NASA/GSFC, Technical Report SEL-95-002, 1995.

[79] R. van Solingen and E. Berghout: *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. McGraw Hill, 1999.

[80] 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.

[81] I. Sommerville and P. Sawyer: *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, 1997.

[82] 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.

[83] K. Wiegers: *Creating a Software Engineering Culture*. Dorset house, 1996.

[84] 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.

[85] 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.

[86] S. Zahran: *Software Process Improvement: Practical Guidelines for Business Success*. Addison Wesley, 1998.

SWEBOK Knowledge Area Description for Software Evolution and Maintenance (version 0.5)

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

NOTES FOR REVIEWERS

This section is intended to provide insight to reviewers and will be deleted in the final document.

This is the current draft (version 0.5) of the knowledge area description for software evolution and maintenance. The title of the knowledge area comes from the list of knowledge areas approved by the Industrial Advisory Board for the Guide to the Software Engineering Body of Knowledge Project. This draft builds on version 0.1 and uses a tailored version of the outline provided in the jump-start document. Style and technical guidelines conform to the format for the International Conference on Software Engineering (no more than 10 pages) and follow the IEEE Computer Style Guide. Themes of quality, measurement, tools, and standards are included.

ABSTRACT

This paper presents an overview of the knowledge area of software evolution and maintenance for the Software Engineering Body of Knowledge (SWEBOK) project. A breakdown of topics is presented for the knowledge area along with a short description of each topic. References are given to materials that provide more in-depth coverage of the key areas of software evolution and maintenance. Important knowledge areas of related disciplines are also identified.

Keywords

Software maintenance, software engineering, software evolution, software maintenance measurement, software maintenance planning.

1 INTRODUCTION

This document was prepared in accordance with the "Knowledge Area Description Specifications for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge" dated March 26, 1999. Thus, brief descriptions of the topics are provided so the reader can select the appropriate reference material according to his/her needs.

The topic of this paper is software evolution and maintenance. As many references and common practice is to refer to this topic as maintenance, references in this paper to maintenance refer to software evolution and maintenance.

Software engineering is defined as the application of the systemic, disciplined, quantifiable approach to the development, operation, and maintenance of software. It 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.

2 BREAKDOWN OF TOPICS FOR SOFTWARE EVOLUTION AND MAINTENANCE

Rationale for the Breakdown

The breakdown of topics for software evolution and 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. They 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, along with a rationale for the breakdown, is provided in the next section. The breakdown of topics is contained in Appendix A.

A basic understanding of software evolution and maintenance is needed as the foundation for the remainder of the topics. Thus, the maintenance concepts section provides the definition of maintenance, its basic concepts, and how the concept of system evolution fits into software engineering. Next in importance is what is performed in maintenance. Maintenance activities and roles address the formal types of maintenance and common activities. As with software development, a process is critical to the success and understanding of software evolution and maintenance. Standard maintenance processes are discussed. Appropriate standards are addressed. Organizing for maintenance may be different than for development. Approaches to organizing for maintenance are addressed.

Software evolution and maintenance present unique and different problems for software engineering. These are addressed in the section titled Problems of Software Maintenance. Cost is always a critical topic when discussing software evolution and maintenance. The section on Maintenance Cost and Maintenance Cost Estimation looks at life cycle costs as well as costs for individual evolution and maintenance tasks. Maintenance measurements addresses the topics of quality and metrics. The final topic, tools and techniques for maintenance, aggregates many sub-topics that are not otherwise addressed in this document.

3 BREAKDOWN OF TOPICS

The breakdown of topics, along with a brief description of each, is provided in this section. Key references including standards, are provided.

Maintenance Concepts

Software maintenance is defined as the modification of a software product after delivery to correct faults, to improve performance, or to adapt the product to a modified environment [1]. A maintainer is an organization that performs maintenance activities [2].

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 [3] [4] [5]. 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.

The focus of software development is on producing code that implements stated requirements and operates correctly. Maintenance is different than development [6]. 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 [6] 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.

Need for Maintenance

Maintenance is needed to ensure that the system continues to satisfy user requirements. The system changes due to corrective and non-corrective maintenance. According to Martin and McClure [7], maintenance must be performed in order to:

Correct errors.

Correct design flaws.

Interface with other systems.

Make enhancements.

Make necessary changes to the system.

Make changes in files or databases.

Improve the design.

Convert programs so that different hardware, software, system features, and telecommunications facilities can be used.

The four major aspects that maintenance focuses on are [6]:

Maintaining control over the system's day-to-day functions.

Maintaining control over system modification.

Perfecting existing acceptable functions.

Preventing system performance from degrading to unacceptable levels.

Accordingly, software must evolve and be maintained.

System Evolution

The previous section indicated that there is a need for maintenance, that software is more than fixing bugs, and that the software evolves. The concept of software evolution and maintenance was first addressed by Lehman in 1980. Simply put he stated that maintenance is really evolutionary developments and that maintenance decisions are aided by understanding what happens to systems over time. His "Five Laws of Software Evolution" 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 [6], Sommerville [4], and Arthur [8] have excellent discussions regarding system evolution.

Planning

Software evolution and maintenance should be planned. Whereas developments typically can last for 1-2 years, the operation and maintenance phase, during which software evolution and maintenance occurs, lasts for 5-6 years [6]. Maintenance planning should begin with the decision to develop a new system [5]. A concept and then a maintenance plan should be developed. The concept for maintenance should address:

The scope of software maintenance.

The tailoring of the postdelivery process.

The designation of who will provide maintenance.

An estimate of life-cycle costs.

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 is addressed in IEEE 1219 [1] and ISO/IEC [FDIS] 14764 [9]. ISO/IEC [FDIS] 14764 [9] provides guidelines for a maintenance plan.

Maintenance Activities and Roles

Maintenance activities are similar to those of software development. Maintainers perform analysis, design, coding, testing, and documenting. However, for software evolution and maintenance, the activities involve processes unique to maintenance. Maintainers must possess an intimate knowledge of the code's structure and content [6]. 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 [10]. 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.

Activities

Maintainers must perform other activities than those commonly associated with software development. 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. CM is a critical element of the maintenance process [1]. 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 [5] [12] [13]. Maintenance also includes activities such as planning, migration, and retiring of systems [1] [2] [5] [9].

Categories of maintenance

E. B. Swanson of UCLA was one of the first to examine what really happens in maintenance. He believed that by studying the maintenance phase a better understanding of maintenance would result. Swanson was able to create three different categories of maintenance. These are reflected in IEEE 1219 [1], ISO/IEC [FDIS] 14764 [9], and numerous texts**.** Swanson's categories of maintenance and his definitions are as follows:

Corrective maintenance. Reactive modification of a software product performed after delivery to correct discovered faults.

Adaptive maintenance. Modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment.

Perfective maintenance. Modification of a software product after delivery to improve performance or maintainability.

Common practice is to refer to Adaptive and Perfective maintenance as enhancements. Another type of maintenance, preventive maintenance, is defined in the IEEE Standard on Software Maintenance [1] and the ISO Standard on Software Maintenance [9]. 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.

Of note is that Sommerville [4], Pfleeger [6] 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 "Five Laws of Software Evolution."

Maintenance Process

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

Standards

Schneidewind [11] 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” [1] in 1993. Later the International Organization for Standards (ISO), developed an international standard for software life-cycle processes, ISO/IEC 12207 [2], which included a maintenance process. ISO/IEC 14764 [9] [FDIS] elaborates the maintenance process of ISO/IEC 12207 [2]. All maintainers should develop maintenance processes bases on these standards. Often these are tailored to meet local requirements.

Maintenance Process Models

Process models provide needed operations and detailed inputs/outputs to those operations. Maintenance process models are provided in ISO 1219 [1] and ISO/IEC 14764 [9].

The maintenance process model described in IEEE 1219 [1], 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 2.1.

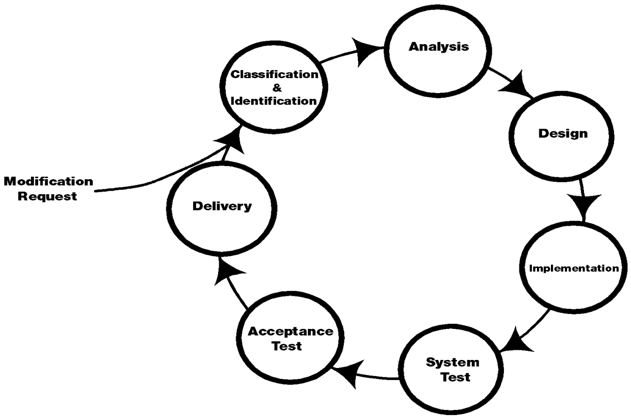


Figure 3.1: The IEEE Maintenance Process

ISO/IEC FDIS 14764 [9] is an elaboration of the maintenance activity from ISO/IEC 12207 [2]. 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 2.2.

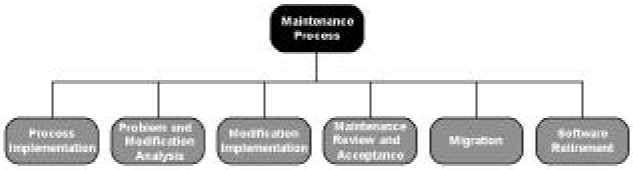

Maintenance Process
Process Implementation
Problem and Modification Analysis
Modification Implementation
Maintenance Review and Acceptance
Migration
Software Retirement

Figure 3.2: IEEE Maintenance Process Activities

Takang and Grubb [12] provide a history of maintenance process models leading up to the development of the IEEE and ISO/IEC process models. A good overview of the maintenance process is given by Sommerville [4].

Organization Aspect of Maintenance

The team that develops the software is not always used to maintain the system once it is operational. Often, a separate team 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 [5] [6] [13]. That decision should be made on a case-by-case basis. Outsourcing of maintenance is becoming a major industry. Dorfman and Thayer [10] provide some guidance in the area of outsourcing maintenance. 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 [5], regardless of the organizational structure.

Problems of Software Maintenance

It is important to understand that software evolution and maintenance provide 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.

Technical

Limited Understanding [6]. 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. Pressman [3] relates that it is often difficlt 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.

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 [10]. Finding time to test is often difficult [6].

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.

Management

Alignment with organizational issues. Dorfman and Thayer [10] relate that return on investment is not clear with maintenance. Thus, there is a constant struggle to obtain resources.

Staffing. Maintenance personnel often are viewed as second class citizens [6] and morale suffers [10]. Maintenance is not viewed as glamorous work [3]. Deklava provides a list of staffing related problems based on survey data [5].

Process issues. Maintenance requires several activities that are not found in software development, e.g. help desk support. These present challenges to management [10].

Maintenance Cost and Maintenance Cost Estimation

Maintenance costs are high due to all the problems of maintaining a system [6]. 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 evolution and maintenance.

Cost

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. Sommerville [4] and Pfleeger [6] address some of the technical and non-technical factors affecting maintenance.

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 [8]. It is performed after a change request enters the configuration management process. It is used in concert with the cost estimation techniques discussed below.

Cost estimation

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.

Parametric models

The most significant and authoritative work in the area of parametric models for estimating was performed by Boehm [14]. 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 [4] [5] [6]. Of significance is that data from past projects is needed in order to use the models. Jones [15] discusses all aspects of estimating costs and provides a detailed chapter on maintenance estimating.

Experience

If a parametric model is not used to estimate maintenance, experience must be used. Sound judgement, reason, a work breakdown structure, educated guesses, and use of empirical/historical data are several approaches. Clearly the best approach is to have empirical data. That data should be provided as a result of a metrics program.

Maintenance Measurements

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 [12]. Software metric is often synonymous with software measurement. Grady and Caswell [16] discuss establishing a corporate-wide metrics program. Software metrics are vital for software process improvement but the process must be measurable.

Takang and Grubb [12] state that measurement is undertaken for evaluation, control, assessment, improvement, and prediction. A program must be established with specific goals in mind.

Establishing a metrics program

Successful implementation strategies were used at Hewlett-Packard [16] and at the NASA/Software Engineering Laboratory [5]. Common to many approaches is to use the Goal, Question, Metric (GQM) paradigm put forth by Basili [17]. This approach states that a metric program would consist of: identifying organizational goals; defining the questions relevant to the goals; and then selecting measures that answer the questions.

The *IEEE Standard For a Software Quality Metrics*

Methodology, ANSI/IEEE 1061 992, [18] 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.

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.

Specific Measures

There are metrics that are common to all efforts and the Software Engineering Institute (SEI) identified these as: size; effort; schedule; and quality [5]. Those metrics are a good starting point for a maintainer. ANSI/IEEE 1061 [18] provides examples of metrics together with a complete example of the use of the standard.

Takang and Grubb [12] group metrics into areas of: size; complexity; quality; understandability; maintainability; and cost estimation.

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 [1] provides suggested metrics for software programs. Stark, et al [17] provides a suggested list of maintenance metrics used at NASA's Mission Operations Directorate.

Tools and Techniques for Maintenance

This section provides a number of related areas that support software evolution and maintenance. Many other topics, e.g., CM, are orthogonal in nature and are covered in other papers in the Guidebook. Tools are crucial for maintaining large systems. Often tools are integrated into software engineering environments to support the maintenance effort. It is one of the more important supporting areas for maintenance. Program comprehension, also referred to as program understanding, is a necessary technique for performing maintenance. The subtopics of re-engineering, and reverse engineering, are important for legacy systems. These are discussed in the subsections that follow.

Maintenance tools

Takang and Grubb [12] define a software tool as any artifact that supports a maintainer in performing a task. Tools have been in use for years. Debuggers and cross-compilers go back to the early days of computing. Tools have evolved over the years and are well known and used heavily in software development. Tools and related environments are less used in maintenance.

Tools have evolved and are now commonly referred to as Computer-Aided Software Engineering (CASE) tools. Often times these are placed in environments which is defined as all of the automated facilities that a software engineer has available for development or maintenance [4].

Software development and maintenance are specialized activities and need separate systems dedicated to them. Takang and Grubb [12] classify maintenance tools based on the specific tasks they support as follows:

Program understanding and reverse engineering

Testing

Configuration management

Documentation and measurement

For program understanding and reverse engineering, tools include the program slicer, static analyzer, dynamic analyzer, and cross-referencer. Tools that support testing include simulators, test case generators, and test path generators. Tools for configuration must perform code management, version control, and change tracking. Documentation tools include hypertext-based tools, data flow and control chart generators, requirements tracers, and CASE tools [12]. Measurement tools are the same ones used in software development.

IEEE 1219 [1] provides a detailed table of methods and tools mapped to the IEEE maintenance activities.

Program Comprehension

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. Based on the importance of this subtopic, an annual IEEE workshop is now held to address program comprehension [10]. 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 [12] provide a detailed chapter on comprehension.

Re-engineering

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. Re-engineering is the most radical (and expensive) form of alteration [10]. It is not undertaken to improve maintainability but is used to replace aging legacy systems. Arnold [19] provides a comprehensive compendium of topics, e.g., concepts, tools and techniques, case studies, and risks and benefits associated with re-engineering.

Reverse engineering

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 redoucumentation. Another type is design recovery [10].

Impact Analysis

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 [8]. It is performed after a change request enters the configuration management process. Arthur [8] states that the objectives of impact analysis are:

Determine the scope of a change in order to plan and implement work.

Develop accurate estimates of resources needed to perform the work.

Analyze the cost/benefits of the requested change.

Communicate to others the complexity of a given change.

4 RECOMMENDED REFERENCES FOR SOFTWARE EVOLUTION AND MAINTENANCE

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.

References

[1] *IEEE STD 1219: Standard for Software Maintenance*

[2] *ISO/IEC 12207: Information Technology-Software Life Cycle Processes*

[3] R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, fourth edition, 1997.

[4] I. Sommerville. *Software Engineering.* McGraw-Hill, fifth edition, 1996.

[5] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing your Software Investment.* Wiley, 1997.

[6] S. L. Pfleeger. *Software Engineering—Theory and Practice.* Prentice Hall, 1998.

[7] J. Martin and C. McClure. *Software Maintenance: The Problem and its Solutions.* Prentice-Hall, 1983.

[8] L. J. Arthur. *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons, 1988.

[9] *ISO/IEC 14764: [FDIS} Software Engineering-Software Maintenance*

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

[11] N. F. Schneidewind. *The State of Software Maintenance*. IEEE, 1987.

[12] A. Takang and P. Grubb. *Software Maintenance Concepts and Practice.* International Thomson Computer Press, 1997.

[13] G. Parikh. *Handbook of Software Maintenance*. John Wiley & Sons, 1986.

[14] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[15] T. C. Jones. *Estimating Software Costs.* McGraw-Hill, 1998.

[16] R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a Company-wide Program*. Prentice-Hall, 1987.

[17] G. E. Stark, L. C. Kern, and C. V. Vowell. *A Software Metric Set for Program Maintenance Management*. Journal of Systems and Software, 1994.

[18] R. S. Arnold. *Software Engineering*. IEEE Computer Society, 1992.

[19] ANSI/IEEE STD 1061. *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Computer Society Press, 1992.

Coverage of the Software Evolution and Maintenance Breakdown topics by the Recommended References.

The cross-reference is shown in Appendix B.

5 KNOWLEDGE AREAS OF RELATED DISCIPLINES

The Guide to the Software Body of Knowledge presents the core Body of Knowledge, i.e., the generally accepted knowledge in the field expected from a graduate with four years of experience. The document "A Baseline for a List of Related Disciplines for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge," dated March 26, 1999 provides a list of potential knowledge areas for software engineers from other disciplines. The following provides a listing of Knowledge

Areas of the Related Disciples that are relevant to the software evolution and maintenance knowledge area. All are keyed to the breakdowns provided in that document.

Computer Science

Foundations

- Complexity analysis
- Complexity classes
- Discrete mathematics
- Program semantics

Algorithms and Data Structures

- Basic data structures
- Abstract data types
- Sorting and searching
- Parallel and distributed algorithms

Computer Architecture

- Digital logic
- Digital systems
- Machine level representation of data
- Number representations
- Assembly level machine organization
- Interfacing and communication
- Alternative architectures
- Digital signal processing
- Performance

Intelligence Systems

- Artificial intelligence
- Agents
- Soft computing

Information Management

- Database models
- Search Engines
- Data mining/warehousing
- Digital libraries
- Transaction processing
- Data compression

Computing at the Interface

- Human-computer interaction
- Graphics
- Vision
- Visualization
- Multimedia
- User-level application generators

Operating Systems

- Tasks, processes and threads
- Process coordination and synchronization
- Scheduling and dispatching
- Physical and virtual memory organizations
- File systems
- Networking fundamentals
- Security
- Protection
- Distributed systems
- Real-time computing
- Embedded systems
- Mobile computing infrastructure

Programming Fundamentals and Skills

- Introduction to programming languages
- Recursive algorithms/programming
- Programming paradigms
- Program-solving strategies
- Compilers/translation
- Code generation

Net-centric Computing

- Computer-supported cooperative work
- Collaboration Technology
- Distributed objects computing
- E-commerce
- Enterprise computing
- Network-level security

Computational Science

- Numerical analysis
- Scientific computing
- Parrallel algorithms
- Modeling and simulation

Social, Ethical, Legal and Professional Issues

- Historical and social context of computing
- Philosophical ethics
- Intellectual property
- Copyrights, patents, and trade secrets
- Risks and liabilities
- Responsibilities of computing professionals
- Computer crime

Mathematics

- Discrete Mathematics
- Calculus
- Probability
- Linear Algebra
- Mathematical Logic

Project Management

- Project Integration Management
- Project Scope Management
- Project Time Management
- Project Quality Management
- Project Human Resource Management
- Project Communications Management
- Project Risk Management
- Project procurement Management

Computer Engineering

- Digital Data Manipulation
- Processor Design
- Digital Systems Design
- Computer Organization
- Storage Devices and Systems
- Peripherals and Communication
- High Performance Systems
- System Design
- Measurement and Instrumentation
- Codes and Standards
- Embedded Systems Software
- Computer Modeling and Simulation

Systems Engineering

Process

- Project Planning
- System breakdown structure
- Design
- Component specification
- Integration
- Maintenance & Operations
- Configuration Management
- Documentation

Essential Functional Processes

- Development
- Test Distribution
- Operations
- Support
- Training
- Disposal

Techniques & Tools

- Metrics
- Privacy
- Process Improvement
- Reliability
- Safety
- Security
- Vocabulary

Management and Management Science

Organizational Environment

- Organizational Characteristics
- Organizational Functions
- Organizational Dynamics

Information Systems Management

- Data Resource Management
- IS Staffing

Training

Management Science

- Optimization
  - Linear Programming
  - Mathematical Programming
- Statistics
- Simulation

Cognitive Science and Human Factors

The Nature of HCI

- (Meta-) Models of HCI

Use and Context of Computers

- Human Social Organization and Work
- Application Areas
- Human-Machine Fit and Adaptation

Human Characteristics

- Language, Communication, Interaction

Computer System and Interface Architecture

- Input and Output Devices
- Computer Graphics

Development Process

Design Approaches

Implementation Techniques

Example Systems and Case Studies

6 SUMMARY

Appendix C identifies the software evolution and maintenance topic areas and the associated Bloom’s taxonomy level of understanding on each topic. The levels of understanding from lower to higher are: knowledge, comprehension, application, analysis, synthesis, and evaluation.

Appendix D applies the Vincenti categorization to the software evolution and maintenance topics.

APPENDIX A – SUMMARY OF THE SOFTWARE EVOLUTION AND MAINTENANCE BREAKDOWN

Maintenance Concepts

- *Need for Maintenance*
- *System Evolution*
- *Planning*

Maintenance Activities and Roles

- *Activities*
- *Categories of Maintenance*

Maintenance Process

- *Standards*
- *Maintenance Process Models*

Organization Aspect of Maintenance

Problems of Software Maintenance

- *Technical*
    - Limited Understanding
    - Testing
    - Impact Analysis
- *Management*
    - Alignment with organizational issues
    - Staffing
    - Process issues

Maintenance cost and Maintenance Cost Estimation

- *Cost*
- *Cost estimation*
    - Parametric models
    - Experience

Maintenance Measurements

- *Establishing a Metrics Program*
- *Specific Maintenance Measures*

Tools and Techniques for Maintenance

- *Maintenance Tools*
- *Program Comprehension*
- *Re-engineering*
- *Reverse Engineering*
- *Impact Analysis*

APPENDIX B - COVERAGE OF THE BREAKDOWN TOPICS BY THE RECOMMENDED REFERENCES

TOPIC	REFERENCE																		
	IEEE 1219	**ISO 12207**	**Pre 97**	**Som 96**	**Pig 97**	**Pfl 98**	**MM 83**	**Art 88**	**ISO 14764**	**DT 97**	**Sch 87**	**TG 97**	**Par 86**	**Boe 81**	**Jon 98**	**GC 87**	**SKV 94**	**AI 92**	**Arn 92**
	[1]	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**	**[10]**	**[11]**	**[12]**	**[13]**	**[14]**	**[15]**	**[16]**	**[17]**	**[18]**	**[19]**
Maintenance Concepts	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							
System Evolution				X	X	X		X											
Planning	X	X			X	X	X		X										
Maintenance Activities and Roles		X			X	X	X	X		X					X				
Activities	X	X			X	X	X	X	X										
Categories of Maintenance	X	X	X	X	X	X	X	X	X	X		X	X						
Maintenance Process	X	X		X	X	X	X	X	X	X	X	X							
Standards	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	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									
Management						X				X		X							
Alignment with organizational issues			X	X		X				X									
Staffing				X	X							X	X						
Process issues										X									

APPENDIX B - COVERAGE OF THE BREAKDOWN TOPICS BY THE RECOMMENDED REFERENCES (Continued)

TOPIC	REFERENCE																		
	IEEE 1219	**ISO 12207**	**Pre 97**	**Som 96**	**Pig 97**	**Pfl 98**	**MM 83**	**Art 88**	**ISO 14764**	**DT 97**	**Sch 87**	**TG 97**	**Par 86**	**Boe 81**	**Jon 98**	**GC 87**	**SKV 87**	**AI 92**	**Arn 92**
	[1]	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**	**[10]**	**[11]**	**[12]**	**[13]**	**[14]**	**[15]**	**[16]**	**[17]**	**[18]**	**[19]**
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				
Parametric models			X	X	X	X								X	X			X	
Experience	X				X									X					
Maintenance Measurements				X	X	X	X		X			X				X	X	X	
Establishing a Metrics Program					X											X	X		
Specific Maintenance Measures			X	X	X	X	X					X					X	X	
Tools and Techniques for Maintenance												X			X				X
Maintenance Tools	X			X	X	X	X		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											

APPENDIX C - BLOOM’S TAXONOMY

TOPIC	BLOOM LEVEL
Maintenance Concepts	Comprehension
Need for Maintenance	Comprehension
System Evolution	Comprehension
Planning	Comprehension
Maintenance Activities and Roles	Comprehension
Activities	Comprehension
Categories of Maintenance	Comprehension
Maintenance Process	Synthesis
Standards	Comprehension
Maintenance Process Models	Synthesis
Organization Aspect of Maintenance	Comprehension
Problems of Software Maintenance	Comprehension
Technical	Synthesis
Limited Understanding	Synthesis
Testing	Synthesis
Process issues	Synthesis
Management	Comprehension
Alignment with organizational issues	Comprehension
Staffing	Comprehension
Impact Analysis	Synthesis
Maintenance cost and Maintenance Cost Estimation	Comprehension
Cost	Comprehension
Cost estimation	Synthesis
Parametric models	Synthesis
Experience	Synthesis
Maintenance Measurements	Synthesis
Establishing a Metrics Program	Comprehension
Specific Maintenance Measures	Synthesis
Tools and Techniques for Maintenance	Synthesis
Maintenance Tools	Synthesis
Program Comprehension	Synthesis
Re-engineering	Synthesis
Reverse Engineering	Synthesis
Impact Analysis	Synthesis

APPENDIX D - VINCENTI CATEGORIZATION

FUNDAMENTAL DESIGN CONCEPTS
Maintenance concepts
Need for maintenance
System evolution
Maintenance activities and roles
Activities
Categories of maintenance
Organization aspect of maintenance

CRITERIA AND SPECIFICATIONS
Maintenance process
Standards
Maintenance Measurements
Establishing a metrics program
Specific measures

THEORETICAL TOOLS
Tools and Techniques for Maintenance
Maintenance tools
Program comprehension
Re-engineering
Reverse engineering
Impact Analysis

QUANTITATIVE DATA
Maintenance Cost and Cost Estimation
Cost estimation
Parametric models

PRACTICAL CONSIDERATIONS
Maintenance concepts
Planning
Problems of software maintenance
Technical
Limited understanding
Testing
Impact analysis
Management
Alignment with organizational issues
Staffing
Process issues
Maintenance Cost and Cost Estimation
Cost estimation
Experience

DESIGN INSTRUMENTALITIES
Maintenance Process
Maintenance process models

KNOWLEDGE AREA: SOFTWARE QUALITY ANALYSIS

The Software Quality Analysis knowledge area of the SWEBOK was defined as covering the following major topics:

(1) Software Quality Assurance
(2) Verification and Validation
(3) Dependability / Quality

Considered together, these subjects are ubiquitous in software engineering, transcending software lifecycles. There needs to be quality analysis and quality oversight on all activities and tasks of the planning, development and support processes. Production of quality products is the key to customer satisfaction. Software without the requisite features and degree of quality is an indicator of failed (or at least flawed) software engineering.

A principle of quality management in any kind of engineering task (and even outside of engineering) is "*avoid* mistakes instead of repairing them". The most important prevention occurs during all the product engineering steps, and this prevention depends on the quality, training, and conscientiousness of the people, the adequacy and appropriateness of the standards, practices, methodologies, and tools, appropriate schedules, and finally, the commitment to improvement of all of the above. Elsewhere in the SWEBOK, knowledge is provided about these issues designed to produce quality software conforming to the needs of the customer. Yet 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. The software engineer who is building the product uses many of the techniques of this section as part of "prevention", or building the quality in. This section of the SWEBOK discusses the knowledge needed for quality analysis as part of SQA and V&V activities. The organizational quality management is addressed in the knowledge areas Software Engineering Management and Software Engineering Process.

The quality analysis process is designed (1) *to provide additional assurance that the product is of the requisite quality and* (2) *to provide management with visibility into the process being used and the product being built*. This visibility can be used not only to improve the product being produced and the process being used in the immediate project, but also to improve the process that will be used in future projects.

Quality oversight includes initial assessment that all selected processes are appropriate for the software product, and then continuous monitoring as these are planned, implemented, and changed when necessary for quality improvement (including new functionality) or when software maintenance is performed. But quality also needs to be defined, since it has many facets. It is, like beauty, "in the mind of the beholder", so the facets that are important need to be decided for the particular environment and purpose for which the software is required and is being designed, as part of the system requirements. The degree of quality in each facet must also be decided, since there are tradeoffs and costs to be considered.

Dependability / quality is an attribute of a system, and although we wish to apply it to the software for the purposes of this exposition, it really refers to the impact of the software on the entire system. Software quality assurance (SQA) is a process designed to assure a quality product; it is a planned and systematic pattern of all actions necessary to provide adequate confidence that the product conforms to specified technical requirements. Software Verification and Validation (V&V) is an important process to provide an objective assessment of software products and processes throughout the software life cycle, that is, the V&V process provides management with visibility into the quality of the product.

This knowledge area contains pointers to many other SWEBOK knowledge areas, such as Software Engineering Process, Software Engineering Management, Software Engineering Infrastructure, Testing, and Requirements Engineering. In each case, we have tried not to describe that knowledge area. For testing, however, there is a fair amount of overlap. This knowledge area of the SWEBOK is arranged as follows:

I. Defining Quality Product

I. DEFINING QUALITY PRODUCT

The goal of software engineering is a quality product, but quality itself can mean different things. There is the set of qualities specifically associated with system dependability such as security, safety, fault tolerance; quality that makes for convenience of use, such as usability, adaptability, and clarity of displays; and quality that facilitates future upgrades and maintenance. Although different terminology is used, there is some consensus about the attributes that define software quality and dependability over a range of products. These definitions provide the base knowledge from which individual quality products are planned, built, analyzed, measured, and improved.

The quality of a given software product is defined as **"the totality of characteristics [of the product] that bear on its ability to satisfy stated or implied needs"** [1]. The needs mentioned are the needs of all parties that hold a stake in the product, including the system users. It is assumed that the needs are obtained from them through the process of Requirements Engineering, which is addressed in another part of the SWEBOK. For the purposes of this section, we stress that facets of quality must be addressed in the requirements and discussed with the stakeholders. A

[1] From *Quality—Vocabulary*, Draft Intl. Standard 8402, International Organization for Standardization, Geneva, 1993.

thorough knowledge of these facets, some quantification of their priority and the degree to which they are expected allows the software engineer(s) to plan and budget a project and go into the product development process with high confidence of satisfying the needs, or alternatively, to return to the stakeholders for further discussion. For their part, the stakeholders may not be aware of the implications of what they expect (costs, tradeoffs, alternatives) and they may not have considered how the quality of the system they expect will be described or measured.

Some standard descriptions of software quality and quality attributes are discussed. But it is well to remember that a particular aspect of quality that is important for one piece of software may be less important or even irrelevant for another. The quality definition for a given product included in the requirements must take into consideration a number of constraints, some of which are imposed by the environment and purpose for which the software developed and some of which may be firm quality constraints. In addition, as seen in references for measurement in Section II, metrics for quality attributes may be based on slightly different subcharacteristics for the primary quality attributes. Viewpoints, or facets, about quality may not always be clear to the stakeholders, the realization of quality and dependability attributes are to be optimized subject to the environment, to the firm quality constraints, and to tradeoffs with one another**.**

A. ISO9126 Quality Characteristics

The software quality characteristics identified in this section with each major attribute are those given in the ISO9126 standard. However, other authors provide different subsets, and often provide metrics for those subcharacteristics. These definitions and metrics are not provided here, or in the measurement section, but are provided in the reference material. For example, functionality could also be characterized by completeness, correctness, security, compatibility, interoperability and portability could be measured by hardware independence, software independence, installability, and reusability.

1. Functionality

Functionality is in some sense more fundamental to the working of the system than other attributes. The definition in ISO/IEC 9126 of Functionality implies that the functionality is the short answer to the question "What do you need for this software to do?" without going into details of how it is going to accomplish it. From the quality point of view, the execution of the functionality is the baseline as to whether the needs of the customer are accomplished. After that, the other quality attributes are to be realized as well as possible, but may be compromised for economy purposes and traded off against one another. The attributes of functionality are defined as follows:

- **Suitability:** Attribute of software that bears on the presence and appropriateness of a set of functions for specified tasks. (ISO 9126: 1991, A.2.1.1)
- **Accuracy**: Attributes of software that bear on the provision of right or agreed results or effects. (ISO 9126: 1991, A.2.1.2)
- **Interoperability:** Attributes of software that bear on its ability to interact with specified systems. (ISO 9126: 1991, A.2.1.3) (NOTE -- Interoperability is used in place of compatibility in order to avoid possible ambiguity with replaceability. (ISO 9126: 1991, A.2.1.3)
- **Compliance:** Attributes of software that make the software adhere to application-related standards or conventions or regulations in laws and similar prescriptions. (ISO 9126: 1991, A.2.1.4)
- **Security:** Attributes of software that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs and data. (ISO 9126: 1991, A.2.1.5)

2. Reliability

Reliability is a set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time. (ISO 9126: 1991, 4.2) [NOTE: Wear or aging does not occur in software. Limitations in reliability are due to faults in requirements, design and implementation. Failures due to these faults depend on the way the software product is used and the program options selected rather than on elapsed time. (ISO 9126: 1991, 4.2)]. Subcharacteristics include the following:

- **Maturity**: Attributes of software that bear on the frequency of failure by faults in the software. (ISO 9126: 1991, A.2.2.1)

- **Fault tolerance**: Attributes of software that bear on its ability to maintain a specified level of performance in cases of software faults or of infringement of its specified interface. (ISO 9126: 1991, A.2.2.2)
- **Recoverability:** Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it. (ISO 9126: 1991, A.2.2.3)

3. Usability

Usability is a set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users. [NOTES: 1. ``Users" may be interpreted as most directly meaning the users of interactive software. Users may include operators, and users and indirect users who are under the influence of or dependent on the use of the software. Usability must address all of the different user environments that the software may affect, which may include preparation for usage and evaluation of results. 2.Usability defined in this International Standard as a specific set of attributes of a software product differs from the definition from an ergonomic point of view, where other characteristics such as efficiency and effectiveness are also seen as constituents of usability.] Usability subcharacterisitcs include the following:

- **Understandability:** Attributes of software that bear on the users' effort for recognizing the logical concept and its applicability. (ISO 9126: 1991, A.2.3.1)
- **Learnability:** Attributes of software that bear on the users' effort for learning its application (for example, operation control, input, output). (ISO 9126: 1991, A.2.3.2)
- **Operability:** Attributes of software that bear on the users' effort for operation and operation control. (ISO 9126: 1991, A.2.3.3)

4. Efficiency

Efficiency is a set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions. (ISO 9126: 1991, 4.4) [NOTE -- Resources may include other software products, hardware facilities, materials, (e.g. print paper, floppy disks) and services of operating, maintaining or sustaining staff. (ISO 9126: 1991, 4.4)] Efficiency subcharacteristics include the following:

- **Time behavior**: Attributes of software that bear on response and processing times and on throughput rates in performing its function. (ISO 9126: 1991, A.2.4.1)
- **Resource behavior**: Attributes of software that bear on the amount of resources used and the duration of such use in performing its function. (ISO 9126: 1991, A.2.4.2)

5. Maintainability

Maintainability is a set of attributes that bear on the effort needed to make specified modifications. (ISO 9126: 1991, 4.5) [NOTE -- Modifications may include corrections, improvements or adaptation of software to changes in environment, and in requirements and functional specifications. (ISO 9126: 1991, 4.5)] Maintainability includes the following subcharacteristics:

- **Analyzability:** Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified. (ISO 9126: 1991, A.2.5.1)
- **Changeability:** Attributes of software that bear on the effort needed for modification, fault removal or for environmental change. (ISO 9126: 1991, A.2.5.2)
- **Stability:** Attributes of software that bear on the risk of unexpected effect of modifications. (ISO 9126: 1991, A.2.5.3)
- **Testability:** Attributes of software that bear on the effort needed for validating the modified software. (ISO 9126: 1991, A.2.5.4)

6. Portability

Portability is a set of attributes that bear on the ability of software to be transferred from one environment to another. (ISO 9126: 1991, 4.6) [NOTE: The environment may include organizational, hardware or software environment. (ISO 9126: 1991, 4.6)]. Portability includes the following subcharacteristics:

- **Adaptability:** Attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered. (ISO 9126: 1991, A.2.6.1)
- **Installability:** Attributes of software that bear on the effort needed to install the software in a specified environment. (ISO 9126: 1991, A.2.6.2)
- **Conformance:** Attributes of software that make the software adhere to standards or conventions relating to portability. (ISO 9126: 1991, A.2.6.3)
- **Replaceability:** Attributes of software that bear on the opportunity and effort of using it in the place of specified other software in the environment of that software. (ISO 9126: 1991, A.2.6.4) [NOTES: 1.Replaceability is used in place of compatibility in order to avoid possible ambiguity with interoperability. 2.Replaceability with a specific software does not imply that this software is replaceable with the software under consideration. 3.Replaceability may include attributes of both **installability** and **adaptability**. The concept has been introduced as a subcharacteristic of its own because of its importance.

B. Dependability

For many systems, dependability of the overall system (hardware, software, and humans) is the main goal after the basic functionality is realized, and since software dependability is a major factor in that attribute, we define it separately. There are sometimes tradeoffs in emphasizing one quality attribute over another, and there are costs associated with achieving high quality. 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 (see above). Reliability is defined similarly, but not identically, in the two places. In Moore’s treatment, Kiang’s factors are used, and we have used them also, but added Trustability from Laprie.

- **Availability:** The product’s readiness for use on demand
- **Reliability:** The longevity of product performance
- **Maintainability**: The ease of maintenance and upgrade
- **Maintenance support:** Continuing support to achieve availability performance objectives
- **Trustability:** System’s ability to provide users with information about service correctness

Dependability is one way of expressing quality principally for systems requiring the highest integrity; that is, their failure may cause grave consequences with respect to safety and / or security. The following are supporting definitions for quantifying dependability factors:

- **Threat (or hazard):** A state of the system or its environment that can have adverse effects. Threat generally refers to security, while hazard refers to safety.
- **Risk:** A combination of the probability of the threat and the adverse consequences of its occurrence.
- **Risk Analysis:** Study of threats or hazards, their frequency, and their consequences.
- **Integrity:** The capability of a system and software to contain risk.
- **Software Integrity Level: A** software property necessary to maintain system risks within acceptable limits. For software that performs a [risk] mitigating function, the property is the reliability with which the software must perform the mitigating function. For software whose failure can lead to a system threat, the property is the frequency or probability of that failure (ISO DIS 15026).

C. Quality Facets Related to Process and Special Situations

Various other traits of software are considered important enough to be evaluated. Among these are ones that make software useful at other parts in the current project or in future projects. For instance, one wants whole modules or objects to be reusable. One wants software to be traceable to requirements insofar as possible (this may be thought of as a version of understandability), and one knows that generally software is more easily understood and less liable to puzzling errors if the individual software components have clean interfaces and are cohesive. So these attributes are often included in planning and evaluation for quality purposes:

- **Code/Object Reusability**
- **Requirements Traceability**
- **Module Cohesion and Coupling**

There are special situations in which one needs additional quality criteria with some types of systems that may not be common to other software. The software engineer has to be aware that these exist, in order to be able accurately to analyze the quality of the product. But if near real time performance is needed and the system requires the best answer available to that time, certain software can guarantee the best answer that is available by that time. It is said to have an "anytime" property.
Since software is very versatile and requirements reflect this versatility regularly, it is hard to anticipate the requirements, so we just list a few examples of these special types of systems:

- **Intelligent and Knowledge Based Systems** - Anytime property, Explanation Capability
- **Human Interface and Interaction Systems** - Adaptivity, Intelligent Help, Display Salience
- **Information Systems** - Ease of query, High Recall, High Precision
- **Client/Server Systems** – Efficient download

II. Software Quality Analysis

While the previous section, Defining Quality, provides a body of knowledge about the facets of software quality, this section discusses a body of knowledge for software quality analysis. Software quality analysis consists of the processes to analyze whether the software products under development or maintenance meet the definition of quality as tailored to the particular project requirements. Planning for a specific quality product and its analysis takes into consideration many factors. Topics include, but are not limited to, (1) the environment of the system in which the software will reside; (2) the system and software requirements; (3) the standard components to be used in the system; (4) the specific software standards used in developing the software; (5) the software standards used for quality; (6) the methods and software tools to be used for development; (6) the methods, tools, and measurements used for quality; (7) the budget, staff, project organization, plans and schedule, and (8) the intended users and use of the system.

Some quality attributes (e.g., safety, security) may be in conflict with each other or with other functional requirements, and tradeoffs may be needed. Techniques may be used in the planning process to help identify where potential trouble spots may be. Hazard and threat analyses help to identify potential problems for safety and security. Other techniques help to control the product and changes to it. The possible consequences of failure of the system and the probability of failure are major factors in defining the degree and scope of its quality requirements. Planning identifies how all of the means available can meet the needs, as stated in the quality requirements, while ensuring they do not contradict the systems requirements and mandatory standards. Planning also identifies additional resources and techniques that may be needed beyond those originally proposed for the product.

A. Definition of Quality Analysis

While quality is the obligation and responsibility of every person on the project, quality analysis consists of processes that evaluate risks, ensure plans are implemented properly, and examines work products for detects. Quality analysis supports the goal of quality to provide an organized framework that prevents the introduction of defects in a work product. But, there will be risks on a project, and there will be mistakes. While the management of risk is the responsibility of management, the identification of risk items, and the presentation of their likelihood and expected impact in an understandable and practical manner is the responsibility of the quality analysis program. The quality analysis program exercises processes to detect defects before the work product containing them enters the next effort. The two principal processes are software quality assurance (SQA) and verification and validation (V&V).

From the perspective of quality analysis, factors such as those identified in Section I affect the type of quality analysis program planned for the product. Quality programs for the organization at large are discussed in the sections on Software Engineering Process and Software Engineering management. In this section, the issue is quality analysis for a product, where the term <u>product</u> means any output, or artifact, of any process used to build the final software product. In this sense, a product can be a software requirements specification for one software component of a system, or an entire system specification, or a design module, code, test documentation, reports from quality analysis tasks, etc.

The role of quality analysis in building the quality product is to provide management with visibility through the performance of tasks that assess (examine and measure) the quality of the outputs of the software development processes while they are developed. These include assessment activities of SQA and V&V that can be performed both without (static) and with (dynamic) program execution. They may be performed by different people, for example, by an individual designer, a design team, or by an independent peer or group.

1. Software Quality Assurance

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 considering the factors above to ensure that the problem is clearly and adequately stated and that the solution's requirements are properly defined, and expressed. Then SQA keeps the quality maintained throughout the development of the product.

2. Software Verification and Validation

The V&V process determines whether development products of a given 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.[2] V&V provides an examination of every product relative both to its immediate predecessor and to the system requirements it must satisfy.

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.

For each process, (SQA or V&V), we discuss plans, activities and techniques, and measurement.

B. Process Plans

1. SQA Plan

The SQA plan defines the processes and procedures that will be used to ensure that software developed for a specific product is of the highest quality possible and meets its requirements. This plan may be governed by software quality standards, life cycle standards, quality management standards, and company policies and procedures for quality and quality improvement. This plan defines SQA tasks, their management, and their schedule in relation to tasks in the software development plan. The 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, conventions, and metrics that govern the project, how they will be checked and monitored to ensure adequacy or compliance. The SQA plan identifies the procedures for problem reporting and corrective action, resources such as tools, techniques and methodologies, security for physical media, training, SQA documentation to be retained. The SQA plan addresses assurance of other supplier software to the project. The SQA plan also addresses risk management methods and procedures.

2. V&V Plan

[2] IEEE Std. 1012-1998

Verification activities yield confirmation by examination of a specific product, that is, output of a process, and provisions of objective evidence that specified requirements have been fulfilled. Here, these requirements refer to the requirements of that product, relative to the product from which it is derived; e.g., code is developed to satisfy the requirements of a design description for a specific module.   Validation yields confirmation by examination of a specific product and provisions of 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. Verification and validation 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. The V&V plan is the instrument that explains how each technique will satisfy the objectives of V&V.

C. Activities and Techniques for Quality Analysis

Other sections of the SWEBOK address the actual construction of the software product, that is, the specification of the software requirements and the design and construction of the product based on the specification. In the SQA or V&V process, SQA and V&V results are collected into a report for management before corrective actions are taken. The management of quality analysis is tasked with ensuring the quality of these reports, that is, that the results are accurate.

The specific tasks a software engineer performs in quality analysis may depend upon both his 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 any quality analysis activities is aware of and understands all considerations affecting quality analysis: standards for quality analysis, the various resources that influence the product, techniques, and measurement (e.g., what to measure and how to evaluate the product from the measurements). Other sections of the SWBOK address the software engineering process.

There are two principal categories for quality analysis:

- Static techniques (no code execution)
- Dynamic techniques (code execution).

Both are used in SQA and in V&V. Specific objectives and organization depend on the project and product requirements.

1. Static Techniques

Static techniques may be people intensive activities or analytic activities conducted by individuals, with or without the assistance of automated tools. These support both SQA and V&V. Their specific implementation can serve the purpose of SQA, verification, or validation, at every stage of development. These static analysis techniques frequently apply to more than one type of product and seem to fit into these groups:

- People intensive (conducted in some type of meeting)
- Analytic techniques for direct defect finding (intended to find faults)
- Analytic techniques to provide support or produce preliminary findings for use with other static or dynamic techniques.

a. People Intensive Static Techniques

Two or more people perform these techniques in a meeting. Preparation may be performed ahead of time. Resources may include checklists and results from analytic techniques and testing. The subject of the technique is not necessarily the completed product, but may be a portion at any stage of its development. For example, a subset of the software requirements may be reviewed for a particular set of functionality, or several design modules may be inspected, or separate reviews may be conducted for each category of test for each of its associated documents (plans, designs, cases and procedures, reports).   With the exception of the walkthrough, these 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.

Management reviews are discussed in the knowledge area Software Engineering Management. It should be noted that management may request review support of various plans (e.g., contingency, Software Configuration Management, software V&V) and reports (e.g., risk assessment, anomaly reports) by software engineers who are performing quality analysis tasks.

	Concept	**Reqmt.**	**Design**	**Code**	**Test**	**Install.**	**Operat.**	**Mainten.**
Audit		X	X	X	X	X	X	X
Inspection	X	X	X	X	X			
Review		X	X	X	X	X		X
Walkthrough		X	X	X	X			

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. Software inspections generally involve the author of a product, while reviews likely do not. Inspections are usually conducted on a relatively small section of the product, in a couple hour sessions, whereas reviews and audits are usually broader. The reviews that fall under SQA are technical reviews, that is, on technical products. Technical reviews and inspections examine products such as software requirement specifications, software design documents, test documentation, user documentation, installation procedures but the coverage of the material varies with the technique. The walkthrough is conducted by members of the development group to examine a specific product.

Specific reviews and audits are sometimes known by the following names:

- System Requirements and System Design [3]
- Software Specification (requirements)
- Preliminary Design
- Detailed Design
- Test Readiness
- Post-implementation
- Functional Audit
- Physical Audit

b. Analysis Intensive Static Techniques

These techniques are generally applied individually, although it may be that several people are assigned the technique, in which case a review may be conducted on the results. Some are tool-driven; others are primarily manual. Most techniques listed as support may of course find some defects directly but are typically used as support to other techniques. Some however 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.

2. Dynamic Techniques

Different kinds of testing are performed throughout the development of software systems. Some testing may fall under the SQA process, and some under V&V, again depending on project organization. Because both the SQA and

[3] Included for completeness, although these are at system level. Software quality analysis may be involved.

V&V plans address testing, this section includes some commentary about testing, but technical references to theory, techniques for testing, and automation are fully provided in the knowledge area on Software Testing.

The quality analysis process examines 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 outputs of the development processes, that is, the analysis consists of validating the code by testing to many objectives and strategies, and collects, analyzes and measures the results. SQA ensures that appropriate types of tests are planned, developed, and implemented, and V&V develops plans, strategies, cases and procedures.

Two types of testing fall under SQA and V&V because of their responsibility for quality of materials used in the project:

- Evaluation and test of tools to be used on the project
- Conformance test (or review of conformance test) of components to be used in the product.

The SWEBOK knowledge area on Software Testing addresses special purpose testing such as usability testing. Many of these types are also considered and performed during planning for SQA or V&V testing. We list them here only for completeness:

- Certification test
- Conformance test
- Configuration test
- Installation test
- Reliability test
- Usability test
- Security test
- Safety test
- Statistical test
- Stress test
- Volume test

Occasionally the V&V process may be asked to perform these other testing activities:

- Test monitoring
- Test witnessing
- Test evaluation

Supporting techniques for testing fall under test management, planning and documentation. And V&V testing generally includes component or module, integration, system, and acceptance testing.

D. Measurement in Software Quality Analysis

The words “error”, “fault”, and “failure” are often used interchangeably and the word “defect” may be used in place of any of these in describing software problems.   IEEE Std 610.12-1990 (“IEEE Standard Glossary of Software Engineering Terminology”) also gives the terms multiple, but more precise meanings, among which we will use the following in our discussions:

- Error: “A difference…between a computed result and the correct result”
- Fault: “An incorrect step, process, or data definition in a computer program”
- Failure: “The [incorrect] result of a fault”
- Mistake: “A human action that produces an incorrect result”.

We will try to be consistent by using the terms as defined above in this section on measurement for quality analysis, which is concerned with faults discovered through the quality analysis process. Mistakes (as defined) are the subject of the quality improvement process, which is covered in the Knowledge Area Software Engineering Process. Failures found in testing that are the results of software faults are also of interest in this section, but we will use "Fault" as the primary term. It is the fault - its discovery, its measurement, and its removal from any software product - that is the subject of measurement for quality analysis. “Defect” (not defined in 610.12-1990) is considered here to be the same as fault. Note that "Failure" is the term used in reliability models, because these models are built from failures (in the sense of the system being unable to perform its requirements) of systems during testing.

Quality may be defined and planned for a project and its product, and various activities of quality analysis may be applied to all the products related to the project, but without measurement one still does not know enough about the product. Evaluations are effective when there is measurement against which to base the evaluations.

1. Fundamentals of Measurement

Theories of measurement establish the foundation on which meaningful measurements can be made. Measuring implies numbers, and various scales apply to different types of data. Measurement models and frameworks for software quality enable the software engineer to establish specific product measures as part of the product concept. These are discussed in the following references:

- Theory
- Scales
- Models
- Framework

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 seems to be the basis for many software measurement programs. Other important elements deal with experimentation and data collection. A measurement program has costs in time and money, but may also result in savings, and methods exist to help estimate the costs of a measurement program. Key topics for measurement planning are:

- GQM
- Example of applied GQM
- Experimentation
- Methods
- Costing
- Data Collection process
- Examples of company metrics programs

2. Metrics

Data can be collected on various characteristics of software products. Many of the metrics are related to the quality characteristics defined in Section I 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:

- Quality Characteristics Measures
- Reliability Models & Measures
- Defect Features, .e.g., Counts, density
- Customer satisfaction
- Product features:
  - Size (SLOC, function points, number of requirements)
  - Structure metrics (e.g., modularity, complexity, control flow)

- Object-Oriented Metrics

3. Measurement Techniques

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:

- Statistically based (e.g., Pareto analysis, run charts, scatter plots, normal distribution)
- Statistical tests (e.g., binomial test; chi-squared test)
- Trend analysis
- Prediction, e.g., Reliability models

4. Defect Characterization

The quality analysis processes may discover defects. Part of the quality analysis is to characterize those defects to enable understanding of the product, to facilitate corrections to the process or the product and to inform the project management or customer. Defects can be described in several manners:

- Defect Classification and Descriptions
- Defect Analysis
- Measuring adequacy of the quality analysis activities
- Test coverage
- Benchmarks, profiles, baselines, defect densities

5. Additional quality analysis concerns that measurement supports

- When to stop testing
- Debugging (McConnell)
- SQA and V&V reports

III. Knowledge topics and Reference Matrices

A. References Keyed to Text Sections

REFERENCES: **Section I.A** **Defining Software Quality**	Dorfman	Fenton&Pf.	Kiang	Laprie	Lewis	Lyu	Moore	Musa	Pfleeger	Pressman	Ratikin	Sommerville	Wallace	
Functionality											X			
Suitability														
Accuracy														
Interoperability														
Compliance														
Security														
Reliability						x	X	X	X	X	X	X	X	
Maturity														
Fault Tolerance						X		X						
Recoverability														
Efficiency						X	X			X				
Time behavior														
Resource behavior														
Usability		X					X		X	X	X	X		
Understandability					X				X	X		X		
Learnability														
Operability														
Maintainability		X	X				X		X	X		X		
Analyzabiity														
Changeability														
Stability														
Testability					X				X	X				
Portability							X		X	X	X			
Adaptability												X		
Installability														
Conformance														
Replaceability														
Dependability				X		X	X	X				X	X	
Availability		x				X	X	X	X	X		X		
Reliability						X	X	X	X	X		X	X	
Trustability				X		X		X						
Integrity						X	X	X	X	X				
Other facets									X	X				
Code/object Reusability							X		X	X		X	X	
Reqts. Traceability	X				X				X	X		X	X	
Explanation Capability													X	

REFERENCES: **Section II.A and II.B** **Definition &** **Planning of Quality** **Analysis**	Grady	Horch	Kazman	Lewis	Lyu	McConnell	Moore	Musa	Pfleeger	Pressman	Ratikin	Schulmyer	Sommerville	Wallace/Fujii	Wallace
Overall							X		X	X			X		
SQA		X		X		X	X		X	X	X	x	X		
VV				X			X		X	X	X	x	X	x	X
Independent V&V				X					X	X	x		X	x	X
Hazard, threat anal.							X		X	X			X		X
Risk assessment	x	X		X	x		X	X					X		
Performance analysis			X						X	X					

REFERENCES: **Overall Quality Checks**	Kan	Lewis	Musa	Pfleeger	Pressman	Ratikin	Sommerville	Wakid
Cleanroom	X		x	X	X	x	X	
Config. Mgt. Assessment		X		X	X		X	X
Distr. Arch. Assessment				x				
Disast. Recov. Plan Assess.				X	X			
Evaluation of New Constraints								
Reusability Assessment								
Security Assessment								

REFERENCES: **Section II.C.1.a** **People-Intensive** **Review Techniques**	Ackerman	Grady	Horch	McConnell	Pfleeger	Pressman	Ratikin	Schulmyer	Sommerville	Wall/Fujii	Wallace	Lewis
Audit			X				x			X		X
Inspection	x	x	X	X	X	X	X	X	X	X	X	
Review			X	X	X	X			X	X	x	X
Walkthrough			X	X	X	X			X	x	X	

REFERENCES: **Direct Defect Finding Techniques**	Beizer	Fent;F7Pf	Fried./Voas	Hetzel	Horch	Ippolito	Leveson	Lewis	Lyu	Moore	Musa	Pfleeger	Pressman	Ratikin	Rubin	Schulmeyer	Sommerville	Wakid	Wallace
Algorithm Analysis		X		X				X										X	X
Boundary Value Anal.			X									X	X				X	X	X
Change Impact Anal.									X			X	X	X		X	X		
Checklists					X									x					
Consistency Analysis							X						X			X			
Control Flow Analysis	X	X						X	X			X	X					X	X
Database Analysis	X	X	X					X							X			X	X
Data Flow Analysis	X	X	X					X	X	X					X			X	X
Distrib. Arch. Assess.													X						
Evaluation of Doc.: Concept, Requirements			X					X	X						X			X	X
Evaluation of Doc.: Design, Code, Test			X					X	X						X			X	
Evaluation of Doc.: User, Installation			X					X	X						X			X	
Event Tree Analysis			X																X
Fault Tree Analysis			X			X			X	X					X				
Graphical Analysis	X	X									x							X	
Hazard Analysis		X	X			X	X		X	X					X				
Interface Analysis	X		X		X			X	X						X				X
Formal Proofs			X						X	X					X				X
Mutation Analysis			X						X									X	X
Perform. Monitoring									X										X
Prototyping			X						X	X					X				X
Reading			X																X
Regression Analysis			X		X			X	X									X	X
Simulation			X																X
Sizing & Timing Anal.			X					x	X	X								X	X
Threat Analysis									X	X					X				

REFERENCES: **Support to Other Techniques** An= Analysis	Beizer	Conte	Fried./Voas	Hetzel	Houk	Leveson	Lewis	Lyu	Musa	Pfleeger	Pressman	Ratikin	Rubin	Sommerville	Freid/Voas	Wallace/Fuji	Wallace
Change Impact An								X		X	X			X			
Checklists				X	X		X					X	X				
Complexity An	X	X						X	X		X						
Coverage An	X							X	X								
Consistency An										x	X		X				
Criticality An						X	X				X						X
Hazard An			X			X					X	X		X			
Sensitivity An			X												X		
Slicing	X														x		X
Test documents	X	x						X	X							X	x
Tool evaluation							X	X								X	
Traceability An							X	X		X				X		X	X
Threat An			X					X		X	X			X			

REFERENCES: **Testing Special to SQA or V&V**	Fried./Voas	Leveson	Lyu	Musa	Pfleeger	Pressman	Ratikin	Rubin	Schulmeyer	Sommerville	Voas	Wakid	Wallace/Fuji
Conformance Test.	X											X	
Configuration Test.						X							
Certification Testing			X	X		X	X		X		X	X	
Reliability Testing	X	X	X	X					X				
Safety Testing	X		x	x					X				
Security Testing					X								
Statistical Testing			X	x	X	X			X	X			
Usability Testing					X			X					
Test Monitoring													X
Test Witnessing													X

REFERENCES: **Section II.D** **Measurement in Software Quality Analysis**	Basili	Beizer	Conte	Chillarage*	Fenton	Fenton/Pfleg.	Fried./Voas	Grady	Hetzel	Horch	Kan	Musa/Acker. man	Lewis	Lyu	Musa	Peng	Pfleeger	Pressman	McConnell	Ratikin	Schulmeyer	Salamonn	Sommerville	Wakid	Zelkowitz
Benchmarks, profiles, etc.		X				X											X		X		X				
Company Metric Progs.					X	X		X			X						X	X							
Costing		x	x					X		X			X		X		X		X	X	X		X		
Customer satisfaction											X							X							
Data Collection process	X		X		X	X		X																	
Debugging		x	x			X								X					X				X		
Defect Analysis		x x	x	X			X	X	X	X	X			X	X	X	X	X	X						
Defect Classif. and Descr.		X		X		X	X	X		X	X		X	X	X		X	X							
Defect Features			X	X		X		X		X	X			x						X					
Example of applied GQM						X		X																	
Experimentation:			X	x	X	X									X										X
Framework					X	X																			
GQM	X				X	X		X			X												X		
Methods			X		X			X			X			X	x			x							
Metrics			X			X		X		X	X			X		X	X	X		X	X	X	X		
Models					X	X								X	x										
Prediction						X					X			X	X					X					
Prod. features: O/O Metr.																					X				
Prod. Features: Structure			X		X	X		X						X						X					
Product features: Size			X			X		X			X			X											
Quality Attributes													X				X	X					X		
Quality Character. Meas.						X							X			X				X					
Reliab. Models & Meas.			X			X	X				X			X	X				X	X					
Scales			X		X	X					X														
SQA & V&V reports ***										X										X				X	
Statistical tests			X			X								X		X	X						X		
Statistical Analysis & measurement			X		x	X				X	X			x	x	X				X					
Test coverage															X					X					
Theory			X		X	X					X														
Trend analysis														X											
When to stop testing							X					X			X *										

* Also see Reference 22, Musa-Ackerman

B. Standards

Standards for quality analysis address quality definitions, quality analysis planning specifically for SQA and V&V processes and their documentation, security and safety tests, and measurement. For the most part, the standards do not specify how to conduct specific techniques for quality analysis. One exception is the standard on software reviews (IEEE Std 1028). Other standards may either govern the software quality analysis process (e.g., ISO 12207) or enter into the planning process for software quality analysis, and these are addressed elsewhere in the SWEBOK, e.g. Software Configuration Management.

	Quality Requirements & planning	Reviews/ Audits	SQA/ V&V planning	Safety/ security analysis, tests	Documentation of quality analysis	Measurement
ISO 9126	X					
IEC 61508	X					
ISO/IEC 15026	X					
ISO FDIS 15408	X			X		
FIPS 140-1	X			X		
IEEE 730		X	X		X	
IEEE 1008			X			
IEEE 1012		X	X	X	X	
IEEE 1028		X				
IEEE 1228				X		
IEEE 829					X	
IEEE 982.1,.2						X
IEEE 1044						X
IEEE 1061						X

IV. References

A. Basic SWEBOK References

1. Dorfman, M., and R. H. Thayer, *Software Engineering.* IEEE Computer Society Press, 1997. [D]
2. Moore, J. W., *Software Engineering Standards: A User's Road Map*. IEEE Computer Society Press, 1998. [M]
3. Pfleeger, S. L., *Software Engineering – Theory and Practice*. Prentice Hall, 1998. [Pf]
4. Pressman, R. S., *Software Engineering: A Practitioner's Approach* (4th edition). McGraw-Hill, 1997. [Pr]
5. Sommerville, I., *Software Engineering* (5th edition). Addison-Wesley, 1996. [S]
6. Vincenti, W. G., What Engineers Know and How They Know It – Analytical Studies form Aeronautical History. Baltimore and London: John Hopkins, 1990. [V]

B. Additional References

1. Ackerman , Frank A., "Software Inspections and the Cost Effective Production of Reliable Software," [Dorf] pp. 235-255
2. 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.
3. Beizer. Boris, *Software Testing Techniques*, International Thomson Press, 1990.
4. Chillarege, Ram, Inderpal S. Bhandari, Jarir K Chaar et al, Orthogonal Defect Classification; a Concept for In-process Measurements, *IEEE Transactions in Software Engineering*, November 1982, 943-956
5. Chilllarege, Ram, Chap 9, pp359-400, in [Lyu].
6. Conte, S.D., et al, *Software Engineering Metrics and Models*, The Benjamin / Cummings Publishing Company, Inc. , 1986
7. Fenton, Norman E. and Shari Lawrence Pfleeger, *Software Metrics,* International Thomson Computer Press, 1996
8. Fenton, Norman E., *Software Metrics,* International Thomson Computer Press, 1995
9. Freidman, Michael A., and Jeffrey M. Voas, *Software Assessment*, John Wiley & Sons, Inc., 1995
10. Grady, Robert B, *Practical Software Metrics for project Management and Process Management,* Prentice Hall, Englewood Cliffs, NJ 07632, 1992
11. Hetzel, William, *The Complete Guide to Software Testing,* QED Information Sciences, Inc., 1984, pp177-197
12. Horch, John W., *Practical Guide to Software Quality Management*, Artech-House Publishers, 1996

13. 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 Technology, Jan 1995. [Ipp] http:// hissa.nist.gov/HAZARD/
14. Kan, Stephen, H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley Publishing Co., 1994]
15. 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.
16. 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.
17. Laprie, J.C., *Dependability: Basic Concepts and Terminology , IFIP WG 10.4*, Springer-Verlag, New York 1991
18. Leveson, Nancy, *SAFEWARE: system safety and requirements,*Addison-Wesley, 1995
19. Lewis, Robert O., *Independent Verification and Validation*, John Wiley & Sons, Inc., 1992 [Lew]
20. Lyu, Michael, *Handbook of Reliability Engineering*
21. McConnell, Seven C., *Code Complete: a practical handbook of software construction*, Microsoft Press, 1993
22. Musa, John D., and A. Frank Ackerman, "Quantifying Software Validation: When to stop testing?" *IEEE Software*, May 1989, 31-38.
23. Musa, John, *Software Reliability Engineering*, McGraw Hill, 1998
24. Palmer, James D., "Traceability," In: [Dorf], PP. 266-276
25. 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/
26. Ratikin, Steven R., *Software Verification and Validation, A Practitioner's Guide*, Artech House, Inc., 1997
27. Rubin, Jeffrey, *Handbook of Usability Testing*, JohnWiley & Sons, 1994
28. Salamon, W.J., D. R. Wallace, "Quality Characteristics and Metrics for Reusable Software (Preliminary Report)," NISTIR 5459, May 1994. http://hissa.nist.gov/REUSEMET/
29. Schulmeyer, Gordon C., and James I. McManus, *Handbook of Software Quality Assurance*, Third Edition, Prentice Hall, NJ, 1998.
30. Schulmeyer, Gordon C., and James L. McManus, *Handbook of Software Quality Assurance*, Van Nostrand Reinhold Company, Inc., 1987SEE NEW BOOK 1998
31. Voas, Jeffrey , "Certifying Software For High Assurance Environments, " *IEEE Software*, July-August, 1999.
32. Wakid, Shukri, D. Richard Kuhn, and Dolores R. Wallace, "Software Measurement: Testing and Certification," *IEEE Software*, July-August 1999
33. Wallace, Dolores R., and Roger U. Fujii, "Software Verification and Validation: An Overview," *IEEE Software*, May 1989, 10-17
34. Wallace, Dolores R., Laura Ippolito, and Barbara Cuthill, AReference Information for the Software Verification and Validation Process,@ NIST SP 500-234, NIST, Gaithersburg, MD 20899, April, 1996. http://hissa.nist.gov/VV234/
35. Zelkowitz, Marvin V. and Dolores R. Wallace, Experimental Models for Validating Computer Technology, *Computer*, Vol. 31 No.5, 1998 pp.23-31.

C. Standards

1. FIPS 140-1, Jan. 1994, Security Requirements for Cryptographic Modules, Jan. 1994
2. IEC 61508 Functional Safety - Safety -related Systems Parts 1,2
3. IEEE 610.12-1990, Standard Glossary of Software Engineering Terminology
4. IEEE 730-1998 Software Quality Assurance Plans
5. IEEE 829 -1998 Software Test Documentation
6. IEEE Std 982.1 and 982.2 Standard Dictionary of Measures to Produce Reliable Software
7. IEEE 1008-1987 Software Unit Test
8. IEEE 1012-1998 Software Verification and Validation
9. IEEE 1028 -1997 Software Reviews
10. IEEE 1044 -1993 Standard Classification for Software Anomalies
11. IEEE Std 1061-1992 Standard for A Software Quality Metrics Methodology

12. IEEE Std 1228-1994 Software Safety Plans
13. ISO 9126 Software Product Evaluation - Quality Characteristics and Guidelines for Their Use, 1991
14. ISO 12207 Software Life Cycle Processes 1995
15. ISO/IEC 15026:1998, Information technology -- System and software integrity levels.
16. The Common Criteria for Information Technology Security Evaluation (CC) VERSION 2.0 / ISO FDIS 15408

V. Appendices

A. Bloom's Taxonomy Related to Software Quality Analysis

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 choose the roles of programmer or designer who 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 will turn over the products to others who will conduct integration and higher levels of testing. This person may be asked to submit data on development tasks, but will not conduct analyses on faults or on measurements. The Software Quality Assurance or Verification and Validation specialist will plan and implement the processes for these two areas. The project manager of the development project will use the information from the software quality analysis processes to make decisions.

We use the following abbreviations for Bloom's taxonomy:

Knowledge:	KN	Analysis :	AN
Comprehension:	CO	Synthesis	SY
Application:	AP	Evaluation	EV

Topic	**Programmer**	**SQA/VV**	**Project Manager**
Definition of Quality	SY	SY	AN
Definition of SQA, VV process	AP	SY	AN
Plans	AP	SY	AN
Activities and Techniques			
Static- people intensive	EV	EV	AN
Static- Analysis	EV	EV	AN
Dynamic	SY, EV*	EV	AN
Measurement			
Fundamentals	AP	EV	AN
Metrics	AP	EV	AN
Techniques	AP	EV	AN
Defect Characterization	AP	EV	AN
Additional concerns	AP	EV	AN

* The knowledge level depends on the responsibility of the programmer regarding testing.

B. Vincinti's Categories Related to Software Quality Analysis

Fundamental Design Concept	Knowledge Area or Source
Concepts Related to Design Project:	
Operating principles of hardware components	Computer Science/ECE (see Appendix C)
Operating principles of operating systems	Computer Science (see Appendix C)
Operating principles of programming languages, data structures, algorithms, heuristic methods	Computer Science (see Appendix C)
Relevant data communication methods & protocols	Computer Science/ECE (see Appendix C)
Necessary principles of human-computer interaction	Computer Science/CogSci (see Appendix C)
SE Project Quality Management	Software Engineering Management Knowledge Area
Specifically Related to Software Quality Analysis:	
Quality plan requires identification of relevant attributes and standards for project	QA §I: Defining Quality Project; Requirements Engineering Knowledge Area
Prevention of problems is primary to building quality product	Software Process, Software Infrastructure, Configuration Control Knowledge Areas
Software quality assurance process	QA §II.A.1: Software Quality Assurance
Customer satisfaction is primary goal	QA §II.A.2: Software Verification & Validation

Criteria and Specifications	Knowledge Area or Source
Product quality specifications	QA §I: Defining Quality Product; Software Requirements Analysis Knowledge Area
Quality process plans	QA §II.B: Process Plans
Specifications of relevant measurements and tests	QA §II.D: Measurement in Software Quality Analysis; Software Testing Knowledge Area

Theoretical Tools	Knowledge Area or Source
Formal/Mathematical V&V	QA §II.D.1.a: Measurement in Software Quality Analysis; Logic and Discrete Mathematics; Theory of Computing
Analysis of Algorithms	Computer Science (see Appendix C)
Necessary Human Factors & Interaction	Computer Science (see Appendix C)
Design of Quality Metrics	QA §II.D: Measurement in Software Quality Analysis

Quantitative Data	Knowledge Area or Source
Error, Fault, Failure Data	QA §II.D.4: Defect Characterization
Comparative Metrics	QA §II.D: Measurement in Software Quality Analysis
Human Performance Data	Computer Science/CogSci*
Benchmark Data/Testing Data	QA §II.D.1-3: Fundamentals of Measurement, Metrics, and Techniques

Practical Considerations	Knowledge Areas
Dependability Needs	QA §I.B: Dependability
Special Situation Measures	QA §I.C Special Situations; Computer Science (see Appendix C)
Special Human Needs	QA §I.3 Usability; Computer Science/CogSci (see Appendix C)
Saving Future Costs	QA §I: Defining Quality Product (Design of reusable code); Software Evolution and Maintenance Knowledge Area
Quality Criteria Tradeoffs	QA §I: Defining Quality Product

Design Instrumentalities	Knowledge Areas
Object Oriented Methods	Software Process Knowledge Area
Structured Development Methods	Software Process, Software Infrastructure Knowledge Areas
CASE Tools and Methods	Software Infrastructure Knowledge Area
Code Reuse	QA §I: Defining Quality Product: Software Infrastructure Knowledge Area (Reuse Libraries)
Clean room	QA §II.A.1: Software Quality Assurance
Independent V&V Teams	QA §II.A.2: Software Verification & Validation
Testing Procedures	QA §II.A.2: Software Verification & Validation; Software Testing Knowledge Area
Misc. V&V Techniques	QA §II.A.2: Software Verification & Validation

C. Knowledge from Related Disciplines

Knowledge Item	Related Discipline/Subject Area	Knowledge Level (Bloom*)
Operating principles of hardware components	Computer Science: AR or ECE: Digital Systems	***CO***
Operating principles of operating systems	Computer Science: OS	CO/AP
Operating principles of programming languages, data structures, algorithms, heuristic methods	Computer Science: PF, AL, IM,	AP
Relevant data communication methods & protocols	Computer Science: AR, NC or ECE: Data Communication	CO
Necessary principles of human-computer interaction	Computer Science: CI or Cognitive Science: HCI	AN
Analysis of Algorithms	Computer Science: FO, AL	CO
Necessary Human Factors & Interaction	Computer Science: CI	CO
Human Performance Data	Computer Science/CogSci*	AP
Special Situation Measures	Computer Science: IS, IM, CN, NC, etc. (as appropriate to situation)	AP
Special Human Needs	Comp. Science: CI; Cognitive Science (as appropriate to need)	CO
Metrics Design and Use	Math/Stats: Applied Statistics	AN/SY
* Abbreviations of Bloom Levels are those used in Appendix A		

SWEBOK: Software Requirements Engineering Knowledge Area Description

Version 0.5

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

Table of contents

1. Introduction

This document proposes a breakdown of the SWEBOK Software Requirements Engineering Knowledge Area. 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 SRE.

Requirements engineering is of great economic importance to the software industry. The cost of fixing errors in the system requirements tends to increase exponentially the longer they remain undetected. Projects waste enormous resources rewriting code that enshrines mistaken assumptions about customers' needs and environmental constraints. There is therefore considerable risk involved in requirements engineering yet the maturity of most software development organizations' requirements engineering processes lags well behind that of their down-stream life-cycle processes. This is compounded by the fact that requirements engineering is tightly time- and resource-bounded and time spent analysing requirements is often wrongly perceived to be unproductive time. To help move forward from this situation, this knowledge area description sets out a proposal for the fundamental knowledge needed by software engineers to perform effective requirements engineering.

The issue of who should perform requirements engineering is, however, controversial. Requirements engineering requires a set of skills that intersect with those of software engineering but which also includes skills outside of this set. It is also important Requirements engineering is not always a 'software' only activity. For certain classes of system (e.g. embedded systems), it may need expert input from disciplines. For this reason, we refer in this document to the 'requirements engineer' rather than the 'software engineer'. This is not to suggest that software engineers cannot 'do' requirements engineering, but rather to emphasize that the role of the requirements engineer is a distinct one that mediates between the domain of software development and the domain of the software's customer(s) (application domain). Requirements engineering may not be the preserve of software engineers but it should be performed by people with software engineering knowledge.

2. References used

All the references from the jump-start documents are used here, namely [Dor97, Pfl98, Pres97, Som96, Tha97]. [Dor97, Pfl98, Pres97, Som96] place requirements engineering in the context of software engineering. This is a useful first step in understanding requirements engineering. [Tha97] is an extensive collection of papers tackling many issues in requirements engineering, and most of the topics that make up the knowledge area (section .

This initial list of references has been supplemented with additional requirements engineering specific and other relevant references[1]. Key additional references are [Dav93, Kot98, Lou95, Som97, Tha90, Tha93]. [Dav93] contains extensive discussion on requirements structuring, models and analysis. [Kot98] is an extensive text on requirements engineering covering most of the knowledge area topics. [Som97] is a useful practitioner’s guide to good requirements engineering. [Tha90] is an extensive and useful collection of standards and guidelines for requirements engineering.

3. System requirements

At its most basic, a computer-based system requirement can be understood as a property that the system must exhibit in order for it to adequately perform its function. This function 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, controlling 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 and, by extension, 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.

[1] [Ama97, And96, Che90,Gog93, Hal96, Har93, Hum88, Hum89, Pau96, Sid96, Rud94]

Clearly, requirements will vary in intent and in the kinds of properties they represent. A crude distinction is often drawn between:

- Capabilities that the system must provide, such as saving a file or modulating a signal. Capabilities are sometimes known as functional requirements.
- Constraints that the system must operate under, such as having a probability of generating a fatal error during any hour of operation of less than $1 * 10^{-8}$. Constraints are sometimes known as non-functional or quality requirements.
- A constraint on the development of the system (e.g. 'the system must be developed using Ada').
- A specific constraint on the system (e.g. 'the sensor must be polled 10 times per second').

In practice, this distinction is hard to apply rigidly and it is difficult to formulate a consistent definition of the distinction between them. Constraints are particularly hard to define and tend to vary from vaguely expressed goals to very specific bounds on the software's performance. Examples of these might be: that the software must increase the call-center's throughput by 20%; and that the task will return a result within 10ms. In the former case, the requirement is at a very high level and will need to be elaborated into a number of specific capabilities which, if all satisfied by the software, should achieve the required improvement in throughput. In the latter case, the requirement will constrain the manner in which the software engineers implement the functionality needed to compute the result.

This also illustrates the range of levels at which requirements are formulated. The standard, though rather unsatisfactory, way of distinguishing levels of requirements is as 'user' or 'system' requirements. User requirements are the direct requirements of the people who use or otherwise have a stake in (the 'stakeholders') the system. They consider the system to be a 'black box' and are only concerned with what is externally exhibited by the system. Their collection, organisation and analysis is crucial to gaining a clear understanding of the problem for which the computer system is to provide a solution and its likely cost. This understanding has to be validated and trade-offs negotiated before further resources are committed to the project. The user requirements therefore need to be expressed in terms that allow the stakeholders to do this validation. This usually means the requirements are expressed using a structured form of natural language to describe the system's effect on the stakeholders' business domain. Exceptionally, this may be supplemented by models using specialized software and systems engineering notations, but used carefully so that they can be understood with minimal explanation.

Software engineers who are experts in the stakeholders' domain are very rare so they cannot be expected to take a list of user requirements, interpret their meaning and translate them into a configuration of software and other system components that satisfy the user requirements. In most cases, therefore, user requirements are elaborated by the requirements engineer (together with *expert* input on the application domain), into a number of more detailed requirements that more precisely describe what the system must do. This usually entails deploying engineering skills to construct models of the system in order to understanding the logical partitioning of the system, its context in the operational environment and the data and control communications between the logical entities. A side-effect of this is that new requirements (emergent properties) will emerge as a conceptual architecture of the system starts to take shape. It is also likely to reveal problems with the user requirements (such as missing information) which have to be resolved in partnership with the stakeholders.

The result of this process is a set of detailed and more rigorously defined requirements. These are the system requirements (software and system requirements). Because the intended readership is technical, the system requirements can make use of modeling notations to supplement their textual descriptions. The requirements enable a system architect to propose a solution architecture. They form the baseline for the real development work so they should enable detailed planning and costs to be derived.

The above description implies a deterministic process where user requirements are elicited from the stakeholders, elaborated into system requirements and passed over to the development team. This is an idealized view. In practice, many things conspire to make requirements engineering one of the riskiest and most poorly understood parts of the software life cycle. The requirements engineer may interpret the stakeholders' requirements wrongly, the stakeholders may have difficulty articulating their requirements, or technology, market conditions or the customer's business processes may change mid-way through development.

All of these things militate against the ideal of having a requirements baseline frozen and in place before development begins. Requirements *will* change and this change must be managed by continuing to 'do' requirements engineering throughout the life-cycle of the project. In a typical project the activities of the requirements engineer evolve over time from elicitation and modeling to impact analysis and trade-off negotiation. To do this effectively requires that the requirements are carefully documented and managed so that the impact of proposed changes can be traced back to the affected user requirements and forwards to the affected implementation components.

What we have described above refers to requirements on the system (the 'product'). Requirements can also be imposed on how the system is developed (the 'process'). Process requirements are also important because they can impact on development schedules, development costs and the skills required of the developers. Thus, it is important to (for example) understand the implications of working for a customer who requires that the software contractor has achieved accreditation to CMM level 3. An important process requirement that every project has is a cost constraint.

4. The context of software requirements engineering

For many types of system it is impossible to separate the requirements for the software from the broader requirements for the system as a whole. As well as software, the system may include computer hardware and other types of hardware device which are interfaced to the computer and operational processes which are used when the system is installed in some working environment.

Computer-based systems fall into two broad types:

1. User-configured systems where a purchaser puts together a system from existing software products. The vast majority of personal computer systems are of this type. ‘Systems engineering’ is the responsibility of the buyer of the software, which is installed on a general-purpose computer. The software requirements for this type of system are created by the companies which develop the different software products from customer requests and from their perception of what is marketable.

2. Custom or bespoke systems where a customer produces a set of requirements for a hardware/software system and a contractor develops and delivers that system. The customer and the contractor may be different divisions within the same organization or may be separate companies. The system requirements describe services to be provided by the system as a whole and, as part of the systems engineering process, the specific requirements for the software in the system are derived. Custom systems vary in size from very small embedded systems in consumer devices such as microwaves ovens, to gigantic command and control systems such as military messaging systems.

   There are three important classes of custom systems:

   - Information systems
     These are systems that are primarily concerned with processing information that is held in some kind of database. They are usually implemented using standard computer hardware (e.g. mainframe computers, workstations, PCs) and are built on top of commercial operating systems. For these systems, requirements engineering is primarily software requirements engineering.
   - Embedded systems
     These are systems where software is used as a controller in some broader hardware system, They range from simple systems (e.g. in a CD player) to very complex control systems (e.g. in a chemical plant). They often rely on special purpose hardware and operating systems. Requirements engineering for these systems involves both hardware and software requirements engineering.
   - Command and control systems
     These are, essentially, a combination of information systems and embedded systems where special-purpose computers provide information, which is collected and stored in a database and then used to help people make decision. These systems usually involve a mix of different types of

computer, which are networked in some way. Within the whole system, there may be several embedded systems and information systems. Air traffic control systems, railway signaling systems, military communication systems are examples of command and control systems. Requirements engineering for these systems includes hardware and software specification and a specification of the operational procedures and processes.

The complete requirements engineering process spans several activities from the initial statement of requirements through to the more detailed development specific software requirements. The process takes place after a decision has been made to acquire a system. Before that, there are activities concerned with establishing the business need for the system, assessing the feasibility of developing the system, and setting the budget for the system development.

The requirements engineering process often includes some initial design activity where the overall structure (architecture) of the system is defined. This activity has to be carried out at this stage to help structure the system requirements and to allow the requirements for different sub-systems to be developed in parallel.

The requirements engineering processes will vary across organizations and projects. Important factors that are likely to influence this variation include:

- The nature of the project. A market-driven project that is developing a software product for general sale imposes different demands on the requirements engineering process than does a customer-driven project that is developing bespoke software for a single customer. For example, a strategy of incremental releases is often used in market-driven projects while this is often unacceptable for bespoke software. Incremental release imposes a need for rigorous prioritisation and resource estimation for the requirements to select the best subset of requirements for each release. Similarly, requirements elicitation is usually easier where there is a single customer than where there are either only potential customers (for a new product) or thousands of existing but heterogeneous existing customers (for a new release of an existing product).
- The nature of the application. Software requirements engineering means different things to (for example) information systems and embedded software. In the former case, a software development organisation is usually the lead or the only contractor. The scale and complexity of information systems projects can cause enormous difficulties, but at least the software developer is involved from initial concept right through to hand-over. For embedded software the software developer may be a subcontractor responsible for a single subsystem and have no direct involvement with eliciting the user requirements, partitioning the requirements to subsystems or defining acceptance tests. The requirements will simply be allocated to the software subsystem by a process that is opaque to the software contractor. A complex customer / main contractor / subcontractor project hierarchy is just one of many factors that can greatly complicate resolution of any problems that emerge as the software contractor analyses the allocated requirements.

Because of the artificiality in distinguishing between system and software requirements, we henceforth omit the word ‘software’ and talk about ‘requirements engineering’ except where it is helpful to make a distinction. This is also reflected by the coverage of requirements engineering by standards. Requirements documentation is the only aspect of requirements engineering that is covered by dedicated standards. Wider requirements engineering issues tend to be covered only as activities of software engineering or systems engineering. Current process improvement and quality standards offer only limited coverage of requirements engineering issues despite its place at the root of software quality problems.

5. Software requirements engineering breakdown: processes and activities

There are broadly two common ways of decomposing requirements engineering knowledge. The first is exemplified by the European Space Agency's software engineering standard PSS-05 [Maz96]. Here, the knowledge area is decomposed at the first level into the 2 topics of user requirements definition and software requirements specification. This is broadly what we described in section 2 and is a useful starting point (which is why we described it). However, as a basis for the knowledge area breakdown it confuses process activities and products with knowledge and practice.

Process knowledge is crucial but we prefer to separate out process issues from knowledge. The knowledge area topics correspond approximately to process tasks but they tend to be enacted concurrently and iteratively rather than in series as is implied by a breakdown based on PSS-05's model. The breakdown we have chosen broadly corresponds to that in [Som97 and Kot98] and is orthogonal to PSS-05. It comprises 5 topics:

- The requirements engineering process.
- Requirements elicitation.
- Requirements analysis.
- Requirements validation.
- Requirements management.

Figure 1 shows the coarse-grain requirements engineering process model.

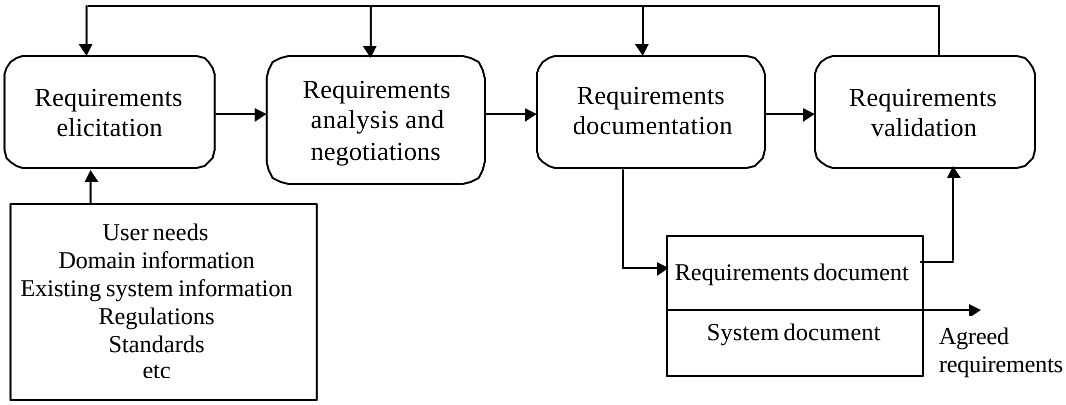


Figure 1 Coarse grain requirements engineering process

5.1 The requirements engineering process

This section is concerned with introducing the requirements engineering process and to orient the remaining 4 topics and show how requirements engineering dovetails with the overall software life-cycle. This section also deals with contractual and project organisation issues. The topic is broken down into 4 subtopics.

5.1.1 Process models.

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:

- 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;
- is connected to the overall software process by the life-cycle of requirements and the involvement of the requirements engineer;
- will need to be tailored to the organisation and project context.

In particular, the subtopic provides the context that allows the software engineer to understand how requirements elicitation, analysis, validation and management fit together.

5.1.2 Process actors.

This subtopic introduces the roles of the people who enact the requirements engineering process. The key people are the system stakeholders (users and other people who have a stake in the software) and the requirements engineer. The software engineering roles of (for example) system architect, tester and quality assurance are also described with reference to requirements engineering.

5.1.3 Process support.

This subtopic introduces the resources required and consumed by the requirements engineering process. This includes human resources, training requirements, methods and tools.

5.1.4 Process improvement.

This subtopic is concerned with quality. 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.

Links to common themes	
Quality	Software process improvement (SPI) recognizes the link between process and software quality. The process improvement subtopic is the one most concerned with quality here. This 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) [Som 97].
Standards	SPI standards as above. In addition, life-cycle software engineering standards ISO/IEC 12207 and ESA PSS-05 apply, in part, to the software requirements engineering process.
Measurement	Measurement is poorly developed for software requirements engineering processes. Where measurement is applied at all it is at a coarse level of granularity: e.g. total numbers of requirements, numbers of requirements changes.
Tools	We are aware of no application of tools to manage the requirements engineering process although there may be a role for workflow.

5.2 Requirements elicitation

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.

5.2.1 Requirements sources

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:

- 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.
- 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 necessary between conflicting requirements and sometimes to act as a 'user' champion.
- System stakeholders. 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. This subtopic is designed to alert the requirements engineer to the need to identify, represent and manage the 'viewpoints' of many different types of stakeholder.
- 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.

- 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.

5.2.2 Elicitation techniques

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:

- 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.
- 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.
- 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 5.3.2. (conceptual modeling) because recent modeling notations have attempted to integrate scenario notations with object-oriented analysis techniques.
- 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 are a 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 (5.4.2).

Links to common themes	
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.
Standards	Only very general guidance is available for elicitation from current standards.
Measurement	N/A
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.

5.3 Requirements analysis

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

- detect and resolve conflicts between requirements;
- discover the bounds of the system and how it must interact with its environment;

- elaborate user requirements to software requirements.

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).

5.3.1 Requirements classification

Requirements can be classified on a number of dimensions. Common useful classifications include:

- As capabilities or constraints. Distinguishing between capabilities and constraints can be hard but is worth-while, particularly where the software's 'qualities' (in safety-related system, for example) are especially important. It is common to sub-classify constraints as, for example, performance, security, reliability, maintainability, etc. requirements. Many such requirements imply the need for special metrics to specify and validate the requirements and have implications for the resources that need to be allocated to testing. It is useful to perform this classification early in the requirements engineering process to help understand the full implication of user requirements.
- According to priority. In general, it will be impossible to implement every user requirement because of mutual incompatibilities between requirements and because of insufficient resources. These conflicting demands have to be traded-off and one of the most important items of information needed to do this is the requirements' priorities. In practice, it can be hard to do this and for many organizations a 3-level classification such as mandatory/highly desirable/desirable provides a crude but workable scheme.
- According to the requirements' cost/impact. This classification complements that of priority since if the cost of implementing a mandatory requirement is unaffordable, this will reveal a serious problem with the stakeholder's expectations. The impact of changing a requirement is also important since it strongly affects the cost of doing so.
- According to their scope. The scope of a requirement can also affect its priority and cost. Requirements with global scope (for example goals - see 5.2.1) tend to be costly and of high priority; responsibility for their satisfaction cannot be allocated to a single component of the architecture.
- According to 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.
- According to whether they are product or process requirements.

Other classifications may be appropriate, depending upon the development organization's normal practice and the application itself. There is a strong overlap between requirements classification and requirements attributes (5.5.1).

5.3.2 Conceptual modeling

The development of models of the problem is fundamental to requirements analysis. The purpose is to aid understanding of the problem rather than to initiate design of the solution. However, the boundary between requirements engineering and design is often a blurred one in practice and the requirements engineer may find themselves unavoidably having to model aspects of the solution. 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:

- 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.
- The expertise of the requirements engineer. It is generally better to adopt a modeling notation or method that the requirements engineer has experience with.

- 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.
- 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.

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.

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.

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.

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.

5.3.3 Requirements negotiation

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 unwise 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	
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.
Standards	Software engineering standards stress the need for analysis. More detailed guidance is provided only by de-facto modeling 'standards' such as UML.
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.
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.

5.4 Requirements validation

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.

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.

5.4.1 The conduct of requirements reviews.

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.

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

5.4.2 Prototyping.

Prototyping is commonly employed for validating the requirements engineer's interpretation of the user 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. They may also 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.

5.4.3 Model validation.

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).

5.4.4 Acceptance tests.

An essential property of a user 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.

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

5.5 Requirements management

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.

There are 4 main topics concerned with requirements management.

5.5.1 Change management

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.

5.5.2 Requirements attributes

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 classifications attached to the requirement (see 4.3.1) 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.

5.5.3 Requirements tracing

Requirements tracing is concerned with recovering the source of requirements and predicting the effects of requirements. Tracing is fundamental to performing impact analysis when requirements change. A requirement should be traceable backwards to the requirements and stakeholders that motivated it (from a software requirement back to the user 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 user requirement into the software requirements that have been elaborated from it and on into the code modules that implement it).

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.

5.5.4 Requirements documentation

This subtopic is concerned with the role and readership of requirements documentation, with documentation standards and with the establishment of good practice.

Requirements documents are the normal medium for recording and communicating requirements. It is good practice to record the user requirements and the software requirements in different documents. The user requirements definition document organizes the user requirements and makes them available to the stakeholders for validation. Once validated, the user requirements can be elaborated and recorded in the software requirements specification document (SRS). Once this has been validated, it serves as the basis for subsequent development.

Requirements documents need to be subjected to version control as requirements change. Modern requirements management tools allow requirements to be documented in electronic form along with their associated attributes and traceability links. Such tools usually include document generators that allow requirements documents to be generated in defined formats so, for example, the software requirements can be used to (semi-) automatically generate an SRS that conforms to IEEE std 830.

Links to common themes	
Quality	Requirements management is a level 2 key practice area in the software CMM and this has boosted recognition of its importance for quality.
Standards	Software engineering life-cycle and documentation standards (e.g.

	IEEE std 830) exist and are widely used in some domains.
Measurement	Mature organizations may measure the number of requirements changes and use quantitative measures of impact assessment.
Tools	There are a number of requirements management tools on the market such as DOORS and RTM.

Appendix A - Summary of requirements engineering breakdown

Requirements engineering topics	**Subtopics**
The requirement engineering process	Process models Process actors Process support Process improvement
Requirements elicitation	Requirements sources Elicitation techniques
Requirement analysis	Requirements classification Conceptual modelling Requirements negotiation
Requirements validation	Requirements reviews Prototyping Model validation Acceptance tests
Requirements management	Change management Requirements attributes Requirements tracing Requirements documentation

Appendix B Matrix of topics and references

In Table 1 shows the topic/reference matrix. The table is organized according to requirements engineering topics in Appendix A. 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	[Dav93]	[Dor97]	[Kot98]	[Lou95]	[Maz96]	[Pfl98]	[Pre97]	[Som96]	[Som97]	[Tha90]	[Tha97]
Requirements engineering process	X	X	X	X				X	X	X	X
Requirement process models			X	X				X	X	X	X
Requirement process actors	X		X	X					X	X	X
Requirement process support									X	X	X
Requirement process improvement			X						X		
Requirements elicitation	X	X	X	X		X			X	X	X
Requirements sources	X		X	X		X	X		X	X	X
Elicitation techniques	X		X	X		X				X	X
Requirements analysis	X	X	X	X			X	X	X	X	X
Requirements classification	X	X	X	X				X	X		X
Conceptual modeling	X		X	X			X	X		X	X
Requirements negotiation			X						X		
Requirements validation	X	X		X				X	X	X	X
Requirements reviews			X	X					X		X
Prototyping	X		X	X					X	X	X
Model validation			X						X		X
Acceptance tests	X								X	X	
Requirements management	X	X	X	X				X	X	X	X
Change management			X						X		X
Requirement attributes			X						X		X
Requirements tracing			X						X	X	X
Requirements documentation	X		X		X		X	X	X		X

Table 1 Topics and their references

Appendix C - Software requirements engineering and Bloom's taxonomy

Appendix D – Software requirements engineering references

[Ama97] K, EL Amam, J. Drouin, et al. *SPICE: The theory and Practice of Software Process Improvement and Capability Determination*. IEEE Computer Society Press, 1997.

[And96] S.J. Andriole, *Managing Systems Requirements: Methods, Tools and Cases*. McGraw-Hill, 1996.

[Che90] P. Checkland and J. Scholes, *Soft Sysems Methodology in Action*. John Wiley and Sons, 1990.

[Dav93] A.M. Davis, *Software Requirements: Objects, Functions and States*. Prentice-Hall, 1993.

[Dor97] M. Dorfman and R.H. Thayer, *Software Engineering*. IEEE Computer Society Press, 1997.

[Gog93] J.A. Goguen and C. Linde, Techniques for requirements elicitation, Proc. Intl' Symp. On Requirements Engineering. IEEE Press, 1993.

[Hal96] A. Hall, Using Formal Methods to Develop an ATC Information System. IEEE Software 13(2), pp66-76, 1996.

[Har93] R. Harwell. et al, What is a Requirement. Proc 3rd Ann. Int'l Symp. Nat'l Council Systems Eng., pp17-24, 1993

[Hum89] W. Humphery, *Managing the Software Process*. Addison Wesley, 1989.

[Hum88] W. Humphery, Characterizing the Software Process, IEEE Software 5(2): 73-79, 1988.

[Pfl98] S.L. Pfleeger, *Software Engineering-Theory and Practice*. Prentice-Hall, 1998.

[Pres97] R.S. Pressman, *Software Engineering: A Practitioner's Approach (4 edition)*. McGraw-Hill, 1997.

[Tha90] R.H. Thayer and M. Dorfman, *Standards, Guidelines and Examples on System and Software Requirements Engineering*. IEEE Computer Society, 1990.

[Tha97] R.H. Thayer and M. Dorfman, *Software Requirements Engineering (2nd Ed)*. IEEE Computer Society Press, 1997.

[Kot98] G. Kotonya, and I. Sommerville, *Requirements Engineering: Processes and Techniques*. John Wiley and Sons, 1998.

[Lou95] P. Loucopolos and V. Karakostas, *System Requirements Engineering*. McGraw-Hill, 1995.

[Maz96] C. Mazza, J. Fairclough, B. Melton, et al, *Software Engineering Guides*. Prentice-Hall, 1996.

[Pau96] M.C. Paulk, C.V. Weber, et al., *Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, 1995.

[Sid96] J. Siddiqi and M.C. Shekaran, Requirements Engineering: The Emerging Wisdom, IEEE Software, pp15-19, 1996.

[Som96] I. Sommerville, *Software Engineering (5th edition)*. Addison-Wesley, 1996.

[Som97] I. Sommerville and P. Sawyer, *Requirements engineering: A Good Practice Guide*. John Wiley and Sons, 1997

[Rud94] J. Rudd and S. Isense, Twenty-two Tips for a Happier, Healthier Prototype. ACM Interactions, 1(1), 1994.

KA Description of Software Testing V. 0.5

A. Bertolino
Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche
via S. Maria, 46
56126 Pisa, Italy
+39 050 593478
bertolino@iei.pi.cnr.it

READ ME

This is Version 0.5 of the Software Testing KA Description. I have done my best to provide -within the close deadline- a preliminary document according to the specifications. However, I am heavily relying on the feedback from the first review round to improve this draft document, as I am fully aware that it still needs more work than that I could realistically put on it. Above all, some of the topics certainly need more or better reference material: I have left some reference fields annotated with "???", for topics I would like to include, but for which I have not yet been able to find adequate reference material. On the other side, the number of references is already higher than the suggested size of 15. In this regard, note that by referencing books or survey articles that report on them, I have omitted to include some famous papers in the testing literature, in order to reduce the size of references list. For these matters, I warmly ask reviewers to give suggestions for addition, replacement and reduction of references. Some other questions/comments are given within [[ ]].

Keywords
Test levels, Test techniques, Test related measures, Organizing and controlling the test process, Automated testing

RATIONALE OF BREAKDOWN

Software testing consists of the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behaviour.

In the above definition, and in the following as well, underlined words correspond to key issues in identifying the KA of software testing. In particular:

- dynamic: this term means that 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 KA; [[maybe we should here point at where in the BOK they are covered?]] nor is program execution on symbolic inputs, or symbolic evaluation;
- finite: the number of executions to be observed in testing must be obviously finite, and -more than this- it must be manageable. Indeed, testing always implies a *trade-off* between limited resources and inherently unlimited test requirements: this conflict points to well known problems of testing, of both technical nature (criteria for deciding test adequacy) and managerial nature (estimating the effort to put in testing);
- 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.
- expected: it must be possible to decide about whether the observed outcomes of program execution are acceptable or not, otherwise the testing effort would be useless. This is referred to as the *oracle problem*, which can be addressed with different approaches. For instance, the observed behaviour may be checked against user's expectances (commonly referred to as testing for *validation*) or against a specification of allowed behavior (testing for *verification*).

When the outcome of a test execution diverges from the expected, a failure is observed, whose cause is a fault.

These and other basic definitions within software testing area, as well as an introduction to the terminology, will be provided in Part A *Basic Concepts and Definitions*. In the same section, the scope of the KA will be laid down.

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.

The latter argument is expanded in Part B *Test Levels*. This consists of two (orthogonal) subsections: B.1 lists the phases 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 "types of testing". Clearly not all listed types of testing apply to every system, nor any possible type has been listed, but those most generally applied.

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.

Very often, testing is aimed at exposing 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 trying it over 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 behaviours (generally in the form of subclasses of the input domain). However, a comprehensive view of the KA of testing must include other as important objectives for testing, e.g., reliability measurement, usability evaluation, contractor acceptance, for which different approaches would be taken. Note that the test objective varies with the test object, i.e., *different purposes are addressed in different phases*.

The test objective is strictly related with the identification of the test set, both in its consistency -*how much testing is enough for achieving the stated objective?*- and in 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 italicised 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*.

As already 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 will be described in Part C.

Sometimes, it can happen that confusion is made between test objectives and techniques. For instance, branch coverage (see C.2) 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 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.

Test-related measures are dealt accordingly in Part D.

Testing concepts, strategies, techniques and metrics need to be integrated into a defined and controlled process, which is run by people. The test process should support testing activities and 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.

Issues relative to organizing and controlling the test process are expanded in Part E.

Software testing is a very expensive and labour-intensive component of development. For this reason, a key issue in testing has always been pushing automation as much as possible.

Existing tools and concepts related to automating the constituent parts of test process are addressed in Part F.

BREAKDOWN OF TOPICS WITH SUCCINT DESCRIPTIONS

This section gives a list of topics for Software Testing KA, according to the rationale explained in Sect.1. The topics are accompanied by succint description and references.

A. Basic Concepts and Definitions

A.1 Faults vs. Failures

[[perhaps all part A.1 should be moved to the SQA KA?]]

⇨ The Fault-Error-Failure chain (Ly:p12-13; Ly:c2s2.2.2; Jo:c1p3; Pe:c1p5-6; Pf:p278-279; ZH+:s3.5; 982.2:fig3.1.1-1; 982.2:fig6.1-1;)

When speaking of software malfunctioning, there is some inconsistency in the literature in the meaning given to relevant terms such as *fault* or *error.* However, all references agree that a malfunctioning perceived by the user of a software system, a "failure", is one of the two extremes of a cause-effect chain, that can be several steps long. It is important to distinguish between:

- the *cause* for malfunctioning: *fault*
- a system state corruption, not yet propagated outside: *error*
- an undesired effect on the delivered service: *failure*

Note that in this chain, not always the cause can be univocally identified. This is why some authors instead of faults prefer to speak in terms of *failure-causing inputs* [FH+]. Testing can reveal failures (with appropriate program instrumentation, also errors), but then to remove them it is the fault that must be fixed.

⇨ Types, classification and statistics of faults (1044, 1044.1, Pf:p280-284; Be:c2; Be:Appendix; Jo:c1s1.5; Ly:c9)

Most times, testing is aimed at finding (and removing) defects. Therefore, it can be useful to know which types of faults can occur 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 testing literature is rich of classifications and taxonomy of faults, including an IEEE standard on classification of software "anomalies" [1044]. An important property for fault classification is orthogonality, i.e., ensuring that each fault can be univocally identified as belonging to one class [Ly].

A.2 Theoretical foundations

- Definitions of testing and related terminology (Be:c1; Jo:c1; 610)

A comprehensive introduction to the topics dealt with within the KA of software testing is provided in the first chapter of the two referenced books. In the Standard Glossary, definitions of software testing terms can be found.

- Test selection criteria/Test adequacy criteria (or stopping rules) (Pf:c7s7.8; WW+; ZH+:s1)

A test criterion is a means of deciding which a suitable set of test cases should be; 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. A criterion can be used for selecting T, or for checking if a selected T is adequate, i.e., to decide if the testing can be stopped.

- Testing effectiveness/Objectives for testing (Be:c1s1.4; FH+; Pe:p61-65; Pe:p434-437; So:p447)

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 lenght in the references provided.

- Debug testing, or testing for defect removal (So:p464-466; Be:c1s.1.3;)

In testing for defect removal 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.

- The oracle problem (Be:c1s4.1-4.2-4.3; We;)

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.

- Theoretical and practical limitations of testing (Ho)

Testing theory warns against putting too 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 aforism that testing can only show the presence of bugs, but never their absence.

- Path sensitizing/infeasible paths (Be:c3s4;)

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. Finding a set of solutions to the compound predicate expression corresponding a control flow path is called path sensitization.

- Software testability (VM; BM)

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 criterion (e.g. [BM]); on the other hand, as the likelihood (possibly measured statistically) that the system exposes a failure under testing, *if* it is faulty [Vo]. Both meanings are important.

A.3 Laying down the KA

Here the relation between the KA of software testing and other related areas is briefly discussed. We see testing as distinct 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.

[[I am in need for reference suggestions here]]

- Testing vs. Static Analysis Techniques (Be:p.8; Pe:p359-360; 1008:p19)
- Testing vs. Correctness Proofs (Be:c1s5; Pf:p298; ???)
- Testing vs. Debugging (Be:c1s2.1; 1008:p19)
- Testing vs. Programming (Be:c1s2.3; ???)
- Testing within SQA (???)
- Testing within CMM (Po:p117-123)
- Testing within Cleanroom (Pf:p381-386)
- Testing and Certification (???)

B. Test Levels

B.1 Test phases (Jo:c12; Be:c1s3.7; Pf:p284-286)

Testing of large software systems usually involves several stages.

- Unit testing (Be:p.21; 1008)

Unit testing verifies the functioning in isolation of the smallest testable pieces of software (the individual subprograms).

- Module ("Component") testing (Be:p21; 1008)

A module or a component in this context is a set of tightly related units. Some authors do not even distinguish between unit and module testing.

[[Is it useful to distinguish between unit and module, or should I rather stay with one?]]

- Integration testing (Be:p21; Jo:c13; Pf:c7s7.4)

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.

⇨ System testing (Be:p.22; Jo:c14)

System testing is concerned with the behaviour of a whole system, and at this level the main goal is not to find faults (most of them should have been already found at finer levels of testing), but rather to demonstrate performance in general. The kind of properties one may want to verify at this level, including conformance, reliability, usability among others, are discussed below under part "Types of testing".

⇨ Acceptance/qualification testing (Pe:c10; Pe:p447; Pf:c8s8.5; 12207:s5.3.9)

Acceptance testing checks the system behavior against the customer's requirements (the "contract"), and is usually conducted by or with the customer.

⇨ Installation testing (Pe:c9; Pf:c8s8.4)

After completion of system and acceptance testing, the system is verified upon installation in the target environment.

⇨ Alpha and Beta Testing (???)

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 experienced problems.

B.2 Types of Testing (Pe:p61-65; Pe:p178-180, Pe:p196-212; Pf:p349-351)

Testing can be aimed at verifying different properties of a system or subsystem. Test cases can be designed to demonstrate 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 found in the literature.

⇨ Conformance testing/Functional testing/Correctness testing (Pe:p179, p207; ???)

Conformance testing is aimed at verifying whether the observed behaviour of the tested system conforms to its specification.

⇨ Reliability achievement and evaluation by testing (Pf:c8s.8.4; Ly:c6,c7; Musa and Ackermann in Po:p146-154; So:c18s18.3)

By testing we can identify and fix faults, thereby making the software more reliable. In this very general sense, testing is a means to achieve reliability. On the other hand, by randomly generating test cases accordingly to the operational profile, we can derive statistical measures of reliability. Using reliability growth models, we can pursue together both objectives (see also part D.1).

⇨ Regression testing: (Jo:c14s14.7.3; Pe:c11, c12, p369-370; Pf:p338-339; RH)

According to 610.12, 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 behaviour is unchanged except insofar as required.

[[All topics listed below are very sketchy in the references provided; on the other hand, I would think that just knowing their different purposes could be sufficient here. Open to suggestions]]

⇨ Performance testing (Pe:p180, p203, p363-364)

This is specifically aimed at verifying that the system meets the specified performance requirements.

⇨ Stress testing (Pe:p179, p202, p361-362; Pf:p349; So:c22s22.3.4)

Stress testing exercises a system beyond the maximum design load.

⇨ Volume testing (Pe:p185, p487; Pf:p349)

Volume testing is concerned with evaluating system behaviour when internal program or system limitations are exceeded.

⇨ Back-to-back testing (So:c22s22.3.5)

A same test set is presented to two implemented versions of a system, and the results are compared with each other.

⇨ Recovery testing (Pe:p179, p201, p364-365)

It is aimed at verifying system restart capabilities after a "disaster".

⇨ Configuration testing (Pf:p349)

In those cases in which a system is built to serve different users, configuration testing analyzes the system under the various specified configurations.

⇨ Usability testing (Pe:p179, p208)

It evaluates the ease of use of the system for the end users.

C. Test Techniques

C.1 Specification-based

⇨ Boundary-value analysis (Jo:c5; Pe:p376, p448;)

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.

⇨ Equivalence partitioning (Jo:c6; Pe:p376;

So:c23s23.1.1)

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.

⇨ Category-partition (OB)

This is a systematic, stepwise method to analyse informal functional specifications, and decompose them into test suites through a series of defined steps: first those *categories* of information are identified that characterize parameters and environment conditions affecting functional behaviour. Each category, then, is partitioned into classes of "similar" values, or *choices*.

⇨ Decision table (Be:c10s3; Jo:c7; Pe:p377)

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* (Pe:p449; Pf:p345-349).

⇨ Finite-state machine-based (Be:c11; Pe:p377; Jo:c4s4.3.2; ???)

By modelling 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 systems.

⇨ Testing from formal specifications (ZH+:s2.2)

Giving the specifications in a formal language (i.e., one with precisely defined sintax 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 or algebraic specifications are distinguished.

C.2 Code-based

⇨ Reference models for code-based testing (flowgraph, call graph) (Be:c3s2.2; Jo:c4p41-50).

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.

⇨ Control flow-based criteria (Be:c3; Pe:p378; ZH+:s2.1.1).

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.

⇨ Data flow-based criteria (Be:c5; Jo:c10; ZH+:s2.1.2)

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 definition clear path between d and u w.r.t V, i.e. definition d of V must reach use u.

C.3 Fault-based

With different degrees of formalisation, fault based testing techniques devise test cases specifically aimed at revealing categories of likely or pre-defined faults.

⇨ Error guessing (Pe:p370-371; ???)

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.

⇨ Fault seeding (Pe:p379; Pf:p318-320; ZH+:s3.1)

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.

⇨ Mutation (Pe:p380; ZH+:s3.2-s3.3)

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. An underlying assumption of mutation testing, the coupling effect, is that by looking for simple syntactic faults, also more complex faults will be found. Strong and weak mutation techniques have been developed.

C.4 Usage-based

⇨ Operational profile (Jo:c14s14.7.2; Ly:c5; Ly:p534-539)

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.

⇨ (Musa's) SRET (Ly:c6)

Software Reliability Engineered Testing 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".

C.5 Specialized techniques

The techniques above apply to all types of software. However, for some kinds of applications some additional knowhow is required. Here below a list of few "specialized" testing techniques is provided. Of course, the classification is not to be intended as mutually exlusive; for instance, one can be interested in "operational testing" of "object-oriented" "distributed" software systems.

⇨ Object-oriented testing (Jo:c15; Pf:c7s7.5; Bi;)

⇨ Component-based testing (???only have found workshop refs so far)

⇨ GUI testing (OA+; ???)

⇨ Testing of concurrent programs (CT; ???)

⇨ Protocol conformance testing (Sidhu and Leung in Po:p102-115; BP)

⇨ Testing of distributed systems (???)

⇨ Testing of hard-real-time systems (Sc).

C.6 Selecting and combining techniques (Pe:p359-360;p381-383)

⇨ Functional and structural (Be:c1s.2.2; Jo:c11s11.3; Pe:p358-359, p383; Po:p3-4; Po:Appendix 2)

Several authors have pointed out the complementarity of functional and structural approaches to test selection: in fact, they use different sources of information and highlight different kinds of problems. They should be used in combination, compatibly with budget availability.

⇨ Coverage and operational/Saturation effect (Ly:p541-547)

This topic discusses the differences and complementarity of deterministic and statistical approaches to test case selection.

D. Test related measures

Test related measures can be divided into two classes: those relative to evaluating the program under test, that count and predict either faults (e.g., fault density) or failures (e.g., reliability). And those relative to evaluating the test suite derived according to a selected criterion, as is usually done by measuring code coverage achieved by the executed tests. Measures relative to the test process for management purposes are instead considered in part E.

D.1 Evaluation of the program under test (982.2)

⇨ Cyclomatic complexity (Be:c7s4.2; Jo:c9s9.3.3; 982.2:sA16)

This is a very popular static measure of program complexity; it is reported here since it is often used to guide structural testing.

[[Put cyclomatic complexity away from testing? Leave and introduce reference to my own Beta metric-not as well known, but performing better?]]

⇨ Remaining number of defects/Fault density (Pe:p515; 982.2:sA1; Ly:c9)

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.

⇨ Life test, reliability evaluation (Musa and Ackermann in Po:p146-154)

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.

⇨ Reliability growth models (Ly:c3, c4)

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.

D.2 Evaluation of the tests performed

⇨ Coverage/thoroughness measures (Jo:c9s9.2; Pf:p300-301; Pe:p491; Pe:p457; 982.2:sA5-sA6)

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.

⇨ Comparison and relative effectiveness of different techniques (Jo:c8,c11; ZH+:s5; Weyuker in Po p64-72; FH+)

Several studies have been recently conducted to compare the relative effectiveness of different test techniques. It is important to precise against which property we are assessing the techniques, i.e., what we exactly mean for "effectiveness". Possible interpretations are how many tests are needed to find the first failure, or which technique discovers a higher number of failures, or how much a technique actually improves reliability. Analytical and empirical comparisons between different techniques have been conducted according to each of the above specified notions of "effectiveness".

E. Organizing and Controlling the Test Process

E.1 Management concerns

- Attitudes/Egoless programming (Be:c13s3.2; Pf:p286)

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.

- Test process (Be:c1s3.1-3.2; Jo:c12s12.1,12.2; Pe:c2-c3; Pe:p183; Pf:p332-342; Po:p10-11; Po:Appendix 1; 12207:s5.3.9;s5.4.2;s6.4;s6.5)

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 organised, together with people, tools, policies, measurements, into a well defined process, which is integral part to the life cycle. In the traditional view, test activities complete design and code activities forming a V model.

- Test documentation (829; Be:c13s5; Pe:c19; Pf:c8s8.8)

Documentation is an integral part of the formalization of the test process. Beizer's reference provocatively proposes to throw away software documentation, including the source code, after a product is released, because the object code is everything is needed to run it. Unfortunately, this "insane proposal" is still common practice for tests after they are run. More rationally, test documentation should be performed and continually updated, at the same standards as other types of documentation.

- Internal vs. independent test team (Be:c13s2.2-2.3; Pe:p58-61; Pf:p342-343)

Formalization of the test process requires formalising 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, indepedent perspective, or finally of both internal and external members. The decision will be determined by considerations of costs, schedule and application criticality.

- Cost/effort estimation and other process metrics (Pe:p55-58; Pe:p437-441; Pe:Appendix B; Pf:p375; Po:p139-145; 982.2:sA8-sA9)

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.

E.2 Test Activities

- Planning (Pe p65; Pf c7s7.6; 829:s4; 1008:s1, s2, s3;)
- Test case generation (Pe p484; Po c2; 1008:s4, s5;)
- Test environment development (???)
- Execution (1008:s6, s7;)
- Test results evaluation (Pe c20; Po p18-20; Po p131-138)
- Trouble reporting/Test log (829:s9-s10;)
- Defect Tracking (???)

F. Automated Testing

F.1 Testing tool support

- What can be automated (and what cannot) (Be:c13s6; Pe:p360-361; Po:p124-129; Po:p199-213)

This topic focuses on theoretical and practical issues about which activities of the testing process can and/or should be automated, also providing a description of available tool support.

- Tool selection criteria (Po:p214-223)

This topic provides guidance to managers and testers on how to select those tools that will be most useful to their organization.

Appendix/F.2 Tools

- Surveys of available tools (Pe:c18; Lyu Appendix A; Pf c7s7.7)

This appendix is intended to provide testers with an up-to-date repertoire of available tools.

[[Not sure of what kind of reference should I provide here for tools? web-sites?]]

Capture/Replay

Test harness (drivers, stubs) (Be:c13s6.3.3;

Test generators (Be:c13s6.4;

Oracle/file comparators/assertion checking (Pe:451;

Coverage analyzer/Instrumentor (Be:c13s6.2;

Tracers

Regression testing tools

Reliability evaluation tools (Lyu Appendix A)

REFERENCES

GENERAL BOOKS

Pf Pfleeger, S.L. *Software Engineering Theory and Practice*, Prentice Hall, 1998.

So Sommerville, I. *Software Engineering* Fifth Edition, Addison Wesley, 1996.

TESTING BOOKS

Be Beizer, B. *Software Testing Techniques* 2nd Edition. Van Nostrand Reinhold, 1990.

Jo Jorgensen, P.C., *Software Testing A Craftsman's Approach*, CRC Press, 1995.

Ly Lyu, M.R. (Ed.), *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996.

Pe Perry, W. *Effective Methods for Software Testing*, Wiley, 1995.

Po Poston, R.M. *Automating Specification-based Software Testing*, IEEE, 1996.

SURVEY PAPERS

Bi Binder, R.V. Testing Object-Oriented Software: a Survey. *STVR*, 6, 3/4 (Sept-Dec. 1996) 125-252.

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.

SPECIFIC PAPERS

[[I am uncertain here about the opportunity to include underlined references, as they are "specific" (i.e., perhapsthey do not fulfill the requirement of generally accepted knowledge)]]

BM Bache, R., and Müllerburg, M. Measures of Testability as a Basis for Quality Assurance. *Software Engineering Journal*, 5 (March 1990) 86-92.

BP Bochmann, G.V., and Petrenko, A. Protocol Testing: Review of Methods and Relevance for Software Testing. *ACM Proc. ISSTA 94* (Seattle, Washington, USA, August 1994) 109-124.

CT Carver, R.H., and Tai, K.C., Replay and testing for concurrent programs. *IEEE Software* (March 1991) 66-74

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.

Ho Howden, W.E., Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering,* 2, 3, (Sept. 1976) 208-215.

OA+ Ostrand, T., Anodide, A., Foster, H., and Goradia, T. A Visual Test Development Environment for GUI Systems. *ACM Proc. ISSTA 98* (Clearwater Beach, Florida, USA, March 1998) 82-92.

OBOstrand, 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.

RHRothermel, G., and Harrold, M.J., Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22, 8 (Aug. 1996) 529-

Sc Schütz, W. Fundamental Issues in Testing Distributed Real-Time Systems. *Real-Time Systems Journal*. 7, 2, (Sept. 1994) 129-157.

VM Voas, J.M., and Miller, K.W. Software Testability: The New Verification. *IEEE Software*, (May 1995) 17-28.

WeWeyuker, E.J. On Testing Non-testable Programs. *The Computer Journal*, 25, 4, (1982) 465-470

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.

STANDARDS

610 IEEE Std 610.12-1990, Standard Glossary of Software Engineering Terminology.

829 IEEE Std 829-1998, Standard for Software Test Documentation.

982.2 IEEE Std 982.2-1998, Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.

1008 IEEE Std 1008-1987 (R 1993), Standard for Software Unit Testing.

1044 IEEE Std 1044-1993, Standard Classification for Software Anomalies.

1044.1 IEEE Std 1044.1-1995, Guide to Classification for Software Anomalies.

12207 IEEE/EIA 12207.0-1996, Industry Implementation of Int. Std. ISO/IEC 12207:1995, Standard for Information Technology-Software Life cycle processes

Knowledge Area Description Specifications for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge

Version 0.25

Approved by the Industrial Advisory Board

Prepared by

Pierre Bourque

Robert Dupuis

Alain Abran

James W. Moore

Leonard Tripp

March 26, 1999

Introduction[1]

This document presents and interim version (version 0.2) of the specifications for 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. Version 0.1 of these specifications has been given to the Knowledge Area Specialists. They have used them to produce version 0.1 of the Knowledge Area Descriptions and have commented on the specifications. Their comments have been included in this version of the document.

This baseline set of specifications may of course be improved through feedback obtained from the two important review cycles of the Guide scheduled for this spring and next fall.

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.

Content Guidelines

The following guidelines are presented in a schematic form in the figure found on page 8. 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.

Criteria and requirements for proposing the breakdown(s) of topics within a Knowledge Area

The following requirements and criteria should be used when proposing a breakdown of topics within a given Knowledge Area:

[1] Text in bold is for the benefit of the Knowledge Area Specialists and indicates changes between version 0.1 of this document and version 0.2.

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.

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”.

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 entitled “What do we mean by “generally accepted knowledge”?” found below for a more detailed discussion on this.

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.

e) The proposed breakdown of topics must, as much as possible, be compatible with the various schools of thought within software engineering.

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.

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.

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), measurement, tools and standards.

   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.

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.

j) Proposed topic names must be significant enough to be meaningful even when cited outside the Guide to the Software Engineering Body of Knowledge.

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.

Criteria and requirements for describing topics and for describing the rationale underlying the proposed breakdown(s) within the Knowledge Area

l) Topics need only to be sufficiently described so the reader can select the appropriate reference material according to his/her needs.

m) Knowledge Area Specialists are expected to provide a text describing the rationale underlying the proposed breakdown(s).

Criteria and requirements for rating topics according to Bloom's taxonomy

n) 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.

   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. This Appendix will be reviewed in a separate process.

Criteria and Requirements for selecting Reference Material

o) Specific reference material must be identified for each topic. Each reference material can of course cover multiple topics.

p) 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.

q) Proposed Reference Material must be in English.

r) A maximum of 15 Reference Materials can be recommended for each Knowledge Area.

s) 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.

t) A matrix of reference material versus topics must be provided.

Criteria and Requirements for identifying Knowledge Areas of the Related Disciplines

u) 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.

   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.

   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.

What do we mean by “generally accepted knowledge”?

The software engineering body of knowledge is an all-inclusive term that describes the sum of knowledge within the profession of software engineering. However, the 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.

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:

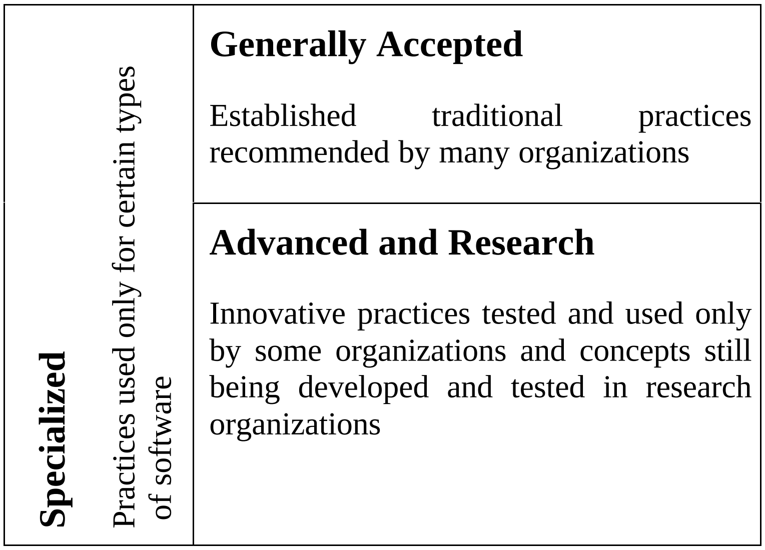
> *‘“Generally accepted” means that the knowledge and practices described are applicable to most projects most of the time, and that there is widespread consensus about their value and usefulness. “Generally accepted” does not mean that the knowledge and practices described are or should be applied uniformly on all projects; the project management team is always responsible for determining what is appropriate for any given project.’*

The Guide to the Project Management Body of Knowledge is now an IEEE Standard.

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.

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.

[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



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

Figure 1 Categories of knowledge

Length of Knowledge Area Description

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.

Role of Editorial Team

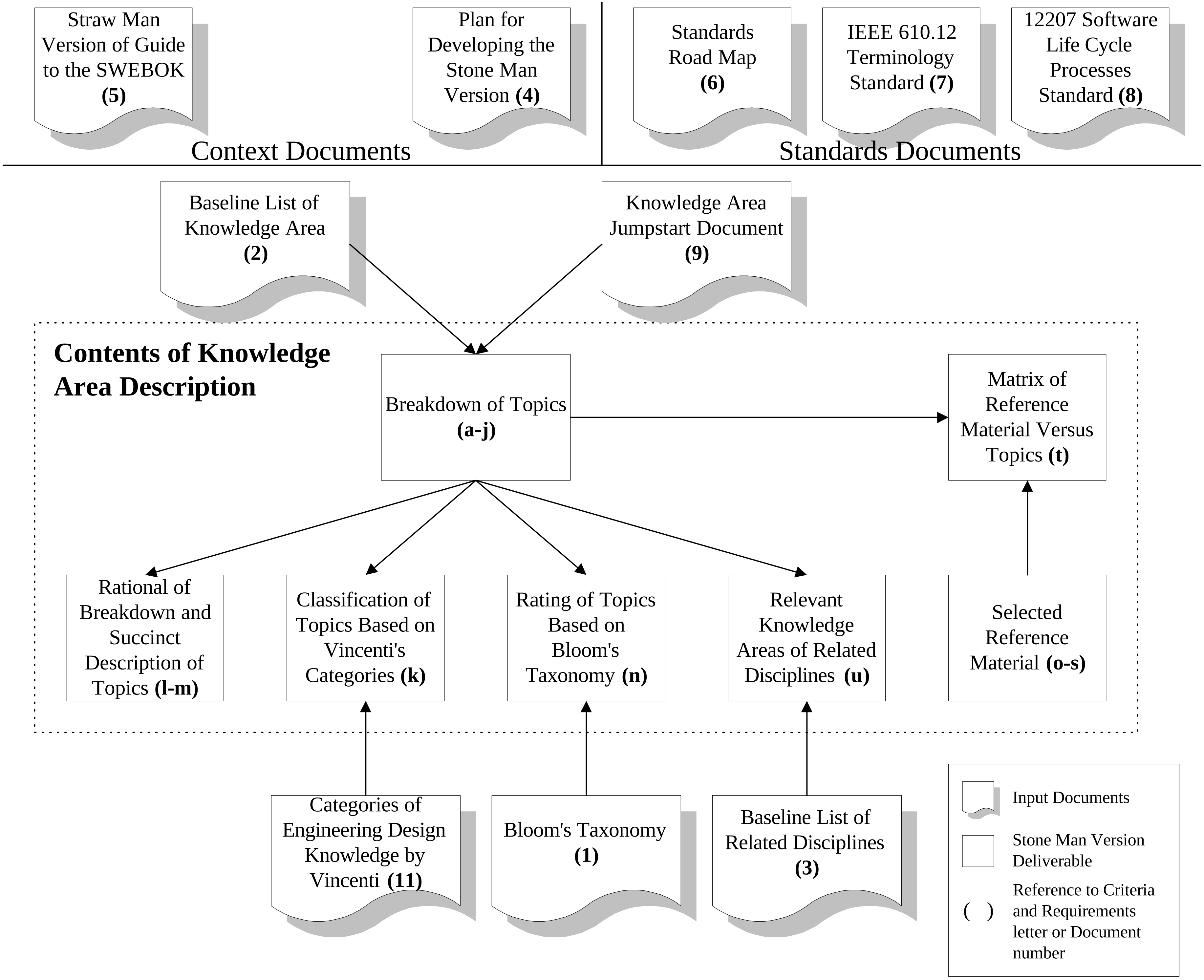
Alain Abran and James W. Moore are the Executive Editors and are responsible for maintaining good relations with the IEEE CS, the ACM, the Industrial Advisory Board and the Panel of Experts as well as for the overall strategy, approach, organization and funding of the project.

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.

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.

Summary

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

Important Related Documents (in alphabetical order of first author)

1) Bloom *et al.*, **Bloom's Taxonomy of the Cognitive Domain**

Please refer to http://www.valdosta.peachnet.edu/~whuitt/psy702/cogsys/bloom.html for a description of this hierarchy of educational objectives.

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.

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.

This document is available at www.swebok.org.

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.

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.

The current version is available at www.swebok.org

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.

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.

This document is available at www.swebok.org

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.

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.

This report is available at www.swebok.org.

6) J. W. Moore, **Software Engineering Standards, A User's Road Map**. Los Alamitos: IEEE Computer Society Press, 1998.

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.

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.

7) **IEEE Standard Glossary of Software Engineering Terminology**, IEEE, Piscataway, NJ std 610.12-1990, 1990.

The hierarchy of references for terminology is Merriam Webster's Collegiate Dictionary (10th Edition), IEEE Standard 610.12 and new proposed definitions if required.

8) **Information Technology – Software Life Cycle Processes**, International Standard, Technical ISO/IEC 12207:1995(E), 1995.

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.

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.

9) **Knowledge Area Jumpstart Documents**

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”.

10) **Merriam Webster's Collegiate Dictionary** (10th Edition).

See note for IEEE 610.12 Standard.

11) W. G. Vincenti, **What Engineers Know and How They Know It - Analytical Studies from Aeronautical History**. Baltimore and London: Johns Hopkins, 1990.

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:

- 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;
- they are viewed by Vincenti as applicable to all branches of engineering;
- 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;
- due to generic nature of the categories, knowledge within each knowledge area could evolve and progress significantly while the framework itself would remain stable;

Authorship of Knowledge Area Description

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.

Style and Technical Guidelines

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).

Knowledge Area Descriptions are expected to follow the IEEE Computer Society Style Guide. See http://computer.org/author/style/cs-style.htm

Microsoft Word 97 is the preferred submission format. Please contact the Editorial Team if this is not feasible for you.

Editing (to be confirmed)

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.

Release of copyright

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.

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.

For more information, See http://computer.org/copyright.htm