Software Engineering

Collection Editor:
Hung Vo

Software Engineering

Collection Editor:
Hung Vo

Authors:
Trung Hung VO
Hung Vo

Online:
< http://cnx.org/content/col10790/1.1/ >

C O N N E X I O N S

Rice University, Houston, Texas

Collection structure revised: July 29, 2009
PDF generated: October 27, 2012
For copyright and attribution information for the modules contained in this collection, see p. 117.

Table of Contents

1 Student Manual

1.1 Syllabus 1
1.2 Assignment 4
1.3 Exercise 11
1.4 Project 12
1.5 Tools 13

2 Lecture notes

2.1 Introduction 15
2.2 Software development process 21
2.3 Requirements analysis 27
2.4 Software Design 41
2.5 Software construction 48
2.6 Software Testing 55
2.7 Software Maintenance 70
2.8 Software configuration management 85
2.9 Software quality management 97
2.10 Software engineering management 107

Index 116
Attributions 117

Chapter 1

Student Manual

1.1 Syllabus[1]

1.1.1 Description

Software engineering is a very broad field. It encompase virtual everything a person might want to know in order to develop software - software that is correct, on time, and on budget. Most other computer science courses emphasize the technical foundations of software development, such as programming, algorithms, data structures, languages, etc.

This course focusses on the pragmatic aspects, such as requirements analysis, cost estimation, design, team organization, quality control, configuration management, verification, testing, and maintenance. Students work in teams on projects for real clients. This work includes a feasibility study, requirements analysis, object-oriented design, implementation, testing, and delivery to the client. Additional topics covered in lectures include professionalism, project management, and the legal framework for software development.

This course is compiled from documents of MIT OpenCourseWare, the Connexion project of Rice University, and from free online courses and documents such as

http://www.ecs.syr.edu/faculty/fawcett/handouts/webpages/FawcettHome.htm[2] ,

http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-171Fall2003/CourseHome/[3] ,

http://en.wikipedia.org/wiki/Software_engineering[4] ,

http://www.cs.cornell.edu/courses/cs501/2008sp/[5] ,

http://www.sei.cmu.edu/[6] ,

http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/[7] ,

http://www.ee.unb.ca/kengleha/courses/CMPE3213/IntroToSoftwareEng.htm[8] ,

http://cs.wwc.edu/~aabyan/435/intro.html[9] ,

http://www.ocwconsortium.org/[10] ,

and etc...

We have arranged their contents to create a complete course. We hope it will be useful to study Software Engineering.

[1] This content is available online at <http://cnx.org/content/m28909/1.1/>.
[2] http://www.ecs.syr.edu/faculty/fawcett/handouts/webpages/FawcettHome.htm
[3] http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-171Fall2003/CourseHome/
[4] http://en.wikipedia.org/wiki/Software_engineering
[5] http://www.cs.cornell.edu/courses/cs501/2008sp/
[6] http://www.sei.cmu.edu/
[7] http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/
[8] http://www.ee.unb.ca/kengleha/courses/CMPE3213/IntroToSoftwareEng.htm
[9] http://cs.wwc.edu/~aabyan/435/intro.html
[10] http://www.ocwconsortium.org/

1.1.2 Course Objectives

Theory, the students have

- got an overview of software engineering,
- got a survey of the history, ethics and risks of software engineering,
- understood various approaches and methodologies used in different phases of software development life-cycle, including requirements analysis and pecification, software design, software construction, software maintenance, and software process.

Practice

- students will execute a real software engineering project,
- students can apply that knowledge in their future research and practice.

1.1.3 Prerequisites

The formal prerequisites for this course are Programming Languages (C/C++, Perl, Java, .Net,...), Data Structures and Algorithms, and Database Management Systems.

The following are the specific capabilities you will need from the prerequisite courses:

- experience with the software development process
- skill in independent programming and problem solving
- skill using an object oriented language
- mathematical maturity, including at least:
  - methods of proof: induction, cases
  - elementary formal logic: working with formulae using logical connectives, quantifiers, modus ponens, implication, satisfaction
- finite state machines: state diagrams, state tables, reachability
- formal languages: BNF, regular expressions

1.1.4 Organization

Class sessions will be a combination of lecture and seminar formats. We'll cover the course's primary topics in these sessions, with reading and homework assigments that provide opportunities to gain a deeper understanding of the underlying issues and techniques of software engineering. Actual labs and software development projects will provide a hands-on approach to exploring these topics throughout the semester. Given that this is a 3 credit course, it is expected that you will spend an average of 10 hours per week (including in-class and lab time) on the course and related material for the duration of the semester.

1.1.5 Readings

Here are some general books on software engineering:

- Sommerville, Ian, Software Engineering, Eighth Edition, Addison-Wesley, 2007.
- Frederick P. Brooks, Jr., The Mythical Man Month, Addison-Wesley, 1972.
- Pfleeger, Shari Lawrence, Software Engineering Theory and Practice, second edition, Prentice- Hall 2001.
- Bernd Bruegge and Allen H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns and Java, Second Edition, Prentice Hall, 2004.

1.1.6 Grading Plan

Since software engineering is a broad subject matter, mastery of the topic as covered by the scope of our course can be manifested in a variety of ways. Your grade in the course will be based on your achievement of the course objectives as demonstrated in your homework assigments, programming deliveries, and participation in class discussions.

Course component grading weight :

- Exercises: 40%
- Group Software Project: 50%
- Participation: 10%

1.1.7 Schedule

Week/Session	Topic	Reading
Week 1	Introduction	Software Engineering; Ian Sommerville; Eighth Edition; Chapters 1, 2, 3. The Mythical Man Month; Frederick P. Brooks
Week 2	Software development process	Software Engineering; Ian Sommerville; Eighth Edition; Chapters 4
Week 3	Requirements analysis	Software Engineering; Ian Sommerville; Eighth Edition; Chapter 6
Week 4	Discussion on assignment #1	
Week 5	Software Design	Software Engineering; Ian Sommerville; Eighth Edition; Chapters 11, 14. Object-Oriented Software Engineering Using UML; Bernd Bruegge and Allen H. Dutoit
Week 6	Software construction	Software Engineering; Ian Sommerville; Eighth Edition; Chapters 17, 18
Week 7	Software Testing	Software Engineering; Ian Sommerville; Eighth Edition; Chapter 23

continued on next page

Week 8	Discussion on assignment #2	
Week 9	Software Maintenance	Software Engineering; Ian Sommerville; Eighth Edition; Chapter 21
Week 10	Software configuration management	Software Engineering; Ian Sommerville; Eighth Edition; Chapter 29
Week 11	Discussion on assignment #3	
Week 12	Software quality management	Software Engineering; Ian Sommerville; Eighth Edition; Chapters 27, 28
Week 13	Software engineering management	Software Engineering; Ian Sommerville; Eighth Edition; Chapters 14, 15, 16. Software Engineering Theory and PracticePfleeger; Shari Lawrence
Week 14	Assignment #4	
Week 15	Summary of principles of instruction, learning and project	Course evaluation

Table 1.1

1.1.8 Suggestions for Success

- Make sure you know what you want to get out of the course before you get very far into it; there are lots of directions to go in, and having a focus will help to inspire you
- Update your journal often; use it as a tool to develop / explore ideas and track your progress
- Remember that the larger software development project is something to be considered throughout the semester; don't wait until the end!
- Raise concerns early; if an assignment or milestone doesn't seem feasible, say so as soon as you can
- "Work hard, learn lots, stay excited, and have fun." –Ray Ontko

1.2 Assignment[11]

1.2.1 Assignment 1. Project Feasibility Study and Plan

Write a short feasibility report that describes the project that you have selected. The exact form of the report is up to you, but it should be well written and suitable to present to an external client. The length is likely to be between five and ten pages.

The report should include the following:

- The client for whom the work will be done.
- Visibility plan. How will you keep in contact with the client and report progress? How will you communicate among your team?
- A statement of the task to be undertaken.
- A preliminary requirements analysis.

[11] This content is available online at <http://cnx.org/content/m28886/1.1/>.

- Suggested deliverables.
- Process to be followed, e.g., modified waterfall model, iterative refinement, prototype, phased development, etc.
- Outline plan, showing principal activities and milestones.
- Discussion of business considerations.
- Risk analysis. What can go wrong? What is your fallback plan?
- Probable technical requirements

This report is a group assignment. All members of the project team should share in the production of the report.

1.2.2 Assignment 2. Progress Report and Presentation

During the semester each team will give three presentations with associated reports on the work completed. You will make a 45 minute presentation to the client, the Instructor and the Teaching Assistant assigned to your project. Everybody is expected to be a presenter at least once.

The first progress report and presentation should complete one third of the total work from the feasibility study.

If you are following an iterative process the first progress report should mark a major milestone when you can report visible progress to your client. Typically, this will include a first set of requirements, a provisional design and a prototype that can be used to demonstrate the functionality of the system, including user interface design.

If you are following a modified waterfall development process, this progress report should include completion of the requirements phase. Carry out the requirements analysis, definition and specification for your project. Write a requirements document. The exact form of the document is up to you, but it should be well written and suitable to present to your client. In writing a requirements report, pay particular attention to the following:

- The report must be understandable by the client.
- The requirements must be specified in sufficient detail to test against the implementation.
- The requirements must be the client's, not your own concepts.
- Design concepts must be clearly separated from requirements.
- Requirements should be partitioned into those that must be met by the first release and those that are optional.

1.2.3 Assignment 3. Progress Report and Presentation

The second presentation and report will follow the same format as the first. You will make a 45 minute presentation to the client, the Instructor and the Teaching Assistant assigned to your project. Remember that everybody is expected to be a presenter at least once during the semester.

The second progress report and presentation should complete two thirds of the total work from the feasibility study.

If you are following an iterative process this progress report should mark a major milestone when you can report visible progress to your client.

If you are following a modified waterfall process, this progress report will be the completion of the design phase. You should probably have already begun implementation.

The exact form of the report is up to you, but it should be well written and suitable to present to your client. In writing a design report, pay particular attention to the following:

- The report must be understandable by the client.
- The design should include both system and program design.
- Design concepts must be clearly separated from requirements and the implementation.
- Update your schedule to confirm that you will have an operational system by the end of the semester.

1.2.4 Assignment 4. Final Presentation

The final presentation and report will follow the same format as the others. You will make a 45 minute presentation to the client, the Instructor and the Teaching Assistant assigned to your project. Remember that everybody is expected to be a presenter at least once during the semester.

This presentation should describe to the client and the course team what has been accomplished during the semester and should include a demonstration of your system in operation. The documentation should be a complete hand-over package, which has been entered into your project management system, so that the client's staff can put your project into production, extend and maintain it.

1.2.5 Assignment grading rubric

Content & Criteria		Points Possible	Points Obtained	Comments
Feasibility and Planning		10		
	What is the scope of the proposed project?	2		
	Is the project technically feasible?	2		
	What are the projected benefits?	2		
	What are the costs, timetable?	2		
	Project Risk Identification?	2		
Requirements		10		
	Requirements identification	2		
	Requirements analysis	2		
	Requirements definition	2		
	Requirements specification	2		
	Requirements report	2		

continued on next page

System and Software Design		20		
	Design describes the system from the software developers' viewpoint	5		
	System design: Match the requirements to hardware or software systems. Establishes an overall system architecture	5		
	Program design: Represent the software system functions in a form that can be transformed into one or more executable programs	5		
	Tools: apply tools for designing	5		
Implementation and Testing		10		
	Is the level of language that the program offers clearly indicated?	1		
	Is it easy to start the program?	1		
	Is the user interface easy to understand? (For example, is the screen layout clear and easy to interpret?)	2		

continued on next page

	Does the program include scoring?	2		
	Individual components are tested against specifications.	2		
	The individual program units are integrated and tested against the design by the students as a complete system.	1		
	The complete system is tested against the requirements by the client.	1		
Maintenance		10		
	Operation: The system is put into practical use.	2.5		
	Maintenance: Errors and problems are identified and fixed.	2.5		
	Evolution: The system evolves over time as requirements change, to add new functions or adapt the technical environment.	2.5		

continued on next page

	Phase out: The system is withdrawn from service.	2.5		
Quality of software		10		
	Maintainability- easy to maintain to meet changing needs	2.5		
	Dependability- reliability, security, safety	2.5		
	Efficiency- not wasting system resources	2.5		
	Usability- user friendly, good documentation	2.5		
Performance of each student's effort in the project		10		
	Student had a major role in the success of the project	2		
	Student always showed up on time for project meetings	2		
	Student completed his/her part of the project.	2		
	Student showed a lot of interest in the project	2		
	Student played a major role in the project design, implementation and validation	2		

continued on next page

Report		10		
	Overview/introduction	2		
	Organization of body	2		
	Quality of result	2		
	Effective data presentation	2		
	Conclusion	2		
Oral presentation		10		
	Organization (presentation refined and clearly explained, information logically introduced and explained, the area of study's connection to the broad topic and the group's work is clearly explained)	2		
	Content (presentation reveals good use of sources, provides pertinent information)	2		
	Presentation (speaks clearly, uses appropriate language, uses appropriate body language, meets time specifications)	2		

continued on next page

	Questions	4		

Table 1.2

1.3 Exercise[12]

1.3.1 Exercises

Exercises (from the readings, assigned labs, software projects, etc.) will be used to complement the lectures and readings. Exercises are due at the beginning of class as specified in the course schedule.

.Here are some exercises (Software Engineering, Ian Sommerville, Eighth Edition, Addison-Wesley):

1. What are the differences between generic software product development and custom software development?
2. What are the important attributes which all software products should have?
3. What are differences between a software process and a software process model?
4. Explain why system testing costs are particularly high for generic software products that are sold to a very wide market.
5. What are the three principal types of critical system? Explain the differences between these.
6. Suggest reasons why dependability is important in critical systems.
7. What are the most important dimensions of system dependability?
8. Why is the cost of assuring dependability exponential?
9. Explain why programs that are developed using evolutionary development are likely to be difficult to maintain.
10. Explain how both are waterfall model of the process and prototyping model can be accommodated in the spiral process model.
11. What are the advantages of providing static and dynamic views of the software process as in the Rational Unifield Process.
12. Suggest why it is important to make a distinction between developing the user requirements and developing system requirements in the requirements engineering process.
13. Explain why a software system that is used in a real-world environment must change or become progressively less useful.
14. Explain why the intangibility of software systems poses special problems for software project management.
15. Explain why the best programmer do not always make the best software managers.
16. Explain why the process of project planning is iterative and why a plan must be continually reviewed during a software project.
17. Briefly explain the purpose of each of the sections in a software project plan.
18. What is the critical distinction between a milestone and a deliverable?
19. Identify and briefly describe four types of requirements that may be defined for a computer-based system.
20. Discuss the problems of using natural language for defining user and system requirements, and show, using small examples, how structuring natural language into forms can help avoid some of these difficulties.
21. Based on your project (for assignment), draw a data-flow diagram modeling the data processing.
22. What is the fundamental difference between hardware and software failures?
23. During the first decades of computers, programming was learned through the guidance of a more experienced programmer, in the way of an apprenticeship. Where in software engineering can you still see the effect of these "good old days"? Compare to other fields of engineering such as construction that has also originally been learned by starting as a handyman at a construction site.

[12] This content is available online at <http://cnx.org/content/m28911/1.1/>.

24. Software is often a part of various devices and systems where malfunction can have a very dramatic effect on people or the entire society - consider for example banking systems, power distribution or hospital equipment. What is (or, what should be) the responsibility of software providers (organizations or individual software engineers) for their work? In some countries software engineering is a certified profession in a similar way as doctors: what are the advantages and disadvantages of this solution? Do you think that it solves the problem above: why / why not?
25. Explain why the process of project planning is iterative and requires continuous reviewing and revising of the plan. Consider each part of the project plan one by one and try to think of reasons that may cause changes in the part. How could you make it easier to update the plan?
26. What product, hardware, personnel, and project factors does the project have that should be taken into account as adjustment factors (cost drivers) when computing the final effort estimate?
27. Suggest reasons why this statement is true - or is it true? What should the project manager of a late software project do to cure the situation?
28. Give non-functional requirements to your project (for assignments).
29. Explain why it is almost inevitable that the requirements of different stakeholders will conflict in some ways. How should the project group handle conflicting requirements?
30. What is the status of modeling in requirements engineering? Why is one model usually not sufficient in the modeling phase?
31. Create a UML use case diagram of a WWW-based store software. You can concentrate on the actors and use cases. Give a short description of each use case (but don't get into too deep details).
32. Create a UML class diagram based on aggregates and composition of a structure of a laptop computer.
33. Create a data flow diagram of an ATM (automatic teller machine).
34. Create a UML activity diagram of an ATM.
35. You have been assigned to calculate the number of days between two given dates (the same date = 0 days). Create three different abstractions of your solution.
36. How do components differ from objects? Why does object-oriented design become easier when a new abstract entity, component, is added between subsystems and objects?
37. What kind of effects do you see in the use of software product families on requirements engineering, design, implementation, testing and project management?
38. Why does integration testing reveal errors although unit testing has been done well? Why does system testing reveal errors although integration testing has been done well? Why does beta-testing reveal errors although system testing has been done well?
39. Standards and defined work processes are an essential part of quality assurance. On the other hand, software engineers sometimes oppose them, claiming that they stifle technological innovation. Give examples of situations where adhering to standards might be harmful. What problems (if any) would bending the rules cause in your example cases?
40. Software process measurement often involves measuring the work of individual people in the process, such as time spent or faults made in certain tasks. What problems are there in collecting this kind of information and how could these problems be reduced?
41. Suggest a few application domains where the SEI capability model is unlikely to be appropriate. Give reasons why this is the case. Do you think that some of the process types described by Sommerville (informal, managed, methodical, improving) would be better in describing typical software processes in these application fields?

1.4 Project[13]

Software Engineering Project aims to provide students with the experience of developing a medium-scale computing project in a small team. All aspects of the development process are considered: problem specification, requirement extraction, system design, implementation, integration, testing and documentation. This

[13] This content is available online at <http://cnx.org/content/m28926/1.1/>.

course provides students with the experience of working in a team and dealing with the associated problems of communication and team management.

A client can be any person or organization except yourself (e.g., a member of faculty or staff, a department of university, a local company or other external organization, a student body, etc.). Some potential projects and clients will be suggested but you are encouraged to identify your own. There should be a firm intention by the client to use the software in production.

In selecting a project, think broadly. Your project can be an application, system software, or even a toolkit. Software engineering covers everything from Palm Pilots to supercomputers. The only conditions are that there must be a real client and real users.

Here are some suggested projects. They will be discussed during class. Additional project suggestions may be posted here during the first week of class.

1.4.1 Development a web portal

We reuse a available web portal such as DotnetNuke, IBM Web Sphere, Liferay, PHP-Nuke... to develop a new application. We would like to transform it into a database-driven site and would like for your students to build the architecture, including a database and web interface.

For example, a web portal for management a library, our hope is to have searching capabilities, including the option to search the entire site or selected areas of the site. Once the initial architecture is built, our library staff would like to be able to add records and keywords into the database from a web-based form. It would be important for our staff to maintain the system with very little programming knowledge on our part. Our preference for software would be MySQL/php, but we are very flexible on that part.

1.4.2 Interactive Helpdesk and Inventory System

The aim of this project is to build a Helpdesk and Inventory system for a university selected (example, for the university of Danang). The system has been designed to allow the ICT Staff to administrate problems in their computer systems easily and efficently, by providing an easy to use Helpdesk program. It also has an integrated inventory system for both Hardware and Software, and the Hardware system automatically updates itself with the latest information on all the computers that the university has.

1.4.3 An IDE for Microcontrollers in Linux

The puropse of this project is to provide an easy to use Integrated Developmment Environment for the Atmel series of AVR microcontrollers in Linux. Currently there are few options available if using Linux whilst developing for an AVR microcontroller, with most software being Windows only. Thus the client for this project, who wishes to use Linux for development, has the need for such software.

The main goal of the project is to streamline the tasks of compiling, debugging, emulating and transferring AVR programs. This will be accomplished by developing a plugin for the free open source text editor jEdit, which will integrate all the tools required into one easy to use package.

1.5 Tools[14]

http://xmlbasedsrs.tigris.org/[15]

Writing software requirements specifications as XML documents has quite a few advantages. Using open source tools like Emacs, PSGML, CVS and xsltproc gives us a powerful Requirements Engineering tool.

- Gather information using Emacs
- Validating specifications using psgml and XSLT

[14] This content is available online at <http://cnx.org/content/m28950/1.1/>.
[15] http://xmlbasedsrs.tigris.org/

- Keeping track of changes using CVS and XSLT
- Creating nice output with xsltproc
- Transforming requirements data to project management tools

http://en.wikipedia.org/wiki/List_of_UML_tools[16]

This page lists Unified Modeling Language tools, classified by their proprietary or non-proprietary status.

http://subversion.tigris.org/[17]

Subversion is an open source version control system. Subversion was originally designed to be a better CVS, so it has most of CVS's features. Generally, Subversion's interface to a particular feature is similar to CVS's, except where there's a compelling reason to do otherwise.

Subversion has since expanded beyond its original goal of replacing CVS, but its history influenced its feature and interface choices; Subversion today should still feel very familiar to CVS users.

http://www.iterating.com/productclasses/Software-Engineering-Tools[18]

Software Engineering Tools are also known as Computer-Aided Software Engineering Tools (CASE Tools). They are graphical, interactive tools used for the analyzing and designing the phases of application software development.

http://www.opensourcetesting.org/[19]

There are some open source software testing tools.

http://en.wikipedia.org/wiki/List_of_project_management_software[20]

This is a list of notable software project management applications.

[16] http://en.wikipedia.org/wiki/List_of_UML_tools
[17] http://subversion.tigris.org/
[18] http://www.iterating.com/productclasses/Software-Engineering-Tools
[19] http://www.opensourcetesting.org/
[20] http://en.wikipedia.org/wiki/List_of_project_management_software

Chapter 2

Lecture notes

2.1 Introduction[1]

2.1.1 Introduction

Virtually all countries now depend on complex computer-based systems. More and more products incorporate computers and controlling software in some form. The software in these systems represents a large and increasing proportion of the total system costs. Therefore, producing software in a cost-effective way is essential for the functioning of national and international economies.

Software engineering is an engineering discipline whose goal is the cost-effective development of software systems. Software is abstract and intangible. It is not constrained by materials, governed by physical laws or by manufacturing processes. In some ways, this simplifies software engineering as there are no physical limitations on the potential of software. In other ways, however, this lack of natural constraints means that software can easily become extremely complex and hence very difficult to understand.

Software engineering is still a relatively young discipline. The notion of 'software engineering' was first proposed in 1968 at a conference held to discuss what was then called the 'software crisis'. This software crisis resulted directly from the introduction of powerful, third generation computer hardware. Their power made hitherto unrealisable computer applications a feasible proposition. The resulting software was orders of magnitude larger and more complex than previous software systems.

Early experience in building these systems showed that an informal approach to software development was not good enough. Major projects were sometimes years late. They cost much more than originally predicted, were unreliable, difficult to maintain and performed poorly. Software development was in crisis. Hardware costs were tumbling whilst software costs were rising rapidly. New techniques and methods were needed to control the complexity inherent in large software systems.

These techniques have become part of software engineering and are now widely although not universally used. However, there are still problems in producing complex software which meets user expectations, is delivered on time and to budget. Many software projects still have problems and this has led to some commentators (Pressman, 1997) suggesting that software engineering is in a state of chronic affliction.

As our ability to produce software has increased so too has the complexity of the software systems required. New technologies resulting from the convergence of computers and communication systems place new demands on software engineers. For this reason and because many companies do not apply software engineering techniques effectively, we still have problems. Things are not as bad as the doomsayers suggest but there is clearly room for improvement.

We have made tremendous progress since 1968 and that the development of software engineering has markedly improved our software. We have a much better understanding of the activities involved in software

[1] This content is available online at <http://cnx.org/content/m28922/1.1/>.

development. We have developed effective methods of software specification, design and implementation. New notations and tools reduce the effort required to produce large and complex systems.

Software engineers can be rightly proud of their achievements. Without complex software we would not have explored space, would not have the Internet and modern telecommunications, and all forms of travel would be more dangerous and expensive. Software engineering has contributed a great deal in its short lifetime and I am convinced that, as the discipline matures, its contributions in the 21st century will be even greater.

2.1.1.1 What is software?

Many people equate the term software with computer programs. In fact, this is too restrictive a view. Software is not just the programs but also all associated documentation and configuration data which is needed to make these programs operate correctly. A software system usually consists of a number of separate programs, configuration files which are used to set up these programs, system documentation which describes the structure of the system and user documentation which explains how to use the system and, for software products, web sites for users to download recent product information.

Software engineers are concerned with developing software products i.e. software which can be sold to a customer. There are two types of software product:

1. Generic products: These are stand-alone systems which are produced by a development organisation and sold on the open market to any customer who is able to buy them. Sometimes they are referred to as shrink-wrapped software. Examples of this type of product include databases, word processors, drawing packages and project management tools.

2. Bespoke (or customised) products: These are systems which are commissioned by a particular customer. The software is developed specially for that customer by a software contractor. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process and air traffic control systems.

An important difference between these different types of software is that, in generic products, the organisation which develops the software controls the software specification. For custom products, the specification is usually developed and controlled by the organisation who are buying the software. The software developers must work to that specification.

2.1.1.2 What is software engineering?

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

1. Engineering discipline: Engineers make things work. They apply theories, methods and tools where these are appropriate but they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods to support them. Engineers also recognise that they must work to organisational and financial constraints so they look for solutions within these constraints.

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

In general, software engineers adopt a systematic and organised approach to their work as this is often the most effective way to produce high-quality software. However, engineering is all about selecting the most appropriate method for a set of circumstances and a more creative, informal approach to development may be effective in some circumstances. Informal development is particularly appropriate for the development of web-based e-commerce systems which requires a blend of software and graphical design skills.

2.1.1.3 What's the difference between software engineering and computer science?

Essentially, computer science is concerned about theories and methods which underlie computers and software systems whereas software engineering is concerned with the practical problems of producing software. Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers.

Ideally, all of software engineering should be underpinned by theories of computer science but in reality this is not the case. Software engineers must often use ad hoc approaches to develop the software. Elegant theories of computer science are cannot always be applied to real, complex problems which require a software solution.

2.1.1.4 What is the difference between software engineering and system engineering?

System engineering or, more precisely, computer-based system engineering is concerned with all aspects of the development and evolution of complex systems where software plays a major role. System engineering is therefore concerned with hardware development, policy and process design and system deployment as well as software engineering. System engineers are involved in specifying the system, defining its overall architecture and then integrating the different parts to create the finished system. They are less concerned with the engineering of the system components (hardware, software, etc.).

System engineering is an older discipline than software engineering. People have been specifying and assembling complex industrial systems such as aircraft and chemical plants for more than 100 years. However, as the percentage of software in systems has increased, software engineering techniques such as use-case modelling, configuration management, etc. are being used in the systems engineering process.

2.1.1.5 What is a software process?

A software process is the set of activities and associated results which produce a software product. These activities are mostly carried out by software engineers. There are four fundamental process activities which are common to all software processes. These activities are:

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

Different software processes organise these activities in different ways and are described at different levels of detail. The timing of the activities varies as does the results of each activity. Different organisations may use different processes to produce the same type of product. However, some processes are more suitable than others for some types of application. If an inappropriate process is used, this will probably reduce the quality or the usefulness of the software product to be developed.

2.1.1.6 What is a software process model?

A software process model is a simplified description of a software process which is presented from a particular perspective. Models, by their very nature, are simplifications so a software process model is an abstraction of the actual process which is being described. Process models may include activities which are part of the software process, software products and the roles of people involved in software engineering. Some examples of the types of software process model which may be produced are:

1. A workflow model: This shows the sequence of activities in the process along with their inputs, outputs and dependencies. The activities in this model represent human actions.

2. A dataflow or activity model: This represents the process as a set of activities each of which carries out some data transformation. It shows how the input to the process such as a specification is transformed to an output such as a design. The activities here may be at a lower-level than activities in a workflow model. They may represent transformations carried out by people or by computers.

3. A role/action model: This represents the roles of the people involved in the software process and the activities for which they are responsible. There are a number of different general models or paradigms of software development:

- The waterfall approach: This takes the above activities and represents them as separate process phases such as requirements specification, software design, implementation, testing and so on. After each stage is defined it is 'signed-off' and development goes on to the following stage.
- Evolutionary development: This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from very abstract specifications. This is then refined with customer input to produce a system which satisfies the customer's needs.

2.1.1.7 What are the costs of software engineering?

There is no simple answer to this question as the precise distribution of costs across the software process depends on the process used and the type of software which is being developed. If we take the total cost of developing a complex software system as 100 cost units, the distribution of these cost units is 25 for specification, 25 for design, 25 for development and the rest for intergration and testing.

This cost distribution holds where the costs of specification, design, implementation and integration are measured separately. Notice that system integration and testing is the most expensive development activity.

The evolution costs for generic software products are particularly hard to estimate. In many cases, there is little formal evolution of a product. Once a version of the product has been released, work starts on the next release and, for marketing reasons, this is likely to be presented as a new (but compatible) product rather than a modified version of a product which the user has already bought. Therefore, the evolution costs are not assessed separately as they are in customised software but are simply the development costs for the next version of the system.

2.1.1.8 What are software engineering methods?

A software engineering method is a structured approach to software development whose aim is to facilitate the production of high-quality software in a cost-effective way. Methods such as Structured Analysis (DeMarco, 1978) and JSD (Jackson, 1983) were first developed in the 1970s. These methods attempted to identify the basic functional components of a system and function-oriented methods are still widely used. In the 1980s and 1990s, these function-oriented methods were supplemented by object-oriented methods such as those proposed by Booch (Booch, 1994) and Rumbaugh (Rumbaugh, Blaha et al., 1991). These different approaches have now been integrated into a single unified approach built around the Unified Modeling Language (UML) (Fowler and Scott, 1997) (Booch, Rumbaugh et al., 1999; Rumbaugh, Jacobson et al., 1999; Rumbaugh, Jacobson et al., 1999).

All methods are based on the idea of developing models of a system which may be represented graphically and using these models as a system specification or design.

There is no ideal method and different methods have different areas where they are applicable. For example, object-oriented methods are often appropriate for interactive systems but not for systems with stringent real-time requirements.

2.1.1.9 What is CASE?

The acronym CASE stands for Computer-Aided Software Engineering. It covers a wide range of different types of program which are used to support software process activities such as requirements analysis, system modelling, debugging and testing.

All methods now come with associated CASE technology such as editors for the notations used in the method, analysis modules which check the system model according to the method rules and report generators to help create system documentation. The CASE tools may also include a code generator which automatically

generates source code from the system model and some process guidance which gives advice to the software engineer on what to do next.

This type of CASE tool, aimed at supporting analysis and design, is sometimes called an upper-CASE tool because it supports early phases of the software process. By contrast, CASE tools which are designed to support implementation and testing such as debuggers, program analysis systems, test case generators and program editors are sometimes called lower-CASE tools.

2.1.1.10 What are the attributes of good software?

As well as the services which it provides, software products have a number of other associated attributes which reflect the quality of that software. These attributes are not directly concerned with what the software does, Rather, they reflect its behaviour while it is executing and the structure and organisation of the source program and associated documentation. Examples of these attributes (sometimes called non-functional attributes) are the software's response time to a user query and the understandability of the program code.

The specific set of attributes which you might expect from a software system obviously depends on its application. Therefore, a banking system must be secure, an interactive game must be responsive, a telephone switching system must be reliable, etc.

2.1.1.11 What are the key challenges facing software engineering?

Software engineering in the 21st century faces three key challenges:

1. The legacy challenge: The majority of software systems which are in use today were developed many years ago yet they perform critical business functions. The legacy challenge is the challenge of maintaining and updating this software in such a way that excessive costs are avoided and essential business services continue to be delivered.

2. The heterogeneity challenge: Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and with different kinds of support systems. The heterogeneity challenge is the challenge of developing techniques to build dependable software which is flexible enough to cope with this heterogeneity.

3. The delivery challenge: Many traditional software engineering techniques are time-consuming. The time they take is required to achieve software quality. However, businesses today must be responsive and change very rapidly. Their supporting software must change equally rapidly. The delivery challenge is the challenge of shortening delivery times for large and complex systems without compromising system quality.

Of course, these are not independent. For example, it may be necessary to make rapid changes to a legacy system to make it accessible across a network. To address these challenges we will need new tools and techniques as well as innovative ways of combining and using existing software engineering methods.

2.1.2 Professional and ethical responsibility

Like other engineers, software engineers must accept that their job involves wider responsibilities than simply the application of technical skills. Their work is carried out within a legal and social framework. Software engineering is obviously bounded by local, national and international laws. Software engineers must behave in an ethical and morally responsible way if they are to be respected as professionals.

It goes without saying that engineers should uphold normal standards of honesty and integrity. They should not use their skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are areas where standards of acceptable behaviour are not bounded by laws but by the more tenuous notion of professional responsibility.

Some of these are:

1. Confidentiality: Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.

2. Competence: Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outwith their competence.

3. Intellectual property rights: Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.

4. Computer misuse: Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

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

The ACM and the IEEE have cooperated to produce a joint code of ethics and professional practice. The rationale behind this code is summarised in the first two paragraphs of the longer form:

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

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

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

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

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

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

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that

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

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

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

You must make up your own mind in these matters. In this case, the potential for damage, the extent of the damage and the people affected by the damage should influence the decision. If the situation is very dangerous, it may be justified to publicise it using the national press. However, you should always try to resolve the situation while respecting the rights of your employer.

Another ethical issue is participation in the development of military and nuclear systems. Some people feel strongly about these issues and do not wish to participate in any systems development associated with military systems. Others will work on military systems but not on weapons systems. Yet others feel that national defence is an overriding principle and have no ethical objections to working on weapons systems. The appropriate ethical position here depends entirely on the views of the individuals who are involved.

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

The general area of ethics and professional responsibility is one which has received increasing attention over the past few years. It can be considered from a philosophical standpoint where the basic principles of ethics are considered and software engineering ethics are discussed with reference to these basic principles.

References :

http://en.wikipedia.org/wiki/Software_engineering, http://cs.wwc.edu/~aabyan/435/intro.html, http://www.cc.gatech.edu/classes/AY2000/cs3802_fall/index.html, http://www.engin.umd.umich.edu/CIS/course.des/cis37

2.2 Software development process[2]

A software development process is a structure imposed on the development of a software product. Synonyms include software lifecycle and software process. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.

[2]This content is available online at <http://cnx.org/content/m28927/1.2/>.

2.2.1 Processes and meta-processes

A growing body of software development organizations implement process methodologies. Many of them are in the defense industry, which in the U.S. requires a rating based on 'process models' to obtain contracts. The international standard for describing the method of selecting, implementing and monitoring the life cycle for software is ISO 12207.

The Capability Maturity Model (CMM) is one of the leading models. Independent assessments grade organizations on how well they follow their defined processes, not on the quality of those processes or the software produced. CMM is gradually replaced by CMMI. ISO 9000 describes standards for formally organizing processes with documentation.

ISO 15504, also known as Software Process Improvement Capability Determination (SPICE), is a "framework for the assessment of software processes". This standard is aimed at setting out a clear model for process comparison. SPICE is used much like CMM and CMMI. It models processes to manage, control, guide and monitor software development. This model is then used to measure what a development organization or project team actually does during software development. This information is analyzed to identify weaknesses and drive improvement. It also identifies strengths that can be continued or integrated into common practice for that organization or team.

2.2.2 Process activities/steps

DesignOperation andMaintenanceRequirementImplementationFeasibility andPlanningTesting

1. Requirements analysis: The most important task in creating a software product is extracting the requirements. Customers typically know what they want, but not what software should do, while incomplete, ambiguous or contradictory requirements are recognized by skilled and experienced software engineers. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect.

The system's services, constraints and goals are established by consultation with system users. They are then defined in a manner that is understandable by both users and development staff.

This phase can be divided into:

- Feasibility study (often carried out separately)
- Requirements analysis
- Requirements definition
- Requirements specification

2. Design

- System design: Partition the requirements to hardware or software systems. Establishes an overall system architecture. The architecture of a software system refers to an abstract representation of that system. Architecture is concerned with making sure the software system will meet the requirements of the product, as well as ensuring that future requirements can be addressed. The architecture step also addresses interfaces between the software system and other software products, as well as the underlying hardware or the host operating system.
- Software design: Represent the software system functions in a form that can be transformed into one or more executable programs. Specification is the task of precisely describing the software to be written, possibly in a rigorous way. In practice, most successful specifications are written to understand and fine-tune applications that were already well-developed, although safety-critical software systems are often carefully specified prior to application development. Specifications are most important for external interfaces that must remain stable.

3. Implementation (or coding): Reducing a design to code may be the most obvious part of the software engineering job, but it is not necessarily the largest portion. An important (and often overlooked) task is documenting the internal design of software for the purpose of future maintenance and enhancement. Documentation is most important for external interfaces.

4. Testing: Testing of parts of software, especially where code by two different engineers must work together, falls to the software engineer.

6. Operation and maintenance: A large percentage of software projects fail because the developers fail to realize that it doesn't matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are occasionally resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, its very important to have training classes for the most enthusiastic software users (build excitement and confidence), shifting the training towards the neutral users intermixed with the avid supporters, and finally incorporate the rest of the organization into adopting the new software. Users will have lots of questions and software problems which leads to the next phase of software. Maintaining and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. Not only may it be necessary to add code that does not fit the original design but just determining how software works at some point after it is completed may require significant effort by a software engineer. About 2/3 of all software engineering work is maintenance, but this statistic can be misleading. A small part of that is fixing bugs. Most maintenance is extending systems to do new things, which in many ways can be considered new work. In comparison, about 2/3 of all civil engineering, architecture, and construction work is maintenance in a similar way.

2.2.3 Process models

A decades-long goal has been to find repeatable, predictable processes or methodologies that improve productivity and quality. Some try to systematize or formalize the seemingly unruly task of writing software. Others apply project management techniques to writing software. Without project management, software projects can easily be delivered late or over budget. With large numbers of software projects not meeting their expectations in terms of functionality, cost, or delivery schedule, effective project management is proving difficult.

2.2.3.1 Waterfall process

In 1970 Royce proposed what is presently referred to as the waterfall model as an initial concept, a model which he argued was flawed (Royce 1970). His paper then explored how the initial model could be developed into an iterative model, with feedback from each phase influencing subsequent phases, similar to many methods used widely and highly regarded by many today. It is only the initial model that received notice; his own criticism of this initial model has been largely ignored. The "waterfall model" quickly came to refer not to Royce's final, iterative design, but rather to his purely sequentially ordered model. This article will use this popular meaning of the phrase waterfall model. For an iterative model similar to Royce's final vision, see the spiral model.

Despite Royce's intentions for the waterfall model to be modified into an iterative model, use of the "waterfall model" as a purely sequential process is still popular, and, for some, the phrase "waterfall model" has since come to refer to any approach to software creation which is seen as inflexible and non-iterative. Those who use the phrase waterfall model pejoratively for non-iterative models that they dislike usually see the waterfall model itself as naive and unsuitable for a "real world" process.

The best-known and oldest process is the waterfall model, where developers (roughly) follow these steps in order:

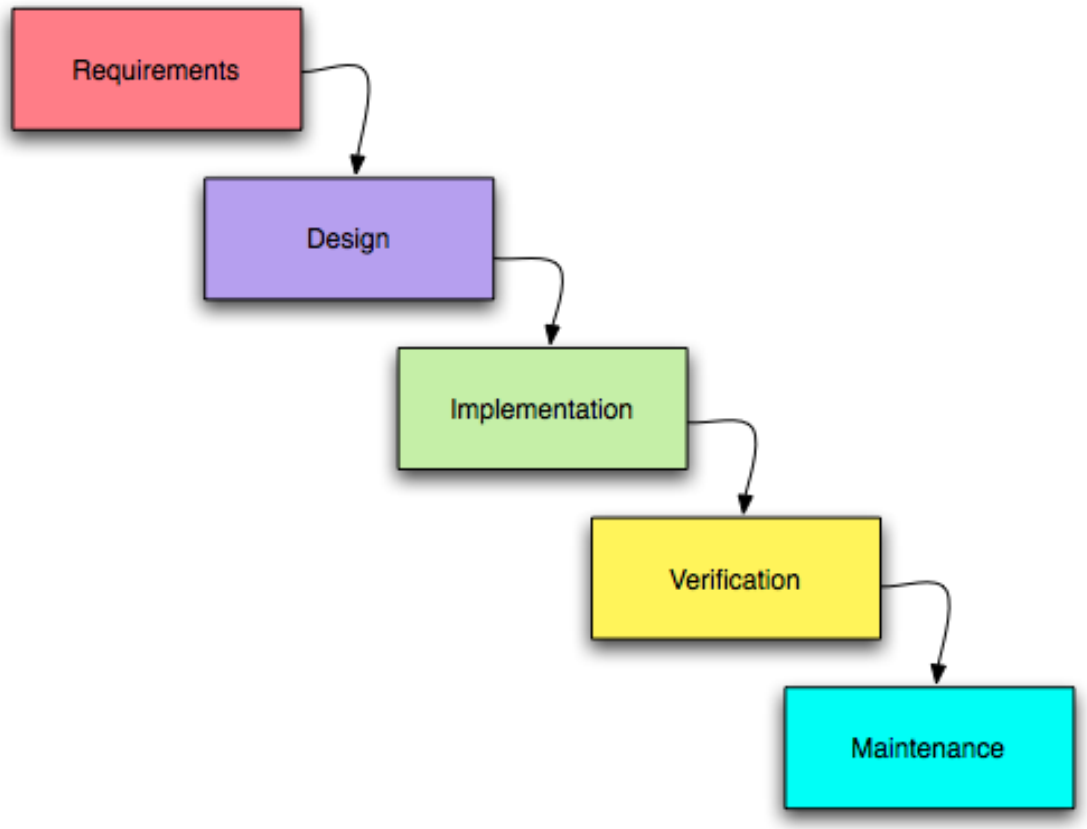


Figure 2.1: Process activities/steps

To follow the waterfall model, one proceeds from one phase to the next in a purely sequential manner. For example, one first completes "requirements specification" — they set in stone the requirements of the software. When the requirements are fully completed, one proceeds to design. The software in question is designed and a "blueprint" is drawn for implementers (coders) to follow — this design should be a plan for implementing the requirements given. When the design is fully completed, an implementation of that design is made by coders. Towards the later stages of this implementation phase, disparate software components produced by different teams are integrated. After the implementation and integration phases are complete, the software product is tested and debugged; any faults introduced in earlier phases are removed here. Then the software product is installed, and later maintained to introduce new functionality and remove bugs.

Thus the waterfall model maintains that one should move to a phase only when its preceding phase is completed and perfected. Phases of development in the waterfall model are thus discrete, and there is no jumping back and forth or overlap between them.

There is a misconception that the process has no provision for correcting errors in early steps (for example, in the requirements). In fact this is where the domain of requirements management comes in which includes change control.

This approach is used in high risk projects, particularly large defense contracts. The problems in waterfall do not arise from "immature engineering practices, particularly in requirements analysis and requirements management." Studies of the failure rate of the specification, which enforced waterfall, have shown that the

more closely a project follows its process, specifically in up-front requirements gathering, the more likely the project is to release features that are not used in their current form.

More often too the supposed stages are part of joint review between customer and supplier, the supplier can, in fact, develop at risk and evolve the design but must sell off the design at a key milestone called Critical Design Review (CDR). This shifts engineering burdens from engineers to customers who may have other skills.

The waterfall model however is argued by many to be a bad idea in practice, mainly because of their belief that it is impossible to get one phase of a software product's lifecycle "perfected" before moving on to the next phases and learning from them (or, at least, the belief that this is impossible for any non-trivial program). For example, clients may not be aware of exactly what requirements they want before they see a working prototype and can comment upon it; they may change their requirements constantly, and program designers and implementers may have little control over this. If clients change their requirements after a design is finished, that design must be modified to accommodate the new requirements, invalidating quite a good deal of effort if overly large amounts of time have been invested into "Big Design Up Front". (Thus, methods opposed to the naive waterfall model–such as those used in Agile software development–advocate less reliance on a fixed, static requirements document or design document). Designers may not (or, more likely, cannot) be aware of future implementation difficulties when writing a design for an unimplemented software product. That is, it may become clear in the implementation phase that a particular area of program functionality is extraordinarily difficult to implement. If this is the case, it is better to revise the design than to persist in using a design that was made based on faulty predictions and that does not account for the newly discovered problem areas.

The idea behind the waterfall model may be "measure twice; cut once", and those opposed to the waterfall model argue that this idea tends to fall apart when the problem being measured is constantly changing due to requirement modifications and new realizations about the problem itself. The idea behind those who object to the waterfall model may be "time spent in reconnaissance is seldom wasted".

In summary, the criticisms of a non-iterative development approach (such as the waterfall model) are as follows:

- Many software projects must be open to change due to external factors; the majority of software is written as part of a contract with a client, and clients are notorious for changing their stated requirements. Thus the software project must be adaptable, and spending considerable effort in design and implementation based on the idea that requirements will never change is neither adaptable nor realistic in these cases.
- Unless those who specify requirements and those who design the software system in question are highly competent, it is difficult to know exactly what is needed in each phase of the software process before some time is spent in the phase "following" it. That is, feedback from following phases is needed to complete "preceding" phases satisfactorily. For example, the design phase may need feedback from the implementation phase to identify problem design areas. The counter-argument for the waterfall model is that experienced designers may have worked on similar systems before, and so may be able to accurately predict problem areas without time spent prototyping and implementing.
- Constant testing from the design, implementation and verification phases is required to validate the phases preceding them. Constant "prototype design" work is needed to ensure that requirements are non-contradictory and possible to fulfill; constant implementation is needed to find problem areas and inform the design process; constant integration and verification of the implemented code is necessary to ensure that implementation remains on track. The counter-argument for the waterfall model here is that constant implementation and testing to validate the design and requirements is only needed if the introduction of bugs is likely to be a problem. Users of the waterfall model may argue that if designers (et cetera) follow a disciplined process and do not make mistakes that there is no need for constant work in subsequent phases to validate the preceding phases.
- Frequent incremental builds (following the "release early, release often" philosophy) are often needed to build confidence for a software production team and their client.
- It is difficult to estimate time and cost for each phase of the development process without doing some

"recon" work in that phase, unless those estimating time and cost are highly experienced with the type of software product in question.
- The waterfall model brings no formal means of exercising management control over a project and planning control and risk management are not covered within the model itself.
- Only a certain number of team members will be qualified for each phase; thus to have "code monkeys" who are only useful for implementation work do nothing while designers "perfect" the design is a waste of resources. A counter-argument to this is that "multiskilled" software engineers should be hired over "specialized" staff.

2.2.3.2 Iterative process

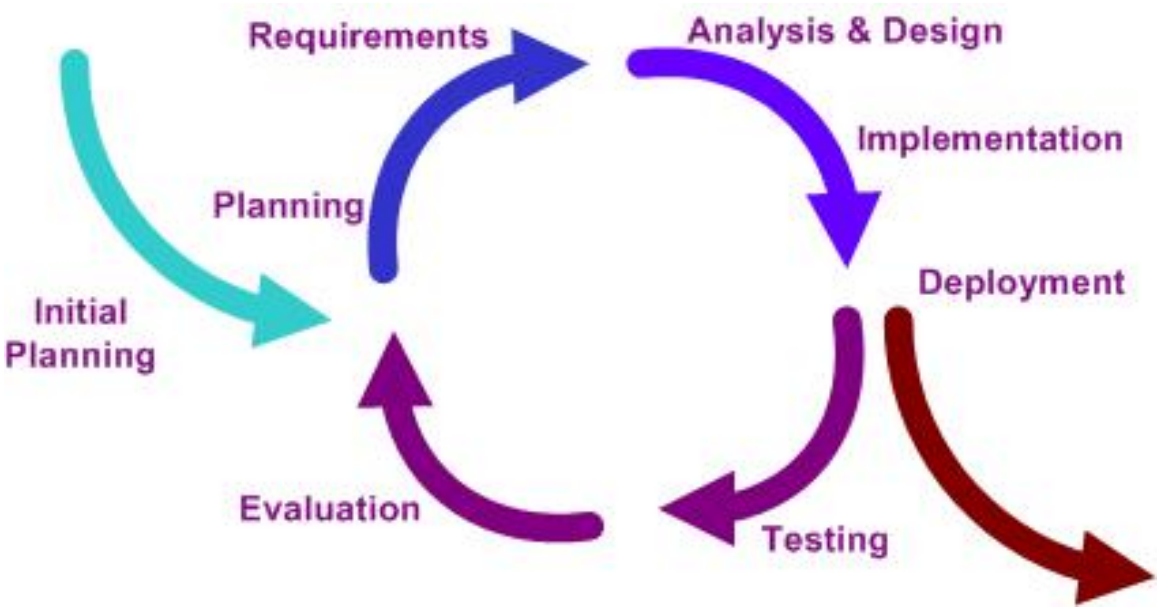


Figure 2.2: Iterative process

Iterative development prescribes the construction of initially small but ever larger portions of a software project to help all those involved to uncover important issues early before problems or faulty assumptions can lead to disaster. Iterative processes are preferred by commercial developers because it allows a potential of reaching the design goals of a customer who does not know how to define what they want.

While Iterative development approaches have their advantages, software architects are still faced with the challenge of creating a reliable foundation upon which to develop. Such a foundation often requires a fair amount of upfront analysis and prototyping to build a development model. The development model often relies upon specific design patterns and entity relationship diagrams (ERD). Without this upfront foundation, Iterative development can create long term challenges that are significant in terms of cost and quality.

Critics of iterative development approaches point out that these processes place what may be an unreasonable expectation upon the recipient of the software: that they must possess the skills and experience of a seasoned software developer. The approach can also be very expensive if iterations are not small enough to mitigate risk; akin to... "If you don't know what kind of house you want, let me build you one and see

if you like it. If you don't, we'll tear it all down and start over." By analogy the critic argues that up-front design is as necessary for software development as it is for architecture. The problem with this criticism is that the whole point of iterative programming is that you don't have to build the whole house before you get feedback from the recipient. Indeed, in a sense conventional programming places more of this burden on the recipient, as the requirements and planning phases take place entirely before the development begins, and testing only occurs after development is officially over.

These approaches have been developed along with web based technologies. As such, they are actually more akin to maintenance life cycles given that most of the architecture and capability of the solutions is embodied within the technology selected as the back bone of the application.

References:

http://en.wikipedia.org/wiki/Software_development_process, http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-171Fall2003/CourseHome/, http://www.cs.cornell.edu/courses/cs501/2008sp/, http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/, http://www.ee.unb.ca/kengleha/courses/CMPE3213/IntroT http://www.d.umn.edu/~gshute/softeng/process.html, http://www.sei.cmu.edu/community/easel/demos/SWprocess.html, etc...

2.3 Requirements analysis[3]

2.3.1 Introduction

In systems engineering and software engineering, requirements analysis encompasses those tasks that go into determining the requirements of a new or altered system, taking account of the possibly conflicting requirements of the various stakeholders, such as users. Requirements analysis is critical to the success of a project.

Systematic requirements analysis is also known as requirements engineering. It is sometimes referred to loosely by names such as requirements gathering, requirements capture, or requirements specification. The term "requirements analysis" can also be applied specifically to the analysis proper (as opposed to elicitation or documentation of the requirements, for instance).

Requirements must be measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

[3]This content is available online at <http://cnx.org/content/m14621/1.6/>.

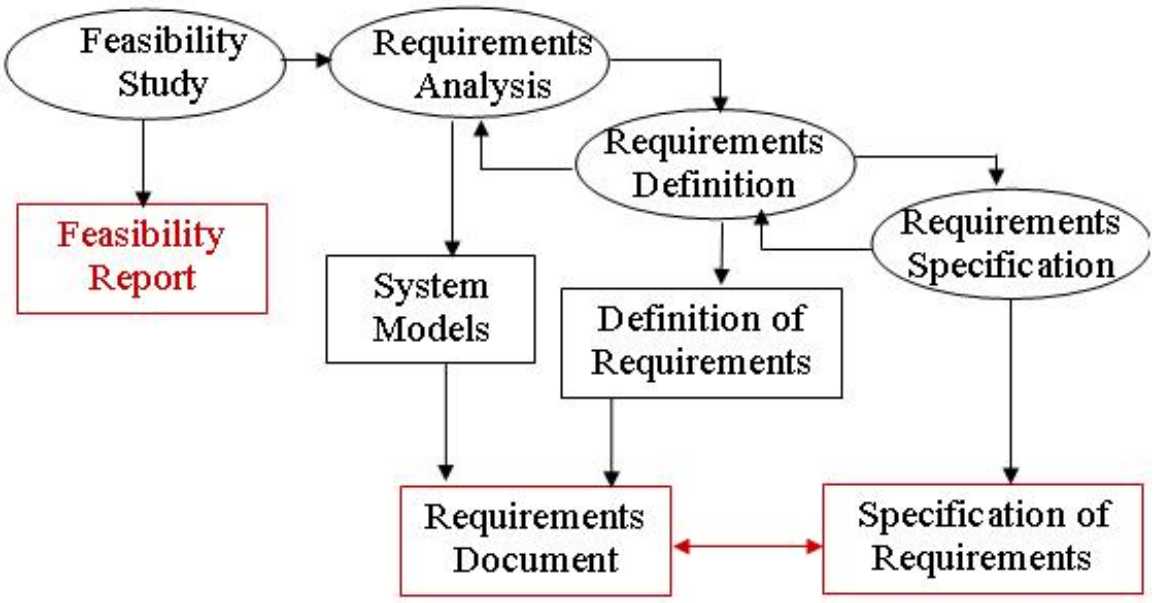


Figure 2.3: Requirements process

2.3.2 Software Requirements Fundamentals

2.3.2.1 Definition of a Software Requirement

At its most basic, a software requirement is a property which must be exhibited in order to solve some problem in the real world. This session refers to requirements on "software" because it is concerned with problems to be addressed by software. Hence, a software requirement is a property which must be exhibited by software developed or adapted to solve a particular problem. The problem may be to automate part of a task of someone who will use the software, to support the business processes of the organization that has commissioned the software, to correct shortcomings of existing software, to control a device, and many more. The functioning of users, business processes, and devices is typically complex. By extension, therefore, the requirements on particular software are typically a complex combination of requirements from different people at different levels of an organization and from the environment in which the software will operate.

An essential property of all software requirements is that they be verifiable. It may be difficult or costly to verify certain software requirements. For example, verification of the throughput requirement on the call center may necessitate the development of simulation software. Both the software requirements and software quality personnel must ensure that the requirements can be verified within the available resource constraints.

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

2.3.2.2 Product and Process Requirements

A distinction can be drawn between product parameters and process parameters. Product parameters are requirements on software to be developed (for example, "The software shall verify that a student meets all

prerequisites before he or she registers for a course.").

A process parameter is essentially a constraint on the development of the software (for example, "The software shall be written in Ada."). These are sometimes known as process requirements.

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

2.3.2.3 Functional and Non-functional Requirements

Functional requirements describe the functions that the software is to execute; for example, formatting some text or modulating a signal. They are sometimes known as capabilities or statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.

Nonfunctional requirements are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as constraints or quality requirements.

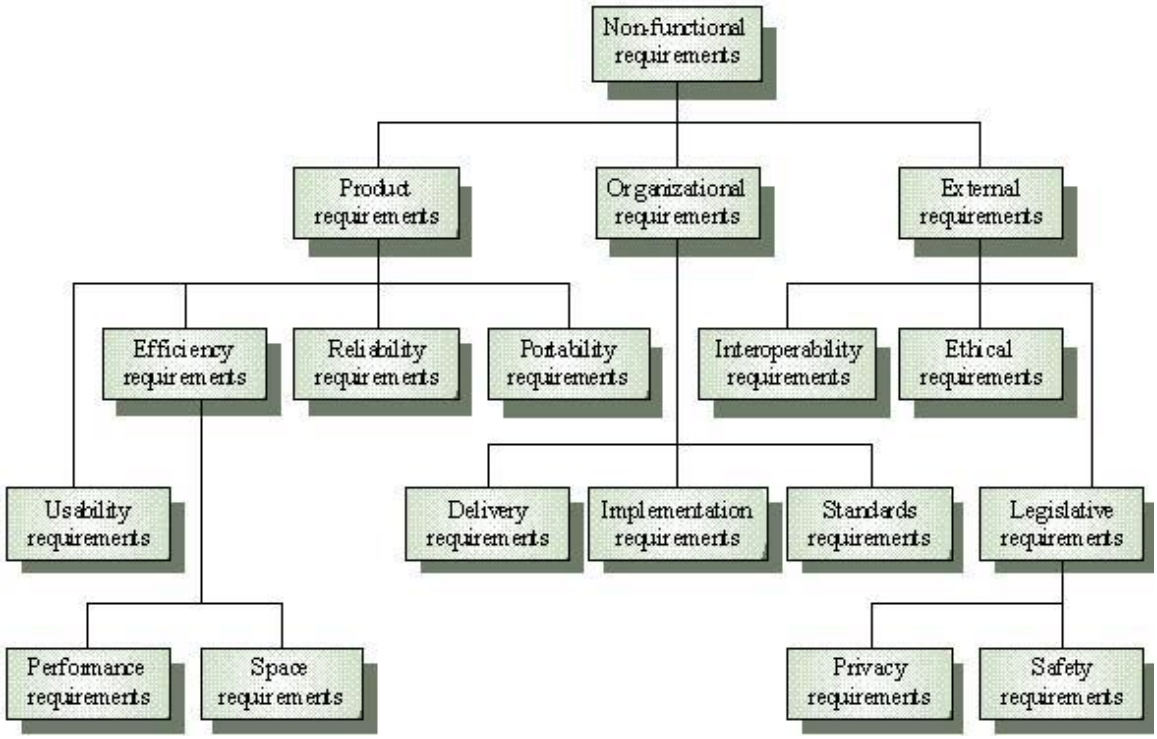


Figure 2.4: Nonfunctional Requirements

They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, or one of many other types of software requirements.

2.3.2.4 Emergent Properties

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

2.3.2.5 Quantifiable Requirements

Software requirements should be stated as clearly and as unambiguously as possible, and, where appropriate, quantitatively. It is important to avoid vague and unverifiable requirements which depend for their interpretation on subjective judgment ("the software shall be reliable"; "the software shall be user-friendly"). This is particularly important for nonfunctional requirements. Two examples of quantified requirements are the following: a call center's software must increase the center's throughput by 20%; and a system shall have a probability of generating a fatal error during any hour of operation of less than 1 * 10[U+F02D]8. The throughput requirement is at a very high level and will need to be used to derive a number of detailed requirements. The reliability requirement will tightly constrain the system architecture.

2.3.2.6 System Requirements and Software Requirements

In this topic, system means "an interacting combination of elements to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements," as defined by the International Council on Systems Engineering.

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

The literature on requirements sometimes calls system requirements "user requirements." We can define "user requirements" in a restricted way as the requirements of the system's customers or end-users. System requirements, by contrast, encompass user requirements, requirements of other stakeholders (such as regulatory authorities), and requirements without an identifiable human source.

2.3.3 Requirements Process

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

2.3.3.1 Process Models

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

- Is not a discrete front-end activity of the software life cycle, but rather a process initiated at the beginning of a project and continuing to be refined throughout the life cycle
- Identifies software requirements as configuration items, and manages them using the same software configuration management practices as other products of the software life cycle processes
- Needs to be adapted to the organization and project context

In particular, the topic is concerned with how the activities of elicitation, analysis, specification, and validation are configured for different types of projects and constraints.

2.3.3.2 Process Actors

This topic introduces the roles of the people who participate in the requirements process. This process is fundamentally interdisciplinary, and the requirements specialist needs to mediate between the domain of the stakeholder and that of software engineering. There are often many people involved besides the requirements

specialist, each of whom has a stake in the software. The stakeholders will vary across projects, but always include users/operators and customers (who need not be the same).

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

- Users: This group comprises those who will operate the software. It is often a heterogeneous group comprising people with different roles and requirements.
- Customers: This group comprises those who have commissioned the software or who represent the software's target market.
- Market analysts: A mass-market product will not have a commissioning customer, so marketing people are often needed to establish what the market needs and to act as proxy customers.
- Regulators: Many application domains such as banking and public transport are regulated. Software in these domains must comply with the requirements of the regulatory authorities.
- Software engineers: These individuals have a legitimate interest in profiting from developing the software by, for example, reusing components in other products. If, in this scenario, a customer of a particular product has specific requirements which compromise the potential for component reuse, the software engineers must carefully weigh their own stake against those of the customer.

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

2.3.3.3 Process Support and Management

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

2.3.3.4 Process Quality and Improvement

This topic is concerned with the assessment of the quality and improvement of the requirements process. Its purpose is to emphasize the key role the requirements process plays in terms of the cost and timeliness of a software product, and of the customer's satisfaction with it. It will help to orient the requirements process with quality standards and process improvement models for software and systems. Process quality and improvement is closely related to both the Software Quality KA and the Software Engineering Process KA. Of particular interest are issues of software quality attributes and measurement, and software process definition. This topic covers

- Requirements process coverage by process improvement standards and models
- Requirements process measures and benchmarking
- Improvement planning and implementation

2.3.4 Requirements Elicitation

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

One of the fundamental tenets of good software engineering is that there be good communication between software users and software engineers. Before development begins, requirements specialists may form the conduit for this communication. They must mediate between the domain of the software users (and other stakeholders) and the technical world of the software engineer.

2.3.4.1 Requirements Sources

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

- Goals: The term goal (sometimes called "business concern" or "critical success factor") refers to the overall, high-level objectives of the software. Goals provide the motivation for the software, but are often vaguely formulated. Software engineers need to pay particular attention to assessing the value (relative to priority) and cost of goals. A feasibility study is a relatively low-cost way of doing this.
- Domain knowledge: The software engineer needs to acquire, or have available, knowledge about the application domain. This enables them to infer tacit knowledge that the stakeholders do not articulate, assess the trade-offs that will be necessary between conflicting requirements, and, sometimes, to act as a "user" champion.
- Stakeholders: Much software has proved unsatisfactory because it has stressed the requirements of one group of stakeholders at the expense of those of others. Hence, software is delivered which is difficult to use or which subverts the cultural or political structures of the customer organization. The software engineer needs to identify, represent, and manage the "viewpoints" of many different types of stakeholders.
- The operational environment: Requirements will be derived from the environment in which the software will be executed. These may be, for example, timing constraints in real-time software or interoperability constraints in an office environment. These must be actively sought out, because they can greatly affect software feasibility and cost, and restrict design choices.
- The organizational environment: Software is often required to support a business process, the selection of which may be conditioned by the structure, culture, and internal politics of the organization. The software engineer needs to be sensitive to these, since, in general, new software should not force unplanned change on the business process.

2.3.4.2 Elicitation Techniques

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

- Interviews: a "traditional" means of eliciting requirements. It is important to understand the advantages and limitations of interviews and how they should be conducted.
- Scenarios: a valuable means for providing context to the elicitation of user requirements. They allow the software engineer to provide a framework for questions about user tasks by permitting "what if" and "how is this done" questions to be asked. The most common type of scenario is the use case.
- Prototypes: a valuable tool for clarifying unclear requirements. They can act in a similar way to scenarios by providing users with a context within which they can better understand what information they need to provide. There is a wide range of prototyping techniques, from paper mock-ups of screen designs to beta-test versions of software products, and a strong overlap of their use for requirements elicitation and the use of prototypes for requirements validation.
- Facilitated meetings: The purpose of these is to try to achieve a summative effect whereby a group of people can bring more insight into their software requirements than by working individually. They can brainstorm and refine ideas which may be difficult to bring to the surface using interviews. Another

advantage is that conflicting requirements surface early on in a way that lets the stakeholders recognize where there is conflict. When it works well, this technique may result in a richer and more consistent set of requirements than might otherwise be achievable. However, meetings need to be handled carefully (hence the need for a facilitator) to prevent a situation from occurring where the critical abilities of the team are eroded by group loyalty, or the requirements reflecting the concerns of a few outspoken (and perhaps senior) people are favored to the detriment of others.

- Observation: The importance of software context within the organizational environment has led to the adaptation of observational techniques for requirements elicitation. Software engineers learn about user tasks by immersing themselves in the environment and observing how users interact with their software and with each other. These techniques are relatively expensive, but they are instructive because they illustrate that many user tasks and business processes are too subtle and complex for their actors to describe easily.

2.3.5 Requirements Analysis

This topic is concerned with the process of analyzing requirements to

- Detect and resolve conflicts between requirements
- Discover the bounds of the software and how it must interact with its environment
- Elaborate system requirements to derive software requirements

The traditional view of requirements analysis has been that it be reduced to conceptual modeling using one of anumber of analysis methods such as the Structured Analysis and Design Technique (SADT). 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).

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

2.3.5.1 Requirements Classification

Requirements can be classified on a number of dimensions:

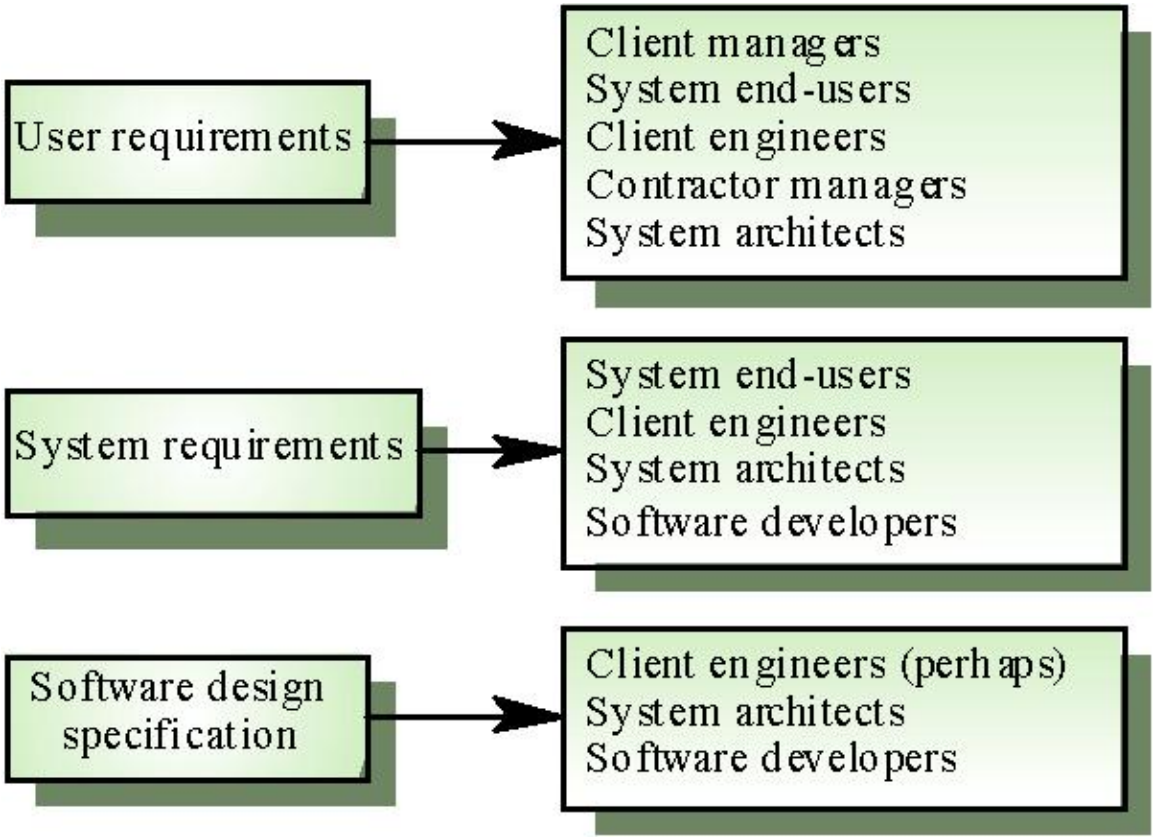


Figure 2.5: Requirement types

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

There is a strong overlap between requirements classification and requirements attributes.

2.3.5.2 Conceptual Modeling

The development of models of a real-world problem is key to software requirements analysis. Their purpose is to aid in understanding the problem, rather than to initiate design of the solution. Hence, conceptual models comprise models of entities from the problem domain configured to reflect their real-world relationships and dependencies.

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

- The nature of the problem. Some types of software demand that certain aspects be analyzed particularly rigorously. For example, control flow and state models are likely to be more important for real-time software than for management information software, while it would usually be the opposite for data models.
- The expertise of the software engineer. It is often more productive to adopt a modeling notation or method with which the software engineer has experience.

- The process requirements of the customer. Customers may impose their favored notation or method, or prohibit any with which they are unfamiliar. This factor can conflict with the previous factor.
- The availability of methods and tools. Notations or methods which are poorly supported by training and tools may not achieve widespread acceptance even if they are suited to particular types of problems.

Note that, in almost all cases, it is useful to start by building a model of the software context. The software context provides a connection between the intended software and its external environment. This is crucial to understanding the software's context in its operational environment and to identifying its interfaces with the environment.

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 which guides the application of the notations. There is little empirical evidence to support claims for the superiority of one notation over another. However, the widespread acceptance of a particular method or notation can lead to beneficial industry-wide pooling of skills and knowledge. This is currently the situation with the UML (Unified Modeling Language).

Formal modeling using notations based on discrete mathematics, and which are traceable 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.

Two standards provide notations which may be useful in performing conceptual modeling–IEEE Std 1320.1, IDEF0 for functional modeling; and IEEE Std 1320.2, IDEF1X97 (IDEFObject) for information modeling.

2.3.5.3 Architectural Design and Requirements Allocation

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

Allocation is important to permit detailed analysis of requirements. Hence, for example, once a set of requirements has been allocated to a component, the individual requirements can be further analyzed to discover further requirements on how the component needs to interact with other components in order to satisfy the allocated requirements. In large projects, allocation stimulates a new round of analysis for each subsystem.

Architectural design is closely identified with conceptual modeling. The mapping from real-world domain entities to software components is not always obvious, so architectural design is identified as a separate topic. The requirements of notations and methods are broadly the same for both conceptual modeling and architectural design.

2.3.5.4 Requirements Negotiation

Another term commonly used for this sub-topic is "conflict resolution." This concerns resolving problems with requirements where conflicts occur between two stakeholders requiring mutually incompatible features, between requirements and resources, or between functional and non-functional requirements. In most cases, it is unwise for the software engineer to make a unilateral decision, and so it becomes 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 be traceable back to the customer. We have classified this as a software requirements analysis topic because problems emerge as the result of analysis. However, a strong case can also be made for considering it a requirements validation topic.

2.3.5.5 Requirements Specification

For most engineering professions, the term "specification" refers to the assignment of numerical values or limits to a product's design goals. Typical physical systems have a relatively small number of such values. Typical software has a large number of requirements, and the emphasis is shared between performing the numerical quantification and managing the complexity of interaction among the large number of requirements. So, in software engineering jargon, "software requirements specification" typically refers to the production of a document, or its electronic equivalent, which can be systematically reviewed, evaluated, and approved.

For complex systems, particularly those involving substantial non-software components, as many as three different types of documents are produced: system definition, system requirements, and software requirements. For simple software products, only the third of these is required.

There are some approaches to requirements specification:

- Natural language
- Structured natural language
- Design description language
- Requirements specification language
- Graphical notation
- Formal specification

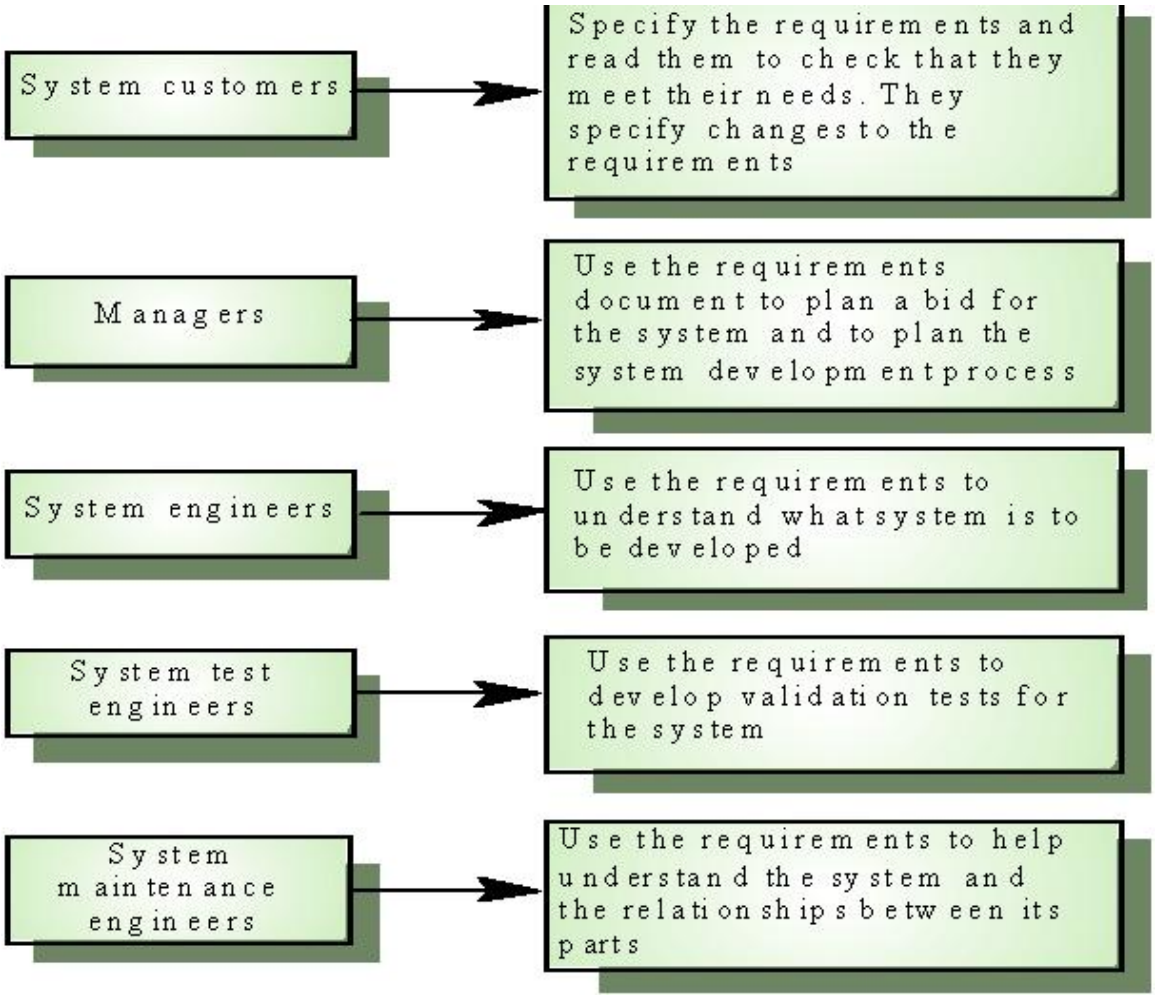


Figure 2.6: Types of requirement document

2.3.5.6 The System Definition Document

This document (sometimes known as the user requirements document or concept of operations) records the system requirements. It defines the high-level system requirements from the domain perspective. Its readership includes representatives of the system users/customers (marketing may play these roles for market-driven software), so its content must be couched in terms of the domain. The document lists the system requirements along with background information about the overall objectives for the system, its target environment and a statement of the constraints, assumptions, and non-functional requirements. It may include conceptual models designed to illustrate the system context, usage scenarios and the principal domain entities, as well as data, information, and workflows. IEEE Std 1362, Concept of Operations Document, provides advice on the preparation and content of such a document. (IEEE1362-98)

2.3.5.7 System Requirements Specification

Developers of systems with substantial software and non-software components, a modern airliner, for example, often separate the description of system requirements from the description of software requirements. In this view, system requirements are specified, the software requirements are derived from the system requirements, and then the requirements for the software components are specified. Strictly speaking, system requirements specification is a systems engineering activity and falls outside the scope of this Guide. IEEE Std 1233 is a guide for developing system requirements.

2.3.5.8 Software Requirements Specification

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

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

Organizations can also use a software requirements specification document to develop their own validation and verification plans more productively.

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

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

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

2.3.6 Requirements validation

The requirements documents may be subject to validation and verification procedures. The requirements may be validated to ensure that the software engineer has understood the requirements, and it is also important to verify that a requirements document conforms to company standards, and that it is understandable, consistent, and complete. Formal notations offer the important advantage of permitting the last two properties to be proven (in a restricted sense, at least). Different stakeholders, including representatives of the customer and developer, should review the document(s). Requirements documents are subject to the same software configuration management practices as the other deliverables of the software life cycle processes.

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

2.3.6.1 Requirements Reviews

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

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

2.3.6.2 Prototyping

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

2.3.6.3 Model Validation

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

2.3.6.4 Acceptance Tests

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

Identifying and designing acceptance tests may be difficult for non-functional requirements. To be validated, they must first be analyzed to the point where they can be expressed quantitatively.

2.3.7 Practical Considerations

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

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

Not every organization has a culture of documenting and managing requirements. It is frequent in dynamic start-up companies, driven by a strong "product vision" and limited resources, to view requirements documentation as an unnecessary overhead. Most often, however, as these companies expand, as their customer base grows, and as their product starts to evolve, they discover that they need to recover the requirements that motivated product features in order to assess the impact of proposed changes. Hence, requirements documentation and change management are key to the success of any requirements process.

2.3.7.1 Iterative Nature of the Requirements Process

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

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

In almost all cases, requirements understanding continues to evolve as design and development proceeds. This often leads to the revision of requirements late in the life cycle. Perhaps the most crucial point in understanding requirements engineering is that a significant proportion of the requirements will change. This is sometimes due to errors in the analysis, but it is frequently an inevitable consequence of change in the "environment": for example, the customer's operating or business environment, or the market into which software must sell. Whatever the cause, it is important to recognize the inevitability of change and take steps to mitigate its effects. Change has to be managed by ensuring that proposed changes go through a defined review and approval process, and, by applying careful requirements tracing, impact analysis, and software configuration management. Hence, the requirements process is not merely a front-end task in software development, but spans the whole software life cycle. In a typical project, the software requirements activities evolve over time from elicitation to change management.

2.3.7.2 Change Management

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

2.3.7.3 Requirements Attributes

Requirements should consist not only of a specification of what is required, but also of ancillary information which helps manage and interpret the requirements. This should include the various classification dimensions of the requirement 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 important requirements attribute, however, is an identifier which allows the requirements to be uniquely and unambiguously identified.

2.3.7.4 Requirements Tracing

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

2.3.7.5 Measuring Requirements

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

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Figure 2.7: Requirement measurements

2.4 Software Design[4]

2.4.1 Introduction

Software design is a process of defining the architecture, components, interfaces, and other characteristics of a system or component and planning for a software solution. After the purpose and specifications of software is determined, software developers will design or employ designers to develop a plan for a solution.

A software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design.

Viewed as a process, software design is the software engineering life cycle activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction. More precisely, a software design (the result) must describe the

[4]This content is available online at <http://cnx.org/content/m28936/1.2/>.

software architecture and the interfaces between those components. It must also describe the components at a level of detail that enable their construction.

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

In a standard listing of software life cycle processes such as IEEE/EIA 12207 Software Life Cycle Processes, software design consists of two activities that fit between software requirements analysis and software construction:

- Software architectural design (sometimes called top-level design): describing software's top-level structure and organization and identifying the various components.
- Software detailed design: describing each component sufficiently to allow for its construction.

2.4.2 Concepts of software design

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

2.4.2.1 General Design Concepts

Software is not the only field where design is involved. In the general sense, we can view design as a form of problem-solving. For example, the concept of a wicked problem–a problem with no definitive solution–is interesting in terms of understanding the limits of design. A number of other notions and concepts are also of interest in understanding design in its general sense: goals, constraints, alternatives, representations, and solutions.

2.4.2.2 Context of Software Design

To understand the role of software design, it is important to understand the context in which it fits, the software engineering life cycle. Thus, it is important to understand the major characteristics of software requirements analysis vs. software design vs. software construction vs. software testing.

2.4.2.3 Software Design Process

Software design is generally considered a two-step process:

2.4.2.3.1 Architectural design

Architectural design describes how software is decomposed and organized into components (the software architecture).

2.4.2.3.2 Detailed design

Detailed design describes the specific behavior of these components. The output of this process is a set of models and artifacts that record the major decisions that have been taken.

2.4.2.4 Enabling Techniques

According to the Oxford English Dictionary, a principle is "a basic truth or a general law ... that is used as a basis of reasoning or a guide to action". Software design principles, also called enabling techniques, are key notions considered fundamental to many different software design approaches and concepts.

2.4.2.4.1 Abstraction

Abstraction is "the process of forgetting information so that things that are different can be treated as if they were the same". In the context of software design, two key abstraction mechanisms are parameterization and specification. Abstraction by specification leads to three major kinds of abstraction: procedural abstraction, data abstraction, and control (iteration) abstraction.

2.4.2.4.2 Coupling and cohesion

Coupling is defined as the strength of the relationships between modules, whereas cohesion is defined by how the elements making up a module are related.

2.4.2.4.3 Decomposition and modularization

Decomposing and modularizing large software into a number of smaller independent ones, usually with the goal of placing different functionalities or responsibilities in different components.

2.4.2.4.4 Encapsulation/information hiding

Encapsulation/information hiding means grouping and packaging the elements and internal details of an abstraction and making those details inaccessible.

2.4.2.4.5 Separation of interface and implementation

Separating interface and implementation involves defining a component by specifying a public interface, known to the clients, separate from the details of how the component is realized.

2.4.2.4.6 Sufficiency, completeness and primitiveness

Achieving sufficiency, completeness, and primitiveness means ensuring that a software component captures all the important characteristics of an abstraction, and nothing more.

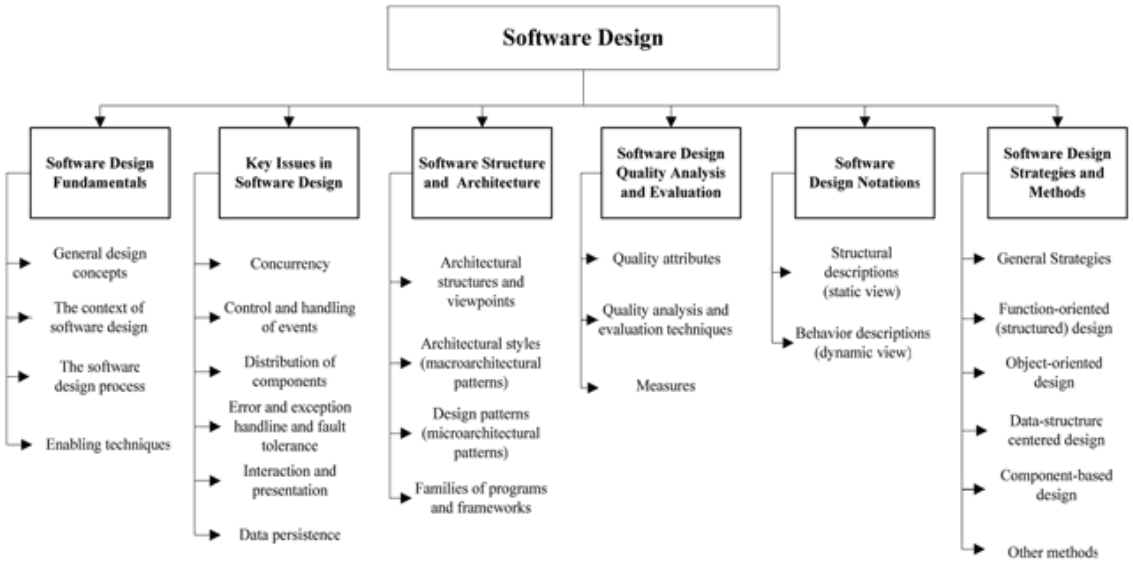


Figure 2.8: Breakdown of topics for the Software Design

2.4.3 Software Structure and Architecture

In its strict sense, a software architecture is "a description of the subsystems and components of a software system and the relationships between them". Architecture thus attempts to define the internal structure -according to the Oxford English Dictionary, "the way in which something is constructed or organized" - of the resulting software. During the mid-1990s, however, software architecture started to emerge as a broader discipline involving the study of software structures and architectures in a more generic way. This gave rise to a number of interesting ideas about software design at different levels of abstraction. Some of these concepts can be useful during the architectural design (for example, architectural style) of specific software, as well as during its detailed design (for example, lower-level design patterns). But they can also be useful for designing generic systems, leading to the design of families of programs (also known as product lines). Interestingly, most of these concepts can be seen as attempts to describe, and thus reuse, generic design knowledge.

2.4.3.1 Architectural Structures and Viewpoints

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

An architectural style is "a set of constraints on an architecture [that] defines a set or family of architectures that satisfies them". An architectural style can thus be seen as a meta-model which can provide

software's high-level organization (its macroarchitecture).

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

2.4.3.2 Design Patterns

Succinctly described, a pattern is "a common solution to a common problem in a given context." While architectural styles can be viewed as patterns describing the high-level organization of software (their macroarchitecture), other design patterns can be used to describe details at a lower, more local level (their microarchitecture).

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

2.4.3.3 Families of Programs and Frameworks

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

In OO programming, a key related notion is that of the framework: a partially complete software subsystem that can be extended by appropriately instantiating specific plug-ins (also known as hot spots).

2.4.3.4 Software Design Quality Analysis and Evaluation

This section includes a number of quality and evaluation topics that are specifically related to software design.

2.4.3.4.1 Quality Attributes

Various attributes are generally considered important for obtaining a software design of good quality - various "ilities" (maintainability, portability, testability, traceability), various "nesses" (correctness, robustness), including "fitness of purpose".

An interesting distinction is the one between quality attributes discernable at run-time (performance, security, availability, functionality, usability), those not discernable at run-time (modifiability, portability, reusability, integrability, and testability), and those related to the architecture's intrinsic qualities (conceptual integrity, correctness, and completeness, buildability).

2.4.3.4.2 Quality Analysis and Evaluation Techniques

Various tools and techniques can help ensure a software design's quality.

- Software design reviews: informal or semiformal, often group-based, techniques to verify and ensure the quality of design artifacts.
- Static analysis: formal or semiformal static (non-executable) analysis that can be used to evaluate a design (for example, fault-tree analysis or automated cross-checking).
- Simulation and prototyping: dynamic techniques to evaluate a design (for example, performance simulation or feasibility prototype.

2.4.3.4.3 Measures

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

- Function-oriented (structured) design measures: the design's structure, obtained mostly through functional decomposition; generally represented as a structure chart (sometimes called a hierarchical diagram) on which various measures can be computed.
- Object-oriented design measures: the design's overall structure is often represented as a class diagram, on which various measures can be computed. Measures on the properties of each class's internal content can also be computed.

2.4.4 Software Design Notations

Many notations and languages exist to represent software design artifacts. Some are used mainly to describe a design's structural organization, others to represent software behavior. Certain notations are used mostly during architectural design and others mainly during detailed design, although some notations can be used in both steps. In addition, some notations are used mostly in the context of specific. Here, they are categorized into notations for describing the structural (static) view vs. the behavioral (dynamic) view.

2.4.4.1 Structural Descriptions (static view)

The following notations, mostly (but not always) graphical, describe and represent the structural aspects of a software design - that is, they describe the major components and how they are interconnected (static view):

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

2.4.4.2 Behavioral Descriptions (dynamic view)

The following notations and languages, some graphical and some textual, are used to describe the dynamic behavior of software and components. Many of these notations are useful mostly, but not exclusively, during detailed design.

- Activity diagrams: used to show the control flow from activity ("ongoing non-atomic execution within a state machine") to activity.

- Collaboration diagrams: used to show the interactions that occur among a group of objects, where the emphasis is on the objects, their links, and the messages they exchange on these links.
- Data flow diagrams (DFDs): used to show data flow among a set of processes.
- Decision tables and diagrams: used to represent complex combinations of conditions and actions.
- Flowcharts and structured flowcharts: used to represent the flow of control and the associated actions to be performed.
- Sequence diagrams: used to show the interactions among a group of objects, with emphasis on the time-ordering of messages.
- State transition and state-chart diagrams: used to show the control flow from state to state in a state machine.
- Formal specification languages: textual languages that use basic notions from mathematics (for example, logic, set, sequence) to rigorously and abstractly define software component interfaces and behavior, often in terms of pre- and post-conditions.
- Pseudocode and program design languages (PDLs): structured-programming-like languages used to describe, generally at the detailed design stage, the behavior of a procedure or method.

2.4.5 Software Design Strategies and Methods

There exist various general strategies to help guide the design process. In contrast with general strategies, methods are more specific in that they generally suggest and provide a set of notations to be used with the method, a description of the process to be used when following the method and a set of guidelines in using the method. Such methods are useful as a means of transferring knowledge and as a common framework for teams of software engineers.

2.4.5.1 General Strategies

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

2.4.5.2 Function-Oriented (Structured) Design

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

2.4.5.3 Object-Oriented Design

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

2.4.5.4 Data-Structure-Centered Design

Data-structure-centered design (for example, Jackson, Warnier-Orr) starts from the data structures a program manipulates rather than from the function it performs. The software engineer first describes the input

and output data structures (using Jackson's structure diagrams, for instance) and then develops the program's control structure based on these data structure diagrams. Various heuristics have been proposed to deal with special cases—for example, when there is a mismatch between the input and output structures.

2.4.5.5 Component-Based Design (CBD)

A software component is an independent unit, having well-defined interfaces and dependencies that can be composed and deployed independently. Component-based design addresses issues related to providing, developing, and integrating such components in order to improve reuse.

2.4.5.6 Other Methods

Other interesting but less mainstream approaches also exist: formal and rigorous methods and transformational methods.

References:

http://en.wikipedia.org/wiki/Software_design, http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-171Fall2003/CourseHome/, http://www.cs.cornell.edu/courses/cs501/2008sp/, http://www.sei.cmu.edu/,http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/, http://www.ee.unb.ca/kengleha/courses/CMPE3213/IntroToSoftwareEng.htm, http://www.developerdotstar.com/mag/articles/reeves_design.html, http://trace.wisc.edu/docs/software_guidelines/softwa etc...

2.5 Software construction[5]

2.5.1 Introduction

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

The software construction is linked to all the other software engineering, most strongly to software design and software testing. This is because the software construction process itself involves significant software design and test activity. It also uses the output of design and provides one of the inputs to testing, both design and testing being the activities. Detailed boundaries between design, construction, and testing (if any) will vary depending upon the software life cycle processes that are used in a project.

[5]This content is available online at <http://cnx.org/content/m28929/1.1/>.

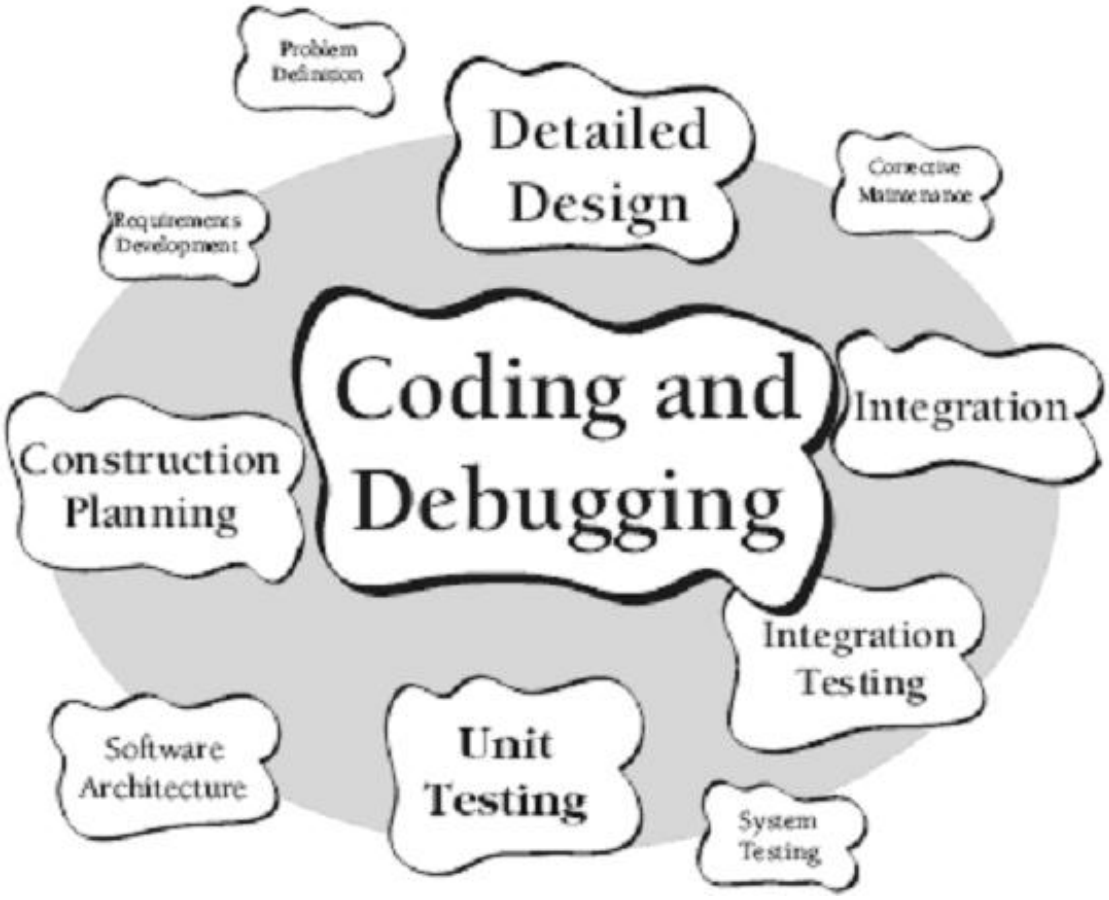


Figure 2.9: Software construction activities

Although some detailed design may be performed prior to construction, much design work is performed within the construction activity itself. Thus the software construction is closely linked to the software design.

Throughout construction, software engineers both unit-test and integration-test their work. Thus, the software construction is closely linked to the software testing as well.

Software construction typically produces the highest volume of configuration items that need to be managed in a software project (source files, content, test cases, and so on). Thus, the software construction is also closely linked to the software configuration management.

2.5.2 Software Construction Fundamentals

The fundamentals of software construction include:

- Minimizing complexity
- Anticipating change
- Constructing for verification
- Standards in construction

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

2.5.2.1 Minimizing Complexity

A major factor in how people convey intent to computers is the severely limited ability of people to hold complex structures and information in their working memories, especially over long periods of time. This leads to one of the strongest drivers in software construction: minimizing complexity. The need to reduce complexity applies to essentially every aspect of software construction, and is particularly critical to the process of verification and testing of software constructions.

In software construction, reduced complexity is achieved through emphasizing the creation of code that is simple and readable rather than clever.

2.5.2.2 Anticipating Change

Most software will change over time, and the anticipation of change drives many aspects of software construction. Software is unavoidably part of changing external environments, and changes in those outside environments affect software in diverse ways.

Anticipating change is supported by many specific techniques:

- Communication methods (for example, standards for document formats and contents)
- Programming languages (for example, language standards for languages like Java and C++)
- Platforms (for example, programmer interface standards for operating system calls)
- Tools (for example, diagrammatic standards for notations like UML (Unified Modeling Language))

2.5.2.3 Constructing for Verification

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

2.5.2.4 Standards in Construction

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

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

2.5.3 Managing Construction

2.5.3.1 Construction Models

Numerous models have been created to develop software, some of which emphasize construction more than others.

Some models are more linear from the construction point of view, such as the waterfall and staged-delivery life cycle models. These models treat construction as an activity which occurs only after significant prerequisite work has been completed - including detailed requirements work, extensive design work, and

detailed planning. The more linear approaches tend to emphasize the activities that precede construction (requirements and design), and tend to create more distinct separations between the activities. In these models, the main emphasis of construction may be coding.

Other models are more iterative, such as evolutionary prototyping, Extreme Programming, and Scrum. These approaches tend to treat construction as an activity that occurs concurrently with other software development activities, including requirements, design, and planning, or overlaps them. These approaches tend to mix design, coding, and testing activities, and they often treat the combination of activities as construction.

Consequently, what is considered to be "construction" depends to some degree on the life cycle model used.

2.5.3.2 Construction Planning

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

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

Construction planning also defines the order in which components are created and integrated, the software quality management processes, the allocation of task assignments to specific software engineers, and the other tasks, according to the chosen method.

2.5.3.3 Construction Measurement

Numerous construction activities and artifacts can be measured, including code developed, code modified, code reused, code destroyed, code complexity, code inspection statistics, fault-fix and fault-find rates, effort, and scheduling. These measurements can be useful for purposes of managing construction, ensuring quality during construction, improving the construction process, as well as for other reasons.

2.5.4 Practical considerations

Construction is an activity in which the software has to come to terms with arbitrary and chaotic real-world constraints, and to do so exactly. Due to its proximity to real-world constraints, construction is more driven by practical considerations than some other KAs, and software engineering is perhaps most craft-like in the construction area.

2.5.4.1 Construction Design

Some projects allocate more design activity to construction; others to a phase explicitly focused on design. Regardless of the exact allocation, some detailed design work will occur at the construction level, and that design work tends to be dictated by immovable constraints imposed by the real-world problem that is being addressed by the software. Just as construction workers building a physical structure must make small-scale modifications to account for unanticipated gaps in the builder's plans, software construction workers must make modifications on a smaller or larger scale to flesh out details of the software design during construction.

2.5.4.2 Construction Languages

Construction languages include all forms of communication by which a human can specify an executable problem solution to a computer.

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

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

Programming languages are the most flexible type of construction languages. They also contain the least amount of information about specific application areas and development processes, and so require the most training and skill to use effectively.

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

- Linguistic
- Formal
- Visual

Linguistic notations 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 providing an immediate intuitive understanding of what will happen when the underlying software construction is executed.

Formal notations rely less on intuitive, everyday meanings of words and text strings and more on definitions backed up by precise, unambiguous, and formal (or mathematical) definitions. Formal construction notations and formal methods are at the heart of most forms of system programming, where accuracy, time behavior, and testability are more important than ease of mapping into natural language. Formal constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions.

Visual notations rely much less on the text-oriented notations of both linguistic and formal construction, and instead rely on direct visual interpretation and placement of visual entities that represent the underlying software. Visual construction tends to be somewhat limited by the difficulty of making "complex" statements using only movement of visual entities on a display. However, it can also be a powerful tool in cases where the primary programming task is simply to build and "adjust" a visual interface to a program, the detailed behavior of which has been defined earlier.

2.5.4.3 Object-Oriented Languages

There are almost two dozen major object-oriented programming languages in use today. But the leading commercial o-o languages are far fewer in number. These are:

- C++
- Smalltalk
- Java

2.5.4.3.1 C++

C++ is an object-oriented version of C. It is compatible with C (it is actually a superset), so that existing C code can be incorporated into C++ programs. C++ programs are fast and efficient, qualities which helped make C an extremely popular programming language. It sacrifices some flexibility in order to remain efficient, however. C++ uses compile-time binding, which means that the programmer must specify the specific class of an object, or at the very least, the most general class that an object can belong to. This makes for high run-time efficiency and small code size, but it trades off some of the power to reuse classes.

C++ has become so popular that most new C compilers are actually C/C++ compilers. However, to take full advantage of object-oriented programming, one must program (and think!) in C++, not C. This can often be a major problem for experienced C programmers. Many programmers think they are coding in C++, but instead are only using a small part of the language's object-oriented power.

2.5.4.3.2 Smalltalk

Smalltalk is a pure object-oriented language. While C++ makes some practical compromises to ensure fast execution and small code size, Smalltalk makes none. It uses run-time binding, which means that nothing about the type of an object need be known before a Smalltalk program is run.

Smalltalk programs are considered by most to be significantly faster to develop than C++ programs. A rich class library that can be easily reused via inheritance is one reason for this. Another reason is Smalltalk's dynamic development environment. It is not explicitly compiled, like C++. This makes the development process more fluid, so that "what if" scenarios can be easily tried out, and classes definitions easily refined. But being purely object-oriented, programmers cannot simply put their toes in the o-o waters, as with C++. For this reason, Smalltalk generally takes longer to master than C++. But most of this time is actually spent learning object-oriented methodology and techniques, rather than details of a particular programming language. In fact, Smalltalk is syntactically very simple, much more so than either C or C++.

Unlike C++, which has become standardized, The Smalltalk language differs somewhat from one implementation to another. The most popular commercial "dialects" of Smalltalk are:

- VisualWorks from ParcPlace-Digitalk, Inc.
- Smalltalk/V and Visual Smalltalk from ParcPlace-Digitalk Inc.
- VisualAge from IBM

1. VisualWorks

VisualWorks is arguably the most powerful of Smalltalks. VisualWorks was developed by ParcPlace, which grew out of the original Xerox PARC project that invented the Smalltalk language. VisualWorks is platform-independent, so that an application written under one operating system, say, Microsoft Windows, can work without any modification on any of a wide range of platform supported by ParcPlace, from Sun Solaris to Macintosh. VisualWorks also features a GUI (Graphic User Interface) builder that is well-integrated into the product.

2. Smalltalk/V and Visual Smalltalk

Digitalk's versions of Smalltalk are somewhat smaller and simpler, and are specifically tailored to IBM compatible PCs. A Macintosh version was available, but support has since been abandoned. This does not bode well for Digitalk cross-platform efforts. Digitalk has a separate GUI builder, called PARTS Workbench (bundled with Visual Smalltalk), which allows quick construct of an application.

ParcPlace and Digitalk were merged into a single company, ParcPlace-Digitalk, Inc. The future of the Digitalk product line is uncertain, and it may just be spun off back into a separate company.

3. VisualAge

IBM's version of Smalltalk, VisualAge, is comparable to Smalltalk/V with PARTS. Both of these Smalltalks allow programmers to readily exploit machine-specific features, at the expense of some portability. IBM has adapted existing industry standards for such things as file management and screen graphics. When IBM talks, people listen, and IBM has made a substantial commitment to Smalltalk.

2.5.4.3.3 Java

Java is the latest, flashiest object-oriented language. It has taken the software world by storm due to its close ties with the Internet and Web browsers. It is designed as a portable language that can run on any web-enabled computer via that computer's Web browser. As such, it offers great promise as the standard Internet and Intranet programming language.

Java is a curious mixture of C++ and Smalltalk. It has the syntax of C++, making it easy (or difficult) to learn, depending on your experience. But it has improved on C++ in some important areas. For one thing,

it has no pointers, low-level programming constructs that make for error-prone programs. Like Smalltalk, it has garbage collection, a feature that frees the programmer from explicitly allocating and de-allocating memory. And it runs on a Smalltalk-style virtual machine, software built into your web browser which executes the same standard compiled Java bytecodes no matter what type of computer you have.

Java development tools are being rapidly deployed, and are available from such major software companies as IBM, Microsoft, and Symantec.

2.5.4.4 Coding

The following considerations apply to the software construction coding activity:

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

2.5.4.5 Construction Testing

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

- Unit testing
- Integration testing

The purpose of construction testing is to reduce the gap between the time at which faults are inserted into the code and the time those faults are detected. In some cases, construction testing is performed after code has been written. In other cases, test cases may be created before code is written.

Construction testing typically involves a subset of types of testing. For instance, construction testing does not typically include system testing, alpha testing, beta testing, stress testing, configuration testing, usability testing, or other, more specialized kinds of testing.

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

2.5.4.6 Reuse

Implementing software reuse entails more than creating and using libraries of assets. It requires formalizing the practice of reuse by integrating reuse processes and activities into the software life cycle. However, reuse is important enough in software construction that it is included here as a topic.

The tasks related to reuse in software construction during coding and testing are:

- The selection of the reusable units, databases, test procedures, or test data
- The evaluation of code or test reusability
- The reporting of reuse information on new code, test procedures, or test data

2.5.4.7 Construction Quality

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

- Unit testing and integration testing
- Test-first development
- Code stepping
- Use of assertions
- Debugging
- Technical reviews
- Static analysis (IEEE1028)

The specific technique or techniques selected depend on the nature of the software being constructed, as well as on the skills set of the software engineers performing the construction.

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

2.5.4.8 Integration

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

Concerns related to construction integration include planning the sequence in which components will be integrated, creating scaffolding to support interim versions of the software, determining the degree of testing and quality work performed on components before they are integrated, and determining points in the project at which interim versions of the software are tested.

References:

http://en.wikipedia.org/wiki/Software_development, http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-171Fall2003/CourseHome/, http://www.cs.cornell.edu/courses/cs501/2008sp/, http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/, http://www.ee.unb.ca/kengleha/courses/CMPE3213/IntroT http://www.generaldigital.com/software_services/qualifications.htm, etc...

2.6 Software Testing[6]

2.6.1 Introduction

Software Testing is the process of executing a program or system with the intent of finding errors. Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion, wear-and-tear, generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects, or bugs, will be buried in and remain latent until activation.

Software bugs will almost always exist in any software module with moderate size: not because programmers are careless or irresponsible, but because the complexity of software is generally intractable, and

[6] This content is available online at <http://cnx.org/content/m28939/1.1/>.

humans have only limited ability to manage complexity. It is also true that for any complex systems, design defects can never be completely ruled out.

Discovering the design defects in software, is equally difficult, for the same reason of complexity. Because software and any digital systems are not continuous, testing boundary values are not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible. Exhaustively testing a simple program to add only two integer inputs of 32-bits (yielding 2^64 distinct test cases) would take hundreds of years, even if tests were performed at a rate of thousands per second. Obviously, for a realistic software module, the complexity can be far beyond the example mentioned here. If inputs from the real world are involved, the problem will get worse, because timing and unpredictable environmental effects and human interactions are all possible input parameters under consideration.

A further complication has to do with the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, the software may now work for a test case that it didn't work for previously. But its behavior on pre-error test cases that it passed before can no longer be guaranteed. To account for this possibility, testing should be restarted. The expense of doing this is often prohibitive.

An interesting analogy parallels the difficulty in software testing with the pesticide (known as the Pesticide Paradox): Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual. But this alone will not guarantee to make the software better, because the Complexity Barrier principle states: Software complexity (and therefore that of bugs) grows to the limits of our ability to manage that complexity. By eliminating the (previous) easy bugs you allowed another escalation of features and complexity, but his time you have subtler bugs to face, just to retain the reliability you had before. Society seems to be unwilling to limit complexity because we all want that extra bell, whistle, and feature interaction. Thus, our users always push us to the complexity barrier and how close we can approach that barrier is largely determined by the strength of the techniques we can wield against ever more complex and subtle bugs.

Regardless of the limitations, testing is an integral part in software development. It is broadly deployed in every phase in the software development cycle. Typically, more than 50% percent of the development time is spent in testing. Testing is usually performed for the following purposes:

2.6.1.1 To improve quality

As computers and software are used in critical applications, the outcome of a bug can be severe. Bugs can cause huge losses. Bugs in critical systems have caused airplane crashes, allowed space shuttle missions to go awry, halted trading on the stock market, and worse. Bugs can kill. Bugs can cause disasters. In a computerized embedded world, the quality and reliability of software is a matter of life and death.

Quality means the conformance to the specified design requirement. Being correct, the minimum requirement of quality, means performing as required under specified circumstances. Debugging, a narrow view of software testing, is performed heavily to find out design defects by the programmer. The imperfection of human nature makes it almost impossible to make a moderately complex program correct the first time. Finding the problems and get them fixed, is the purpose of debugging in programming phase.

2.6.1.2 For Verification & Validation (V&V)

An important purpose of testing is verification and validation. Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work. We can also compare the quality among different products under the same specification, based on results from the same test.

We can not test quality directly, but we can test related factors to make quality visible. Quality has three sets of factors: functionality, engineering, and adaptability. These three sets of factors can be thought of as dimensions in the software quality space. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail. Table 1 illustrates some of the most frequently cited quality considerations.

Functionality(exterior quality)	Engineering(interior quality)	Adaptability(future quality)
Correctness	Efficiency	Flexibility
Reliability	Testability	Reusability
Usability	Documentation	Maintainability
Integrity	Structure	
Table 1. Typical Software Quality Factors		

Table 2.1

Good testing provides measures for all relevant factors. The importance of any particular factor varies from application to application. Any system where human lives are at stake must place extreme emphasis on reliability and integrity. In the typical business system usability and maintainability are the key factors, while for a one-time scientific program neither may be significant. Our testing, to be fully effective, must be geared to measuring each relevant factor and thus forcing quality to become tangible and visible.

Tests with the purpose of validating the product works are named clean tests, or positive tests. The drawbacks are that it can only validate that the software works for the specified test cases. A finite number of tests can not validate that the software works for all situations. On the contrary, only one failed test is sufficient enough to show that the software does not work. Dirty tests, or negative tests, refers to the tests aiming at breaking the software, or showing that it does not work. A piece of software must have sufficient exception handling capabilities to survive a significant level of dirty tests.

A testable design is a design that can be easily validated, falsified and maintained. Because testing is a rigorous effort and requires significant time and cost, design for testability is also an important design rule for software development.

2.6.1.3 For reliability estimation

Software reliability has important relations with many aspects of software, including the structure, and the amount of testing it has been subjected to. Based on an operational profile (an estimate of the relative frequency of use of various inputs to the program), testing can serve as a statistical sampling method to gain failure data for reliability estimation.

Software testing is not mature. It still remains an art, because we still cannot make it a science. We are still using the same testing techniques invented 20-30 years ago, some of which are crafted methods or heuristics rather than good engineering methods. Software testing can be costly, but not testing software is even more expensive, especially in places that human lives are at stake. Solving the software-testing problem is no easier than solving the Turing halting problem. We can never be sure that a piece of software is correct. We can never be sure that the specifications are correct. No verification system can verify every correct program. We can never be certain that a verification system is correct either.

2.6.2 Test Levels

2.6.2.1 The target of the test

Software testing is usually performed at different levels along the development and maintenance processes. That is to say, the target of the test can vary: a single module, a group of such modules (related by purpose, use, behavior, or structure), or a whole system. Three big test stages can be conceptually distinguished, namely Unit, Integration, and System. No process model is implied, nor are any of those three stages assumed to have greater importance than the other two.

2.6.2.1.1 Unit testing

Unit testing verifies the functioning in isolation of software pieces which are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of tightly related units. A test unit is defined more precisely in the IEEE Standard for Software Unit Testing (IEEE1008-87), which also describes an integrated approach to systematic and documented unit testing. Typically, unit testing occurs with access to the code being tested and with the support of debugging tools, and might involve the programmers who wrote the code.

2.6.2.1.2 Integration testing

Integration testing is the process of verifying the interaction between software components. Classical integration testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured software.

Modern systematic integration strategies are rather architecture-driven, which implies integrating the software components or subsystems based on identified functional threads. Integration testing is a continuous activity, at each stage of which software engineers must abstract away lower-level perspectives and concentrate on the perspectives of the level they are integrating. Except for small, simple software, systematic, incremental integration testing strategies are usually preferred to putting all the components together at once, which is pictorially called "big bang" testing.

2.6.2.1.3 System testing

System testing is concerned with the behavior of a whole system. The majority of functional failures should already have been identified during unit and integration testing. System testing is usually considered appropriate for comparing the system to the non-functional system requirements, such as security, speed, accuracy, and reliability. External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level.

2.6.2.2 Objectives of Testing

Testing is conducted in view of a specific objective, which is stated more or less explicitly, and with varying degrees of precision. Stating the objective in precise, quantitative terms allows control to be established over the test process.

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

Other important objectives for testing include (but are not limited to) reliability measurement, usability evaluation, and acceptance, for which different approaches would be taken. Note that the test objective varies with the test target; in general, different purposes being addressed at a different level of testing.

References recommended above for this topic describe the set of potential test objectives. The sub-topics listed below are those most often cited in the literature. Note that some kinds of testing are more appropriate for custom-made software packages, installation testing, for example; and others for generic products, like beta testing.

2.6.2.2.1 Qualification testing

Qualification testing checks the system behavior against the customer's requirements, however these may have been expressed; the customers undertake, or specify, typical tasks to check that their requirements have been met or that the organization has identified these for the target market for the software. This testing activity may or may not involve the developers of the system.

2.6.2.2.2 Installation testing

Usually after completion of software and acceptance testing, the software can be verified upon installation in the target environment. Installation testing can be viewed as system testing conducted once again according to hardware configuration requirements. Installation procedures may also be verified.

2.6.2.2.3 Alpha and beta testing

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

2.6.2.2.4 Reliability achievement and evaluation

In helping to identify faults, testing is a means to improve reliability. By contrast, by randomly generating test cases according to the operational profile, statistical measures of reliability can be derived. Using reliability growth models, both objectives can be pursued together.

Software reliability refers to the probability of failure-free operation of a system. It is related to many aspects of software, including the testing process. Directly estimating software reliability by quantifying its related factors can be difficult. Testing is an effective sampling method to measure software reliability. Guided by the operational profile, software testing (usually black-box testing) can be used to obtain failure data, and an estimation model can be further used to analyze the data to estimate the present reliability and predict future reliability. Therefore, based on the estimation, the developers can decide whether to release the software, and the users can decide whether to adopt and use the software. Risk of using software can also be assessed based on reliability information. The primary goal of testing should be to measure the dependability of tested software.

There is agreement on the intuitive meaning of dependable software: it does not fail in unexpected or catastrophic ways. Robustness testing and stress testing are variances of reliability testing based on this simple criterion.

The robustness of a software component is the degree to which it can function correctly in the presence of exceptional inputs or stressful environmental conditions. Robustness testing differs with correctness testing in the sense that the functional correctness of the software is not of concern. It only watches for robustness problems such as machine crashes, process hangs or abnormal termination. The oracle is relatively simple, therefore robustness testing can be made more portable and scalable than correctness testing. This research has drawn more and more interests recently, most of which uses commercial operating systems as their target.

Stress testing, or load testing, is often used to test the whole system rather than the software alone. In such tests the software or system are exercised with or beyond the specified limits. Typical stress includes resource exhaustion, bursts of activities, and sustained high loads.

2.6.2.2.5 Regression testing

According to (IEEE610.12-90), regression testing is the "selective retesting of a system or component to verify that modifications have not caused unintended effects..." In practice, the idea is to show that software which previously passed the tests still does. Beizer defines it as any repetition of tests intended to show that the software's behavior is unchanged, except insofar as required. Obviously a trade-off must be made between the assurance given by regression testing every time a change is made and the resources required to do that.

Regression testing can be conducted at each of the test levels. The target of the test and may apply to functional and nonfunctional testing.

2.6.2.2.6 Correctness testing

Correctness is the minimum requirement of software, the essential purpose of testing. Correctness testing will need some type of oracle, to tell the right behavior from the wrong one. The tester may or may not know the inside details of the software module under test, e.g. control flow, data flow, etc. Therefore, either a white-box point of view or black-box point of view can be taken in testing software. We must note that the black-box and white-box ideas are not limited in correctness testing only.

2.6.2.2.6.1 Black-box testing

The black-box approach is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure. It is also termed data-driven, input/output driven, or requirements-based testing. Because only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing - a testing method emphasized on executing the functions and examination of their input and output data. The tester treats the software under test as a black box - only the inputs, outputs and specification are visible, and the functionality is determined by observing the outputs to corresponding inputs. In testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. All test cases are derived from the specification. No implementation details of the code are considered.

It is obvious that the more we have covered in the input space, the more problems we will find and therefore we will be more confident about the quality of the software. Ideally we would be tempted to exhaustively test the input space. But as stated above, exhaustively testing the combinations of valid inputs will be impossible for most of the programs, let alone considering invalid inputs, timing, sequence, and resource variables. Combinatorial explosion is the major roadblock in functional testing. To make things worse, we can never be sure whether the specification is either correct or complete. Due to limitations of the language used in the specifications (usually natural language), ambiguity is often inevitable. Even if we use some type of formal or restricted language, we may still fail to write down all the possible cases in the specification. Sometimes, the specification itself becomes an intractable problem: it is not possible to specify precisely every situation that can be encountered using limited words. And people can seldom specify clearly what they want - they usually can tell whether a prototype is, or is not, what they want after they have been finished. Specification problems contributes approximately 30 percent of all bugs in software.

The research in black-box testing mainly focuses on how to maximize the effectiveness of testing with minimum cost, usually the number of test cases. It is not possible to exhaust the input space, but it is possible to exhaustively test a subset of the input space. Partitioning is one of the common techniques. If we have partitioned the input space and assume all the input values in a partition is equivalent, then we only need to test one representative value in each partition to sufficiently cover the whole input space. Domain testing partitions the input domain into regions, and consider the input values in each domain an equivalent class. Domains can be exhaustively tested and covered by selecting a representative value(s) in each domain. Boundary values are of special interest. Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not. Boundary value analysis requires one or more boundary values selected as representative test cases. The difficulties with domain testing are that incorrect domain definitions in the specification can not be efficiently discovered.

Good partitioning requires knowledge of the software structure. A good testing plan will not only contain black-box testing, but also white-box approaches, and combinations of the two.

2.6.2.2.6.2 White-box testing

Contrary to black-box testing, software is viewed as a white-box, or glass-box in white-box testing, as the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. White-box testing is also called glass-box testing, logic-driven testing or design-based testing.

There are many techniques available in white-box testing, because the problem of intractability is eased by specific knowledge and attention on the structure of the software under test. The intention of exhausting some aspect of the software is still strong in white-box testing, and some degree of exhaustion can be achieved, such as executing each line of code at least once (statement coverage), traverse every branch statements (branch coverage), or cover all the possible combinations of true and false condition predicates (Multiple condition coverage).

Control-flow testing, loop testing, and data-flow testing, all maps the corresponding flow structure of the software into a directed graph. Test cases are carefully selected based on the criterion that all the nodes or paths are covered or traversed at least once. By doing so we may discover unnecessary "dead" code - code that is of no use, or never get executed at all, which can not be discovered by functional testing.

In mutation testing, the original program code is perturbed and many mutated programs are created, each contains one fault. Each faulty version of the program is called a mutant. Test data are selected based on the effectiveness of failing the mutants. The more mutants a test case can kill, the better the test case is considered. The problem with mutation testing is that it is too computationally expensive to use. The boundary between black-box approach and white-box approach is not clear-cut. Many testing strategies mentioned above, may not be safely classified into black-box testing or white-box testing. It is also true for transaction-flow testing, syntax testing, finite-state testing, and many other testing strategies not discussed in this text. One reason is that all the above techniques will need some knowledge of the specification of the software under test. Another reason is that the idea of specification itself is broad - it may contain any requirement including the structure, programming language, and programming style as part of the specification content.

We may be reluctant to consider random testing as a testing technique. The test case selection is simple and straightforward: they are randomly chosen. Random testing is more cost effective for many programs. Some very subtle errors can be discovered with low cost. And it is also not inferior in coverage than other carefully designed testing techniques. One can also obtain reliability estimate using random testing results based on operational profiles. Effectively combining random testing with other testing techniques may yield more powerful and cost-effective testing strategies.

2.6.2.2.7 Performance testing

Not all software systems have specifications on performance explicitly. But every system will have implicit performance requirements. The software should not take infinite time or infinite resource to execute. "Performance bugs" sometimes are used to refer to those design problems in software that cause the system performance to degrade.

Performance has always been a great concern and a driving force of computer evolution. Performance evaluation of a software system usually includes: resource usage, throughput, stimulus-response time and queue lengths detailing the average or maximum number of tasks waiting to be serviced by selected resources. Typical resources that need to be considered include network bandwidth requirements, CPU cycles, disk space, disk access operations, and memory usage. The goal of performance testing can be performance bottleneck identification, performance comparison and evaluation, etc. The typical method of doing performance testing is using a benchmark - a program, workload or trace designed to be representative of the typical system usage.

2.6.2.2.8 Security testing

Software quality, reliability and security are tightly coupled. Flaws in software can be exploited by intruders to open security holes. With the development of the Internet, software security problems are becoming even more severe.

Many critical software applications and services have integrated security measures against malicious attacks. The purpose of security testing of these systems include identifying and removing software flaws that may potentially lead to security violations, and validating the effectiveness of security measures. Simulated security attacks can be performed to find vulnerabilities.

2.6.2.2.9 Testing automation

Software testing can be very costly. Automation is a good way to cut down time and cost. Software testing tools and techniques usually suffer from a lack of generic applicability and scalability. The reason is straight-forward. In order to automate the process, we have to have some ways to generate oracles from the specification, and generate test cases to test the target software against the oracles to decide their correctness. Today we still don't have a full-scale system that has achieved this goal. In general, significant amount of human intervention is still needed in testing. The degree of automation remains at the automated test script level.

The problem is lessened in reliability testing and performance testing. In robustness testing, the simple specification and oracle: doesn't crash, doesn't hang suffices. Similar simple metrics can also be used in stress testing.

2.6.2.2.10 When to stop testing?

Testing is potentially endless. We can not test till all the defects are unearthed and removed - it is simply impossible. At some point, we have to stop testing and ship the software. The question is when.

Realistically, testing is a trade-off between budget, time and quality. It is driven by profit models. The pessimistic, and unfortunately most often used approach is to stop testing whenever some, or any of the allocated resources - time, budget, or test cases - are exhausted. The optimistic stopping rule is to stop testing when either reliability meets the requirement, or the benefit from continuing testing cannot justify the testing cost. This will usually require the use of reliability models to evaluate and predict reliability of the software under test. Each evaluation requires repeated running of the following cycle: failure data gathering - modeling - prediction. This method does not fit well for ultra-dependable systems, however, because the real field failure data will take too long to accumulate.

2.6.2.2.11 Alternatives to testing

Software testing is more and more considered a problematic method toward better quality. Using testing to locate and correct software defects can be an endless process. Bugs cannot be completely ruled out. Just as the complexity barrier indicates: chances are testing and fixing problems may not necessarily improve the quality and reliability of the software. Sometimes fixing a problem may introduce much more severe problems into the system, happened after bug fixes.

Using formal methods to "prove" the correctness of software is also an attracting research direction. But this method can not surmount the complexity barrier either. For relatively simple software, this method works well. It does not scale well to those complex, full-fledged large software systems, which are more error-prone.

In a broader view, we may start to question the utmost purpose of testing. Why do we need more effective testing methods anyway, since finding defects and removing them does not necessarily lead to better quality. An analogy of the problem is like the car manufacturing process. In the craftsmanship epoch, we make cars and hack away the problems and defects. But such methods were washed away by the tide of pipelined manufacturing and good quality engineering process, which makes the car defect-free in the manufacturing phase. This indicates that engineering the design process (such as clean-room software engineering) to make the product have less defects may be more effective than engineering the testing process. Testing is used solely for quality monitoring and management, or, "design for testability". This is the leap for software from craftsmanship to engineering.

2.6.3 Test Techniques

One of the aims of testing is to reveal as much potential for failure as possible, and many techniques have been developed to do this, which attempt to "break" the program, by running one or more tests drawn from identified classes of executions deemed equivalent. The leading principle underlying such techniques is to be

as systematic as possible in identifying a representative set of program behaviors; for instance, considering subclasses of the input domain, scenarios, states, and dataflow.

It is difficult to find a homogeneous basis for classifying all techniques, and the one used here must be seen as a compromise. The classification is based on how tests are generated from the software engineer's intuition and experience, the specifications, the code structure, the (real or artificial) faults to be discovered, the field usage, or, finally, the nature of the application. Sometimes these techniques are classified as white-box, also called glassbox, if the tests rely on information about how the software has been designed or coded, or as black-box if the test cases rely only on the input/output behavior. One last category deals with combined use of two or more techniques. Obviously, these techniques are not used equally often by all practitioners. Included in the list are those that a software engineer should know.

2.6.3.1 Based on the software engineer's intuition and experience

2.6.3.1.1 Ad hoc testing

Perhaps the most widely practiced technique remains ad hoc testing: tests are derived relying on the software engineer's skill, intuition, and experience with similar programs. Ad hoc testing might be useful for identifying special tests, those not easily captured by formalized techniques.

2.6.3.1.2 Exploratory testing

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

2.6.3.2 Specification-based techniques

2.6.3.2.1 Equivalence partitioning

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 only one) is taken from each class.

2.6.3.2.2 Boundary-value analysis

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

2.6.3.2.3 Decision table

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 technique is cause-effect graphing.

2.6.3.2.4 Finite-state machine-based

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

2.6.3.2.5 Testing from formal specifications

Giving the specifications in a formal language allows for automatic derivation of functional test cases, and, at the same time, provides a reference output, an oracle, for checking test results. Methods exist for deriving test cases from model-based or algebraic specifications.

2.6.3.2.6 Random testing

Tests are generated purely at random, not to be confused with statistical testing from the operational profile. This form of testing falls under the heading of the specification-based entry, since at least the input domain must be known, to be able to pick random points within it.

2.6.3.3 Code-based techniques

2.6.3.3.1 Control-flow-based criteria

Control-flow-based coverage criteria is aimed at covering all the statements or blocks of statements in a program, or specified combinations of them. Several coverage criteria have been proposed, like condition/decision coverage. The strongest of the control-flow-based criteria is path testing, which aims to execute all entry-to-exit control flow paths in the flowgraph. Since path testing is generally not feasible because of loops, other less stringent criteria tend to be used in practice, such as statement testing, branch testing, and condition/decision testing. The adequacy of such tests is measured in percentages; for example, when all branches have been executed at least once by the tests, 100% branch coverage is said to have been achieved.

2.6.3.3.2 Data flow-based criteria

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

2.6.3.3.3 Reference models for code-based testing

Although not a technique in itself, the control structure of a program is graphically represented using a flowgraph in code-based testing techniques. A flowgraph is a directed graph the nodes and arcs of which correspond to program elements. For instance, nodes may represent statements or uninterrupted sequences of statements, and arcs the transfer of control between nodes.

2.6.3.4 Fault-based techniques

With different degrees of formalization, fault-based testing techniques devise test cases specifically aimed at revealing categories of likely or predefined faults.

2.6.3.4.1 Error guessing

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

2.6.3.4.2 Mutation testing

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: if a test case is successful in identifying the difference between the program and a mutant, the latter is said to be "killed." Originally conceived as a technique to evaluate a test set, mutation testing is also a testing criterion in itself: either tests are randomly generated until enough mutants have been killed, or tests are specifically designed to kill surviving mutants. In the latter case, mutation testing can also be categorized as a code-based technique. The underlying assumption of mutation testing, the coupling effect, is that by looking for simple syntactic faults, more complex but real faults will be found. For the technique to be effective, a large number of mutants must be automatically derived in a systematic way.

2.6.3.5 Usage-based techniques

2.6.3.5.1 Operational profile

In testing for reliability evaluation, the test environment must reproduce the operational environment of the software as closely as possible. The idea is to infer, from the observed test results, the future reliability of the software when in actual use. To do this, inputs are assigned a probability distribution, or profile, according to their occurrence in actual operation.

2.6.3.5.2 Software Reliability Engineered Testing

Software Reliability Engineered Testing (SRET) is a testing method 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."

2.6.3.6 Techniques based on the nature of the application

The above techniques apply to all types of software. However, for some kinds of applications, some additional know-how is required for test derivation. A list of a few specialized testing fields is provided here, based on the nature of the application under test:

- Object-oriented testing
- Component-based testing
- Web-based testing
- GUI testing
- Testing of concurrent programs
- Protocol conformance testing
- Testing of real-time systems
- Testing of safety-critical systems (IEEE1228-94)

2.6.3.7 Selecting and combining techniques

2.6.3.7.1 Functional and structural

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

2.6.3.7.2 Deterministic vs. random

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

2.6.4 Test-related measures

Sometimes, test techniques are confused with test objectives. Test techniques are to be viewed as aids which help to ensure the achievement of test objectives. For instance, branch coverage is a popular test technique. Achieving a specified branch coverage measure should not be considered the objective of testing per se: 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-related measures, which provide an evaluation of the program under test based on the observed test outputs, and those which evaluate the thoroughness of the test set.

Measurement is usually considered instrumental to quality analysis. Measurement may also be used to optimize the planning and execution of the tests. Test management can use several process measures to monitor progress.

2.6.4.1 Evaluation of the program under test (IEEE982.1-98)

2.6.4.1.1 Program measurements to aid in planning and designing testing (IEE982.1-88)

Measures based on program size (for example, source lines of code or function points) or on program structure (like complexity) are used to guide testing. Structural measures can also include measurements among program modules in terms of the frequency with which modules call each other.

2.6.4.1.2 Fault types, classification, and statistics (EEE1044-93)

The testing literature is rich in classifications and taxonomies of faults. To make testing more effective, it is important to know which types of faults could be found in the software under test, and the relative frequency with which these faults have occurred in the past. This information can be very useful in making quality predictions, as well as for process improvement.

2.6.4.1.3 Fault density (IEEE982.1-88)

A program under test can be assessed by counting and classifying the discovered faults by their types. For each fault class, fault density is measured as the ratio between the number of faults found and the size of the program

2.6.4.1.4 Life test, reliability evaluation

A statistical estimate of software reliability, which can be obtained by reliability achievement and evaluation, n be used to evaluate a product and decide whether or not testing can be stopped.

2.6.4.1.5 Reliability growth models

Reliability growth models provide a prediction of reliability based on the failures observed under reliability achievement and evaluation They assume, in general, that the faults that caused the observed failures have been fixed (although some models also accept imperfect fixes), and thus, on average, the product's reliability exhibits an increasing trend. There now exist dozens of published models. Many are laid down on some common assumptions, while others differ. Notably, these models are divided into failure-count and time-between-failure models.

2.6.4.2 Evaluation of the tests performed

2.6.4.2.1 Coverage/thoroughness measures (IEEE982.1-88)

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

2.6.4.2.2 Fault seeding

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

2.6.4.2.3 Mutation score

In mutation testing, the ratio of killed mutants to the total number of generated mutants can be a measure of the effectiveness of the executed test set.

2.6.4.2.4 Comparison and relative effectiveness of different techniques

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

2.6.5 Test Process

Testing concepts, strategies, techniques, and measures need to be integrated into a defined and controlled process which is run by people. The test process supports testing activities and provides guidance to testing teams, from test planning to test output evaluation, in such a way as to provide justified assurance that the test objectives will be met cost-effectively.

2.6.5.1 Practical considerations

2.6.5.1.1 Attitudes/Egoless programming

A very important component of successful testing is a collaborative attitude towards testing and quality assurance activities. Managers have a key role in fostering a generally favorable reception towards failure discovery during development and maintenance; for instance, by preventing a mindset of code ownership among programmers, so that they will not feel responsible for failures revealed by their code.

2.6.5.1.2 Test guides

The testing phases could be guided by various aims, for example: in risk-based testing, which uses the product risks to prioritize and focus the test strategy; or in scenario-based testing, in which test cases are defined based on specified software scenarios.

2.6.5.1.3 Test process management (IEEE1074-97, IEEE12207.0-96:s5.3.9)

Test activities conducted at different levels must be organized, together with people, tools, policies, and measurements, into a well-defined process which is an integral part of the life cycle. In IEEE/EIA Standard 12207.0, testing is not described as a stand-alone process, but principles for testing activities are included along with both the five primary life cycle processes and the supporting process. In IEEE Std 1074, testing is grouped with other evaluation activities as integral to the entire life cycle.

2.6.5.1.4 Test documentation and work products (IEEE829-98)

Documentation is an integral part of the formalization of the test process. The IEEE Standard for Software Test Documentation (IEEE829-98) provides a good description of test documents and of their relationship with one another and with the testing process. Test documents may include, among others, Test Plan, Test Design Specification, Test Procedure Specification, Test Case Specification, Test Log, and Test Incident or Problem Report. The software under test is documented as the Test Item. Test documentation should be produced and continually updated, to the same level of quality as other types of documentation in software engineering.

2.6.5.1.5 Internal vs. independent test team

Formalization of the test process may involve formalizing the test team organization as well. The test team can be composed of internal members (that is, on the project team, involved or not in software construction), of external members, in the hope of bringing in an unbiased, independent perspective, or, finally, of both internal and external members. Considerations of costs, schedule, maturity levels of the involved organizations, and criticality of the application may determine the decision.

2.6.5.1.6 Cost/effort estimation and other process measures (IEEE982.1-88)

Several measures related to the resources spent on testing, as well as to the relative fault-finding effectiveness of the various test phases, are used by managers to control and improve the test process. These test measures may cover such aspects as number of test cases specified, number of test cases executed, number of test cases passed, and number of test cases failed, among others.

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

2.6.5.1.7 Termination

A decision must be made as to how much testing is enough and when a test stage can be terminated. Thoroughness measures, such as achieved code coverage or functional completeness, as well as estimates of fault density or of operational reliability, provide useful support, but are not sufficient in themselves. The decision also involves considerations about the costs and risks incurred by the potential for remaining failures, as opposed to the costs implied by continuing to test.

2.6.5.1.8 Test reuse and test patterns

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

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

2.6.5.2 Test Activities

Under this topic, a brief overview of test activities is given; as often implied by the following description, successful management of test activities strongly depends on the Software Configuration Management process.

2.6.5.2.1 Planning

Like any other aspect of project management, testing activities must be planned. Key aspects of test planning include coordination of personnel, management of available test facilities and equipment (which may include magnetic media, test plans and procedures), and planning for possible undesirable outcomes. If more than one baseline of the software is being maintained, then a major planning consideration is the time and effort needed to ensure that the test environment is set to the proper configuration.

2.6.5.2.2 Test-case generation

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

2.6.5.2.3 Test environment development

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

2.6.5.2.4 Execution

Execution of tests should embody a basic principle of scientific experimentation: everything done during testing should be performed and documented clearly enough that another person could replicate the results. Hence, testing should be performed in accordance with documented procedures using a clearly defined version of the software under test.

2.6.5.2.5 Test results evaluation

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

2.6.5.2.6 Problem reporting/Test log

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

2.6.5.2.7 Defect tracking

Failures observed during testing are most often due to faults or defects in the software. Such defects can be analyzed to determine when they were introduced into the software, what kind of error caused them to be created (poorly defined requirements, incorrect variable declaration, memory leak, programming syntax error, for example), and when they could have been first observed in the software. Defect-tracking information is used to determine what aspects of software engineering need improvement and how effective previous analyses and testing have been.

References:

http://en.wikipedia.org/wiki/Software_testing, http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-171Fall2003/CourseHome/, http://www.cs.cornell.edu/courses/cs501/2008sp/, http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/, http://www.ee.unb.ca/kengleha/courses/CMPE3213/Intro7 http://www.cs.kuleuven.ac.be/~dirk/ada-belgium/aia/contents.html#5, http://www.softwareqatest.com/qatfaq1.html, etc...

2.7 Software Maintenance[7]

2.7.1 Introduction

In software engineering, software maintenance is the process of enhancing and optimizing deployed software (software release), as well as remedying defects. Software maintenance is one of the phases in the software development process, and follows deployment of the software into the field. The software maintenance phase involves changes to the software in order to correct defects and deficiencies found during field usage as well as the addition of new functionality to improve the software's usability and applicability.

Software maintenance involves a number of specific techniques. One technique is static slicing, which is used to identify all the program code that can modify some variable. It is generally useful in refactoring program code and was specifically useful in assuring Y2K compliance.

The software maintenance phase is an explicit part of the waterfall model of the software development process which was developed during the structured programming movement of computer programming. The other major model, the spiral model developed during the object oriented movement of software engineering makes no explicit mention of a maintenance phase. Nevertheless, this activity is notable, considering the fact that two-thirds of a software system's lifetime cost involves maintenance.

[7] This content is available online at <http://cnx.org/content/m14717/1.1/>.

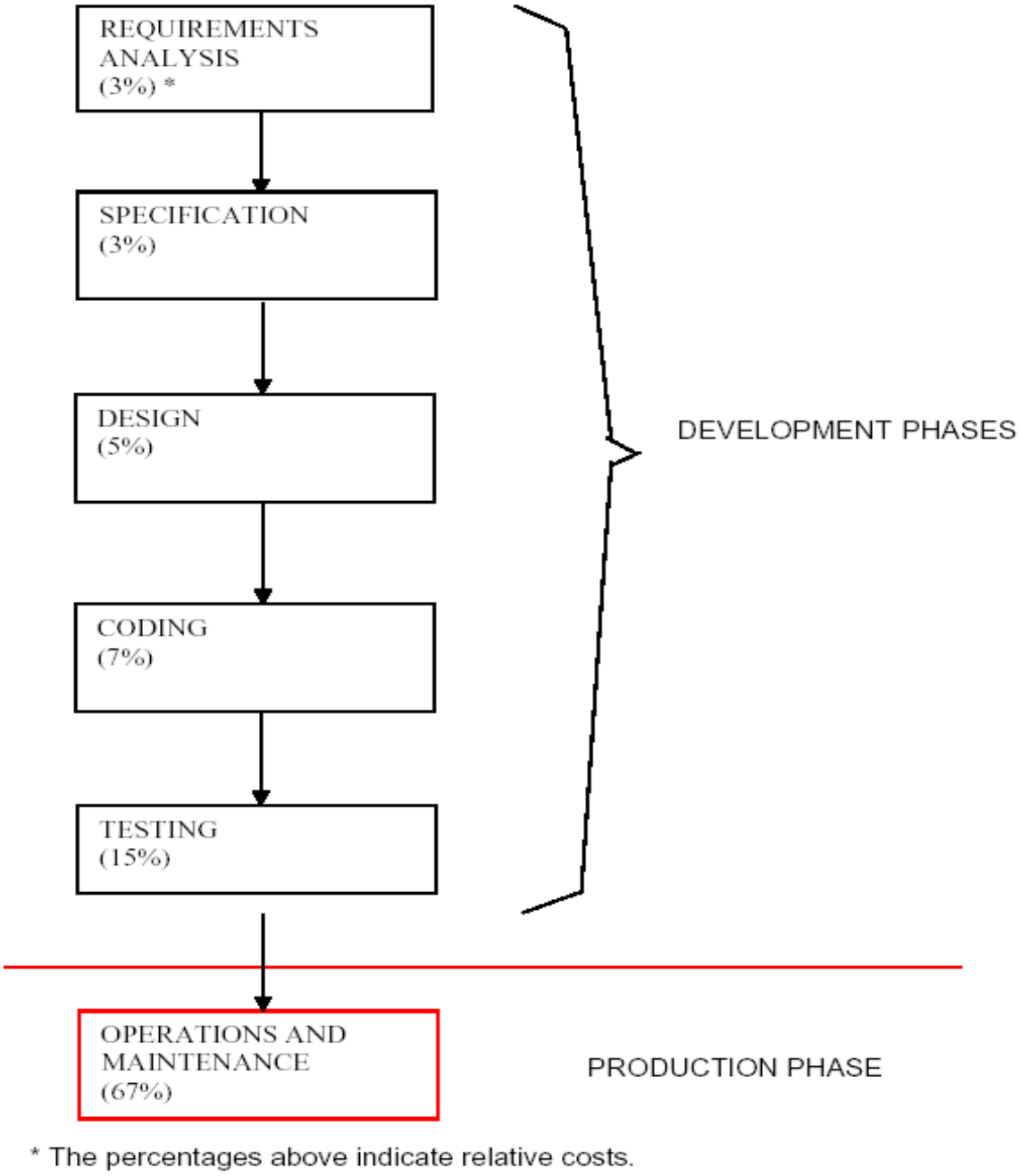


Figure 2.10

In a formal software development environment, the developing organization or team will have some mechanisms to document and track defects and deficiencies. Software just like most other products, is typically released with a known set of defects and deficiencies. The software is released with the issues because the development organization decides the utility and value of the software at a particular level of

quality outweighs the impact of the known defects and deficiencies.

The known issues are normally documented in a letter of operational considerations or release notes so that the users of the software will be able to work around the known issues and will know when the use of the software would be inappropriate for particular tasks.

With the release of the software, other, undocumented defects and deficiencies will be discovered by the users of the software. As these issues are reported into the development organization, they will be entered into the defect tracking system.

The people involved in the software maintenance phase are expected to work on these known issues, address them, and prepare for a new release of the software, known as a maintenance release, which will address the documented issues.

2.7.2 Software Maintenance Fundamentals

This section introduces the concepts and terminology that form an underlying basis to understanding the role and scope of software maintenance. The topics provide definitions and emphasize why there is a need for maintenance. Categories of software maintenance are critical to understanding its underlying meaning.

2.7.2.1 Definitions and Terminology

Software maintenance is defined as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. The standard also addresses maintenance activities prior to delivery of the software product, but only in an information appendix of the standard.

The IEEE/EIA 12207 standard for software life cycle processes essentially depicts maintenance as one of the primary life cycle processes, and describes maintenance as the process of a software product undergoing "modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity".

2.7.2.2 Nature of Maintenance

Software maintenance sustains the software product throughout its operational life cycle. Modification requests are logged and tracked, the impact of proposed changes is determined, code and other software artifacts are modified, testing is conducted, and a new version of the software product is released. Also, training and daily support are provided to users. Pfleeger states that "maintenance has a broader scope, with more to track and control" than development.

A maintainer is defined by IEEE/EIA 12207 as an organization which performs maintenance activities, sometimes refer to individuals who perform those activities, contrasting them with the developers.

IEEE/EIA 12207 identifies the primary activities of software maintenance as: process implementation; problem and modification analysis; modification implementation; maintenance review/acceptance; migration; and retirement.

Maintainers can learn from the developer's knowledge of the software. Contact with the developers and early involvement by the maintainer helps reduce the maintenance effort. In some instances, the software engineer cannot be reached or has moved on to other tasks, which creates an additional challenge for the maintainers. Maintenance must take the products of the development, code, or documentation, for example, and support them immediately and evolve/maintain them progressively over the software life cycle.

2.7.2.3 Need for Maintenance

Maintenance is needed to ensure that the software continues to satisfy user requirements. Maintenance is applicable to software developed using any software life cycle model (for example, spiral). The system changes due to corrective and non-corrective software actions. Maintenance must be performed in order to:

- Correct faults

- Improve the design
- Implement enhancements
- Interface with other systems
- Adapt programs so that different hardware, software, system features, and telecommunications facilities can be used
- Migrate legacy software
- Retire software

The maintainer's activities comprise four key characteristics, according to Pfleeger:

- Maintaining control over the software's day-to-day functions
- Maintaining control over software modification
- Perfecting existing functions
- Preventing software performance from degrading to unacceptable levels

2.7.2.4 Majority of Maintenance Costs

Maintenance consumes a major share of software life cycle financial resources. A common perception of software maintenance is that it merely fixes faults. However, studies and surveys over the years have indicated that the majority, over 80%, of the software maintenance effort is used for non-corrective actions. Jones describes the way in which software maintenance managers often group enhancements and corrections together in their management reports. This inclusion of enhancement requests with problem reports contributes to some of the misconceptions regarding the high cost of corrections. Understanding the categories of software maintenance helps to understand the structure of software maintenance costs. Also, understanding the factors that influence the maintainability of a system can help to contain costs. Pfleeger presents some of the technical and non-technical factors affecting software maintenance costs, as follows:

- Application type
- Software novelty
- Software maintenance staff availability
- Software life span
- Hardware characteristics
- Quality of software design, construction, documentation and testing

2.7.2.5 Evolution of Software

Lehman first addressed software maintenance and evolution of systems in 1969. Over a period of twenty years, his research led to the formulation of eight "Laws of Evolution". Key findings include the fact that maintenance is evolutionary developments, and that maintenance decisions are aided by understanding what happens to systems (and software) over time. Others state that maintenance is continued development, except that there is an extra input (or constraint)–existing large software is never complete and continues to evolve. As it evolves, it grows more complex unless some action is taken to reduce this complexity.

Since software demonstrates regular behavior and trends, these can be measured. Attempts to develop predictive models to estimate maintenance effort have been made, and, as a result, useful management tools have been developed.

2.7.2.6 Categories of Maintenance

Maintenance consists of four parts:

- Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems. It deals with fixing bugs in the code.

- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment. It deals with adapting the software to new environments.
- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability. It deals with updating the software according to changes in user requirements.
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults. It deals with updating documentation and making the software more maintainable.

All changes to the system can be characterized by these four types of maintenance. Corrective maintenance is 'traditional maintenance' while the other types are considered as 'software evolution'.

2.7.3 Key Issues in Software Maintenance

A number of key issues must be dealt with to ensure the effective maintenance of software. It is important to understand that software maintenance provides unique technical and management challenges for software engineers. Trying to find a fault in software containing 500K lines of code that the software engineer did not develop is a good example. Similarly, competing with software developers for resources is a constant battle. Planning for a future release, while coding the next release and sending out emergency patches for the current release, also creates a challenge. The following section presents some of the technical and management issues related to software maintenance. They have been grouped under the following topic headings:

- Technical issues
- Management issues
- Cost estimation and Measures

2.7.3.1 Technical Issues

2.7.3.1.1 Limited understanding

Limited understanding refers to how quickly a software engineer can understand where to make a change or a correction in software which this individual did not develop. Research indicates that some 40% to 60% of the maintenance effort is devoted to understanding the software to be modified. Thus, the topic of software comprehension is of great interest to software engineers. Comprehension is more difficult in text-oriented representation, in source code, for example, where it is often difficult to trace the evolution of software through its releases/versions if changes are not documented and when the developers are not available to explain it, which is often the case. Thus, software engineers may initially have a limited understanding of the software, and much has to be done to remedy this.

2.7.3.1.2 Testing

The cost of repeating full testing on a major piece of software can be significant in terms of time and money. Regression testing, the selective retesting of a software or component to verify that the modifications have not caused unintended effects, is important to maintenance. As well, finding time to test is often difficult. There is also the challenge of coordinating tests when different members of the maintenance team are working on different problems at the same time. When software performs critical functions, it may be impossible to bring it offline to test.

2.7.3.1.3 Impact analysis

Impact analysis describes how to conduct, cost effectively, a complete analysis of the impact of a change in existing software. Maintainers must possess an intimate knowledge of the software's structure and content. They use that knowledge to perform impact analysis, which identifies all systems and software products

affected by a software change request and develops an estimate of the resources needed to accomplish the change. Additionally, the risk of making the change is determined. The change request, sometimes called a modification request (MR) and often called a problem report (PR), must first be analyzed and translated into software terms. It is performed after a change request enters the software configuration management process. Arthur states that the objectives of impact analysis are:

- Determination of the scope of a change in order to plan and implement work
- Development of accurate estimates of resources needed to perform the work
- Analysis of the cost/benefits of the requested change
- Communication to others of the complexity of a given change

The severity of a problem is often used to decide how and when a problem will be fixed. The software engineer then identifies the affected components. Several potential solutions are provided and then a recommendation is made as to the best course of action.

Software designed with maintainability in mind greatly facilitates impact analysis.

2.7.3.1.4 Maintainability

How does one promote and follow up on maintainability issues during development? The IEEE [IEEE610.12-90] defines maintainability as the ease with which software can be maintained, enhanced, adapted, or corrected to satisfy specified requirements. ISO/IEC defines maintainability as one of the quality characteristics (ISO9126-01).

Maintainability sub-characteristics must be specified, reviewed, and controlled during the software development activities in order to reduce maintenance costs. If this is done successfully, the maintainability of the software will improve. This is often difficult to achieve because the maintainability sub-characteristics are not an important focus during the software development process. The developers are preoccupied with many other things and often disregard the maintainer's requirements. This in turn can, and often does, result in a lack of system documentation, which is a leading cause of difficulties in program comprehension and impact analysis. It has also been observed that the presence of systematic and mature processes, techniques, and tools helps to enhance the maintainability of a system.

2.7.3.2 Management Issues

2.7.3.2.1 Alignment with organizational objectives

Organizational objectives describe how to demonstrate the return on investment of software maintenance activities. Bennett states that "initial software development is usually project-based, with a defined time scale and budget. The main emphasis is to deliver on time and within budget to meet user needs. In contrast, software maintenance often has the objective of extending the life of software for as long as possible. In addition, it may be driven by the need to meet user demand for software updates and enhancements. In both cases, the return on investment is much less clear, so that the view at senior management level is often of a major activity consuming significant resources with no clear quantifiable benefit for the organization."

2.7.3.2.2 Staffing

Staffing refers to how to attract and keep software maintenance staff. Maintenance is often not viewed as glamorous work. Deklava provides a list of staffing-related problems based on survey data. As a result, software maintenance personnel are frequently viewed as "second-class citizens" and morale therefore suffers.

2.7.3.2.3 Process

Software process is a set of activities, methods, practices, and transformations which people use to develop and maintain software and the associated products. At the process level, software maintenance activities share much in common with software development (for example, software configuration management is

a crucial activity in both). Maintenance also requires several activities which are not found in software development. These activities present challenges to management.

2.7.3.2.4 Organizational aspects of maintenance

Organizational aspects describe how to identify which organization and/or function will be responsible for the maintenance of software. The team that develops the software is not necessarily assigned to maintain the software once it is operational.

In deciding where the software maintenance function will be located, software engineering organizations may, for example, stay with the original developer or go to a separate team (or maintainer). Often, the maintainer option is chosen to ensure that the software runs properly and evolves to satisfy changing user needs. Since there are many pros and cons to each of these options, the decision should be made on a case-by-case basis. What is important is the delegation or assignment of the maintenance responsibility to a single group or person, regardless of the organization's structure.

2.7.3.2.5 Outsourcing

Outsourcing of maintenance is becoming a major industry. Large corporations are outsourcing entire portfolios of software systems, including software maintenance. More often, the outsourcing option is selected for less mission-critical software, as companies are unwilling to lose control of the software used in their core business. Carey reports that some will outsource only if they can find ways of maintaining strategic control. However, control measures are hard to find. One of the major challenges for the outsourcers is to determine the scope of the maintenance services required and the contractual details. McCracken states that 50% of outsourcers provide services without any clear service-level agreement. Outsourcing companies typically spend a number of months assessing the software before they will enter into a contractual relationship. Another challenge identified is the transition of the software to the outsourcer.

2.7.3.3 Maintenance Cost Estimation

Software engineers must understand the different categories of software maintenance, discussed above, in order to address the question of estimating the cost of software maintenance. For planning purposes, estimating costs is an important aspect of software maintenance.

2.7.3.3.1 Cost estimation

It was mentioned in "Impact Analysis", that impact analysis identifies all systems and software products affected by a software change request and develops an estimate of the resources needed to accomplish that change.

Maintenance cost estimates are affected by many technical and non-technical factors. ISO/IEC14764 states that "the two most popular approaches to estimating resources for software maintenance are the use of parametric models and the use of experience". Most often, a combination of these is used.

2.7.3.3.2 Parametric models

Some work has been undertaken in applying parametric cost modeling to software maintenance. Significance is that data from past projects are needed in order to use the models. Jones discusses all aspects of estimating costs, including function points (IEEE14143.1-00), and provides a detailed chapter on maintenance estimation.

2.7.3.3.3 Experience

Experience, in the form of expert judgment (using the Delphi technique, for example), analogies, and a work breakdown structure, are several approaches which should be used to augment data from parametric models.

Clearly the best approach to maintenance estimation is to combine empirical data and experience. These data should be provided as a result of a measurement program.

2.7.3.4 Software Maintenance Measurement

Grady and Caswell discuss establishing a corporate-wide software measurement program, in which software maintenance measurement forms and data collection are described. The Practical Software and Systems Measurement (PSM) project describes an issue-driven measurement process that is used by many organizations and is quite practical.

There are software measures that are common to all endeavors, the following categories of which the Software Engineering Institute (SEI) has identified: size; effort; schedule; and quality. These measures constitute a good starting point for the maintainer.

2.7.3.4.1 Specific Measures

Abran presents internal benchmarking techniques to compare different internal maintenance organizations. The maintainer must determine which measures are appropriate for the organization in question. IEEE1219-98 suggests measures which are more specific to software maintenance measurement programs. That list includes a number of measures for each of the four sub-characteristics of maintainability:

- Analyzability: Measures of the maintainer's effort or resources expended in trying to diagnose deficiencies or causes of failure, or in identifying parts to be modified
- Changeability: Measures of the maintainer's effort associated with implementing a specified modification
- Stability: Measures of the unexpected behavior of software, including that encountered during testing
- Testability: Measures of the maintainer's and users' effort in trying to test the modified software

Certain measures of the maintainability of software can be obtained using available commercial tools.

2.7.4 Maintenance Process

The Maintenance Process subarea provides references and standards used to implement the software maintenance process. The Maintenance Activities topic differentiates maintenance from development and shows its relationship to other software engineering activities.

2.7.4.1 Maintenance Processes

Maintenance processes provide needed activities and detailed inputs/outputs to those activities, and are described in software maintenance standards IEEE 1219 and ISO/IEC 14764.

The maintenance process model described in the Standard for Software Maintenance (IEEE1219) starts with the software maintenance effort during the post-delivery stage and discusses items such as planning for maintenance.

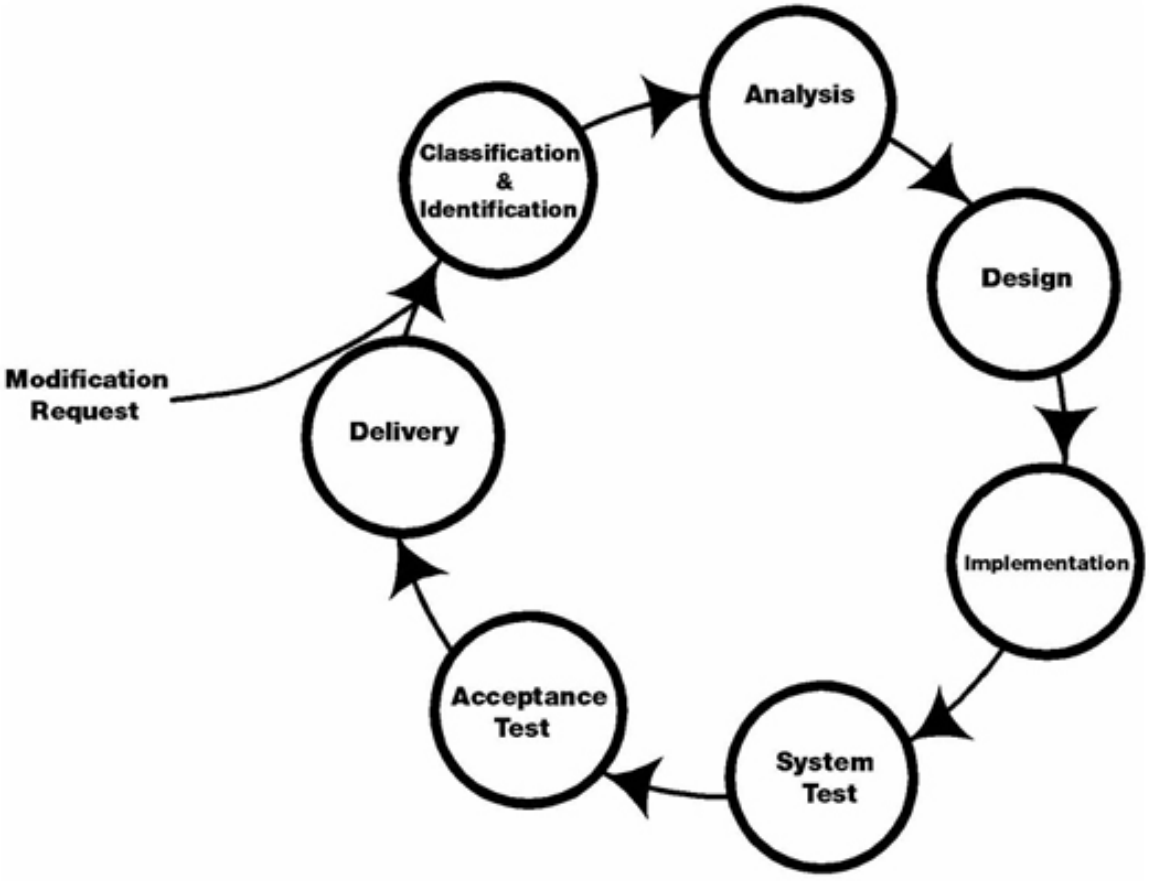


Figure 2.11

ISO/IEC 14764 is an elaboration of the IEEE/EIA 12207.0-96 maintenance process. The activities of the ISO/IEC maintenance process are similar to those of the IEEE, except that they are aggregated a little differently.

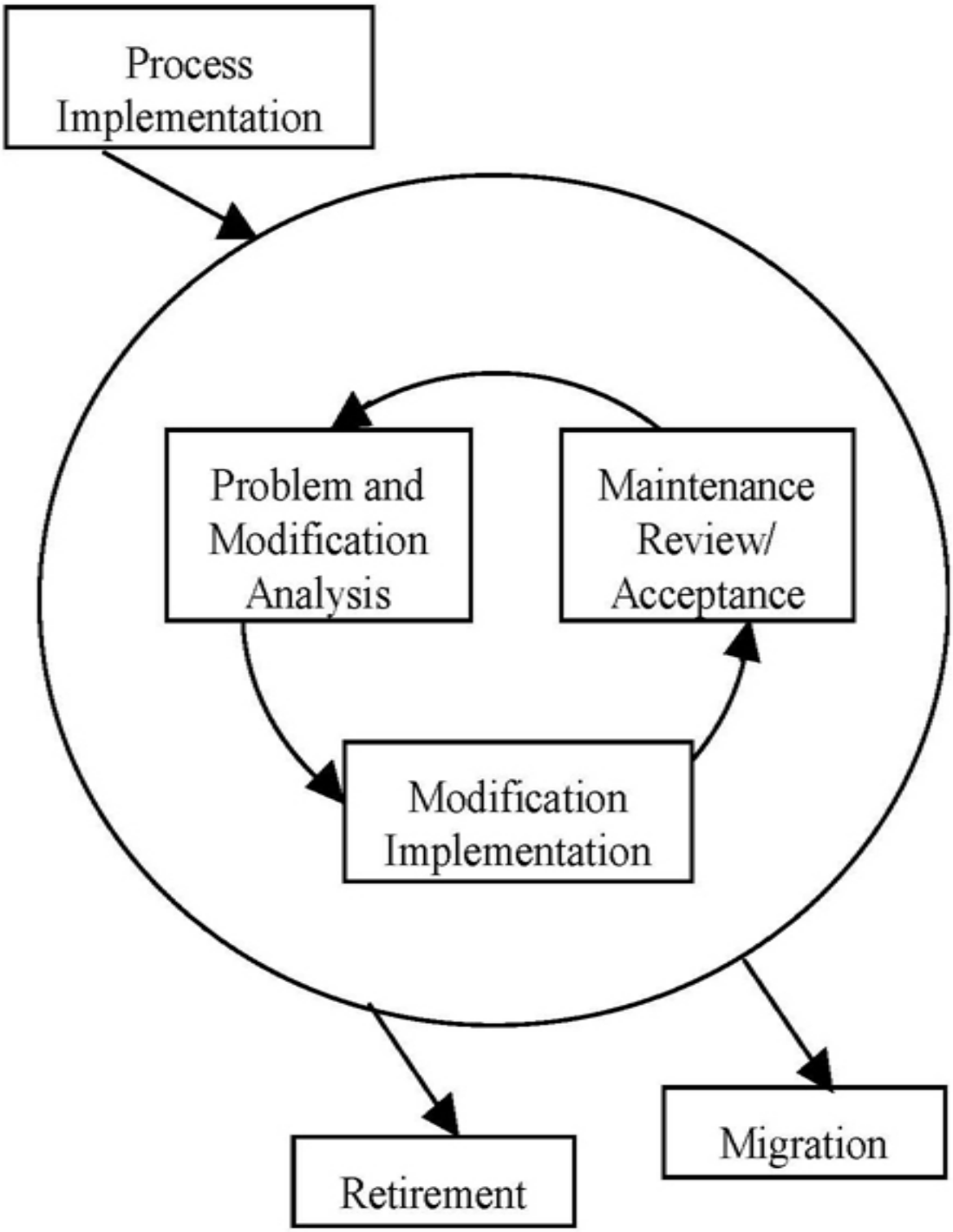

Process Implementation
Problem and Modification Analysis
Maintenance Review/ Acceptance
Modification Implementation
Retirement
Migration

Figure 2.12

Each of the ISO/IEC 14764 primary software maintenance activities is further broken down into tasks, as follows.

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

2.7.4.2 Maintenance Activities

As already noted, many maintenance activities are similar to those of software development. Maintainers perform analysis, design, coding, testing, and documentation. They must track requirements in their activities just as is done in development, and update documentation as baselines change. ISO/IEC14764 recommends that, when a maintainer refers to a similar development process, he must adapt it to meet his specific need. However, for software maintenance, some activities involve processes unique to software maintenance.

2.7.4.2.1 Unique activities

There are a number of processes, activities, and practices that are unique to software maintenance, for example:

- Transition: a controlled and coordinated sequence of activities during which software is transferred progressively from the developer to the maintainer.
- Modification Request Acceptance/Rejection: modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.
- Modification Request and Problem Report Help Desk: an end-user support function that triggers the assessment, prioritization, and costing of modification requests.
- Impact Analysis.
- Software Support: help and advice to users concerning a request for information (for example, business rules, validation, data meaning and ad-hoc requests/reports).
- Service Level Agreements (SLAs) and specialized (domain-specific) maintenance contracts which are the responsibility of the maintainers.

2.7.4.2.2 Supporting activities

Maintainers may also perform supporting activities, such as software maintenance planning, software configuration management, verification and validation, software quality assurance, reviews, audits, and user training.

2.7.4.2.3 Maintenance planning activity

An important activity for software maintenance is planning, and maintainers must address the issues associated with a number of planning perspectives:

- Business planning (organizational level)
- Maintenance planning (transition level)
- Release/version planning (software level)
- Individual software change request planning (request level)

At the individual request level, planning is carried out during the impact analysis. The release/version planning activity requires that the maintainer:

- Collect the dates of availability of individual requests
- Agree with users on the content of subsequent releases/versions
- Identify potential conflicts and develop alternatives
- Assess the risk of a given release and develop a back-out plan in case problems should arise
- Inform all the stakeholders

Whereas software development projects can typically last from some months to a few of years, the maintenance phase usually lasts for many years. Making estimates of resources is a key element of maintenance planning. Those resources should be included in the developers' project planning budgets. Software maintenance planning should begin with the decision to develop a new system and should consider quality objectives (IEEE1061-98). A concept document should be developed, followed by a maintenance plan.

The concept document for maintenance should address:

- The scope of the software maintenance
- Adaptation of the software maintenance process
- Identification of the software maintenance organization
- An estimate of software maintenance costs

The next step is to develop a corresponding software maintenance plan. This plan should be prepared during software development, and should specify how users will request software modifications or report problems. Software maintenance planning is addressed in IEEE 1219 and ISO/IEC 14764. ISO/IEC14764 provides guidelines for a maintenance plan.

Finally, at the highest level, the maintenance organization will have to conduct business planning activities (budgetary, financial, and human resources) just like all the other divisions of the organization.

2.7.4.2.4 Software configuration management

The IEEE Standard for Software Maintenance, IEEE 1219, describes software configuration management as a critical element of the maintenance process. Software configuration management procedures should provide for the verification, validation, and audit of each step required to identify, authorize, implement, and release the software product.

It is not sufficient to simply track Modification Requests or Problem Reports. The software product and any changes made to it must be controlled. This control is established by implementing and enforcing an approved software configuration management (SCM) process. SCM for software maintenance is different from SCM for software development in the number of small changes that must be controlled on operational software. The SCM process is implemented by developing and following a configuration management plan and operating procedures. Maintainers participate in Configuration Control Boards to determine the content of the next release/version.

2.7.4.2.5 Software quality

It is not sufficient, either, to simply hope that increased quality will result from the maintenance of software. It must be planned and processes implemented to support the maintenance process. The activities and techniques for Software Quality Assurance (SQA), V&V, reviews, and audits must be selected in concert with all the other processes to achieve the desired level of quality. It is also recommended that the maintainer adapt the software development processes, techniques and deliverables, for instance testing documentation, and test results.

2.7.5 Techniques for Maintenance

This subarea introduces some of the generally accepted techniques used in software maintenance.

2.7.5.1 Program Comprehension

Programmers spend considerable time in reading and understanding programs in order to implement changes. Code browsers are key tools for program comprehension. Clear and concise documentation can aid in program comprehension.

2.7.5.2 Reengineering

Reengineering is defined as the examination and alteration of software to reconstitute it in a new form, and includes the subsequent implementation of the new form. Dorfman and Thayer state that reengineering is the most radical (and expensive) form of alteration. Others believe that reengineering can be used for minor changes. It is often not undertaken to improve maintainability, but to replace aging legacy software. Arnold provides a comprehensive compendium of topics, for example: concepts, tools and techniques, case studies, and risks and benefits associated with reengineering.

2.7.5.3 Reverse engineering

Reverse engineering is the process of analyzing software to identify the software's components and their interrelationships and to create representations of the software in another form or at higher levels of abstraction. Reverse engineering is passive; it does not change the software, or result in new software. Reverse engineering efforts produce call graphs and control flow graphs from source code. One type of reverse engineering is redocumentation. Another type is design recovery. Refactoring is program transformation which reorganizes a program without changing its behavior, and is a form of reverse engineering that seeks to improve program structure.

Finally, data reverse engineering has gained in importance over the last few years where logical schemas are recovered from physical databases.

2.7.6 Tools

2.7.6.1 Introduction

A software maintenance tool is an artifact that supports a software maintainer in performing a task. The use of tools for software maintenance simplifies the tasks and increases efficiency and productivity.

There are several criteria for selecting the right tool for the task. These criteria are capability, features, cost/benefit, platform, programming language, ease of use, openness

of architecture, stability of vendor, and organizational culture.

Capability decides whether the tool is capable of fulfilling the task. Once it has been decided that a method can benefit from being automated, then the features of the tool need to be considered for the job.

The tool must be analyzed for the benefits it brings against its cost. The benefit indicators of a tool are quality, productivity, responsiveness, and cost reduction. The environment that the tool runs on is called the platform. The language of the source code is called the programming language. It's important to select a tool that supports a language that is an industry standard.

The tool should have a similar feel to the ones that the users are already familiar with. The tool should have the ability to be integrated with different vendors' tools. This will help when a tool will need to run with other tools. The openness of the architecture plays an important role when the maintenance problem is complex. Therefore, it is not always sufficient to use only one tool. There may need to be multiple tools running together.

It is also important to consider the vendor's credibility. The vendor should be capable of supporting the tool in the future. If the vendor is not stable, the vendor could run out of business and not be able to support the tool. Another important factor is the culture of the organization. Every culture has its own work pattern. Therefore, it is important to take into consideration whether the tool is going to be accepted by the target users.

The chosen tools must support program understanding and reverse engineering, testing, configuration management, and documentation.

Selecting a tool that promotes understanding is very important in the implementation of change since a large amount of time is used to study and understand programs.

Tools for reverse engineering also accomplish the same goal. The tools mainly consist of visualization tools, which assist the programmer in drawing a model of the system.

Examples of program understanding and reverse engineering tools include the program slicer static analyzer, dynamic analyzer, cross-referencer and dependency analyzer.

Slicing is the mechanical process of marking all the sections of a program text that may influence the value of a variable at a given point in the program. Program slicing helps the programmers select and view only the parts of the program that are affected by the changes. Static analyzer is used in analyzing the different parts if the program such as modules, procedures, variables, data elements, objects and classes. A static analyzer allows general viewing of the program text and generates summaries of contents and usage of selected elements in the program text, such as variables or objects.

A dynamic analyzer could be used to analyze the program while it is executing. A data flow analyzer is a static analysis tool that allows the maintainer to track all possible data flow and control flow paths in the program. It allows analysis of the program to better outline the underlying logic of the program. It also helps display the relationship between components of the system. A cross-referencer produces information on the usage of a program. This tool helps the user focus on the parts that are affected by the change.

A dependency analyzer assists the maintainer to analyze and understand the interrelationships between entities in a program. Such a tool provides capabilities to set up and query the database of the dependencies in a program. It also provides graphical representations of the dependencies. Testing is the most time consuming and demanding task in software maintenance.

Therefore, it could benefit the most from tools. A test simulator tool helps the maintainer

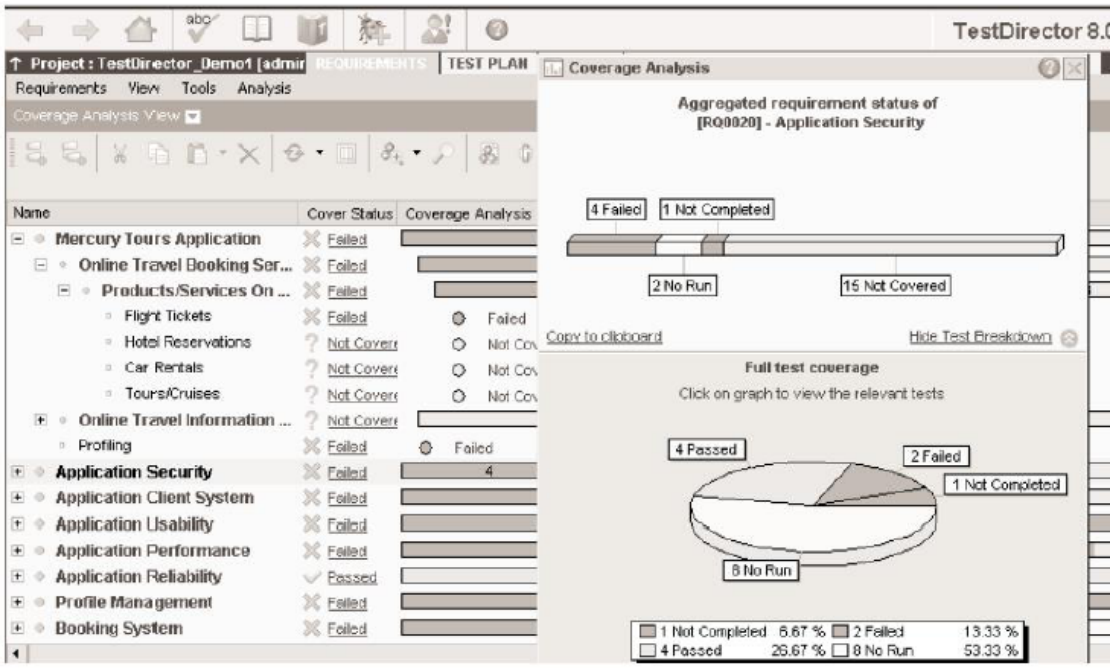
to test the effects of the change in a controlled environment before implementing the change on the actual system. A test case generator produces test data that is used to test the functionality of the modified system, while a test path generator helps the maintainer to find all the data flow and control flow paths affected by the changes.

Configuration management benefits from automated tools. Configuration management and version control tools help store the objects that form the software system. A source control system is used to keep a history of the files so that versions can be tracked and the programmer can keep track of the file changes.

2.7.6.2 Commercially available products

There are numerous products on the market available for software maintenance. One type

of product is bug tracking tools, which play an important role in maintenance. Bugzilla by the Mozilla Foundation is an example of such a tool. Other bug tracking products are Test Director by Mercury Interactive, Silk Radar by Segue Software, SQA Manager by Rational software, and QA director by Compuware.



TestDirector's Requirements Manager links test cases to testing requirements, ensuring traceability.

Figure 2.13

ProTeus III Expert CMMS by Eagle Technology, Inc. is a maintenance software package that lets users schedule preventative maintenance, generate automatic work orders, document equipment maintenance history, track assets and inventory, track personnel, create purchase orders, and generate reports. Microsoft Visual Source Safe is a source control system tool that is used by configuration management.

Products that are specific to programming languages are CCFinder and JAAT which

is specifically designed for JAVA programs. CCFinder identifies code clones in JAVA program. JAAT executes alias analysis for JAVA programs. For C++ programs, there is a tool called OCL query-based debugger which is a tool to debug C++ programs using queries formulated in the object constraint language.

2.7.6.3 Summary of tools

The task of software maintenance has become so vital and complex that automated support is required to do it effectively. The use of tools simplifies tasks, increase efficiency and productivity. There are numerous tools available on the market for maintenance.

2.8 Software configuration management[8]

2.8.1 Introduction

A system can be defined as a collection of components organized to accomplish a specific function or set of functions. The configuration of a system is the functional and/or physical characteristics of hardware, firmware, or software, or a combination of these, as set forth in technical documentation and achieved in a product. It can also be thought of as a collection of specific versions of hardware, firmware, or software items combined according to specific build procedures to serve a particular purpose. Configuration management (CM), then, is the discipline of identifying the configuration of a system at distinct points in time for the purpose of systematically controlling changes to the configuration, and maintaining the integrity and traceability of the configuration throughout the system life cycle. It is formally defined as "A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements."

Software configuration management (SCM) is a critical element of software engineering.

Unfortunately, in practice it is often ignored until absolutely necessary. It may be introduced at first customer release, possibly through customer pressure. Tool support for SCM is limited in that only certain aspects of software development and maintenance are accommodated. SCM methods and tools are often viewed as intrusive by developers, a management tool that imposes additional work with little perceived benefit to the tasks of the developer.

Software configuration management (SCM) is a supporting software life cycle process which benefits project management, development and maintenance activities, assurance activities, and the customers and users of the end product.

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

SCM is closely related to the software quality assurance (SQA) activity. SQA processes provide assurance that the software products and processes in the project life cycle conform to their specified requirements by planning, enacting, and performing a set of activities to provide adequate confidence that quality is being built into the software. SCM activities help in accomplishing these SQA goals.

The SCM activities are: management and planning of the SCM process, software configuration identification, software configuration control, software configuration status accounting, software configuration auditing, and software release management and delivery.

The figure following shows a stylized representation of these activities:

[8] This content is available online at <http://cnx.org/content/m14730/1.1/>.

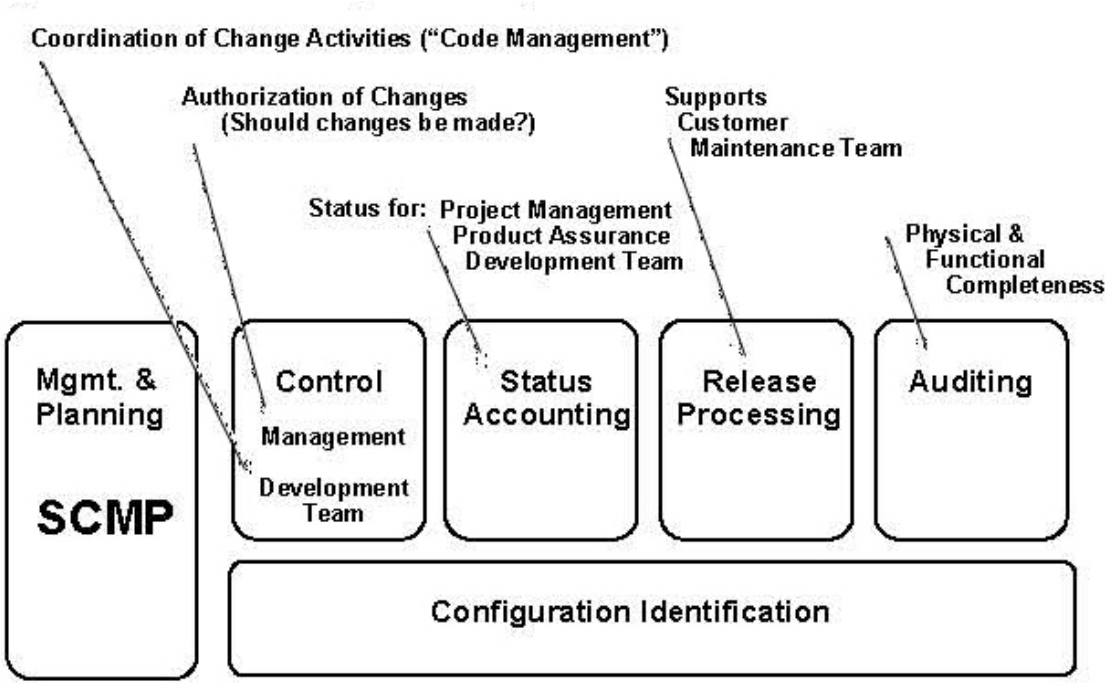


Figure 2.14

2.8.2 Management of the SCM Process

SCM controls the evolution and integrity of a product by identifying its elements, managing and controlling change, and verifying, recording, and reporting on configuration information. From the software engineer's perspective, SCM facilitates development and change implementation activities. A successful SCM implementation requires careful planning and management. This, in turn, requires an understanding of the organizational context for, and the constraints placed on, the design and implementation of the SCM process.

2.8.2.1 Organizational Context for SCM

To plan an SCM process for a project, it is necessary to understand the organizational context and the relationships among the organizational elements. SCM interacts with several other activities or organizational elements.

The organizational elements responsible for the software engineering supporting processes may be structured in various ways. Although the responsibility for performing certain SCM tasks might be assigned to other parts of the organization such as the development organization, the overall responsibility for SCM often rests with a distinct organizational element or designated individual.

Software is frequently developed as part of a larger system containing hardware and firmware elements. In this case, SCM activities take place in parallel with hardware and firmware CM activities, and must be consistent with system-level CM. Buckley describes SCM within this context. Note that firmware contains hardware and software, therefore both hardware and software CM concepts are applicable.

SCM might interface with an organization's quality assurance activity on issues such as records management and non-conforming items. Regarding the former, some tems under SCM control might also be project

records subject to provisions of the organization's quality assurance program. Managing nonconforming items is usually the responsibility of the quality assurance activity; however, SCM might assist with tracking and reporting on software configuration items falling into this category.

Perhaps the closest relationship is with the software development and maintenance organizations.

It is within this context that many of the software configuration control tasks are conducted. Frequently, the same tools support development, maintenance, and SCM purposes.

2.8.2.2 Constraints and Guidance for the SCM Process

Constraints affecting, and guidance for, the SCM process come from a number of sources. Policies and procedures set forth at corporate or other organizational levels might influence or prescribe the design and implementation of the SCM process for a given project. In addition, the contract between the acquirer and the supplier might contain provisions affecting the SCM process. For example, certain configuration audits might be required, or it might be specified that certain items be placed under CM. When software products to be developed have the potential to affect public safety, external regulatory bodies may impose constraints. Finally, the particular software life cycle process chosen for a software project and the tools selected to implement the software affect the design and implementation of the SCM process.

Guidance for designing and implementing an SCM process can also be obtained from "best practice," as reflected in the standards on software engineering issued by the various standards organizations. Moore provides a roadmap to these organizations and their standards. Best practice is also reflected in process improvement and process assessment models such as the Software Engineering Institute's Capability Maturity Model Integration (SEI/CMMI) and ISO/IEC15504 Software Engineering–Process Assessment (ISO/IEC 15504-98).

2.8.2.3 Planning for SCM

The planning of an SCM process for a given project should be consistent with the organizational context, applicable constraints, commonly accepted guidance, and the nature of the project (for example, size 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 an SCM Plan (SCMP), which is typically subject to SQA review and audit.

2.8.2.3.1 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 by organizational element. The overall authority and reporting channels for SCM should also be identified, although this might be accomplished at the project management or quality assurance planning stage.

2.8.2.3.2 SCM resources and schedules

Planning for SCM identifies the staff and tools involved in carrying out SCM activities and tasks. It addresses scheduling questions by establishing necessary sequences of SCM tasks and identifying their relationships to the project schedules and milestones established at the project management planning stage. Any training requirements necessary for implementing the plans and training new staff members are also specified.

2.8.2.3.3 Tool selection and implementation

Different types of tool capabilities, and procedures for their use, support SCM activities. Depending on the situation, these tool capabilities can be made available with some combination of manual tools, automated tools providing a single SCM capability, automated tools integrating a range of SCM (and perhaps other) capabilities, or integrated tool environments which serve the needs of multiple participants in the software engineering process (for example, SCM, development, V&V). Automated tool support becomes increasingly important, and increasingly difficult to establish, as projects grow in size and as project environments become more complex. These tool capabilities provide support for:

- the SCM Library
- the software change request (SCR) and approval procedures
- code (and related work products) and change management tasks
- reporting software configuration status and collecting SCM measurements
- software configuration auditing
- managing and tracking software documentation
- performing software builds
- managing and tracking software releases and their delivery

The tools used in these areas can also provide measurements for process improvement. Royce describes seven core measures of value in managing software engineering processes. Information available from the various SCM tools relates to Royce's Work and Progress management indicator and to his quality indicators of Change Traffic and Stability, Breakage and Modularity, Rework and Adaptability, and MTBF (mean time between failures) and Maturity. Reporting on these indicators can be organized in various ways, such as by software configuration item or by type of change requested.

We can represent a mapping of tool capabilities and procedures to SCM Activities:

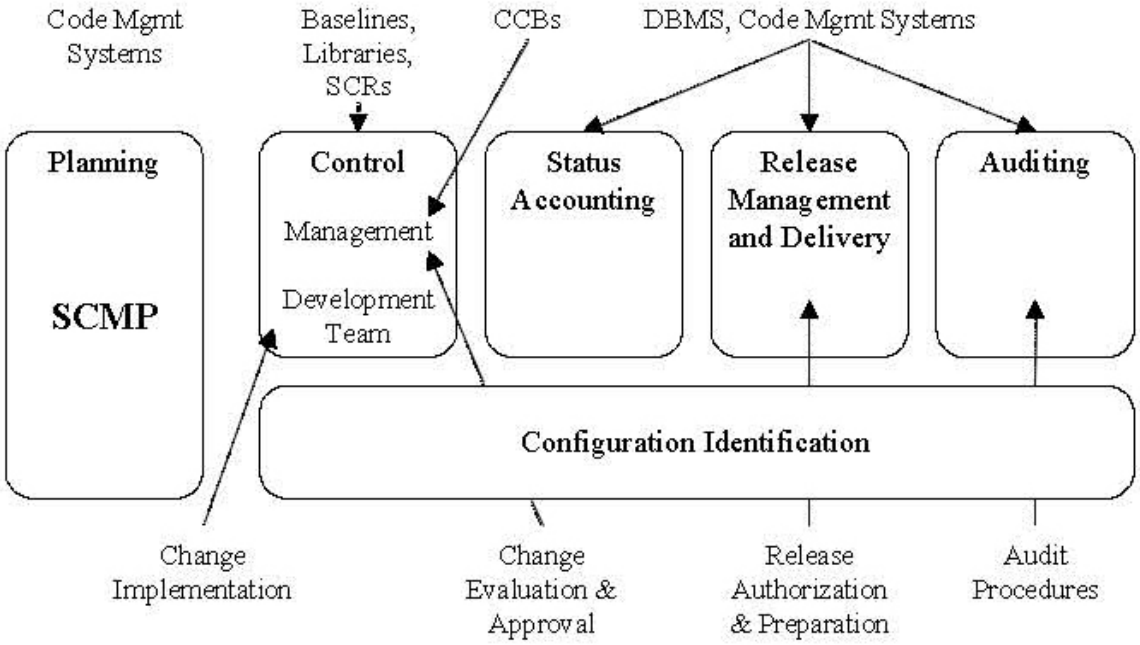


Figure 2.15

In this example, code management systems support the operation of software libraries by controlling access to library elements, coordinating the activities of multiple users, and helping to enforce operating procedures. Other tools support the process of building software and release documentation from the software elements contained in the libraries. Tools for managing software change requests support the change control procedures applied to controlled software items. Other tools can provide database management and reporting capabilities for management, development, and quality assurance activities. As mentioned above, the capabilities of several tool types might be integrated into SCM systems, which in turn are closely coupled to various other software activities.

In planning, the software engineer picks SCM tools fit for the job. Planning considers issues that might arise in the implementation of these tools, particularly if some form of culture change is necessary.

2.8.2.3.4 Vendor/Subcontractor Control

A software project might acquire or make use of purchased software products, such as compilers or other tools. SCM planning considers if and how these items will be taken under configuration control (for example, integrated into the project libraries) and how changes or updates will be evaluated and managed.

Similar considerations apply to subcontracted software. In this case, the SCM requirements to be imposed on the subcontractor's SCM process as part of the subcontract and the means for monitoring compliance also need to be established. The latter includes consideration of what SCM information must be available for effective compliance monitoring.

2.8.2.3.5 Interface control

When a software item will interface with another software or hardware item, a change to either item can affect the other. The planning for the SCM process considers how the interfacing items will be identified and how changes to the items will be managed and communicated. The SCM role may be part of a larger, system-level process for interface specification and control, and may involve interface specifications, interface control plans, and interface control documents. In this case, SCM planning for interface control takes place within the context of the system-level process.

2.8.2.4 SCM Plan

The results of SCM planning for a given project are recorded in a Software Configuration Management Plan (SCMP), a "living document" which serves as a reference for the SCM process. It is maintained (that is, updated and approved) as necessary during the software life cycle. In implementing the SCMP, it is typically necessary to develop a number of more detailed, subordinate procedures defining how specific requirements will be carried out during day-to-day activities.

Guidance on the creation and maintenance of an SCMP, based on the information produced by the planning activity, is available from a number of sources, such as IEEE828-98. This reference provides requirements for the information to be contained in an SCMP. It also defines and describes six categories of SCM information to be included in an SCMP:

- Introduction (purpose, scope, terms used)
- SCM Management (organization, responsibilities, authorities, applicable policies, directives, and procedures)
- SCM Activities (configuration identification, configuration control, and so on)
- SCM Schedules (coordination with other project activities)
- SCM Resources (tools, physical resources, and humanresources)
- SCMP Maintenance

2.8.2.5 Surveillance of Software Configuration Management

After the SCM process has been implemented, some degree of surveillance may be necessary to ensure that the provisions of the SCMP are properly carried out. There are likely to be specific SQA requirements for ensuring compliance with specified SCM processes and procedures. This could involve an SCM authority ensuring that those with the assigned responsibility perform the defined SCM tasks correctly. The software quality assurance authority, as part of a compliance auditing activity, might also perform this surveillance.

The use of integrated SCM tools with process control capability can make the surveillance task easier. Some tools facilitate process compliance while providing flexibility for the software engineer to adapt procedures. Other tools enforce process, leaving the software engineer with less flexibility. Surveillance requirements and the level of flexibility to be provided to the software engineer are important considerations in tool selection.

2.8.2.5.1 SCM measures and measurement

SCM measures can be designed to provide specific information on the evolving product or to provide insight into the functioning of the SCM process. A related goal of monitoring the SCM process is to discover opportunities for process improvement. Measurements of SCM processes provide a good means for monitoring the effectiveness of SCM activities on an ongoing basis. These measurements are useful in characterizing the current state of the process, as well as in providing a basis for making comparisons over time. Analysis of the measurements may produce insights leading to process changes and corresponding updates to the SCMP.

Software libraries and the various SCM tool capabilities provide sources for extracting information about the characteristics of the SCM process (as well as providing project and management information). For example, information about the time required to accomplish various types of changes would be useful in an evaluation of the criteria for determining what levels of authority are optimal for authorizing certain types of changes.

Care must be taken to keep the focus of the surveillance on the insights that can be gained from the measurements, not on the measurements themselves.

2.8.2.5.2 In-process audits of SCM

Audits can be carried out during the software engineering process to investigate the current status of specific elements of the configuration or to assess the implementation of the SCM process. In-process auditing of SCM provides a more formal mechanism for monitoring selected aspects of the process and may be coordinated with the SQA function.

2.8.3 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.

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

2.8.3.1.1 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. It can be viewed as a part of an overall system

configuration.

2.8.3.1.2 Software configuration item

A software configuration item (SCI) is an aggregation of software designated for configuration management and is treated as a single entity in the SCM process. A variety of items, in addition to the code itself, is typically controlled by SCM. Software items with potential to become SCIs include plans, specifications and design documentation, testing materials, software tools, source and executable code, code libraries, data and data dictionaries, and documentation for installation, maintenance, operations, and software use.

Selecting SCIs is an important process in which a balance must be achieved between providing adequate visibility for project control purposes and providing a manageable number of controlled items.

2.8.3.1.3 Software configuration item relationships

The structural relationships among the selected SCIs, and their constituent parts, affect other SCM activities or tasks, such as software building or analyzing the impact of proposed changes. Proper tracking of these relationships is also important for supporting traceability. The design of the identification scheme for SCIs should consider the need to map the identified items to the software structure, as well as the need to support the evolution of the software items and their relationships.

2.8.3.1.4 Software version

Software items evolve as a software project proceeds. A version of a software item is a particular identified and specified item. It can be thought of as a state of an evolving item. A revision is a new version of an item that is intended to replace the old version of the item. A variant is a new version of an item that will be added to the configuration without replacing the old version.

2.8.3.1.5 Baseline

A software baseline is a set of software configuration 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 configuration item that has been agreed on. In either case, the baseline can only be changed through formal change control procedures. A baseline, together with all approved changes to the baseline, represents the current approved configuration.

Commonly used baselines 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. Change authority for this baseline typically rests primarily with the development organization, but may be shared with other organizations (for example, SCM or Test). The product baseline corresponds to the completed software product delivered for system integration. The baselines to be used for a given project, along with their associated levels of authority needed for change approval, are typically identified in the SCMP.

2.8.3.1.6 Acquiring software configuration items

Software configuration items are placed under SCM control at different times; that is, they are incorporated into a particular baseline at a particular point in the software life cycle. The triggering event is the completion of some form of formal acceptance task, such as a formal review.

This is an acquisition of items:

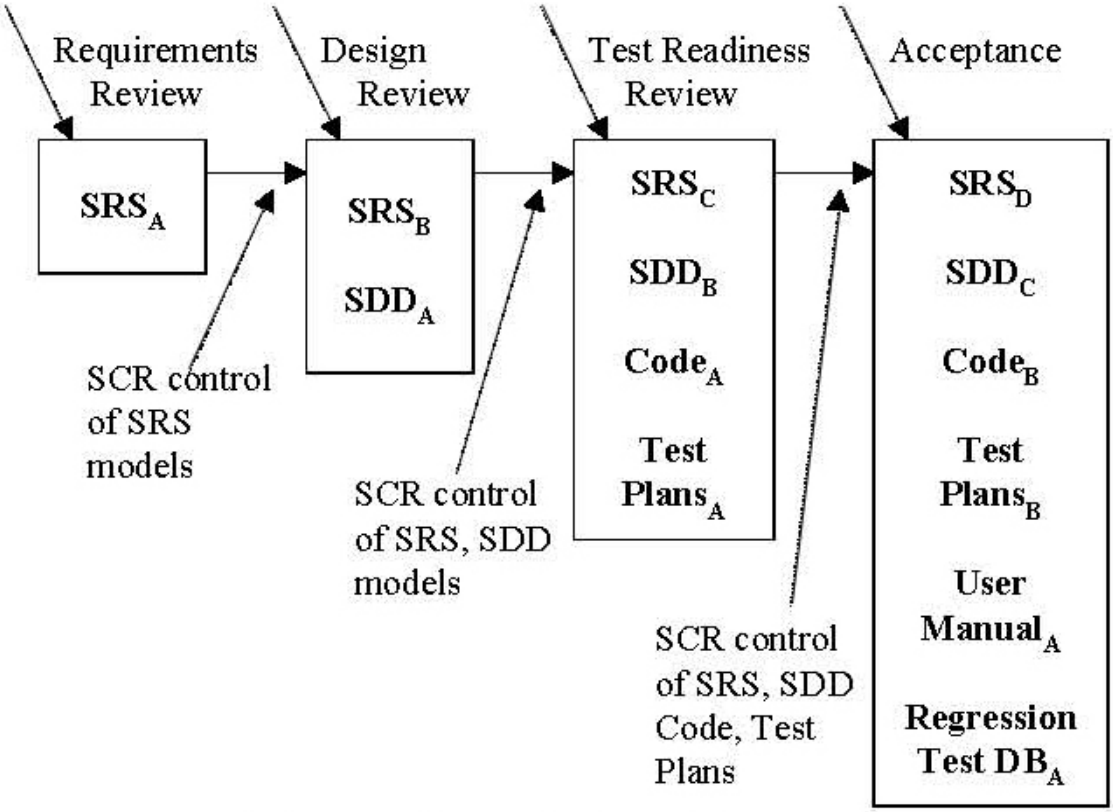


Figure 2.16

Following the acquisition of an SCI, changes to the item must be formally approved as appropriate for the SCI and the baseline involved, as defined in the SCMP. Following approval, the item is incorporated into the software baseline according to the appropriate procedure.

2.8.3.2 Software Library

A software library is a controlled collection of software and related documentation designed to aid in software development, use, and maintenance. It is also instrumental in software release management and delivery activities. Several types of libraries might be used, each corresponding to a particular level of maturity of the software item. For example, a working library could support coding and a project support library could support testing, while a master library could be used for finished products. An appropriate level of SCM control (associated baseline and level of authority for change) is associated with each library. Security, in terms of access control and the backup facilities, is a key aspect of library management.

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

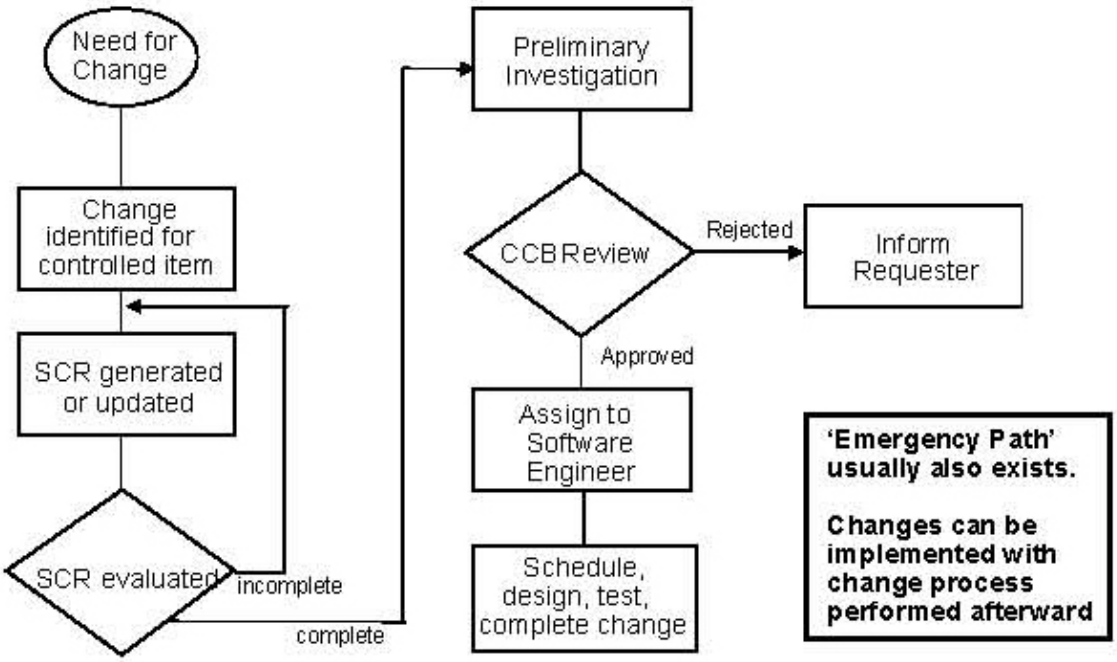
These libraries are also an important source of information for measurements of work and progress.

2.8.4 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 from project requirements, as well as waivers of them. Information derived from these activities is useful in measuring change traffic and breakage, and aspects of rework.

2.8.4.1 Requesting, Evaluating, and Approving Software Changes

The first step in managing changes to controlled items is determining what changes to make. The software change request process provides formal procedures for submitting and recording change requests, evaluating the potential cost and impact of a proposed change, and accepting, modifying, or rejecting the proposed change. Requests for changes to software configuration items may be originated by anyone at any point in the software life cycle and may include a suggested solution and requested priority. One source of change requests is the initiation of corrective action in response to problem reports. Regardless of the source, the type of change (for example, defect or enhancement) is usually recorded on the SCR.



This flow of a Change Control Process provides an opportunity for tracking defects and collecting change activity measurements by change type. Once an SCR is received, a technical evaluation (also known as an impact analysis) is performed to determine the extent of the modifications that would be necessary should the change request be accepted. A good understanding of the relationships among software (and possibly, hardware) items is important for this task. Finally, an established authority, commensurate with the affected baseline, the SCI involved, and the nature of the change, will evaluate the technical and managerial aspects of the change request and either accept, modify, reject, or defer the proposed change.

2.8.4.1.1 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 may actually reside with the leader or an assigned individual rather than a multi-person board. There can be multiple levels of change authority depending on a variety of criteria, such as the criticality of the item involved, the nature of the change (for

example, impact on budget and schedule), or the current point in the life cycle. The composition of the CCBs used for a given system varies depending on these criteria (an SCM representative would always be present). All stakeholders, appropriate to the level of the CCB, are represented. When the scope of authority of a CCB is strictly software, it is known as a Software Configuration Control Board (SCCB). The activities of the CCB are typically subject to software quality audit or review.

2.8.4.1.2 Software change request process

An effective software change request (SCR) process requires the use of supporting tools and procedures ranging from paper forms and a documented procedure to an electronic tool for originating change requests, enforcing the flow of the change process, capturing CCB decisions, and reporting change process information. A link between this tool capability and the problem-reporting system can facilitate the tracking of solutions for reported problems. Change process descriptions and supporting forms (information) are given in a variety of references.

2.8.4.2 Implementing Software Changes

Approved SCRs are implemented using the defined software procedures in accordance with the applicable schedule requirements. Since a number of approved SCRs might be implemented simultaneously, it is necessary to provide a means for tracking which SCRs are incorporated into particular software versions and baselines. As part of the closure of the change process, completed changes may undergo configuration audits and software quality verification. This includes ensuring that only approved changes have been made. The change request process described above will typically document the SCM (and other) approval information for the change.

The actual implementation of a change is supported by the library tool capabilities, which provide version management and code repository support. At a minimum, these tools provide check-in/out and associated version control capabilities. More powerful tools can support parallel development and geographically distributed environments. These tools may be manifested as separate specialized applications under the control of an independent SCM group. They may also appear as an integrated part of the software engineering environment. Finally, they may be as elementary as a rudimentary change control system provided with an operating system.

2.8.4.3 Deviations and Waivers

The constraints imposed on a software engineering effort or the specifications produced during the development activities might contain provisions which 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 from, or waivers of, the provisions.

2.8.5 Software Configuration Status Accounting

Software configuration status accounting (SCSA) is the recording and reporting of information needed for effective management of the software configuration.

2.8.5.1 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 and measurements are needed to support the SCM process and to meet the configuration status reporting needs of management, software engineering, and other related activities. The types of information available include

the approved configuration identification, as well as the identification and current implementation status of changes, deviations, and waivers.

Some form of automated tool support is necessary to accomplish the SCSA data collection and reporting tasks. This could be a database capability, or it could be a standalone tool or a capability of a larger, integrated tool environment.

2.8.5.2 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 software quality activities. Reporting can take the form of ad hoc queries to answer specific questions or the periodic production of predesigned reports. Some information produced by the status accounting activity during the course of the life cycle might become quality assurance records.

In addition to reporting the current status of the configuration, the information obtained by the SCSA can serve as a basis 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.

2.8.6 Software Configuration Auditing

A software audit is an activity performed to independently evaluate the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures. Audits are conducted according to a well-defined process consisting of various auditor roles and responsibilities. Consequently, each audit must be carefully planned. An audit can require a number of individuals to perform a variety of tasks over a fairly short period of time. Tools to support the planning and conduct of an audit can greatly facilitate the process.

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 (for example, in contracts covering critical software): the Functional Configuration Audit (FCA) and the Physical Configuration Audit (PCA). Successful completion of these audits can be a prerequisite for the establishment of the product baseline. Buckley contrasts the purposes of the FCA and PCA in hardware versus software contexts, and recommends careful evaluation of the need for a software FCA and PCA before performing them.

2.8.6.1 Software Functional Configuration Audit

The purpose of the software FCA is to ensure that the audited software item is consistent with its governing specifications. The output of the software verification and validation activities is a key input to this audit.

2.8.6.2 Software Physical Configuration Audit

The purpose of the software physical configuration audit (PCA) is to ensure that the design and reference documentation is consistent with the as-built software product.

2.8.6.3 In-process Audits of a Software Baseline

As mentioned above, audits can be carried out during the development process to investigate the current status of specific elements of the configuration. In this case, an audit could be applied to sampled baseline items to ensure that performance is consistent with specifications or to ensure that evolving documentation continues to be consistent with the developing baseline item.

2.8.7 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 activity. This includes internal releases as well as distribution to customers. When different versions of a software item are available for delivery, such as versions for different platforms or versions with varying capabilities, it is frequently necessary to recreate specific versions and package the correct materials for delivery of the version. The software library is a key element in accomplishing release and delivery tasks.

2.8.7.1 Software Building

Software building is the activity of combining the correct versions of software configuration items, using the appropriate configuration data, into an executable program for delivery to a customer or other recipient, such as the testing activity. For systems with hardware or firmware, the executable program is delivered to the system-building activity. Build instructions ensure that the proper build steps are taken and in the correct sequence. In addition to building software for new releases, it is usually also necessary for SCM to have the capability to reproduce previous releases for recovery, testing, maintenance, or additional release purposes.

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

A tool capability is useful for selecting the correct versions of software items for a given target environment and for automating the process of building the software from the selected versions and appropriate configuration data. For large projects with parallel development or distributed development environments, this tool capability is necessary. Most software engineering environments provide this capability. These tools vary in complexity from requiring the software engineer to learn a specialized scripting language to graphics-oriented approaches that hide much of the complexity of an "intelligent" build facility.

The build process and products are often subject to software quality verification. Outputs of the build process might be needed for future reference and may become quality assurance records.

2.8.7.2 Software Release Management

Software release management encompasses the identification, packaging, and delivery of the elements of a product, for example, executable program, documentation, release notes, and configuration data. Given that product changes can occur on a continuing basis, one concern for release management is determining when to issue a release. The severity of the problems addressed by the release and measurements of the fault densities of prior releases affect this decision. The packaging task must identify which product items are to be delivered, and then select the correct variants of those items, given the intended application of the product. The information documenting the physical contents of a release is known as a version description document. The release notes typically describe new capabilities, known problems, and platform requirements necessary for proper product operation. The package to be released also contains installation or upgrading instructions. The latter can be complicated by the fact that some current users might have versions that are several releases old. Finally, in some cases, the release management activity might be required to track the distribution of the product to various customers or target systems. An example would be a case where the supplier was required to notify a customer of newly reported problems.

A tool capability is needed for supporting these release management functions. It is useful to have a connection with the tool capability supporting the change request process in order to map release contents to the SCRs that have been received. This tool capability might also maintain information on various target platforms and on various customer environments.

2.9 Software quality management[9]

2.9.1 Introduction

What is software quality, and why is it so important that it is pervasive in the Software Engineering Body of Knowledge? Within an information system, software is a tool, and tools have to be selected for quality and for appropriateness. That is the role of equirements. But software is more than a tool. It dictates the performance of the system, and it is therefore important to the system quality.

The notion of "quality" is not as simple as it may seem. For any engineered product, there are many desired qualities relevant to a particular project, to be discussed and determined at the time that the product requirements are determined. Qualities may be present or absent, or may be matters of degree, with tradeoffs among them, with practicality and cost as major considerations. The software engineer has a responsibility to elicit the system's quality requirements that may not be explicit at the outset and to discuss their importance and the difficulty of attaining them. All processes associated with software quality (e.g. building, checking, improving quality) will be designed with these in mind

and carry costs based on the design. Thus, it is important to have in mind some of the possible attributes of quality.

Various researchers have produced models (usually taxonomic) of software quality characteristics or attributes that can be useful for discussing, planning, and rating the quality of software products. The models often include metrics to "measure" the degree of each quality attribute the product attains.

Usually these metrics may be applied at any of the product levels. They are not always direct measures of the quality characteristics of the finished product, but may be relevant to the achievement of overall quality. Each model may have a different set of attributes at the highest level of the taxonomy, and selection of and definitions for the attributes at all levels may differ. The important point is that the system software requirements define the quality requirements and the definitions of the attributes for them.

2.9.2 Software Quality Fundamentals

Agreement on quality requirements, as well as clear communication to the software engineer on what constitutes quality, require that the many aspects of quality be formally defined and discussed.

A software engineer should understand the underlying meanings of quality concepts and characteristics and their value to the software under development or to maintenance.

The important concept is that the software requirements define the required quality characteristics of the software and influence the measurement methods and acceptance criteria for assessing these characteristics.

2.9.2.1 Software Engineering Culture and Ethics

Software engineers are expected to share a commitment to software quality as part of their culture.

Ethics can play a significant role in software quality, the culture, and the attitudes of software engineers. The IEEE Computer Society and the ACM have developed a code of ethics and professional practice based on eight principles to help software engineers reinforce attitudes related to quality and to the independence of their work.

2.9.2.2 Value and Costs of Quality

The notion of "quality" is not as simple as it may seem. For any engineered product, there are many desired qualities relevant to a particular perspective of the product, to be discussed and determined at the time that the product requirements are set down. Quality characteristics may be required or not, or may be required to a greater or lesser degree, and trade-offs may be made among them.

The cost of quality can be differentiated into prevention cost, appraisal cost, internal failure cost, and external failure cost.

[9] This content is available online at <http://cnx.org/content/m28899/1.1/>.

A motivation behind a software project is the desire to create software that has value, and this value may or may not be quantified as a cost. The customer will have some maximum cost in mind, in return for which it is expected that the basic purpose of the software will be fulfilled. The customer may also have some expectation as to the quality of the software. Sometimes customers may not have thought through the quality issues or their related costs. Is the characteristic merely decorative, or is it essential to the software? If the answer lies somewhere in between, as is almost always the case, it is a matter of making the customer a part of the decision process and fully aware of both costs and benefits. Ideally, most of these decisions will be made in the software requirements process, but these issues may arise throughout the software life cycle. There is no definite rule as to how these decisions should be made, but the software engineer should be able to present quality alternatives and their costs.

2.9.2.3 Models and Quality Characteristics

Terminology for software quality characteristics differs from one taxonomy (or model of software quality) to another, each model perhaps having a different number of hierarchical levels and a different total number of characteristics. Various authors have produced models of software quality characteristics or attributes which can be useful for discussing, planning, and rating the quality of software products. ISO/IEC has defined three related models of software product quality (internal quality, external quality, and quality in use) (ISO9126-01) and a set of related parts (ISO14598-98).

2.9.2.3.1 Software engineering process quality

Software quality management and software engineering process quality have a direct bearing on the quality of the software product.

Models and criteria which evaluate the capabilities of software organizations are primarily project organization and management considerations, and, as such, are covered in the Software Engineering Management and Software Engineering Process.

Of course, it is not possible to completely distinguish the quality of the process from the quality of the product.

Process quality influences the quality characteristics of software products, which in turn affect quality-in-use as perceived by the customer.

Two important quality standards are TickIT and one which has an impact on software quality, the ISO9001-00 standard, along with its guidelines for application to software.

Another industry standard on software quality is CMMI. CMMI intends to provide guidance for improving processes. Specific process areas related to quality management are process and product quality assurance, process verification, and process validation. CMMI classifies reviews and audits as methods of verification, and not as specific processes like.

There was initially some debate over whether ISO9001 or CMMI should be used by software engineers to ensure quality. This debate is widely published, and, as a result, the position has been taken that the two are complementary and that having ISO9001 certification can help greatly in achieving the higher maturity levels of the CMMI.

2.9.2.3.2 Software product quality

The software engineer needs, first of all, to determine the real purpose of the software. In this regard, it is of prime importance to keep in mind that the customer's requirements come first and that they include quality requirements, not just functional requirements. Thus, the software engineer has a responsibility to elicit quality requirements which may not be explicit at the outset and to discuss their importance as well as the level of difficulty in attaining them. All processes associated with software quality (for example, building, checking, and improving quality) will be designed with these requirements in mind, and they carry additional costs.

Standard ISO9126-01 defines, for two of its three models of quality, the related quality characteristics and sub-characteristics, and measures which are useful for assessing software product quality.

The meaning of the term "product" is extended to include any artifact which is the output of any process used to build the final software product. Examples of a product include, but are not limited to, an entire system requirements specification, a software requirements specification for a software component of a system, a design module, code, test documentation, or reports produced as a result of quality analysis tasks. While most treatments of quality are described in terms of the final software and system performance, sound engineering practice requires that intermediate products relevant to quality be evaluated throughout the software engineering process.

2.9.2.4 Quality Improvement

The quality of software products can be improved through an iterative process of continuous improvement which requires management control, coordination, and feedback from many concurrent processes: the software life cycle processes; the process of error/defect detection, removal, and prevention; and the quality improvement process.

The theory and concepts behind quality improvement, such as building in quality through the prevention and early detection of errors, continuous improvement, and customer focus, are pertinent to software engineering. These concepts are based on the work of experts in quality who have stated that the quality of a product is directly linked to the quality of the process used to create it.

Approaches such as the Total Quality Management (TQM) process of Plan, Do, Check, and Act (PDCA) are tools by which quality objectives can be met. Management sponsorship supports process and product evaluations and the resulting findings. Then, an improvement program is developed identifying detailed actions and improvement projects to be addressed in a feasible time frame. Management support implies that each improvement project has enough resources to achieve the goal defined for it. Management sponsorship must be solicited frequently by implementing proactive communication activities. The involvement of work groups, as well as middle-management support and resources allocated at project level.

2.9.3 Software Quality Management Processes

Software quality management (SQM) applies to all perspectives of software processes, products, and resources. It defines processes, process owners, and requirements for those processes, measurements of the process and its outputs, and feedback channels. Software quality management processes consist of many activities. Some may find defects directly, while others indicate where further examination may be valuable. The latter are also referred to as direct-defect-finding activities. Many activities often serve as both.

Planning for software quality involves:

- Defining the required product in terms of its quality characteristics.
- Planning the processes to achieve the required product.

These aspects differ from, for instance, the planning SQM processes themselves, which assess planned quality characteristics versus actual implementation of those plans. The software quality management processes must address how well software products will, or do, satisfy customer and stakeholder requirements, provide value to the customers and other stakeholders, and provide the software quality needed to meet software requirements.

SQM can be used to evaluate the intermediate products as well as the final product.

Some of the specific SQM processes are defined in standard (IEEE12207.0-96):

- Quality assurance process
- Verification process
- Validation process
- Review process
- Audit process

These processes encourage quality and also find possible problems. But they differ somewhat in their emphasis.

SQM processes help ensure better software quality in a given project. They also provide, as a by-product, general information to management, including an indication of the quality of the entire software engineering process. The Software Engineering Process and Software Engineering Management KAs discuss quality programs for the organization developing the software. SQM can provide relevant feedback for these areas.

SQM processes consist of tasks and techniques to indicate how software plans (for example, management, development, configuration management) are being implemented and how well the intermediate and final products are meeting their specified requirements. Results from these tasks are assembled in reports for management before corrective action is taken. The management of an SQM process is tasked with ensuring that the results of these reports are accurate.

As described in this KA, SQM processes are closely related; they can overlap and are sometimes even combined. They seem largely reactive in nature because they address the processes as practiced and the products as produced; but they have a major role at the planning stage in being proactive in terms of the processes and procedures needed to attain the quality characteristics and degree of quality needed by the stakeholders in the software.

Risk management can also play an important role in delivering quality software. Incorporating disciplined risk analysis and management techniques into the software life cycle processes can increase the potential for producing a quality product.

2.9.3.1 Software Quality Assurance

SQA processes provide assurance that the software products and processes in the project life cycle conform to their specified requirements by planning, enacting, and performing a set of activities to provide adequate confidence that quality is being built into the software. This means ensuring that the problem is clearly and adequately stated and that the solution's requirements are properly defined and expressed. SQA seeks to maintain the quality throughout the development and maintenance of the product by the execution of a variety of activities at each stage which can result in early identification of problems, an almost inevitable feature of any complex activity. The role of SQA with respect to process is to ensure that planned processes are appropriate and later implemented according to plan, and that relevant measurement processes are provided to the appropriate organization.

The SQA plan defines the means that will be used to ensure that software developed for a specific product satisfies the user's requirements and is of the highest quality possible within project constraints. In order to do so, it must first ensure that the quality target is clearly defined and understood. It must consider management, development, and maintenance plans for the software. Refer to standard (IEEE730-98) for details.

The specific quality activities and tasks are laid out, with their costs and resource requirements, their overall management objectives, and their schedule in relation to those objectives in the software engineering management, development, or maintenance plans. The SQA plan should be consistent with the software configuration management plan. The SQA plan identifies documents, standards, practices, and conventions governing the project and how they will be checked and monitored to ensure adequacy and compliance. The SQA plan also identifies measures, statistical techniques, procedures for problem reporting and corrective action, resources such as tools, techniques, and methodologies, security for physical media, training, and SQA reporting and documentation. Moreover, the SQA plan addresses the software quality assurance activities of any other type of activity described in the software plans, such as procurement of supplier software to the project or commercial off-the-shelf software (COTS) installation, and service after delivery of the software. It can also contain acceptance criteria as well as reporting and management activities which are critical to software quality.

2.9.3.2 Verification & Validation

For purposes of brevity, Verification and Validation (V&V) are treated as a single topic in this Guide rather than as two separate topics as in the standard (IEEE12207.0-96). "Software V&V is a disciplined approach to assessing software products throughout the product life cycle. A V&V effort strives to ensure that quality is built into the software and that the software satisfies user requirements" (IEEE1059-93).

V&V addresses software product quality directly and uses testing techniques which can locate defects so that they can be addressed. It also assesses the intermediate products, however, and, in this capacity, the intermediate steps of the software life cycle processes.

The V&V process determines whether or not products of a given development or maintenance activity conform to the requirement of that activity, and whether or not the final software product fulfills its intended purpose and meets user requirements. Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Validation is an attempt to ensure that the right product is built, that is, the product fulfills its specific intended purpose. Both the verification process and the validation process begin early in the development or maintenance phase. They provide an examination of key product features in relation both to the product's immediate predecessor and to the specifications it must meet.

The purpose of planning V&V is to ensure that each resource, role, and responsibility is clearly assigned. The resulting V&V plan documents and describes the various resources and their roles and activities, as well as the techniques and tools to be used. An understanding of the different purposes of each V&V activity will help in the careful planning of the techniques and resources needed to fulfill their purposes.

The plan also addresses the management, communication, policies, and procedures of the V&V activities and their interaction, as well as defect reporting and documentation requirements.

2.9.3.3 Reviews and Audits

For purposes of brevity, reviews and audits are treated as a single topic in this Guide, rather than as two separate topics as in (IEEE12207.0-96). The review and audit process is broadly defined in (IEEE12207.0-96) and in more detail in (IEEE1028-97). Five types of reviews or audits are presented in the IEEE1028-97 standard:

- Management reviews
- Technical reviews
- Inspections
- Walk-throughs
- Audits

2.9.3.3.1 Management reviews

The purpose of a management review is to monitor progress, determine the status of plans and schedules, confirm requirements and their system allocation, or evaluate the effectiveness of management approaches used to achieve fitness for purpose. They support decisions about changes and corrective actions that are required during a software project. Management reviews determine the adequacy of plans, schedules, and requirements and monitor their progress or inconsistencies. These reviews may be performed on products such as audit reports, progress reports, V&V reports, and plans of many types, including risk management, project management, software configuration management, software safety, and risk assessment, among others.

2.9.3.3.2 Technical reviews

"The purpose of a technical review is to evaluate a software product to determine its suitability for its intended use. The objective is to identify discrepancies from approved specifications and standards. The results should provide management with evidence confirming (or not) that the product meets the specifications and adheres to standards, and that changes are controlled" (IEEE1028-97).

Specific roles must be established in a technical review: a decision-maker, a review leader, a recorder, and technical staff to support the review activities. A technical review requires that mandatory inputs be in place in order to proceed:

- Statement of objectives
- A specific software product
- The specific project management plan
- The issues list associated with this product
- The technical review procedure

The team follows the review procedure. A technically qualified individual presents an overview of the product, and the examination is conducted during one or more meetings. The technical review is completed once all the activities listed in the examination have been completed.

2.9.3.3.3 Inspections

The purpose of an inspection is to detect and identify software product anomalies. Two important differentiators of inspections as opposed to reviews are as follows:

- An individual holding a management position over any member of the inspection team shall not participate in the inspection.
- An inspection is to be led by an impartial facilitator who is trained in inspection techniques.

Software inspections always involve the author of an intermediate or final product, while other reviews might not. Inspections also include an inspection leader, a recorder, a reader, and a few (2 to 5) inspectors. The members of an inspection team may possess different expertise, such as domain expertise, design method expertise, or language expertise. Inspections are usually conducted on one relatively small section of the product at a time. Each team member must examine the software product and other review inputs prior to the review meeting, perhaps by applying an analytical technique to a small section of the product, or to the entire product with a focus only on one aspect, for example, interfaces. Any anomaly found is documented and sent to the inspection leader. During the inspection, the inspection leader conducts the session and verifies that everyone has prepared for the inspection. A checklist, with anomalies and questions germane to the issues of interest, is a common tool used in inspections. The resulting list often classifies the anomalies and is reviewed for completeness and accuracy by the team. The inspection exit decision must correspond to one of the following three criteria:

- Accept with no or at most minor reworking
- Accept with rework verification
- Reinspect

Inspection meetings typically last a few hours, whereas technical reviews and audits are usually broader in scope and take longer.

2.9.3.3.4 Walk-throughs

The purpose of a walk-through is to evaluate a software product. A walk-through may be conducted for the purpose of educating an audience regarding a software product. The major objectives are to:

- Find anomalies
- Improve the software product
- Consider alternative implementations
- Evaluate conformance to standards and specifications

The walk-through is similar to an inspection but is typically conducted less formally. The walk-through is primarily organized by the software engineer to give his teammates the opportunity to review his work, as an assurance technique.

2.9.3.3.5 Audits

The purpose of a software audit is to provide an independent evaluation of the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures. The audit is a formally organized activity, with participants having specific roles, such as lead auditor, another auditor, a recorder, or an initiator, and includes a representative of the audited organization. The audit will identify instances of nonconformance and produce a report requiring the team to take corrective action.

While there may be many formal names for reviews and audits such as those identified in the standard (see IEEE1028- 97), the important point is that they can occur on almost any product at any stage of the development or maintenance process.

2.9.4 Practical Considerations

2.9.4.1 Software Quality Requirements

2.9.4.1.1 Influence factors

Various factors influence planning, management, and selection of SQM activities and techniques, including:

- The domain of the system in which the software will reside (safety-critical, mission-critical, business-critical)
- System and software requirements
- The commercial (external) or standard (internal) components to be used in the system
- The specific software engineering standards applicable
- The methods and software tools to be used for development and maintenance and for quality evaluation and improvement
- The budget, staff, project organization, plans, and scheduling of all the processes
- The intended users and use of the system
- The integrity level of the system

Information on these factors influences how the SQM processes are organized and documented, how specific SQM activities are selected, what resources are needed, and which will impose bounds on the efforts.

2.9.4.1.2 Dependability

In cases where system failure may have extremely severe consequences, overall dependability (hardware, software, and human) is the main quality requirement over and above basic functionality. Software dependability includes such characteristics as fault tolerance, safety, security, and usability. Reliability is also a criterion which can be defined in terms of dependability (ISO9126).

The body of literature for systems must be highly dependable ("high confidence" or "high integrity systems"). Terminology for traditional mechanical and electrical systems which may not include software has been imported for discussing threats or hazards, risks, system integrity, and related concepts, and may be found in the references cited for this section.

2.9.4.1.3 Integrity levels of software

The integrity level is determined based on the possible consequences of failure of the software and the probability of failure. For software in which safety or security is important, techniques such as hazard analysis for safety or threat analysis for security may be used to develop a planning activity which would identify where potential trouble spots lie. The failure history of similar software may also help in identifying which techniques will be most useful in detecting faults and assessing quality.

2.9.4.2 Defect Characterization

SQM processes find defects. Characterizing those defects leads to an understanding of the product, facilitates corrections to the process or the product, and informs project management or the customer of the status of the process or product. Many defect (fault) taxonomies exist, and, while attempts have been made to gain consensus on a fault and failure taxonomy, the literature indicates. Defect (anomaly) characterization is also used in audits and reviews, with the review leader often presenting a list of anomalies provided by team members for consideration at a review meeting.

As new design methods and languages evolve, along with advances in overall software technologies, new classes of defects appear, and a great deal of effort is required to interpret previously defined classes. When tracking defects, the software engineer is interested in not only the number of defects but also the types. Information alone, without some classification, is not really of any use in identifying the underlying causes of the defects, since specific types of problems need to be grouped together in order for determinations to be made about them. The point is to establish a defect taxonomy that is meaningful to the organization and to the software engineers.

SQM discovers information at all stages of software development and maintenance. Typically, where the word "defect" is used, it refers to a "fault" as defined below. However, different cultures and standards may use somewhat different meanings for these terms, which have led to attempts to define them. Partial definitions taken from standard (IEEE610.12-90) are:

- 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"
- Failures found in testing as a result of software faults are included as defects in the discussion in this section. Reliability models are built from failure data collected during software testing or from software in service, and thus can be used to predict future failures and to assist in decisions on when to stop testing.

One probable action resulting from SQM findings is to remove the defects from the product under examination. Other actions enable the achievement of full value from the findings of SQM activities. These actions include analyzing and summarizing the findings, and using measurement techniques to improve the product and the process as well as to track the defects and their removal.

Data on the inadequacies and defects found during the implementation of SQM techniques may be lost unless they are recorded. For some techniques (for example, technical reviews, audits, inspections), recorders are present to set down such information, along with issues and decisions. When automated tools are used, the tool output may provide the defect information. Data about defects may be collected and recorded on an SCR (software change request) form and may subsequently be entered into some type of database, either manually or automatically, from an analysis tool. Reports about defects are provided to the management of the organization.

2.9.4.3 Software Quality Management Techniques

SQM techniques can be categorized in many ways: static, people-intensive, analytical, dynamic.

2.9.4.3.1 Static techniques

Static techniques involve examination of the project documentation and software, and other information about the software products, without executing them. These techniques may include people-intensive activities or analytical activities conducted by individuals, with or without the assistance of automated tools.

2.9.4.3.2 People-intensive techniques

The setting for people-intensive techniques, including reviews and audits, may vary from a formal meeting to an informal gathering or a desk-check situation, but (usually, at least) two or more people are involved. Preparation ahead of time may be necessary. Resources other than the items under examination may include checklists and results from analytical techniques and testing. These activities are discussed in (IEEE1028-97) on reviews and audits.

2.9.4.3.3 Analytical techniques

A software engineer generally applies analytical techniques. Sometimes several software engineers use the same technique, but each applies it to different parts of the product. Some techniques are tool-driven; others are manual. Some may find defects directly, but they are typically used to support other techniques. Some also include various assessments as part of overall quality analysis. Examples of such techniques include complexity analysis, control flow analysis, and algorithmic analysis.

Each type of analysis has a specific purpose, and not all types are applied to every project. An example of a support technique is complexity analysis, which is useful for determining whether or not the design or implementation is too complex to develop correctly, to test, or to maintain. The results of a complexity analysis may also be used in developing test cases. Defect-finding techniques, such as control flow analysis, may also be used to support another activity. For software with many algorithms, algorithmic analysis is important, especially when an incorrect algorithm could cause a catastrophic result. There are too many analytical techniques to list them all here. The list and references provided may offer insights into the selection of a technique, as well as suggestions for further reading.

Other, more formal, types of analytical techniques are known as formal methods. They are used to verify software requirements and designs. Proof of correctness applies to critical parts of software. They have mostly been used in the verification of crucial parts of critical systems, such as specific security and safety requirements.

2.9.4.3.4 Dynamic techniques

Different kinds of dynamic techniques are performed throughout the development and maintenance of software. Generally, these are testing techniques, but techniques such as simulation, model checking, and symbolic execution may be considered dynamic. Code reading is considered a static technique, but experienced software engineers may execute the code as they read through it. In this sense, code reading may also qualify as a dynamic technique. This discrepancy in categorizing indicates that people with different roles in the organization may consider and apply these techniques differently.

Some testing may thus be performed in the development process, SQA process, or V&V process, again depending on project organization. Because SQM plans address testing, this section includes some comments on testing.

2.9.4.3.5 Testing

The assurance processes described in SQA and V&V examine every output relative to the software requirement specification to ensure the output's traceability, consistency, completeness, correctness, and performance. This confirmation also includes the outputs of the development and maintenance processes, collecting, analyzing, and measuring the results. SQA ensures that appropriate types of tests are planned, developed, and implemented, and V&V develops test plans, strategies, cases, and procedures.

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

- Evaluation and test of tools to be used on the project (IEEE1462-98)
- Conformance test (or review of conformance test) of components and COTS products to be used in the product; there now exists a standard for software packages (IEEE1465-98)

Sometimes an independent V&V organization may be asked to monitor the test process and sometimes to witness the actual execution to ensure that it is conducted in accordance with specified procedures. Again, V&V may be called upon to evaluate the testing itself: adequacy of plans and procedures, and adequacy and accuracy of results.

Another type of testing that may fall under the heading of V&V organization is third-party testing. The third party is not the developer, nor is in any way associated with the development of the product. Instead, the third party is an independent facility, usually accredited by some body of authority. Their purpose is to test a product for conformance to a specific set of requirements.

2.9.4.4 Software Quality Measurement

The models of software product quality often include measures to determine the degree of each quality characteristic attained by the product.

If they are selected properly, measures can support software quality (among other aspects of the software life cycle processes) in multiple ways. They can help in the management decision-making process. They can find problematic areas and bottlenecks in the software process; and they can help the software engineers assess the quality of their work for SQA purposes and for longer-term process quality improvement.

With the increasing sophistication of software, questions of quality go beyond whether or not the software works to how well it achieves measurable quality goals.

There are a few more topics where measurement supports SQM directly. These include assistance in deciding when to stop testing. For this, reliability models and benchmarks, both using fault and failure data, are useful.

The cost of SQM processes is an issue which is almost always raised in deciding how a project should be organized. Often, generic models of cost are used, which are based on when a defect is found and how much effort it takes to fix the defect relative to finding the defect earlier in the development process. Project data may give a better picture of cost.

Finally, the SQM reports themselves provide valuable information not only on these processes, but also on how all the software life cycle processes can be improved.

While the measures for quality characteristics and product features may be useful in themselves (for example, the number of defective requirements or the proportion of defective requirements), mathematical and graphical techniques can be applied to aid in the interpretation of the measures. These fit into the following categories:

- Statistically based (for example, Pareto analysis, runcharts, scatter plots, normal distribution)
- Statistical tests (for example, the binomial test, chi-squared test)
- Trend analysis
- Prediction (for example, reliability models)

The statistically based techniques and tests often provide a snapshot of the more troublesome areas of the softwareproduct under examination. The resulting charts andgraphs are visualization aids which the decision-makerscan use to focus resources where they appear most needed. Results from trend analysis may indicate that a schedule has not been respected, such as in testing, or that certain classes of faults will become more intense unless some corrective action is taken in development. The predictive techniques assist in planning test time and in predicting failure.

References:

http://en.wikipedia.org/wiki/Software_quality_assurance, http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-171Fall2003/CourseHome/, http://www.cs.cornell.edu/courses/cs501/2008sp/, http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/, http://www.ee.unb.ca/kengleha/courses/CMPE3213/IntroT http://www.softwarecertifications.org/qai_cmsq.htm, http://satc.gsfc.nasa.gov/assure/agbsec3.txt, etc...

2.10 Software engineering management[10]

2.10.1 Introduction

Software Engineering Management can be defined as the application of management activities - planning, coordinating, measuring, monitoring, controlling, and reporting - to ensure that the development and maintenance of software is systematic, disciplined, and quantified.

The software engineering management therefore addresses the management and measurement of software engineering. While measurement is an important aspect of all topics of software engineering, it is here that the topic of measurement programs is presented.

While it is true to say that in one sense it should be possible to manage software engineering in the same way as any other process, there are aspects specific to software products and the software life cycle processes which complicate effective management - just a few of which are as follows:

- The perception of clients is such that there is often a lack of appreciation for the complexity inherent in software engineering, particularly in relation to the impact of changing requirements.
- It is almost inevitable that the software engineering processes themselves will generate the need for new or changed client requirements.
- As a result, software is often built in an iterative process rather than a sequence of closed tasks.
- Software engineering necessarily incorporates aspects of creativity and discipline—maintaining an appropriate balance between the two is often difficult.
- The degree of novelty and complexity of software is often extremely high.
- There is a rapid rate of change in the underlying technology.

With respect to software engineering, management activities occur at three levels: organizational and infrastructure management, project management, and measurement program planning and control.

Aspects of organizational management are important in terms of their impact on software engineering - on policy management, for instance: organizational policies and standards provide the framework in which software engineering is undertaken. These policies may need to be influenced by the requirements of effective software development and maintenance, and a number of software engineering-specific policies may need to be established for effective management of software engineering at an organizational level. For example, policies are usually necessary to establish specific organization-wide processes or procedures for such software engineering tasks as designing, implementing, estimating, tracking, and reporting. Such policies are essential to effective long-term software engineering management, by establishing a consistent basis on which to analyze past performance and implement improvements, for example.

Another important aspect of management is personnel management: policies and procedures for hiring, training, and motivating personnel and mentoring for career development are important not only at the project level but also to the longer-term success of an organization. Software engineering personnel may present unique training or personnel management challenges (for example, maintaining currency in a context where the underlying technology undergoes continuous and rapid change). Communication management is also often mentioned as an overlooked but major aspect of the performance of individuals in a field where precise understanding of user needs and of complex requirements and designs is necessary. Finally, portfolio management, which is the capacity to have an overall vision not only of the set of software under development but also of the software already in use in an organization, is necessary. Furthermore, software reuse is a key factor in maintaining and improving productivity and competitiveness. Effective reuse requires a strategic vision that reflects the unique power and requirements of this technique.

Organizational culture and behavior, and functional enterprise management in terms of procurement, supply chain management, marketing, sales, and distribution, all have an influence, albeit indirectly, on an organization's software engineering process.

The Software Engineering Management consists of both the software project management process, in its first five subareas, and software engineering measurement in the last subarea. While these two subjects are

[10] This content is available online at <http://cnx.org/content/m28902/1.1/>.

often regarded as being separate, and indeed they do possess many unique aspects, their close relationship has led to their combined treatment in software engineering. Unfortunately, a common perception of the software industry is that it delivers products late, over budget, and of poor quality and uncertain functionality. Measurement-informed management - an assumed principle of any true engineering discipline - can help to turn this perception around. In essence, management without measurement, qualitative and quantitative, suggests a lack of rigor, and measurement without management suggests a lack of purpose or context. In the same way, however, management and measurement without expert knowledge is equally ineffectual, so we must be careful to avoid over-emphasizing the quantitative aspects of Software Engineering Management (SEM). Effective management requires a combination of both numbers and experience.

The following working definitions are adopted here:

- Management process refers to the activities that are undertaken in order to ensure that the software engineering processes are performed in a manner consistent with the organization's policies, goals, and standards.
- Measurement refers to the assignment of values and labels to aspects of software engineering (products, processes, and resources) and the models that are derived from them, whether these models are developed using statistical, expert knowledge or other techniques.
- The software engineering project management subareas make extensive use of the software engineering measurement subarea.
- Software Requirements, where some of the activities to be performed during the Initiation and Scope definition phase of the project are described
- Software Configuration Management, as this deals with the identification, control, status accounting, and audit of the software configuration along with software release management and delivery
- Software Engineering Process, because processes and projects are closely related.
- Software Quality, as quality is constantly a goal of management and is an aim of many activities that must be managed.

2.10.2 Topics for software engineering management

As the Software Engineering Management is viewed here as an organizational process which incorporates the notion of process and project management, we have created a breakdown that is both topic-based and life cycle-based. However, the primary basis for the top-level breakdown is the process of managing a software engineering project. There are six major subareas:

- Initiation and scope definition, which deals with the decision to initiate a software engineering project
- Software project planning, which addresses the activities undertaken to prepare for successful software engineering from a management perspective
- Software project enactment, which deals with generally accepted software engineering management activities that occur during software engineering
- Review and evaluation, which deal with assurance that the software is satisfactory
- Closure, which addresses the post-completion activities of a software engineering project
- Software engineering measurement, which deals with the effective development and implementation of measurement programs in software engineering organizations (IEEE12207.0-96)

2.10.2.1 Initiation and Scope Definition

The focus of this set of activities is on the effective determination of software requirements via various elicitation methods and the assessment of the project's feasibility from a variety of standpoints. Once feasibility has been established, the remaining task within this process is the specification of requirements validation and change procedures.

2.10.2.1.1 Determination and Negotiation of Requirements

Software requirement methods for requirements elicitation (for example, observation), analysis (for example, data modeling, use-case modeling), specification, and validation (for example, prototyping) must be selected and applied, taking into account the various stakeholder perspectives. This leads to the determination of project scope, objectives, and constraints. This is always an important activity, as it sets the visible boundaries for the set of tasks being undertaken, and is particularly so where the novelty of the undertaking is high.

2.10.2.1.2 Feasibility Analysis

Software engineers must be assured that adequate capability and resources are available in the form of people, expertise, facilities, infrastructure, and support (either internally or externally) to ensure that the project can be successfully completed in a timely and cost-effective manner (using, for example, a requirement-capability matrix). This often requires some "ballpark" estimation of effort and cost based on appropriate methods (for example, expert-informed analogy techniques).

2.10.2.1.3 Process for the Review and Revision of Requirements

Given the inevitability of change, it is vital that agreement among stakeholders is reached at this early point as to the means by which scope and requirements are to be reviewed and revised (for example, via agreed change management procedures). This clearly implies that scope and requirements will not be "set in stone" but can and should be revisited at predetermined points as the process unfolds (for example, at design reviews, management reviews). If changes are accepted, then some form of traceability analysis and risk analysis should be used to ascertain the impact of those changes. A managed-change approach should also be useful when it comes time to review the outcome of the project, as the scope and requirements should form the basis for the evaluation of success.

2.10.2.2 Software Project Planning

The iterative planning process is informed by the scope and requirements and by the establishment of feasibility. At this point, software life cycle processes are evaluated and the most appropriate (given the nature of the project, its degree of novelty, its functional and technical complexity, its quality requirements, and so on) is selected. Where relevant, the project itself is then planned in the form of a hierarchical decomposition of tasks, the associated deliverables of each task are specified and characterized in terms of quality and other attributes in line with stated requirements, and detailed effort, schedule, and cost estimation is undertaken. Resources are then allocated to tasks so as to optimize personnel productivity (at individual, team, and organizational levels), equipment and materials utilization, and adherence to schedule. Detailed risk management is undertaken and the "risk profile" of the project is discussed among, and accepted by, all relevant stakeholders. Comprehensive software quality management processes are determined as part of the planning process in the form of procedures and responsibilities for software quality assurance, verification and validation, reviews, and audits. As an iterative process, it is vital that the processes and responsibilities for ongoing plan management, review, and revision are also clearly stated and agreed.

2.10.2.2.1 Process Planning

Selection of the appropriate software life cycle model (for example, spiral, evolutionary prototyping) and the adaptation and deployment of appropriate software life cycle processes are undertaken in light of the particular scope and requirements of the project. Relevant methods and tools are also selected. At the project level, appropriate methods and tools are used to decompose the project into tasks, with associated inputs, outputs, and completion conditions (for example, work breakdown structure). This in turn influences decisions on the project's high-level schedule and organization structure.

2.10.2.2.2 Determine Deliverables

The product(s) of each task (for example, architectural design, inspection report) are specified and characterized. Opportunities to reuse software components from previous developments or to utilize off-the-shelf software products are evaluated. Use of third parties and procured software are planned and suppliers are selected.

2.10.2.2.3 Effort, Schedule, and Cost Estimation

Based on the breakdown of tasks, inputs, and outputs, the expected effort range required for each task is determined using a calibrated estimation model based on historical size-effort data where available and relevant, or other methods like expert judgment. Task dependencies are established and potential bottlenecks are identified using suitable methods (for example, critical path analysis). Bottlenecks are resolved where possible, and the expected schedule of tasks with projected start times, durations, and end times is produced. Resource requirements (people, tools) are translated into cost estimates. This is a highly iterative activity which must be negotiated and revised until consensus is reached among affected stakeholders (primarily engineering and management).

2.10.2.2.4 Resource Allocation

Equipment, facilities, and people are associated with the scheduled tasks, including the allocation of responsibilities for completion. This activity is informed and constrained by the availability of resources and their optimal use under these circumstances, as well as by issues relating to personnel (for example, productivity of individuals/teams, team dynamics, organizational and team structures).

2.10.2.2.5 Risk Management

Risk identification and analysis (what can go wrong, how and why, and what are the likely consequences), critical risk assessment (which are the most significant risks in terms of exposure, which can we do something about in terms of leverage), risk mitigation and contingency planning (formulating a strategy to deal with risks and to manage the risk profile) are all undertaken. Risk assessment methods (for example, decision trees and process simulations) should be used in order to highlight and evaluate risks. Project abandonment policies should also be determined at this point in discussion with all other stakeholders. Software-unique aspects of risk, such as software engineers' tendency to add unwanted features or the risks attendant in software's intangible nature, must influence the project's risk management.

2.10.2.2.6 Quality Management

Quality is defined in terms of pertinent attributes of the specific project and any associated product(s), perhaps in both quantitative and qualitative terms. These quality characteristics will have been determined in the specification of detailed software requirements.

Thresholds for adherence to quality are set for each indicator as appropriate to stakeholder expectations for the software at hand. Procedures relating to ongoing SQA throughout the process and for product (deliverable) verification and validation are also specified at this stage (for example, technical reviews and inspections).

2.10.2.2.7 Plan Management

How the project will be managed and how the plan will be managed must also be planned. Reporting, monitoring, and control of the project must fit the selected software engineering process and the realities of the project, and must be reflected in the various artifacts that will be used for managing it. But, in an environment where change is an expectation rather than a shock, it is vital that plans are themselves managed. This requires that adherence to plans be systematically directed, monitored, reviewed, reported, and,

where appropriate, revised. Plans associated with other management-oriented support processes (for example, documentation, software configuration management, and problem resolution) also need to be managed in the same manner.

2.10.2.3 Software Project Enactment

The plans are then implemented, and the processes embodied in the plans are enacted. Throughout, there is a focus on adherence to the plans, with an overriding expectation that such adherence will lead to the successful satisfaction of stakeholder requirements and achievement of the project objectives. Fundamental to enactment are the ongoing management activities of measuring, monitoring, controlling, and reporting.

2.10.2.3.1 Implementation of Plans

The project is initiated and the project activities are undertaken according to the schedule. In the process, resources are utilized (for example, personnel effort, funding) and deliverables are produced (for example, architectural design documents, test cases).

2.10.2.3.2 Supplier Contract Management

Prepare and execute agreements with suppliers, monitor supplier performance, and accept supplier products, incorporating them as appropriate.

2.10.2.3.3 Implementation of measurement process

The measurement process is enacted alongside the software project, ensuring that relevant and useful data are collected.

2.10.2.3.4 Monitor Process

Adherence to the various plans is assessed continually and at predetermined intervals. Outputs and completion conditions for each task are analyzed. Deliverables are evaluated in terms of their required characteristics (for example, via reviews and audits). Effort expenditure, schedule adherence, and costs to date are investigated, and resource usage is examined. The project risk profile is revisited, and adherence to quality requirements is evaluated.

Measurement data are modeled and analyzed. Variance analysis based on the deviation of actual from expected outcomes and values is undertaken. This may be in the form of cost overruns, schedule slippage, and the like. Outlier identification and analysis of quality and other measurement data are performed (for example, defect density analysis). Risk exposure and leverage are recalculated, and decisions trees, simulations, and so on are rerun in the light of new data. These activities enable problem detection and exception identification based on exceeded thresholds. Outcomes are reported as needed and certainly where acceptable thresholds are surpassed.

2.10.2.3.5 Control Process

The outcomes of the process monitoring activities provide the basis on which action decisions are taken. Where appropriate, and where the impact and associated risks are modeled and managed, changes can be made to the project. This may take the form of corrective action (for example, retesting certain components), it may involve the incorporation of contingencies so that similar occurrences are avoided (for example, the decision to use prototyping to assist in software requirements validation), and/or it may entail the revision of the various plans and other project documents (for example, requirements specification) to accommodate the unexpected outcomes and their implications.

2.10.2.3.6 Reporting

At specified and agreed periods, adherence to the plans is reported, both within the organization (for example to the project portfolio steering committee) and to external stakeholders (for example, clients, users). Reports of this nature should focus on overall adherence as opposed to the detailed reporting required frequently within the project team.

2.10.2.4 Review and Evaluation

At critical points in the project, overall progress towards achievement of the stated objectives and satisfaction of stakeholder requirements are evaluated. Similarly, assessments of the effectiveness of the overall process to date, the personnel involved, and the tools and methods employed are also undertaken at particular milestones.

2.10.2.4.1 Determining Satisfaction of Requirements

Since attaining stakeholder (user and customer) satisfaction is one of our principal aims, it is important that progress towards this aim be formally and periodically assessed. This occurs on achievement of major project milestones (for example, confirmation of software design architecture, software integration technical review). Variances from expectations are identified and appropriate action is taken.

2.10.2.4.2 Reviewing and Evaluating Performance

Periodic performance reviews for project personnel provide insights as to the likelihood of adherence to plans as well as possible areas of difficulty (for example, team member conflicts). The various methods, tools, and techniques employed are evaluated for their effectiveness and appropriateness, and the process itself is systematically and periodically assessed for its relevance, utility, and efficacy in the project context. Where appropriate, changes are made and managed.

2.10.2.5 Closure

The project reaches closure when all the plans and embodied processes have been enacted and completed. At this stage, the criteria for project success are revisited. Once closure is established, archival, post mortem, and process improvement activities are performed.

2.10.2.5.1 Determining Closure

The tasks as specified in the plans are complete, and satisfactory achievement of completion criteria is confirmed. All planned products have been delivered with acceptable characteristics. Requirements are checked off and confirmed as satisfied, and the objectives of the project have been achieved. These processes generally involve all stakeholders and result in the documentation of client acceptance and any remaining known problem reports.

2.10.2.5.2 Closure Activities

After closure has been confirmed, archival of project materials takes place in line with stakeholder-agreed methods, location, and duration. The organization's measurement database is updated with final project data and post-project analyses are undertaken. A project post mortem is undertaken so that issues, problems, and opportunities encountered during the process are analyzed, and lessons are drawn from the process and fed into organizational learning and improvement .

2.10.3 Software Engineering Measurement

The importance of measurement and its role in better management practices is widely acknowledged, and so its importance can only increase in the coming years. Effective measurement has become one of the cornerstones of organizational maturity.

Key terms on software measures and measurement methods have been defined in ISO15939 on the basis of the ISO international vocabulary of metrology ISO93. Nevertheless, readers will encounter terminology differences in the literature; for example, the term "metrics" is sometimes used in place of "measures."

This topic follows the international standard ISO/IEC 15939, which describes a process which defines the activities and tasks necessary to implement a software measurement process and includes, as well, a measurement information model.

2.10.3.1 Establish and Sustain Measurement Commitment

- Accept requirements for measurement. Each measurement endeavor should be guided by organizational objectives and driven by a set of measurement requirements established by the organization and the project. For example, an organizational objective might be "first-to-market with new products". This in turn might engender a requirement that factors contributing to this objective be measured so that projects might be managed to meet this objective.
- Define scope of measurement. The organizational unit to which each measurement requirement is to be applied must be established. This may consist of a functional area, a single project, a single site, or even the whole enterprise. All subsequent measurement tasks related to this requirement should be within the defined scope. In addition, the stakeholders should be identified.
- Commitment of management and staff to measurement. The commitment must be formally established, communicated, and supported by resources (see next item).
- Commit resources for measurement. The organization's commitment to measurement is an essential factor for success, as evidenced by assignment of resources for implementing the measurement process. Assigning resources includes allocation of responsibility for the various tasks of the measurement process (such as user, analyst, and librarian) and providing adequate funding, training, tools, and support to conduct the process in an enduring fashion.

2.10.3.2 Plan the Measurement Process

- Characterize the organizational unit. The organizational unit provides the context for measurement, so it is important to make this context explicit and to articulate the assumptions that it embodies and the constraints that it imposes. Characterization can be in terms of organizational processes, application domains, technology, and organizational interfaces. An organizational process model is also typically an element of the organizational unit characterization.
- Identify information needs. Information needs are based on the goals, constraints, risks, and problems of the organizational unit. They may be derived from business, organizational, regulatory, and/or product objectives. They must be identified and prioritized. Then, a subset to be addressed must be selected and the results documented, communicated, and reviewed by stakeholders.
- Select measures. Candidate measures must be selected, with clear links to the information needs. Measures must then be selected based on the priorities of the information needs and other criteria such as cost of collection, degree of process disruption during collection, ease of analysis, ease of obtaining accurate, consistent data.
- Define data collection, analysis, and reporting procedures. This encompasses collection procedures and schedules, storage, verification, analysis, reporting, and configuration management of data.
- Define criteria for evaluating the information products. Criteria for evaluation are influenced by the technical and business objectives of the organizational unit. Information products include those associated with the product being produced, as well as those associated with the processes being used to manage and measure the project.

- Review, approve, and provide resources for measurement tasks.
- The measurement plan must be reviewed and approved by the appropriate stakeholders. This includes all data collection procedures, storage, analysis, and reporting procedures; evaluation criteria; schedules; and responsibilities. Criteria for reviewing these artifacts should have been established at the organizational unit level or higher and should be used as the basis for these reviews. Such criteria should take into consideration previous experience, availability of resources, and potential disruptions to projects when changes from current practices are proposed.
- Resources should be made available for implementing the planned and approved measurement tasks. Resource availability may be staged in cases where changes are to be piloted before widespread deployment. Consideration should be paid to the resources necessary for successful deployment of new procedures or measures.
- Acquire and deploy supporting technologies. This includes evaluation of available supporting technologies, selection of the most appropriate technologies, acquisition of those technologies, and deployment of those technologies.

2.10.3.3 Perform the Measurement Process

- Integrate measurement procedures with relevant processes. The measurement procedures, such as data collection, must be integrated into the processes they are measuring. This may involve changing current processes to accommodate data collection or generation activities. It may also involve analysis of current processes to minimize additional effort and evaluation of the effect on employees to ensure that the measurement procedures will be accepted. Morale issues and other human factors need to be considered. In addition, the measurement procedures must be communicated to those providing the data, training may need to be provided, and support must typically be provided. Data analysis and reporting procedures must typically be integrated into organizational and/or project processes in a similar manner.
- Collect data. The data must be collected, verified, and stored.
- Analyze data and develop information products. Data may be aggregated, transformed, or recoded as part of the analysis process, using a degree of rigor appropriate to the nature of the data and the information needs. The results of this analysis are typically indicators such as graphs, numbers, or other indications that must be interpreted, resulting in initial conclusions to be presented to stakeholders. The results and conclusions must be reviewed, using a process defined by the organization (which may be formal or informal). Data providers and measurement users should participate in reviewing the data to ensure that they are meaningful and accurate, and that they can result in reasonable actions.
- Communicate results. Information products must be documented and communicated to users and stakeholders.

2.10.3.4 Evaluate Measurement

- Evaluate information products. Evaluate information products against specified evaluation criteria and determine strengths and weaknesses of the information products. This may be performed by an internal process or an external audit and should include feedback from measurement users. Record lessons learned in an appropriate database.
- Evaluate the measurement process. Evaluate the measurement process against specified evaluation criteria and determine the strengths and weaknesses of the process. This may be performed by an internal process or an external audit and should include feedback from measurement users.
- Identify potential improvements. Such improvements may be changes in the format of indicators, changes in units measured, or reclassification of categories. Determine the costs and benefits of potential improvements and select appropriate improvement actions. Communicate proposed improvements to the measurement process owner and stakeholders for review and approval. Also communicate lack of potential improvements if the analysis fails to identify improvements.

References:
http://en.wikipedia.org/wiki/Software_engineering, http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-171Fall2003/CourseHome/, http://www.cs.cornell.edu/courses/cs501/2008sp/, http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/, http://www.ee.unb.ca/kengleha/courses/CMPE3213/Intro7 http://cs.wwc.edu/~aabyan/435/Management.html, http://www.sei.cmu.edu/programs/sepm/, etc...

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

A Assignment, § 1.2(4)

E Exercise, § 1.3(11)

I Iterative process, § 2.2(21)

P Process models, § 2.2(21)

R Requirements analysis, § 2.3(27)
Requirements specification, § 2.3(27)

S Software configuration, § 2.8(85)
Software construction, § 2.5(48)
Software design, § 2.4(41)
Software engieering, § 2.10(107)
software engineering, § 2.5(48), § 2.8(85), § 2.9(97)
Software engineering management, § 2.10(107)
Software Engineering, OpenSource, Free tools, § 1.5(13)
Software evaluation, § 2.6(55)
Software maintenance, § 2.7(70)
Software process, § 2.1(15), § 2.2(21)
Software quality management, § 2.9(97)
Software Testing, § 2.1(15), § 2.6(55)
Sofware engineering, § 2.4(41)
Sostware Engineering, § 2.1(15)
Syllabus, § 1.1(1)

W Waterfall process, § 2.2(21)

Attributions

Collection: *Software Engineering*
Edited by: Hung Vo
URL: http://cnx.org/content/col10790/1.1/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Syllabus"
By: Hung Vo
URL: http://cnx.org/content/m28909/1.1/
Pages: 1-4
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Assignment"
By: Hung Vo
URL: http://cnx.org/content/m28886/1.1/
Pages: 4-11
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Exercise"
By: Hung Vo
URL: http://cnx.org/content/m28911/1.1/
Pages: 11-12
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Project"
By: Hung Vo
URL: http://cnx.org/content/m28926/1.1/
Pages: 12-13
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Tools"
By: Hung Vo
URL: http://cnx.org/content/m28950/1.1/
Pages: 13-14
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Introduction"
By: Hung Vo
URL: http://cnx.org/content/m28922/1.1/
Pages: 15-21
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Software development process"
By: Hung Vo
URL: http://cnx.org/content/m28927/1.2/
Pages: 21-27
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Requirements analysis"
By: Trung Hung VO
URL: http://cnx.org/content/m14621/1.6/
Pages: 27-41
Copyright: Trung Hung VO
License: http://creativecommons.org/licenses/by/2.0/

Module: "Software Design"
By: Hung Vo
URL: http://cnx.org/content/m28936/1.2/
Pages: 41-48
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Software construction"
By: Hung Vo
URL: http://cnx.org/content/m28929/1.1/
Pages: 48-55
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Software Testing"
By: Hung Vo
URL: http://cnx.org/content/m28939/1.1/
Pages: 55-70
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Software Maintenance"
By: Trung Hung VO
URL: http://cnx.org/content/m14717/1.1/
Pages: 70-85
Copyright: Trung Hung VO
License: http://creativecommons.org/licenses/by/2.0/

Module: "Software configuration management"
By: Trung Hung VO
URL: http://cnx.org/content/m14730/1.1/
Pages: 85-96
Copyright: Trung Hung VO
License: http://creativecommons.org/licenses/by/2.0/

Module: "Software quality management"
By: Hung Vo
URL: http://cnx.org/content/m28899/1.1/
Pages: 97-106
Copyright: Hung Vo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Software engineering management"
By: Hung Vo
URL: http://cnx.org/content/m28902/1.1/
Pages: 107-115

Software Engineering
Software Engineering (SE) has been defined as the application of sound engineering principles and techniques to the process of software development and delivery. Its purpose is to make formal and predictable the building of software systems that really correspond to the client's requirements and that are reliable, efficient and user friendly. This course introduces to fundamental software engineering process and focusses on the pragmatic aspects, such as requirements analysis, design, development, quality control, configuration management, verification, testing, and maintenance.

About Connexions
Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.