# Summary/Resumen ~ Swebok v3.2014

Betancourt Velázquez Nancy Etzel 181080029

Negrete Quiroz Alejandro 181080153

Garrido de la Cruz Karla Gabriela 181080157

Rojas Hernández Axel Joel 181080176

Materia:Fundamentos de Ingeniería de software

Grupo: ISC-5AV  SCC 1007

# Chapter 1 - Software Requirements

1.- Software Requirements
It means to analyze, especificación and validation from the software requerimientos, we have:
Requirements - A software requirement is a property that must be exhibited by something in order to solve some problem in the real world, therefore is a complex combination of requirements from different levels and the docket where the software will work.

1.1. Definition of a Software Requirement
Define parameters of the problem.

1.2. Product and Process Requirements
What am I going to use to create the software…

1.3 Functional and Nonfunctional Requirements
Functions and capacities of the software

1.4. Emergent Properties
Requirements that can- not be addressed by a single component but that depend on how all the software components interoperate.

1.5. Quantifiable Requirements
Software requirements should be stated as clearly and as unambiguously as possible, and, where appropriate, quantitatively.

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

2. Requirements Process
2.1. Process Models
It’s an analysis subprocess that we will need to determine to the development

2.2. Process Actors
Includes all the people, users or not, who participates in the project development

2.3. Process Support and Management
This section introduces the project management resources required and consumed by the require- ments process.

2.4. Process Quality and Improvement
refers to the cost generated by quality control and the levels of compliance to achieve the goal and customer satisfaction

3. Requirements Elicitation

is concerned with the origins of software requirements and how the software engineer can collect them

3.1.Requirements Sources
Requirements have many sources in typical soft- ware, and it is essential that all potential sources be identified and evaluated.

3.2. Elicitation Techniques
Interviews, prototypes, meetings, observations.

4. Requirements Analysis
It's an audit of the whole information compiled, it works to set limits and see if the project worth it.

4.1. Requirements Classification
Functional and nonfunctional, Product or the process, The requirement priority, Volatility/stability.

4.2. Conceptual Modeling
Their purpose is to aid in understanding the situation in which the problem occurs, as well as depicting a solution.

4.3. Architectural Design and Requirements Allocation
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.

4.4. Requirements Negotiation
Requirements prioritization is necessary, not only as a means to filter important requirements, but also in order to resolve conflicts and plan for staged deliveries, which means making complex decisions that require detailed domain knowledge and good estimation skills.

4.5. Formal Analysis
This permits static validation that the software specified by the requirements does indeed have the properties that the customer, users, and software engineer expect it to have.

5. Requirements Specification
The term "spec- ification" refers to the assignment of numerical values or limits to a product's design goals.

6. Requirements Validation
The require- ments may be validated to ensure that the software engineer has understood the requirements

# Chapter 2 - Software Design

In the general sense, design can be viewed as a form of problem solving.

The most important part of software design is understanding the problem, we couldn't solve the problem if we really don’t understand. Once that we have a clear idea of the problem generally have to consider two steps: Architectural dressing and detail design.
The output of these two processes is a set of models and artifacts that record the major decisions that have been taken, along with an explanation of the rationale for each nontrivial decision. Also a number of key issues must be dealt with when designing software: performance, security, reliability, usability, etc. Another important issue is how to decompose, organize, and package software components.

I remember when a teacher told us about how to solve a problem following the rules of a software engineer. At first if you can, you should try to do a Flow chart, with this you'll see how the software will work and if this could have mistakes or any other problems.

Third point, a software architecture is “the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both”. Sometimes software doesn´t have to be developed from cero, developers can also use frameworks and reuse parts of a program that could help ours.

The “User Interface Design”
User interface design should ensure that interaction between the human and the machine provides for effective operation and control of the machine.
The most important of the user interface is the principal's: Learnability, user familiarity, consistency, minimal surprise, recoverability, user guidance, user diversity.
We have to be empathetic with the users, some of them could be not familiarized with program terms or stay in a computer all day like us. the interface should be simple and concise. To avoid unnecessary buttons or more steps of the necessary while the use of the program.

# Chapter 3 - SOFTWARE CONSTRUCTION

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

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

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

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

**Standards in Construction**

**Standards that directly affect construction issues include:**

- **communication methods (for example, standards for document formats and contents)**
- **programming languages (for example, language standards for languages like Java and C++)**
- **coding standards (for example, standards for naming conventions, layout, and indentation)**
- **platforms (for example, interface standards for operating system calls)**
- **tools (for example, diagrammatic standards for notations like UML (Unified Modeling Language)).**

**Managing Construction**

**As modern equipment and materials became available, project schedules were compressed into a few years. In fact, although the construction phase is the most visible stage of getting a building built, most projects are in the design-and-review phase at least as long as the actual construction period.**

**As the building process compressed in time, the engineers and architects of projects were able to work on several projects at once. To find the time to do several projects, they gave up many of their construction duties. This evolution led into a specialization of engineering and architecture for the design phase, relegating construction.**

**There are a number of excellent design firms that have developed on a total-capability basis—despite the problems introduced by strong decentralization of skills. These firms, however, are not in the majority, and they will always face the problem of maintaining a large workload in order to balance areas of expertise. There is a load level beyond which the all-purpose firm becomes either non-competitive, underqualified or overpriced. The ties that bind the design team together are professionalism and money. Neither can survive independently over very many projects.**

## Practical Considerations

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

**Construction Design**
**Construction Languages**
**Coding**
**Construction Testing**
**Construction for Reuse**
**Construction with Reuse**
**Construction Quality**
**Integration**

**These are just a few examples of what we could find while we are in the software construction.**

# CHAPTER 4.-SOFTWARE TESTING

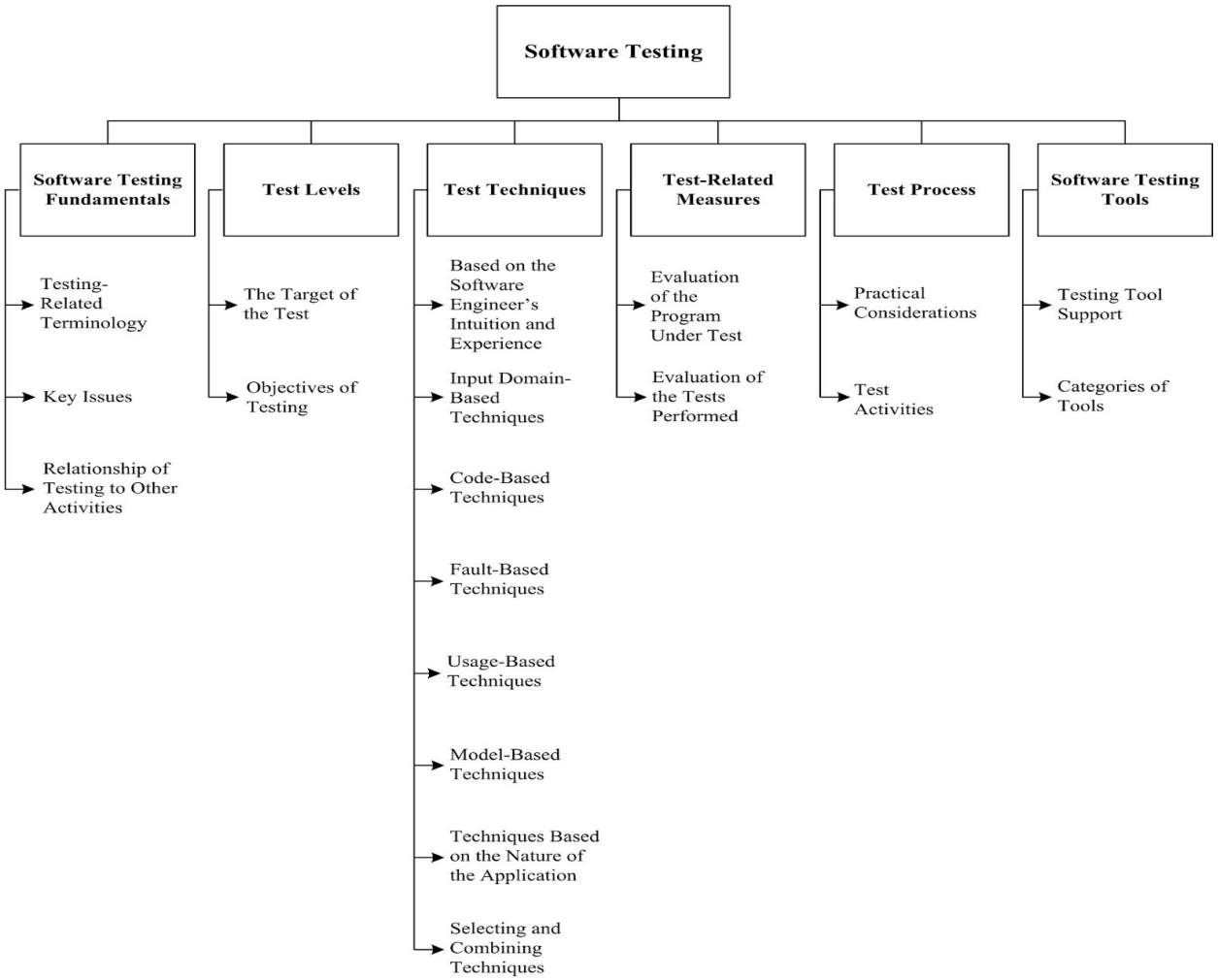
Software testing consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain.

**Dynamic**: This term means that testing always implies executing the program on selected inputs.

**Finite**: Even in simple programs, so many test cases are theoretically possible that exhaustive testing could require months or years to execute.

**Selected**: The many proposed test techniques differ essentially in how the test set is selected, and software engineers must be aware that different selection criteria may yield vastly different degrees of effectiveness.

**Expected**: It must be possible, although not always easy, to decide whether the observed outcomes of program testing are acceptable or not; otherwise, the testing effort is useless.



## 1. Software Testing Fundamentals

### 1.1.1. Definitions of Testing and RelatedTerminology

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

### 1.1.2. Faults vs. Failures

Many terms are used in the software engineering literature to describe a malfunction: notably fault, failure, and error, among others. Thus testing can reveal failures, but it is the faults that can and must be removed. The more generic term defect can be used to refer to either a fault or a failure, when the distinction is not important.

## 1.2. Key Issues

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

A test selection criterion is a means of selecting test cases or determining that a set of test cases is sufficient for a specified purpose.

### 1.2.2. Testing Effectiveness / Objectives for Testing

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

### 1.2.3. Testing for Defect Discovery

This is quite different from testing to demonstrate that the software meets its specifications or other desired properties, in which case testing is successful if no failures are observed under realistic test cases and test environments.

### 1.2.4. The Oracle Problem

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

### 1.2.5. Theoretical and Practical Limitations of Testing

Testing theory warns against ascribing an unjustified level of confidence to a series of successful tests. Unfortunately, most established results of testing theory are negative ones, in that they state what testing can never achieve as opposed to what is actually achieved. The obvious reason for this is that complete testing is not feasible in realistic software.

### 1.2.6. The Problem of Infeasible Paths

They are a significant problem in path-based testing, particularly in automated derivation of test inputs to exercise control flow paths.

### 1.2.7. Testability

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

## 1.3. Relationship of Testing to Other Activities

Software testing is related to, but different from, static software quality management techniques, proofs of correctness, debugging, and program construction.

- Testing vs. Static Software Quality Management Techniques
- Testing vs. Correctness Proofs and Formal Verification
- Testing vs. Debugging
- Testing vs. Program Construction

# 2. Test Levels

Software testing is usually performed at different levels throughout the development and maintenance processes.

## 2.1. The Target of the Test

Three test stages can be distinguished: unit, integration, and system. These three test stages do not imply any process model, nor is any one of them assumed to be more important than the other two.

### 2.1.1. Unit Testing

Typically, unit testing occurs with access to the code being tested and with the support of debugging tools. The programmers who wrote the code typically, but not always, conduct unit testing.

### 2.1.2. Integration Testing

Integration testing is the process of verifying the interactions among software components. Classical integration testing strategies, such as top down and bottom-up, are often used with hierarchically structured software. For other than small, simple software, incremental integration testing strategies are usually preferred to putting all of the components together at once—which is often called “big bang” testing.

### 2.1.3. System Testing

System testing is concerned with testing the behavior of an entire system. Effective unit and integration testing will have identified many of the software defects. External interfaces to other applications, utilities, hardware devices, or the operating environments are also usually evaluated at this level.

### 2.2. Objectives of Testing

Testing is conducted in view of specific objectives, which are stated more or less explicitly and with varying degrees of precision. 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. The subtopics listed below are those most often cited in the literature.

#### 2.2.1. Acceptance / Qualification Testing

The customer or a customer's representative thus specifies or directly undertakes activities to check that their requirements have been met, or in the case of a consumer product, that the organization has satisfied the stated requirements for the target market.

#### 2.2.2. Installation Testing

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

#### 2.2.3. Alpha and Beta Testing

These users report problems with the product. Alpha and beta testing are often uncontrolled and are not always referred to in a test plan.

#### 2.2.4. Reliability Achievement and Evaluation

Testing improves reliability by identifying and correcting faults. In addition, statistical measures of reliability can be derived by randomly generating test cases according to the operational profile of the software.

#### 2.2.5. Regression Testing

According to, regression testing is the "selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements." For incremental development, the purpose of regression testing is to show that software behavior is unchanged by incremental changes to the software, except insofar as it should.

#### 2.2.6. Performance Testing

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

### 2.2.7. Security Testing
Security testing is focused on the verification that the software is protected from external attacks. In particular, security testing verifies the confidentiality, integrity, and availability of the systems and its data.

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

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

### 2.2.10. Recovery Testing
Recovery testing is aimed at verifying software restart capabilities after a system crash or other “disaster.”

### 2.2.11. Interface Testing
Interface defects are common in complex systems. Interface testing aims at verifying whether the components interface correctly to provide the correct exchange of data and control information. This involves the generation of parameters of the API calls, the setting of external environment conditions, and the definition of internal data that affect the API.

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

### 2.2.13. Usability and Human Computer Interaction Testing
The main task of usability and human computer interaction testing is to evaluate how easy it is for end users to learn and to use the software.

## 3. Test Techniques
These techniques
attempt to “break” a program by being as systematic as possible in identifying inputs that will produce representative program behaviors; for instance, by considering subclasses of the input domain, scenarios, states, and data flows. The following list includes those testing techniques that are commonly used, but some practitioners rely on some of the techniques more than others.

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

#### 3.1.1. Ad Hoc

Perhaps the most widely practiced technique is ad hoc testing: tests are derived relying on the software engineer's skill, intuition, and experience with similar programs.

#### 3.1.2. Exploratory Testing

The effectiveness of exploratory testing relies on the software engineer's knowledge, which can be derived from various sources: observed product behavior during testing, familiarity with the application, the platform, the failure process, the type of possible faults and failures, the risk associated with a particular product, and so on.

### 3.2. Input Domain-Based Techniques

#### 3.2.1. Equivalence Partitioning

This criterion or relation may be different computational results, a relation based on control flow or data flow, or a distinction made between valid inputs that are accepted and processed by the system and invalid inputs, such as out of range values, that are not accepted and should generate an error message or initiate error processing.

#### 3.2.2. Pairwise Testing

Pairwise testing belongs to combinatorial testing, which in general also includes higher-level combinations than pairs: these techniques are referred to as t-wise, whereby every possible combination of t input variables is considered.

#### 3.2.3. Boundary-Value Analysis

Test cases are chosen on or near the boundaries of the input domain of variables, with the underlying rationale that many faults tend to concentrate near the extreme values of inputs.

#### 3.2.4. Random Testing

Tests are generated purely at random. This form of testing falls under the heading of input domain testing since the input domain must be known in order to be able to pick random points within it. Fuzz testing or fuzzing is a special form of random testing aimed at breaking the software; it is most often used for security testing.

### 3.3. Code-Based Techniques

#### 3.3.1. Control Flow-Based Criteria

The strongest of the control flow based criteria is path testing, which aims to execute all entry-to-exit control flow paths in a program's control flow graph. Since exhaustive path testing is generally not feasible because of loops, other less stringent criteria focus on coverage of paths that limit loop iterations such as statement coverage, branch coverage, and condition/ decision testing.

### 3.3.2. Data Flow-Based Criteria

The strongest criterion, all definition use paths, requires that, for each variable, every control flow path segment from a definition of that variable to a use of that definition is executed. In order to reduce the number of paths required, weaker strategies such as all-definitions and all-uses are employed.

### 3.3.3. Reference Models for Code-Based Testing

Although not a technique in itself, the control structure of a program can be graphically represented using a flow graph to visualize code based testing techniques. A flow graph is a directed graph, the nodes and arcs of which correspond to program elements.

## 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.

### 3.4.1. Error Guessing

A good source of information is the history of faults discovered in earlier projects, as well as the software engineer's expertise.

### 3.4.2. Mutation Testing

A mutant is a slightly modified version of the program under test, differing from it by a small syntactic change. The underlying assumption of mutation testing, the coupling effect, is that by looking for simple syntactic faults, more complex but real faults will be found. For the technique to be effective, a large number of mutants must be automatically generated and executed in a systematic way.

## 3.5. Usage-Based Techniques

### 3.5.1. Operational Profile

The goal is to infer from the observed test results the future reliability of the software when in actual use. To do this, inputs are assigned probabilities, or profiles, according to their frequency of occurrence in actual operation.

### 3.5.2. User Observation Heuristics

Specialized heuristics, also called usability inspection methods, are applied for the systematic observation of system usage under controlled conditions in order to determine how well people can use the system and its interfaces.

### 3.6. Model-Based Testing Techniques

The key components of model-based testing are [13]: the notation used to represent the model of the software or its requirements; workflow models or similar models; the test strategy or algorithm used for test case generation; the supporting infrastructure for the test execution; and the evaluation of test results compared to expected results. Due to the complexity of the techniques, model-based testing approaches are often used in conjunction with test automation harnesses. Model-based testing techniques include the following.

#### 3.6.1. Decision Tables

Test cases are systematically derived by considering every possible combination of conditions and their corresponding resultant actions. A related technique is cause-effect graphing

#### 3.6.2. Finite-State Machines

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

#### 3.6.3. Formal Specifications

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

#### 3.6.4. Workflow Models

Workflow models specify a sequence of activities performed by humans and/or software applications, usually represented through graphical notations. Each sequence of actions constitutes one workflow.

### 3.7. Techniques Based on the Nature of the Application

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

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

### 3.8. Selecting and Combining Techniques

#### 3.8.1. Combining Functional and Structural

These two approaches to test selection are not to be seen as alternatives but rather as complements; in fact, they use different sources of information and have been shown to highlight different kinds of problems. They could be used in combination, depending on budgetary considerations.

#### 3.8.2. Deterministic vs. Random

Test cases can be selected in a deterministic way, according to one of many techniques, or randomly drawn from some distribution of inputs, such as is usually done in reliability testing.

## 4. Test-Related Measures

Testing techniques can be viewed as aids that help to ensure the achievement of test objectives. For instance, branch coverage is a popular testing technique. Achieving a specified branch coverage measure (e.g., 95% branch coverage) should not be the
objective of testing per se: it is a way of improving the chances of finding failures by attempting to systematically exercise every program branch at every decision point.

### 4.1. Evaluation of the Program Under Test

#### 4.1.1. Program Measurements That Aid in Planning and Designing Tests

Structural measures also include measurements that determine the frequency with which modules call one another.

#### 4.1.2. Fault Types, Classification, and Statistics

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

#### 4.1.3. Fault Density

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

#### 4.1.4. Life Test, Reliability Evaluation

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

#### 4.1.5. Reliability Growth Models

They assume, in general, that when the faults that caused the observed failures have been fixed (although some models also accept imperfect fixes), the estimated product's reliability exhibits, on average, an increasing trend.

### 4.2. Evaluation of the Tests Performed

#### 4.2.1. Coverage / Thoroughness Measures

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

#### 4.2.2. Fault Seeding

When the tests are executed, some of these seeded faults will be revealed as well as, possibly, some faults that were already there. In theory, depending on which and how many of the artificial faults are discovered, testing effectiveness can be evaluated and the remaining number of genuine faults can be estimated. In practice, statisticians question the distribution and representativeness of seeded faults relative to genuine faults and the small sample size on which any extrapolations are based.

#### 4.2.3. Mutation Score

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

#### 4.2.4. Comparison and Relative Effectiveness of Different Techniques

Several studies have been conducted to compare the relative effectiveness of different testing techniques. Analytical and empirical comparisons between different techniques have been conducted according to each of the notions of effectiveness specified above.

## 5. Test Process

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

### 5.1. Practical Considerations

#### 5.1.1. Attitudes / Egoless Programming

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

### 5.1.2. Test Guides

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

### 5.1.3. Test Process Management

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

### 5.1.4. Test Documentation and Work Products

Test documents may include, among others, the test plan, test design specification, test procedure specification, test case specification, test log, and test incident report. The software under test is documented as the test item. Test documentation should also be under the control of software configuration management.

### 5.1.5. Test-Driven Development

In this way, TDD develops the test cases as a surrogate for a software requirements specification document rather than as an independent check that the software has correctly implemented the requirements. Rather than a testing strategy, TDD is a practice that requires software developers to define and maintain unit tests; it thus can also have a positive impact on elaborating user needs and software requirements specifications.

### 5.1.6. Internal vs. Independent Test Team.

The testing team can be composed of internal members (that is, on the project team, involved or not in software construction), of external members (in the hope of bringing an unbiased, independent perspective), or of both internal and external members.

### 5.1.7. Cost/Effort Estimation and Test Process Measures

These test measures may cover such aspects as number of test cases specified, number of test cases executed, number of test cases passed, and number of test cases failed, among others. Evaluation of test phase reports can be combined with root-cause analysis to evaluate test process effectiveness in finding faults as early as possible. Such an evaluation can be associated with the analysis of risks.

### 5.1.8. Termination

Thoroughness measures, such as achieved code coverage or functional coverage, as well as estimates of fault density or of operational reliability, provide useful support but are not sufficient in themselves. The decision also involves considerations about the costs and risks incurred by possible remaining failures, as opposed to the costs incurred by continuing to test.

### 5.1.9. Test Reuse and Test Patterns

A repository of test materials should be under the control of software configuration management so that changes to software requirements or design can be reflected in changes to the tests conducted. The test solutions adopted for testing some application types under certain circumstances, with the motivations behind the decisions taken, form a test pattern that can itself be documented for later reuse in similar projects.

## 5.2. Test Activities

As shown in the following description, successful management of test activities strongly depends on the software configuration management process.

### 5.2.1. Planning

Key aspects of test planning include coordination of personnel, availability of test facilities and equipment, creation and maintenance of all test-related documentation, and planning for possible undesirable outcomes.

### 5.2.2. Test-Case Generation

Test cases should be under the control of software configuration management and include the expected results for each test.

### 5.2.3. Test Environment Development

It should facilitate development and control of test cases, as well as logging and recovery of expected results, scripts, and other testing materials.

### 5.2.4. Execution

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

### 5.2.5. Test Results Evaluation

In most cases, “successful” means that the software performed as expected and did not have any major unexpected outcomes. Not all unexpected outcomes are necessarily faults but are sometimes determined to be simply noise.Before a fault

can be removed, an analysis and debugging effort is needed to isolate, identify, and describe it.

### 5.2.6. Problem Reporting / Test Log

Testing activities can be entered into a testing log to identify when a test was conducted, who performed the test, what software configuration was used, and other relevant identification information. Unexpected or incorrect test results can be recorded in a problem reporting system, the data for which forms the basis for later debugging and fixing the problems that were observed as failures during testing.

### 5.2.7. Defect Tracking

Defect tracking information is used to determine what aspects of software testing and other processes need improvement and how effective previous approaches have been.

## 6. Software Testing Tools

### 6.1. Testing Tool Support

Appropriate tools can alleviate the burden of clerical, tedious operations and make them less error-prone. Sophisticated tools can support test design and test case generation, making it more effective.

### 6.1.1. Selecting Tools

Tool selection depends on diverse evidence, such as development choices, evaluation objectives, execution facilities, and so on. In general, there may not be a unique tool that will satisfy particular needs, so a suite of tools could be an appropriate choice.

### 6.2. Categories of Tools

We categorize the available tools according to their functionality:

• Test harnesses: provide a controlled environment in which tests can be launched and the test outputs can be logged.

• Test generators: The generation can be random, path-based, mode lbased, or a mix thereof.

• Capture/replay: tools automatically reexecute, or replay, previously executed tests which have recorded inputs and outputs

• Oracle/file: comparators/assertion checking tools assist in deciding whether a test outcome is successful or not.

• Coverage analyzers and instrumentals: The analysis can be done thanks to program instrumentals that insert recording probes into the code.

• Tracers: record the history of a program's execution paths.

• Regression testing tools support the reexecution of a test suite after a section of software has been modified. They can also help to select a test subset according to the change made.

• Reliability evaluation tools support test results analysis and graphical visualization in order to assess reliability-related measures according to selected models.

# CHAPTER 5 SOFTWARE MAINTENANCE

## 1. Software Maintenance Fundamentals

The topics provide definitions and emphasize why there is a need for maintenance. Categories of software maintenance are critical to understanding its underlying meaning.

### 1.1. Definitions and Terminology

The purpose of software maintenance is defined in the international standard for software maintenance: ISO/IEC/IEEE 14764 [1*].1 In the context of software engineering, software maintenance is essentially one of the many technical processes.

### 1.2. Nature of Maintenance

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. IEEE 14764 identifies the primary activities of software maintenance as process implementation, problem and modification analysis, modification implementation, maintenance review/acceptance, migration, and retirement.

### 1.3. Need for Maintenance

Software products change due to corrective and noncorrective software actions. Maintenance must be performed in order to

• correct faults;
• improve the design;
• implement enhancements;
• interface with other software;

• adapt programs so that different hardware, software, system features, and telecommunications facilities can be used
• migrate legacy software.
• retire software.

Five key characteristics comprise the maintainer's activities:
• maintaining control over the software's dayto- day functions;
• maintaining control over software modification;
• perfecting existing functions;
• identifying security threats and fixing security vulnerabilities; and
• preventing software performance from degrading to unacceptable levels.

**1.4. Majority of Maintenance Costs**

Also, understanding the factors that influence the maintainability of software can help to contain costs. Some environmental factors and their relationship to software maintenance costs include the following:
• Operating environment refers to hardware and software.
• Organizational environment refers to policies, competition, process, product, and personnel.

**1.5. Evolution of Software**

Some state that maintenance is continued development, except that there is an extra input (or constraint)–in other words, 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.

**1.6. Categories of Maintenance**

Three categories (types) of maintenance have been defined: corrective, adaptive, and perfective. IEEE 14764 includes a fourth category–preventative.

• Corrective maintenance: reactive modification (or repairs) of a software product, performed after delivery to correct discovered problems.
• Adaptive maintenance: modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
• Perfective maintenance: modification of a software product after delivery to provide enhancements for users, improvement of program documentation, and recoding to improve software performance, maintainability, or other software attributes.
• Preventive maintenance: modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults.

## 2. Key Issues in Software Maintenance

Similarly, competing with software developers for resources is a constant battle. Planning for a future release, which often includes coding the next release while sending out emergency patches for the current release, also creates a challenge. They have been grouped under the following topic headings:

• technical issues,
• management issues,
• cost estimation, and
• measurement.

2.1. Technical Issues

2.1.1. Limited Understanding

### 2.1.2. Testing

The cost of repeating full testing on a major piece of software is significant in terms of time and money. In order to ensure that the requested problem reports are valid, the maintainer should replicate or verify problems by running the appropriate tests. The Software Testing KA provides additional information and references on this matter in its subtopic on regression testing.

### 2.1.3. Impact Analysis

Impact analysis describes how to conduct, costeffectively, 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. 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.

IEEE 14764 states the impact analysis tasks:

• analyze MRs/PRs;
• replicate or verify the problem;
• develop options for implementing the modification;
• document the MR/PR, the results, and the execution options;
• obtain approval for the selected modification option.

### 2.1.4. Maintainability

As a primary software quality characteristic, maintainability should be specified, reviewed, and controlled during software development activities in order to reduce maintenance costs. When done successfully, the software's maintainability will improve. Maintainability is often difficult to achieve because the subcharacteristics

are often not an important focus during the process of software development. The developers are, typically, more preoccupied with many other activities and frequently prone to disregard the maintainer's requirements.

## 2.2. Management Issues

### 2.2.1. Alignment with Organizational Objectives

The main emphasis is to deliver a product that meets user needs on time and within budget. 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.

### 2.2.2. Staffing

Staffing refers to how to attract and keep software maintenance staff. Maintenance is not often viewed as glamorous work. As a result, software maintenance personnel are frequently viewed as "second-class citizens," and morale therefore suffers.

### 2.2.3. Process

The software life cycle process is a set of activities, methods, practices, and transformations that people use to develop and maintain software and its associated products. At the process level, software maintenance activities share much in common with software development.

### 2.2.4. Organizational Aspects of Maintenance

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 permanent maintenance specific team (or maintainer).

- allows for specialization;
- creates communication channels;
- promotes an egoless, collegiate atmosphere;
- reduces dependency on individuals;
- allows for periodic audit checks.

### 2.2.5. Outsourcing

Outsourcing and offshoring software maintenance has become a major industry. Organizations are outsourcing entire portfolios of software, including software maintenance. More often, the outsourcing option is selected for less mission-critical software, as organizations are unwilling to lose control of the software used in their core business. Outsourcing requires a significant initial investment and the setup of a maintenance process that will require automation.

### 2.3. 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.

#### 2.3.1. Cost Estimation

Maintenance cost estimates are affected by many technical and nontechnical actors. IEEE 14764 states that “the two most popular approaches to estimating resources for software maintenance are the use of parametric models and the use of experience”.

#### 2.3.2. Parametric Models

Of significance is that historical data from past maintenance are needed in order to use and calibrate the mathematical models. Cost driver attributes affect the estimates.

#### 2.3.3. Experience

Experience, in the form of expert judgment, is often used to estimate maintenance effort. Clearly, the best approach to maintenance estimation is to combine historical data and experience. The cost to conduct a modification (in terms of number of people and amount of time) is then derived.

### 2.4. Software Maintenance Measurement

There are several software measures that can be derived from the attributes of the software, the maintenance process, and personnel, including size, complexity, quality, understandability, maintainability, and effort. Complexity measures of software can also be obtained using available commercial tools. These measures constitute a good starting point for the maintainer’s measurement program.

#### 2.4.1. Specific Measures

The maintainer must determine which measures are appropriate for a specific organization based on that organization’s own context. The software quality model suggests measures that are specific for software maintenance. Measures for subcharacteristics of maintainability include the following.

## 3. Maintenance Process

In addition to standard software engineering processes and activities described in IEEE 14764, there are a number of activities that are unique to maintainers.

### 3.1. Maintenance Processes

Maintenance processes provide needed activities and detailed inputs/outputs to those activities as described in IEEE 14764.

• process implementation,

• problem and modification analysis,
• modification implementation,
• maintenance review/acceptance,
• migration, and
• software retirement.

### 3.2. Maintenance Activities

The maintenance process contains the activities and tasks necessary to modify an existing software product while preserving its integrity. These activities and tasks are the responsibility of the maintainer. Maintainers perform analysis, design, coding, testing, and documentation. However, for software maintenance, some activities involve processes unique to software maintenance.

#### 3.2.1. Unique Activities

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

• Program understanding: activities needed to obtain a general knowledge of what a software product does and how the parts work together.

• Transition: a controlled and coordinated sequence of activities during which software is transferred progressively from the developer to the maintainer.

• Modification request acceptance/rejection: modifications requesting work beyond a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.

• Maintenance help desk: an end-user and maintenance coordinated support function that triggers the assessment, prioritization, and costing of modification requests.

• Impact analysis: a technique to identify areas impacted by a potential change

• Maintenance Service-Level Agreements (SLAs) and maintenance licenses and contracts: contractual agreements that describe the services and quality objectives.

#### 3.2.2. Supporting Activities

Maintainers may also perform support activities, such as documentation, software configuration management, verification and validation, problem resolution, software quality assurance, reviews, and audits. Another important support activity consists of training the maintainers and users.

#### 3.2.3. Maintenance Planning Activities

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

• business planning (organizational level),
• maintenance planning (transition level),
• release/version planning (software level)
• individual software change request planning (request level).

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.

The maintenance concept for each software product needs to be documented in the plan and should address the
• scope of the software maintenance,
• adaptation of the software maintenance process,
• identification of the software maintenance organization
• estimate of software maintenance costs.

### 3.2.4. Software Configuration Management

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. The SCM process is implemented by developing and following a software configuration management plan and operating procedures. Maintainers participate in Configuration Control Boards to determine the content of the next release/version.

### 3.2.5. Software Quality

quality will result from the maintenance of software. Maintainers should have a software quality program. It must be planned and processes must be implemented to support the maintenance process. It is also recommended that the maintainer adapt the software development processes, techniques and deliverables (for instance, testing documentation), and test results. More details can be found in the Software Quality KA.

## 4. Techniques for Maintenance

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

### 4.1. Program Comprehension

Code browsers are key tools for program comprehension and are used to organize and present source code. Clear and concise documentation can also aid in program comprehension.

### 4.2. 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. It is often not undertaken to improve maintainability but to replace aging legacy software.

### 4.3. Reverse Engineering

Reverse engineering is the process of analyzing software to identify the software's components and their inter-relationships 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. Tools are key for reverse engineering and related tasks such as redocumentation and design recovery.

### 4.4. Migration

Migrating software can also entail a number of additional activities such as

• notification of intent: a statement of why the old environment is no longer to be supported, followed by a description of the new environment and its date of availability;

• parallel operations: make available the old and new environments so that the user experiences a smooth transition to the new environment;

• notification of completion: when the scheduled migration is completed, a notification is sent to all concerned;

• postoperative review: an assessment of parallel operation and the impact of changing to the new environment;

• data archival: storing the old software data.

### 4.5. Retirement

An analysis should be performed to assist in making the retirement decision. This analysis should be included in the retirement plan, which covers retirement requirements, impact, replacement, schedule, and effort.

## 5. Software Maintenance Tools

This topic encompasses tools that are particularly important in software maintenance where existing software is being modified.

• program slicers, which select only parts of a program affected by a change;

• static analyzers, which allow general viewing and summaries of a program content;
• dynamic analyzers, which allow the maintainer to trace the execution path of a program;
• data flow analyzers, which allow the maintainer to track all possible data flows of a program;
• cross-referencers, which generate indices of program components; and
• dependency analyzers, which help maintainers analyze and understand the interrelationships between components of a program.

# CHAPTER 6 SOFTWARE CONFIGURATION MANAGEMENT

Software configuration management (SCM) is a supporting-software life cycle process that benefits project management, development and maintenance activities, quality assurance activities, as well as the customers and users of the end product. The concepts of configuration management apply to all items to be controlled, although there are some differences in implementation between hardware CM and software CM.

## 1. Management of the SCM Process

A successful SCM implementation requires careful planning and management. This, in turn, requires an understanding of the organizational context for, and the constraints placed on, the design and implementation of the SCM process.

### 1.1. Organizational Context for SCM

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

### 1.2. 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. Finally, the particular software life cycle process chosen for a software project and the level of formalism selected to implement the software affect the design and implementation of the SCM process.

### 1.3. Planning for SCM

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. For instance, continuous integration is a common practice in many software development approaches. It is typically characterized by frequent build-test-deploy cycles. SCM activities must be planned accordingly.

### 1.3.1. SCM Organization and Responsibilities

To prevent confusion about who will perform given SCM activities or tasks, organizational roles to be involved in the SCM process need to be clearly identified. The overall authority and reporting channels for SCM should also be identified, although this might be accomplished at the project management or quality assurance planning stage.

### 1.3.2. SCM Resources and Schedules

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.

### 1.3.3. Tool Selection and Implementation

The following questions should be considered:

• Organization: what motivates tool acquisition from an organizational perspective?
• Tools: can we use commercial tools or develop them ourselves?
• Environment: what are the constraints imposed by the organization and its technical context?
• Legacy: how will projects use (or not) the new tools?
• Financing: who will pay for the tools' acquisition, maintenance, training, and customization?
• Scope: how will the new tools be deployed for instance, through the entire organization or only on specific projects?
• Ownership: who is responsible for the introduction of new tools?
• Future: what is the plan for the tools' use in the future?
• Change: how adaptable are the tools?
• Branching and merging: are the tools' capabilities compatible with the planned branching and merging strategies?
• Integration: do the various SCM tools integrate among themselves? With other tools in use in the organization?
• Migration: can the repository maintained by the version control tool be ported to another version control tool while maintaining complete history of the configuration items it contains?

### 1.3.4. Vendor/Subcontractor Control

A software project might acquire or make use of purchased software products, such as compilers or other tools. When using subcontracted software, both the SCM requirements to be imposed on the subcontractor's SCM process as part of the subcontract and the means for monitoring compliance need to be established.

### 1.3.5. Interface Control

The SCM role may be part of a larger, system-level process for interface specification and control; it may involve interface specifications, interface control plans, and interface control documents. In this case, SCM planning for interface control takes place within the context of the system level process.

## 1.4. SCM Plan

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. This reference provides requirements for the information to be contained in an SCMP; it also defines and describes six categories of SCM information to be included in an SCMP:

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

## 1.5. Surveillance of Software Configuration Management

There are likely to be specific SQA requirements for ensuring compliance with specified SCM processes and procedures. The person responsible for SCM ensures that those with the assigned responsibility perform the defined SCM tasks correctly. The software quality assurance authority, as part of a compliance auditing activity, might also perform this surveillance. The use of integrated SCM tools with process control capability can make the surveillance task easier. Some tools facilitate process compliance while providing flexibility for the software engineer to adapt procedures.

### 1.5.1. SCM Measures and Measurement

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. Discussion of software process and product measurement is presented in the Software Engineering Process KA. Software measurement programs are described in the Software Engineering Management KA.

### 1.5.2. In-Process Audits of SCM

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.

## 2. Software Configuration Identification

Software configuration identification identifies items to be controlled, establishes identification schemes for the items and their versions, and establishes the tools and techniques to be used in acquiring and managing controlled items.

### 2.1. Identifying Items to Be Controlled

One of the first steps in controlling change is identifying the software items to be controlled. This involves understanding the software configuration within the context of the system configuration, selecting software configuration items.

### 2.1.1. Software Configuration

Software configuration is the functional and physical characteristics of hardware or software as set forth in technical documentation or achieved in a product. It can be viewed as part of an overall system configuration.

### 2.1.2. Software Configuration Item

The SCM typically controls a variety of items in addition to the code itself. Software items with potential to become SCIs include plans, specifications and design documentation, testing materials, software tools, source and executable code, code libraries, data and data dictionaries, and documentation for installation, maintenance, operations, and software use. Selecting SCIs is an important process in which a balance must be achieved between providing adequate visibility for project control purposes and providing a manageable number of controlled items.

### 2.1.3. Software Configuration Item Relationships

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 identified items to the software structure, as well as the need to support the evolution of the software items and their relationships.

### 2.1.4. Software Version

Software items evolve as a software project proceeds. A version of a software item is an identified instance of an item. It can be thought of as a state of an evolving item. A variant is a version of a program resulting from the application of software diversity.

### 2.1.5. Baseline

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. The product baseline corresponds to the completed software product delivered for system integration. The baselines to be used for a given project, along with the associated levels of authority needed for change approval, are typically identified in the SCMP.

### 2.1.6. Acquiring Software Configuration Items

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. 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.

### 2.2. Software Library

A software library is a controlled collection of software and related documentation designed to aid in software development, use, or maintenance. Several types of libraries might be used, each corresponding to the software item's particular level of maturity. For example, a working library could support coding and a project support library could support testing, while a master library could be used for finished products. 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.

## 3. 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.

### 3.1. Requesting, Evaluating, and Approving Software Changes

The first step in managing changes to controlled items is determining what changes to make. A change request (CR) is a request to expand or reduce the project scope; modify policies, processes, plans, or procedures; modify costs or budgets; or revise schedules. This provides an opportunity for tracking defects and collecting change activity measurements by change type. Once an SCR is received, a technical evaluation (also known as an impact analysis) is performed to determine the extent of the modifications that would be necessary should the change request be accepted.

### 3.1.1. 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. All stakeholders, appropriate to the level of the CCB, are represented. When the scope of authority of a CCB is strictly software, it is known as a Software Configuration Control Board (SCCB). The activities of the CCB are typically subject to software quality audit or review.

### 3.1.2. Software Change Request Process

An effective software change request (SCR) process requires the use of supporting tools and procedures for originating change requests, enforcing the flow of the change process, capturing CCB decisions, and reporting change process information.

## 3.2. Implementing Software Changes

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. Changes may be supported by source code version control tools. These tools allow a team of software engineers, or a single software engineer, to track and document changes to the source code. These tools may be manifested as separate, specialized applications under the control of an independent SCM group. They may also appear as an integrated part of the software engineering environment. Finally, they may be as elementary as a rudimentary change control system provided with an operating system.

## 3.3. Deviations and Waivers

The constraints imposed on a software engineering effort or the specifications produced during the development activities might contain provisions that cannot be satisfied at the designated point in the life cycle. A deviation is a written authorization, granted prior to the manufacture of an item, to depart from a particular performance or design requirement for a specific number of units or a specific period of time. In these cases, a formal process is used for gaining approval for deviations from, or waivers of, the provisions.

## 4. Software Configuration Status Accounting

is an element of configuration management consisting of the recording and reporting of information needed to manage a configuration effectively.

### 4.1. Software Configuration Status Information

The SCSA activity designs and operates a system for the capture and reporting of necessary information as the life cycle proceeds. The types of information available include the approved configuration identification as well as the identification and current implementation status of changes, deviations, and waivers. Some form of automated tool support is necessary to accomplish the SCSA data collection and reporting tasks; this could be a database capability, a stand-alone tool, or a capability of a larger, integrated tool environment.

### 4.2. Software Configuration Status Reporting

In addition to reporting the current status of the configuration, the information obtained by the SCSA can serve as a basis of various measurements. Examples include the number of change requests per SCI and the average time needed to implement a change request.

## 5. Software Configuration Auditing

Consequently, each audit must be carefully planned. An audit can require a number of individuals to perform a variety of tasks over a fairly short period of time. Tools to support the planning and conduct of an audit can greatly facilitate the process. Software configuration auditing determines the extent to which an item satisfies the required functional and physical characteristics. Informal audits of this type can be conducted at key points in the life cycle.

### 5.1. Software Functional Configuration Audit

The output of the software verification and validation activities (see Verification and Validation in the Software Quality KA) is a key input to this audit.

### 5.2. Software Physical Configuration Audit

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

### 5.3. In-Process Audits of a Software Baseline

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

consistent with specifications or to ensure that evolving documentation continues to be consistent with the developing baseline item.

## 6. Software Release Management and Delivery

When different versions of a software item are available for delivery (such as versions for different platforms or versions with varying capabilities), it is frequently necessary to recreate specific versions and package the correct materials for delivery of the version. The software library is a key element in accomplishing release and delivery tasks.

### 6.1. Software Building

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. In addition to building software for new releases, it is usually also necessary for SCM to have the capability to reproduce previous releases for recovery, testing, maintenance, or additional release purposes. Software is built using particular versions of supporting tools, such as compilers (see Compiler Basics in the Computing Foundations KA). tool capability is useful for selecting the correct versions of software items for a given target environment and for automating the process of building the software from the selected versions and appropriate configuration data. For projects with parallel or distributed development environments, this tool capability is necessary. Most software engineering environments provide this capability.

### 6.2. Software Release Management

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. This tool capability might also maintain information on various target platforms and on various customer environments.

## 7. Software Configuration Management Tools

Individual support tools are appropriate and typically sufficient for small organizations or development groups without variants of their software products or other complex SCM requirements. They include:

• Version control tools: track, document, and store individual configuration items such as source code and external documentation.

• Build handling tools: in their simplest form, such tools compile and link an executable version of the software. More advanced building tools extract the latest

version from the version control software, perform quality checks, run regression tests, and produce various forms of reports, among other tasks.
• Change control tools: mainly support the control of change requests and events notification (for example, change request status changes, milestones reached).
Companywide-process support tools can typically automate portions of a companywide process, providing support for workflow managements, roles, and responsibilities. They are able to handle many items, data, and life cycles. Such tools add to project-related support by supporting a more formal development process, including certification requirements.

# CAPÍTULO 7. CHAPTER 7

## SOFTWARE ENGINEERING MANAGEMENT

### Project Planning

The first step in software project planning should be selection of an appropriate software develop-ment life cycle model and perhaps tailoring it based on project scope, software requirements, and a risk assessment. Software quality management processes should be determined as part of the planning process and result in procedures and responsibilities for software quality assurance, verification and validation, reviews, and audits . Software development life cycle mod-els span a continuum from predictive to adaptive . Adaptive SDLCs are designed to accommodate emergent software requirements and iterative adjustment of plans.

Well-known SDLCs include the waterfall, incremental, and spiral models plus various forms of agile software development . Tools may include tools for project scheduling, software require-ments, software design, software construction, software maintenance, software configuration management, software engineering process, soft-ware quality, and others. Software-unique aspects of risk, such as software engineers' tendency to add unneeded features, or the risks related to soft-ware's intangible nature, can influence risk man-agement of a software project. Particular atten-tion should be paid to the management of risks related to software quality requirements such as safety or security .

Software quality requirements should be identi-fied, perhaps in both quantitative and qualitative terms, for a software project and the associated work products. Thresholds for acceptable qual-ity measurements should be set for each software quality requirement based on stakeholder needs and expectations. Procedures concerned with ongoing Software Quality Assurance and quality improvement throughout the development process, and for verification and validation of the deliverable software product, should also be specified during quality planning . For software projects, where change is an expec-tation, plans should be managed.

During software project enactment plans are implemented and the processes embodied in the plans are enacted. The measurement process should be enacted dur-ing the software project to ensure that relevant and useful data are collected . The project risk profile should be revisited, and adherence to software quality requirements eval-uated . In all cases, software configuration control and software con-figuration management procedures should be adhered to , decisions should be documented and communicated to all relevant parties, plans should be revisited and revised when necessary, and relevant data recorded .

Similarly, assessments of the effectiveness of the software process, the personnel involved, and the tools and methods employed should also be undertaken reg-ularly and as determined by circumstances. Because achieving stakeholder satisfaction is a principal goal of the software engineering manager, progress towards this goal should be assessed periodically. Variances from software requirements should be identified and appropri-ate actions should be taken. As in the control process activity above , software configuration control and software configuration management procedures should be followed , decisions docu-mented and communicated to all relevant parties, plans revisited and revised where necessary, and relevant data recorded .

The measurement procedures, such as data collection, should be integrated into the software processes they are measuring. This may involve chang-ing current software processes to accommo-date data collection or generation activities. Collection can sometimes be automated by using software engineer-ing management tools to analyze data and develop reports. Software engineering management tools are often used to provide visibility and control of software engineering management processes.

There has been a recent trend towards the use of integrated suites of software engineer-ing tools that are used throughout a project to plan, collect and record, monitor and control, and report project and product information. Some projects use automated estima-tion tools that accept as input the estimated size and other characteristics of a software product and produce estimates of the required total effort, schedule, and cost.

# CAPÍTULO 8. CHAPTER 8

# SOFTWARE ENGINEERING PROCESS

An engineering process consists of a set of interrelated activities that transform one or more inputs into outputs while consuming resources to accom-plish the transformation. Many of the processes of traditional engineering disciplines are concerned with transforming energy and physical entities from one form into another, as in a hydroelectric dam that transforms potential energy into electrical energy or a petroleum refinery that uses chemical processes to transform crude oil into gasoline. In this knowledge area , software engineer-ing processes are concerned with work activities accomplished by software engineers to develop, maintain, and operate software, such as require-ments, design, construction, testing, configura-tion management, and other software engineering processes. For readability, «software engineering process» will be referred to as «software process» in this KA.

Software Engineering Manage-ment is concerned with tailoring, adapting, and implementing software processes for a specific software project .

Mod-els and methods support a systematic approach to software development and modification. The Software Quality KA is concerned with the planning, assurance, and control processes for project and product quality.

Measurement and measurement results in the Engineering Founda-tions KA are essential for evaluating and control-ling software processes.

As illustrated in Figure 8.1, this KA is concerned with software process definition, software life cycles, software process assessment and improve-ment, software measurement, and software engi-neering process tools. This topic is concerned with a definition of soft-ware process, software process management, and software process infrastructure. As stated above, a software process is a set of interrelated activities and tasks that transform input work products into output work products. At minimum, the description of a software pro-cess includes required inputs, transforming work activities, and outputs generated.

As illustrated in Figure 8.2, a software process may also include its entry and exit criteria and decomposition of the work activities into tasks, which are the smallest units of work subject to management accountability. A software process may include subprocesses. The tasks of the requirements validation activity might include requirements reviews, prototyping, and model validation.

The output of requirements validation is typically a validated software requirements specification that provides inputs to the software design and software test-ing processes.

Complete definition of a software process may also include the roles and competencies, IT sup-port, software engineering techniques and tools, and work environment needed to perform the process, as well as the approaches and measures used to determine the efficiency and effectiveness of performing the process.

In addition, a software process may include interleaved technical, collaborative, and adminis-trative activities. It must be emphasized that there is no best soft-ware process or set of software processes. Soft-ware processes must be selected, adapted, and applied as appropriate for each project and each organizational context.

Changing to a new process, improving an existing process, organizational change, and infrastructure change are closely related, as all are usually initiated with the goal of improving the cost, development sched-ule, or quality of the software products. A software process infrastructure can provide process definitions, policies for inter-preting and applying the processes, and descrip-tions of the procedures to be used to implement the processes.

Additionally, a software process infrastructure may provide funding, tools, train-ing, and staff members who have been assigned responsibilities for establishing and maintaining the software process infrastructure. Software process infrastructure varies, depend-ing on the size and complexity of the organization and the projects undertaken within the organiza-tion.

In the latter case, various organizational units may be established to oversee implementation and improvement of the software processes.

A common misperception is that establishing a software process infrastructure and implementing repeatable software processes will add time and cost to software development and maintenance. A software development life cycle includes the software processes used to specify and transform software requirements into a deliv-erable software product. A software product life cycle includes a software development life cycle plus additional software processes that provide for deployment, maintenance, support, evolution, retirement, and all other

inception-to-retirement processes for a software product, including the software configuration management and software quality assurance processes that are applied throughout a software product life cycle.

A software product life cycle may include multi-ple software development life cycles for evolving and enhancing the software. Individual software processes have no tempo-ral ordering among them. Life cycle models typically emphasize the key software processes within the model and their temporal and logical interdependen-cies and relationships. Detailed definitions of the software processes in a life cycle model may be provided directly or by reference to other documents.

In addition to conveying the temporal and logical relationships among software processes, the software development life cycle model includes the control mechanisms for applying entry and exit criteria . The output of one software process often provides the input for oth-ers .

Concurrent execution of several software process activities may produce a shared output . Some software processes may be regarded as less effective unless other software processes are being performed at the same time .

Many distinct software processes have been defined for use in the various parts of the soft-ware development and software maintenance life cycles.

Processes include software pro-cesses for development, operation, and maintenance of software. Software processes in addition to those listed above include the following. Project management processes include pro-cesses for planning and estimating, resource management, measuring and controlling, leading, managing risk, managing stakeholders, and coor-dinating the primary, supporting, organizational, and cross-project processes of software develop-ment and maintenance projects.

Software processes are also developed for particular needs, such as process activities that address software quality characteristics . For example, secu-rity concerns during software development may necessitate one or more software processes to protect the security of the development environ-ment and reduce the risk of malicious acts. Soft-ware processes may also be developed to provide adequate grounds for establishing confidence in the integrity of the software.

Linear SDLC models are sometimes referred to as predictive software development life cycle models, while iterative and agile SDLCs are referred to as adaptive software development life cycle models. A distinguishing feature of the various soft-ware development life cycle models is the way in which software requirements are managed. Lin-ear development models typically develop a com-plete set of software requirements, to the extent possible, during project initiation and planning. The software requirements are then rigorously controlled.

Changes to the software requirements are based on change requests that are processed by a change control board . An incremental model produces successive increments of work-ing, deliverable software based on partitioning of the software requirements to be implemented in each of the increments. The software require-ments may be rigorously controlled, as in a linear model, or there may be some flexibility in revising the software requirements as the software product evolves.

The Software Engineering Measurement topic in the Software Engineering Management

KA describes a process for implementing a software measurement program.

Software information models allow modeling, analysis, and prediction of software process and software product attributes to provide answers to relevant questions and achieve process and product improvement goals.

Continuous evaluation of the model may indi-cate a need for adjustments over time as the con-text in which the model is applied changes. The Goals/Questions/Metrics method was originally intended for establishing measure-ment activities, but it can also be used to guide analysis and improvement of software processes.

Software process measurement techniques are used to collect process data and work product data, transform the data into useful information, and analyze the information to identify process activities that are candidates for improvement. In some cases, new software processes may be needed. Process mea-surement techniques can be used to collect both quantitative and qualitative data. The purpose of quantitative process measurement techniques is to collect, transform, and analyze quantitative process and work product data that can be used to indicate where process improve-ments are needed and to assess the results of process improvement initiatives.

Quantitative process measurement techniques are used to col-lect and analyze data in numerical form to which mathematical and statistical techniques can be applied. Quantitative process data can be collected as a byproduct of software processes.

# CAPITULO 9. CHAPTER 9

# SOFTWARE ENGINEERING MODELS

# AND METHODS

This chapter on software engineering models and methods is divided into four main topic areas.

The discussion guides the reader through a summary of heuristic methods, formal methods, prototyping, and agile methods. Modeling of software is becoming a pervasive technique to help software engineers understand, engineer, and communicate aspects of the soft-ware to appropriate stakeholders. Stakeholders are those persons or parties who have a stated or implied interest in the software . Modeling provides the software engineer with an organized and systematic approach for repre-senting significant aspects of the software under study, facilitating decision-making about the soft-ware or elements of it, and communicating those significant decisions to others in the stakeholder communities.

Modeling typically involves developing only those aspects or features of the software that need specific answers, abstracting away any nonessential information. A model is an abstraction or simplification of a software component. A consequence of using abstraction is that no single abstraction com-pletely describes a software component. Models are constructed to represent real-world objects and their behaviors to answer specific questions about how the software is expected to operate.

The fact that a model is an abstraction with missing infor-mation can lead one into a false sense of com-pletely understanding the software from a single model. As a practical matter, there is usually a good understanding of the semantics of a specific software model due to the modeling language selected, how that modeling language is used to express entities and relations within that model, the experience base of the modeler, and the context within which the modeling has been undertaken and so represented. With many software engineers working on a model part over time coupled with tool updates and perhaps new requirements, there are opportunities for

portions of the model to represent something different from the original author's intent and initial model context. Typically, the postconditions represent how the state of the software has changed, how param-eters passed to the function or method have changed, how data values have changed, or how the return value has been affected.

These invariants are relevant and necessary to the software and the correct operation of the function or method.

# CHAPTER 13

# COMPUTING FOUNDATIONS

## Software

Software engineering is concerned with the application of computers, computing, and software to practical purposes, specifically the design, construction, and operation of efficient and economical software systems. Thus, at the core of software engineering is an understanding of computer science. Such courses include programming, data structure, algorithms, computer organization, operating systems, compilers, databases, networking, distributed systems, and so forth. Thus, when breaking down topics, it can be tempting to decompose the Computing Foundations KA according to these often-found divisions in relevant courses.

However, a purely course-based division of topics suffers serious drawbacks. For one, not all courses in computer science are related or equally important to software engineering. Thus, some topics that would otherwise be covered in a computer science course are not covered in this KA. Second, some topics discussed in this guideline do not exist as standalone courses in undergraduate or graduate computer science programs.

Consequently, such topics may not be adequately covered in a purely course-based breakdown. The Computing Foundations KA is divided into seventeen different topics. A topic's direct usefulness to software engineers is the criterion used for selecting topics for inclusion in this KA .

Problem solving refers to the thinking and activities conducted to answer or derive a solution to a problem. There are many ways to approach a problem, and each way employs different tools and uses different processes. These different ways of

approaching problems gradually expand and define themselves and finally give rise to different disciplines. • Formulate the real problem.

• Analyze the problem.

• Design a solution search strategy.

Gerard Voland writes, «It is important to recognize that a specific problem should be formulated if one is to develop a specific solution» .

Some general techniques to help one formulate the real problem include statement-restatement, determining the source and the cause, revising the statement, analyzing present and desired state, and using the fresh eye approach.

Once the problem statement is available, the next step is to analyze the problem statement or situation to help structure our search for a solution.

Four types of analysis include situation analysis, in which the most urgent or critical aspects of a situation are identified first; problem analysis, in which the cause of the problem must be determined; decision analysis, in which the action needed to correct the problem or eliminate its cause must be determined; and potential problem analysis, in which the action needed to prevent any reoccurrences of the problem or the development of new problems must be determined.

The first task in solving a problem using a computer is to determine what to tell the computer to do. There may be many ways to tell the story, but all should take the perspective of a computer such that the computer can eventually solve the problem. In general, a problem should be expressed in such a way as to facilitate the development of algorithms and data structures for solving it. The result of the first task is a problem statement.

The next step is to convert the problem statement into algorithms that solve the problem. To the discipline of software engineering, the ultimate objective of problem solving is to transform a problem expressed in natural language into electrons running around a circuit. The conversion of a problem statement into algorithms and algorithms into program codes usually follows a «stepwise refinement» in which we start with a problem statement, rewrite it as a task, and recursively decompose the task into a few simpler subtasks until the task is so simple that solutions to it are straightforward. Abstraction is an indispensible technique associated with problem solving.

It refers to both the process and result of generalization by reducing the information of a concept, a problem, or an observable phenomenon so that one can focus on the «big picture.» One of the most important skills in any engineering undertaking is framing the levels of abstraction appropriately. «Through abstraction,» according to Voland, «we view the problem and its possible solution paths from a higher level of

conceptual understanding. As a result, we may become better prepared to recognize possible relationships between different aspects of the problem and thereby generate more creative design solutions» . This is particularly true in computer science in general and in software engineering in particular .

Most of the time, these different levels of abstraction are organized in a hierarchy. There are many ways to structure a particular hierarchy and the criteria used in determining the specific content of each layer in the hierarchy varies depending on the individuals performing the work. At no time, shall there be any loop in a hierarchy. A hierarchy often forms naturally in task decomposition.

These alternate abstractions do not form a hierarchy but rather complement each other in helping understanding the problem and its solution. Though beneficial, it is as times difficult to keep alternate abstractions in sync. Programming is composed of the methodologies or activities for creating computer programs that perform a desired function. It is an indispensible part in software construction.

In general, programming can be considered as the process of designing, writing, testing, debugging, and maintaining the source code. This source code is written in a programming language. Programming involves design, writing, testing, debugging, and maintenance. Design is the conception or invention of a scheme for turning a customer requirement for computer software into operational software.

It is the activity that links application requirements to coding and debugging. Writing is the actual coding of the design in an appropriate programming language. Testing is the activity to verify that the code one writes actually does what it is supposed to do. Debugging is the activity to find and fix bugs in the source code .

Maintenance is the activity to update, correct, and enhance existing programs.

Programming is highly creative and thus somewhat personal. Different people often write different programs for the same requirements. This diversity of programming causes much difficulty in the construction and maintenance of large complex software. Various programming paradigms have been developed over the years to put some standardization into this highly creative and personal activity.

When one programs, he or she can use one of several programming paradigms to write the code. The major types of programming paradigms are discussed below. Interfaces exist between procedures to facilitate correct and smooth calling operations of the programs. Under structured programming, programmers often follow established protocols and rules of thumb when writing code.

The primary features of OOP are that objects representing various abstract and concrete entities are created and these objects interact with each other to collectively fulfill the desired functions. AOP aims to isolate secondary or supporting functions

from the main program's business logic by focusing on the cross sections of the objects.

The primary motivation for AOP is to resolve the object tangling and scattering associated with

In functional programming, all computations are treated as the evaluation of mathematical functions. In contrast to the imperative programming that emphasizes changes in state, functional programming emphasizes the application of functions, avoids state and mutable data, and provides referential transparency.

Programs must be written in some programming language with which and through which we describe necessary computations. In other words, we use the facilities provided by a programming language to describe problems, develop algorithms, and reason about problem solutions. To write any program, one must understand at least one programming language. A programming language is designed to express computations that can be performed by a computer.

In a practical sense, a programming language is a notation for writing programs and thus should be able to express most data structures and algorithms. Some, but not all, people restrict the term «programming language» to those languages that can express all possible algorithms.

Languages

Just like natural languages, many programming languages have some form of written specification of their syntax and semantics . In general, a programming language supports such constructs as variables, data types, constants, literals, assignment statements, control statements, procedures, functions, and comments. A machine language uses ones and zeros to represent instructions and variables, and is directly understandable by a computer. An assembly language contains the same instructions as a machine language but the instructions and variables have symbolic names that are easier for humans to remember.

Assembly languages cannot be directly understood by a computer and must be translated into a machine language by a utility program called an assembler. There often exists a correspondence between the instructions of an assembly language and a machine language, and the translation from assembly code to machine code is straightforward.

In comparison to low-level programming languages, it often uses natural-language elements and is thus much easier for humans to understand. Examples of high-level programming languages include C, C++, C#, and Java. Most programming languages allow programmers to specify the individual instructions that a computer is to execute. Such programming languages are called imperative programming languages because one has to specify every step clearly to the computer.

But some programming languages allow programmers to only describe the function to be performed without specifying the exact instruction sequences to be executed. Such programming languages are called declarative programming languages. The key point to note is that declarative programming only describes what the program should accomplish without describing how to accomplish it. For this reason, many people believe declarative programming facilitates easier software development.

Declarative programming languages include Lisp and Prolog, while imperative programming languages include C, C++, and JAVA. In high-level programming languages, syntax errors are caught during the compilation or translation from the high-level language into machine code.

Static debugging usually takes the form of code review, while dynamic debugging usually takes the form of tracing and is closely associated with testing. All three techniques are used at various stages of program development. The main activity of dynamic debugging is tracing, which is executing the program one piece at a time, examining the contents of registers and memory, in order to examine the results at each step. There are three ways to trace a program.

This method is tedious but useful in verifying each step of a program. This technique lets one quickly execute selected code sequences to get a high-level overview of the execution behavior. The UNIX lint program is an early example. Programs work on data.

But data must be expressed and organized within computers before being processed by programs. This organization and expression of data for programs' use is the subject of data structure and representation. Simply put, a data structure tries to store and organize data in a computer in such a way that the data can be used efficiently. There are many types of data structures and each type of structure is suitable for some kinds of applications.

Data structures are computer representations of data. Data structures are used in almost every program. In a sense, no meaningful program can be constructed without the use of some sort of data structure. Some design methods and programming languages even organize an entire software system around data structures.

Fundamentally, data structures are abstractions defined on a collection of data and its associated operations. Often, data structures are designed for improving program or algorithm efficiency. Examples of such data structures include stacks, queues, and heaps. At other times, data structures are used for conceptual unity , such as the name and address of a person.

Often, a data structure can determine whether a program runs in a few seconds or in a few hours or even a few days. From the perspective of physical and logical

ordering, a data structure is either linear or nonlinear. heterogeneous, static vs. dynamic, persistent vs.

stateless structures.

Such analysis usually abstracts away the particular details of a specific computer and focuses on the asymptotic, machine-independent analysis. There are three basic types of analysis. In worst-case analysis, one determines the maximum time or resources required by the algorithm on any input of size n. The third type of analysis is the best-case analysis, in which one determines the minimum time or resources required by the algorithm on any input of size n.

Among the three types of analysis, average-case analysis is the most relevant but also the most difficult to perform. It solves the big problem by recursively solving the smaller problems and combing the solutions to the smaller problems to form the solution to the big problem. The underlying assumption for divide and conquer is that smaller problems are easier to solve. The dynamic programming strategy improves on the divide and conquer strategy by recognizing that some of the sub-problems produced by division may be the same and thus avoids solving the same problems again and again.

# CHAPTER 14
# MATHEMATICAL FOUNDATIONS

Software professionals live with programs. In a very simple language, one can program only for something that follows a well-understood, nonambiguous logic. The Mathematical Foundations knowledge area helps software engineers comprehend this logic, which in turn is translated into programming language code. On the contrary, a software engineer needs to have a precise abstraction on a diverse application domain.

In this KA, techniques that can represent and take forward the reasoning and judgment of a software engineer in a precise manner are defined and discussed. The language and methods of logic that are discussed here allow us to describe mathematical proofs to infer conclusively the absolute truth of certain concepts beyond the numbers. In short, you can write a program for a problem only if it follows some logic. The objective of this KA is to help you develop the skill to identify and describe such logic.
A proposition is a statement that is either true or false, but not both. Some examples of propositions are given below.

It depends on the values of the variables a and b. Propositional logic is the area of logic that deals with propositions. A truth table displays the relationships between the truth values of propositions. A Boolean variable is one whose value is either true or false.

Computer bit operations correspond to logical operations of Boolean variables. The basic logical operators including negation, conjunction , disjunction , exclusive or , and implication are to be studied. Compound propositions may be formed using various logical operators. A compound proposition that is always true is a tautology.

A compound proposition that is always false is a contradiction. A compound proposition that is neither a tautology nor a contradiction is a contingency. Compound propositions that always have the same truth value are called logically equivalent. A predicate is a verb phrase template that describes a property of objects or a relationship among objects represented by the variables.

A variable x that is introduced into a logical expression by a quantifier is bound to the closest enclosing quantifier. A variable is said to be a free variable if it is not bound to a quantifier. Similarly, in a block-structured programming language, a variable in a logical expression refers to the closest quantifier within whose scope it appears.

For example, the assertion a is greater than

There is no mechanism in propositional logic to find out whether or not the two are equivalent to one another. Hence, in propositional logic, each equivalent proposition is treated individually rather than dealing with a general formula that covers all equivalences collectively. Predicate logic is supposed to be a more powerful logic that addresses these issues. In a sense, predicate logic is an extension of propositional logic to formulas involving terms and predicates.
For example, to show that if n is odd then n2

Proof by Contradiction. A proposition p is true by contradiction if proved based on the truth of the implication ¬ p q where q is a contradiction. This is a contradiction. An inference rule is a pattern establishing that if a set of premises are all true, then it can be deduced that a certain conclusion statement is true.

Proof by Induction. Proof by induction is done in two phases. In the second phase, it is established that if the proposition holds for an arbitrary positive integer k, then it must also hold for the next greater integer, k + 1. It may be noted here that, for a proof by mathematical induction, it is not assumed that P is true for all positive integers k.

Proving a theorem or proposition only requires us to establish that if it is assumed P is true for any arbitrary positive integer k, then P is also true. The correctness of mathematical induction as a valid proof technique is beyond discussion of the current text. Let us prove the following proposition using induction.

The basis step together with the inductive step of the proof show that P is true and the conditional statement P P is true for all positive integers k.

+ |B|.
• In general if A1, A2, ….

|A| * |B|.
• In general if A1, A2, …, An are disjoint sets, then |A1 A2 … An| = |A1| * |A2| * ….

Using the product rule, the answer would be

200 * 30 = 6000.

|B| − |A B|.
A phenomenon is said to be random if individual outcomes are uncertain but the long-term pattern of many individual outcomes is predictable. The probability of any outcome for a random phenomenon is the proportion of times the outcome would occur in a very long series of repetitions.

The probability P of any event A satisfies 0

If S is the sample space in a probability model, the P = 1. All possible outcomes together must have probability of 1. If two events have no outcomes in common, the probability that one or the other occurs is the sum of their individual probabilities.
The graph in Figure 14.8 is a simple graph that consists of a set of vertices or nodes and a set of edges connecting unordered pairs. The edges in simple graphs are undirected. Such graphs are also referred to as undirected graphs. For example, in Figure 14.8, e1, may be replaced by e1, as the pair between vertices A and C is unordered.

Such edges are called parallel or multiple edges. Figure 14.9 is a multigraph where edges e3 and e4 are multiple edges. In a weighted graph G = , each edge has a weight associated with it. The weight of an edge typically represents the numeric value associated with the relationship between the corresponding two vertices.

If the vertices A, B, and C represent three cities in a state, the weights, for example, could be the distances in miles between these cities. Let G = be an undirected graph with edge set E. u, v are said to be adjacent or neighbors or connected. edge e is incident with vertices u and v. Let G be a directed graph.

u is adjacent to v, and v is adjacent from u. the initial vertex of e is u. the terminal vertex of e is v.

a loop at a vertex contributes 1 to both indegree and out-degree of this vertex. In an undirected graph, a path of length n from u to v is a sequence of n adjacent edges from vertex u to vertex v. A path traverses the vertices along it.

The graph in Figure 14.8 is a simple graph that consists of a set of vertices or nodes and a set of edges connecting unordered pairs. Alternately, the level of a node X is the length of the unique path from the root of the tree to node X. For example, root node A is at level 0 in Figure 14.15. Nodes B, C, and D are at level 1. The remaining nodes in Figure 14.15 are all at level 2.

The height of a tree is the maximum of the levels of nodes in the tree. A node is called a leaf if it has no children. The degree of a leaf node is 0. K are all leaf nodes with degree 0.

The ancestors or predecessors of a nonroot node X are all the nodes in the path from root to node X. For example, in Figure 14.15, nodes A and D form the set of ancestors for J. The successors or descendents of a node X are all the nodes that have X as its ancestor. For a tree with n nodes, all the remaining n − 1 nodes are successors of the root node. For example, in Figure 14.15, node B has successors in E, F, and G. If node X is an ancestor of node Y, then node Y is a successor of X. Two or more nodes sharing the same parent node are called sibling nodes. For example, in Figure 14.15, nodes E and G are siblings.

However, nodes E and J, though from the same level, are not sibling nodes. Two sibling nodes are of the same level, but two nodes in the same level are not necessarily siblings. A tree is called an ordered tree if the relative position of occurrences of children nodes is significant. A binary tree is formed with zero or more nodes where there is a root node R and all the remaining nodes form a pair of ordered subtrees under the root node.

In a binary tree, no internal node can have more than two children. However, one must consider that besides this criterion in terms of the degree of internal nodes, a binary tree is always ordered. If the positions of the left and right subtrees for any node in the tree are swapped, then a new tree is derived.

# CHAPTER 15
# ENGINEERING FOUNDATIONS

Techniques

The simplest experiments have just two treatments representing two levels of a single independent variable . Once the values of the parameters are known, the distribution of the random variable is completely known and the chance of any event can be computed. The probabilities for a discrete random variable can be computed through the probability mass function, called the pmf. , the probability that the random variable will take that particular value.

Likewise, for a continuous random variable, we have the probability density function, called the pdf. The pdf is very much like density and needs to be integrated over a range to obtain the probability that the continuous random variable lies between certain values. Thus, if the pdf or pmf is known, the chances of the random variable taking a certain set of values may be computed theoretically. Alternately, we might use an interval estimate.

An interval estimate is a random interval with the lower and upper limits of the interval being functions of the sample observations as well as the sample size. A hypothesis is a statement about the possible values of a parameter. In this case, the hypothesis is that the rate of occurrence of defects has reduced. The null hypothesis is the hypothesis of no change and is denoted as H0.

The alternative hypothesis is written as H1. It is important to note that the alternative hypothesis may be one-sided or two-sided. For example, if we have the null hypothesis that the population mean is not less than some given value, the alternative hypothesis would be that it is less than that value and we would have a one-sided test. However, if we have the null hypothesis that the population mean is equal to some given value, the alternative hypothesis would be that it is not equal and we would have a two-sided test .

In order to test some hypotheses, we first compute some statistics. Along with the computation of the statistic, a region is defined such that in case the computed value of the statistic falls in that region, the null hypothesis is rejected. In tests of hypotheses, we need to accept or reject the null hypothesis on the basis of the

evidence obtained. We note that, in general, the alternative hypothesis is the hypothesis of interest.

If the computed value of the statistic does not fall inside the critical region, then we cannot reject the null hypothesis. This indicates that there is not enough evidence to believe that the alternative hypothesis is true. The analysis to find the relationship between two variables is called regression analysis.

Some examples of nominal scales are

On a nominal scale, the names of the different categories are just labels and no relationship between them is assumed.

Measurement in ordinal scale satisfies the transitivity property in the sense that if A > B and B

If a variable is measured in interval scale, most of the usual statistical analyses like mean, standard deviation, correlation, and regression may be carried out on the measured values. An example of a direct measure would be a count of how many times an event occurred, such as the number of defects found in a software product. A derived measure is one that combines direct measures in some way that is consistent with the measurement method. In both cases, the measurement method determines how to make the measurement.

The design of a software product is guided by the features to be included and the quality attributes to be provided. The scope of engineering design is generally viewed as much broader than that of software design. Many disciplines engage in problem solving activities where there is a single correct solution. In engineering, most problems have many solutions and the focus is on finding a feasible solution that best meets the needs presented.

The set of possible solutions is often constrained by explicitly imposed limitations such as cost, available resources, and the state of discipline or domain knowledge.

Their accreditation criteria states

Define the problem. This step includes refining the problem statement to identify the real problem to be solved and setting the design goals and criteria for success. The problem definition is a crucial stage in engineering design. At this stage, the designer attempts to expand his/her knowledge about the problem.

Included in the pertinent information is a list of constraints that must be satisfied by the solution or that may limit the set of feasible solutions. During this stage, different solutions to the same problem are developed. Once alternative solutions have been

identified, they need to be analyzed to identify the solution that best suits the current situation. Physical solutions that involve human users often include analysis of the ergonomics or user friendliness of the proposed solution.

Implement the solution. Implementation refers to development and testing of the proposed solution. Sometimes a preliminary, partial solution called a prototype may be developed initially to test the proposed design solution under certain conditions. Feedback resulting from testing a prototype may be used either to refine the design or drive the selection of an alternative design solution.

One of the most important activities in design is documentation of the design solution as well as of the tradeoffs for the choices made in the design of the solution. This work should be carried out in a manner such that the solution to the design problem can be communicated clearly to others. Engineers use models in many ways as part of their problem solving activities. Models help engineers reason and understand aspects of a problem.

They can also help engineers understand what they do know and what they don't know about the problem at hand.

Analysis

The first step in any root cause analysis effort is to identify the real problem. Techniques such as statement-restatement, why-why diagrams, the revision method, present state and desired state diagrams, and the fresh-eye approach are used to identify and refine the real problem that needs to be addressed. Once the real problem has been identified, then work can begin to determine the cause of the problem. Some of those tools are helpful in identifying the causes for a given problem.

The main line in the diagram represents the problem and the connecting lines represent the factors that led to or influenced the problem. If a problem occurs, then sometimes the checklist can quickly identify tasks that may have been skipped or only partially completed.