CHAPTER - 4

Quality Management and Testing

4.1 Introduction to Quality

Quality is a complex and multifaceted concept that can be described from five different points of view:

- **Transcendental view:** something that immediately recognizes, but cannot explicitly define.
- **User view:** If a product meets an end user's specific goals, it exhibits quality.
- **Manufacturer's view:** Product conforms to the original specification.
- **Product view:** Quality can be tied to inherent characteristics (e.g., functions and features) of a product.
- **Value-based view:** How much a customer is willing to pay for a product?

Thus, Quality can be defined as: Degree of excellence or Fitness for purpose or Best for the customer's use and selling price or Total characteristics of an entity that bear on its ability to satisfy stated or implied needs (ISO).

Quality of design refers to the characteristics that designers specify for a product. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified. *Quality of conformance* focuses on the degree to which the implementation follows the design and the resulting system meets its requirements and performance goals.

A. Software Quality

Software quality can be defined as: *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.* The definition serves to emphasize three important points:

- **Effective software process:** establishes the infrastructure that supports any effort at building a high-quality software product.
- **Useful product:** delivers the content, functions, and features that the end user desires in a reliable, error-free way.
- **Added measurable values:** For **producer**: high-quality software requires less maintenance effort, fewer bug fixes, and reduced customer support. For **end user**: (1) greater software product revenue, (2) better profitability when an application supports a business process, and/or (3) improved availability of information that is crucial or vital for the business.

To achieve high-quality software, four activities must occur: proven software engineering process and practice, solid project management, comprehensive quality control, and the presence of a quality assurance infrastructure.

B. Quality Factors

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes:

1. **Functionality:** The degree to which the software satisfies stated needs as indicated by the following subattributes: suitability, accuracy, interoperability, compliance, and security.

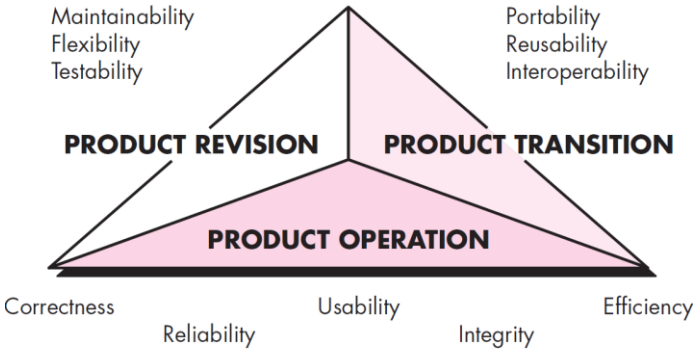


Fig. McCall's software quality factors

2. **Reliability:** The amount of time that the software is available for use as indicated by the following subattributes: maturity, fault tolerance, recoverability.
3. **Usability:** The degree to which the software is easy to use as indicated by the following subattributes: understandability, learnability, operability.
4. **Efficiency:** The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior.
5. **Maintainability:** The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability.
6. **Portability:** The ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, replaceability.

C. Cost of Quality

The software cost depends on *time and money*. The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities and the downstream costs of lack of quality. To understand these costs, an organization must collect metrics to provide a baseline for the current cost of quality, identify opportunities for reducing these costs, and provide a normalized basis of comparison. The cost of quality can be divided into costs associated with prevention, appraisal, and failure.

- *Prevention costs* include (1) the cost of management activities required to plan and coordinate all quality control and quality assurance activities, (2) the cost of added technical activities to develop complete requirements and design models, (3) test planning costs, and (4) the cost of all training associated with these activities.

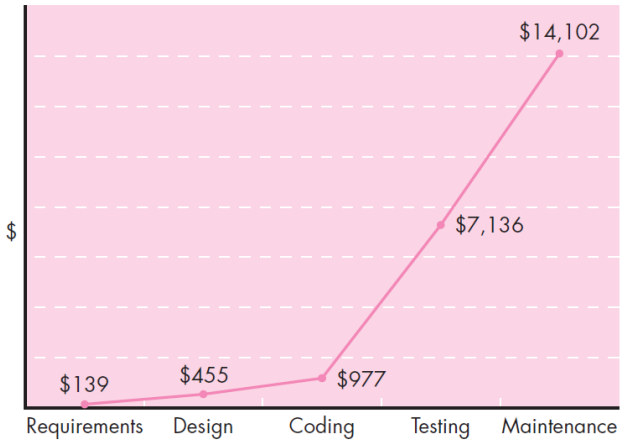


Fig. Relative cost of correcting errors and defects

- *Appraisal costs* include activities to gain insight into product condition the "first time through" each process. Examples of appraisal costs include cost of conducting technical reviews, data collection and metrics evaluation, and testing and debugging.
- *Failure costs* are those that would disappear if no errors appeared before or after shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs.
    - *Internal failure costs* are incurred when you detect an error in a product prior to shipment. Cost required to perform rework (repair) to correct an error is an example of internal failure cost.
    - *External failure costs* are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are complaint resolution, product return and replacement, help line support, and labor costs associated with warranty work. A poor reputation and the resulting loss of business is another external failure cost.

4.2 Software reviews

Technical work needs reviewing for the same reason that pencils need erasers. Thus, software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed. Software reviews "purify" the software engineering activities (*analysis*, *design*, and *coding).* In general, six steps are employed for software review: planning, preparation,

structuring the meeting, noting errors, making corrections (done outside the review), and verifying that corrections have been performed properly. The software review is conducted basically by two methods:

A. Cost impact of software defects

The terms *defect* and *fault* are synonymous within the context of the software process that implies a quality problem discovered *after* the software has been released to end users (or to another framework activity in the software process).

A technical review (TR) is the most effective filter from a quality control standpoint. The primary objective of technical reviews is to find errors during the process so that they do not become defects after release of the software. The obvious benefit of technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process. A number of industry studies indicate that design activities introduce between 50 and 65 percent of all errors during the software process. However, review techniques have been shown to be up to 75 percent effective in uncovering design flaws.

B. Defect Amplification and Removal

A *defect amplification model* can be used to illustrate the generation and detection of errors during the design and code generation actions of a software process. The model is illustrated schematically in figure.

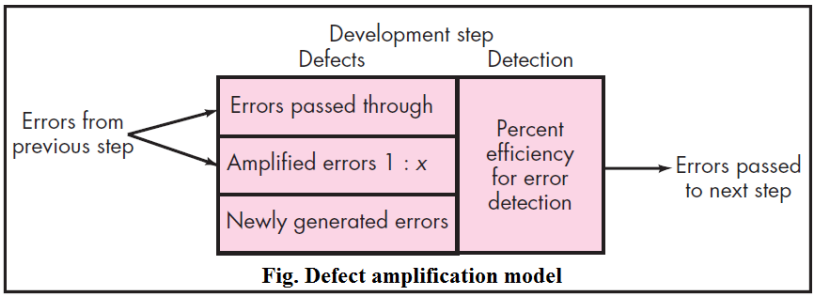


Fig. Defect amplification model

Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified (amplification factor, *x*) by current work. The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review. The following figure illustrates a hypothetical example of defect amplification for a software process with no reviews and reviews are conducted.

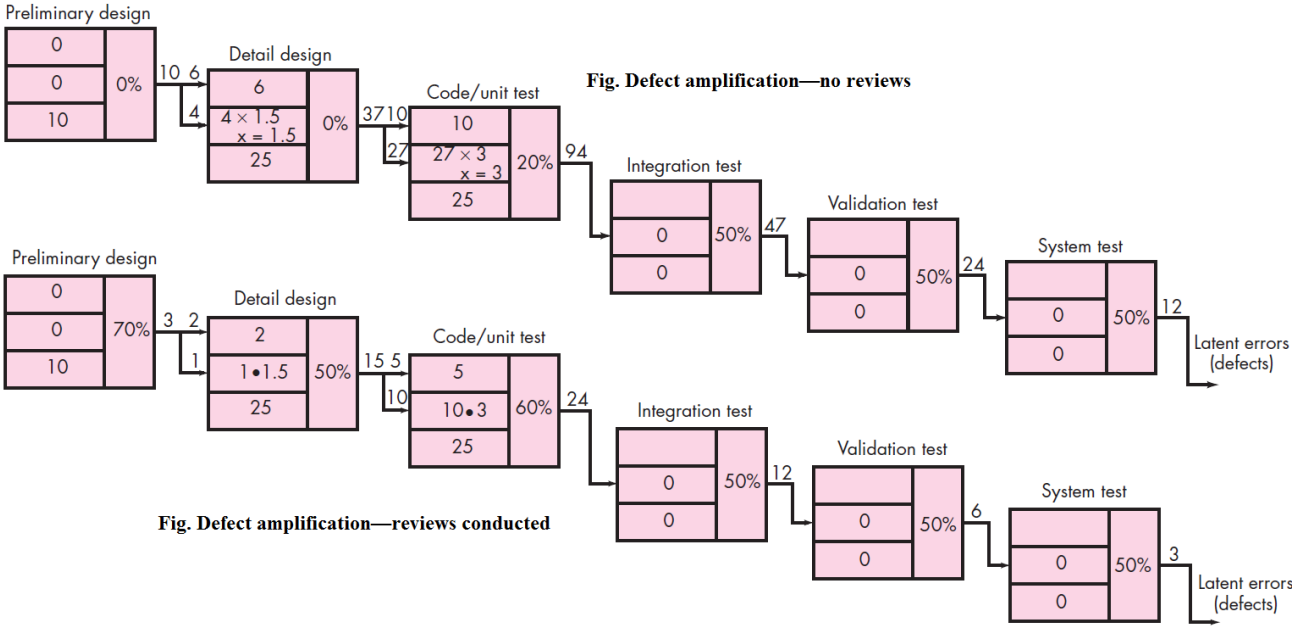


Fig. Defect amplification—no reviews

Fig. Defect amplification—reviews conducted

The effort expended when reviews are used does increase early in the development of a software increment, but this early investment for reviews pays dividends because testing and corrective effort is reduced. As important, the deployment date for development with reviews is sooner than the deployment dates without reviews.

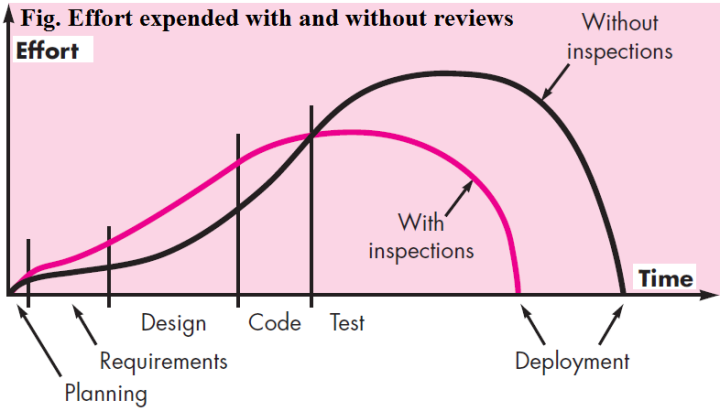


Fig. Effort expended with and without reviews

C. Formal technical reviews

A *formal technical review* (FTR) is a software quality control activity performed by software engineers (and others). The objectives of an FTR are:

1. To uncover errors in function and logic.
2. To verify that the software under review meets its requirements.
3. To ensure that the software has been represented according to predefined standards.
4. To achieve software that is developed in a uniform manner and
5. To make projects more manageable.

In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.

The FTR focuses on relatively small portion of a work product. By narrowing the focus, the FTR has a higher likelihood of uncovering errors. Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended.

a. The review meeting

Every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.

After the development of work product, the review meeting is attended by the review leader, all reviewers, and the producer. At the end of the review, all attendees of the FTR must decide whether to

- Accept the product without further modification,
- Reject the product due to severe errors (once corrected, another review must be performed), or
- Accept the product conditionally (minor errors and no additional review will be required).

b. Review reporting and record keeping

Record keeping and reporting are other quality assurance activities. It is applied to record all issues that have been raised. At the end of review meeting, a review issue list is produced. A review summary report answers the following questions.

- What was reviewed?
- Who reviewed it?
- What were the findings and conclusions?

c. Review guidelines

The following represents a minimum set of guidelines for formal technical reviews:

1. Review the product, not the producer
2. Set an agenda and maintain it
3. Limit debate and rebuttal
4. Enunciate problem areas but don't attempt to solve every problem noted
5. Take written notes
6. Limit the number of participants and insist upon advance preparation
7. Develop a check list each work product that is likely to be reviewed
8. Allocate resources and time schedule for FTRs.
9. Conduct meaningful training for all reviewers
10. Review your earlier reviews

4.3 Software Quality Assurance

The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. Quality assurance establishes the infrastructure to build high quality software that supports solid software engineering methods, project management techniques, and quality control actions. In addition, quality assurance consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control actions.

Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of software product standards, processes, and procedures. SQA is an umbrella activity that *is applied in each step of software engineering*. SQA encompasses procedures for the effective application of methods and tools, oversight of quality control activities such as technical reviews and software testing, procedures for change management, procedures for assuring compliance to standards, and measurement and reporting mechanisms. The basic elements of SQA are: standards, review and audit, testing, error collection and analysis, change management, education and training, vender management, security management, safety and risk management.

Before software quality assurance (SQA) activities can be initiated, it is important to define *software quality* at a number of different levels of abstraction. Once you understand what quality is, a software team must identify a set of SQA activities that will filter errors out of work products before they are passed on.

Product evaluation and process monitoring are the **SQA activities** that assure the software development and control processes. Products are monitored for conformance to standards and processes are monitored for conformance to procedures. Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting. The SQA group serves as the customer's in-house representative i.e. the people who perform SQA must look at the software from the customer's point of view.

A. Element of SQA

Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality. These can be summarized in the following manner:

- **Standards:** The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents, upon following these rules the software will be have better quality.

- **Reviews and audits:** Technical reviews are a quality control activity performed by software engineers for software engineers. Their intent is to uncover errors.
- **Testing:** Software testing is a quality control function that has one primary goal—to find errors.
- **Error/defect collection and analysis:** SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.
- **Change management:** Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality.
- **Education:** Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders.
- **Vendor management:** Three categories of software are acquired from external software vendors—*shrink-wrapped packages* (e.g., Microsoft *Office*), a *tailored shell* that provides a basic skeletal structure that is custom tailored to the needs of a purchaser, and *contracted software* that is custom designed and constructed from specifications provided by the customer organization. The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices.
- **Security management:** The software must have quality to control the cybercrime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for WebApps etc.
- **Safety:** SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.
- **Risk management.** Although the analysis and mitigation of risk is the concern of software engineers, the SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

B. SQA Tasks, Goals and Metrics

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting. Software engineers address quality by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

SQA Tasks

The SQA group is to assist the software team in achieving a high quality end product. The Software Engineering Institute recommends a set of SQA actions that address quality assurance planning, oversight, record keeping, analysis, and reporting. The SQA group generally possess the following tasks:

- Prepares an SQA plan for a project.
- Participates in the development of the project's software process description.
- Reviews software engineering activities to verify compliance with the defined software process.
- Audits designated software work products to verify compliance with those defined as part of the software process.
- Ensures that deviations in software work and work products are documented and handled according to a documented procedure.
- Records any noncompliance and reports to senior management.

Goals	Attributes	Metrics
Requirement quality	Ambiguity, Completeness, Understandability, Volatility, Traceability	Number of ambiguous modifiers, Number of sections/subsections, Number of changes per requirement
Design quality	Architectural integrity, Component completeness, Interface complexity, Patterns	Existence of architectural model, Complexity of procedural design, Layout appropriateness, Number of patterns used
Code quality	Complexity, Maintainability, Understandability, Reusability, Documentation	Cyclomatic complexity, Design factors, Percent internal comments, Percent reused components, Readability index
Quality control effectiveness	Resource allocation, Completion rate, Review effectiveness, Testing effectiveness	Staff hour percentage per activity, Actual vs. budgeted completion time, See review metrics, Origin of error

C. Statistical SQA and Six Sigma

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

1. Information about software errors and defects is collected and categorized.
2. An attempt is made to trace each error and defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the *vital few*).
4. Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.

Six Sigma for Software Engineering

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects' in manufacturing and service-related processes". The term Six Sigma is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

- *Define* customer requirements and deliverables and project goals via well-defined methods of customer communication.
- *Measure* the existing process and its output to determine current quality performance (collect defect metrics).
- *Analyze* defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

- *Improve* the process by eliminating the root causes of defects.
- *Control* the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

D. Quality Assurance System

A *quality assurance system* may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management. Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting their specifications. These systems cover a wide variety of activities encompassing a product's entire life cycle including planning, controlling, measuring, testing and reporting, and improving quality levels throughout the development and manufacturing process. The main importance of quality assurance system can be listed as:

- The customer will be more confidence with the certified organization and their products.
- The production process must be well documented for certification and hence the product of higher quality.
- Point out the weak points of an organization and hence the organization will always intend to reduce them.

ISO 9000 Quality Standards

International Standard Organization (ISO) is a consortium of 63 countries to formulate and foster standardization. It specifies guidelines for maintaining a quality system. The main consideration of ISO is that "*If proper process is followed for production then good quality products are bound to follow automatically.*"

ISO 9000 is a generic name given to a family of standards developed to provide a framework around which a quality management system can effectively be implemented. The ISO 9000 is a series of 3 standards ISO 9001, 9002, and 9003.

- **ISO 9001:** Applies to organizations engaged in *design*, *development*, *production*, and *servicing* of goods, so applicable to most software development organization.
- **ISO 9002:** Applies to organizations which do not design products but are only involved in *production*. E.g. steel and car manufacturing industries (they buy plant and product design from others).
- **ISO 9003:** Applies to organizations involved only in **installation** and **testing** of products.

Shortcomings of ISO Certification

- Requires good software production process to adhere to high quality but no guidelines for defining an appropriate process.
- Variations in the norms of awarding certificates among different ISO agencies in different country and places.
- In manufacturing industry there exists a link between process quality and product quality. Good process quality product but software is not manufactured i.e. it is developed according as the end user requirements.

ISO	Capability Maturity Model (CMM)
▪ Awarded by the international standard body. ▪ No software industry specific. ▪ Confined to quality assurance but not define the ways of improvement. ▪ Generally the validity time duration is of six month.	▪ Assessment for the internal use only. ▪ Developed specifically for software industry use. ▪ Quality assurance and also provides the way of gradual quality achievement. ▪ Generally the validity time deration of one year.

4.4 Software Reliability

Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time". For example: program *X* is estimated to have a reliability of 0.999 over eight elapsed processing hours.

The reliability of a computer program describes the acceptance limit of its overall quality. Unlike many other quality factors, the software reliability can be measured directly and estimated using historical and development data. The software failure determines the software reliability that may be frequently appeared on the program and can be corrected either frequently or after long time.

Measure of Reliability and Availability

In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. Thus, for computer system, a simple measure of reliability is *meantime-between-failure* (MTBF):

MTBF = MTTF + MTTR

Where the acronyms MTTF and MTTR are *mean-time-to-failure* and *mean-time-to repair* respectively

Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as:

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times 100\%$$

Software Safety

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

A modeling and analysis process is conducted as part of software safety. Once hazards are identified and analyzed, safety-related requirements can be specified for the software. Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them.

Software Reliability	Software Safety
▪ The occurrence of a failure does not necessary result in hazard. ▪ Use statistical analysis to determine software failure.	▪ It examines the failure result that lead to a hazard. ▪ Uses the fault tree analysis, real time logic and petri net models to predict hazards.

4.5 Software Testing

Once source code has been generated, software must be tested to uncover as many errors as possible before delivery to your customer. Your goal is to design a series of test cases that have a high likelihood of finding errors. Software testing techniques enter the picture that provide systematic guidance for designing tests that exercise the internal logic of software components, and exercise the input and output domains of the program to uncover errors in program function, behavior and performance. The major software testing principles can be listed as:

- All tests should be traceable to customer requirements.
- Tests should be planned long before testing begins.

- Destructive rather than constructive step.
- Testing should *begin "in the small"* and progress toward testing "*in the large*."
- Exhaustive or complete error free testing is not possible.
- To be most effective, testing should be conducted by an independent third party.
- A good test is *not redundant* because testing time and resources are limited.
- A good test case is one that has a high probability of finding an as-yet-undiscovered error.
- A good test should be neither too simple nor too complex.

Testing is a set of activities that can be planned in advance and conducted systematically. During early stages of testing, a software engineer performs all tests. However, as the testing process progresses, testing specialists may become involved.

Verification and Validation

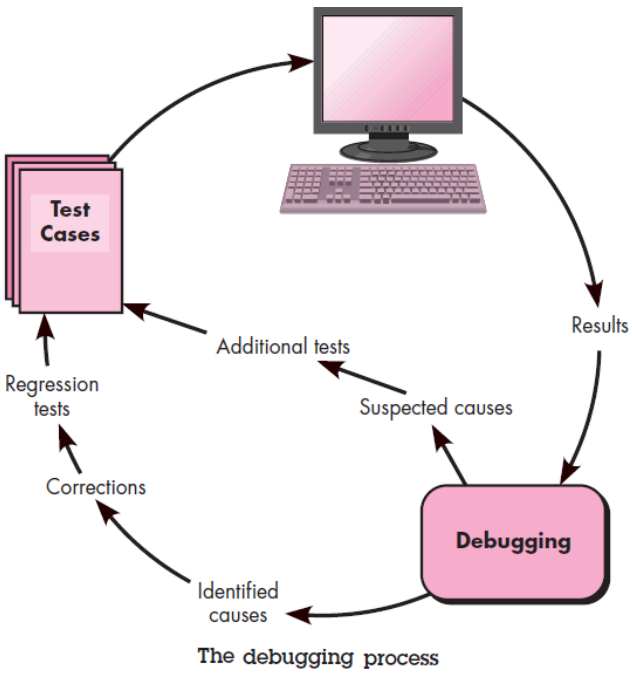
Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). *Verification* refers to the set of tasks that ensure that software correctly implements a specific function. *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:

Verification: "Are we building the product right?"
Validation: "Are we building the right product?"

Verification and validation includes a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing.

The testing and debugging

Debugging occurs as a consequence of successful testing when a test case uncovers an error, debugging results in the removal of the error. Debugging is not testing but always occurs as a consequence of testing. The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered.



The debugging process

The debugging process will either find the cause or correct it, or the cause will not be found. If the cause if not found, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion. As the consequences of error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more. Debugging has one main objective which is to find and correct the cause of a software error. The objective is realized by a combination of systematic evaluation, intuition and luck. The debugging must start from simple to complex. The main three categories for debugging approaches can be given as:

a. Brute Force

The brute force category of debugging is probably the most common and least efficient method for isolating the cause of a software error. We hope that somewhere in the mess of information that is produced we will find a clue that can lead us to the cause of an error. It more frequently leads to wasted effort and time.

b. Backtracking

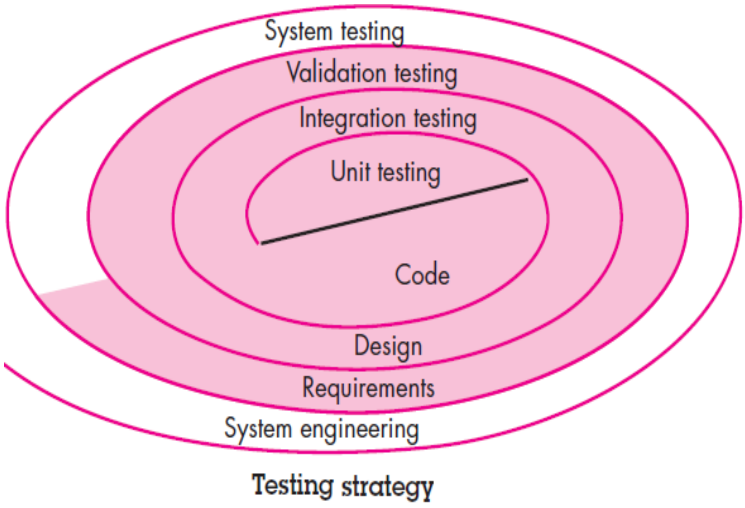
Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found. As the number of source lines increases, the number of potential backward paths may become unmanageably large.

c. Cause Elimination

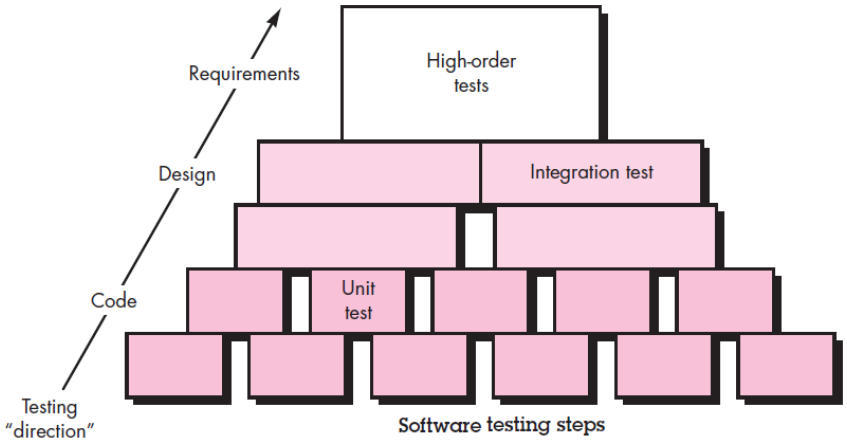
It is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed; tests are conducted to eliminate each.

Software Testing Strategy

It explains the formal plan for testing small component to entire system; test the new changes on the existing modules, requirement of customer involvement time etc. A strategy provides guidance for the practitioner and a set of milestones for the manager. A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements.



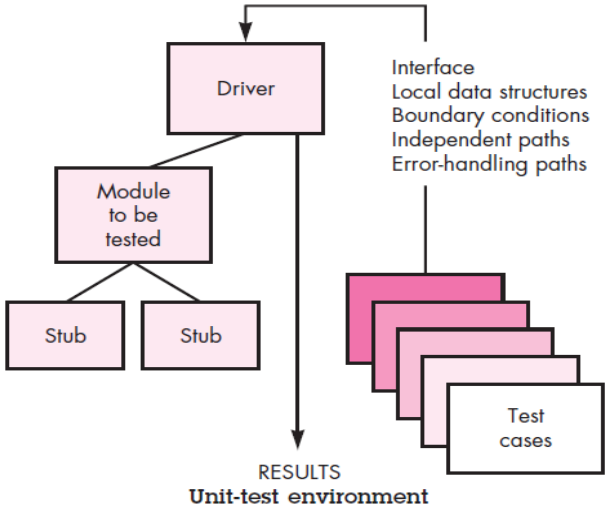
Testing strategy



Software testing steps

- **Unit testing** begins at the vortex of the spiral and concentrates on each unit or component of the software as implemented in source code. Unit test is white-box oriented.
- In **integration testing**, the focus is on design and the construction of the software architecture.
- In **validation testing**, requirements established as part of software requirements analysis are validated against the software that has been constructed. Black-box test case design techniques are the most prevalent during integration.
- At **system testing**, the software and other system elements (hardware, people, and databases) are tested as a whole.

A. Unit Testing



After source level code has been developed, reviewed, and verified, unit test case design begins. Each test case should be coupled with a set of expected results. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing Driver and stub are software, written but not delivered with the final software product. Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure to ensure complete coverage and maximum error detection in parallel for multiple components. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The main considerations towards the unit test are:

- The *module interface* is tested to ensure that information properly flows into and out of the program unit under test.
- The *local data structure* is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- *Boundary conditions* are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- *Independent paths* (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- All *error handling paths* are tested.

B. Integration Testing

After completion of the unit test, there will be problem by the collective result of individual modules on integration. The major problems on putting the individually tested components together will be:

- Data can be lost across an interface
- One module can have an adverse effect on another
- Sub-functions, when combined, may not produce the desired major function
- Global data structures can create problems

The individual components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. Thus, Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The major general steps for the integration testing are listed as:

a. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
b. Depending on the integration approach selected (depth or breadth first), subordinate stubs are replaced one at a time with actual components.

c. Tests are conducted as each component is integrated.
d. On completion of each set of tests, another stub is replaced with the real component.
e. Regression testing may be conducted to ensure that new errors have not been introduced.

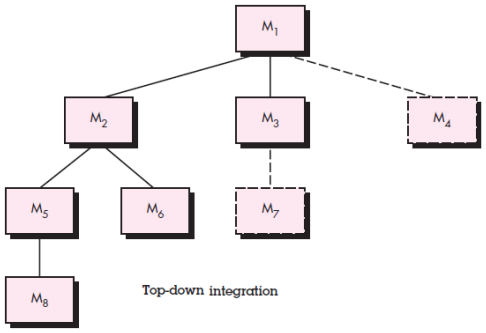
The objective is to take unit-tested components and build a program structure that has been dictated by design. There are basically two approaches for the incremental testing as given below:

- There is often a tendency to attempt non-incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole.
- Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. In the paragraphs that follow, a number of different incremental integration strategies are discussed.

a. Top-down integration testing

It is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner and thus finally the whole programmed structure has been constructed.

- For depth first integration, selecting the left hand path, components M1, M2, M5 would be integrated first. Next, M8 or M6 would be integrated. Then, the central and right hand control paths are built.
- From breath first integration, components M2, M3, and M4 would be integrated first where M1 become the driver. The next control level, M5, M6 (M2 is driver component), and so on.

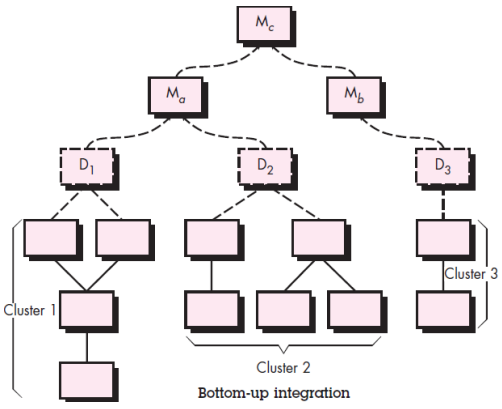


Top-down integration

b. Bottom-up integration testing

This begins construction and testing with *atomic modules* (components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

i. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub-function.
ii. A *driver* (a control program for testing) is written to coordinate test case input and output.
iii. The cluster is tested.
iv. Drivers are removed and clusters are combined moving upward in the program structure.



Bottom-up integration

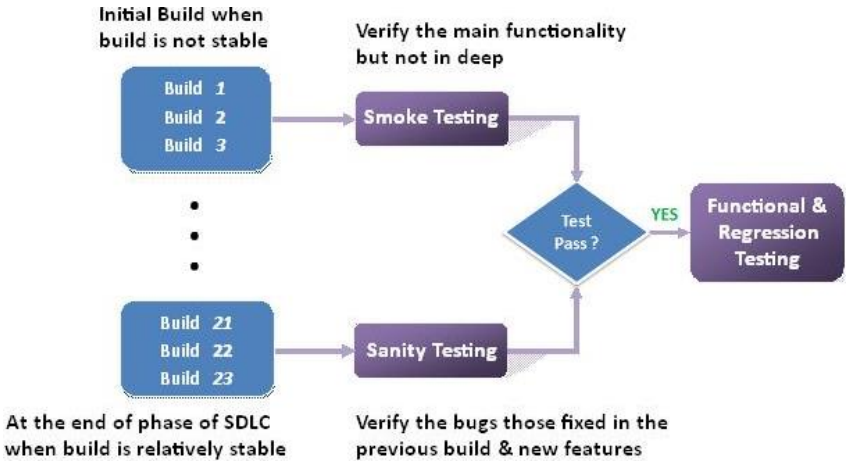
c. Regression testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

Successful tests result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

d. Smoke testing



Smoke testing refers to physical tests similar to detect cracks or breaks of pipes i.e. "the pipes will not leak, the keys seal properly, the circuit will not burn or smoke comes out, or the software will not crash outright". Thus, smoke testing is the initial testing process exercised to check whether the software under test is ready/stable for further testing. In essence, the smoke testing approach encompasses the following activities:

- Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
- A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
- The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

Smoke testing provides a number of benefits when it is applied on complex, time critical software engineering projects: Integration risk is minimized, the quality of the end-product is improved, error diagnosis and correction are simplified & progress is easier to assess.

C. Validation Testing

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between conventional software, object-oriented software, and Web Apps disappears. Testing focuses on user-visible actions and user-recognizable output from the system. Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end user. The validation testing mainly concerned with following tasks:

a. Validation Test Criteria

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. After each validation test case has been conducted, one of two possible conditions exists:

- The function or performance characteristics conform to specification and are accepted
- A deviation from specification is uncovered and a deficiency list is created

b. Configuration Review

An important element of the validation process is a *configuration review.* The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle. The configuration review, sometimes called an *audit.*

c. Alpha and Beta Testing

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements.

- *Alpha test* is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.
- *Beta test* is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. The Beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.

D. System Testing

Software is only one element of a larger computer-based system. Software is incorporated with other system elements (hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers.

A classic system testing problem is "finger-pointing", occurs when an error is uncovered, and each system element developer blames the other for the problem. Thus, system testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. The system testing can be classified in following diversified fields:

a. Recovery Testing

Many computer based systems must recover from faults and resume processing within a pre-specified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur. Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), re-initialization, check-pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

b. Security Testing

Any system that manages sensitive information is a target for illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain. Security testing attempts to verify that protection mechanisms built into a system will protect it from improper penetration. During security testing, the tester plays the role(s) of the individual who desires to penetrate the system.

c. Stress Testing

Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails? Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example:

- special tests may be designed that generate ten interrupts per second, when one or two is the average rate,
- input data rates may be increased by an order of magnitude to determine how input functions will respond
- test cases that require maximum memory or other resources are executed
- test cases that may cause thrashing in a virtual operating system are designed
- Test cases that may cause excessive hunting for disk-resident data are created.

d. Performance Testing

For Real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. External instrumentation can monitor execution intervals, log events (interrupts) as they occur, and sample machine states on a regular basis.

White-box testing

Using white-box testing methods, the software engineer can derive test cases that:

- Guarantee that all independent paths within a module have been exercised at least once,
- Exercise all logical decisions on their true and false sides,
- Execute all loops at their boundaries and within their operational bounds, and
- Exercise internal data structures to ensure their validity

The main Reasons for conducting white-box tests can be listed as:

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. Errors tend to creep into our work when we design and implement function, conditions, or controls that are out of the mainstream. Everyday processing tends to be well understood (and well scrutinized), while "special case" processing tends to fall into the cracks.
- We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. The logical flow of a program is sometimes counterintuitive, meaning that our unconscious assumptions about flow of control and data may lead us to make design errors that are uncovered only once path testing commences.
- Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur.

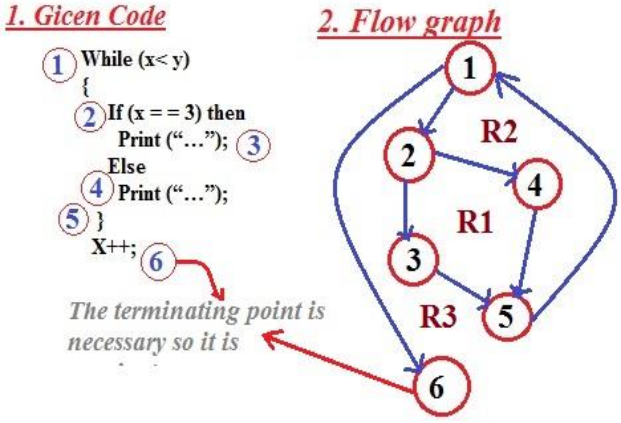
a. Basis path testing

It is white-box testing technique first proposed by Tom McCabe. The basis path method enables test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing. The basics path testing can be done by using the terms: *Flow Graph Notation, Cyclomatic Complexity, Deriving Test Cases, Graph Matrices*

Cyclomatic complexity has a foundation in graph theory and provides us with extremely useful software metric. Complexity is computed in one of three ways:

- The number of regions of the flow graph corresponds to the cyclomatic complexity.
- Cyclomatic complexity, $V(G)$, for a flow graph, G, is defined as $V(G) = E - N + 2$, where E is the number of flow graph edges; N is the number of flow graph nodes.
- Cyclomatic complexity, $V(G)$, for a flow graph, G, is also defined as $V(G) = P + 1$ where P is the number of predicate nodes contained in the flow graph G.

Example



3. Cyclomatic complexity V(G)

a) Here,

Number of region (R) = 3

Thus, V (G) = R = 3

b) Again,

Number of Edges (E) = 7

Number of Nodes (N) = 6

Thus, V (G) = E- N + 2 = 7-6 +2 = 3

C) Also,

The number of Predicate nodes (P) = 2 (i.e. node 1 & 2)

Thus, V (G) = P + 1 = 2 + 1 = 3

4. Basic paths

- Path 1: 1 – 6
- Path 2: 1 – 2 – 3 – 5 – 1 – 6
- Path 3: 1 – 2 – 4 – 5 – 1 – 6

5) Test Cases

- For Path: 1 – 6
  A test case is: X = 4 & Y = 3
- For Path: 1 – 2 – 3 – 5 – 1 – 6
  A test case is: X = 3
- For Path: 1 – 2 – 3 – 5 – 1 – 6
  A test case is: X = 2 or any other except 3

Note:

- Each & every nodes must be terminating on one node so we need the node – 6 on the given example.
- The ***Cyclomatic complexity*** must be equal in all these three cases.
- If there are other paths beside the ***Basic path*** then they are redundant path.
- The ***test case design*** has the highest probability of finding the most errors with a *minimum amount of time and effort.*

b. Control structure testing

The basis path testing technique is one of a number of techniques for control structure testing but it is not sufficient on control structure testing. Some popular control structure testing techniques are described below:

Condition Testing

Condition testing is a test-case design method that exercises the logical conditions contained in a program module. A relational expression takes the form:

*E*1 <relational-operator> *E*2

Where *E*1 and *E*2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠, >, or ≥. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&), and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.

Data Flow Testing

The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with *S* as its statement number,

DEF(*S*) = {*X* | statement *S* contains a definition of *X*}
USE(*S*) = {*X* | statement *S* contains a use of *X*}

If statement *S* is an *if* or *loop statement,* its DEF set is empty and its USE set is based on the condition of statement *S.* The definition of variable *X* at statement *S* is said to be *live* at statement *S'* if there exists a path from statement *S* to statement *S'* that contains no other definition of *X*.

Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests. *Loop testing* is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.

- **Simple loops:** They are sequentially tested for the data of desired domain.
- **Nested loop:** They are started to test from innermost loop.
- **Concatenated loop:** Multiple loops are tested simultaneously if there is no dependency exists.
- **Unstructured loops:** Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs

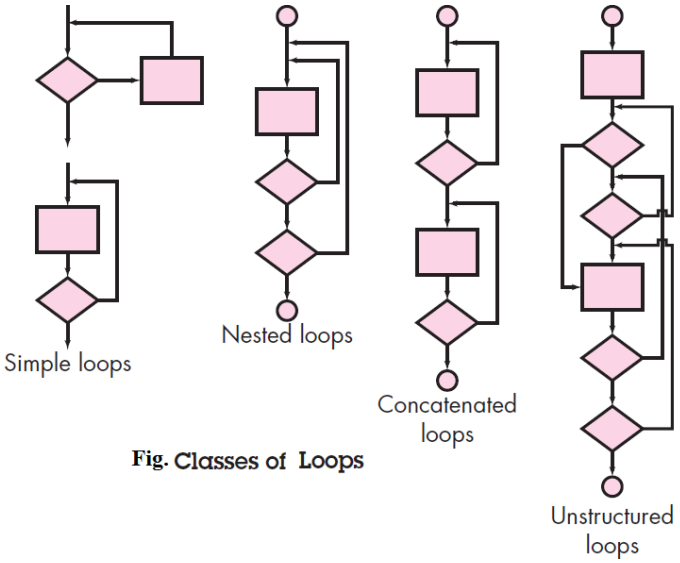


Fig. Classes of Loops

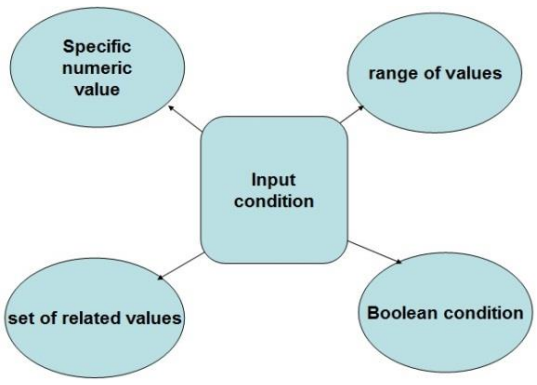
Black-box testing

Black box testing tends to be applied during the later stage of testing (*integration & system testing*). The black box testing can be classified into main two categories as follows:

B. Equivalence partitioning

Divides the input domain of a program into classes of data from which test cases can be derived. An equivalence class represents a set of valid or invalid states for input conditions. Equivalence classes may be defined according to the following guidelines:

- Input condition specifies a range, one valid and two invalid equivalence classes are defined.
- Input condition needs specific value, one valid and two invalid equivalence classes defined.
- Input condition specifies member of a set, one valid/one invalid equivalence class defined.
- Input condition is Boolean, one valid and one invalid class are defined.



The input conditions associated with each data element for the banking application can be specified as:

i. **area code**:	Input condition, *Boolean*-area code may/may not be present. Input condition, *range*—values defined between 200 and 999, with specific exceptions.
ii. **prefix**:	Input condition, *range*—specified value >200
iii. **suffix**:	Input condition, value—four-digit length
iv. **password**:	Input condition, *Boolean*-password may/may not be present. Input condition, *value*—six-character string.
v. **command**:	Input condition, *set*-containing commands noted previously.

C. Boundary value analysis

A greater number of errors tend to occur at the boundaries of the input domain rather than in the center. Boundary value analysis leads to a selection of test cases that exercise bounding values. BVA leads to the selection of test cases at the *edges* of the class. The boundary value analysis can be decomposing in following different prospective:

a. Testing GUIs

Finite state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI.

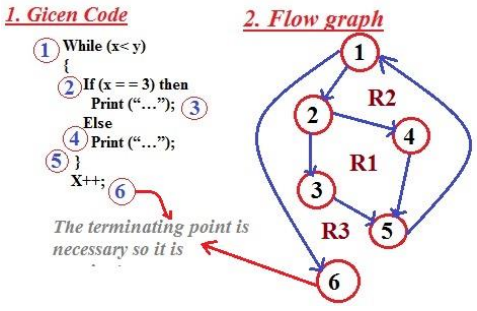
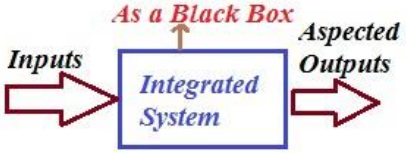
b. Testing of Client/Server Architectures

The major difficulties in testing of C/S architectures and the software can be listed as:

- The distributed nature of client/server environments
- The performance issues associated with transaction processing,
- The potential presence of a number of different hardware platforms,
- The complexities of network communication
- The need to service multiple clients from a centralized or distributed database, the coordination requirements imposed on the server all combine to make

c. Testing for Real-Time Systems

The time-dependent, asynchronous nature of many Real-time applications adds a new and potentially difficult element to the testing mixes—time. Not only does the test case designer have to consider white- and black-box test cases but also event handling (interrupt processing), the timing of the data, and the parallelism of the tasks (processes) that handle the data. Four step strategy for testing Real-time Systems: Task testing, Behavioral testing, Inter-task testing, System Testing

White Box Testing Model	**Black Box Testing Model**
1. It concerned on structural testing of module which explains the internal logic of a program and particular execution path. 2. White box testing is performed early in the testing process (*unit testing*). 3. It is also known as "*glass testing*", "*Clear box testing*" or "*Structural Testing*" as it specify the control structure. 4. White Box Testing require the programming knowledge to test the modules as it tester must correct the error codes. 5. It will be effective for small line or piece of codes. 6. Here, we have to test the logic with specifying the large quantity of test cases.	1. It determines the functional requirement of the software that involves the input conditions, interface errors, error in data structure etc. 2. Black box testing tends to be applied during the later stage of testing (*integration & system testing*) because it focuses on the information domain. 3. It concerns to the functional performance so it is also known as "*Behavioral testing*". 4. Here, the application can be tested without programming knowledge, so only external functional behaviors & GUI features are examined. 5. We can use the Black Box Testing for any small or large module. 6. It can be done by selecting simple test cases such as particular input is given and the prescribed output is examined.
	

4.6 Software Configuration Management

When you build computer software, change happens. And because it happens, you need to manage it effectively. Software configuration management (SCM), also called change management, is a set of activities designed to manage change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

Software configuration management (SCM) is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to (1) identify change, (2) control change, (3) ensure that change is being properly implemented, and (4) report changes to others who may have an interest. The four fundamental sources of change on software product will be:

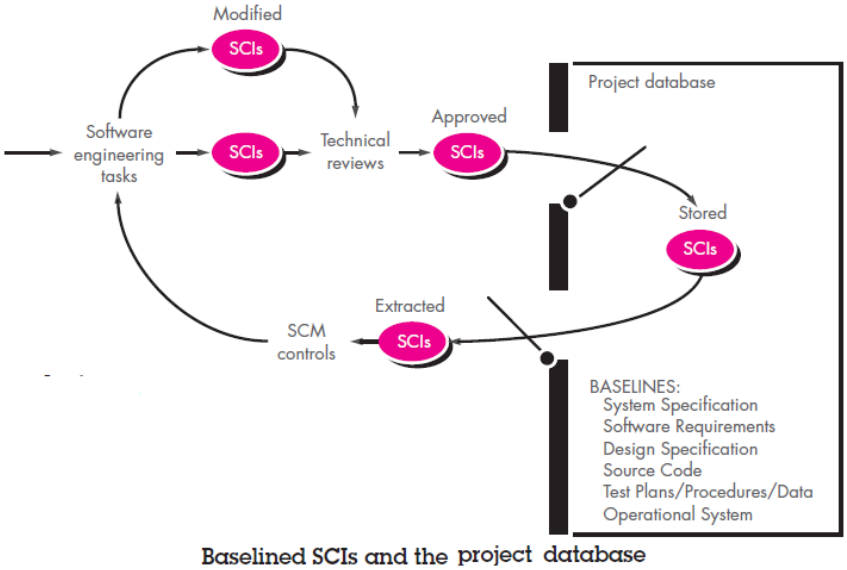
- New business or market conditions dictate changes in product requirements or business rules.
- New stakeholder needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
- Budgetary or scheduling constraints cause a redefinition of the system or product.

Baselines

In the context of software engineering, a baseline is a milestone in the development of software. A baseline is marked by the delivery of one or more software configuration items that have been approved as a consequence of a technical review. For example, the elements of a design model have been documented and reviewed. Errors are found and corrected. Once all parts of the model have been reviewed, corrected, and then approved, the design model becomes a baseline. Further changes to the program architecture (documented in the design model) can be made only after each has been evaluated and approved.

Before a software configuration item (SCI) becomes a baseline (or placed in a *project database*, also called a *project library* or *software repository)*, change can be made quickly or informally. However, once a baseline is established, we figuratively pass through a swinging one way door. Now, the formal procedure must be applied to evaluate and verify each change. The procedure to make changes on baseline product will be:

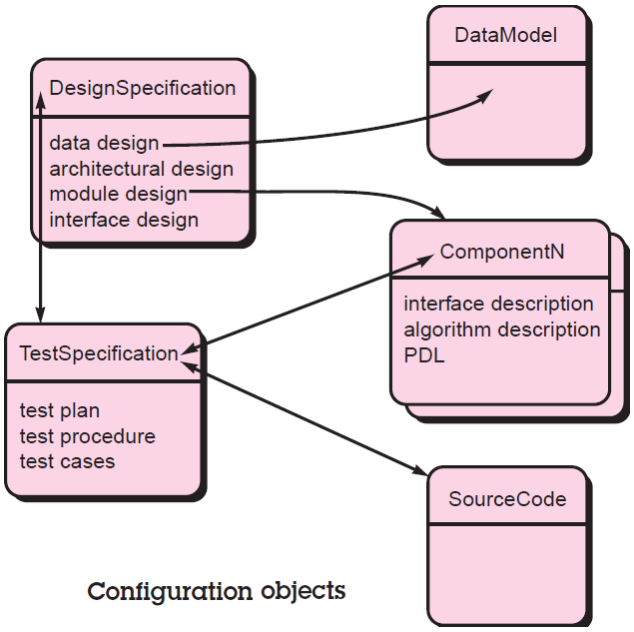
- Send a private copy of module needing modification to change control board (CCB)
- Get permission from the CCB for restoring change module
- After review from CCB, the manager updates the old baseline by restore operation.



Baselined SCIs and the project database

Software Configuration Items (SCI)

The first law of software engineering is "No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle". The change is a fact of life in software development. Customer wants to modify the requirements, developer wants to modify the technical approach, manager wants to modify the project strategy etc.



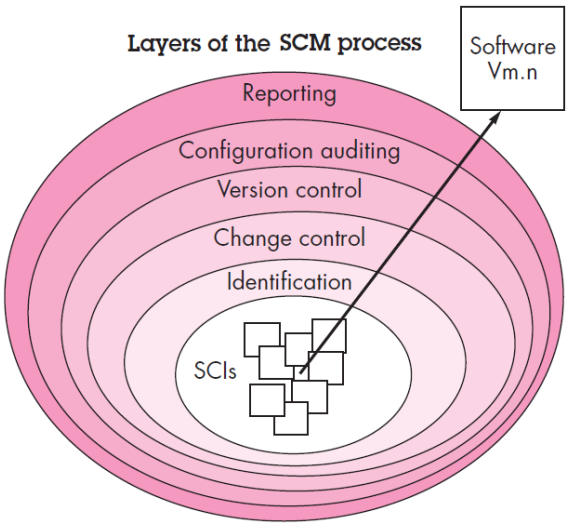
Configuration objects

A *configuration object* has a name, attributes, and is "connected" to other objects by relationships. Referring to Figure, the configuration objects, **Design-Specification, Data-Model, Component-N, Source-Code,** and **Test-Specification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a compositional relation. That is, **Data-Model** and **Component-N** are part of the object **Design-Specification.** A double-headed straight arrow indicates an interrelationship. If a change were made to the **Source-Code** object, the interrelationships enable you to determine what other objects (and SCIs) might be affected.

The SCM Process

The software configuration management process defines a series of tasks that have four primary objectives:

1. to identify all items that collectively define the software configuration,
2. to manage changes to one or more of these items,
3. to facilitate the construction of different versions of an application, and
4. to ensure that software quality is maintained as the configuration evolves over time.



a. Identification

To control and manage software configuration items, each should be separately named and then organized using an object-oriented approach. Two types of objects can be identified: basic objects and aggregate objects.

- A *basic object* is a unit of information that you create during analysis, design, code, or test. For example, a basic object might be a section of a requirements specification, part of a design model, source code for a component, or a suite of test cases that are used to exercise the code.
- An *aggregate object* is a collection of basic objects and other aggregate objects. For example, a *DesignSpecification* is an aggregate object.

b. Change control

Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change, and culminates with a controlled update of the SCI that is to be changed.

c. Version control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. A version control system implements or is directly integrated with four major capabilities:

1. A project database (repository) that stores all relevant configuration objects,
2. A *version management* capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions),
3. A *make facility* that enables you to collect all relevant configuration objects and construct a specific version of the software.

In addition, version control and change control systems often implement an *issues tracking* (also called *bug tracking*) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

d. Configuration Audit

The configuration audit is an SQA activity that helps to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

e. Status Reporting

Configuration status reporting (*status accounting*) is an SCM task that answers the following questions: (1) What happened? (2) Who did it? (3) When did it happen? (4) What else will be affected?

The flow of information for configuration status reporting (CSR) is illustrated in figure.

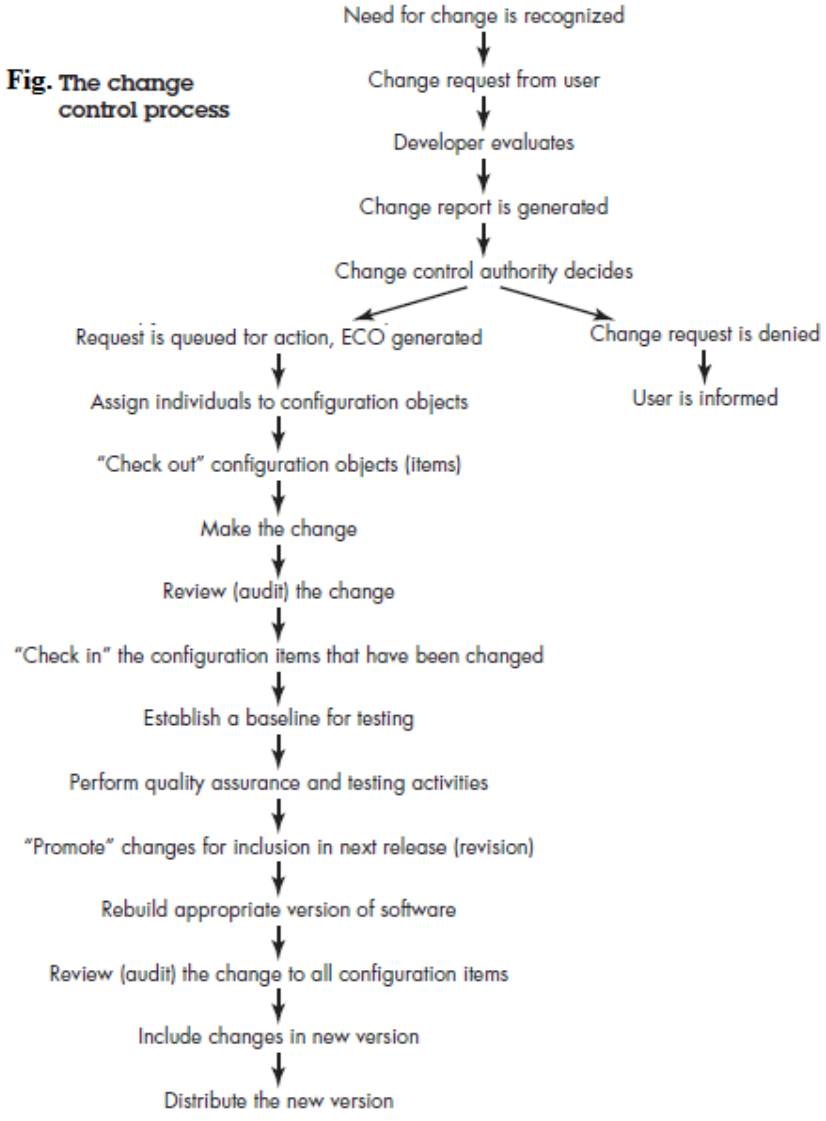


Fig. The change control process