INTRODUCTION

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. Although some detailed design may be performed prior to construction, much design work is performed during the construction activity. Thus, the Software Construction KA is closely linked to the Software Design KA. Throughout construction, software engineers both unit test and integration test their work. Thus, the Software construction KA is closely linked to the Software Testing KA as well. Software construction typically produces the highest number of configuration items that need to be managed in a software project (source files, documentation, test cases, and so on). Thus, the Software Construction KA is also closely linked to the Software Configuration Management KA. While software quality is important in all the KAs, code is the ultimate deliverable of a software project, and thus the Software Quality KA is closely linked to the Software Construction KA. Since software construction requires knowledge of algorithms and of coding practices, it is closely related to the Computing Foundations KA, which is concerned with the computer science foundations that support the design and construction of software products. It is also related to project management, insofar as the management of construction can present considerable challenges.

The importance of software construction

1) Construction is a large part of software development. Depending on the size of the project, construction typically takes 30 to 80 percent of total time spent on a project.

2) Construction is the central activity in software development
   - Requirement and architecture
   - Construction
   - System Testing

SOFTWARE CONSTRUCTION FUNDAMENTALS

Software Construction fundamentals includes:

- minimizing complexity
- anticipating change
- constructing for verification
- reuse
- standards in construction.

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

1.1 **Minimizing Complexity**

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. In software construction, reduced complexity is achieved through emphasizing code creation that is simple and readable rather than clever. It is accomplished through making use of standards (see section 1.5, Standards in Construction), modular design (see section 3.1, Construction Design), and numerous other specific techniques (see section 3.3, Coding). It is also supported by construction-focused quality techniques (see section 3.7, Construction Quality).

1.2 **Anticipating Change**

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. Anticipating change helps software engineers build extensible software, which means they can enhance a software product without disrupting the underlying structure. Anticipating change is supported by many specific techniques (see section 3.3, Coding).

1.3 Constructing for Verification

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

1.4 Reuse

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. Reuse has two closely related facets:"construction for reuse" and "construction with reuse." The former means to create reusable software assets, while the latter means to reuse software assets in the construction of a new solution. Reuse often transcends the boundary of projects, which means reused assets can be constructed in other projects or organizations.

1.5 Standards in Construction

Applying external or internal development standards during construction helps achieve a project’s objectives for efficiency, quality, and cost. Specifically, the choices of allowable programming language subsets and usage standards are important aids in achieving higher security. Standards that directly affect construction issues include

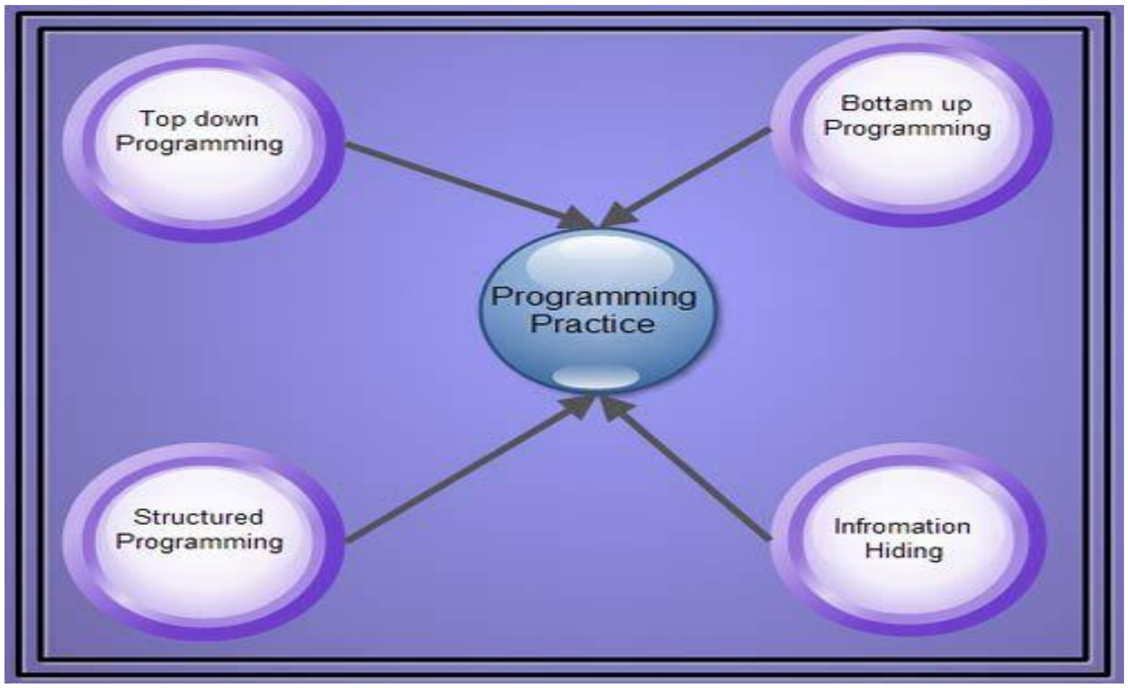
- communication methods (for example, standards for document formats and contents)

- 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)).

Use of external standards. Construction depends on the use of external standards for construction languages, construction tools, technical interfaces, and interactions between the Software Construction KA and other KAs. Standards come from numerous sources, including hardware and software interface specifications (such as the Object Management Group (OMG)) and international organizations (such as the IEEE or ISO). Use of internal standards. Standards may also be created on an organizational basis at the corporate level or for use on specific projects. These standards support coordination of group activities, minimizing complexity, anticipating change, and constructing for verification.

Programming Practices with Top-Down, Bottom-Up, Structured Programming, and Information Hiding

Programming refers to the method of creating a sequence of instructions to enable the computer to perform a task. It is done by developing logic and then writing instructions in a programming language. A program can be written using various programming practices available. A **programming practice** refers to the way of writing a program and is used along with coding style guidelines. Some of the commonly used programming practices include top-down programming, bottom-up programming, structured programming, and information hiding.



Top-down Programming

Top-down programming focuses on the use of modules. It is therefore also known as modular programming. The program is broken up into small modules so that it is easy to trace a particular segment of code in the software program. The modules at the top level are those that perform general tasks and proceed to other modules to perform a particular task. Each module is based on the functionality of its functions and procedures. In this approach, programming begins from the top level of hierarchy and progresses towards the lower levels. The implementation of modules starts with the main module. After the implementation of the main module, the subordinate modules are implemented and the process follows in this way. In top-down programming, there is a risk of implementing data structures as the modules are dependent on each other and they nave to share one or more functions and procedures. In this way, the functions and procedures are globally visible. In addition to modules, the top-down programming uses sequences and the nested levels of commands.

Bottom-up Programming

Bottom-up programming refers to the style of programming where an application is constructed with the description of modules. The description begins at the bottom of the hierarchy of modules and progresses through higher levels until it reaches the top. Bottom-up programming is just the opposite of top-down programming. Here, the program modules are more general and reusable than top-down programming.

It is easier to construct functions in bottom-up manner. This is because bottom-up programming requires a way of passing complicated arguments between functions. It takes the form of constructing abstract data types in languages such as C++ or Java, which can be used to implement an entire class of applications and not only the one that is to be written. It therefore becomes easier to add new features in a bottom-up approach than in a top-down programming approach.

Structured Programming

Structured programming is concerned with the structures used in a computer program. Generally, structures of computer program comprise decisions, sequences, and loops. The **decision structures** are used for conditional execution of statements (for example, 'if statement). The **sequence structures** are used for the sequentially executed statements. The **loop structures** are used for performing some repetitive tasks in the program.

Structured programming forces a logical structure in the program to be written in an efficient and understandable manner. The purpose of structured programming is to make the software code easy to modify when required. Some languages such as Ada, Pascal, and dBase are designed with features that implement the logical program structure in the software code. Primarily, the structured programming focuses on reducing the following statements from the program.

1. 'GOTO' statements.
2. 'Break' or 'Continue' outside the loops.
3. Multiple exit points to a function, procedure, or subroutine. For example, multiple 'Return' statements should not be used.
4. Multiple entry points to a function, procedure, or a subroutine.

Structured programming generally makes use of top-down design because program structure is divided into separate subsections. A defined function or set of similar functions is kept separately. Due to this separation of functions, they are easily loaded in the memory. In addition, these functions can be reused in one or more programs. Each module is tested individually. After testing, they are integrated with other modules to achieve an overall program structure. Note that a key characteristic of a structured statement is the presence of single entry and single exit point. This characteristic implies that during execution, a structured statement starts from one defined point and terminates at another defined point.

Information Hiding

Information hiding focuses on hiding the non-essential details of functions and code in a program so that they are inaccessible to other components of the software. A software developer applies information hiding in software design and coding to hide unnecessary details from the rest of the program. The objective of information hiding is to minimize complexities among different modules of the software. Note that complexities arise when one program or module in software is dependent on several other programs and modules.

Information hiding is implemented with the help of interfaces. An interface is a medium of interaction for software components that are using the properties of the software modules containing data. The implementation of interfaces depends on the syntax and process. Examples of interface include constants, data types, types of procedures, and so on. Interfaces protect other parts of programs when a software design is changed.

Generally, the interfaces act as a foundation to modular programming (top-down programming) and object-oriented programming. In object-oriented programming, interface of an object comprises a set of methods, which are used to interact with the objects of the software programs. Using information hiding, a single program is divided into several modules. These modules are independent of each other and can be used interchangeably in other software programs.

To understand the concept of information hiding, let us consider an example of a program written for 'car'. The program can be organized in several ways. One is to arrange modules without using information hiding. In this case, the modules can be created as 'front part', 'middle part', and 'rear part'. On the other hand, creating modules using information hiding includes specifying names of modules such as 'engine' and 'steering'.

On comparison, it is found that modules created without using information hiding affect other modules. This is because when a module is modified, it affects the data, which does not require modification. However, if modules are created using information hiding, then modules are concerned only with specific segments of the program and not the whole program or other parts of the program. In our example, this statement means that the module 'engine' does not have any affect on the module 'steering'.

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.

Some projects allocate considerable design activity to construction, while others allocate design to a phase explicitly focused on design. Regardless of the exact allocation, some detailed design work will occur at the construction level, and that design work tends to be dictated by constraints imposed by the real-world problem that is being addressed by the software. Just as construction workers building a physical structure must make small-scale modifications to account for unanticipated gaps in the builder's plans, software construction workers must make modifications on a smaller or larger scale to flesh out details of the software design during construction. The details of the design activity at the construction level are essentially the same as described in the Software Design KA, but they are applied on a smaller scale of algorithms, data structures, and interfaces.

Construction languages include all forms of communication by which a human can specify an executable problem solution to a problem. Construction languages and their implementations (for example, compilers) can affect software quality attributes of performance, reliability, portability, and so forth. They can be serious contributors to security vulnerabilities.

The simplest type of construction language is a *configuration language,* in which software engineers choose from a limited set of predefined options to create new or custom software installations. The text-based configuration files used in both the Windows and Unix operating systems are examples of this, and the menu-style selection lists of some program generators constitute another example of a configuration language. Toolkit languages are used to build applications out of elements in toolkits (integrated sets of application-specific reusable parts); they are more complex than configuration languages.

Toolkit languages may be explicitly defined as application programming languages, or the applications may simply be implied by a toolkit's set of interfaces.

Scripting languages are commonly used kinds of application programming languages. In some scripting languages, scripts are called batch files or macros. *Programming languages* are the most flexible type of construction languages. They also contain the least amount of information about specific application areas and development processes therefore, they require the most training and skill to use effectively. The choice of programming language can have a large effect on the likelihood of vulnerabilities being introduced during coding—for example, uncritical usage of C and C++ are questionable choices from a security viewpoint. There are three general kinds of notation used for programming languages, namely

- linguistic (e.g., C/C++, Java)
- formal (e.g., Event-B)
- visual (e.g., MatLab).

The following considerations apply to the software construction coding activity:

- Techniques for creating understandable source code, including naming conventions and source code layout;
- Use of classes, enumerated types, variables, named constants, and other similar entities;
- Use of control structures;

- Handling of error conditions—both anticipated and exceptional (input of bad data, for example);
- Prevention of code-level security breaches (buffer overflows or array index bounds, for example);
- Resource usage via use of exclusion mechanisms and discipline in accessing serially reusable resources (including threads and database locks);
- Source code organization (into statements, routines, classes, packages, or other structures);
- Code documentation;
- Code tuning

SOFTWARE TESTING

Software testing is the process of executing a program to locate an error. a good test case is one that has a high probability of finding undiscovered error. it is impossible to continue testing the software until all faults are detected and removed, as testing of all inputs would require millions of years! Therefore failure probabilities must be inferred from testing a sample of all possible input states called input space. In other words, input space is the set of all possible input states. Similarly, output space is the set of all possible output states for given software.

Software testing is a process, to evaluate the functionality of a software application with an intent to find whether the developed software met the specified requirements or not and to identify the defects to ensure that the product is defect free in order to produce the quality product.

Software Testing Objectives

Software Testing has different goals and objectives.The major objectives of Software testing are as follows:

- **Finding defects** which may get created by the programmer while developing the software.
- Gaining confidence in and providing information about the level of **quality**.
- To prevent defects.
- To make sure that the end result meets the business and user requirements.
- To ensure that it satisfies the BRS that is Business Requirement Specification and SRS that is System Requirement Specifications.
- To gain the confidence of the customers by providing them a quality product.

Seven Principles of Software Testing:

There are seven principles in software testing:

1) Testing shows presence of defects
2) Exhaustive testing is not possible
3) Early testing
4) Defect clustering
5) Pesticide paradox
6) Testing is context dependent
7) Absence of errors fallacy

- **Testing shows presence of defects:** The goal of software testing is to make the software fail. Software testing reduces the presence of defects. Software testing talks about the presence of defects and doesn't talk about the absence of defects. Software testing can ensure that defects are present but it can not prove that software is defects free. Even multiple testing can never ensure that software is 100% bug-free. Testing can reduce the number of defects but not removes all defects.
- **Exhaustive testing is not possible:** It is the process of testing the functionality of a software in all possible inputs (valid or invalid) and pre-conditions is known as exhaustive testing. Exhaustive testing is impossible means the software can never test at every test cases. It can test only some test cases and assume that software is correct and it will produce the correct output in every test cases. If the software will test every test cases then it will take more cost, effort, etc. and which is impractical.
- **Early Testing:** To find the defect in the software, early test activity shall be started. The defect detected in early phases of SDLC will very less expensive. For better performance of software, software testing will start at initial phase i.e. testing will perform at the requirement analysis phase.
- **Defect clustering:** In a project, a small number of the module can contain most of the defects. Pareto Principle to software testing state that 80% of software defect comes from 20% of modules.
- **Pesticide paradox:** Repeating the same test cases again and again will not find new bugs. So it is necessary to review the test cases and add or update test cases to find new bugs.
- **Testing is context dependent:** Testing approach depends on context of software developed. Different types of software need to perform different types of testing. For example, The testing of the e-commerce site is different from the testing of the Android application.
- **Absence of errors fallacy:** If a built software is 99% bug-free but it does not follow the user requirement then it is unusable. It is not only necessary that software is 99% bug-free but it also mandatory to fulfill all the customer requirements.

Verification and Validation:

These two terminologies are also very important in software testing. So, what they denote, let's dig deep into it. Verification focuses on the concern that whether: "Developers are building the system right?". It ensures whether the system is meeting all the functionality as per requirement. The verification process takes place in the 1st position which includes documentation check, coding, etc.

On the other hand, validation deals with whether: "Developers are building the right software?". This ensures whether all functionalities are properly behaving or not. The validation process gets performed after the verification and largely engages in the inspection of your overall software.

VERIFICATION	VALIDATION
It includes checking documents, design, codes and programs.	It includes testing and validating the actual product.
Verification is the static testing.	Validation is the dynamic testing.
It does *not* include the execution of the code.	It includes the execution of the code.
Methods used in verification are reviews, walkthroughs, inspections and desk-checking.	Methods used in validation are Black Box Testing, White Box Testing and non-functional testing.
It checks whether the software conforms to specifications or not.	It checks whether the software meets the requirements and expectations of a customer or not.
It can find the bugs in the early stage of the development.	It can only find the bugs that could not be found by the verification process.
The goal of verification is application and software architecture and specification.	The goal of validation is an actual product.
Quality assurance team does verification.	Validation is executed on software code with the help of testing team.

BLACK BOX TESTING

Black-box testing is a method of software testing that examines the functionality of an application based on the specifications. It is also known as Specifications based testing. Independent Testing Team usually performs this type of testing during the software testing life cycle.

This method of test can be applied to each and every level of software testing such as unit, integration, system and acceptance testing.

There are different techniques involved in Black Box testing.

- Equivalence Class
- Boundary Value Analysis
- Domain Tests
- Orthogonal Arrays
- Decision Tables
- State Models
- Exploratory Testing
- All-pairs testing

WHITE BOX TESTING

White box testing is a testing technique, that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing.

White Box Testing Techniques:

- **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.
- **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.
- **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch is covered.

Advantages of White Box Testing:

- Forces test developer to reason carefully about implementation.
- Reveals errors in "hidden" code.
- Spots the Dead Code or other issues with respect to best programming practices.

Disadvantages of White Box Testing:

- Expensive as one has to spend both time and money to perform white box testing.

- Every possibility that few lines of code are missed accidentally.
- In-depth knowledge about the programming language is necessary to perform white box testing.

LEVEL OF TESTING

Levels of testing include different methodologies that can be used while conducting software testing. The main levels of software testing are −

- Functional Testing
- Non-functional Testing

Functional Testing

This is a type of black-box testing that is based on the specifications of the software that is to be tested. The application is tested by providing input and then the results are examined that need to conform to the functionality it was intended for. Functional testing of a software is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

There are five steps that are involved while testing an application for functionality.

Steps	Description
I	The determination of the functionality that the intended application is meant to perform.
II	The creation of test data based on the specifications of the application.
III	The output based on the test data and the specifications of the application.
IV	The writing of test scenarios and the execution of test cases.
V	The comparison of actual and expected results based on the executed test cases.

An effective testing practice will see the above steps applied to the testing policies of every organization and hence it will make sure that the organization maintains the strictest of standards when it comes to software quality.

Unit Testing

This type of testing is performed by developers before the setup is handed over to the testing team to formally execute the test cases. Unit testing is performed by the respective developers on the individual units of source code assigned areas. The developers use test data that is different from the test data of the quality assurance team.

The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality.

Limitations of Unit Testing

Testing cannot catch each and every bug in an application. It is impossible to evaluate every execution path in every software application. The same is the case with unit testing.

There is a limit to the number of scenarios and test data that a developer can use to verify a source code. After having exhausted all the options, there is no choice but to stop unit testing and merge the code segment with other units.

Integration Testing

Integration testing is defined as the testing of combined parts of an application to determine if they function correctly. Integration testing can be done in two ways: Bottom-up integration testing and Top-down integration testing.

Sr.No.	Integration Testing Method
1	**Bottom-up integration** This testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds.
2	**Top-down integration** In this testing, the highest-level modules are tested first and progressively, lower-level modules are tested thereafter.

In a comprehensive software development environment, bottom-up testing is usually done first, followed by top-down testing. The process concludes with multiple tests of the complete application, preferably in scenarios designed to mimic actual situations.

System Testing

System testing tests the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets the specified Quality Standards. This type of testing is performed by a specialized testing team.

System testing is important because of the following reasons −

- System testing is the first step in the Software Development Life Cycle, where the application is tested as a whole.

- The application is tested thoroughly to verify that it meets the functional and technical specifications.
- The application is tested in an environment that is very close to the production environment where the application will be deployed.
- System testing enables us to test, verify, and validate both the business requirements as well as the application architecture.

Regression Testing

Whenever a change in a software application is made, it is quite possible that other areas within the application have been affected by this change. Regression testing is performed to verify that a fixed bug hasn't resulted in another functionality or business rule violation. The intent of regression testing is to ensure that a change, such as a bug fix should not result in another fault being uncovered in the application.

Regression testing is important because of the following reasons −

- Minimize the gaps in testing when an application with changes made has to be tested.
- Testing the new changes to verify that the changes made did not affect any other area of the application.
- Mitigates risks when regression testing is performed on the application.
- Test coverage is increased without compromising timelines.
- Increase speed to market the product.

Acceptance Testing

This is arguably the most important type of testing, as it is conducted by the Quality Assurance Team who will gauge whether the application meets the intended specifications and satisfies the client's requirement. The QA team will have a set of pre-written scenarios and test cases that will be used to test the application.

More ideas will be shared about the application and more tests can be performed on it to gauge its accuracy and the reasons why the project was initiated. Acceptance tests are not only intended to point out simple spelling mistakes, cosmetic errors, or interface gaps, but also to point out any bugs in the application that will result in system crashes or major errors in the application.

By performing acceptance tests on an application, the testing team will reduce how the application will perform in production. There are also legal and contractual requirements for acceptance of the system.

Alpha Testing

This test is the first stage of testing and will be performed amongst the teams (developer and QA teams). Unit testing, integration testing and system testing when combined together is known as alpha testing. During this phase, the following aspects will be tested in the application −

- Spelling Mistakes
- Broken Links
- Cloudy Directions
- The Application will be tested on machines with the lowest specification to test loading times and any latency problems.

Beta Testing

This test is performed after alpha testing has been successfully performed. In beta testing, a sample of the intended audience tests the application. Beta testing is also known as **pre-release testing**. Beta test versions of software are ideally distributed to a wide audience on the Web, partly to give the program a "real-world" test and partly to provide a preview of the next release. In this phase, the audience will be testing the following −

- Users will install, run the application and send their feedback to the project team.
- Typographical errors, confusing application flow, and even crashes.
- Getting the feedback, the project team can fix the problems before releasing the software to the actual users.
- The more issues you fix that solve real user problems, the higher the quality of your application will be.
- Having a higher-quality application when you release it to the general public will increase customer satisfaction.

Non-Functional Testing

This section is based upon testing an application from its non-functional attributes. Non-functional testing involves testing a software from the requirements which are nonfunctional in nature but important such as performance, security, user interface, etc.

Some of the important and commonly used non-functional testing types are discussed below.

Performance Testing

It is mostly used to identify any bottlenecks or performance issues rather than finding bugs in a software. There are different causes that contribute in lowering the performance of a software −

- Network delay
- Client-side processing
- Database transaction processing
- Load balancing between servers
- Data rendering

Performance testing is considered as one of the important and mandatory testing type in terms of the following aspects −

- Speed (i.e. Response Time, data rendering and accessing)
- Capacity
- Stability
- Scalability

Performance testing can be either qualitative or quantitative and can be divided into different sub-types such as **Load testing** and **Stress testing**.

Load Testing

It is a process of testing the behavior of a software by applying maximum load in terms of software accessing and manipulating large input data. It can be done at both normal and peak load conditions. This type of testing identifies the maximum capacity of software and its behavior at peak time.

Stress Testing

Stress testing includes testing the behavior of a software under abnormal conditions. For example, it may include taking away some resources or applying a load beyond the actual load limit.

The aim of stress testing is to test the software by applying the load to the system and taking over the resources used by the software to identify the breaking point. This testing can be performed by testing different scenarios such as −

- Shutdown or restart of network ports randomly
- Turning the database on or off
- Running different processes that consume resources such as CPU, memory, server, etc.

Usability Testing

Usability testing is a black-box technique and is used to identify any error(s) and improvements in the software by observing the users through their usage and operation.

On the other hand, usability testing ensures a good and user-friendly GUI that can be easily handled. UI testing can be considered as a sub-part of usability testing.

Security Testing

Security testing involves testing a software in order to identify any flaws and gaps from security and vulnerability point of view. Listed below are the main aspects that security testing should ensure −

- Confidentiality
- Integrity
- Authentication

- Availability
- Authorization
- Non-repudiation
- Software is secure against known and unknown vulnerabilities
- Software data is secure
- Software is according to all security regulations
- Input checking and validation
- SQL insertion attacks
- Injection flaws
- Session management issues
- Cross-site scripting attacks
- Buffer overflows vulnerabilities
- Directory traversal attacks

Portability Testing

Portability testing includes testing a software with the aim to ensure its reusability and that it can be moved from another software as well. Following are the strategies that can be used for portability testing −

- Transferring an installed software from one computer to another.
- Building executable (.exe) to run the software on different platforms.

Portability testing can be considered as one of the sub-parts of system testing, as this testing type includes overall testing of a software with respect to its usage over different environments. Computer hardware, operating systems, and browsers are the major focus of portability testing. Some of the pre-conditions for portability testing are as follows −

- Software should be designed and coded, keeping in mind the portability requirements.
- Unit testing has been performed on the associated components.
- Integration testing has been performed.
- Test environment has been established.

DEBUGGING

Debugging is the process of fixing a bug in the software. In other words, it refers to identifying, analyzing and removing errors. This activity begins after the software fails to execute properly and concludes by solving the problem and successfully testing the software. It is considered to be an extremely complex and tedious task because errors need to be resolved at all stages of debugging.

Process of Debugging

Below is the list of stages involved in the process of debugging

1. Identify Error

Identifying errors at an early stage can save a lot of time. If we make a mistake in identifying an error, it leads to a lot of time wastage. Error or bugs occur at a customer site is hard to find. Identifying the right error is import to save time and avoid errors at the user site.

2. Identify the Error Location

After identifying an error, we need to identify the exact location in the code where an error occurs. Identifying an exact location that leads error can help to fix the problem faster.

3. Analyze Error

In this stage, you have to use an appropriate approach to analyze the error. This will help you to understand the problem. This stage is very critical as solving one error may lead to another error.

4. Prove the Analysis

Once the identified error has been analyzed, you have to focus on other errors of the software. This process involves test automation where you need to write the test cases through the test framework.

5. Cover Lateral Damage

In this stage, you have to do unit testing of all the code where you make the changes. If all test cases pass the test, then you can move to the next stage or else you have to resolve the test case that doesn’t pass the test.

6.**Fix and validate:** this is the final stage of the debugging process, where you need to fix all the bugs and test all test script.

Advantages of Debugging

Below is the list of debugging advantages

- **Saves Time:** Performing debugging at the initial stage saves the time of software developers as they can avoid the use of complex codes in software development. It not only saves the time of software developers but also saves their energy.
- **Reports Errors:** It gives error report immediately as soon as they occur. This allows the detection of error at an early stage and makes the software development process a stress free.
- **Easy Interpretations:** It provides easy interpretations by providing more information about data structures
- **Release bug-free software:** By finding errors in software, it allows developers to fix them before releasing them and provides bug-free software to the customers.

SOFTWARE MAINTENANCE

Software maintenance is an integral part of a software life cycle. However, it has not received the same degree of attention that the other phases have. Historically, software development has had a much higher profile than software maintenance in most organizations. This is now changing, as organizations strive to squeeze the most out of their software development investment by keeping software operating as long as possible. The open source paradigm has brought further attention to the issue of maintaining software artifacts developed by others.

Software Maintenance Fundamentals

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

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.

The objective of software maintenance is to modify existing software while preserving its integrity. The international standard also states the importance of having some maintenance activities prior to the final delivery of software (predelivery activities).

2 Nature of Maintenance

Software maintenance sustains the software product throughout its life cycle (from development to operations). 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. The term maintainer is defined as an organization that performs maintenance activities. In this KA, the term will sometimes refer to individuals who perform those activities, contrasting them with the developers.

IEEE 14764 identifies the primary activities of software maintenance as process implementation, problem and modification analysis, modification implementation, maintenance review/acceptance, migration, and retirement. These activities are discussed in section 3.2, Maintenance Activities.

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

3 Need for Maintenance

Maintenance is needed to ensure that the software continues to satisfy user requirements. Maintenance is applicable to software that is developed using any software life cycle model (for example, spiral or linear). 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; and
- retire software.

Five key characteristics comprise the maintainer's activities:

- maintaining control over the software's day-to-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.

2 Key Issues in Software Maintenance

A number of key issues must be dealt with to ensure the effective maintenance of software. Software maintenance provides unique technical and management challenges for software engineers—for example, trying to find a fault in software containing a large number of lines of code that another software engineer developed. 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. The following section presents some of the technical and management issues related to software maintenance. They have been grouped under the following topic headings:

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

2.1 Technical Issues

2.1.1 Limited Understanding

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

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. Regression testing (the selective retesting of software or a component to verify that the modifications have not caused unintended effects) is an important testing concept in maintenance. Additionally, finding time to test is often difficult. Coordinating tests when different members of the maintenance team are working on different problems at the same time remains a challenge. When software performs critical functions, it may be difficult to bring it offline to test. Tests cannot be executed in the most meaningful place–the production system. 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. They use that knowledge to perform impact analysis, which identifies all systems and software products affected by a software change request and develops an estimate of the resources needed to accomplish the change. Additionally, the risk of making the change is determined. The change request, sometimes called a modification request (MR) and often called a problem report (PR), must first be analyzed and translated into software terms. Impact analysis is performed after a change request enters the software configuration management process. 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.

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

Software designed with maintainability in mind greatly facilitates impact analysis. More information can be found in the Software Configuration Management KA.

2.1.4 Maintainability

IEEE 14764 defines maintainability as the capability of the software product to be modified. Modifications may include corrections, improvements, or adaptation of the software to changes in environment as well as changes in requirements and functional specifications.

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. This in turn can, and often does, result in a lack of software documentation and test environments, which is a leading cause of difficulties in program comprehension and subsequent impact analysis. The presence of systematic and mature processes, techniques, and tools helps to enhance the maintainability of software.

2.2 Management Issues

2.2.1 Alignment with Organizational Objectives

Organizational objectives describe how to demonstrate the return on investment of software maintenance activities. Initial software development is usually project-based, with a defined time scale and budget. 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. In both cases, the return on investment is much less clear, so that the view at the senior management level is often that of a major activity consuming significant resources with no clear quantifiable benefit for the organization.

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 (for example, software configuration management is a crucial activity in both). Maintenance also requires several activities that are not found in software development (see section 3.2 on unique activities for details). These activities present challenges to management.

2.2.4 Organizational Aspects of Maintenance

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

In deciding where the software maintenance function will be located, software engineering organizations may, for example, stay with the original developer or go to a permanent maintenance- specific team (or maintainer). Having a permanent maintenance team has many benefits:

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

- allows for periodic audit checks.

Since there are many pros and cons to each option, the decision should be made on a case-bycase basis. What is important is the delegation or assignment of the maintenance responsibility to a single group or person, regardless of the organization's structure.

2.2.5 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. One of the major challenges for outsourcers is to determine the scope of the maintenance services required, the terms of a service- level agreement, and the contractual details. Outsourcers will need to invest in a maintenance infrastructure, and the help desk at the remote site should be staffed with native-language speakers. 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. For planning purposes, cost estimation is an important aspect of planning for software maintenance.

2.3.1 Cost Estimation

Section 2.1.3 describes how impact analysis identifies all systems and software products affected by a software change request and develops an estimate of the resources needed to accomplish that change.

Maintenance cost estimates are affected by many technical and nontechnical factors. 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" [1*, c7s4.1]. A combination of these two can also be used.

2.3.2 Parametric Models

Parametric cost modeling (mathematical models) has been applied to software maintenance. 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. Maintenance estimation historical data should be provided as a result of a measurement program.

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. Included in this category is emergency maintenance, which is an unscheduled modification performed to temporarily keep a software product operational pending corrective maintenance.
- **Adaptive maintenance**: modification of a software product performed after delivery to keep a software product usable in a changed or changing environment. For example, the operating system might be upgraded and some changes to the software may be necessary.
- **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.

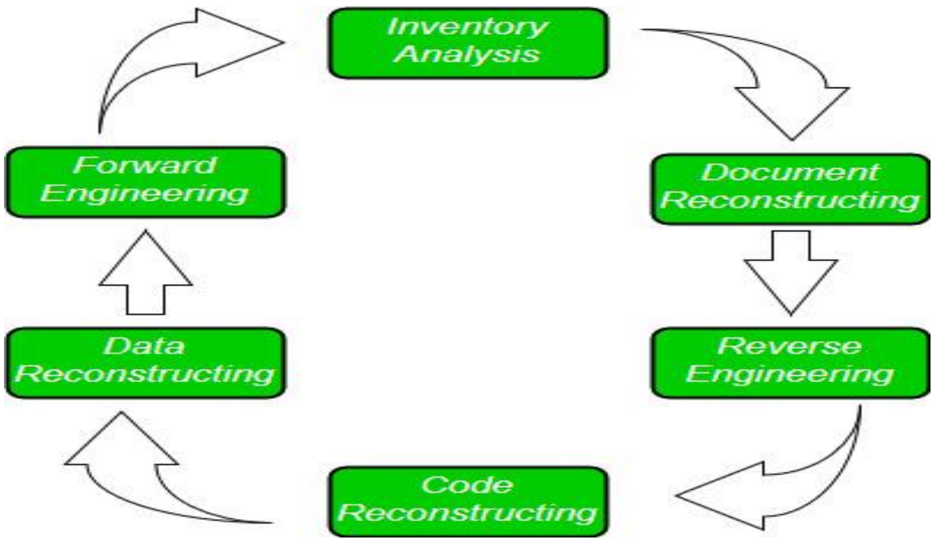
- **Preventive maintenance:** modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults.

Software Re-Engineering

Software Re-Engineering is the examination and alteration of a system to reconstitute it in a new form. The principles of Re-Engineering when applied to the software development process is called software re-engineering. It affects positively at software cost, quality, service to the customer and speed of delivery. In Software Re-engineering, we are improving the software to make it more efficient and effective.

Re-Engineering cost factors:

- The quality of the software to be re-engineered.
- The tool support availability for engineering.
- Extent of the data conversion which is required.
- The availability of expert staff for Re-engineering.



Software Re-Engineering Activities:

1. Inventory Analysis:

Every software organisation should have an inventory of all the applications.

- Inventory can be nothing more than a spreadsheet model containing information that provides a detailed description of every active application.
- By sorting this information according to business criticality, longevity, current maintainability and other local important criteria, candidates for re-engineering appear.
- Resource can then be allocated to candidate application for re-engineering work.

2. Document reconstructing:

Documentation of a system either explains how it operate or how to use it.

- Documentation must be updated.
- It may not be necessary to fully document an application.
- The system is business critical and must be fully re-documented.

3. Reverse Engineering:

Reverse engineering is a process of design recovery. Reverse engineering tools extracts data, architectural and proccedural design information from an existing program.

4. Code Reconstructing:

- To accomplish code reconstructing, the source code is analysed using a reconstructing tool. Violations of structured programming construct are noted and code is then reconstruct.
- The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced.

5. Data Restructuring:

- Data restructuring begins with the reverse engineering activity.
- Current data architecture is dissecred, and necessary data models are defined.
- Data objects and attributes are identified, and existing data structure are reviewed for quality.

6. Forward Engineering:

Forward Engineering also called as renovation or reclamation not only for recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality.