SOFTWARE ENGINEERING

OVERVIEW OF SOFTWARE ENGINEERING

Software Engineering is the process of designing, building, testing and maintaining of software applications. It is a branch of computer science that uses engineering principles and programing languages.

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches. In other words, it is the application of engineering to software.

Sub-disciplines in Software Engineering
Software engineering can be divided into ten sub-disciplines. They are:

- **Software requirements:** The analysis, specification, and validation of requirements for software.
- **Software design:** Software Design consists of the steps a programmer should do before they start coding the program in a specific language. It is usually done with Computer-Aided Software Engineering (CASE) tools and use standards for the format, such as the Unified Modeling Language (UML).
- **Software development:** It is construction of software through the use of programming languages.
- **Software testing Software Testing** is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test.
- **Software maintenance:** This deals with enhancements of Software systems to solve the problems the may have after being used for a long time after they are first completed.
- **Software configuration management:** is the task of tracking and controlling changes in the software. Configuration management practices include revision control and the establishment of baselines.
- **Software engineering management:** The management of software systems borrows heavily from project management.

- **Software development process:** is a structure imposed on the development of a software product. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.
- **Software engineering tools**, (CASE which stands for Computer Aided Software Engineering) CASE tools are a class of software that automates many of the activities involved in various life cycle phases.

- **Software quality:** The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs.

Software engineering is considered to be a subfield of computer science by many academics. Many of the foundations of software engineering come from computer science. The building of a software system is usually considered as a project. Its management borrows many principles from the field of project management. Systems engineers have been dealing with the complexity of large systems for many decades and their knowledge is applied to many software engineering problems.

Software Engineering Goals and Principles

Goals

Stated requirements when they are initially specified for systems are usually incomplete. Apart from accomplishing these stated requirements, a good software system must be able to easily support changes to these requirements over the system's life. Therefore, a major goal of software engineering is to be able to deal with the effects of these changes. The software engineering goals include:

- **Maintainability:** Changes to software without increasing the complexity of the original system design should be possible.

- **Reliability: The** software should be able to prevent failure in design and construction as well as recover from failure in operation. In other words, the software should perform its intended function with the required precision at all times.

- **Efficiency:** The software system should use the resources that are available in an optimal manner.

- **Understand ability*:*** The software should accurately model the view the reader has of the real world. Since code in a large, long-lived software system is usually read more times than it is written, it should be easy to read at the expense of being easy to write, and not the other way around.

FIVE CORE PRINCIPLES OF SOFTWARE ENGINEERING

Five core principles that shape the landscape of software engineering: **Modularity, Abstraction, Encapsulation, Cohesion, and Coupling.**

Modularity

Modularity breaks down the software system into smaller, manageable, and independent units known as modules. These modules can be developed, tested, and maintained separately from the rest of the system.

It's Important

- **Efficiency:** Dividing a complex system into modules allows multiple teams to work on different parts simultaneously.
- **Reusability:** Well-designed modules can often be reused in different parts of the system or even in different projects.
- **Ease of Maintenance:** Since modules are self-contained, updating or debugging one module usually won't affect others.

Abstraction

Abstraction refers to the act of hiding the complex reality while exposing only the necessary parts to the user, offering a more straightforward, simplified interface.

It's Important

- **User-Friendly:** Makes the system easier to understand and use.
- **Reduced Complexity:** Developers can work on one layer of the system without worrying too much about the underlying layers.
- **Enhanced Security:** Sensitive details can be hidden away, protecting the system from malicious activity.

Encapsulation

Encapsulation involves hiding the internal state and requiring all interaction to be performed through an object's methods, allowing for easier maintenance and updates.

It's Important

- **Data Integrity:** Restricts unauthorized access and modification of data.
- **Modifiability:** Changes to the object can be made without affecting the object's external behaviour.
- **Isolation:** Errors in one part of the system do not spill over into other parts.

Cohesion

Cohesion refers to the degree to which the elements within a module belong together, ensuring that each component or module is tasked with a specific functionality.

It's Important

- **Maintainability:** Easier to understand and update the system.
- **Robustness:** Higher cohesion usually correlates with lower error rates.
- **Productivity:** Teams can work more efficiently when tasks are well-defined.

Coupling:
Coupling deals with the level of dependency between different modules. A well-designed system will aim to have low coupling—meaning modules are as independent as possible.

It's Important

- **Flexibility:** Easier to make changes or add features.
- **Adaptability:** Low coupling allows for easier incorporation of new technologies or methods.
- **Risk Isolation:** A bug or an issue in one module is less likely to affect others.

Software engineering, much like any disciplined craft, operates on a bedrock of core principles. Understanding and applying these principles—Modularity, Abstraction, Encapsulation, Cohesion, and Coupling—will set you on a path to creating software that is not just functional but also reliable, maintainable, and scalable.

Evolution of Software Engineering

There are a number of areas where the evolution of software engineering is notable:

(i) Professionanism: The early 1980s witnessed software engineering becoming a full-fledged profession like computer science and other engineering fields.

(ii) Impact of women: In the early days of computer development (1940s, 1950s, and 1960s,), the men were found in the hardware sector because of the mental demand of hardwaring heavy duty equipment which was too strenuous for women. The writing of software was delegated to the women. Some of the women who were into many programming jobs at this time include Grace Hopper and Jamie Fenton. Today, many fewer women work in software engineering than in other professions, this reason for this is yet to be ascertained.

(iii) Processes: Processes have become a great part of software engineering and re praised for their ability to improve software and sharply condemned for their potential to narrow programmers.

(iv) Cost of hardware: The relative cost of software versus hardware has changed substantially over the last 50 years. When mainframes were costly and needed large support staffs, the few organizations purchasing them also had enough to fund big, high-priced custom software engineering projects. Computers can now be said to be much more available and much more powerful, which has a lot of effects on software. The larger market can sustain large projects to create commercial packages, as the practice of companies such as Microsoft. The inexpensive machines permit each programmer to have a terminal capable of fairly rapid compilation. The programs under consideration can use techniques such as garbage collection, which make them easier and faster for the programmer to write. Conversely, many fewer organizations are concerned in employing programmers for large custom software projects, instead using commercial packages as much as possible.

The Pioneering Era

The most key development was that new computers were emerging almost every year or two, making existing ones outdated. Programmers had to rewrite all their programs to run on these new computers. They did not have computers on their desks and had to go to the "computer room" or ―computer laboratory‖. Jobs were run by booking for machine time or by operational staff. Jobs were run by inserting punched cards for input into the computer‘s card reader and waiting for results to come back on the printer.

The field was so new that the idea of management using schedule was absent. Guessing the completion time of project predictions was almost unfeasible Computer hardware was application-based. Scientific and business tasks needed different machines. High level languages like FORTRAN, COBOL, and ALGOL were developed to take care of the need to frequently translate old software to meet the needs of new machines. Systems software was given out for free by the vendors since it must to be installed in the

computer before it is sold. Custom software was sold by a few companies but no sale of packaged software.
Organisation such as like IBM's scientific user group SHARE gave out software free and as a result reuse was order of the day. Academia did not yet teach the principles of computer science. Modular programming and data abstraction were already being used in programming.

1945 to 1965: The origins

The term *software engineering* came into existence in the late 1950s and early 1960s. Programmers have always known about civil, electrical, and computer engineering but found it difficult to marry engineering with software.
In 1968 and 1969, two conferences on software engineering were sponsored by the NATO Science Committee. This gave the field its initial boost. It was widely believed that these conferences marked the official start of the profession of *software engineering*.

1965 to 1985: The software crisis

Software engineering was prompted by the *software crisis* of the 1960s, 1970s, and 1980s. It was the crisis that identified many of the problems of software development. This era was also characterised by: run over budget and schedule, property damage and loss of life caused by poor project management. Initially the software crisis was defined in terms of productivity, but advanced to emphasize quality.
(i) Cost and Budget Overruns: The OS/360 operating system was a classic example. It was a decade-long project from the 1960s and eventually produced one of the most complex software systems at the time.
(ii) Property Damage: Software defects can result in property damage. Poor software security allows hackers to steal identities, costing time, money, and reputations.
(iii) Life and Death: Software defects can kill. Some embedded systems used in radiotherapy machines failed so disastrously that they administered poisonous doses of radiation to patients. The most famous of these failures is the *Therac 25* incident.

1985 to 1989: No silver bullet

For years, solving the software crisis was the primary concern for researchers and companies producing software tools. Apparently, they proclaim every new technology and practice from the 1970s to the 1990s as a *silver bullet* to solve the software crisis. Tools, discipline, formal methods, process, and professionalism were published as silver bullets:

Tools: Particularly underline tools include: Structured programming, object- oriented programming, CASE tools, Ada, Java, documentation, standards, and Unified Modeling Language were touted as silver bullets.

Discipline: Some pundits argued that the software crisis was due to the lack of discipline of programmers.

Formal methods: Some believed that if formal engineering methodologies would be applied to software development, then production of software would become as predictable an industry as other branches of engineering. They advocated proving all programs correct.

Process: Many advocated the use of defined processes and methodologies like the Capability Maturity Model.

Professionalism: This led to work on a code of ethics, licenses, and professionalism.

Fred Brooks (1986), *No Silver Bullet* article, argued that no individual technology or practice would ever make a 10-fold improvement in productivity within 10 years.

Debate about silver bullets continued over the following decade. Supporter for Ada, components, and processes continued arguing for years that their favorite technology would be a silver bullet. Skeptics disagreed. Eventually, almost everyone accepted that no silver bullet would ever be found. Yet, claims about *silver bullets* arise now and again, even today.

" No silver bullet" means different things to different people; some *take" no silver bullet" to mean* that software engineering failed. The pursuit for a single key to success never worked. All known technologies and practices have only made incremental improvements to productivity and quality. Yet, there are no silver bullets for any other profession, either. Others interpret *no silver bullet* as evidence that software engineering has finally matured and recognized that projects succeed due to hard work.

However, it could also be pointed out that there are, in fact, a series of *silver bullets* today, including lightweight methodologies, spreadsheet calculators, customized browsers, in-site search engines, database report generators, integrated design-test coding-editors with memory/differences/undo, and specialty shops that generate niche software, such as information websites, at a fraction of the cost of totally customized website development. Nevertheless, the field of software engineering looks as if it is too difficult and different for a single "silver bullet" to improve most issues, and each issue accounts for only a small portion of all software problems.

1990 to 1999: Importance of the Internet

The birth of internet played a major role in software engineering. With its arrival, information could be gotten from the World Wide Web speedily. Programmers could handle illustrations, maps, photographs, and other images, plus simple animation, at a very fast rate.

It became easier to display and retrieve information as a result of the usage of browser on the HTML language. The widespread of network connections brought in computer viruses and worms on MS Windows computers. These new technologies brought in a lot good innovations such as e-mailing, web-based searching, e-education to mention a few. As a result, many software systems had to be re-designed for international searching. It was also required to translate the information flow in multiple foreign languages. Many

software systems were designed for multi-language usage, based on design concepts from human translators.

2000 to Present: Lightweight Methodologies

This era witnessed increasing demand for software in many smaller organizations. There was also the need for inexpensive software solutions and this led to the growth of simpler, faster methodologies that developed running software, from requirements to deployment. There was a change from rapid-prototyping to entire *lightweight methodologies*. For example, Extreme Programming (XP), tried to simplify many areas of software engineering, including requirements gathering and reliability testing for the growing, vast number of small software systems.

Important figures in the history of software engineering Listed below are some renowned software engineers:

- Charles Bachman (born 1924) is particularly known for his work in the area of databases.
- Fred Brooks (born 1931)) best-known for managing the development of OS/360.
- Peter Chen, known for the development of entity-relationship modeling.
- Edsger Dijkstra (1930-2002) developed the framework for proper programming.
- David Parnas (born 1941) developed the concept of information hiding in modular programming.

SOFTWARE DEVELOPMENT

Software development is the set of activities that results in software products. Software development may include research, new development, modification, reuse, re-engineering, maintenance, or any other activities that result in software products. Particularly the first phase in the software development process may involve many departments, including marketing, engineering, research and development and general management.

The term software development may also refer to computer programming, the process of writing and maintaining the source code.

Stages of Software Development

There are several different approaches to software development. While some take a more structured, engineering-based approach, others may take a more incremental approach, where software evolves as it is developed piece-by-piece. In general, methodologies share some combination of the following stages of software development:

- Market research
- Gathering requirements for the proposed business solution
- Analyzing the problem

- Devising a plan or design for the software-based solution
- Implementation (coding) of the software
- Testing the software
- Deployment
- Maintenance and bug fixing

These stages are collectively referred to as the software development lifecycle (SDLC). These stages may be carried out in different orders, depending on approach to software development. Time devoted on different stages may also vary. The detail of the documentation produced at each stage may not be the same. In ―waterfall‖ based approach, stages may be carried out in turn whereas in a more "extreme" approach, the stages may be repeated over various cycles or iterations. It is important to note that more ―extreme‖ approach usually involves less time spent on planning and documentation, and more time spent on coding and development of automated tests. More ―extreme‖ approaches also encourage continuous testing throughout the development lifecycle. It ensures bug-free product at all times. The ―waterfall‖ based approach attempts to assess the majority of risks and develops a detailed plan for the software before implementation (coding) begins. It avoids significant design changes and re-coding in later stages of the software development lifecycle.

Each methodology has its merits and demerits. The choice of an approach to solving a problem using software depends on the type of problem. If the problem is well understood and a solution can be effectively planned out ahead of time, the more "waterfall" based approach may work the best choice. On the other hand, if the problem is unique (at least to the development team) and the structure of the software solution cannot be easily pictured, then a more "extreme" incremental approach may work best..

LIFE CYCLE MODEL

Software life cycle models describe phases of the software cycle and the order in which those phases are executed. There are a lot of models, and many companies adopt their own, but all have very similar patterns. According to Raymond Lewallen (2005), the general, basic model is shown below:

The General Model

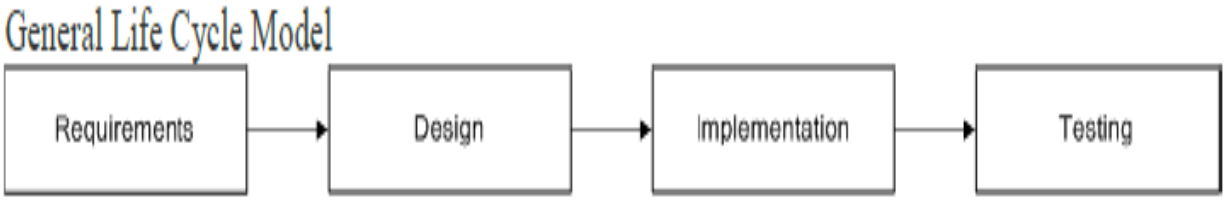

Fig 1 the General Model

Each phase produces deliverables needed by the next phase in the life cycle. Requirements are converted into design. Code is generated during implementation that is driven by the design. Testing verifies the deliverable of the implementation phase against requirements.

Waterfall Model

This is the most common life cycle models, also referred to as a linear-sequential life cycle model. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin. At the end of each phase, there is always a review to ascertain if the project is in the right direction and whether or not to carry on or abandon the project. Unlike the general model, phases do not overlap in a waterfall model.

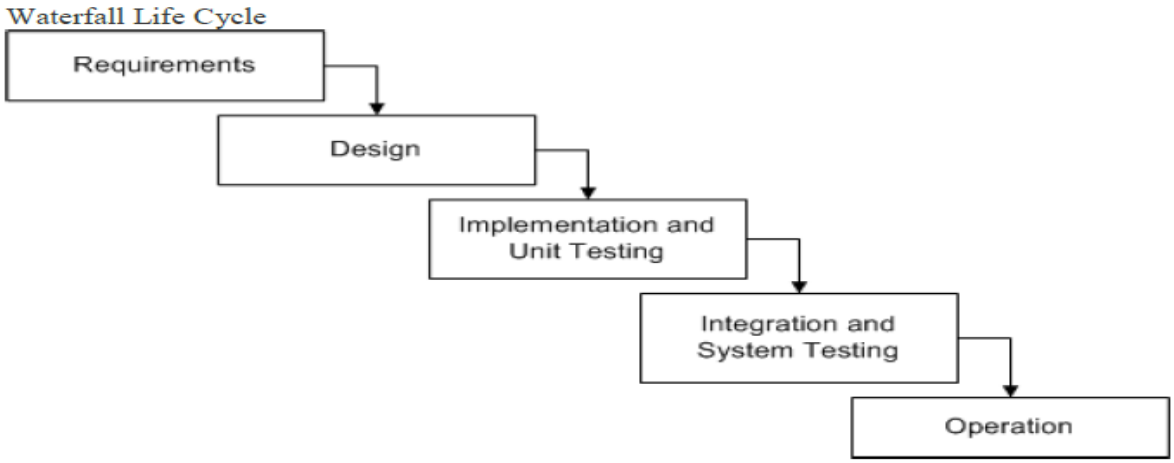

Fig 2 Waterfall Life Cycle

Advantages

- Simple and easy to use.
- Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

Disadvantages

- Adjusting scope during the life cycle can kill a project
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Poor model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Poor model where requirements are at a moderate to high risk of changing.

V-Shaped Model

Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins. Testing is emphasized in this model more so than the waterfall model. The testing procedures are developed early in the life cycle before any coding is done, during each of the phases preceding implementation.

Requirements begin the life cycle model just like the waterfall model. Before development is started, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering.

The high-level design phase focuses on system architecture and design. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together.

The low-level design phase is where the actual software components are designed, and unit tests are created in this phase as well.

The implementation phase is, again, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.

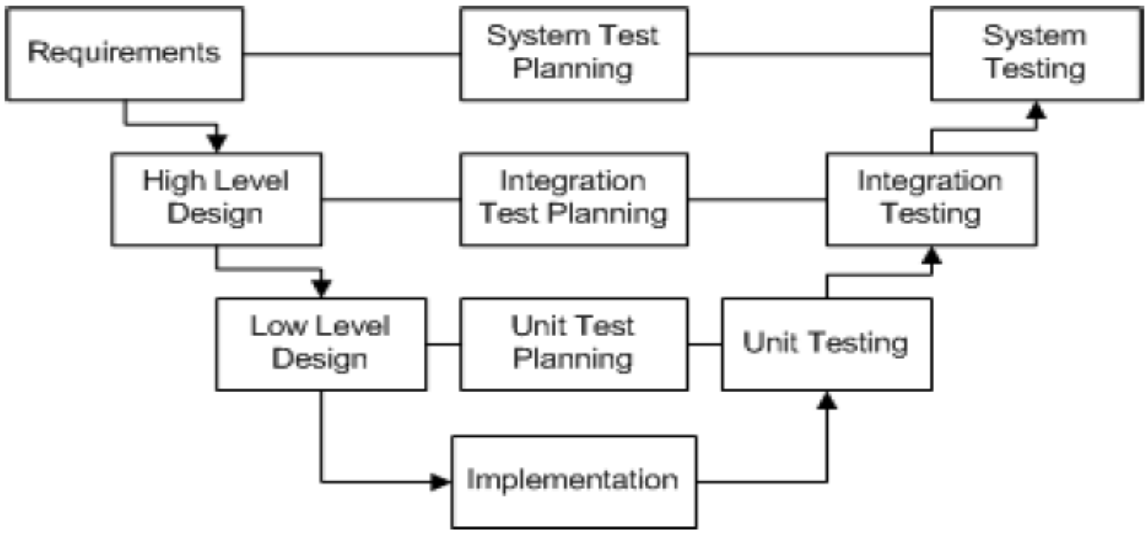

Fig 3 V-Shaped Life Cycle Model

Advantages

- Simple and easy to use.
- Each phase has specific deliverables.
- Higher chance of success over the waterfall model due to the development of test plans early on during the life cycle.
- Works well for small projects where requirements are easily understood.

Disadvantages

- Very rigid, like the waterfall model.
- Little flexibility and adjusting scope is difficult and expensive.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- Model doesn‘t provide a clear path for problems discovered during testing phases.

Incremental Model

The incremental model is an intuitive approach to the waterfall model. It is a kind of a ―multi-waterfall‖ cycle. In that multiple development cycles take at this point. Cycles are broken into smaller, more easily managed iterations. Each of the iterations goes through the requirements, design, implementation and testing phases.

The first iteration produces a working version of software and this makes possible to have working software early on during the software life cycle. Subsequent iterations build on the initial software produced during the first iteration.

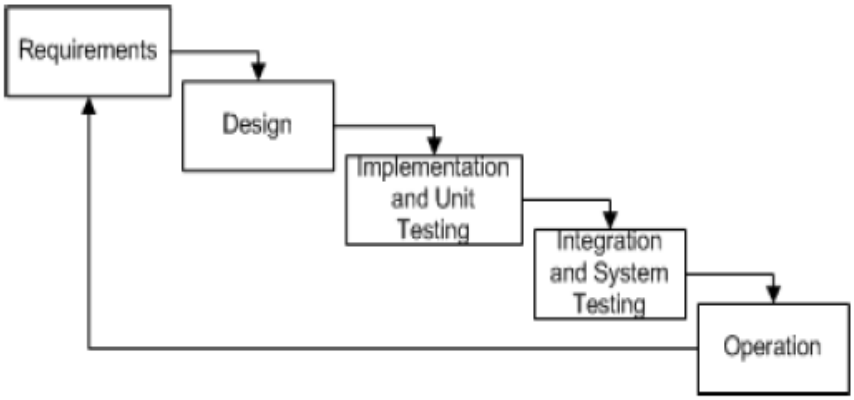

Fig 4 Incremental Life Cycle Model

Advantages

- Generates working software quickly and early during the software life cycle.
- More flexible – inexpensive to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Easier to manage risk because risky pieces are identified and handled during its iteration.
- Each of the iterations is an easily managed landmark

Disadvantages

- Each phase of an iteration is rigid and do not overlap each other.
- Problems as regard to system architecture may arise as a result of inability to gathered requirements up front for the entire software life cycle.

Spiral Model

The spiral model is similar to the incremental model, with more emphases placed on risk analysis. The spiral model has four phases namely Planning, Risk Analysis, Engineering and Evaluation. A software project continually goes through these phases in iterations which are called spirals. In the baseline spiral requirements are gathered and risk is assessed. Each subsequent spiral builds on the baseline spiral.

Requirements are gathered during the planning phase. In the risk analysis phase, a process is carried out to discover risk and alternate solutions. A prototype is produced at the end of the risk analysis phase.
Software is produced in the engineering phase, alongside with testing at the end of the phase. The evaluation phase provides the customer with opportunity to evaluate the output of the project to date before the project continues to the next spiral.
In the spiral model, the angular component denotes progress, and the radius of the spiral denotes cost.

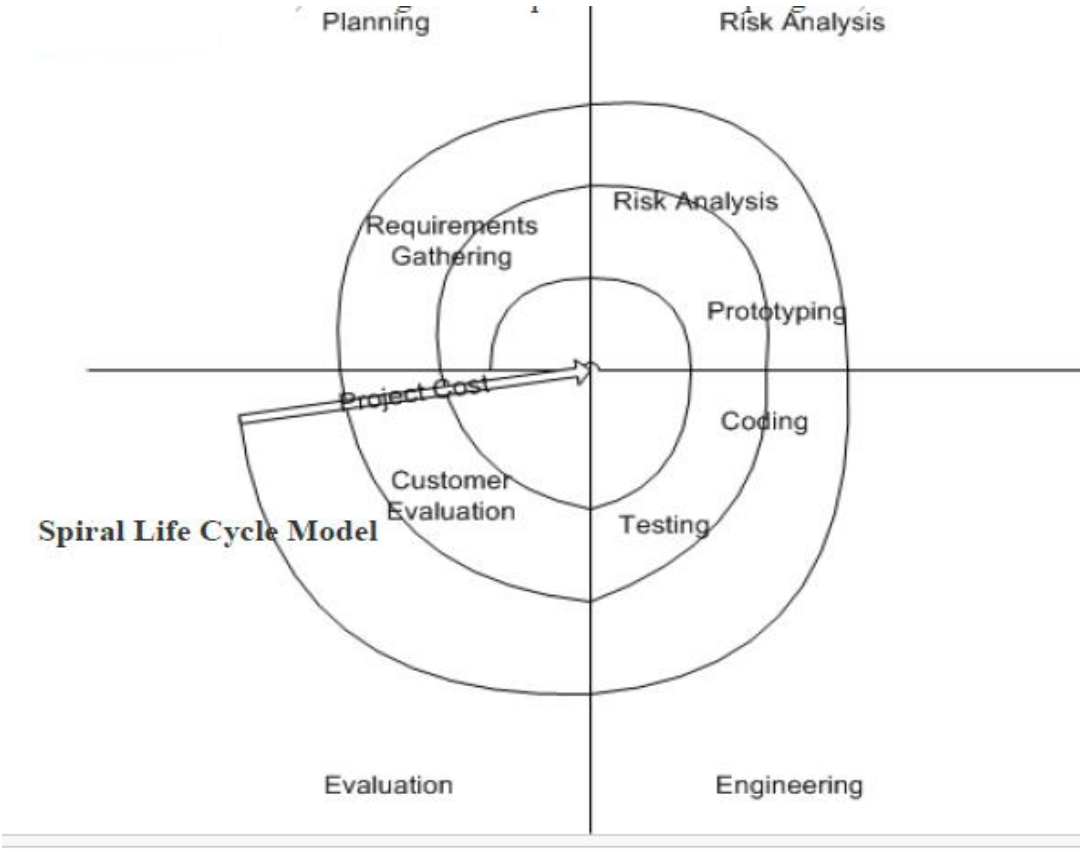

Fig 5 Spiral Life Cycle Model

Merits

- ✓ High amount of risk analysis
- ✓ Good for large and mission-critical projects.
- ✓ Software is produced early in the software life cycle.

Demerits

- ✓ Can be a costly model to use.
- ✓ Risk analysis requires highly specific expertise.
- ✓ Project‘s success is highly dependent on the risk analysis phase.
- ✓ Doesn‘t work well for smaller projects.

Requirements Phase

Business requirements are gathered in this phase. This phase is the main center of attention of the project managers and stake holders. Meetings with managers, stake holders and users are held in order to determine the requirements. The general questions that require answers during a requirements gathering phase are: Who is going to use the system? How will they use the system? What data should be input into the system?
What data should be output by the system? A list of functionality that the system should provide, which describes functions the system should perform, business logic that processes data, what data is stored and used by the system, and how the user interface should work is produced at this point. The requirements development phase may have been preceded by a feasibility study, or a conceptual analysis phase of the project. The requirements phase may be divided into requirements elicitation (gathering the requirements from stakeholders), analysis (checking for consistency and completeness), specification (documenting the requirements) and validation (making sure the specified requirements are correct)
In systems engineering, a **requirement** can be a description of *what* a system must do, referred to as a *Functional Requirement*. This type of requirement specifies something that the delivered system must be able to do. Another type of requirement specifies something about the system itself, and how well it performs its functions. Such requirements are often called *Non-functional requirements*, or 'performance requirements' or 'quality of service requirements.' Examples of such requirements include usability, availability, reliability, supportability, testability, maintainability, and (if defined in a way that's verifiably measurable and unambiguous) ease-of-use.

Types of Requirements

Requirements are categorized as:
Functional requirements which describe the functionality that the system is to execute; for example, formatting some text or modulating a signal.
Non-functional requirements which are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as quality requirements or Constraint

requirements No matter how the problem is solved the constraint requirements must be adhered to.

It is important to note that functional requirements can be directly implemented in software. The non-functional requirements are controlled by other aspects of the system. For example, in a computer system reliability is related to hardware failure rates, performance controlled by CPU and memory. Non-functional requirements can in some cases be broken into functional requirements for software. For example, a system level non-functional safety requirement can be decomposed into one or more functional requirements. In addition, a non-functional requirement may be converted into a process requirement when the requirement is not easily measurable. For example, a system level maintainability requirement may be decomposed into restrictions on software constructs or limits on lines or code.

Requirements analysis

Requirements analysis in systems engineering and software engineering, consist of those activities that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

Requirements analysis is critical to the success of a development project. Requirements must be actionable, measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

The Need for Requirements Analysis

Studies reveal that insufficient attention to Software Requirements Analysis at the beginning of a project is the major reason for critically weak projects that often do not fulfill basic tasks for which they were designed. Software companies are now spending time and resources on effective and streamlined Software Requirements Analysis Processes as a condition to successful projects that support the customer‘s business goals and meet the project‘s requirement specifications.

Requirements Analysis Process: Requirements Elicitation, Analysis And Specification

Requirements Analysis is the process of understanding the client needs and expectations from a proposed system or application. It is a well-defined stage in the Software Development Life Cycle model.

Requirements are a description of how a system should behave, in other words, a description of system properties or attributes. Considering the numerous levels of dealings between users, business processes and devices in worldwide corporations today,

there are immediate and composite requirements from a single application, from different levels within an organization and outside it

The Software Requirements Analysis Process involves the complex task of eliciting and documenting the requirements of all customers, modelling and analyzing these requirements and documenting them as a foundation for system design.

This job (requirements analysis process) is dedicated to a specialized Requirements Analyst. The Requirements Analysis function may also come under the scope of Project Manager, Program Manager or Business Analyst, depending on the organizational hierarchy.

Steps in the Requirements Analysis Process

Fix system boundaries

This is initial step and helps in identifying how the new application fit in into the business processes, how it fits into the larger picture as well as its capacity and limitations.

Identify the customer

This focuses on identifying who the _users‘ or _customers‘ of an application are, that is to say, knowing the group or groups of people who will be directly or indirectly impacted by the new application. This allows the Requirements Analyst to know in advance where he has to look for answers.

Requirements elicitation

Here information is gathered from the multiple stakeholders identified. The Requirements Analyst brings out from each of these groups what their requirements from the application are and what they expect the application to achieve. Taking into account the multiple stakeholders involved, the list of requirements gathered in this manner could go into pages. The level of detail of the requirements list depends on the number and size of user groups, the degree of complexity of business processes and the size of the application.

Problems faced in Requirements Elicitation

- Ambiguous understanding of processes
- Inconsistency within a single process by multiple users
- Insufficient input from stakeholders
- Conflicting stakeholder interests
- Changes in requirements after project has begun

Tools used in Requirements Elicitation

Tools used in Requirements Elicitation include stakeholder interviews and focus group studies. Other methods like flowcharting of business processes and the use of existing documentation like user manuals, organizational charts, process models and systems or process specifications, on-site analysis, interviews with end-users, market research and competitor analysis are also used widely in Requirements Elicitation.
There are of course, modern tools that are better equipped to handle the complex and multilayered process of Requirements Elicitation. Some of the current Requirements Elicitation tools in use are:

- Prototypes
- Use cases
- Data flow diagrams
- Transition process diagrams

- User interfaces

Requirements Analysis
The moment all stakeholder requirements have been gathered, a structured analysis of these can be done after modeling the requirements. Some of the Software Requirements Analysis techniques used are requirements animation, automated reasoning, knowledge-based critiquing, consistency checking, analogical and case-based reasoning.
Requirements Specification
After requirements have been elicited, modeled and analyzed, they should be documented in clear, definite terms. A written requirements document is crucial and as such its circulation should be among all stakeholders including the client, user-groups, the development and testing teams. It has been observed that a well-designed, clearly documented Requirements Specification is vital and serves as a:

a) Base for validating the stated requirements and resolving stakeholder conflicts, if any
b) Contract between the client and development team
c) Basis for systems design for the development team
d) Bench-mark for project managers for planning project development lifecycle and goals
e) Source for formulating test plans for QA and testing teams
f) Resource for requirements management and requirements tracing
g) Basis for evolving requirements over the project life span

Software requirements specification involves scoping the requirements so that it meets the customer‘s vision. It is the result of teamwork between the end-user who is usually not a technical expert, and a Technical/Systems Analyst, who is expected to approach the situation in technical terms.

The software requirements specification is a document that lists out stakeholders‘ needs and communicates these to the technical community that will design and build the system. It is really a challenge to communicate a well-written requirements specification, to both these groups and all the sub-groups within. To overcome this, Requirements Specifications may be documented separately as:

- **User Requirements -** written in clear, precise language with plain text and use cases, for the benefit of the customer and end-user
- **System Requirements -** expressed as a programming or mathematical model, meant to address the Application Development Team and QA and Testing Team.

Requirements Specification serves as a starting point for software, hardware and database design. It describes the function (Functional and Non-Functional specifications) of the system, performance of the system and the operational and user-interface constraints that will govern system development.

Requirements Management

Requirements Management is the all-inclusive process that includes all aspects of software requirements analysis and as well ensures verification, validation and traceability of requirements. Effective requirements management practices assure that all system requirements are stated unmistakably, that omissions and errors are corrected and that evolving specifications can be included later in the project lifecycle.

Design Phase

The software system design is formed from the results of the requirements phase. This is where the details on how the system will work are produced. Deliverables in this phase include hardware and software, communication, software design.

SOFTWARE DESIGN

A software design is a meaningful engineering representation of some software product that is to be built. A design can be traced to the customer's requirements and can be assessed for quality against predefined criteria. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces and components.

The design process is very important. As a labourer, for example one would not attempt to build a house without an approved blueprint so as not to risk the structural integrity and customer satisfaction. In the same way, the approach to building software products is no unlike. The emphasis in design is on quality. It is pertinent to note that, this is the only phase in which the customer‘s requirements can be precisely translated into a finished software product or system. As such, software design serves as the foundation for all software engineering steps that follow regardless of which process model is being employed.

During the design process the software specifications are changed into design models that express the details of the data structures, system architecture, interface, and components. Each design product is re-examined for quality before moving to the next phase of software development. At the end of the design process a design specification document is produced. This document is composed of the design models that describe the data, architecture, interfaces and components.

Design Specification Models

- **Data design** – created by changing the analysis information model (data dictionary and ERD) into data structures needed to implement the software. Part of the data design may occur in combination with the design of software architecture. More detailed data design occurs as each software component is designed.

- **Architectural design** - defines the relationships among the major structural elements of the software, the ―design patterns‖ that can be used to attain the requirements that have been defined for the system, and the constraint that affect the way in which the architectural patterns can be applied. It is derived from the system specification, the analysis model, and the subsystem interactions defined in the analysis model (DFD).
- **Interface design** - explains how the software elements communicate with each other, with other systems, and with human users. Much of the necessary information required is provided by the e data flow and control flow diagrams.
- **Component-level design** – It converts the structural elements defined by the software architecture into procedural descriptions of software components using information acquired from the process specification (PSPEC), control specification (CSPEC), and state transition diagram (STD).

Design Guidelines

In order to assess the quality of a design (representation) the yardstick for a good design should be established. Such a design should:

- exhibit good architectural structure
- be modular
- contain distinct representations of data, architecture, interfaces, and components (modules)
- lead to data structures that are appropriate for the objects to be implemented and be drawn from recognizable design patterns
- lead to components that exhibit independent functional characteristics
- lead to interfaces that reduce the complexity of connections between modules and with the external environment

- be derived using a reputable method that is driven by information obtained during software requirements analysis
- These criteria are not acquired by chance. The software design process promotes good design through the application of fundamental design principles, systematic methodology and through review.

Design Principles

Software design can be seen as both a process and a model.

—The design process is a series of steps that allow the designer to describe all aspects of the software to be built. However, it is not merely a recipe book; for a competent and successful design, the designer must use creative skill, past experience, a sense of what makes —good software, and have a commitment to quality.

The set of principles which has been established to help the software engineer in directing the design process are:

- The design process should not suffer from tunnel vision – Alternative approaches should be considered by a good designer. Designer should judge each approach based on the requirements of the problem, the resources available to do the job and any other constraints.
- The design should be traceable to the analysis model – because a single element of the design model often traces to multiple requirements, it is necessary to have a means of tracking how the requirements have been satisfied by the model
- The design should not reinvent the wheel – Systems are constructed using a suite of design patterns, many of which may have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Design time should be spent in expressing truly fresh ideas and incorporating those patterns that already exist.
- The design should reduce intellectual distance between the software and the problem as it exists in the real world – This means that, the structure of the software design should imitate the structure of the problem domain.
- The design should show uniformity and integration – a design is uniform if it appears that one person developed the whole thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
- The design should be structured to degrade gently, even with bad data, events, or operating conditions are encountered – Well-designed software should never —bomb. It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

- The design should be reviewed to minimize conceptual (semantic) errors – there is sometimes the tendency to focus on minute details when the design is reviewed, missing the forest for the trees. The designer team should ensure that major conceptual elements of the design have been addressed before worrying about the syntax if the design model.
- Design is not coding, coding is not design – Even when detailed designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made of the coding level address the small implementation details that enable the procedural design to be coded.
- The design should be structured to accommodate change
- The design should be assessed for quality as it is being created

With proper application of design principles, the design displays both external and internal quality factors. External quality factors are those factors that can readily be observed by the user, (e.g. speed, reliability, correctness, usability). Internal quality factors have to do with technical quality more so the quality of the design itself. To achieve internal quality factors the designer must understand basic design concepts.

SOFTWARE TESTING

Software testing is an empirical examination carried out to provide stakeholders with information about the quality of the product or service under test. Software Testing in addition provides an objective, independent view of the software to allow the business to value and comprehend the risks associated with implementation of the software.
Software Testing can also be viewed as the process of validating and verifying that a software program/application/product -
(1) meets the business and technical requirements that guided its design and development;
(2) works as expected; and
(3) can be implemented with the same characteristics. It is important to note that depending on the testing method used, software testing, can be applied at any time in the development process, though most of the test effort occurs after the requirements have been defined and the coding process has been completed.

Testing can never totally detect all the defects within software. Instead, it provides a *comparison* that put side by side the state and behavior of the product against the instrument someone applies to recognize a problem. These instruments may include specifications, contracts, comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

Every software product has a target audience. For instance, the audience for video game software is completely different from banking software. Software testing therefore, is the process of attempting to make this assessment whether the software product will be satisfactory to its end users, its target audience, its purchasers, and other stakeholders.

Brief History of software testing

In 1979, Glenford J. Myers introduced the separation of debugging from testing, illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. 1988, Dave Gelperin and William C. Hetzel classified the phases and goals in software testing in the following stages:

- Until 1956 - Debugging oriented.
- 1957–1978 - Demonstration oriented.
- 1983–1987 - Evaluation oriented.
- 1988–2000 - Prevention oriented.

TESTING METHODS

Traditionally, software testing methods are divided into **black box** testing ,**white box** testing and Grey **Box** Testing. A test engineer used these approaches to describe his opinion when designing test cases.

Black box testing

Black box testing considers the software as a "black box" in the sense that there is no knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing and specification-based testing.

Specification-based testing: Specification-based testing intends to test the functionality of software based on the applicable requirements. Consequently, the tester inputs data into, and only sees the output from, the test object. This level of testing usually needs thorough test cases to be supplied to the tester, who can then verify that for a given input, the output value ,either "is" or "is not" the same as the expected value specified in the test case.

Specification-based testing though necessary, but it is insufficient to guard against certain risks.

Merits and Demerits: The black box testing has the advantage of "an unaffiliated opinion in the sense that there is no "bonds" with the code and the perception of the tester is very simple. He believes a code must have bugs and he goes for it. *But,* on the other hand, black box testing has the disadvantage of blind exploring because the tester doesn't know how the software being tested was actually constructed. As a result, there are situations when

(1) a tester writes many test cases to check something that could have been tested by only one test case, and/or
(2) some parts of the back-end are not tested at all.

White box testing

In a **White box testing** the tester has the privilege to the internal data structures and algorithms including the code that implement these.

Types of white box testing
White box testing is of different types namely:

- **API testing (application programming interface)** - Testing of the application using Public and Private APIs

- **Code coverage** - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)

Grey Box Testing
Grey box testing requires gaining access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output cannot be regarded as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test. This difference is important especially when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, changing a data repository can be seen as grey box, because the use would not ordinarily be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to ascertain boundary values or error messages.

Integration Testing

Integration testing is any type of software testing that seeks to reveal clash of individual software modules to each other. Such integration flaws can result, when the new modules are developed in separate *branches*, and then integrated into the main project.
Regression Testing

Regression testing is any type of software testing that attempts to reveal software regressions. Regression of the nature can occurs at any time software functionality, that was previously working correctly, stops working as anticipated. Usually, regressions occur as an unplanned result of program changes, when the newly developed part of the

software collides with the previously existing. Methods of regression testing include re-running previously run tests and finding out whether previously repaired faults have re-appeared. The extent of testing depends on the phase in the release process and the risk of the added features.

Acceptance testing

One of two things below can be regarded as Acceptance testing:
1. A smoke test which is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before integration or regression.
2. Acceptance testing performed by the customer, usually in their lab environment on their own HW, is known as user acceptance testing (UAT).

Non Functional Software Testing
The following methods are used to test non-functional aspects of software:
 Performance testing confirms to see if the software can deal with large quantities of data or users. This is generally referred to as software scalability. This activity of Non Functional Software Testing is often referred to as Endurance Testing.
 Stability testing checks to see if the software can continuously function well in or above an acceptable period. This activity of Non Functional Software Testing is oftentimes referred to as load (or endurance) testing.
 Usability testing is used to check if the user interface is easy to use and understand.
 Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.
 Internationalization and localization is needed to test these aspects of software, for which a pseudo localization method can be used.

Compare to functional testing, which establishes the correct operation of the software in that it matches the expected behavior defined in the design requirements, non-functional testing confirms that the software functions properly even when it receives invalid or unexpected inputs. Non-functional testing, especially for software, is meant to establish whether the device under test can tolerate invalid or unexpected inputs, thereby establishing the robustness of input validation routines as well as error-handling routines. An example of non-functional testing is software fault injection, in the form of fuzzing.
DESTRUCTIVE TESTING

Destructive testing attempts to cause the software or a sub-system to fail, in order to test its robustness.
TESTING PROCESS
Testing process can take two forms: Usually the testing can be performed by an independent group of testers after the functionality is developed before it is sent to the

customer. Another practice is to start software testing at the same time the project starts and it continues until the project finishes. The first practice always results in the testing phase being used as project buffer to compensate for project delays, thereby compromising the time devoted to testing.

SOFTWARE QUALITY ASSURANCE (SQA)

Introduction

Concepts and Definitions

Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes, and procedures. SQA includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition life cycle. Compliance with agreed-upon standards and procedures is evaluated through process monitoring, product evaluation, and audits. Software development and control processes should include quality assurance approval points, where an SQA evaluation of the product may be done in relation to the applicable standards.

Standards and Procedures

Establishing standards and procedures for software development is critical, since these provide the structure from which the software evolves. Standards are the established yardsticks to which the software products are compared. Procedures are the established criteria to which the development and control processes are compared.

Standards and procedures establish the prescribed methods for developing software; the SQA role is to ensure their existence and adequacy. Proper documentation of standards and procedures is necessary since the SQA activities of process monitoring, product evaluation and auditing rely upon clear definitions to measure project compliance.

TYPES OF STANDARDS INCLUDE:

- Documentation Standards specify form and content for planning, control, and product documentation and provide consistency throughout a project.

- Design Standards specify the form and content of the design product. They provide rules and methods for translating the software requirements into the software design and for representing it in the design documentation.
- Code Standards specify the language in which the code is to be written and define any restrictions on use of language features. They define legal language structures, style conventions, rules for data structures and interfaces, and internal code documentation.

Procedures are explicit steps to be followed in carrying out a process. All processes should have documented procedures. Examples of processes for which procedures are needed are configuration management, non-conformance reporting and corrective action, testing, and formal inspections.
If developed according to the NASA DID, the Management Plan describes the software development control processes, such as configuration management, for which there have to be procedures, and contains a list of the product standards.
Standards are to be documented according to the Standards and Guidelines DID in the Product Specification. The planning activities required to assure that both products and processes comply with designated standards and procedures are described in the QA portion of the Management Plan.

Software Quality Assurance Activities

Product evaluation and process monitoring are the SQA activities that assure the software development and control processes described in the project's Management Plan are correctly carried out and that the project's procedures and
standards are followed. Products are monitored for conformance to standards and processes are monitored for conformance to procedures. Audits are a key technique used to perform product evaluation and process monitoring. Review of the Management Plan should ensure that appropriate SQA approval points are built into these processes.

Product evaluation is an SQA activity that assures standards are being followed. Ideally, the first products monitored by SQA should be the project's standards and procedures. SQA assures that clear and achievable standards exist and then evaluates compliance of the software product to the established standards. Product evaluation assures that the software product reflects the requirements of the applicable standard(s) as identified in the Management Plan.

Process monitoring is an SQA activity that ensures that appropriate steps to carry out the process are being followed. SQA monitors processes by comparing the actual steps carried out with those in the documented procedures. The Assurance section of the Management Plan specifies the methods to be used by the SQA process monitoring activity.
A fundamental SQA technique is the audit, which looks at a process and/or a product in depth, comparing them to established procedures and standards. Audits are used to
review management, technical, and assurance processes to provide an indication of the quality and status of the software product.
The purpose of an SQA audit is to assure that proper control procedures are being followed, that required documentation is maintained, and that the developer's status reports accurately reflect the status of the activity. The SQA product is an audit report to management consisting of findings and recommendations to bring the development into conformance with standards and/or procedures.

SQA Relationships to Other Assurance Activities

Some of the more important relationships of SQA to other management and assurance activities are described below.

Configuration Management Monitoring

SQA assures that software Configuration Management (CM) activities are performed in accordance with the CM plans, standards, and procedures. SQA reviews the CM plans for compliance with software CM policies and requirements and
provides follow-up for nonconformances. SQA audits the CM functions for adherence to standards and procedures and prepares reports of its findings.

The CM activities monitored and audited by SQA include baseline control, configuration identification, configuration control, configuration status accounting, and configuration authentication. SQA also monitors and audits the software library.

SQA assures that:

- Baselines are established and consistently maintained for use in subsequent baseline development and control.

- Software configuration identification is consistent and accurate with respect to the numbering or naming of computer programs, software modules, software units, and associated software documents.
- Configuration control is maintained such that the software configuration used in critical phases of testing, acceptance, and delivery is compatible with the associated documentation.
- Configuration status accounting is performed accurately including the recording and reporting of data reflecting the software's configuration identification, proposed changes to the configuration identification, and the implementation status of approved changes.
- Software configuration authentication is established by a series of configuration reviews and audits that exhibit the performance required by the software requirements specification and the configuration of the software is accurately reflected in the software design documents.
- Software development libraries provide for proper handling of software code, documentation, media, and related data in their various forms and versions from the time of their initial approval or acceptance until they have been incorporated into the final media.
- Approved changes to baselined software are made properly and consistently in all products, and no unauthorized changes are made.