SYSTEM AUDIT:

"The process of collecting and evaluating evidence to determine whether a computer system safeguards assets, maintain data integrity, allows organizational goals to be achieved and determine the efficient use of resources". An information technology audit, or information systems audit, is an examination of the management controls within an Information technology infrastructure.

The evaluation of obtained evidence determines if the information systems are safeguarding assets, maintaining data integrity, and operating effectively to achieve the organization's goals or objectives. These reviews may be performed in conjunction with a financial statement audit, internal audit, or other form of attestation engagement.

System audit is an audit to verify that systems are appropriate, are efficient, and are adequately controlled to ensure valid, reliable, timely, and secure input, processing, and output at all levels of a system's activity.

SOFTWARE ENGINEERING QUALITIES:

Software engineering is the study and application of engineering to the design, development, and maintenance of software. Software Engineering Qualities are mainly classified based on the following:

1) Technical Skills
2) Personal Traits

TECHNICAL SKILLS:

1) **Basic Computer Science Skills:** Hopefully, any software engineer will have these skills and more. Research skills, reading comprehension, the ability to know how to use library functions, and understanding computing problems, design patterns, and frameworks are other skills that are valuable to have.

2) **Passion for Code:** Programming isn't for the uninterested. You must have a passion for code, developing it from a purely scientific skill into a craft or an art. Building code is much like developing a painting, a sculpture, or a symphony. With the popularity of Open Source, you don't have to be alone in code creation — the ability to work with software engineers and developers from around the world is possibly through the Internet.

3) **Fearless Refactoring:** Refactoring is the ability to improve code without changing what it does. The ability to realize that no one should be a slave to original code is key here — that old code can become unstable and incompatible over time. Refactoring enables the developer to own the code, instead of the code owning you.

4) **Develops Quality:** In a former era, engineers thought testing was beneath them. Today, experienced engineers know and understand the value of tests, because their goal is to create a working system. Exposing bugs and eliminating them is the best way to develop stellar code. But a good engineer also knows not to waste time writing trivial or redundant tests, instead focusing on testing the essential parts of each component.

5) **Willing to Leverage Existing Code:** Why invent the wheel when it's already working? Life is too short to continuously invent new codes and libraries. Reuse of internal infrastructure, use of third-party libraries, and leveraging web-scale services such as the ones offered by Amazon, are marks of software genius.

6) **Focus on Usable and Maintainable Code:** Software always works better then it is well designed and user-centric. Good engineers work hard to make the system simple and usable. They think about customers all the time and do not try to invent convoluted stuff that can only be understood and appreciated by geeks. A disciplined engineer thinks about the maintainability and evolution of the code from its first line, as well. Expressive names for methods and variables can make the code self-explanatory.

7) **Can Code in Multiple Languages:** Writing FORTRAN in any language is just the tip of the iceberg. Just like a person who can speak several languages, an engineer who isn't tied to one code language can think outside the box and is a more desirable hire. A willingness to learn new languages, new libraries and new ways of building systems goes a long way to creating a great software engineer.

PERSONAL TRAITS:

8) **Vision:** What is the use in developing code, when it won't be applicable a year or two down the road? Visionaries create code and libraries that are open to refactoring, and easy to use in all code languages. Being able to see the impacts of present-day decisions is paramount to building great software.

9) **Attention to Detail:** If you get angry about misspelled database columns, "uncommented" code, projects that aren't checked into source control, software that's not unit tested, unimplemented features, and so on, then you probably try to avoid those issues yourself. Bad installation packages, sloppy deployments, or a misspelled column name can bring down entire systems. Be obsessive about details, and you'll be on your way to becoming a software star.

10) **Business Acumen:** If you don't understand why your software development is so important to your clients' livelihoods, consider this NASA story. "This software never crashes. It never needs to be re-booted. This software is bug-free. It is perfect, as perfect as human beings have achieved.

Consider these stats: the last three versions of the program — each 420,000 lines long-had just one error each. The last 11 versions of this software had a total of 17 errors. Commercial programs of equivalent complexity would have 5,000 errors." The ability to understand, why all the coding is done, as it the fruit for any customer or client.

11) **Curiosity:** The best software engineers are curious about why something is done one way or another, yet with the added ability of being objective about the solutions. Many engineers we know got in trouble as kids for taking things apart to see how they worked. Putting together software is just a creative, and many software engineers also have artistic hobbies. This creativity and curiosity is required to think outside the box when designing programs. The thrill you get from making something work is what keeps you going.

12) **Experience:** If you've been tinkering with software programs since you were a kid, your abilities as an adult will be quadrupled. Beyond hands-on experience, you might also be addicted to math and science, and the ability to stay organized. At the same time, great software engineers also realize that they don't know it all…the ability to continue to learn is essential in a field where change is a constant.

13) **Discipline:** Although you may have passion for your job, this love for your work and for the next project doesn't mean that you can be sloppy. Attention to detail is important, but so is an ability to stay organized. So much bad code belongs to developers who don't do what they know should be done.

14) **Patience:** Bugs are natural. Design glitches are normal. Sloppy coding by other engineers occurs often. Patience is a key quality for software engineers who want to work in this field.

15) **Teamwork:** Few projects are small enough or require so few skills that one person can do them well. Learning how to work as a team in college is one way to get over that "hermit" image…and working as a team online or in the office can only produce stellar projects. Successful engineers also become good communicators. They know how to write clear and concise reports and instructions, and know how to convey ideas to clients and customers.

SOFTWARE DESIGN:

Software design is the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints. Software design may refer to either "all the activities involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems".

Software design usually involves problem solving and planning a software solution. This includes both low-level component and algorithm design and high-level, architecture design.

Software design is the process of implementing software solutions to one or more set of problems. One of the important parts of software design is the software requirements analysis. It is a part of the software development process that lists specifications used in software engineering.

If the software is "semi-automated" or user centered, software design may involve user experience design yielding a story board to help determine those specifications. If the software is completely automated (meaning no user or user interface), a software design may be as simple as a flow chart or text describing a planned sequence of events.

There are also semi-standard methods like Unified Modeling Language and Fundamental modeling concepts. In either case, some documentation of the plan is usually the product of the design. Furthermore, a software design may be platform-independent or platform-specific, depending on the availability of the technology used for the design.

Software design can be considered as creating a solution to a problem in hand with available capabilities. The main difference between Software analysis and design is that the output of a software analysis consists of smaller problems to solve. Also, the analysis should not be very different even if it is designed by different team members or groups. The design focuses on the capabilities, and there can be multiple designs for the same problem depending on the environment that solution will be hosted.

Sometimes the design depends on the environment that it was developed for, whether it is created from reliable frameworks or implemented with suitable design patterns. When designing software, two important factors to consider are its security and usability.

Software Design Principles:

- The design process should not suffer from “tunnel vision.” A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job.
- 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 for tracking how requirements have been satisfied by the design model.
- The design should not reinvent the wheel. Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.

- The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world. That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
- The design should exhibit uniformity and integration. A design is uniform if it appears that one person developed the entire 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 accommodate change. The design concepts discussed in the next section enable a design to achieve this principle.

SOFTWARE PRODUCTION:

A Software Product is defined as a packaged configuration of software components, or a software-based service with auxiliary materials, which is released for and traded in a specific market.

Examples: *ERP software, Bookkeeping service, Operating systems, Desk-top publishing, Computer-aided design, Software development environments, Customer-relationship management.*

Software Production is the process of producing software in ways similar to the manufacturing of tangible goods. In this way of conducting business, each copy of the software is priced and sold as though it was a tangible product. The sales process usually is conducted by per copy or per desktop software licensing.

When this method is used, the software is developed by software engineering firms specializing in such practices and distributed through retail stores and sold on a per unit basis at a margin price to the buyer greater than zero, even though software has a zero marginal cost per copy to the producer. Software manufacturing like all other tangible goods can have errors and Total Quality Management can be implied in the process.

Both proprietary software and free software can be produced in this model, and sold and distributed as commercial software.

Proprietary software or closed source software is computer software licensed under exclusive legal right of the copyright holder with the intent that the licensee is given the right to use the software only under certain conditions, and restricted from other uses, such as modification, sharing, studying, redistribution, or reverse engineering. Usually the source code of proprietary software is not made available.

Free software is a computer software that is distributed along with its source code, and is released under a software license that guarantee users the freedom to run the software for any purpose as well as to study, adapt/modify, and distribute the original software and the adapted/changed versions.

Free software is often developed collaboratively by volunteer computer programmers. Free software differs from proprietary software, which to varying degrees does not give the user freedoms to study, modify and share the software, and threatens users with legal penalties if they do not conform to the terms of restrictive software licenses.

Proprietary software is usually sold as a binary executable program without access to the source code, which prevents users from modifying and patching it, and results in the user becoming dependent on software companies (vendor lock-in) to provide updates and support.

Free software is also distinct from freeware, which does not require payment for use, but includes software where the authors or copyright holders of freeware have retained all of the rights to the software, so that it is not necessarily permissible to reverse engineer, modify, or redistribute freeware.

SOFTWARE SERVICE:

It is defined as a service provided by the software application which makes us to avail the facilities through an interface. It is also a service provided by a software application running online and making its facilities available to users over the Internet via an interface.

A software service is a service program that contains executable code for one or more services.

A service program can be configured to execute in the context of a user account from either the built-in (local), primary, or trusted domain. It can also be configured to run in a special service user account.

A service runs as a background process that can affect system performance, responsiveness, energy efficiency, and security.

SOFTWARE SPECIFICATION:

A specification is a precise, unambiguous and complete statement of the requirements of a system (or program or process), written in such a way that it can be used to predict how the system will behave.

A software specification describes WHAT a program does, a design describes HOW the program does it, and documentation describes WHY it does it.

A specification is a document which forms a contract between the customer and the designer.

Specification is the second phase in the `staged' model of software development, which consists of: Requirements; Specification; Design; Implementation; Testing; Maintenance.

A specification is a document by which we can judge the correctness of an implementation.

SOFTWARE METRICS:

Software metric is a measurement derived from a software product, process, or resource. Its purpose is to provide a quantitative assessment of the extent to which the product, process, or resource possesses certain attributes.

Software metrics are necessary in order to effectively manage software development. This can be done by accurate schedule, better quality products, good productivity and good cost estimates. The goals of software metrics are to identify and measure essential factors which effects software development.

Software metrics are used to obtain objective reproducible measurements that can be useful for quality assurance, performance, debugging, management, and estimating costs.

Software metric is a measure of some property of a piece of software or its specifications. Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development.

With the help of software metrics it is possible to find defects in code (post release and prior to release), predicting defective code, predicting project success, and predicting project risk.

The main objective of software metric is to describe the current state-of-the-art in the measurement of software products and process. There is still some debate around which metrics matter and what they mean.

The utility of metrics is limited to quantifying one of the following goals: Schedule of a software project, Size/complexity of development involved, cost of project, and quality of software.

Metrics that are gathered from requirements phase include size metrics constituting functions, lines of code and complexity. Quality of requirements for example, can be measured as volatility - The degree to which requirements changes over period of time.Traceability can be

from requirements to requirements and requirements to design or test documents. Consistency and Complexity can also be gathered from requirements phase.

Types of Metrics:

1) ***Requirement Metrics***

Requirements engineering deals with elicitation, analysis, communication and validation of the requirements, once errors are identified it is easy to fix them compared to later identification of errors. It is obvious that the cost of fixing these errors in initial stages is lower than fixing them in later stages of software development.

Many of these errors are caused due to changes in the requirements. In order to eliminate errors there should be some measurement. Metrics that can be gathered from requirements are the size of the requirements, requirements traceability, requirements completeness and requirements volatility.

2) ***Product Metrics***

Product metrics measure the software products at any stage of software development. They can be applied from requirements phase through installation, these metrics measures size of the program. They measure number of pages of documents and complexity of software design. Primitive metrics or direct measurement of an attribute involves no other entity. It is independent of other entities. It can be LOC, duration of tests calculated in number of hours, months and number of defects discovered Size of the software can be measured by its length, functionality and complexity.

a) Object Oriented Metrics

Object Oriented Metrics are helpful for the allocation of resources in software development. These metrics are particularly used to identify fault-prone classes and predicts maintenance efforts, error proneness, and error rate.

b) Communication Metrics

Communication Metrics Communication within a team is defined as intra-team communication while communication among members of different teams is known as inter-team communication.

Communication artifacts for example electronic mail, weekly meetings, informal interactions, liaisons, personal e-mail, and informal interactions are available in entire project capturing information such as code, politics and process.

This communication metrics give a good picture of software development process. These metrics measure the incidents of a specified type within a process. These are available in initial phases of software development and are easier to collect.

3) ***Process Metrics***

Process metrics measure the process of software development. It includes the type of methodology used, experience level of human resources and overall development time. Overall development time of the software product is an objective metrics. If given as a task this should have same results from all the observers. Computed metrics or indirect measurement is to make direct measurement interaction visible.

SOFTWARE QUALITY ASSURANCE:

In the context of software engineering, software quality measures how well software is designed (quality of design), and how well the software conforms to that design (quality of conformance), although there are several different definitions. It is often described as the 'fitness for purpose' of a piece of software.

In the context of software engineering, software quality refers to two related but distinct notions that exist wherever quality is defined in a business context:

Software functional quality reflects how well it complies with or conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the fitness for purpose of a piece of software or how it compares to competitors in the marketplace as a worthwhile product.

Software structural quality refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly.

Structural quality is evaluated through the analysis of the software inner structure, its source code, at the unit level, the technology level and the system level.

Software quality measurement quantifies to what extent a software or system rates along each of these five dimensions. An aggregated measure of software quality can be computed through a qualitative or a quantitative scoring scheme or a mix of both and then a weighting system reflecting the priorities. Software quality is decomposed into process quality, product quality, and quality in use.

Process quality: Software processes implement best practices of software engineering in an organizational context. Process quality expresses the degree to which defined processes were followed and completed.

Product quality: Software products are the output of software processes. Product quality is determined by the degree to which the developed software meets the defined requirements.

Quality in use: A product that perfectly matches defined requirements does not guarantee to be useful in the hands of a user when the implemented requirements do not reflect the intended use. Quality in use addresses the degree to which a product is fit for purpose when exposed to a particular context of use.

Measurable elements of software quality, i.e. quality characteristics, have to be defined in order to assess the quality of a software product and to set quality objectives.

A series of attempts to define attributes of software products by which quality can be systematically described has been combined in the ISO/IEC standards 9126:2001 [ISO01] and 25000:2005 [ISO05] respectively.

The standard provides a quality model with six quality characteristics, namely *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability*.

Bugs, i.e. defects, indicate the deviation of the actual quantity of a quality characteristic from the expected quantity. Defects are often associated with deviations in the behavior of a software system, affecting its functionality.

The quality model, however, makes clear that defects concern all quality characteristics of a software system. Hence, a deviation from a defined runtime performance is therefore as much a defect as a deviation from the expected usability or a flawed computation.