**The Software Development Lyfecycle**

1. **What is Software Engineering?**

Welcome to “What Is Software Engineering.” After watching this video, you will be able to define software engineering, list the responsibilities of a software engineer, and compare and contrast software developers and software engineers. Software engineering is the application of scientific principles to the design and creation of software. The field uses a systematic approach to collect and analyze business requirements in order to design, build, and test software applications to satisfy those business requirements. When computing began in the late 1950s, software engineering was a relatively undefined discipline, but over time it transformed into a modernized engineering field. The software engineering field became a discipline in the 1960s and evolved as new technologies were developed and the approach to software development became more scientific. Trends in software engineering transformed from ad hoc programming towards more formal and standardized methods.

Initially, the creation of software lacked a formal development process. As the world widely adopted computers, software became increasingly integral to more aspects of life. The inefficiencies in the software development process made it difficult to meet the rapidly increasing demand for computing resources and complex software. This led to what is known as the “Software Crisis” which began in the mid-1960s and lasted until the mid-1980s. During this period, software development often ran over budget, behind schedule, and consisted of unmanageable, buggy code. By the time older software solutions came to fruition, newer, better, and faster technologies had already been developed, causing software engineers to have to refactor their code, or completely redesign their system. Often software development solutions that worked for small software systems did not scale to large, complex projects.

Now, Some of these issues still exist today, although to a much lesser extent due to the consistent application of engineering principles to the software development process. Computing resources have become more widely available and standardized methodologies for software development allow for large, complex solutions that scale. The solution to the “Software Crisis” involved transforming unorganized coding efforts into an established engineering discipline. The mid 1980s also saw a rise in the growth of computer-aided software engineering or CASE which also helped to relieve the software crisis. CASE tools can be divided into six categories: business analysis and modeling, development tools such as debugging environments, verification and validation tools, configuration management, metrics and measurement, and project management. The term “software engineer” is often used interchangeably with software developer but there are subtle differences. Software engineers are also developers, but the term “software developer” is usually deemed narrower in scope than that of a software engineer.

A software engineer’s knowledge is usually broader. Software engineers take a systematic, big picture approach in their thinking to software development whereas developers may have more creative approaches. Both software engineers and software developers have specialized knowledge, but software engineers use that knowledge to build entire systems whereas software developers use their knowledge to write code to implement specific functionality within a system. Software engineers are often employed on larger scale projects and they are focused on the broad structure rather than solving an immediate problem. Software engineers are tasked with designing, building, and maintaining software systems. Their Responsibilities include writing and testing code, and consulting with stakeholders such as clients, third party software vendors, security specialists, and other team members. The adoption of a measured, scientific approach to software development has influenced the way software is created and designed.

Today, the development process is typically guided by the Software Development Lifecycle or SDLC. The SDLC identifies the steps needed to develop high-quality software. Later in this module, we will discuss the SDLC, the traits of high-quality software, and roles common in the field of software engineering. In this video you learned that: Software engineering is the systematic approach to design and development of software. Responsibilities of a software engineer include: Designing, building, and maintaining software systems, writing and testing code, and consulting with stakeholders, third party vendors, security specialists, and other team members. And finally, software engineers build systems while software developers implement specific functionalities.

1. **Introduction to the SDLC – software development life cycle**

The Software Development Life Cycle, known as the SDLC, is a systematic process to develop high-quality software in a predictable timeframe and budget. The goal of the SDLC is to produce software that meets a client’s business requirements. The SDLC defines phases of the software development process that encompass their own process and deliverables. It is a cycle of planning, design, and development that can be implemented as an iterative approach to software development. Adherence to the SDLC minimizes risks and costs to the development of high-quality, deployable software.

The software development life cycle began to take shape in the mid-1960s as software development began to necessitate a more detailed approach because of its growing complexity. The SDLC led to a more deliberate approach as large corporations needed to manage complex business systems requiring heavy computational resources. In its initial conception, it used what is called the “waterfall method” to manage projects where the development of software follows a linear pattern through discrete stages.

The SDLC has since been adapted, however, to more iterative methods in response to addressing customer needs and shifting requirements. There are some key advantages for businesses in following the SDLC. The first advantage is that it gives development teams a process to follow rather than using an ad hoc approach to improve efficiency and reduce risks.

Secondly, there are discrete phases to the SDLC. Each phase is well defined so that team members know what they should be working on and when. Because of the well-defined phases, it facilitates communication between the customer, other stakeholders, and the development team. The SDLC offers an overview of the process, so stakeholders know where they fit in to that process. Also, since each phase is discrete, cross-domain teams know when they have completed their tasks and when development can move to the next phase. The SDLC provides room for iteration where, at the end of a cycle, the process can circle back to incorporate additional requirements as needed.

Problem solving is incorporated early in the cycle so problems are addressed in a timely fashion and can be addressed in the design phase rather than during coding. Finally, each team member has a well-defined role which reduces conflict and overlapping responsibilities.

Its initial development in the 60s and 70s was driven by the need for a systematic approach because of the growing complexity of software. Key advantages of the SDLC include: A roadmap to the software development process, helping to reduce risk and improve efficiency, Increased communication between the team and stakeholders, Clearly defined and understood responsibilities for each team member, and The ability to be used iteratively, allowing for changing requirements.

1. **Phases of SDLC**

There are generally six phases in the SDLC process: planning, design, development, testing, deployment, and maintenance. Each phase is discrete meaning that tasks from a previous phase do not overlap with tasks in the next phase. The original SDLC was conceived as a traditional waterfall method where the phases are linear, but have since been adapted to introduce iteration so that shifting requirements can be accommodated. Note that some organizations may have different names for each stage. For example, “planning” may be called “requirements” or “strategy” or “analysis”. Also, some organizations may have additional or fewer stages.

**Planning Phase**

In the first stage of the SDLC, the planning phase, requirements are gathered, analyzed, documented and prioritized. When planning a software solution, the following factors must be considered: users of the solution the overall purpose of the solution, data inputs and outputs, legal and regulatory compliance, risk identification, quality assurance requirements, allocation of human and financial resources, and project scheduling. As part of the planning process, labor and material costs are estimated and weighed against time constraints. Also, project teams are identified, and roles of each team member are proposed. If stakeholders are struggling to define requirements, often the development team may produce prototypes during the planning stage to tease out those requirements.

A prototype is a small-scale replica of the end product used to get stakeholder feedback and establish requirements. A prototype is used to test basic design ideas. Though prototyping usually occurs during the planning stage, prototyping can occur at various phases of the SDLC whenever requirements need to be reconsidered or clarified as the project develops. After requirements have been gathered, they are combined into a document called a software requirements specification, or SRS, document. The SRS needs to be clearly understood and approved by all stakeholders. The developers are also involved at this stage so they can gain a clear understanding of these requirements.

**Design Phase**

In the design phase, the requirements gathered from the SRS are used to develop the software architecture. Several team members work together at this stage to design the architecture. The architecture is reviewed by the stakeholders and team. And during this phase, prototypes can be designed. A prototype is a preliminary mock-up of the system, or parts of the system, used for demonstration purposes. The document created in this phase is called a design document, and is used by developers during the next phase, which is the development phase.

**Development phase**

The development phase, sometimes called the “building" phase or the "implementation" phase, is when the developers start the coding process once the design document is completed. The project planners use the design document to determine and assign coding tasks. This phase often requires the use of programming tools, different programming languages, and software stacks. Organizations may also have standards or guidelines that need to be followed.

**Testing phase**

The testing phase is next in the process once the coding is complete. And some large projects have dedicated testing teams. Code needs to be thoroughly tested to ensure it is stable, secure, and meets the requirements outlined in the SRS. Testing can be manual, automated, or a hybrid of both. Product bugs are reported, tracked, and fixed, and code is retested until the software is stable. Some common levels of testing include unit testing, integration testing, system testing, and acceptance testing.

**Deployment phase**

The deployment phase is where the application is released into the production environment and made available to users. This can also happen in stages— first, it is released onto a user acceptance testing, also called UAT, platform and once the customer signs off on the functionality, it is released to production. This approach can be used for making software available on a website, mobile device app store, or a software distribution server on a corporate network.

**Maintenance phase**

Maintenance phase happens once the code has been deployed into a production environment. This phase helps to find any other bugs, identify user interface issues, or UI for short, and identify other requirements that may not have been listed in the SRS. Code enhancements can also be identified at this stage. If bugs are discovered in this phase that were missed during testing, these errors may need to be fixed for high-priority issues or incorporated into the requirements as part of a future software release and the process can start over again.

Planning involves requirement gathering and development of the SRS. The architecture is developed during the design phase and the design document is created. The Development phase is when coding takes place, and then during the testing phase issues with the code are found and fixed if possible. Deployment is when the code is released to the production environment. And finally, in the maintenance stage feedback is collected from stakeholders, other UI issues may be identified, and code enhancements suggested. And this information then can be fed into another software development cycle if necessary.

1. **Building quality software**

The software requirements specification, or SRS, encompasses the process of collecting and documenting the set of requirements that the software needs to adhere to. It may include a set of use cases that describe the business needs and user flows that the software must implement. Software requirements can be classified into four broad categories: functional, external and User Interface, or UI, system features, and non-functional.

Software design is the process of transforming the requirements into a structure that is implementable using code. The software design process translates the requirements into a language the developers can use to write the code. It transforms the requirements into a software solution. The technical lead breaks down requirements into sets of related components with clearly defined behaviors, boundaries, and interactions. These components define the system architecture. The system design incorporates guidance on system functions, performance, security, and platform characteristics. The design communicates business rules and application logic, application programming interface design, which is how apps talk to each other or communicate with the database, user interfaces, and database design.

Code quality refers to the characteristics of the code including attributes such as maintainability, readability, testability, and security. Quality code must fulfill the intended requirements of the software without defects. Additionally, it should be clean and consistent, easy to read and maintain, well documented, and efficient. Coding for quality entails following a set of coding practices during development. These include: following common coding standards, conventions, patterns and styles, using automated tools, known as linters, to detect programmatic and stylistic errors, and commenting in the code itself to make it easy for others to understand and modify.

Software testing is the process of verifying that the software matches established requirements and is free of bugs. Its purpose is to identify errors, gaps, or missing requirements when compared with stated requirements. Properly tested software ensures reliability, security, performance, and efficiency. Software testing can often be automated or done manually. Levels of testing include unit, integration, system, and user acceptance. Unit testing is often done by the developer and tests the smallest component of code that can be isolated from the rest of the system. Once the components are integrated into the larger product, integration testing occurs.

Then, after the larger product is deemed completed, system testing can take place. User acceptance testing, or UAT for short and sometimes called beta testing, is when the software is tested by the intended end user. Types of testing can broadly be divided into three categories, functional, non-functional, and regression. When the newest version of the software is distributed, it is referred to as a “release.” Different types of releases are intended for different audiences. There is generally an “alpha,” a “beta,” and a “GA” release. GA stands for general availability. The alpha release is the first functioning version of the system released to a select group of stakeholders. The alpha release likely contains errors and may not contain the full feature set but does contain most of the desired functionality.

Design changes may still occur during this release stage. The beta release, also called a limited release, is given to the stakeholders outside of the developing organization. One of the intents of the beta release is to try out the software under real conditions, test the functionality, and identify any outstanding bugs or errors. The beta release should meet all the functional requirements. Then, after beta release changes are agreed upon, made, and tested, and a stable version is released. The audience for the GA release is all users. Software documentation should be provided to both non-technical end-users and technical users.

System documentation is geared towards the technical user. Technical users may be other engineers, developers, or architects. System documentation explains how the software operates or how to use it. It consists of README files, inline comments, architecture and design documents, verification information, and maintenance guides.

User documentation is provided to the non-technical end-users to assist them in the use of the product. Generally, user documentation is provided in the form of user guides, instructional videos and manuals, online help, and inline help.

More details about documentation will be discussed in another video. In this video, you learned that: Requirement gathering is collecting and documenting the set of requirements that the software needs to adhere to. Designing transforms requirements into a structure that developers can use. Coding for quality entails following a set of coding practices during development. Testing is the process of verifying that the software matches established requirements and is free of bugs There are three types of releases including: alpha, beta, and general availability. And finally, documenting requires text or video that explains the software to technical and non-technical users.

1. **Requirements**

Requirement gathering is a six-step process of defining a problem to be solved and documenting how to go about solving that problem. These steps include: identifying stakeholders, establishing goals and objectives, eliciting requirements from the stakeholders, documenting the requirements, analyzing and confirming the requirements, and prioritizing.

Generally, the stakeholders work for the organization that requests the development of the software product. Key personnel from the organization may include decision-makers, end-users, system administrators, engineering, marketing, sales, and customer support personnel. It is good to have a representative from every group that the product affects. The goals of the product should be clearly defined. Goals are broad, long-term achievable outcomes. Goals can include customer outcomes and business goals.

Next, objectives should be identified. Objectives are more specific than goals and they are actionable and measurable actions that achieve the stated goals. The next three steps, eliciting, documenting, and requirement confirmation are usually completed iteratively. Elicitation can be accomplished through surveys, questionnaires, and interviews. As the requirements emerge, they should be documented and checked to ensure they align with the goals and objectives. Documented requirements should be easily understood by stakeholders and the project team. In order to confirm the requirements, they should be analyzed to ensure consistency, clarity, and completeness. And after analysis, the requirements should be shared with and approved by the stakeholders. After confirmation, requirements should be prioritized. Labels such as “must-have,” “highly desired,” and “nice to have” are helpful. If possible, order the requirements within those categories.

Typically, there may be three documents that result from the requirements gathering process: software requirements specification, or SRS, user requirements specification, or URS, and system requirements specification, or SysRS. The most common of these is the software requirements specification.

The software requirements specification, or SRS, is a document that captures the functionalities that the software should perform and also establishes benchmarks or service levels for its performance. Parts of an SRS include: A purpose statement that contains the intended use of the SRS, its audience and scope, constraints, assumptions and dependencies, and requirements, which can be sorted into four categories: Functional requirements, External Interface requirements, System Features and Non-functional requirements.

The product’s purpose describes who will have access to the SRS and how they should use it. The scope describes the benefits of the software, its goals, and objectives. The second part of the SRS should detail constraints, assumptions, and dependencies. Constraints describe how the product must operate under given conditions that may limit options in the design phase such as confirmation to standards or hardware limitations. Assumptions may include things like a required operating system or hardware that is needed by the software to function.

Dependencies on other software products should also be noted. Requirements can be classified into four categories. Functional requirements are those that cover the functionalities of the software. External requirements are the requirements that address the behavior of the software in relation to external entities such as users and interactions with other hardware or software. System features are a subset of functional requirements. These are required features for the system to function. There are also non-functional requirements such as specifying performance, safety, security, and quality standards. User requirements describe the business need and expectations of the end-users from the software system. The user requirements are written as “user stories” or “use cases” that answer three questions: Who is the user? What is the function that needs to be performed? And why does the user want this functionality?

User acceptance testing determines if these requirements have been met. Often though, the user requirements and software requirements are combined into a single SRS document. The SRS details the expectations of the software system. The System Requirement Specification document, or SysRS, to differentiate it from the SRS, clearly outlines the requirements of an entire system. The system requirement specification is often used interchangeably with software requirement specification, but the SysRS is actually broader in scope than the SRS. Many software projects develop an SRS rather than a SysRS. The SysRS contains system capabilities, interfaces, and user characteristics. It also may include policy requirements, regulation requirements, personnel requirements, performance requirements, security requirements, and system acceptance criteria. It also outlines expectations of the hardware needed for the system in addition to software requirements.

The requirement gathering process entails identifying stakeholders, establishing goals and objectives, and eliciting, documenting, confirming, and then prioritizing requirements. The SRS documents functional, external, system, and non-functional requirements. The URS documents user stories. And finally, the SysRS documents system capabilities and acceptance criteria, and policy, regulation, personnel, performance, security, and hardware requirements.

1. Software Development Methodologies

There are many ways to approach software development. A specific methodology for developing software is commonly used in order to assist the development team to clarify communication among team members and determine how and when the information is shared. We will discuss three of these approaches: Waterfall, V-shape model, and Agile. In the beginning, when the SDLC was conceived, it implemented what is known as the waterfall method.

**Waterfall method**

Waterfall is a sequential method of software development where the output of one phase is the input for the next phase of the cycle. Development and work on the next phase start only after the completion of the previous phase. All planning, such as defining requirements and architectural design, is done up front. The customer usually does not see the product until it is in the testing phase. For a major version release of the product, the same process is repeated resulting in long intervals, such as years, between releases.

**V-shape model**

The V-shape model is named as such because the phases form the shape of a V. The phases going down the left side of the V are called “verification". Then, going up the right side of the V, those phases are called, "validation." The V-shape model is like waterfall in that it is also sequential. Each phase in verification corresponds with a validation phase. There are four stages that occur on each side of the V. Going down the V are planning, system design, architecture design, and then module design. The bottom of the V is the coding phase. And going back up the V are the four phases that correspond to the phases going down the V: unit testing, integration testing, system testing, and acceptance testing. The tests are written during the verification phases on the left and executed during the validation stages on the right.

**Agile model**

Now, the Agile model is different. It focuses on a collaborative software development process over multiple short cycles rather than a strictly top-down linear process. Agile is what is called an iterative approach to development. It still aligns with the SDLC, but each phase is short. Teams work in cycles, or sprints, which are usually one to four weeks long. Unit testing happens in each sprint to minimize the risk of failure. Rather than the “maintenance” stage of the SDLC, the final stage of the sprint is a feedback stage. At the end of each sprint, a chunk of working code is released at a meeting called the “sprint demo” where stakeholders can see the new functionality and provide feedback. After the sprint demo, the entire process is repeated for every sprint cycle. After several sprint cycles, a minimum viable product, or MVP, is developed so stakeholders can provide feedback on the basic feature set. The MVP contains a feature set to validate assumptions about the software.

The four core values of Agile development outlined in what is known as the "Agile manifesto" are: individuals and interactions over processes and tools working software over comprehensive documentation customer collaboration over contract negotiation, and responding to change over following a plan. The main difference between traditional SDLC methods such as waterfall and the v-shape model compared to the Agile method of software development is the former are sequential whereas Agile is cyclical.

Traditional SDLC methods, such as waterfall and V-shape, center around the whole product being developed before soliciting customer feedback, whereas Agile focuses on quick, short bursts of development. There are pros and cons to each method, though Agile is probably the most popular method used in modern software development. Regarding the pros of the waterfall method, it is easy to understand and follow. Each stage is discrete and well-defined, making it easy for all team members to understand their roles. Also, since planning is done upfront, it is easier than iterative methods to estimate a budget and allocate resources.

That said, waterfall lacks flexibility. Since all planning is done upfront if a requirement is changed or overlooked that change can be hard to incorporate at a later date. Inevitably, unforeseen complications happen, or agreed upon functionality shifts from what was initially envisioned. Like waterfall, the V-shape model is simple and easy to use. It is even more rigid than waterfall but designing test plans during the verification phase saves considerable time during coding and validation phases. Drawbacks are also similar to waterfall because it does not readily accommodate changing requirements. Once an application is in the testing phase it is extremely difficult to go back and change functionality.

Agile development is different, relying on ongoing research, planning, and testing during product development. When adding new features to a project, development still goes through the same phases as in traditional SDLC, but with Agile, new, and changing requirements are handled quickly and easily because planning is initiated at the beginning of each sprint cycle. Most resources are spent on the building phase. At the end of each cycle, the QA team, stakeholders, and the customer have some piece of working code to test against requirements and are encouraged to provide feedback.

As coding languages and technologies have developed in recent years, they now allow for modular design, where developers can focus on smaller chunks of code that are readily integrated into the larger product. These small chunks can be released to provide the MVP. Cons of Agile are that upfront planning such as budgeting and scheduling can be challenging because the overall scope of the product is not clearly defined.

Three of the common approaches to software development include waterfall, V-shape model, and Agile waterfall and V-shape are sequential whereas Agile is iterative both waterfall and V-shape models are easy to implement but neither accommodates changing requirements well and Agile allows for changing requirements but resource allocation can be challenging.

1. **Software Versions**

Software versions tell us a lot about programs and applications. Users can determine what software version they are using, and developers can provide useful information with version numbers. Software version numbers vary in length and meaning; however, most version numbers follow a similar format and represent similar information. Version numbers indicate when the software was released, when it was updated, and if any minor changes or patches were made to the software. Software versioning is how software developers keep track of new software, updates, and patches for programs and applications.

Version numbers can be displayed in several ways. Version numbers can be short or long, depending on the software and the preference of the developer, with 2, 3, or 4 number sets. Each number set is divided by a period. The first release of an application or program might have a 1.0 as the version number to indicate no updates, patches, or fixes to the software.

Note: A version still in beta or testing could have a version number lower than 1, such as 0.9. A program or application with many releases and updates will have a longer number, sometimes 4 different number sets within the version number Some software developers may use dates for their versioning. For example: Ubuntu Linux version 18.04.2 was released in 2018 April. The third number set, point-2, designates an additional change or update.

**What do these numbers mean?**

Some version numbers follow the semantic numbering system and have 4 parts separated by a period, but not all numbering systems follow this 4-part example. In semantic numbering, the first number indicates major changes to the software, such as a new release. The second number indicates that minor changes were made to the software. The third number in the version number indicates patches or minor bug fixes. Finally, the fourth number indicates a build number or a build date, and it can indicate less significant changes made.

Lack of compatibility between old and new versions of software is a common problem. You can troubleshoot compatibility issues by viewing the software version to determine if you are using an outdated version of the software. Sometimes updating software to a newer version will resolve compatibility issues. Some software is backwards compatible. If a program or application is backwards compatible, then the older versions of files, programs, and systems will work properly with newer versions.

Version numbers indicate the history of changes, updates, and patches to software, Some version numbers follow the semantic numbering system and have 4 parts separated by a period, Compatibility with old and new versions of software is a common problem, and you should view the version of the software you are using to determine software compatibility.

1. **Software Testing**

Software Testing is the practice of integrating quality checks throughout the software development cycle. The purpose of testing is to check whether the software matches expected requirements and ensure error-free software. In order to test software, the team writes “test cases.” These test cases are written to verify the functionality of a software application and ensure requirements have been satisfied. Test cases can be written in different stages of the SDLC and may vary depending on the type of test or the method used to develop the software, such as Agile or waterfall. A test case contains: steps, inputs, data, and the expected corresponding outputs.

Regardless of the test type or development method, test cases should always be written after requirements are finalized. Software testing helps evaluate the software to identify whether or not the software product meets requirements and is error-free. Types of testing can be broadly classified into three categories: Functional testing, Non-Functional testing, and Regression testing.

**Functional testing**

Functional testing usually involves black box testing which is a method of testing without looking at source code or internal structure. Functional testing is only concerned with inputs and corresponding outputs of the system under test, also called the SUT. It is entirely based on testing functional requirements. Functional testing can be carried out manually or using automated tools. The goal is to test the functionality of the application making sure the application is usable and accessible. Functional testing tests the SUT, to make sure it meets functional requirements. Functional testing makes sure that when user errors or input edge cases do occur, the software handles those exceptions seamlessly by displaying appropriate error messages.

**Non-functional testing**

Non-functional testing includes testing the application for attributes like performance, security, scalability, and availability. Non-functional testing checks to see if the SUTs non-functional behavior is performing properly. Non-functional testing should answer questions like the following: How does the application behave under stress? What happens when many users log in at the same time? Are the instructions in documents and user manuals consistent with the application’s behavior? Does the application behave similarly under different operating systems? How does the application handle disaster recovery? And how secure is the application?

**Regression testing**

Regression testing, also called maintenance testing, confirms that a recent change to the application, such as a bug fix, does not adversely affect already existing functionality. Regression testing should occur when there has been a change in requirements or when defects have been fixed. In order to conduct regression testing, all or some of the test cases should be selected to test against the application. Regression test case selection and prioritization can be challenging and can depend on several factors. Common reasons for regression test case selection include cases that: have frequent defects , contain frequently used functionality, contain features with recent changes, or are complex test cases, edge cases, and randomly successful or failed test cases.

Now that we have discussed different types of testing, let’s discuss testing levels. There are four testing levels: unit, integration, system, and acceptance. Each level occurs at a different time in the SDLC. There are 4 different levels in order to reduce the amount of time spent on testing by preventing overlap. We will discuss each of these testing levels next.

**Unit testing**

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. It is performed by the software developer or engineer during the development phase of the software development life cycle. Unit testing aims to eliminate construction errors before code is integrated with other modules. Unit testing is intended to increase the quality of the resulting software as well as the efficiency of the overall development process.

**Integration testing**

Integration testing seeks to identify errors when two or more smaller, independent code modules are combined. Integration testing is another type of black-box testing. Prior to integration testing, smaller, independent code modules that passed unit testing are incorporated into the larger software application. After modules are integrated together, then integration testing can occur. Integration testing exposes bugs that occur when those smaller units of code interact with each other. Integration testing uncovers deficiencies in communication with a new module in conjunction with other existing modules, databases, or external hardware. Integration testing uncovers situations where bugs develop due to differing programming logic between modules, for instance. Also, sometimes during module development, requirements change, and the module isn’t fully unit tested. Poor exception handling can cause problems when modules are integrated together.

**System testing**

System testing occurs after integration testing and is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. It validates the system as a fully completed software product. System testing is both functional and non-functional. System testing is done in a staging environment, which should be similar to the production environment.

**Acceptance testing**

And finally, acceptance testing is formal testing with respect to user needs, requirements, and business processes. It determines whether a system satisfies the needs of the users, customers, and other stakeholders. Acceptance testing is usually done by the customer or the stakeholders during the maintenance stage of the SDLC. In this video you learned that: There are three categories of testing: functional, non-functional, and regression .

Unit testing verifies small, independent chunks of code. Integration testing looks for errors when two or more small chunks of code are combined. System testing validates the system as a fully completed software product and acceptance testing verifies correct implementation of user requirements and business processes.

1. **Software documentation**

Software documentation is information about the software that describes what the product is and how to use it. These can be written, video, or graphical assets associated with a software product’s development and use. Documentation can be in any of these three formats. Documentation is an essential aspect of software engineering applicable across all the phases of SDLC. Software Documentation can be written for different types of audiences – such as end users, software developers, QA engineers, system administrators and other stakeholders. Documentation can be divided into two categories, product and process. Product documentation relates to the product’s functionality, whereas process documentation describes how to complete a task. Process documentation should provide the requirements for the quality implementation of a business process.

**Product Documentation**

Now, let’s discuss some specific types of product documentation. There are many types of documentation, and we will discuss five categories including requirements, design, technical, quality assurance, and user documentation.

Requirements documentation is written during the planning phase of the SDLC and is intended for the development team including the developers, architects, and QA personnel. Requirements documentation describes the expected features and functionality of the software system. It includes the software requirements specifications, system requirement specifications, and user acceptance specifications.

Design documentation is written by the software architects and the development team to explain how the software will be built to meet the requirements. It consists of both conceptual and technical design documents. Technical documentation includes comments written in the code to help other developers read the code to understand its behavior. It also may include working papers that explain how the code works and documents that record engineers’ ideas and thoughts during project implementation.

Quality assurance documentation includes all documents that pertain to a testing team’s strategy, progress, and metrics. Types of test documentation include test plans, test data, test scenarios, test cases, test strategies, and traceability matrices. Traceability matrices map test cases to their requirements.

User documentation is intended for end-users and explains how to operate the software or help them to install or troubleshoot the system. End-user documentation includes frequently asked questions, installation and help guides, tutorials, and user manuals.

**Process documentation**

Standard operating procedures, called SOPs, often accompany process documentation. Process documentation provides an overview of a process, but SOPs go through much greater detail. The SOP is written documentation that explains step-by-step how to accomplish a common, yet complex task that is organization specific. For example, checking in code using a code repository is common knowledge for a software engineer. However, an organization might have specific steps to follow for that organization in order to get code merged into the main branch. The SOP documentation explains those steps in detail. SOPs can be in the form of a flowchart, a hierarchical outline, or step-by-step instructions.

Documentation, in any form, must be kept up to date. Take for instance online user manuals. If a cloud-based application user interface changes, then the accompanying online documentation must be updated accordingly. Businesses need to ensure they allot resources for this step. With regards to the software development and the SDLC, updating documentation happens during the maintenance phase. Ideally, documentation should also be reviewed periodically to ensure its accuracy.

In this video you learned that: Documentation comes in three formats: written, video, or graphical. Process documentation describes how to complete a task. Product documentation relates to how a product functions. The types of product documentation include requirements, design, technical, QA, and user. And SOPs are written instructions detailing an organization's specific procedure.

1. **Roles in Software Engineering Projects**

There are several common roles on a software development project. And these roles can have different names depending on the approach being used, such as Agile or waterfall. Sometimes different companies have different names for similar jobs. But, not all projects will have all these roles. The roles we will discuss in this video are project manager or scrum master, stakeholder, system or software architect, UX designer, software developer, tester or QA engineer, site reliability or Ops engineer, product manager or owner, and technical writer or information developer. Now let’s describe each role and some of the responsibilities for each job. Traditional SDLC methods have project managers but in Agile the equivalent role is called a Scrum master. A project manager makes sure the project runs smoothly and facilitates communication about the project.

**The project manager** often deals with bigger picture issues such as:

* Planning, scheduling, and budgeting;
* Allocating personnel and resources;
* Executing the software plan;
* and Team communication.

In Agile, there is a **Scrum master.** Rather than focusing on planning, the Scrum master is focused on ensuring team and individual success. Remember that the four core Agile values prioritize people and communication over process, and the Scrum master is responsible for facilitating that communication. The stakeholders are the people for whom the product is being designed. They include individuals such as the customer, end-users, decision-makers, system administrators, and other key personnel. The stakeholder is mainly responsible for defining project requirements and providing feedback if the team members need clarification on requirements or if a proposed solution cannot be solved as planned. The stakeholders may also sometimes participate in beta testing and acceptance testing before the software is released.

The system architect, designs and describes the architecture of a project as well as communicates that architecture to team members. They are sometimes also called a software architect or a solution architect. They are responsible for designing the essential characteristics of the inner structure and technical aspects of the software. The architect provides technical support across the different stages of the SDLC. Note: software architecture will be discussed in further detail in another module.

UX means user experience. The goal of a UX designer is to balance making the software intuitive but also as robust as it needs to be to address requirements. They define how the software behaves from the user’s perspective. The UX designer determines how the software communicates its functionality to the end-user and how the end-user interacts with it.

Next, the developers write the code that powers the software. Responsibilities include implementing the architecture laid out in the design document, incorporating the requirements laid out in the software requirements specification, and employing the UX requirements determined by the UX designers.

Testers or QA engineers are in-charge of ensuring the quality of the product and that the software solution meets customer requirements. They are responsible for writing and executing test cases to identify bugs or deficiencies and provide this feedback to the development teams.

A site reliability engineer, sometimes called an SRE or ops engineer, bridges development and operations by combining software engineering expertise with IT systems management. They track incidents and facilitate meetings to discuss them. They also:

* automate systems, procedures, and processes;
* assist with trouble shooting;
* and ensure reliability for the customer.

The product manager or product owner has the vision of what the product should look like. They have an intimate understanding of the client’s requirements, and the end-user’s needs. They are responsible for leading development efforts to create the software and for ensuring the product provides the value stakeholders are looking for.

Finally, the technical writer or information developer writes documentation for the end-user. They write documentation on technical material geared towards a non-technical audience. Not only does this documentation help the end-user to use the software, but it also helps the customer so they can provide timely feedback to the development teams. Technical writers may be asked to write user manuals, reports, white papers, and press releases.

1. Summary

* Software engineering is the application of scientific principles to the design and creation of software.
* Responsibilities of a software engineer include designing, building, and maintaining software systems.
* Using the SDLC can improve efficiency and reduce risks by:
* letting team members know what they should be working on and when
* facilitating communication between the customer, other stakeholders, and the development team
* letting stakeholders know where they fit into that process and
* letting cross-domain teams know when they have completed their tasks so development can move to the next phase.
* Common software engineering processes are requirements gathering, design, coding, testing, releasing, and documenting.
* The requirement gathering process entails identifying stakeholders, establishing goals and objectives, eliciting requirements from the stakeholders, documenting the requirements, analyzing, prioritizing, and confirming the requirements.
* An SRS is a document that captures the functionalities that the software should perform and also establishes benchmarks or service levels for its performance.
* A URS is a subset of the SRS that details user specification requirements.
* The SysRS contains the same information as an SRS, but can also additionally include system capabilities, interfaces, and user characteristics, policy requirements, regulation requirements, personnel requirements, performance requirements, security requirements, and system acceptance criteria.
* Waterfall, V-shape model, and agile are all different methodologies for implementing the software development life cycle.
* Functional testing is concerned with inputs and corresponding outputs of the system under test, non-functional testing tests for attributes such as performance, security, scalability, and availability. Whereas regression testing confirms that a recent change to the application, such as a bug fix, does not adversely affect already existing functionality.
* Types of documentation include requirements, design, technical, quality assurance, and user.
* There are many different roles involved in a software engineering project. Some of them include project manager or scrum master, stakeholder, system or software architect, UX designer, software developer, tester or QA engineer, site reliability or Ops engineer, product manager or owner, and technical writer or information developer.