Day- 1 (Unit –1)

(Foundations of Software engineering and Quality

Assurance)

	Topics to be covered
1	Introduction to Software Engineering
2	Overview of Quality Assurance (QA)
3	Differences between QA, QC, and Testing
4	Role of QA in the Software Development Life Cycle (SDLC)

Introduction to Software Engineering:

Software Engineering is the disciplined application of engineering principles to the design, development, maintenance, testing, and evaluation of software. The goal of software engineering is to produce software that meets the requirements of its users, is reliable, efficient, and scalable, and can be maintained and updated easily. It encompasses all phases of the software lifecycle, from requirements gathering to system design, coding, testing, deployment, and maintenance. The focus of software engineering is to create high-quality software that meets functional and non-functional requirements while minimizing risks and costs.

Key aspects of software engineering:

- **Systematic Approach:** Applying a structured approach to software development ensures consistency and quality.
- **Collaboration:** Software engineering involves multiple stakeholders, including developers, testers, designers, business analysts, and clients.
- **Documentation:** Proper documentation is essential for understanding, maintaining, and updating the software.
- **Quality:** Emphasis on delivering software that meets user expectations and performs reliably.

Overview of Quality Assurance (QA):

Quality Assurance (QA) is a systematic process that ensures the quality of software through predefined standards, processes, and methodologies. It aims to prevent defects in the software

rather than just identifying and fixing them later. QA is focused on establishing processes and procedures that help create a high-quality software product by assessing whether the software development processes are effective, consistent, and efficient.

QA encompasses:

- **Process Management:** Designing and maintaining effective processes to avoid defects.
- **Continuous Improvement:** Iterating on processes to ensure higher quality and efficiency.
- **Audits and Reviews:** Reviewing the process and the outcomes to ensure that they meet the required standards.

QA can involve various activities like defining development practices, creating standards, conducting reviews, measuring process effectiveness, and training team members.

Real-life Example: Consider a company that develops a mobile app for online banking. QA in this context ensures that the development processes for features like login authentication, transaction history retrieval, and security measures meet the highest standards. The QA team defines the processes that the development team follows, ensuring that features are implemented according to best practices and reducing the likelihood of issues.

Differences between QA, QC, and Testing:

In the software development context, Quality Assurance (QA), Quality Control (QC), and Testing have distinct roles, although they are all concerned with ensuring the final product is of high quality.

1. **Quality Assurance (QA):**
    - **Focus:** QA is process-oriented. It aims to ensure that the software development processes are defined, consistent, and optimized to avoid defects.
    - **Purpose:** Preventing defects from occurring in the first place by ensuring that the processes are followed correctly and continuously improved.
    - **Activities:** Process management, audits, reviews, standardization, training, and monitoring the development lifecycle.
    - **Example:** Establishing coding standards, ensuring developers follow those standards, and ensuring proper documentation is maintained.
2. **Quality Control (QC):**

- **Focus:** QC is product-oriented. It focuses on identifying and correcting defects in the software during or after the development process.
- **Purpose:** Detecting defects in the product and ensuring that the final product meets the quality standards.
- **Activities:** Inspections, reviews, code quality analysis, and defect detection through various checks.
- **Example:** Performing static code analysis to detect potential vulnerabilities or reviewing the software for adherence to functional specifications.

3. **Testing:**
    - **Focus:** Testing is focused on validating and verifying the functionality of the software.
    - **Purpose:** To confirm that the software behaves as expected, performs under desired conditions, and is free of critical bugs.
    - **Activities:** Executing test cases, reporting bugs, regression testing, unit testing, integration testing, etc.
    - **Example:** Running a test to check whether the login functionality works as expected after an update to the authentication system.

Real-Life Example: In a software product development cycle:

- **QA** ensures that the right processes are being followed, such as using the appropriate design patterns, coding standards, and project management practices.
- **QC** detects defects by inspecting the code, reviewing it for errors, and checking whether it adheres to defined standards.
- **Testing** ensures the software works as expected by executing various test cases, such as testing the login feature and the account balance display in the banking app.

Role of QA in the Software Development Life Cycle (SDLC):

Quality Assurance plays a crucial role in the Software Development Life Cycle (SDLC). It is responsible for ensuring that the software development processes are aligned with quality goals, and that the final product meets the expected quality standards. QA is involved in every phase of the SDLC, from the initial requirement gathering to the final deployment and maintenance.

Key Phases in SDLC and the Role of QA:

1. **Requirement Gathering and Analysis:**
    - **QA Role:** QA contributes by ensuring that the requirements are clearly defined, feasible, and complete. QA checks for gaps or ambiguities in the requirements and works closely with the stakeholders to ensure expectations are realistic.
    - **Example:** QA ensures that the security requirements for user data protection are clearly outlined in the banking app.
2. **Design:**
    - **QA Role:** During the design phase, QA reviews the design documents and architecture to ensure that the design follows best practices, scalability, and maintainability guidelines.
    - **Example:** Ensuring that the mobile app design considers different screen resolutions, UI responsiveness, and accessibility standards.
3. **Development (Implementation):**
    - **QA Role:** QA helps ensure that coding standards are followed, development tools are used effectively, and that the developers are following best practices. They also ensure regular code reviews and check-ins.
    - **Example:** The QA team ensures that the mobile app's backend code follows secure coding practices to prevent data breaches.
4. **Testing:**
    - **QA Role:** QA oversees and ensures the testing phase is well-organized and that appropriate tests are conducted for different aspects like functionality, security, performance, and usability. They also ensure that testing is done at multiple levels (unit testing, integration testing, etc.).
    - **Example:** QA ensures that all user scenarios, such as login, password reset, and account transfer, are thoroughly tested.
5. **Deployment:**
    - **QA Role:** QA ensures that deployment processes are smooth and that the software is packaged correctly. They help ensure proper documentation and rollback strategies are in place.

    - **Example:** In the mobile app, QA ensures that the deployment to the app store follows guidelines, and there are no issues in the deployment pipeline.

6. **Maintenance and Support:**
    - **QA Role:** QA ensures that the maintenance process does not introduce new defects. QA teams work with the support team to ensure any reported bugs or issues are addressed without causing regression.
    - **Example:** If users report a bug in the mobile app after an update, QA investigates whether the issue is a result of the new changes and ensures it is fixed without breaking other features.

Key Benefits of QA in SDLC:

- **Early Defect Prevention:** By ensuring that processes are followed, QA helps catch defects early in the SDLC, reducing the cost and effort of fixing them later.
- **Continuous Improvement:** QA constantly evaluates and improves processes to adapt to new challenges and requirements.
- **Higher User Satisfaction:** QA ensures that the product meets the user's expectations, leading to higher user satisfaction and adoption.
- **Cost Efficiency:** By preventing defects early, QA helps in reducing the cost of fixing issues later in the SDLC.
- **Risk Management:** QA helps identify risks in the development process early on and mitigates them proactively.

Real-Life Example: In a large software company developing an enterprise resource planning (ERP) system:

- **Early in the SDLC**, QA ensures that the requirements for features like inventory tracking, sales management, and data integration are clearly defined, and possible risks, like data loss during migration, are identified.
- **During development**, QA ensures the developers follow a consistent coding style and run frequent code reviews to avoid technical debt.
- **Before deployment**, QA ensures thorough testing, such as stress testing the system with large amounts of data to ensure the ERP software performs well.
- **Post-deployment**, QA continues to monitor user feedback and ensures that the system remains bug-free and adaptable to new business requirements.

(Unit –2)

(Foundations of Software engineering and Quality

Assurance)

	Topics to be covered
1	Software Quality Characteristics (ISO/IEC 25010)
2	Introduction to Software Development Life Cycle (SDLC)
3	Phases of SDLC: Requirements, Design, Development
4	Testing, Deployment, and Maintenance

1. Software Quality Characteristics (ISO/IEC 25010)

ISO/IEC 25010 is a standard that defines the quality characteristics of software and provides a framework for evaluating the quality of software products. The standard identifies eight key characteristics that are important for assessing software quality:

1.1 Functional Suitability:

- **Definition:** The extent to which the software satisfies the specified requirements and fulfills its intended purpose.
- **Example:** A banking app must provide accurate account balances, transaction history, and support for money transfers. If these functionalities are working as expected, the app meets the functional suitability requirement.

1.2 Performance Efficiency:

- **Definition:** The software's performance in terms of responsiveness and resource consumption (e.g., CPU, memory, storage).
- **Example:** A video streaming app should not buffer during playback and should consume minimal bandwidth, even when multiple users access the app simultaneously.

1.3 Compatibility:

- **Definition:** The ability of the software to work with other systems, hardware, software, or platforms.
- **Example:** A web application designed for desktop use must also be compatible with mobile browsers (i.e., responsive design) and must work with different operating systems like Windows, macOS, and Linux.

1.4 Usability:

- **Definition:** The ease with which users can learn, operate, and interact with the software.

- **Example:** A mobile app that allows users to easily navigate through menus, use features intuitively, and complete tasks with minimal effort demonstrates high usability.

1.5 Reliability:

- **Definition:** The software's ability to perform its intended function under specified conditions without failure.
- **Example:** An airline booking system must be highly reliable, ensuring users can book flights without unexpected crashes or errors during peak traffic times.

1.6 Security:

- **Definition:** The software's ability to protect against unauthorized access, data breaches, and other vulnerabilities.
- **Example:** An e-commerce platform must ensure secure payment processing, encryption of sensitive user data, and prevention of cyberattacks (e.g., SQL injections).

1.7 Maintainability:

- **Definition:** The ease with which the software can be modified to correct defects, update features, or adapt to changing environments.
- **Example:** A large-scale enterprise resource planning (ERP) system should allow the software team to quickly modify user roles or integrate new modules with minimal disruption.

1.8 Portability:

- **Definition:** The ability of the software to be transferred from one environment to another.
- **Example:** A cloud-based application should be portable in terms of deployment, allowing it to run on different cloud platforms, like AWS, Google Cloud, or Azure, without significant changes.

2. Introduction to Software Development Life Cycle (SDLC)

The **Software Development Life Cycle (SDLC)** is a structured process that defines the stages involved in the development of a software product. The SDLC serves as a roadmap for creating high-quality software that meets user expectations and business goals. It includes several well-defined phases, each focused on specific tasks, from initial planning to post-deployment maintenance.

The SDLC helps in:

- Ensuring the timely delivery of software.
- Reducing risks during the development process.
- Improving communication among stakeholders.
- Providing clear metrics to evaluate progress.

Key SDLC models include the **Waterfall model**, the **Agile model**, the **Spiral model**, and the **V-Model**. Each model follows a similar sequence but differs in its approach to iteration and flexibility.

3. Phases of SDLC: Requirements, Design, Development

3.1 Requirements Phase (Requirements Gathering and Analysis)

- **Description:** The first phase involves gathering requirements from stakeholders, such as end users, business analysts, and product owners, to define the software's functional and non-functional requirements. The requirements are documented in a detailed specification document that serves as the foundation for the design and development phases.
- **Activities:**
    - Identifying the stakeholders.
    - Conducting interviews, surveys, or focus groups to understand user needs.
    - Analyzing business requirements and translating them into technical specifications.
    - Creating use cases and user stories.
- **Example:** In a project to build a customer relationship management (CRM) system, the requirements phase would involve gathering information on customer data storage, sales tracking, and reporting needs. The requirements would specify that the system must allow sales representatives to input customer data and generate monthly sales reports.

3.2 Design Phase (System Design and Architecture)

- **Description:** In the design phase, the software’s architecture, components, interfaces, and data flow are defined. The design is based on the requirements gathered in the

previous phase, and it outlines how the system will function technically. Design can be broken into two categories: **High-level design (HLD)** and **Low-level design (LLD)**.

  - **High-level design (HLD):** Focuses on the overall system architecture and how the system components interact.
  - **Low-level design (LLD):** Focuses on the detailed design of individual components and modules.

- **Activities:**
  - Defining the software architecture.
  - Creating mockups, wireframes, and UI/UX designs.
  - Designing databases and APIs.
  - Defining algorithms and data structures.
- **Example:** In the CRM system, the design phase would focus on defining the database schema to store customer information and transaction records. It would also define the user interface where sales representatives can view and update customer data.

3.3 Development Phase (Coding and Implementation)

- **Description:** During the development phase, the actual software is built according to the design specifications. Developers write code, implement algorithms, and integrate various modules of the software. This is often the longest phase of the SDLC.
- **Activities:**
  - Writing code for individual modules.
  - Integrating external libraries or services.
  - Conducting unit tests to verify individual components.
  - Collaborating with team members to ensure smooth integration.
- **Example:** In the CRM system, the development team would code the functionality for adding and updating customer records, generating reports, and implementing login authentication.

4. Testing, Deployment, and Maintenance

4.1 Testing Phase

- **Description:** After development, the software undergoes thorough testing to ensure it meets the requirements and functions correctly. This phase identifies bugs, defects, and other issues that must be fixed before deployment. Various testing types are conducted, such as unit testing, integration testing, system testing, acceptance testing, etc.
- **Activities:**
    - **Unit Testing:** Testing individual components or functions to ensure they work as intended.
    - **Integration Testing:** Testing interactions between different components and ensuring data flows smoothly between modules.
    - **System Testing:** Testing the entire system in an environment that simulates production to check if it meets the requirements.
    - **User Acceptance Testing (UAT):** End users test the software to verify that it satisfies business needs.
- **Example:** In the CRM system, the testing phase would involve ensuring that the sales tracking feature works correctly, reports are generated as expected, and that the system handles various user roles (e.g., admin, sales rep) without issues. Additionally, testing would be done for security aspects like user authentication.

4.2 Deployment Phase

- **Description:** The deployment phase involves transferring the software from a development or staging environment to a live production environment where it becomes accessible to end users. Deployment can be done in stages (e.g., beta release or full launch) depending on the deployment strategy.
- **Activities:**
    - Preparing the production environment (servers, databases, etc.).
    - Deploying the software to production.
    - Performing a smoke test to check basic functionality in the live environment.
    - Announcing the launch to stakeholders and users.
- **Example:** For the CRM system, after successful testing, the deployment phase would involve setting up the system on production servers, ensuring that all necessary configurations are in place, and making the system available to users. A beta version may be deployed first to a small group of users for feedback before a full-scale deployment.

4.3 Maintenance Phase

- **Description:** After deployment, the software enters the maintenance phase, where it is monitored, updated, and maintained to ensure that it continues to function correctly and meets user needs. This phase can last for the entire life of the software product. Maintenance involves fixing defects, implementing new features, and making updates for compatibility with evolving environments.
- **Activities:**
    - Monitoring the software for performance and reliability.
    - Addressing user-reported issues and bugs.
    - Rolling out patches and updates for new features or security improvements.
    - Managing version control and documentation.
- **Example:** For the CRM system, the maintenance phase would involve monitoring system performance (e.g., speed, uptime), addressing bugs or security vulnerabilities, and adding new features like email integration or AI-based customer insights as requested by users.

(Unit –3)

(Foundations of Software engineering and Quality

Assurance)

1	Traditional SDLC Models: Waterfall
2	Spiral Model
3	Prototype Model
4	Incremental Model

1. Waterfall Model and Its Use Cases

The **Waterfall model** is one of the oldest and most straightforward Software Development Life Cycle (SDLC) models. It is a **linear sequential model**, where each phase must be completed before the next one begins, and there's no overlap or iteration between the phases.

Phases of the Waterfall Model:

1. **Requirement Gathering**: All project requirements are gathered at the start, and a detailed document is created.

2. **System Design**: Based on the requirements, the system architecture is designed.
3. **Implementation**: The actual code is written based on the design.
4. **Integration and Testing**: The developed system is tested for bugs and integration issues.
5. **Deployment**: After successful testing, the software is deployed to the production environment.
6. **Maintenance**: Ongoing support and maintenance after deployment to fix any issues.

Use Cases:

- **Small-scale projects** where requirements are well understood and unlikely to change.
- **Regulatory or compliance-driven projects** where a structured and documented approach is crucial.
- **Projects with a fixed scope**, timeline, and budget, where the path is clear and predictable.
- **Traditional or legacy systems** that require stable functionality and minimal change over time.

Limitations:

- Difficult to accommodate changes after the project starts.
- Not ideal for large or complex projects where requirements evolve over time.
- Testing only happens after the entire development is completed, which can lead to higher risk of undetected issues.

---

2. Comparison of Spiral and Prototype Models in Terms of Adaptability and Risk Management

Both **Spiral Model** and **Prototype Model** are iterative approaches to software development, but they differ in their focus on **adaptability** and **risk management**.

Spiral Model:

The **Spiral Model** is a risk-driven model that combines elements of both the Waterfall and Prototype models. It is divided into several cycles or "spirals," with each spiral consisting of the following phases:

- **Planning**: Define objectives and alternative strategies.

- **Risk Analysis**: Identify risks and try to mitigate them.
- **Engineering**: Develop and test the software.
- **Evaluation**: Evaluate progress, customer feedback, and risks.

Adaptability:

- The Spiral Model is highly **adaptive**. The project is revisited repeatedly, and changes can be implemented as the project progresses.
- It accommodates evolving requirements, making it ideal for large, complex projects where requirements are unclear or likely to change.

Risk Management:

- The **Spiral Model is focused on risk management**. Each cycle involves a risk assessment phase, which helps identify and mitigate risks early in the project, reducing the likelihood of significant project failures.

Use Cases:

- Large-scale, high-risk, and complex projects such as **aeronautical systems** or **defense systems**.
- Projects where there is uncertainty, and frequent changes in requirements are expected.

Prototype Model:

The **Prototype Model** involves creating an early version of the software (prototype) and using user feedback to refine it in subsequent iterations. It helps in clarifying requirements that may not be well-defined initially.

Adaptability:

- The **Prototype Model** is also **adaptive**, allowing changes to be incorporated after feedback is received from the user on the prototype.
- It is particularly useful when the **end-user requirements are not fully known** upfront.

Risk Management:

- While the **Prototype Model** helps in clarifying user requirements early, it may not always manage technical risks (e.g., architectural decisions) well.
- There's a risk of the system becoming too influenced by user feedback, leading to design compromises and potentially bloated functionality.

Use Cases:

- Projects with unclear or evolving user requirements, such as **new software products** or **client-facing applications**.
- **Applications with heavy user interface** or design-centric features where early user feedback is critical.

Comparison:

- **Adaptability**: Both models are adaptable, but the Spiral Model is more robust in handling a wider range of changes due to its continuous risk assessments and planning.
- **Risk Management**: The Spiral Model offers better risk management with its emphasis on iterative risk analysis, while the Prototype Model manages user feedback and functional risks more effectively but may overlook other technical challenges.

---

3. Advantages of the Incremental Model in Handling Iterative Requirements

The **Incremental Model** is a **flexible and iterative approach** where the system is designed, developed, and tested in small parts (increments) over time. Each increment adds functionality to the system, and users can interact with partial but usable versions of the system as it is developed.

Advantages:

1. **Frequent Releases**: The Incremental Model delivers parts of the system in smaller, more frequent releases, which allows users to start using portions of the system earlier.
2. **Flexibility**: It allows for **adaptation to changing requirements**. Since the system is built incrementally, each new iteration can accommodate changes and new features requested by users.
3. **Risk Mitigation**: Risk is reduced because each increment is tested and evaluated separately. Issues are identified early and can be addressed in the next increment.
4. **Customer Feedback**: Since the system is developed in increments, customers can provide feedback on each iteration. This ensures that the product evolves according to the actual needs of the user, improving the chances of success.

Real-Time Example:

Example: Development of a Mobile Banking Application

- **Initial Release (Increment 1)**: Develop basic features like **user login, account balance**, and **transaction history**. The customer gets to use these features early.
- **Second Increment**: Based on user feedback, new features like **bill payments** and **money transfers** are added.
- **Subsequent Increments**: Further enhancements such as **biometric login** and **multi-currency support** are introduced after receiving more user feedback.
- Each increment can incorporate new customer requirements, and bugs or issues are easier to fix in isolated increments, ensuring that the system evolves in a controlled and manageable manner.

Handling Iterative Requirements:

- With iterative requirements, customers may not know what features they need upfront, or their needs might change. The Incremental Model allows the team to deliver the most essential functionalities first and then refine the system based on ongoing user feedback. It works well when new features are discovered or when the scope evolves over time.