**Q1. Describe the difference in between:**

1. **Smoke & Sanity testing.**
2. Purpose:
   1. Smoke Testing: Smoke testing, also known as build verification testing, is performed to ensure that the most critical functionalities of an application or system are working correctly after a build or release. It aims to identify major failures early in the testing process.
   2. Sanity Testing: Sanity testing is conducted to quickly verify that the specific areas or functionalities of an application or system have been modified or fixed correctly, without testing all the detailed features. It helps in ensuring that the immediate issues have been addressed and the system is ready for further testing.
3. Scope:
   1. Smoke Testing: Smoke testing is typically a broader and more comprehensive test that covers the critical and core functionalities of the system. It aims to verify if the major components are functioning properly and to identify showstopper defects that prevent further testing.
   2. Sanity Testing: Sanity testing focuses on specific areas or features of an application that were changed or fixed during development or after a major release. It is a narrow and targeted test, designed to quickly validate the recent changes.
4. Depth of Testing:
   1. Smoke Testing: Smoke testing involves minimal and shallow testing, where the primary objective is to identify critical defects and ensure the basic functionality of the system. It does not delve into exhaustive testing of individual components or detailed scenarios.
   2. Sanity Testing: Sanity testing goes slightly deeper than smoke testing, focusing on the specific functionalities that were modified or fixed. It may involve executing a few test cases related to the changed area to ensure the desired behavior.
5. Timing:
   1. Smoke Testing: Smoke testing is typically performed after the build is deployed, before further extensive testing begins. It acts as an initial check to ensure the basic stability of the system.
   2. Sanity Testing: Sanity testing is performed after the completion of smoke testing and fixes for specific issues. It acts as a quick check to ensure that the fixes or changes made have not introduced any new problems.
6. Test Objective:
   1. Smoke Testing: The main objective of smoke testing is to identify critical issues that could potentially prevent further testing or make the system unusable.
   2. Sanity Testing: The primary objective of sanity testing is to quickly validate the specific changes or fixes and ensure that the system is in a stable state for further testing.

**2) Validation & Verification**

1. Definition:
   * Verification: Verification is the process of evaluating a system or component to determine whether it meets specified requirements. It involves checking and reviewing the system design, specifications, and code to ensure that they align with the predefined requirements and standards.
   * Validation: Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies the customer's requirements. It involves assessing the system's behavior, functionality, and performance to ensure that it meets the intended purpose and user expectations.
2. Focus:
   * Verification: Verification focuses on ensuring that the system has been built correctly. It involves activities such as code inspections, design reviews, and requirement analysis to verify that the software conforms to established standards, guidelines, and specifications.
   * Validation: Validation focuses on ensuring that the system is doing the right thing. It involves activities such as testing, user acceptance testing (UAT), and customer reviews to validate that the software meets the user's needs and performs as expected.
3. Timeframe:
   * Verification: Verification activities are typically performed during the early stages of the software development life cycle (SDLC), such as during the requirements gathering, design, and coding phases.
   * Validation: Validation activities are typically performed during the later stages of the SDLC, such as during the testing and acceptance phases, when the developed software is evaluated against the user's requirements and expectations.
4. Goal:
   * Verification: The goal of verification is to ensure that the software is built correctly according to the defined specifications, standards, and guidelines. It aims to identify and fix defects or deviations early in the development process.
   * Validation: The goal of validation is to ensure that the software meets the user's requirements and expectations. It aims to validate that the software functions as intended and satisfies the user's needs in the intended environment.
5. Process:
   * Verification: Verification involves activities such as inspections, walkthroughs, code reviews, static analysis, and documentation reviews. It focuses on examining the software artifacts to check for compliance with specifications and standards.
   * Validation: Validation involves activities such as functional testing, system testing, integration testing, performance testing, and user acceptance testing. It focuses on executing the software to evaluate its behavior, functionality, and performance.

**Q2) Explain about Agile Methodology.**

Agile methodology is an iterative and collaborative approach to software development that emphasizes adaptability, customer satisfaction, and delivering working software in shorter cycles. It emerged as a response to traditional waterfall methodologies that often struggled to accommodate changing requirements and lacked flexibility. Here are the key aspects and principles of Agile methodology:

1. Iterative and Incremental Development: Agile projects are divided into small iterations or time-bound cycles, often referred to as sprints. Each sprint typically lasts for a few weeks and involves the completion of a set of prioritized features or user stories. The development process is iterative, meaning that feedback from each iteration is used to refine and improve subsequent iterations.
2. Customer Collaboration: Agile methodologies encourage close collaboration and interaction with the customer or end-users throughout the development process. Customers are actively involved in providing feedback, clarifying requirements, and prioritizing features. This ensures that the software being developed aligns with customer needs and expectations.
3. Emphasis on Individuals and Interactions: Agile methodologies prioritize the importance of effective communication and collaboration within the development team. It emphasizes self-organizing teams, where members have the freedom to make decisions and take ownership of their work. Regular face-to-face interactions are encouraged to facilitate quick feedback and problem-solving.
4. Adaptive and Flexible: Agile methodologies embrace change and uncertainty as inherent aspects of software development. They recognize that requirements can evolve or emerge over time. Agile teams are expected to respond and adapt to changes quickly, adjusting their plans and priorities as needed.
5. Continuous Delivery and Feedback: Agile methodologies promote the concept of delivering working software at the end of each iteration. This allows stakeholders to see tangible progress and provide feedback early in the development process. Continuous feedback loops help identify and address issues promptly, reducing the risk of building software that doesn't meet customer expectations.
6. Agile Manifesto and Principles: Agile methodologies are guided by the Agile Manifesto, which emphasizes four key values:
   * Individuals and interactions over processes and tools
   * Working software over comprehensive documentation
   * Customer collaboration over contract negotiation
   * Responding to change over following a plan

**Q3) Explain about Epic and User Stories.**

1. Epic:
   * Definition: An Epic is a large, high-level user requirement or feature that is too big to be implemented within a single iteration or sprint. It represents a significant piece of functionality that can be broken down into smaller, manageable User Stories.
   * Scope: Epics are broader in scope and provide an overarching goal or objective for the development team. They often span multiple sprints and may require further analysis, design, and decomposition into User Stories before implementation.
   * Examples: Examples of Epics could be "User Authentication and Authorization," "Payment Integration," or "Dashboard Enhancements."
2. User Stories:
   * Definition: User Stories are concise, user-centric requirements or features that capture specific functionality or behavior from the user's perspective. They are written in a simple, structured format that describes the desired outcome, who will benefit, and why it's important.
   * Scope: User Stories represent smaller, more manageable units of work compared to Epics. They are typically sized to be implemented within a single sprint and provide the necessary details for development and testing.
   * Format: User Stories often follow the "As a [role], I want [goal], so that [benefit]" format. For example, "As a registered user, I want to be able to reset my password, so that I can regain access to my account if I forget it."
   * INVEST Criteria: User Stories should meet the INVEST criteria, which stands for Independent, Negotiable, Valuable, Estimable, Small, and Testable. This ensures that they are self-contained, flexible, deliver value, can be estimated, are manageable in size, and can be tested.

The relationship between Epics and User Stories can be seen as a hierarchy or decomposition. Epics represent larger, high-level requirements, while User Stories provide the detailed and specific functionality that contributes to fulfilling the Epic. Epics can be decomposed into multiple User Stories, allowing for incremental development and better management of priorities.

During the Agile development process, Epics and User Stories are used to prioritize, plan, and track the progress of software development. Epics are initially identified, and then they are broken down into User Stories that can be estimated, assigned to specific iterations, and implemented by the development team. This iterative approach ensures a more flexible and customer-centric delivery of software functionality.