Assignment -2

Model-2

1. **What is white box testing and list the types of white testing?**

* White box testing based on an analysis of the internal structure of the component or system.
* The testers have no knowledge of how the system or component is structured inside the box.
* For example, when performing system or acceptance testing, the requirements specification or functional specification many from the basis of the tests.
* Typically used at System Test phase, although can be useful throughout the test lifecycle.
* The tester is oblivious to the system architecture and dose not have access to the source code.

**Types of white box testing**

Top Down testing

Hybrid Approach

Bottom up approach

Unit testing

Integration testing

Execution testing

Operations testing

Mutation testing

Statement Coverage

Brach Coverage

Path Coverage

White Box testing

1. **Unit testing:** - **Unit Testing** is a software testing technique in which individual units or components of a software application are tested in isolation.

* **Unit tests** for these code units and run them automatically every time you make changes.
* Unit testing promotes modular code, ensures better test coverage, and saves time by allowing developers to focus more on coding than manual testing.
* Understanding of the software development process.
* Knowledge of programming languages and development tools.

1. **Integration testing: -** Integration testing is the process of testing the interface between two software units or modules.

* The purpose of integration testing is to expose faults in the interaction between integrated units.
* Integration testing is typically performed after unit testing and before system testing.
* Integration testing is typically performed after unit testing and before system testing.

1. **What is error, defect, bug and failure?**

* **Error: -** A human action that produces an incorrect result.
* An error is a mistake, an incorrect action, or a deviation from what is correct.
* Error can occur in many situations, including in speech, writing, calculations.
* **Defect: -** A flaw in a component or system that can cause the component or system to fail to perform its required function.
* A bug is a type of error in software code, while a defect is a broader term that includes bugs and other issues.
* Defect testing is the process of finding and fixing coding errors in software.
* **Failure: -** Deviation of the component or system from its expected delivery, service or result.
* Failure is the social concept of not meeting a desirable or intended objective, and is usually viewed as the opposite of success.
* Failures occur when defects remain undetected or unaddressed and reach the end users.
* **Bug: -** A fault in a program which causes the program which causes the program to perform in an unintended or unanticipated manner.
* In computer technology, a bug is a coding error in a computer program.

1. **Difference between QA v/s QC tester.**

|  |  |  |  |
| --- | --- | --- | --- |
| S.N. | Quality Assurance | Quality Control | Testing |
| 1 | Activities which ensure the implementation of procedures, procedures and standards in context to verification of developed software and intended requirements. | Activities which ensure the verification of developed software with respect to documented (or not in some cases) requirements. | Activities which ensure the identification of bugs/error/defects in the software. |
| 2 | Focuses on processes and procedures rather than conducting actual testing on the testing on the system. | Focuses on actual testing by executing software with intend to identify bug/defect through implementation of procedures and process. | Focuses on actual testing. |
| 3 | Process oriented activities. | Product oriented activities. | Product oriented activities. |
| 4 | Preventive activities. | It is a corrective process. | It is a preventive process. |
| 5 | It is a subset of software test life cycle (STLC). | QC can be considered as the subset of Quality Assurance. | Testing is the subset of Quality control. |

1. **What is 7 key principles? Explain in detail?**

* Software Testing is a crucial part of the Software Development Life Cycle (SDLC), ensuring that the final product meets quality standards. The **7 key principles of software testing** help testers perform efficient and effective testing.

1. **Testing shows presence of defects**
2. **Exhaustive testing is impossible!**
3. **Early testing**
4. **Defect clustering**
5. **The pesticide paradox**
6. **Testing is context dependent**
7. **Absence of Errors fallacy**

1. **Testing shows presence of defects**

* Testing helps in identifying defects in the software but **cannot guarantee** that the software is completely defect-free.
* Even if no defects are found, it does not mean the system is perfect; there may still be hidden issues.
* The goal of testing is to **reduce the number of defects**, not to prove that the software is completely flawless.
* However, Testing cannot prove that there are no defects present.

1. **Exhaustive testing in impossible!**

* It is impossible to test all possible inputs, conditions, and scenarios in a real-world application.
* Instead of testing everything, **risk-based and prioritized testing** is used.
* Testers focus on the most **important functionalities, high-risk areas, and critical features**.
* For example: In an application in one screen there are 15 input fields, each having 5 possible values, then to test all the valid combinations you would need 30 517 578 125 (515) tests.

1. **Early testing**

* The earlier a defect is found in the SDLC, the cheaper it is to fix.
* Testing should start at the **requirement analysis and design phase** rather than waiting until development is complete.
* Early testing, such as **unit testing and integration testing**, helps in identifying issues before they become costly.
* Testing activities should start as early as possible in the development life cycle.

1. **Defect clustering**

* A small number of modules in a software system often contain the most defects.
* This follows the **Pareto Principle (80-20 rule)**: 80% of the defects come from 20% of the modules.
* Testers focus more on these **high-risk areas** to increase testing efficiency.
* Similarly, most operational failures of a system are usually confined to a small number of modules
* An important consideration in test prioritisation!

1. **The pesticide paradox**

* Running the **same test cases repeatedly** will eventually stop finding new defects.
* To overcome this, test cases should be **regularly reviewed, updated, and modified**.
* New test scenarios should be created to improve defect detection.
* As bugs are eliminated by the programmers, the software improves.

1. **Testing is Context Dependent**

* The approach to testing depends on the **type of software** being tested.
* Testing is done differently in different contexts.
* Different kinds of sites are tested differently.
* For example:
* A banking application requires **high security and precision** testing.
* A gaming application requires **performance and usability** testing.
* The testing strategy must align with the business goals and project requirements.

1. **Absence-of-Errors is a fallacy**

* Even if the software is **100% bug-free,** it may still fail to meet user expectations.
* The software must be **tested against requirements and user needs**, not just for technical correctness.
* A product that is error-free but does not solve the user’s problem is considered a failure.
* Even after defects have been resolved it may still be unusable and/or does not fulfil the users’ needs and expectations.

1. **What is boundary value testing?**

* Boundary Value testing is a software testing technique where test cases are created based on the boundary values of input conditions.
* Boundary value analysis is a methodology for designing test cases that concentrates software testing effort on cases near **the limits of valid ranges.**
* The trick is to concentrate software testing efforts at the extreme ends of the equivalence classes.
* Boundary value Analysis (BVA) uses the same analysis of partitions as EP and is usually used in conjunction with EP in the test case design.
* For example, if a system accepts input between 1 and 100, boundary values would be 0,1,100 and 101. These values are tested to check if the system behaves correctly at the limits.
* This technique is widely used because it helps detect issues with minimal test cases while ensuring good coverage.

1. **What is Integration testing?**

* Integration testing is a type of software testing where individual modules or components of a software application are combined and tested as a group.
* The main goal is to verify how different modules interact with each other**.**

**Why is Integration Testing Important?**

* Ensures that different modules work together correctly.
* Defects issues related to data flow and communication between components.
* Helps uncover interface defects, such as mismatched data formats or incorrect API calls.

**Types of integration testing**

1. **Big Bang Integration Testing** – All components are integrated at once and tested together**.**
2. **Incremental Integration testing –** Components are integrated and tested step by step. It includes:

* **Top-Down Approach –** Testing starts from the top module and moves downward.
* **Bottom-Up Approach** – Testing starts from the lower modules and moves upward.
* **Hybrid (sandwich) Approach** – A combination of top-down and bottom-up testing.

**Example of integration testing**

**Imagine an e-commerce application where:**

* The **login module** sends user credentials to the **Authentication Module.**
* The **Authentication Module** verifies the credentials and sends a response back.
* Integration testing ensures that these two modules communicate properly and work as expected.

1. **What is black box testing? What are the different black box testing techniques?**

* Black box testing is a software testing technique where the tester evaluates the functionality of an application without knowing its internal code structure, implementation, or logic.
* The focus is on input and output—ensuring that the software behaves as expected for given inputs.

1. **Equivalence partitioning (EP)**

* Divides input data into different partitions or groups that should produce the same behaviour.
* Example: If an age input field accepts values from 18 to 60, you can test three partitions: below 18 (invalid), between 18-60 (valid), and above 60 (invalid).

1. **Boundary Value Analysis (BVA)**

* Focuses on testing the boundaries between partitions because errors are more likely to occur at edges.
* Example: If a password field accepts 8-16 characters, test cases include 7 (invalid), 8 (valid), 16 (valid), and 17 (invalid).

1. **Decision Table Testing**

* Uses a decision table to test combinations of inputs and corresponding outputs.
* Example: If an e-commerce site offers a discount based on customer type (new/existing) and order amount, a decision table can help cover all possibilities.

1. **State Transition Testing**

* Used when a system behaves differently based on its current state.
* Example: A login system may transition from "Logged Out" → "Logged In" → "Locked Out" based on user actions.

1. **Use case testing**

* Tests real-world scenarios or user interactions with the system.
* Example: Testing an ATM withdrawal process from card insertion to cash withdrawal.

1. **Error Guessing**

* Relies on the tester's experience and intuition to identify common mistakes.
* Example: Testing a registration form by entering special characters or leaving fields empty to check error handling.

1. **What is Equivalence partitioning testing?**

* **Equivalence Partitioning (EP)** is a **black-box testing technique** used to divide input test data into different groups (partitions) that are expected to behave similarly.
* Instead of testing every possible input, testers select one representative value from each partition, reducing the number of test cases while maintaining test effectiveness.

**Key Concept**

* Inputs are **divided into classes (partitions)** based on similar behaviour.
* One test case from each partition is **enough to represent the entire group**.
* Helps in **reducing redundant test cases** while ensuring good coverage.

Example: Validating Age Input (18-60 years allowed)

|  |  |  |
| --- | --- | --- |
| **Input Range** | **Equivalence Class** | **Test Case Example** |
| Below 18 | Invalid | 16 |
| 18 to 60 | Valid | 30 |
| Above 60 | Invalid | 65 |

**Benefits of Equivalence Partitioning**

* Reduces the number of test cases while maintaining efficiency.
* Ensures better **test coverage** with minimal effort.
* Identifies defects effectively without exhaustive testing.

1. **What is exploratory testing?**

* **Exploratory Testing** is a **hands-on, unscripted testing approach** where testers actively explore the software without predefined test cases.
* Instead of following a structured test plan, testers use their creativity, experience, and intuition to find defects and improve the software's quality.

**Key Features of Exploratory Testing**

* **Unscripted & Dynamic** – No pre-written test cases; testers design tests on the fly.
* **Simultaneous Learning** – Testers learn about the application while testing it.
* **Experience-Based** – Relies on the tester's domain knowledge and skills.
* **Flexible & Adaptive** – Can quickly adjust to new findings or changes in the system.
* **Good for Finding Edge Cases** – Helps discover unexpected bugs that structured tests may miss.

**When to Use Exploratory Testing?**

* When **requirements are unclear** or incomplete.
* When there is **limited time** for testing.
* To **complement scripted testing** for better defect detection.
* In **agile environments**, where software changes frequently.

**Benefits of Exploratory Testing**

* Helps uncover **critical bugs quickly**.
* Encourages **tester creativity and critical thinking**.
* Saves time by avoiding unnecessary documentation.
* Useful in **agile and fast-paced projects**.

1. **What determines the level of risk?**

* In software testing, the **level of risk** is determined by two main factors:

1. **Likelihood (Probability) of Failure** – How likely is it that a defect will occur in the software?

* **Complexity of the code**: More complex code has a higher chance of defects.
* **Changes in code**: Frequent modifications increase the chances of introducing defects.
* **Developer experience**: Less experienced developers may introduce more errors.
* **Technology stack**: New or less familiar technologies can increase the likelihood of issues.

1. **Impact of Failure** – How severe would the consequences be if the defect occurs?

* **Business impact**: A bug in a banking system is more critical than in a gaming app.
* **User impact**: If a large number of users are affected, the risk is higher.
* **Legal and compliance issues**: Some defects may lead to legal consequences (e.g., GDPR violations).