Software Testing and Deployment

Objectives of Software Testing

Software testing is the activity of executing a system or component under specified conditions, observing or recording the results, and evaluating some of the aspects of the system or component. It is a software quality approach used for detecting defects in software and conforming that requirements are correctly implemented before deploying the software.

These are the following objectives of software testing:

- **Direct objectives**
  - To identify and reveal as many errors as possible in the tested software
  - To bring the tested software to an acceptable level of quality after correcting the identified errors and retesting – This means that a certain percentage of bugs tolerable to users will remain unidentified upon installation of the software.
  - To perform the required tests efficiently and effectively and within the budgetary and scheduling limitations
  - To establish with a degree of confidence that the software package is ready for delivery or installment at customer premises
- **Indirect objectives**
  - To compile a record of software errors for use in software process improvement by corrective and preventive actions and for decision-making

The following types of testing may be conducted:

- **Unit Testing** – This is performed by the developer on the completed unit or module of the software and before its integration with other modules. Unit tests are technical tests of the code, and these require writing test harnesses and creating test data. The developer writes these tests, and the objective is to determine if each unit of the software performs its expected output. The unit test case is generally documented, and it should include the test objective and the expected results.
- **Integration Testing** – The project team performs this type of testing on the integrated systems once all of the individual units work correctly in isolation. The objective of this test is to verify that all of the modules and their interfaces work correctly together and to identify and resolve any issues. This test benefits from an existing enterprise architecture that provides information on all existing systems and their alignment with each other.
- **System Testing** – This tests the functionality of an entire system together with interdependencies between system components. It may include security testing, usability testing, and performance testing. The objective of this test is to verify that the implementation is valid with respect to the system requirements. It involves the specification of system test cases in which the execution of the test cases will verify that the system requirements have been correctly implemented.

  A test group generally conducts this type of testing, and the system tests are traceable to the system requirements. Any system requirements that have been incorrectly implemented will be identified, and any defects will be logged and reported to the developers.
- **Performance Testing** – The objective of this testing is to ensure that the performance of the system is within the bounds specified by the non-functional requirements. It may include load performance testing, where the system is subjected to heavy loads over a long period, and stress testing, where the system is subjected to heavy loads during a short time interval.

  Performance testing often involves the simulation of many users using the system as well as the measurement of the response times for various activities. Test tools are employed to simulate a large number of users and heavy loads.
- **Acceptance Testing** – System users or customers perform this testing before accepting the product. The customers will see the product in operation and will determine whether or not the system is fit for purpose. The objective of this test is to demonstrate that the product satisfies the business requirements and meets customer expectations.

Test Process in Software Testing

To ensure high quality in software products, integrate software testing into the software development process. Testing activities begin as soon as development activities begin and are carried out in parallel with the development stages. The objective of integrating testing to software development process is to find errors at each phase and prevent these errors from propagating to other phases.

The common phases of a software development process are the following:

- **Requirements analysis and specification** – The objective of testing in this phase is to evaluate the gathered requirements. Each requirement should be evaluated to ensure it is correct and testable, and the requirements together are complete. Software inspections and prototyping can be employed to ensure the good quality of requirements.

  The activities to perform testing in this phase are to prepare for system testing and requirements analysis activities. The test plan should include the scope and objectives for testing at each phase, and the testing requirements should describe support software needed for testing at each stage.
- **System and software design** – This phase partitions the requirements into hardware or software systems and builds the overall system architecture. The objective of testing in this phase is to verify the mapping between the requirements specification and the design. Any changes to the requirements specification should be reflected in the corresponding design changes.
- The testing at this stage helps to validate the design and interface. The activities to perform testing in this phase is to prepare for acceptance and usability testing. An acceptance test plan should include acceptance test requirements, test criteria, and a testing method.
- **Intermediate design** – In this phase, the software system is broken into components and then classes are associated with each component. Design specifications are written for each component and class. The objective of testing in this phase is to avoid mismatches of interfaces.

  The activities to perform testing in this phase is to prepare unit testing, integration testing, and system testing by writing the test plans. The unit and integration test plans are refined at this phase with information about interfaces and design decisions.
- **Implementation design** – In this phase, the developers start writing and compiling classes and methods. The objective of testing in this phase is to perform effective and efficient unit testing. Unit test results and defects should be saved and reported properly for further processing.
- **System deployment** – This is the process of delivering a completed software product to clients. The testing activities in this phase is to perform system testing, acceptance testing, and usability testing. System testing validates whether the software meets the functional and non-functional requirements, while acceptance testing can be started when the system testing is completed. Test cases are derived from acceptance test plans and test data set up.

Test Case

A **test case** or **test scenario** is a software artifact with a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path to verify compliance with a specific requirement and as documentation for a test item. Test cases comprise user inputs that are provided to the application and the procedure for executing the test case during the test.

The format to document test cases varies depending on what is being tested. For example, a test format for a test case that is testing the code is different from the user interface. Listed below is the general test case format that can be extended and used for design:

- **Test Case ID** – a name and number to identify a test case
- **Objective/Description** – the reason for the test (this may include the type of test)
- **Test Procedure** – the activities or steps required on the part of the tester to carry out the test
- **Test data** – the data to enter the system (made up of valid and invalid sets of test data)
- **Expected results** – the expected results of the test
- **Actual results** – a placeholder for recording the actual result as the test case gets executed.
- **Status** – determines whether the test was a Pass or Fail. The status can also be Not Executed.
- **Remarks** – any comments on the test case or execution
- **Administrative details** – may include the name of the tester carrying out the tests, date of execution of the test, the environment in which the test was executed such as hardware and software.

Unit Testing and Test-Driven Development

Unit testing is the activity of writing code that tests other codes. This test focuses on a single unit of the software system, which can be modules such as methods, class, and interface. It is also a source code that can be compiled and executed. As each unit test runs, it reports the test's success or failure with a simple true or false. If all of the unit tests pass, the production code that they test is considered to be working—meeting its requirements. If even a single unit test fails, the production code overall is deemed not to be meeting its requirements.

Developers perform unit tests before writing any production code to create a safety net to catch any subsequent errors when the code changes. If a unit test transitions from a passing state to a failing state, this means that the last change that developers made will be responsible for breaking the code. The process of writing unit tests and then improving code toward better design creates a positive feedback loop whereby the code quality increases while implementing new requirements. In unit testing, test cases are used to evaluate the input and expected output of the unit.

There are two (2) approaches to unit testing:

- **White-Box Testing** – This testing approach is used to verify the internal logic and program statements of components or software. It involves stepping through every line of code and every branch in the code. The tester should be knowledgeable in the software programming language and should understand the structure of the program to perform this test. Then, s/he can look inside the class to review the code itself. The tester will insert a set of input through the code and compare the outputs from the expected results. This testing approach ensures that all program statements and all control structures are tested. Integrated Development Environments (IDEs) contains tools that perform white-box testing.

- **Black-Box Testing** – In this testing, the tester derives tests from external descriptions of the software, including specifications, requirements, and design. This test is concerned with the inputs and outputs of the system where a set of input is inserted to the software's user interface, and the outputs delivered by the software are compared with the expected outputs. The tester focuses on whether the class or module meets the requirements stated in the specifications.

  Black-box testing is performed from the user interface of the software. It invokes the program, gives the necessary inputs to the software, then processes the test data and user inputs to generate outputs, which can be compared with the expected outputs to determine whether the software functioned correctly or not. The effectiveness of black-box testing depends on the design of test cases. If the test cases were extensive, then the testing would also be extensive and detect more defects in the software.

Unit Testing Tools

Unit testing can be performed manually or automatically using unit testing tools or frameworks. Here are some tools for unit testing:

- **JUnit** and **TestNG** – These are open-source unit testing frameworks designed for Java programming language. The JUnit unit testing framework supports test-driven development and can be run as stand-alone Java programs or within an IDE.
- **NUnit** – This is an open-source unit testing framework for all .NET languages.

Test-Driven Development

The **test-driven development (TDD)** is an iterative process where tests for the code are written before writing the code. This approach forces the developer to write testable code. The application is written with testability in mind, and the developers must consider how to test the application in advance. The unit tests are used as a method for deriving the design of a component in an iterative process. The end result of the process is a completely functional component and a set of unit tests to verify that functionality. These same unit tests can be used when further modifications are made to the component to ensure the new modifications do not change the current functionality.

These are the steps of a test-driven development process:

Step 1. Write a unit test that tests the functionality to satisfy a given requirement. Run the test, but it will not even compile because there is no supporting code yet.

Step 2. Write just enough code to compile the test successfully.

Step 3. Run the test but it will fail because no functionality has been put in the code.

Step 4. Write just enough code to satisfy the test.

Step 5. Run the test. It should now pass.

Step 6. Refactor the code as necessary, verifying with tests. Refactoring is a process of improving the code without changing its external behavior.

Step 7. Repeat all the steps by writing another test and until all requirements are coded and testable.

The test-driven development is intended to generate just enough design to pass the unit tests and prevent over-reengineering the code architecture.

Software Deployment and Deployment Tools

Software deployment is the process of delivering a completed software product to clients. This involves packaging, testing, distributing, and installing the software product. Software files can be deployed manually or through automated deployment tools.

Software Deployment Tools

The deployment tools are used to automate deployment tasks and are designed with the following features:

- **Automation** – This eliminates the manual tasks of deploying software products. These tools perform tasks such as bug detection, patch protection, performance testing, and code analyzing.
- **Security** – It manages the permission settings of users and groups who can access the information in the software.
- **Updates** – These automate system updates, scan vulnerabilities, and practice regular patch management of latest software versions across applications. Software patching involves acquiring, testing, and installing the code – known as a patch – into an executable program to provide an update, fix, or improved version of the program or its supporting data.
- **Monitoring** – It monitors and analyzes the activities and interactions of the software users. This helps in optimizing the performance and eliminates issues in the software before they spread in the entire network.

Some of the commercially available deployment tools are as follows:

- **Jenkins** – It is a self-contained, open-source automation server that can be used to automate all activities related to building, testing, and delivering or deploying software. This can be installed through native system packages, Docker, or even run standalone by any machine with a Java Runtime Environment (JRE) installed. Jenkins' functionalities can be extended through the installation of plugins in IDEs.
- **Octopus Deploy** – This automated deployment tool is compatible with ASP.Net, Java, Node.js, and Windows services, as well as various script languages and database types. This is designed to automate application deployments in the cloud, corporate data center, and on-site.
- **Bamboo** – This offers support for the delivery aspect of continuous delivery. Deployment projects automate the releasing into each environment while letting the project team control the flow with per-environment permissions. This supports real-time monitoring across all tools and flag errors of software as soon as they occur.
- **SolarWinds Patch Manager** – This is an automated patch management software for Microsoft servers, workstations, and third-party applications. This facilitates easier patching, reporting, and information gathering for servers and workstations, and allows the project team to manage patch deployments with advanced scheduling and rebooting across servers and workstations.

REFERENCES:

4 Best Software Deployment Tools in 2019. (2019). In *DNSstuff*. Retrieved from https://www.dnsstuff.com/software-deployment-tools

Ammann, P. & Offutt, J. (2017). *Introduction to software testing* (2nd ed.). Retrieved from https://books.google.com.ph/books?id=bQtQDQAAQBAJ&printsec=frontcover#v=onepage&q&f=false

Bamboo. (n.d.). In *Atlassian*. Retrieved from https://www.atlassian.com/software/bamboo

Chemuturi, M. (2014). *Mastering software quality assurance. Best practices, tools and techniques for software developers.* Retrieved from https://books.google.com.ph/books?id=rhO8YW7LaukC&printsec=frontcover#v=onepage&q&f=false

Dennis, A., Wixom, B.H., & Tegarden, D. (2015). *Systems analysis & design. An object-oriented approach with UML* (5th ed.). Hoboken: John Wiley & Sons, Inc.

Galin, D. (2018). *Software quality. Concepts and practice.* Retrieved from https://books.google.com.ph/books?id=cwlbvgAACAAJ&printsec=frontcover#v=onepage&q&f=false

Jenkins User Documentation. (n.d.). In *Jenkins.* Retrieved from https://jenkins.io/doc/

McLean Hall, G. (2017). *Adaptive code. Agile coding with design patterns and SOLID principles* (2nd ed.). Retrieved from https://books.google.com.ph/books?id=18SuDgAAQBAJ&printsec=frontcover#v=onepage&q&f=false

O'Regan, G. (2017). *Concise guide to software engineering. From fundamentals to application methods.* Cham, Switzerland: Springer International Publishing AG.

Patch Manager. (n.d.). In *SolarWinds*. Retrieved from https://www.solarwinds.com/patch-manager

Unhelkar, B. (2018). *Software engineering with UML.* Boca Raton, Florida: CRC Press.

Walkinshaw, N. (2017). *Software quality assurance. Consistency in the face of complexity and change.* Cham, Switzerland: Springer International Publishing AG.