

**International Institute of Professional Studies**

**Devi Ahilya Vishwavidyalaya**

**Indore, M.P**



**Blood Bank Management**

This project is submitted for **V-semester**

For degree of

**Masters in Information Technology– 2024**

Guided by:  
Dr. Shaligram Prajapat Sir

Submitted by:  
Vaidehi Dangi

IT-2k19-69

## **BONAFIDE CERTIFICATE**

Certified that this project titled " Blood Bank Management " is a Bonafide work of Vaidehi Dangi (IT-2k19-69) who carried out the researched and completed the project under my supervision. Certified further, that to the best of my knowledge, the work reported herein does not form part of any other project on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Internal examiner

External examiner

## **RECOMMENDATIONS**

The Project work entitled "Blood Bank Management " submitted by Vaidehi Dangi is satisfactory account of the bonafide work under my supervision and is recommended towards the end of their 5<sup>th</sup> semester of MTECH (I.T)-2024.

**Guided By:**

**Dr. Shaligram Prajapat**

## **ACKNOWLEDGEMENT**

I would like to express my gratitude toward staff of IIPS -DAVV as well as the honorable director Dr. B. K. Tripathi for providing us a great opportunity to complete a project on “Blood Bank Management”. My sincere thanks go to Dr. Shaligram Prajapat for his support and guidance for the completion of this project.

## **ABSTRACT**

The purpose of this study was to develop a blood management information system to assist in the management of blood donor records and ease/or control the distribution of blood in various parts of the country basing on the hospital demands. Without quick and timely access to donor records, creating market strategies for blood donation, lobbying and sensitization of blood donors becomes very difficult. The blood management information system offers functionalities to quick access to donor records collected from various parts of the country. It enables monitoring of the results and performance of the blood donation activity such that relevant and measurable objectives of the organization

can be checked. It provides to management timely, confidential and secure medical reports that facilitates planning and decision making and hence improved medical service delivery. The reports generated by the system give answers to most of the challenges management faces as far as blood donor records are concerned.

The proposed of Blood Bank App helps the people who are in need of a blood by giving them all details of blood group availability or regarding the donors with the same blood group. They don't need to go anywhere to search the blood when they need. They just need to use this software then all the result will appear in just a second. Our life is so busy so we don't have time to spend going here and there, we can use technical way to search the blood by using the Blood Bank software we can find

thousands of people who are donating the blood and also get the detail the of that person that in which city he belongs to and what is the Blood group of that person. So, this is the most useful software ever.



# TESTING

## UNIT TESTING

### Introduction

In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method. Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process. It forms the basis for component testing. Ideally, each test case is independent from the others. Substitutes such as method stubs, mock objects, fakes, and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended.



## **Benefits**

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits.

1) Find problems early: Unit testing finds problems early in the development cycle. In test-driven development (TDD), which is frequently used in both extreme programming and scrum, unit tests are created before the code itself is written. When the tests pass, that code is considered complete. The same unit tests are run against that function frequently as the larger code base is developed either as the code is changed or via an automated process with the build. If the unit tests fail, it is considered to be a bug either in the changed code or the tests themselves. The unit tests then allow the location of the fault or failure to be easily traced. Since the unit tests alert the development team of the problem before handing the code off to testers or clients, it is still early in the development process.

2) Facilitates Change: Unit testing allows the programmer to refactor code or upgrade system libraries at a later date, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified. Unit tests detect changes which may break a design contract.

3) Simplifies Integration : Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.

4) Documentation: Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit, and how to use it, can look at the unit tests to gain a basic understanding of the unit's interface (API). Unit test cases embody characteristics that are critical to the success of the unit. These characteristics can indicate appropriate/inappropriate use of a unit as well as negative behaviours that are to be trapped by the unit.



# **INTEGRATION TESTING**

Integration testing (sometimes called integration and testing, abbreviated I&T) is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

## **Purpose**

The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items. These "design items", i.e., assemblages (or groups of units), are exercised through their interfaces using black-box testing, success and error cases being simulated via appropriate parameter and data inputs. Simulated usage of shared data areas and inter-process communication is tested and individual subsystems are exercised through their input interface. Test cases are constructed to test whether all the components within assemblages interact correctly, for example across

procedure calls or process activations, and this is done after testing individual modules, i.e., unit testing. The overall idea is a "building block" approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages. Software integration testing is performed according to the software development life cycle (SDLC) after module and functional tests. The cross dependencies for software integration testing are: schedule for integration testing, strategy and selection of the tools used for integration, define the cyclomatic complexity of the software and software architecture, reusability of modules and life-cycle and versioning management. Some different types of integration testing are big-bang, top-down, and bottom-up, mixed (sandwich) and risky-hardest. Other Integration Patterns are: collaboration integration, backbone integration, layer integration, client-server integration, distributed services integration and high-frequency integration.





## **Big Bang**

In the big-bang approach, most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing. This method is very effective for saving time in the integration testing process. However, if the test cases and their results are not recorded properly, the entire integration process will be more complicated and may prevent the testing team from achieving the goal of integration testing. A type of big-bang integration testing is called "usage model testing" which can be used in both software and hardware integration testing. The basis behind this type of integration testing is to run user-like workloads in integrated user-like environments. In doing the testing in this manner, the environment is proofed, while the individual components are proofed indirectly through their use. Usage Model testing takes an optimistic approach to testing, because it expects to have few problems with the individual components. The strategy relies heavily on the component developers to do the isolated unit testing for their product. The goal of the strategy is to avoid redoing the testing done by the developers, and instead flesh-out problems caused by the interaction of the components in the environment.

## **Top-down And Bottom-up**

Bottom-up testing is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher-level components. The process is repeated until the component at the top of the hierarchy is tested. All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower-level integrated modules, the next level of modules will be formed and can be used for integration testing. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage. Top-down testing is an approach to integrated testing where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module. Sandwich testing is an approach to combine top-down testing with bottom-up testing.

# **SOFTWARE VERIFICATION AND VALIDATION**

## **Introduction**

In software project management, software testing, and software engineering, verification and validation (V&V) is the process of checking that a software system meets specifications and that it fulfils its intended purpose. It may also be referred to as software quality control. It is normally the responsibility of software testers as part of the software development lifecycle. Validation checks that the product design satisfies or fits the intended use (high-level checking), i.e., the software meets the user requirements. This is done through dynamic testing and other forms of review. Verification and validation are not the same thing, although they are often confused. Boehm succinctly expressed the difference between

Validation: Are we building the right product?

Verification: Are we building the product, right?

According to the Capability Maturity Model (CMMI-SW v1.1)

Software Verification: The process of evaluating software to determine whether the products of a given

development phase satisfy the conditions imposed at the start of that phase.

**Software Validation:** The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.

In other words, software verification is ensuring that the product has been built according to the requirements and design specifications, while software validation ensures that the product meets the user's needs, and that the specifications were correct in the first place.

Software verification ensures that "you built it right".

Software validation ensures that "you built the right thing". Software validation confirms that the product, as provided, will fulfil its intended use.

## From Testing Perspective

- Fault – wrong or missing function in the code.
- Failure – the manifestation of a fault during execution.
- Malfunction – according to its specification the system does not meet its specified functionality

Both verification and validation are related to the concepts of quality and of software quality assurance. By themselves, verification and validation do not guarantee software quality; planning, traceability, configuration management and other aspects of software engineering are required. Within the modelling and simulation (M&S) community, the definitions of verification, validation and accreditation are similar:

- M&S Verification is the process of determining that a computer model, simulation, or federation of models and simulations implementations and their associated data accurately represent the developer's conceptual description and specifications.
- M&S Validation is the process of determining the degree to which a model, simulation, or federation of models and simulations, and their associated

data are accurate representations of the real world from the perspective of the intended use(s).

## **Classification of Methods**

In mission-critical software systems, where flawless performance is absolutely necessary, formal methods may be used to ensure the correct operation of a system. However, often for non-missioncritical software systems, formal methods prove to be very costly and an alternative method of software V&V must be sought out. In such cases, syntactic methods are often used.

## **Test Cases**

A test case is a tool used in the process. Test cases may be prepared for software verification and software validation to determine if the product was built according to the requirements of the user. Other methods, such as reviews, may be used early in the life cycle to provide for software validation.





## **Black-Box**

Testing Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings. This method of test can be applied virtually to every level of software testing: unit, integration, system and acceptance. It typically comprises most if not all higher-level testing, but can also dominate unit testing as well.

## **Test Procedures**

Specific knowledge of the application's code/internal structure and programming knowledge in general is not required. The tester is aware of what the software is supposed to do but is not aware of how it does it. For instance, the tester is aware that a particular input returns a certain, invariable output but is not aware of how the software produces the output in the first place.

## **Test Cases**

Test cases are built around specifications and requirements, i.e., what the application is supposed to

do. Test cases are generally derived from external descriptions of the software, including specifications, requirements and design parameters. Although the tests used are primarily functional in nature, non-functional tests may also be used. The test designer selects both valid and invalid inputs and determines the correct output, often with the help of an oracle or a previous result that is known to be good, without any knowledge of the test object's internal structure.

## **White-Box Testing**

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e., black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g., in-circuit testing (ICT). White-box testing can be applied at the unit, integration and system levels of the software testing process. Although traditional testers tended to think of white-box testing as being done at the unit level, it is used for integration and system testing more frequently today. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test. Though this method of test design can uncover many errors or problems, it has the potential to miss unimplemented parts of the specification or missing requirements.

## **Levels**

1) Unit testing: White-box testing is done during unit testing to ensure that the code is working as intended, before any integration happens with previously tested code. White-box testing during unit testing catches any defects early on and aids in any defects that happen later on after the code is integrated with the rest of the application and therefore prevents any type of errors later on.

2) Integration testing: White-box testing at this level are written to test the interactions of each interface with each other. The Unit level testing made sure that each code was tested and working accordingly in an isolated environment and integration examines the correctness of the behaviour in an open environment through the use of white-box testing for any interactions of interfaces that are known to the programmer.

3) Regression testing: White-box testing during regression testing is the use of recycled white-box test cases at the unit and integration testing levels.



## Procedures

White-box testing's basic procedures involves the tester having a deep level of understanding of the source code being tested. The programmer must have a deep understanding of the application to know what kinds of test cases to create so that every visible path is exercised for testing. Once the source code is understood then the source code can be analysed for test cases to be created. These are the three basic steps that white-box testing takes in order to create test cases:

- Input involves different types of requirements, functional specifications, detailed designing of documents, proper source code, security specifications. This is the preparation stage of white box testing to layout all of the basic information.
- Processing involves performing risk analysis to guide whole testing process, proper test plan, execute test cases and communicate results. This is the phase of building test cases to make sure they thoroughly test the application the given results are recorded accordingly.

- Output involves preparing final report that encompasses all of the above preparations and results.

# SYSTEM TESTING

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black-box testing, and as such, should require no knowledge of the inner design of the code or logic. As a rule, system testing takes, as its input, all of the "integrated" software components that have passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called assemblages) or between any of the assemblages and the hardware. System testing is a more limited type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole. System testing is performed on the entire system in the context of a Functional Requirement Specification(s) (FRS) and/or a System Requirement Specification (SRS). System testing tests not only the design, but also the behaviour and even the believed expectations of the customer. It is also intended to test up to and beyond the bounds defined in the software/hardware requirements specification(s).