CHAPTER4

SYSTEM TESTING IMPLEMENTATION AND DOCUMENTATION

4.1 Introduction

There are a lot of different tests you can run on a system after it's been developed, like unit tests, functional tests, usability tests, and others. Each type of test has its own purpose and helps find different kinds of issues. In this chapter, we'll go over the different types of testing and what they're used for. System testing is an important part of developing software. The main goal of system testing is to make sure the whole system meets the requirements laid out in the system specification. This means testing not just the program itself, but also any other software or hardware it interacts with, and again the users that the system interacts with and their specific needs and preferences.

4.2 Testing Approaches

Testing is very important to anticipate system behavior. The goal of testing is to ensure that systems meet their targets and requirements. During any type of testing, system functionality is put to the test. Testing can unveil any unexpected behavior that may arise from user interactions outside the normal development environment. A lot can occur. The common norm is that we as developers build software and systems thinking that we have anticipated all the errors and solved every bug, but users tend to push our software and code to the limit. By the end of the testing phase, the system would run devoid of errors or at least would be at a decent starting point which can be built upon soon.

4.2.1 Unit Testing

Unit testing is the process of testing the smallest parts of your code, like individual functions or methods, to make sure they work correctly. It's a key part of software development that improves code quality by testing each unit in isolation. Unit testing and selected aspects of test-driven development can be used to improve learning and encourage emphasis on quality and correctness. (Olan, n.d)

A unit test exercises a "unit" of code in isolation and compares actual with expected results. In Java, the unit is usually a class. Unit tests invoke one or more methods from a class to produce observable results that are verified automatically. So essentially, we break down our code as much as possible into their simplest units. We can analyze our functions and even our operations to ensure that we are free of errors as we would catch bugs early, improve our code quality because we might have to refactor, and it would also save as time in the future.

4.24 Functional Testing

Functional testing is a quality assurance process and a type of black box testing that bases its test cases on the specifications of the software component under test. It is the process of using test scenarios and cases to test the software, and involves using the software in all the different ways a user might use it to ensure it works correctly each time(functional testing, 2022)
Functional testing focusses on ensuring that software behaves as expected by verifying its functionality against the specified requirements. This type of testing would ensure that software meets the functional specifications and functions as a user may expect. Functional testing also improves the user experience by stimulating real world scenarios, enhancing system reliability and reduces risks by identifying issues before deployment while ensuring compatibility

4.3.2 Usability Testing

Usability testing can be defined as a technique used in user-centered design to evaluate a product by testing it with real users. According to Jakob Nielsen, a usability expert, "Usability is a quality attribute that assesses how easy user interfaces are to use." Usability testing specifically focuses on the user experience and understanding of the application. (Nielsen, 2012).

Usability testing mainly improves the User experience as it focuses on how real users interact with the application ensuring that the system is satisfying to use. It also enhances accessibility by evaluating the system availability to users with varying ability and needs. Lastly, usability testing boosts retention and engagement as when and if you take the time to do it you ensure that users keep on returning and continue using your system over time.

4.2.4 Acceptance Testing

Acceptance testing is defined as a type of software testing that evaluates whether a system meets its business and user requirements. It is the final stage before the software is released to production. An acceptance test is a formal description of the behavior of a software product, generally expressed as an example or a usage scenario. Acceptance testing ensures user validation as software functions as the user expect, providing the confidence that it meets their needs and expectation. It serves as a final quality check before deployment as it acts as the ultimate validation step. Acceptance testing also combats the likelihood of costly post release fixes and customer dissatisfaction, hence saving time and costs.

4.2.5 Selected Testing Approach

For a facial recognition based attendance application, we thought it best to go with a hybrid approach where we consider three types of testing namely usability testing, functional testing, and finally acceptance testing since we liked to know if the system meets the requirements of the Kantanka Financial Co-operative Union (KFCU). We proceeded to do our unit testing by checking all our software and code units breaking everything down as we go. So, if I imported some functions from a script into my main file, it would be considered a unit, but I would also have to check all the functions and or classes in that script.
Usability testing was also considered because well the system will be used by people, so we wanted them to have a feel of our interface and identify any challenges or any choices that were made in bad taste on our part.
And finally, we would like to do an acceptance test with the Kantanka Financial and Co-operative union (KFCU) to ensure that we met all key business requirements

4.3 IMPLEMENTATION OF CURRENT SYSTEM

Getting a group of people to learn and use a new system can be challenging. Introducing a new application and its procedures can have a big impact on an organization, so it’s important to choose the right implementation strategy. System implementation is a crucial part of the development process, and there are four main strategies to consider: parallel, pilot, phased out, and direct implementation. To pick the best approach, the team needs to carefully evaluate each method and decide which one fits the situation best.

4.3.1 Parallel Implementation

In Parallel Implementation the new system is implemented alongside the old system for the time being. This is best for high-risk organizations and ventures where it would be a major disaster for errors to occur. An example may be any application that involves healthcare or a banking system. If such is the case that it’s a high-risk environment, then it’s worth giving Parallel implementation a go. Although you would have to keep in mind that it is highly costly to run two systems at the same time. In our case, our system is a new and independent system so we don’t really have to worry about this so we would not in fact go with parallel implementation.

4.3.2 Pilot Implementation

Pilot implementation suggests that we make our system available for use but just to a portion of our intended users, so those users would basically be test-running the software and help us to detect any issues with scalability if any. During the implementation, the user base would be increased gradually and then gradually until all users have access to the software. At this points all the problems would have been picked off during any of the pilot groups and be taken care of. This implementation is cheap but would cost you a lot in the form of a longer implementation (Time constraint)

4.3.3 Direct Implementation

"In a direct cutover strategy, the new system replaces the old one on a designated turn-on date" (Laudon & Laudon, 2020, p. 412). In direct Implementation the system is implemented all at once, all its features to all the user base devoid of any previous system. This implementation is best for low-risk avenues and situations where the precedent if any is not valid and to be discarded anyway. This will be our implementation strategy for our verification application, as it is low risk and it’s an independent system which is not really building or dependent on any prior software at the Kantanka Financial Co-operative Union (KFCU)

4.3.4Phased Implementation

With Phased Implementation, the system is introduced in modules(phases) over time. This implementation would generally take longer but would be ideal if you are dealing with a very complex system with a lot of features and sub-features. This way you could roll each feature one at a time. Again, we would not go with this because a facial verification app is very straightforward to implement. At the very least the base version is, and we could always roll our updates and other features later through version control.

4.4 System Documentation

Pending…………………

4.5 Implementation Challenges

Implementation brought many challenges including the implementation method to use, direct, parallel, phased or pilot implementation. We also had to consider the technology to use in building the verification app that uses the model, considering which frameworks are best for integration with machine learning technologies and then the cost also of these services. Also, there was the case of going ahead to build and maintain a server of our own instead of depending on just any deployment service and their APIs to work. Additionally, we faced challenges like ensuring data security and privacy, especially since the app handles sensitive attendance information. There was also the issue of user adoption, as not everyone might be comfortable or familiar with using a new system, which could lead to resistance or errors during the transition. Another challenge was scalability, making sure the app could handle a growing number of users without performance issues. We also had to think about compatibility with different devices and operating systems, as well as how to handle updates and bug fixes without disrupting the system. Lastly, there was the challenge of training staff and users to properly use the app, which required time and resources to create effective training materials and support systems.

4.6 Chapter Summary

This chapter focused on the testing, implementation, and documentation of the facial verification system. We explored various testing approaches, including unit testing, functional testing, usability testing, and acceptance testing, all of which were essential to ensure the system functions as intended and delivers a seamless user experience. After carefully evaluating the different implementation strategies, parallel, pilot, direct, and phased we opted for direct implementation, as it aligned best with the low-risk, standalone nature of our system.

The documentation section detailed the technologies and tools used, such as Python, TensorFlow, and OpenCV, along with the preprocessing steps, model architecture, and training process. I also highlighted the system's limitations, particularly its reliance on a limited dataset and its suitability for verification rather than large-scale recognition tasks.

Throughout the implementation phase, we encountered several challenges, such as selecting the appropriate technology stack, addressing data security and privacy concerns, ensuring scalability, and managing user adoption. Despite these hurdles, the system was successfully developed, tested, and prepared for deployment.