B.Sc. in Computer Science and Engineering Thesis

Quality Assurance in Software Development: The Impact of Unit Test Automation

Submitted by

ABU SAKIB
201120014

ARIFUL ISLAM
201120016

FAZLE ELAHI FAHIM
201120030

Supervised by

A.H.M Saiful Islam

Department of Computer Science and Engineering

Notre Dame University Bangladesh

Dhaka, Bangladesh

December 2023

CANDIDATES' DECLARATION

This is to certify that the work presented in this thesis, titled, "Quality Assurance in Software Development: The Impact of Unit Test Automation", is the outcome of the investigation and research carried out by us under the supervision of A.H.M Saiful Islam.

It is also declared that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

ABU SAKIB
201120014

ARIFUL ISLAM
201120016

FAZLE ELAHI FAHIM
201120030

CERTIFICATION

This thesis titled, **“Quality Assurance in Software Development: The Impact of Unit Test Automation”**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in December 2023.

BOARD OF EXAMINERS

ACKNOWLEDGEMENT

We are indebted to the completion of this undertaking could not have been possible without the participation and assistance of so many people whose names may not all be enumerated. Their contributions are sincerely appreciated and gratefully acknowledged. We are thankful to our supervisor A.H.M Saiful Islam (Professor of the Dept. of CSE) for his continuous guidance and support. Also to all relatives, friends, and others who in one way or another shared their support, either morally, financially, or physically, thank you. Above all, to the Great Almighty, the author of knowledge and wisdom, for his countless love.

Dhaka
December 2023

ABU SAKIB

ARIFUL ISLAM

FAZLE ELAHI FAHIM

Contents

List of Figures

ABSTRACT

Testing is an integral part of software development with the goal of verifying a system's requirements. One of the most commonly used methods for verifying code is unit testing. If done properly, unit testing can guarantee the intended functionality of a code unit These are the benefits of unit testing Early detection of defects, Improved code quality, and Increased confidence in code. Furthermore, sufficiently tested code provides dependents with a level of trust in the code's abilities. The unit testing process has several flaws; it is difficult to identify which code must be tested because of very huge code or a huge number of functions; and it is challenging to maintain tests pertaining to inconstant systems. Automatic test case generation tools are a common alternative to writing tests manually. However, these tools often focus on high code coverage but produce flaky and unreliable tests. This thesis aims to find some questions and answers (If the developer has the freedom to choose function parameter value how much it would be helpful, In manual testing developer also can choose parameter values. So how much time is reduced if we automate the traditional manual unit test). We have come up with these two questions after the literature study. To solve the first question we have developed a prototype according to the question and conducted a user survey. The result we got is satisfied. As for question 2, we collected 100 real-world Python functions and made a dataset after we conducted the manual test and prototype test for the same Python function, after we got the prototype it was faster compared to the manual test. in order to determine the effectiveness of various code analysis metrics, a more thorough the study would be needed.

Chapter 1

Introduction

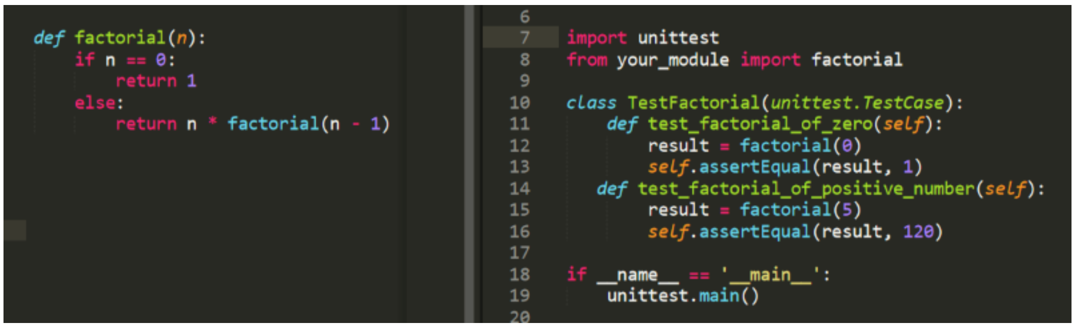
Software testing is an essential method used to validate that a system meets both its functional and non-functional requirements [1]. There are various types of testing methods, which can be broadly classified into manual and automated approaches. Manual testing necessitates testers to execute programs with predetermined test data and assess whether the results align with their expectations. In contrast, automated tests involve the creation of code or scripts by testers or developers, which are typically integrated directly into the ongoing development process [2].

The significance of different testing methods extends to both customers and developers. Testing not only serves to demonstrate that a product can deliver its promised features but also verifies its ability to meet performance, security, and availability requirements. Additionally, testing plays a critical role in identifying defects that could lead to a suboptimal user experience, unpredictable system behavior, or even complete system failures [1]. Some of these defects may have the potential to result in severe security vulnerabilities, affecting both customers and product owners. A prominent example of such a security breach is the Ariane 5 rocket's maiden flight ended in a catastrophic failure just 40 seconds after liftoff. The failure was due to a software error where a 64-bit floating-point number related to the rocket's horizontal velocity was converted to a 16-bit signed integer, causing an overflow and system failure. Adequate unit testing and boundary checks could have identified this issue before launch. And also the Equifax Data Breach, which exposed the sensitive data of millions of Americans and led to Equifax incurring 45.45 million dollars in settlements [3].

Testing is frequently an integral component of software development methodologies like Scrum and Extreme Programming. These methodologies even promote test-driven development (TDD), where unit tests are defined before the production code is written, ensuring a more robust and reliable development process [1].

1.1 UNIT TESTING

Unit testing is a fundamental software testing approach that focuses on verifying the accuracy of individual components or units within a software application. These units are usually small, self-contained entities like functions, methods, or classes, each responsible for specific tasks or functionalities [4]. The sample unit test is shown in Figure 1.1. The primary objective of unit testing is to thoroughly evaluate each unit in isolation to ensure it functions as designed and produces expected outcomes when provided with predefined inputs. This meticulous process aids in detecting and addressing issues at a granular level, resulting in higher software quality and reliability [2].



```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

```python
import unittest
from your_module import factorial

class TestFactorial(unittest.TestCase):
    def test_factorial_of_zero(self):
        result = factorial(0)
        self.assertEqual(result, 1)
   def test_factorial_of_positive_number(self):
        result = factorial(5)
        self.assertEqual(result, 120)

if __name__ == '__main__':
    unittest.main()
```

Figure 1.1: Unit test

Figure 1.2 shows that Unit tests can become difficult to maintain and time-consuming to run as the code under test becomes more complex.

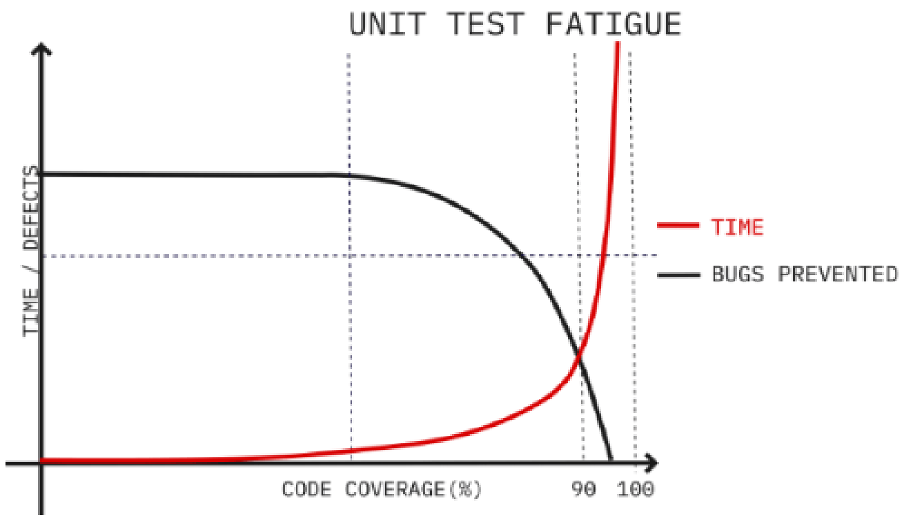


Figure 1.2: The price of software: Unit testing
[5]

This is because there are more possible paths through the code and more interactions between different parts of the code. This makes it more difficult to write unit tests that cover all of the important paths and to isolate the parts of the code that are being tested from the rest of the code.

As a result, developers may be tempted to skip unit tests or to write unit tests that are not as thorough as they should be. This can lead to an increase in the number of bugs in the code. There are a number of things that can be done to mitigate this problem, such as refactoring unit tests regularly, using test doubles, and using continuous integration. However, it is important to be aware of the potential for unit test fatigue and to take steps to avoid it.

1.2 Types of Unit Test

Unit testing offers several advantages in software development Unit tests help detect and rectify issues in the early development stages, reducing the cost and complexity of fixes. Writing unit tests encourages cleaner, more modular, and maintainable code, enhancing overall code quality. These tests act as a safeguard against regressions, ensuring that new code changes do not introduce new defects into existing functionality. Unit tests serve as living documentation, exemplifying how code should be utilized and its expected behavior. Unit tests promote collaboration among team members, fostering a shared understanding of expected behavior and code functionality [6].

- **Function/Method Testing:** In this type of unit testing, individual functions or methods are tested in isolation. Developers create tests that focus on the specific behavior of these functions, providing inputs and checking outputs.
- **Class Testing:** Unit tests can also be applied to test individual classes. In object-oriented programming, this involves testing the behavior of a class, its methods, and interactions within the class.
- **White-Box Testing:** White-box testing involves testing a unit with knowledge of its internal code and logic. Test cases are created to verify the behavior of the code paths, control structures, and edge cases.
- **Black-Box Testing:** In contrast to white-box testing, black-box testing focuses on testing a unit's behavior from the perspective of inputs and expected outputs. Test cases are created without knowledge of the unit's internal code.
- **Random testing:** Often referred to as ”fuzz testing,” is a software testing technique that involves feeding a program or system with random or unexpected inputs to discover vulnerabilities, defects, or unexpected behaviors. Random testing is particularly useful for finding edge cases and unanticipated issues that may not be caught by traditional testing methods.

1.3 Motivation

Developers sometimes show reluctance towards unit testing for several reasons. They may perceive it as time-consuming, feel overwhelmed by complex codebases, prioritize short-term goals, or be skeptical about its benefits. However, embracing unit testing is crucial. It brings significant advantages such as early issue detection, code quality enhancement, regression prevention, and living documentation. It supports continuous integration, fosters collaboration, and reduces technical debt. Overcoming initial reservations and investing in unit testing ultimately leads to higher-quality, reliable software, benefiting both developers and end-users alike. The improvement of unit tests to automatic unit tests could be helpful to developers because of its less time-consuming and instant feedback. Because of this developers will have confidence in their code after being tested.

1.4 Objective

Increased Code Quality: The overall code quality is enhanced. Reduction of Debugging Effort: It reduces the amount of effort required for debugging and troubleshooting at later stages of development or in production. Quality Assurance: Unit testing is an essential component of the overall quality assurance process. It ensures that each unit functions as expected before integration into the larger system. Faster Development and Release: Although writing unit tests may seem time-consuming initially, it often results in faster development and release cycles. Verify Correctness: Unit testing aims to verify the correctness of each unit for unexpected behaviors in the code.

1.5 Challenges

Developing and implementing automated unit tests can be highly beneficial, but it also comes with its set of challenges.

- Lack of Expertise: Developing effective unit tests requires expertise in testing frameworks, best practices, and the specific technology stack being used. we lack experience in this area, there can be a steep learning curve.
- Time and Resource Constraints: Developing comprehensive unit tests can be time-consuming, and many times we have to change features in order to build automatic unit tests quickly. Because of a lack of Resources, we have to test multiple languages which language would be best for building automatic unit test

- User Study: After developing the prototype we faced some problems with testing our prototype. we couldn't find any student who would try out the prototype because many people are not associated with unit tests.

Addressing these challenges requires a commitment to investing in testing skills, the adoption of best practices, and a well-defined testing strategy. Overcoming these obstacles. The study will lead to the benefits of automated unit testing, including improved code quality, early issue detection, and more reliable software.

1.6 Contribution

Developer Control: Allows developers to choose function parameter values for testing.
Coverage Improvement: Reduces gaps in test coverage for a more thorough examination.
Time Savings: Automation significantly saves time compared to manual testing.
User-Friendly: The Prototype provides a user-friendly interface for efficient testing.
Accuracy Boost: Improves the accuracy of unit tests, increasing developer confidence.
Comparison with Randoop: Prototype surpasses tools like Randoop in practical benefits.

1.7 Book Organization

In chapter 1, we introduce the thesis topic, defines unit test, discusses different types of unit test, presents the motivation behind the research, and highlights the challenges faced. In Chapter 2, a review of existing literature on unit test analysis and related results and limitations is conducted. In Chapter 3 focuses on the methods used during the research process. Chapter 4.1 presents the code analysis process used in the application and the method used to generate tests. In Chapter 5 covers the design of the applications that were developed. In Chapter 6 presents the result analysis testing application and results from a user study along with feedback. The final chapter 7 summary and conclusion also mention future works.

Chapter 2

Literature Review

In the vast galaxy of software development, a literature review on unit testing emerges as a guiding star, illuminating the path to software quality assurance. It doesn't merely depict the current state of unit testing practices; rather, it's akin to a diagnostic tool, meticulously examining the effectiveness of various testing methodologies. Much like a lighthouse casting its beam across turbulent waters, it reveals both the brilliant successes and the concealed pitfalls within the realm of unit testing. By conducting a critical evaluation of the strengths and limitations of these approaches, this review empowers researchers and practitioners to make decisions imbued with foresight. It ensures that our expedition through the unit-testing cosmos is brilliantly illuminated with the radiant light of knowledge and the seasoned wisdom of experience. This luminous journey guides us toward the adoption of more efficient testing practices, fortifying our quest for increased software reliability, and ultimately, achieving the zenith of software quality. In essence, this literature review acts as the master key that unlocks doors to enhanced testing practices and stands as the sentinel for the delivery of not just software but trustworthy, impeccable software [7].

2.1 Unit Testing Effectiveness: Insights from Research and Study

The Authors W. Eric Wong and Vidroha Debroy's (2022) [8] survey on Software systems with a large number of functions are often complex and difficult to debug. This is because it can be difficult to identify the specific functions that are responsible for a given fault. Additionally, fault localization can be a time-consuming process, and the time and effort required to localize a fault can increase significantly as the number of functions in the system increases. This paper surveys the state of the art in fault localization techniques for software systems with a large number of functions. The authors identify the challenges of fault localization in these systems, and they review a number of techniques that have been proposed to address these challenges. The authors conclude the paper by discussing a number of future research directions in the area of fault localization for software systems with a large number of functions. One promising direction is the development of new fault localization techniques that are more scalable and effective for large systems. Another promising direction is the development of tools and frameworks that can help testers write and maintain fault unit tests for large systems.

The Authors Alon et al. (2018) [9] conduct a comprehensive empirical study of the effectiveness of automatic unit test generation tools. In their study, they applied three state-of-the-art unit test generation tools for Java (Randoop, EvoSuite, and Agitar) to the 357 real faults in the Defects4J dataset. They then investigated how well the generated test suites performed at detecting these faults. Their findings show that automatically generated test suites can be effective in finding real faults. Overall, the automatically generated test suites detected 55.7 percent of the faults. However, the authors also found that only 19.9 percent of all the individual test suites detected a fault. This suggests that automatic unit test generation tools are not yet perfect, but they can be valuable tools for finding real faults in software. The authors also identified a number of challenges that need to be addressed in order to improve the effectiveness of automatic unit test generation tools. These challenges include Code coverage, Fault propagation, and Test sensitivity.

The Authors Garousi et. al (2019) [10] present a systematic literature review of literature reviews in software testing. The review identified 150 secondary studies, of which 11 were focused on unit testing. The results of the review show that unit testing is a well-studied topic, but there is still room for research in areas such as unit test design, selection, and maintenance.

2.2 Unit Testing Effectiveness: Insights from Development

The Authors Runeson and Per's (2006) [11] literature on unit testing in software development, as revealed by a comprehensive survey, highlights the evolving landscape of developers' practices and challenges. The survey delves into the distribution of time among various development activities, indicating that a significant portion is allocated to writing new code, debugging, and refactoring. Notably, developers spend a substantial amount of time on activities other than coding, with writing tests and handling failing tests comprising a considerable portion. The study emphasizes the perceived importance of writing realistic tests and the challenges associated with determining what to test and addressing flaky tests. Automated unit test generation emerges as a notable practice, with developers expressing a willingness to use such tools frequently. The findings underscore the need for improvements in unit testing, particularly in aiding developers in decision-making for test creation, ensuring test realism, and facilitating maintenance. Developers' perspectives on the difficulty of maintaining tests and the desire for more tool support suggest opportunities for enhancing the overall efficiency and effectiveness of unit testing in software development.

The Authors Daniel Graziotin et. al (2013) [12] investigate the effectiveness of unit tests in test-driven development (TDD). TDD is a software development practice in which developers write unit tests before they write the code itself. The results of the study show that unit tests in TDD are effective at detecting bugs, but they can be time-consuming to write and maintain.

The Authors Jonathan Bergqvist, and Marten Bj Orkman (2022) [13] present a prototype for a unit testing application tailored for JavaScript, aiming to streamline unit test creation and maintenance. It addresses challenges in unit testing, including difficulties in code selection and maintaining evolving systems. Automatic test case generation tools, while faster, can yield unreliable tests. The research question explored is how to create a testing application for this purpose. The user study reveals that participants, particularly those with limited unit testing experience, found the application user-friendly. However, they suggested enhancements like displaying dependent function names and running generated tests. The paper concludes by emphasizing the need for more comprehensive testing applications and open-source collaboration to address these challenges.

Chapter 3

Methodology

This chapter covers the research process and methodologies used in order to answer the research question. The chapter starts by showing how we approach the research question to find out the answer.

3.1 Research Focus

To create more tangible goals from a literature study we have developed some research questions. it can be split into several sub-questions. The answers to these questions have been attempted together to find out the answer. The workflow of the research process is shown in Figure. 3.1.

Research question 1.

If the developer has the freedom to choose a function parameter value how much it would be helpful?

Research question 2.

In manual testing developer also can choose parameter values. so How much time is reduced if we automate the traditional manual unit test?

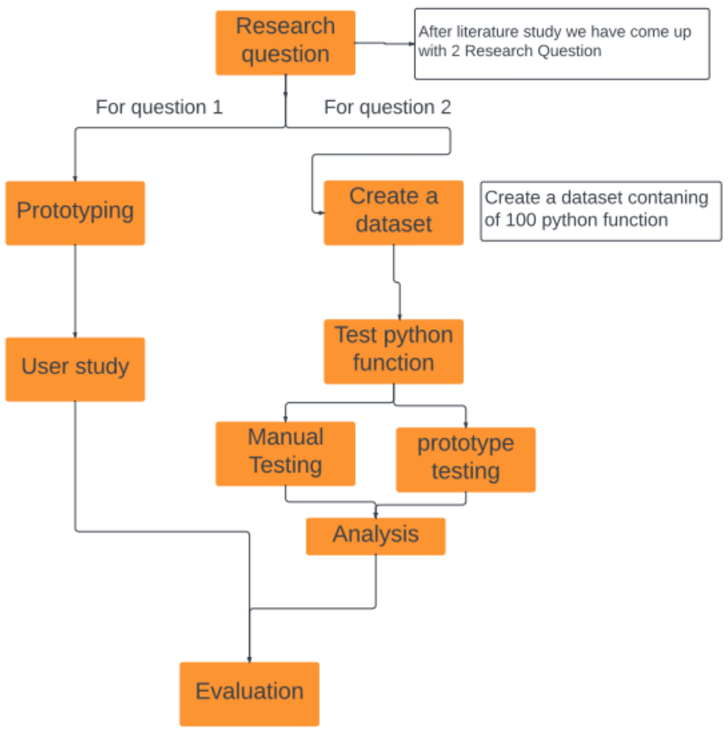


Figure 3.1: Workflow of the research process

3.2 Research Process

3.2.1 Prototyping

Planning

Platform development started by defining a requirements specification document. The requirements added to this document were organized and prioritized using the MoSCoW method(Must-Have, Should-Have, Could-Have, Won't-Have). In the document, the overall functional requirements for the application's back- and front-end were specified. These requirements were solidified by breaking down what is necessary to achieve the previously mentioned goals.

The application was then planned and designed further by using the top-down approach in that the front end, meaning the part of the application that the end user would interact with, was designed first. Front-end of the design was created with the web-based diagramming application Lucidchart. The necessary communication between the front-end and back-end was realized using the same diagramming application via Notepad. The final step of the application planning was to create a suggested file structure for the whole project, both the front end and the back end.

Development

The application was developed by using the bottom-up approach in which the back end and its least functionally dependent components were developed first. After having the required parameters provided by the back end, the front end was constructed. Development and collaboration were orchestrated via the services provided by TeamViewer.while also allowing development to take place in parallel. During daily meetings at the start of the day, there would be discussions of what had been done the day before and what was planned for the day.

3.2.2 User Study

The next stage in the research process focused on conducting a qualitative user study. This study consisted of multiple interviews in which the interviewee got to use the testing application and answer questions related to their experience. Interviews were chosen instead of surveys because time was limited and to hopefully get more insightful answers from each respondent. In order to find people for the study, requests were sent to students and developers asking if they would like to partake in an interview. These requests were sent a week prior to when the interviews would start to take place. In preparation for the interviews, we told them they could test any function they wanted there were no limitations to choosing any functions, and the interviewee was to create tests.

3.2.3 Dataset creation

We have collected 100 real-world Python functions from online to test out the prototype to see how much time it would reduce compared to a manual test.

3.2.4 Evaluation

To wrap up the research process, the evaluation phase attempts to answer the research questions. The first question is answered by creating the prototype and conducting a user study. Next, the metric is evaluated during the user study and the results provide an answer to the question. The second question is answered by creating and analyzing a data set con- training 100 Python functions. Moreover, the research process and prototypes are evaluated to find areas of improvement.

3.3 Development Tools and Services

The following list specifies the different tools and services used during the prototyping phase.
Lucidchart Online tool for creating graphs and flow charts was used to design the API calls to the server and application interface.
PyCharm IDE for Python applications that were used to develop the back-end of the testing application.
WebStorm IDE for web applications that were used to develop the front-end of the testing and demo application.
Git Version control program used to track project files.
GitHub Platform for hosting files that are version-controlled with Git.

Chapter 4

Code Analysis and Test Generation

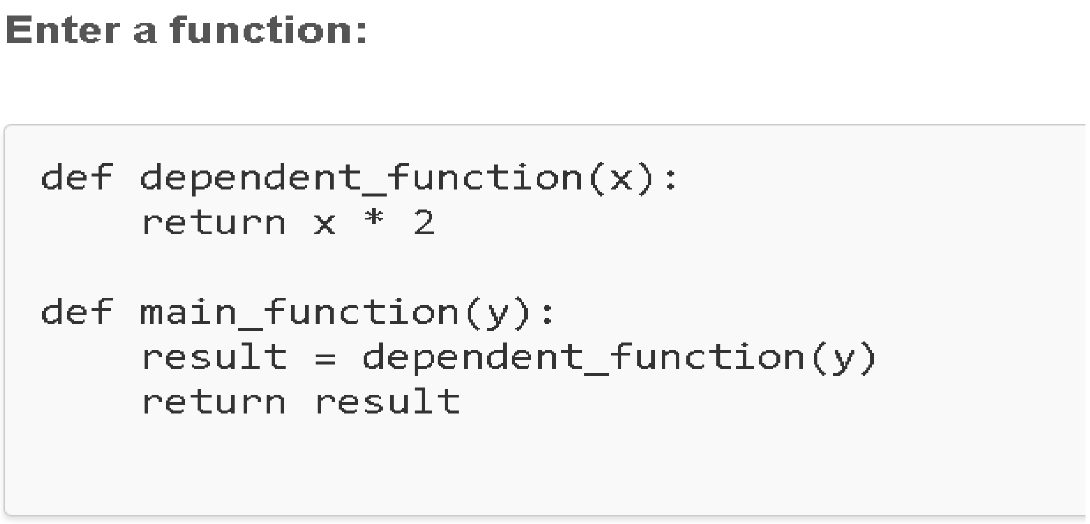
4.1 Code Analysis and Test Generation

This chapter covers the units of code the prototype can handle, the analysis process of a Python prototype, and how tests created by the user become automatic tests.

4.2 Determining What Code to Unit Test

For testing the prototype it can handle complex code, and testers don't need to upload any file to test their function they can just put their code to test. The only thing that matters is the code has to be in the function format. As a result, the tool would instead help a tester to test their function and show the result whether the function has passed or not. In this prototype, to test a function that has a dependence function there is no order that needs to be maintained. Developer code can be in any order whether the dependence function is above or below still our prototype can test the function. However, the aim was to test the developer function.

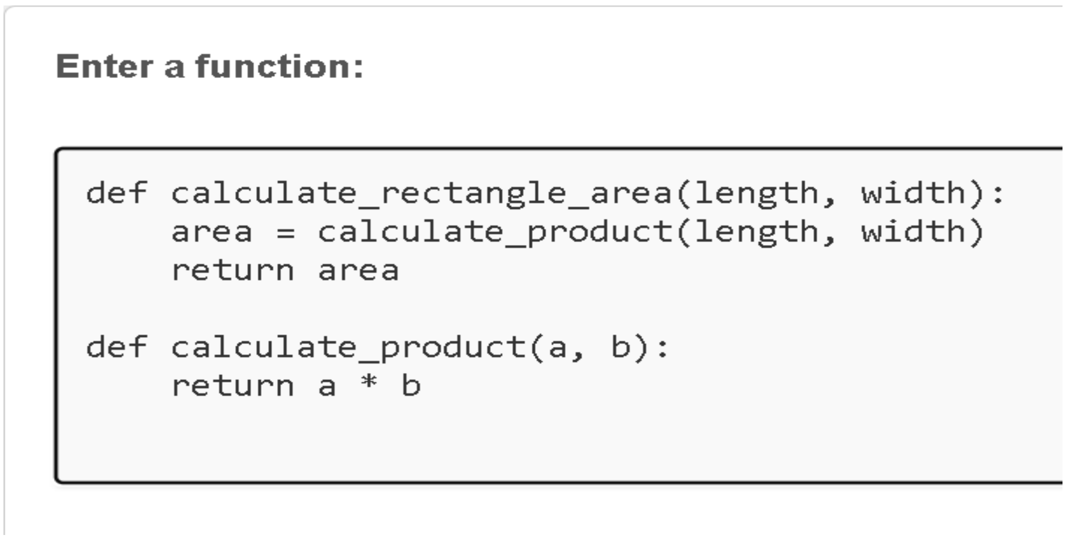
In Figure 4.1 for an example in Python of these metrics, where the function `dependent_function()` has one dependent function and the function `main_function()` depends on one function. The test will be passed even if the dependent function is above the main function.



```
def dependent_function(x):
    return x * 2

def main_function(y):
    result = dependent_function(y)
    return result
```

Figure 4.1: Example of a function with one dependent (above) and a function with one dependency (below)

In Figure 4.2 for an example in Python of these metrics, where the function `calculate_product(a,b)` has one dependent function and the function `calculate_rectangle_area(a,b)` depends on one function. The test will be passed even if the dependent function is below the `calculate_rectangle_area(a,b)`.



```
def calculate_rectangle_area(length, width):
    area = calculate_product(length, width)
    return area

def calculate_product(a, b):
    return a * b
```

Figure 4.2: Example of a function with one dependent (above) and a function with one dependency (below)

4.3 Defining Code Units to Test in our prototype

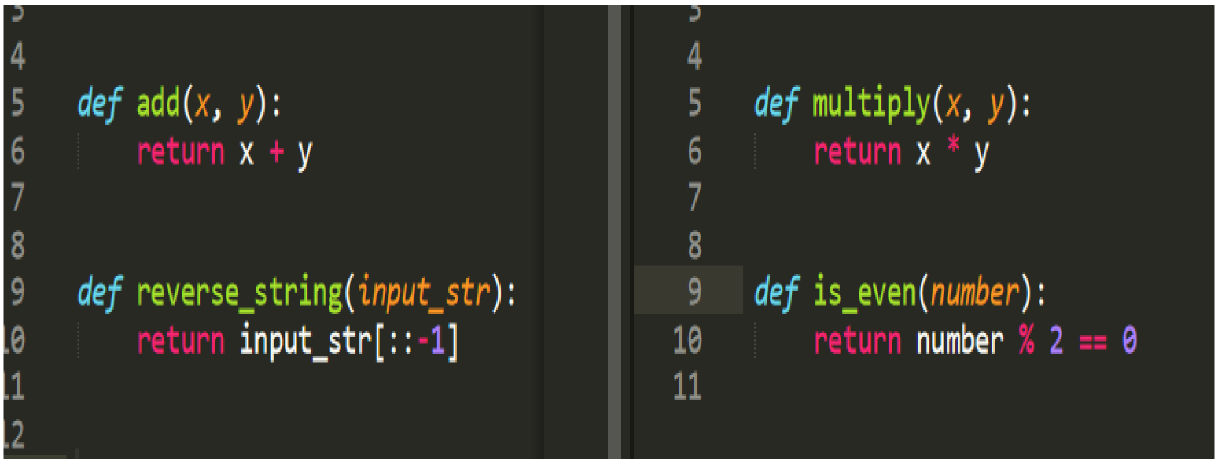
In order to determine code units to test in a Python prototype. Developers can submit Python code for testing various types of functions. These functions can encompass a wide range of functionalities, serving as a versatile tool for code validation and experimentation.

Developers commonly submit arithmetic functions, allowing them to test operations like addition, subtraction, multiplication, and division. String manipulation functions are also prevalent, enabling testers to evaluate code for tasks such as string reversal or substring identification. Logical functions are another category, allowing testers to test logical operations or conditions, such as checking for even numbers.

Additionally, developers can submit functions designed for data processing. These functions operate on lists and enable tasks like filtering data. testers often customize and submit their user-defined functions, the complexity and purpose of which can vary widely. These custom functions are executed and tested based on user-specified test cases, providing a means for testers to validate the behavior and correctness of their code.

The Flask app offers a text input area for testers to provide their Python function code. The application then executes these functions and assesses their behavior against the defined criteria. This dynamic environment enables testers to interactively test and refine their code. It serves as a valuable resource for both beginners and experienced developers, fostering code testing, learning, and experimentation.

In Figure 4.3 we can see that simple functions can be tested like addition, multiplication, etc.

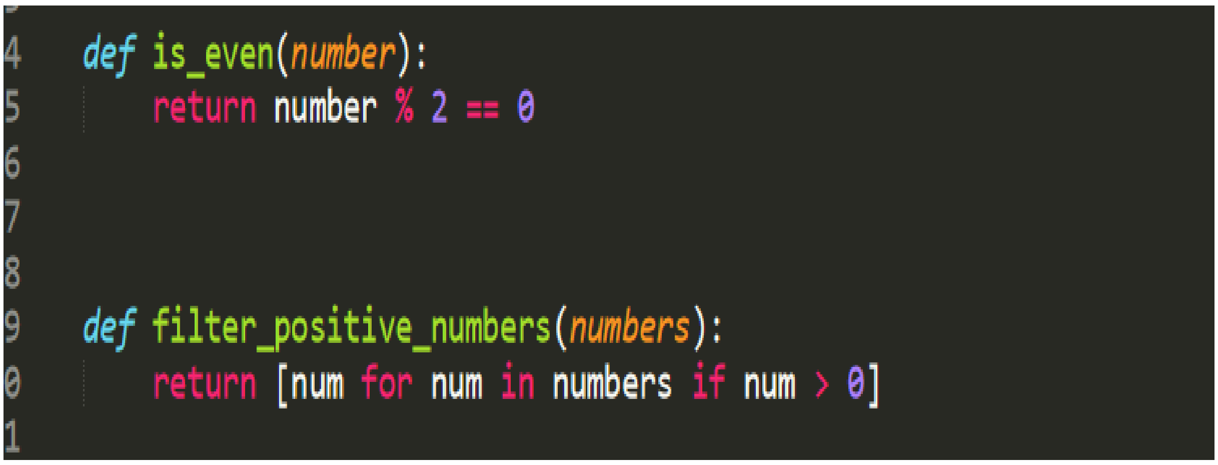


```python
def add(x, y):
    return x + y


def reverse_string(input_str):
    return input_str[::-1]
```

```python
def multiply(x, y):
    return x * y


def is_even(number):
    return number % 2 == 0
```

Figure 4.3: Example 1 simple function can be tested)

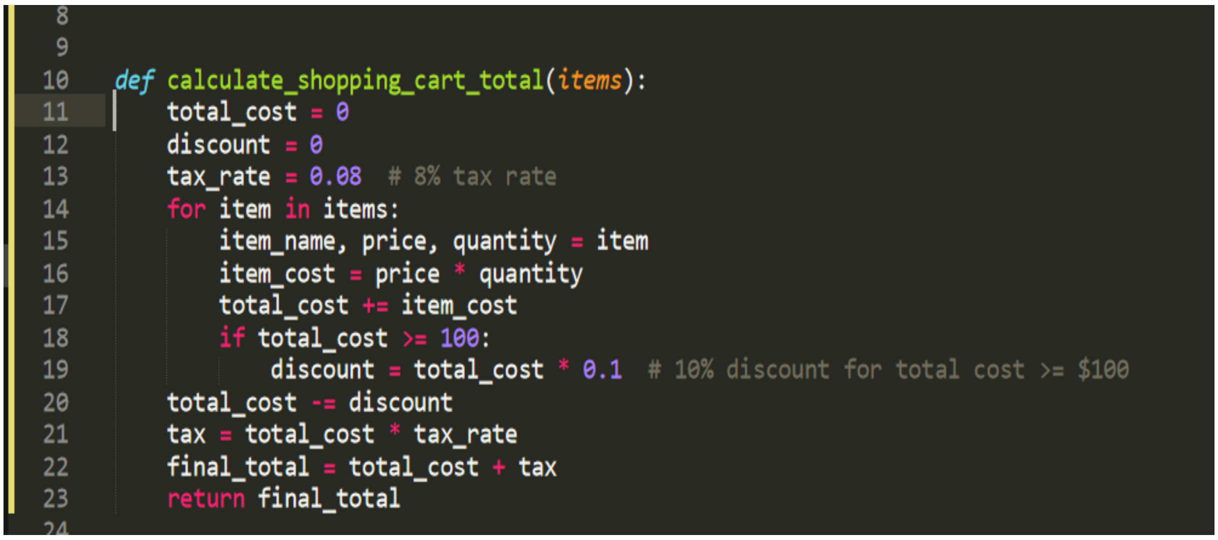
In Figure 4.4 we can see that functions that will return Boolean value can be tested.



```python
def is_even(number):
    return number % 2 == 0



def filter_positive_numbers(numbers):
    return [num for num in numbers if num > 0]

```

Figure 4.4: Example 2 return Boolean value and list value can be tested

In Figure 4.5 we can see that functions that have some complexity can be tested.



```python


def calculate_shopping_cart_total(items):
    total_cost = 0
    discount = 0
    tax_rate = 0.08  # 8% tax rate
    for item in items:
        item_name, price, quantity = item
        item_cost = price * quantity
        total_cost += item_cost
        if total_cost >= 100:
            discount = total_cost * 0.1  # 10% discount for total cost >= $100
    total_cost -= discount
    tax = total_cost * tax_rate
    final_total = total_cost + tax
    return final_total
```

Figure 4.5: Example 3 complex code can be tested

4.4 Analysis of a Python prototype

Our prototype constitutes a web application built using Flask to facilitate the testing and automatic generation of unit tests for Python functions. This application encompasses various components, including Flask routes, functions, HTML templates, CSS styles, and JavaScript. At its core, this Python code leverages Flask, a popular web framework, to create a web application that serves as a testing platform for Python functions. Flask is a micro web framework that is known for its simplicity and flexibility. The application is structured into distinct modules that collectively offer a comprehensive solution for function testing. The web application primarily consists of two routes: one for testing functions and generating unit tests, and another for property-based random testing. These routes handle HTTP requests and render HTML templates. The core functionality of generating unit tests involves parsing user-provided Python functions. The application takes the user's Python code and extracts the function name, return type, parameter count, and expected result. It then generates unit test code for the provided function, including test cases with user-defined arguments. This generated test code is useful for ensuring the correctness of the user's function. Property-based random testing is another crucial aspect of this application. Users can input Python functions, specify the number of random tests, and define the data type for expected results. The application then runs these functions with randomly generated arguments and compares the actual results to assumed results within a specified range. This process assesses the robustness of the functions under different conditions and is a valuable testing technique. The application also employs HTML templates for the user interface. These templates are styled using CSS for a clean and user-friendly appearance. JavaScript is integrated to enhance the user experience by enabling the copying of function code and resetting form fields. To initiate the web application, the code includes a conditional check that runs the Flask app when executed directly. This allows users to access the functionality through a local web server.

4.5 Analysis of how the prototype test tester code

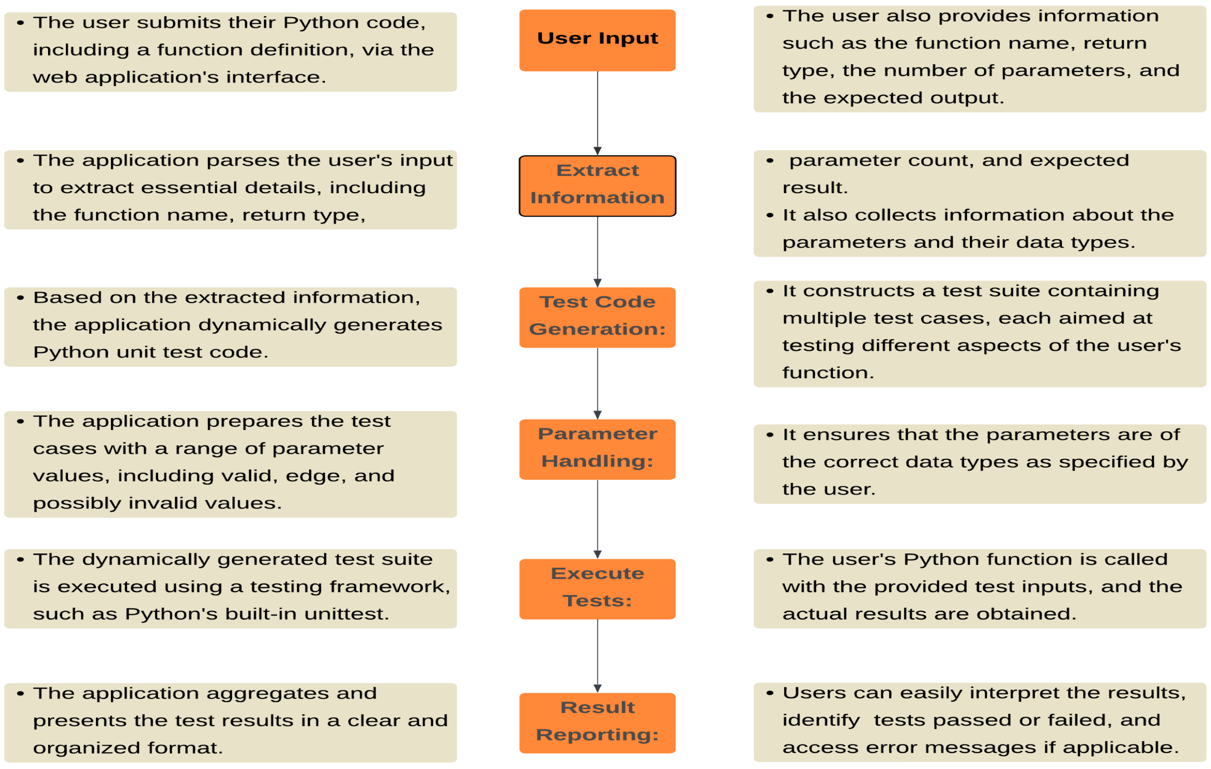
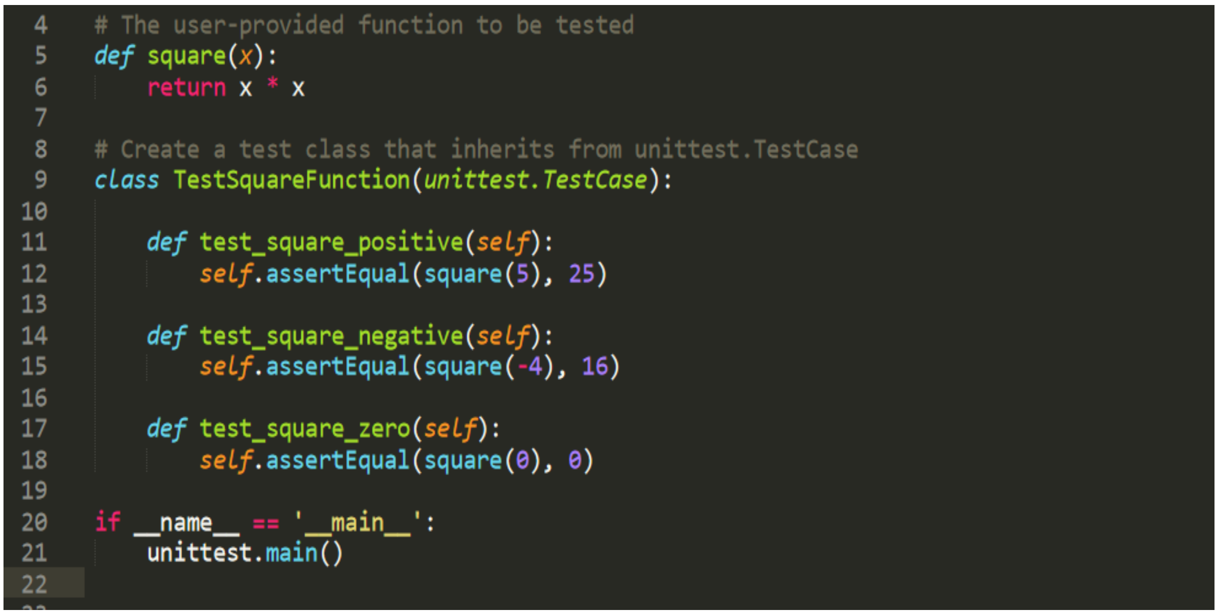

• The user submits their Python code, including a function definition, via the web application's interface.
User Input
• The user also provides information such as the function name, return type, the number of parameters, and the expected output.
• The application parses the user's input to extract essential details, including the function name, return type,
Extract Information
• parameter count, and expected result.
• It also collects information about the parameters and their data types.
• Based on the extracted information, the application dynamically generates Python unit test code.
Test Code Generation:
• It constructs a test suite containing multiple test cases, each aimed at testing different aspects of the user's function.
• The application prepares the test cases with a range of parameter values, including valid, edge, and possibly invalid values.
Parameter Handling:
• It ensures that the parameters are of the correct data types as specified by the user.
• The dynamically generated test suite is executed using a testing framework, such as Python's built-in unittest.
Execute Tests:
• The user's Python function is called with the provided test inputs, and the actual results are obtained.
• The application aggregates and presents the test results in a clear and organized format.
Result Reporting:
• Users can easily interpret the results, identify tests passed or failed, and access error messages if applicable.

Figure 4.6: Flow chart of how prototype work

When a user provides their Python code and associated parameters for testing, the provided Python application analyzes and processes the input to generate unit tests and display the results effectively. The flowchart of the protype is given in Figure. 4.6. First, the user submits their Python code, including a function definition, to the web application. The application then extracts essential information from this input, including the function name, return type, the number of parameters, and the expected output. Next, the application utilizes this extracted information to dynamically generate unit test code. It constructs a test suite, complete with test cases that include the function's parameters. The test cases are designed to cover various scenarios, ensuring thorough testing of the user's code. The test code generation includes handling different data types based on the specified return type (e.g., int, float, bool, str, list). It also caters to functions that may not return a value (specified as ”Null”). After generating the test code, the application executes it, running the user-provided Python function with the input parameters. It captures both the actual result and the expected result. The results are then compared. If the actual result matches the expected result, the test is marked as ”Passed.” If not, it's marked as ”Failed.” This comparison ensures that the user's function behaves as intended. The test results are presented to the user in a clear and readable format. This feedback includes the details of each test case, whether it passed or failed, and any error messages or issues encountered during the testing process. The user is provided with immediate feedback, allowing them to quickly assess the correctness of their function. If a test case fails, they can identify the specific input and expected output that caused the failure, aiding in debugging and code improvement. The application efficiently processes user-provided Python code and parameters by dynamically generating unit tests, executing these tests, and presenting the results in a user-friendly manner. This feedback loop empowers the user to verify the functionality of their code, identify issues, and make necessary improvements to enhance the quality and reliability of their Python functions.

in Figure 4.7 our prototype got a function from the user and the prototype wrote the class method dynamically to test that function. and it's applicable to almost all function



```python
# The user-provided function to be tested
def square(x):
    return x * x

# Create a test class that inherits from unittest.TestCase
class TestSquareFunction(unittest.TestCase):

    def test_square_positive(self):
        self.assertEqual(square(5), 25)

    def test_square_negative(self):
        self.assertEqual(square(-4), 16)

    def test_square_zero(self):
        self.assertEqual(square(0), 0)

if __name__ == '__main__':
    unittest.main()
```

Figure 4.7: Example of how our prototype tests a function

Chapter 5

Design of the Prototype and Demo Application

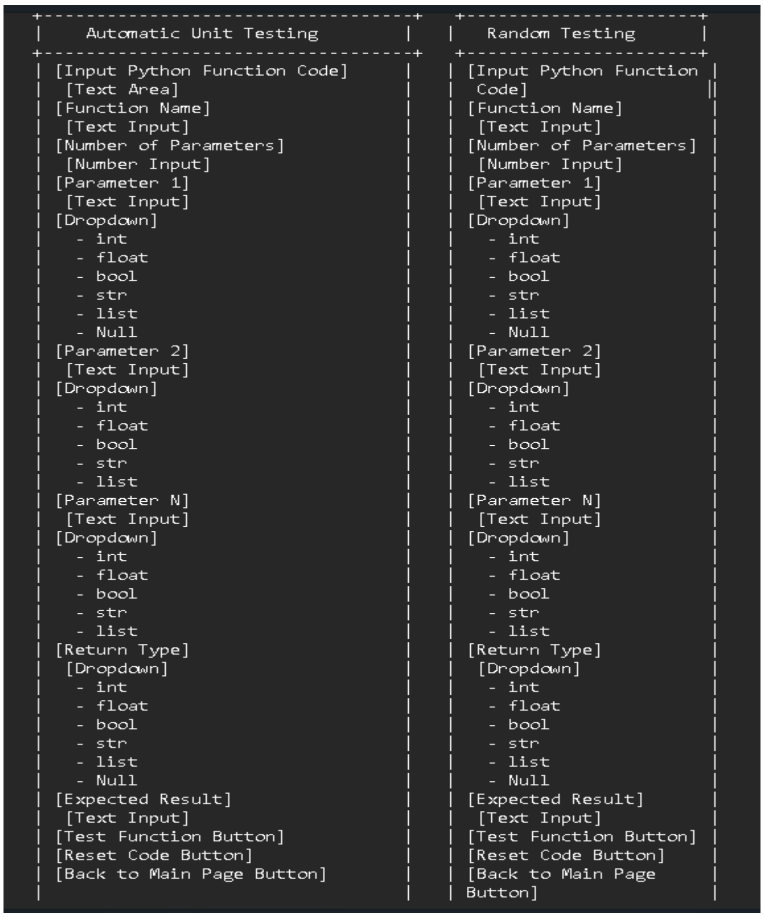
In this chapter, we will delve into the prototype design and the demonstration application that was instrumental in conducting the user study. The chapter includes an overview of the key components and interactions, taking inspiration from the system architecture diagram.

5.1 Testing Application Prototype

The testing application was designed as a web application that would run locally on the client's computer. Figure 5.1 gives an overview of how the application was structured. The following sections will go into more detail on how the front end and the back end was designed.

5.1.1 Front-End

The web application is designed for software testing, consisting of two key components: "Automatic Unit Testing" and "Random Testing." Users can input Python function code and specify testing parameters, and each component offers a distinct approach to testing. In the "Automatic Unit Testing" section, users can input Python function code and provide details such as the function name, the number of parameters, and parameter-specific information like name and data type. Additionally, users can specify the expected return type and the desired outcome. This information empowers the application to generate unit tests automatically. To initiate the testing process, users can click the "Test Function Button." If they want to start afresh, the "Reset Code Button" allows them to clear the input code. In case they need to return to the main page, a dedicated "Back to Main Page Button" is available. Conversely, the "Random Testing" section enables users to input Python function code similarly, including the function name and parameter details. However, instead of return type and expected outcome, users can define the number of random tests to execute and set constraints on the input values, such as minimum and maximum values and data types. The ”Run Random Tests” button initiates the random testing process, while the ”Reset Code” button clears the code input. The ”Back to Main Page Button” facilitates easy navigation back to the main application page. In Figure 5.1 contains all the components of our prototype front end



```
+-----------------------------------+   +----------------------+
|    Automatic Unit Testing         |   |   Random Testing     |
+-----------------------------------+   +----------------------+
| [Input Python Function Code]      |   | [Input Python Function |
|  [Text Area]                      |   |  Code]               ||
| [Function Name]                   |   | [Function Name]       |
|  [Text Input]                     |   |  [Text Input]         |
| [Number of Parameters]            |   | [Number of Parameters] |
|  [Number Input]                   |   |  [Number Input]       |
| [Parameter 1]                     |   | [Parameter 1]         |
|  [Text Input]                     |   |  [Text Input]         |
| [Dropdown]                        |   | [Dropdown]            |
|   - int                           |   |   - int               |
|   - float                         |   |   - float             |
|   - bool                          |   |   - bool              |
|   - str                           |   |   - str               |
|   - list                          |   |   - list              |
|   - Null                          |   |   - Null              |
| [Parameter 2]                     |   | [Parameter 2]         |
|  [Text Input]                     |   |  [Text Input]         |
| [Dropdown]                        |   | [Dropdown]            |
|   - int                           |   |   - int               |
|   - float                         |   |   - float             |
|   - bool                          |   |   - bool              |
|   - str                           |   |   - str               |
|   - list                          |   |   - list              |
| [Parameter N]                     |   | [Parameter N]         |
|  [Text Input]                     |   |  [Text Input]         |
| [Dropdown]                        |   | [Dropdown]            |
|   - int                           |   |   - int               |
|   - float                         |   |   - float             |
|   - bool                          |   |   - bool              |
|   - str                           |   |   - str               |
|   - list                          |   |   - list              |
| [Return Type]                     |   | [Return Type]         |
|  [Dropdown]                       |   |  [Dropdown]           |
|   - int                           |   |   - int               |
|   - float                         |   |   - float             |
|   - bool                          |   |   - bool              |
|   - str                           |   |   - str               |
|   - list                          |   |   - list              |
|   - Null                          |   |   - Null              |
| [Expected Result]                 |   | [Expected Result]     |
|  [Text Input]                     |   |  [Text Input]         |
| [Test Function Button]            |   | [Test Function Button] |
| [Reset Code Button]               |   | [Reset Code Button]   |
| [Back to Main Page Button]        |   | [Back to Main Page    |
|                                   |   | Button]               |
```

Figure 5.1: Elements of front end in prototype

These pages were: written in HTML along with JavaScript to handle dynamic input parameters; and styled with the help of the styling library Bootstrap and the CSS; in order to serve it for the back end.

5.1.2 Back-End

The back end of the testing application is mainly responsible for analyzing the user’s project code generating tests and determining whether the test has been passed or not. From Figure 5.1, it can be seen that the back end gets all the user information through the flask request method Furthermore, the server application uses a unit test module to test the test cases that are generated through the back end python code.. All components in the back end were written using Python and with the help of several frameworks and libraries. These include the Flask Web framework, which was used to create the application; the JavaScript for dynamic websites; Flask-route to establish communication with the pages through route; Lastly, the web server hosting the application was the built-in development server for Flask.

5.1.3 Web Application

In figure 5.2 this web application streamlines the testing process, offering both automated and property-based testing approaches. Users can test their functions, and receive comprehensive test results through an intuitive and powerful interface. It empowers developers to ensure the reliability and correctness of their Python functions with ease and efficiency.

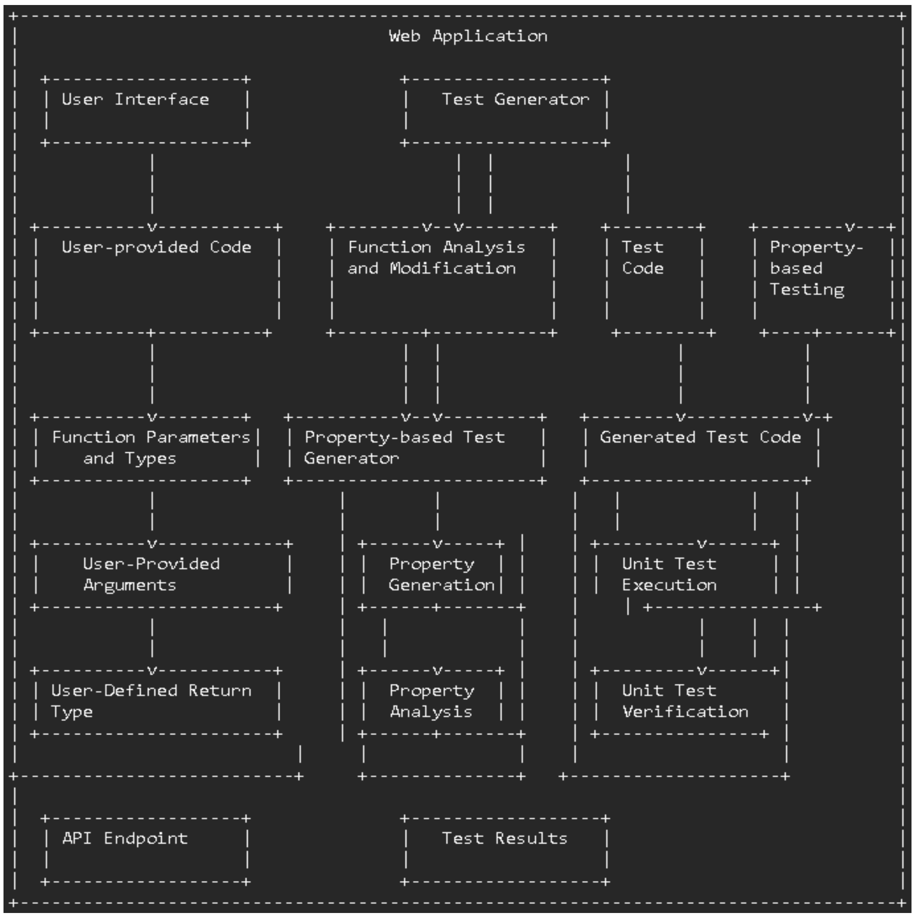


Figure 5.2: Diagram showing the structure of the testing application

Chapter 6

Results analysis

This chapter presents a brief overview of the results of the two questions and the results from the conducted user study.

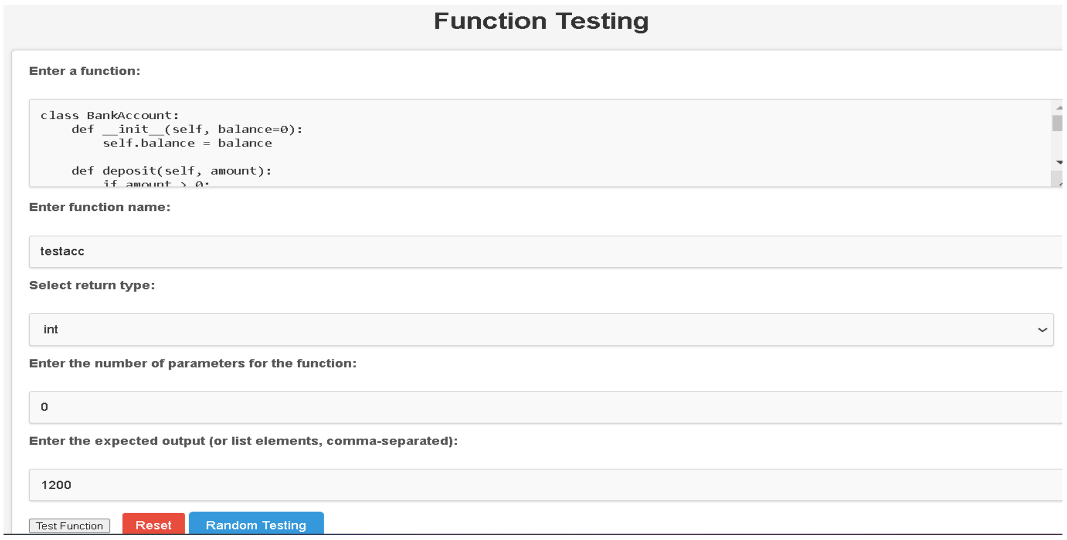
6.1 Answer of our First question

Structure of the Interview

The interviews were each divided into three phases and lasted around 20 minutes. During the interview, the participants were asked about their prior experiences and views on unit testing. The first phase of the interview consisted of explaining the functionality of the application; how to use it and what was possible in this prototype.
During the second phase, the participant was asked to create tests for any function that they wanted. If they had any problems we would explain how to perform various actions, but we would not tell the participants what test cases to verify for a code unit. The participants were asked if they wanted to continue creating tests and were allowed to create as many as they wanted before the third phase.
During the third and final phase of the interview, the participants were asked about their experiences using the application.

Interview Results

Every participant in the user study was a computer science student. Most of the participants had previously written unit tests, and they were all familiar with the idea of white-box unit testing. However, they still had little expertise in unit testing. During the interview's test development stage, a number of participants expressed that, in contrast to standard automatic unit tests, which simply produce such parameter values at random for their functions, they would have preferred to be allowed to enter their chosen input and parameters. One person said that being able to accomplish that would be useful. Consequently, it would be a strong model. The ease of use and self-explanatory nature of the test creation process were acknowledged. Many of the attendees made the analogy that it can be difficult to begin developing unit tests. The conventional approach to writing tests requires familiarity with particular testing frameworks before beginning. A subset of the participants wrote tests for the more complex code units, which call for objects or arrays as input and produce modified versions of these as output. Some had difficulty getting started, but they soon grasped the process of creating these inputs and outputs. A couple mentioned that they would have preferred to be able to create these arrays or objects in a more conventional manner. A subset of the participants wrote tests for the more complex code units, which call for objects or arrays as input and produce modified versions of these as output. Some had difficulty getting started, but they soon grasped the process of creating these inputs and outputs. A couple mentioned that they would have preferred to be able to create these arrays or objects in a more conventional manner. Also, some users want to try random testing for coverage edge cases when they are able to do the random testing participants were satisfied with the result. Several users have noted that, in comparison to automatic unit test tools, testing their function takes a little longer because the latter cannot accept user input parameters. Participants claim that even though it takes a little longer than usual testing tools because users can provide any input they choose, it is far more reliable. In Figure 6.1 one participant is trying to test complex code to test the prototype.



Enter a function:

```
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
```

Enter function name: testacc

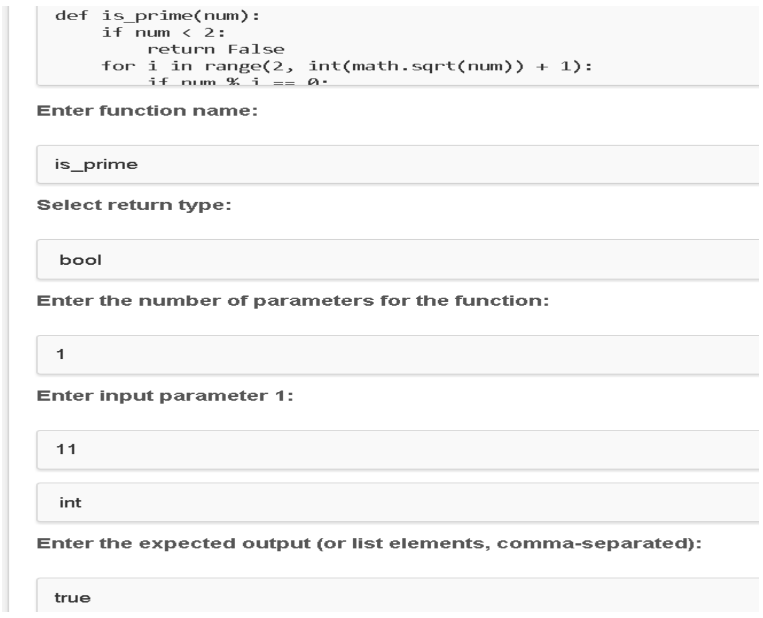
Select return type: int

Enter the number of parameters for the function: 0

Enter the expected output (or list elements, comma-separated): 1200

Figure 6.1: Complex code to test(Bank Account class)

In Figure 6.2 another participant is trying to test a function that has one parameter and also gives the value that the participant wants to try to test



```
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
```

Enter function name:

is_prime

Select return type:

bool

Enter the number of parameters for the function:

1

Enter input parameter 1:

11

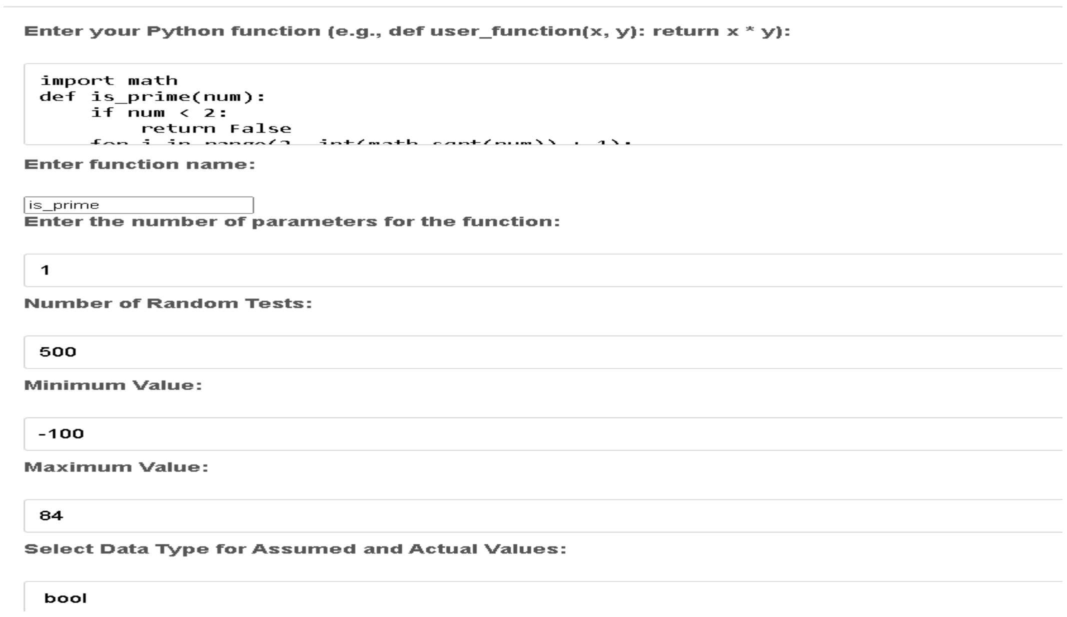
int

Enter the expected output (or list elements, comma-separated):

true

Figure 6.2: simple code with one parameter to test

In Figure 6.3 another participant is trying random testing to test a function to ensure the edge cases.



Enter your Python function (e.g., def user_function(x, y): return x * y):

```
import math
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(math.sqrt(num)) + 1):
```

Enter function name:

is_prime

Enter the number of parameters for the function:

1

Number of Random Tests:

500

Minimum Value:

-100

Maximum Value:

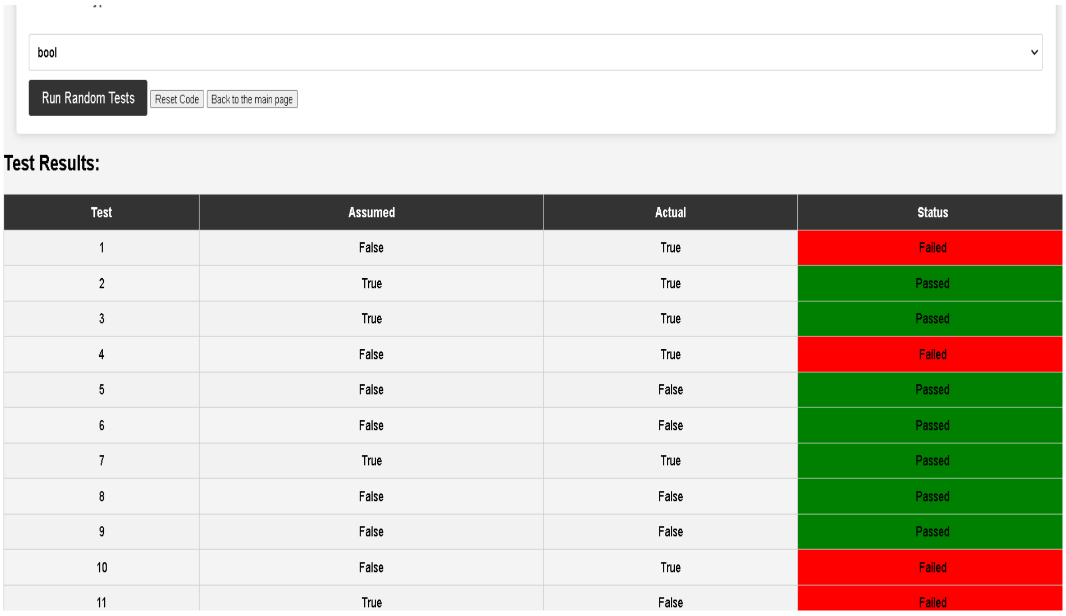
84

Select Data Type for Assumed and Actual Values:

bool

Figure 6.3: random testing to a function.

In Figure 6.4 this is the result of the random testing in Figure 6.3 function. The result consists of 4 columns (test number, assume value, and expected value). If the assumed value matches the expected value the result will be passed if not then failed.



Test Results:

Test	Assumed	Actual	Status
1	False	True	Failed
2	True	True	Passed
3	True	True	Passed
4	False	True	Failed
5	False	False	Passed
6	False	False	Passed
7	True	True	Passed
8	False	False	Passed
9	False	False	Passed
10	False	True	Failed
11	True	False	Failed

Figure 6.4: Resutls of the random test

6.1.1 Participants Feedback

User feedback summarize: Feedback from the interview has provided valuable insights about the prototype for unit testing. Users commend the platform's flexibility, allowing customization of input parameters and expected results, enhancing precision in testing. The support for various data types and a user-friendly interface simplify the testing process, with clear error reporting aiding debugging efforts. While users appreciate the interactive and user-driven testing approach, there is room for improvement, such as expanding data type support and adding advanced testing features. Overall, with ongoing development and user feedback, the prototype holds the potential to become a a powerful tool for unit testing, offering testers effective ways to verify code correctness. In Figure 6.5 the slices represent different levels of agreement with a statement. The largest slice, which is 75 parentage of the pie, represents the percentage of people who strongly agree with the statement. The next largest slice, which is 12.5 parentage of the pie, represents the percentage of Agree people. The remaining 12.5 parentage of the pie is divided between people who disagree with the statement.

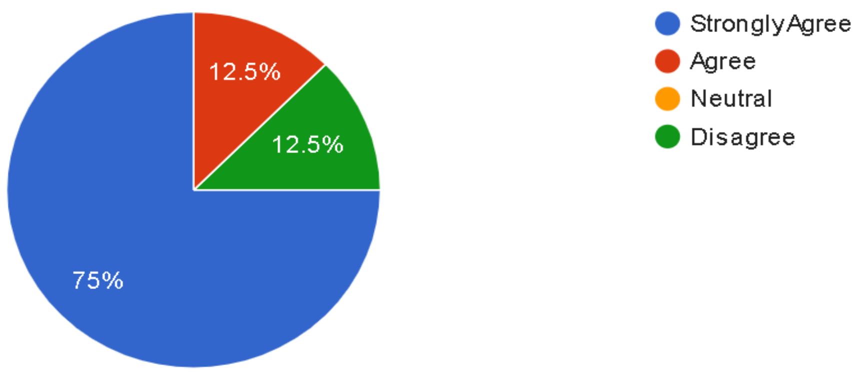


Figure 6.5: user Study results

6.2 Answer of our Second question

we have collected 100 Python functions to test and compare the time that each test and how much time is needed. In Figure 6.6 for each function test it almost took 5 minutes and that's too much time to just test one function. Because of this reason, developers do not want to unit test.

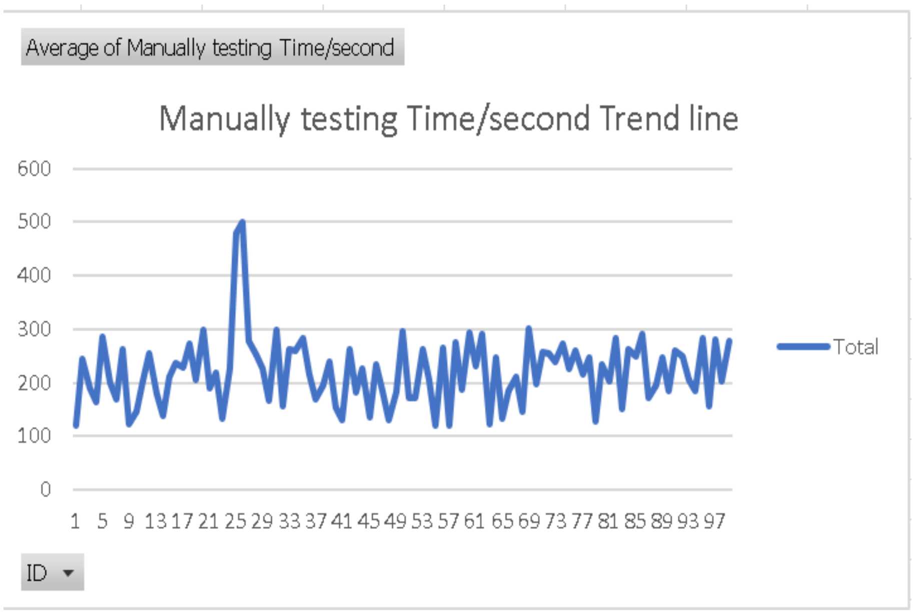


Figure 6.6: Manual testing time trending line

In order to overcome this limitation we have developed a prototype for automating the unit test and it took less time to test one function. In Figure 6.7 for each function test it almost took 5 minutes and that's too much time to just test one function. Because of this reason, developers do not want to unit test.

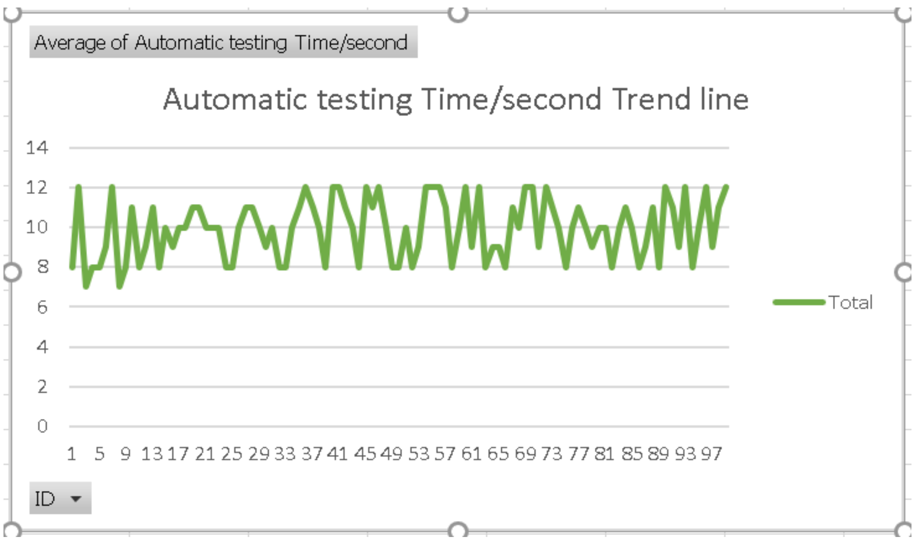


Figure 6.7: Automating testing time trending line

And now we have compared both testing trending lines to see which one is better and also to have a better look. In Figure 6.8 we can see that the the blue trending line is for automatic testing and the yellow one is for manual testing and if we compare both the blue trending line almost took negligible time compared to the yellow line.

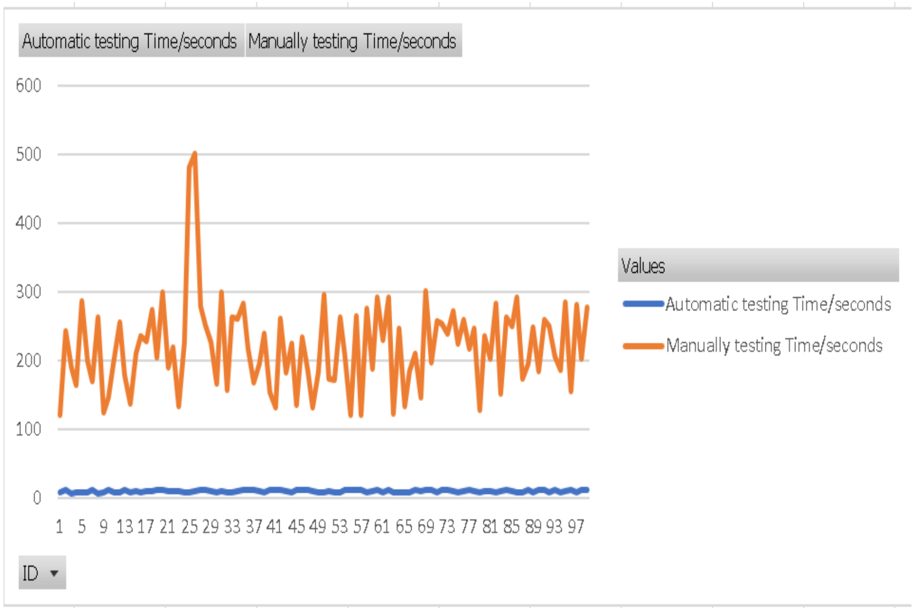


Figure 6.8: Manual vs Automating testing time trending line

6.2.1 Evaluation Method

The interview method could have been improved in many ways, all of which we realized after conducting them. If there had been more time, we would have used the results from the interview in order to conduct another user study. We have read that it is a good idea to have test interviews before the actual interview[53], and we feel this would have improved our study a lot. Our intention was to focus on test creation and maintenance, but we feel there was too little focus on the latter. We had not prepared the user study enough to allow the participants to use the maintenance functionality to its intended extent, thus we could not arrive at any conclusions regarding it.

Chapter 7

Conclusion and Future Work

We embarked on a journey to explore the realm of automatic unit testing in modern software development, focusing on its advantages and potential challenges. Our investigation led us to formulate two key questions: the impact of developer freedom in choosing function parameter values and the time efficiency gained by automating traditional manual unit tests. Through the development of a Python prototype, we provided developers with a flexible environment to test their code with dynamically chosen inputs. The subsequent user study demonstrated a positive response, highlighting the prototype's adaptability, user-friendliness, and potential as a powerful tool for precise unit testing. In addressing the time efficiency aspect, we conducted a comprehensive analysis of manual versus automated unit testing. The results unequivocally showed that manual testing, while essential, is time-consuming, often discouraging developers from conducting thorough unit tests. Conversely, our automated testing prototype showcased a significant reduction in testing time, making unit testing more feasible and attractive for developers. The visual representation of testing time trends vividly illustrated the superiority of automated testing over its manual counterpart. The negligible time required for automated testing, as depicted in the trending lines, emphasized its potential to revolutionize the landscape of unit testing practices. The development of a fully functioning application capable of supporting test generation across various programming languages poses significant challenges. Currently, our prototype is limited to dynamic input, exception expectations, and dynamic output, with the need for expansion to support lists of strings. Efforts to extend the prototype to other languages have proven difficult, leading to attempts to convert user code to Python, but encountering obstacles such as API key pricing and unsuccessful Python library usage. Estimating the time and effort required for a comprehensive prototype in this category is elusive, even for Python alone. Despite the challenges faced, we believe it's not an impossible task. To enhance future studies, the evaluation process would benefit from a more advanced product and a comprehensive user study involving participants with broader development and testing experiences. The prototype should be tested extensively, allowing participants to use it in their existing testing scenarios, potentially yielding more accurate and valuable results.

References

[1] I. Sommerville, *Software Engineering, 9/E.* Pearson Education India, 2011.

[2] M. Olan, "Unit testing: test early, test often," *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, pp. 319–328, 2003.

[3] H. Krasner, "The cost of poor software quality in the us: A 2020 report," *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, pp. 1–46, 2021.

[4] E. G. Barriocanal, M.-Á. S. Urbán, I. A. Cuevas, and P. D. Pérez, "An experience in integrating automated unit testing practices in an introductory programming course," *ACM SIGCSE Bulletin*, vol. 34, no. 4, pp. 125–128, 2002.

[5] F. G. Pascual and W. Q. Meeker, "Estimating fatigue curves with the random fatigue-limit model," *Technometrics*, vol. 41, no. 4, pp. 277–289, 1999.

[6] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[7] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering–a systematic literature review," *Information and software technology*, vol. 51, no. 1, pp. 7–15, 2009.

[8] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[9] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 201–211, IEEE, 2015.

[10] V. Garousi and M. V. Mäntylä, "A systematic literature review of literature reviews in software testing," *Information and Software Technology*, vol. 80, pp. 195–216, 2016.

[11] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[12] A. Tosun, M. Ahmed, B. Turhan, and N. Juristo, “On the effectiveness of unit tests in test-driven development,” in *Proceedings of the 2018 International Conference on Software and System Process*, pp. 113–122, 2018.

[13] M. Björkman and J. Bergqvist, “Creating a unit testing application prototype for javascript,” 2022.