Tampere University of Applied Sciences

Software Testing

Understanding of the objectives, mindsets, techniques, and strategies

Jingjing Yang

5G00DM62-3004 Software Implementation and Testing

March 2024

Bachelor's Degree Programme in Software Engineering

CONTENTS

1 INTRODUCTION

Software testing, a cornerstone in the software development lifecycle, is crucial for ensuring that the final product is robust, reliable, and meets the intended requirements.

In this report, we start with the objectives and mindsets of successful software testing. We navigate through the varied philosophies of different testing techniques like black-box and white-box testing. Each philosophy contributes uniquely to the comprehension and evaluation of software quality. We then explore the various methods related to those techniques like Equivalence Partitioning, Boundary Value Analysis, and Error Guessing.

Additionally, we discuss different types of testing and see how they come together to form a robust framework, ensuring the delivery of high-quality software products. We distinguish between Unit, Function, System, and Acceptance Testing within the software development process and outline their specific focuses, methodologies, and purposes.

Lastly, we explore structured test planning due to the complexities of testing large systems.

2 OVERVIEW OF SOFTWARE TESTING

2.1 The Essence of Software Testing

Software testing is a critical and integral component of the software development process. It plays a key role in ensuring that the software or application performs to the expected standards. It helps identify bugs, errors, or malfunctions that could potentially lead to unwanted and detrimental outcomes.

2.2 Objectives and Mindset in Testing

Software testing should be seen as a process aiming to identify errors within a program, thus improving its overall quality once errors are fixed. While successful test cases can trigger these errors, the ultimate goal of software testing is to establish some level of confidence that the software does what it should do, and doesn't do what it shouldn’t. A more appropriate definition for testing should be: " Testing is the process of executing a program with the intent of finding errors. " (Myers, 2011, p.6-7).

Myers comes up with ten testing principles, most of which may seem obvious, yet they are all too often overlooked. For example, a programmer should avoid attempting to test his or her own program; checking whether the program "did not do what it should do" is only half of the test, the other half of the test is to check whether the program "did what it should not do" (Myers, 2011, p.12-18).

2.3 A Successful Test

A test case should include two parts: a description of the input data for the program and an accurate description of the expected output results given the input data.

If a test uncovers errors in a program and these errors can be fixed, then this reasonably designed and effectively executed test is considered "successful". If

the test ultimately confirms that no other errors can be detected, it is also considered as "successful" (Myers, 2011, p.6-7).

A “unsuccessful” test is one that fails to properly inspect the program. In most cases, a test that fails to find errors is considered "unsuccessful", because the idea that software does not contain errors is generally impractical. A test case that can discover new errors is unlikely to be considered "unsuccessful". In other words, if it can find errors, it proves that it is worth designing.

3 TESTING TECHNIQUES

Multiple techniques for software testing are available, each with its unique strengths and use cases. An approach suggested is to develop test cases using black-box methods first, then supplement as necessary with white-box methods (Myers, 2011, p.42).

3.1 Examining Outputs with Black-Box Testing

Black-box testing, also referred to as data-driven or input/output-driven testing, is a technique where the evaluation is based solely on the software's outputs, without any insight into its internal code structure. This approach relies on the software’s specifications, concentrating on the software's function rather than its process (Myers, 2011, p.8-9).

This method is especially beneficial for testers lacking extensive programming knowledge but needing to confirm that the software performs as anticipated under different conditions.

3.2 Methods Of Black-Box Testing

Equivalence Partitioning is a black-box testing method that aims to reduce the number of test cases by dividing input data of a software unit into partitions of equivalent data. The idea is to select a representative from each partition as the input for the test case. If the representative value causes an error, it's expected that all other inputs from the same partition will also cause an error, and vice versa. There are two types of equivalence classes: valid ones, which represent valid inputs, and invalid ones, which represent erroneous input values. This approach is used to save time and effort by not testing redundant data (Myers, 2011, p.49-55).

Boundary Value Analysis is a software testing technique that focuses on the values at the boundaries of input and output equivalence classes. This method is based on the observation that errors are most likely to occur at the

boundaries of equivalence classes. Unlike Equivalence Partitioning, which selects any element from an equivalence class, Boundary Value Analysis specifically picks elements that test each edge of the equivalence class. It considers both the input conditions (input space) and the result space (output equivalence classes). This method is used to uncover types of errors that Equivalence Partitioning might miss (Myers, 2011, p.55-61).

Cause-effect graphing is a structured testing method that involves dividing software specifications into sections, identifying input-output relationships, and creating a Boolean graph to outline test cases. It provides a systematic approach to test case design and is effective for complex scenarios where the relationships between inputs and outputs need careful analysis (Myers, 2011, p.61-80).

Error guessing is a less formal testing technique based on the tester's experience and intuition. It involves identifying probable errors or error-prone situations, and creating test cases to expose these potential errors. This method is often used when the tester has a deep understanding of common error patterns and is effective for quickly finding obvious errors (Myers, 2011, p.80-84).

3.3 Understanding Internal Logic with White-Box Testing

White-box testing, also known as logic-driven testing, involves a meticulous examination of the software's internal logic. An understanding of the code structure, algorithms, and internal components of the software is crucial with this approach (Myers, 2011, p.10).

3.4 Methods Of White-Box Testing

It is vital for detecting logical errors and confirming that all potential pathways through the software are tested. Techniques such as statement coverage, decision coverage, and condition coverage can be applied to achieve a thorough assessment of the software's internal operations (Myers, 2011, p.42).

Logic coverage testing is a method used in software testing where the goal is to execute each statement in a program at least once, but this is considered a weak criterion for a reasonable white-box test. The strength of this method is questioned because it might miss certain errors like incorrect logical decisions or undetected paths (Myers, 2011, p.43-44).

A stronger logic coverage criterion, called decision coverage or branch coverage, requires that each decision in the program has a true and a false outcome at least once, meaning each branch direction must be traversed at least once. Decision coverage usually satisfies statement coverage since every statement is on some path starting from either a branch statement or from the entry point of the program. However, there are exceptions like programs with no decisions, multiple entry points, or statements within ON-units (Myers, 2011, p.44-49).

In summary, for programs with only one condition per decision, a sufficient number of test cases are needed to invoke all outcomes of each decision and each point of entry at least once, ensuring all statements are executed. For programs with multiple conditions in their decisions, test cases should cover all possible combinations of condition outcomes and points of entry.

4 DIFFERENT TESTING TYPES

Different types of software testing have a distinct focus and purpose within the development process. The overarching structure of the testing process is designed to prevent unnecessary overlap of testing types and ensures that large classes of errors are not overlooked.

Higher-order testing methods, including function, system, acceptance, and installation testing, are most suitable for software products resulting from contracts or intended for wide usage. For smaller programs or those without formal requirements, function testing might suffice. As the size of the software increases, the need for more extensive higher-order testing also grows due to a higher ratio of design errors to coding errors in larger programs (Myers, 2011, p.113-118).

4.1 Unit Testing

Unit/Module testing involves testing the individual subprograms, subroutines, classes, or procedures within a program before the entire program is tested as a whole. This testing aims to manage the combined elements of testing by focusing on smaller units initially, which aids in debugging and introduces parallelism in the testing process (Myers, 2011, p.85).

Nonincremental testing involves testing all modules as standalone entities and then integrating them all at once. On the other hand, incremental testing involves testing and integrating modules one by one or in small subsets. Incremental testing is preferred as it allows for early detection of interface mismatches and facilitates easier debugging since modules are integrated and tested early in the development process (Myers, 2011, p.96-100).

There are two approaches to incremental testing: top-down and bottom-up. Top-down testing starts with high-level modules and progresses downwards. On the other hand, bottom-up testing begins with the lowest level modules and

integrates upwards. Each approach requires a unique set of drivers and stubs to simulate module functions during testing (Myers, 2011, p.101-110).

Unit testing is conducted at the developer level and is akin to a micro-level inspection of the software. Individual components or modules of the software are tested separately to ensure their proper functioning.

For instance, if we're developing a calculator application, each mathematical operation (addition, subtraction, multiplication, and division) would be a separate unit that would be tested individually for accuracy.

4.2 Function Testing

Function testing is a process to find discrepancies between a program and its external specification, which details the program's intended behavior from an end-user's perspective. While function testing on small programs can be a black-box activity, for larger programs, it relies on prior module testing to achieve white-box logic coverage (Myers, 2011, p.119).

For function testing in a calculator application, we would need to test each function or operation individually to ensure it behaves as expected, which includes both the calculation and the interface elements. For instance, we would test the addition function by inputting various numbers and verifying that the correct sum is displayed to the user.

4.3 System Testing

System testing is a high-level testing that verifies if an entire system meets its original objectives, focusing on the overall behavior and efficacy of the system rather than individual functions. Test cases are derived from both external specifications and user documentation to ensure the system functions as intended and meets user requirements. It requires clearly defined, written, and measurable objectives for accurate performance evaluation (Myers, 2011, p.119-122).

In the context of our calculator, system testing would involve testing the complete application, including the user interface and all operations, to ensure it works as expected.

4.4 Acceptance Testing

Acceptance testing is a process where the program is evaluated against its initial requirements and current needs of end users to ensure that it meets the objectives for which it was created. Unlike other forms of testing typically performed by the development organization, acceptance testing is usually conducted by the customer or the end user of the program. This kind of testing is not generally seen as the developer’s responsibility; instead, it's a final verification performed by the users who will be using the program in its operational setting (Myers, 2011, p.131-132)..

Here, we would test our calculator in real-world conditions, using it as an end-user would, to ensure it performs as expected and is user-friendly.

5 COMPLEXITY OF TESTING LARGE SYSTEMS

Testing large systems can be very complex, which may involve tens of thousands of test cases, the repair of numerous errors, and the coordination of a large team over an extended period. Effective management of such an extensive process is crucial.

5.1 Avoiding Underestimation of Resources

A common mistake in test planning is the assumption that no errors will be found, leading to an underestimation of required resources such as time and personnel.

5.2 Structured Test Planning

A well-structured plan serves as a roadmap for the testing process, outlining objectives, methodologies, resources, schedules, and responsibilities. The plan needs to be adaptable, as software testing often encounters unforeseen challenges and changes in scope (Myers, 2011, p.53).

A test plan should include specific components:

- Objectives for each testing phase.
- Completion criteria for each phase.
- Schedules indicating when test cases will be designed, written, and executed.
- Responsibilities of individuals involved in each phase.
- Test case libraries and standards for systematic identification, writing, and storing of test cases.
- Identification of necessary tools and a plan for their development, acquisition, and usage.
- Allocation of computer time and hardware configurations.
- Integration plans to determine how and in what order various components will be tested together.
- Tracking and debugging procedures for monitoring progress and errors.

- Plans for regression testing to ensure recent changes haven't adversely affected existing functionalities (Myers, 2011, p.133-135).

5.3 Role of Documentation and Metrics

The documenting of test plans, test cases, and test results is important. This documentation not only aids in the transparency and reproducibility of tests but also serves as a valuable reference for future maintenance and testing efforts.

Metrics, on the other hand, are tools to measure the effectiveness and efficiency of the testing process, guiding improvements and decision-making.

6 ANALYSIS AND SUMMARY

The choice of testing techniques, whether black-box or white-box depends on the context and goal of the testing process. Black-Box Testing involves checking the software's input and output without considering its internal workings. White-Box Testing examines the internal logic and structure of the code to identify logical errors and test all software paths. Each method introduced previously can provide a set of specific, useful test cases, but none can individually provide a complete set of test cases. Often, a blend of these techniques yields the most comprehensive testing results.

Software testing is a multi-stage process. It begins with Unit Testing, where individual components of the software are tested separately to confirm their functionality. This is followed by Functional Testing, which verifies that the software behaves as expected. After these individual tests, Integration Testing is conducted to ensure the different software modules interact seamlessly. System Testing then examines the entire system as a whole to ensure it meets specific requirements. The final stage is Acceptance Testing, where the completed software product is tested in real-world scenarios.

The necessity of a well-structured testing plan, detailed documentation, and precise metrics has been emphasized. These critical aspects enhance the effectiveness of the testing process.

Multiple challenges can arise in software testing, including dealing with changes and modifications and managing the testing process effectively. Despite these challenges, a strategic approach, continuous learning, and adaptation can ensure high-quality software products that meet user needs and expectations.

In conclusion, software testing is a crucial and complex part of the software development lifecycle. It involves various types of testing, each serving a unique purpose. While the testing process appears intricate and time-consuming, it enables the early identification and rectification of bugs and errors, saving considerable time and reducing costs.

REFERENCES

Myers, GJ, Sandler, C, & Badgett, T 2011, The Art of Software Testing, John Wiley & Sons, Incorporated, Hoboken. Available from: ProQuest Ebook Central. [26 February 2024].