Systematic Solution for Low Yielding Test Cases Due To Pesticide Paradox

Nitin Kapoor
Associate Product Owner, Siemens Technologies Services Pvt ltd Bangalore, Karnataka

***Abstract*—The test case creation, handling, and maintenance methodology outlined in this research paper addresses the critical issue of diminishing bug detection rates during test suite execution, where traditional approaches fail to identify root causes of this phenomenon. Through empirical analysis, it has been established that test suites require continuous evolution and optimization through strategic addition of new test cases and refinement of existing ones. This dynamic maintenance approach significantly enhances the test suite's effectiveness in detecting software defects and ensures comprehensive coverage of both existing and emerging functionalities within the application under test.**

***Index Terms*—Software Industry, Software Quality, Testing, Quality Improvement, Regression Testing, Testing Efficiency, Test Case Optimization.**

I. INTRODUCTION

The target audience for this research paper encompasses quality assurance professionals, test engineers, and software development practitioners involved in both manual and automated testing methodologies. This includes individuals responsible for test case design, execution, and maintenance within the software development lifecycle (SDLC).

Test cases are meticulously designed artifacts that serve as formal validation mechanisms to verify whether an application conforms to its specified functional requirements and non-functional parameters. These test artifacts are instrumental in maintaining software quality standards and ensuring requirement traceability throughout the development process.

Following a comprehensive peer review process, test cases undergo systematic execution in multiple iterative cycles on the Software Under Test (SUT). This methodical approach facilitates the evaluation of software quality metrics prior to production deployment or market release. The testing cycles typically follow a structured progression through various environments: development, integration, staging, and pre-production.

The primary responsibility of Quality Assurance (QA) engineers lies in detecting and documenting software defects through systematic test execution. This involves meticulous documentation of test results, categorizing failed test scenarios, and creating detailed defect reports in bug tracking systems with appropriate severity and priority classifications.

However, empirical evidence suggests a concerning trend where the defect detection rate demonstrates a declining trajectory over successive testing cycles. This phenomenon potentially indicates either inadequate test coverage, ineffective test design, or possible testing process inefficiencies, ultimately compromising the delivered software quality. This observation necessitates a thorough analysis of current testing methodologies and potential implementation of more robust quality assurance frameworks.

II. PROBLEM STATEMENT

The core of a test case is Execution of Test steps and Expected Result. Any deviation from expected behavior is marked as bug. But it has been observed over period from version to version, there is downfall in number of bugs getting exposed, due to reason mentioned below:

1. Developer keen on fixing bugs locked.
2. Developer keen on improving the area which are more prone to bugs.

It gives illusion to all stakeholders that quality of software is increasing where else it may not. It’s a

failure from testing side that undiscovered bugs are not exposed, due to reason mentioned above. When software is released in market then number of bugs reported by end users goes drastically up, which indicates decrease in quality of software, hence tarnishing the brand value of the company and the product.

To remediate these identified concerns, the following evidence-based solutions or approaches are recommended.

III. APPROACHES

3.1 *Approach-1: Delete Old and Add new*

According to this optimization methodology, test cases exhibiting zero defect detection rate across multiple execution cycles can be systematically eliminated from the test suite, facilitating the integration of new test scenarios. Test cases consistently yielding "Pass" status across multiple software iterations can be deprecated or archived in a non-executable repository, effectively optimizing the active test suite inventory. Refer to table below.

Test Case Vs Status	Version 1	Version 2	Version 3	Version 4	Action Item
Test case 1	Passed	Passed	Passed	Passed	Discarded
Test case 2	Passed	Failed	Passed	Passed	-
Test case 3	Failed	Passed	Passed	Passed	-
Test case 4	Failed	Passed	Failed	Passed	-

Analysis of the empirical data demonstrates that test cases exhibiting consistent 'Passed' status across four or more consecutive software iterations can be candidates for either deprecation or replacement with more comprehensive test scenarios, as the probability of future failures approaches statistical insignificance. The threshold of four iterations is not deterministic but correlates positively with the prediction confidence - higher succession of passes indicates increased reliability in subsequent versions. Test cases that manifest failures within intermediate versions should be retained in the test suite until achieving stability, demonstrated by consistent passes across multiple iterations. This optimization methodology remains applicable irrespective of the test execution paradigm - manual or automated.

Advantages:

- Less maintenance of test cases since obsolete are discarded or deleted, so number of effective test cases remain in control.
- Testing efforts remain almost same since number of test cases deleted or replaced are same.

Disadvantage:

- No fool proof way to ensure that deleted test case will not fail in coming versions, which might be an important functionality.
- Link to the requirement might get disconnected, so no way to make sure that delinked requirement will get covered on addition of new test cases.

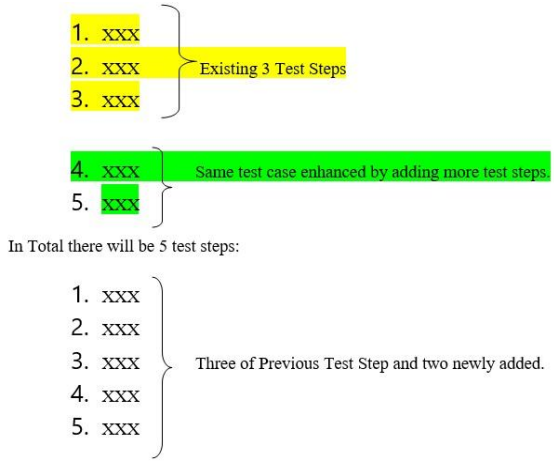
Constraint:

1. This optimization method applies only to functional test cases that consistently pass due to unchanged code segments across multiple versions.
2. This is not applicable for test cases like which are mandatory to make 100% assurance for release of the software. For e.g.: test cases for Platform Test like “Software should be able install and run with latest Windows security patch”.

Windows security patches are regularly released worldwide, so even though tests might be passing with previous released patches, but we need to execute tests to assure it works fine with released one. Hence this optimization/approach is not applicable for these tests.

3.2 *Approach-2: Keep Old and Expand*

According to this approach, keep old test and expand it by adding more test steps to it.



Advantages:

- Old coverage remains and with addition of more test steps coverage increases. It provides deeper coverage.
- Link to legacy requirements remains.

- Chances of getting bugs are more.

Disadvantage:

- Execution time increases more, since in total test steps are more.
- Maintenance becomes difficult, as the size of test case increases.

3.3 Approach-4: Expand and delete

In this technique/approach, enhancement of current test cases is achieved by incorporating additional execution steps after existing one while removing in-effective ones that do not yield defects. Essentially, new steps are integrated to extend test coverage, supplanting ineffective ones. This concept can be visualized through examining application state transitions diagram as explained below:

Let test case-1 described as such:

Test Case-1

Precondition:

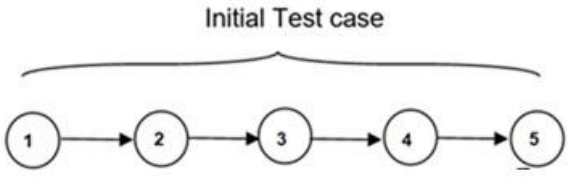
1. Precondition-1
2. Precondition-2

Execution Steps:

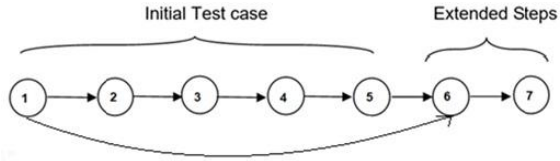
1. Execution Step-1
2. Execution Step-2
3. Execution Step-3
4. Execution Step-4
5. Execution Step-5

Expected Behavior:

Above test case can be represented with diagram below. Execution step-1 is represented by circle with numeral “1” and consequent circles with numbers of corresponding execution steps.



In this situation, the test case isn't finding any bugs. To address this, we add more steps to improve testing depth and remove the unproductive steps. This adjustment moves execution from step-1 directly to step-6. If needed, we change step-1 to set the application state for step-6, ensuring proper execution. This simplified strategy helps improve test coverage by focusing on steps that enhance defect detection.



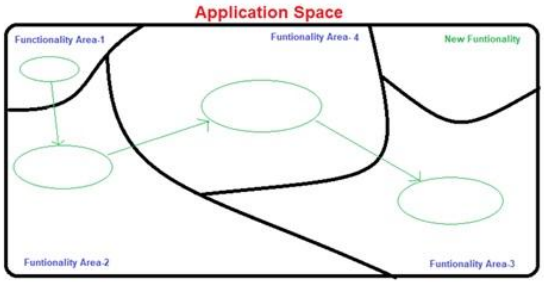
Advantage:

1. Total time taken for execution would be less than previous execution since less steps to execute.
2. Better penetration in terms of coverage.

3.4 *Approach-4: Creation of Customer Workflow*

In this approach, test cases are developed based on customer workflows which cover stabilized legacy functionalities, rather than focusing solely on functionality-centric tests. Traditionally, a test case includes detailed steps and expected behaviors aimed at verifying a specific function. However, this method emphasizes functional areas from which customer workflows may begin, traverse through, and eventually transition to other workflows, culminating in a different functional area.

Consider the diagram below:



As an end-user, the workflow might start at Functional Area-1. With specific user action, the control transitions to Functional Area-2, then to Functional Area-4, and so forth, finally ending at Functional Area-3. This mirrors the exact user interactions experienced by the customer, enabling quicker feedback on the software's quality.

Once the new functionality stabilizes and errors are resolved, the scope can expand to incorporate the new areas into existing customer workflows. Ultimately, these workflows serve as comprehensive "Regression Tests."

Advantages:

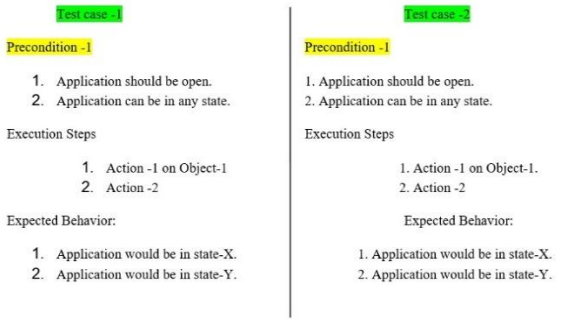
1. Eliminates the need to scrutinize individual functionalities in depth.
2. Provides a more targeted approach for identifying issues.
3. Facilitates faster feedback loops.
4. Can be conducted both manually and through automation, with minimal time investment compared to extensive test suites.

Disadvantages:

1. Edge or corner cases leading to crashes or functional failures might be difficult to detect, which could result in oversight.
2. Testers must possess domain knowledge to accurately cover these customer scenarios.

3.5 *Approach-5: Merging into One*

According to this approach/technique merge old test cases into one test case. Meaning to say apply "Merging". Let's try to understand this with help of example:



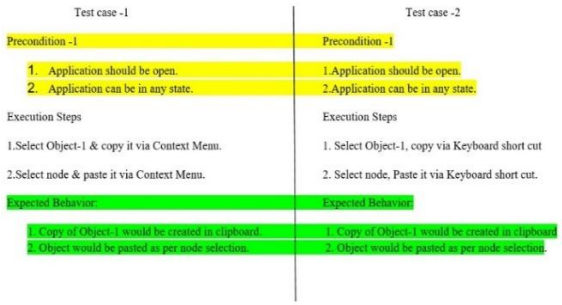
Lines marked with "Yellow" color states precondition to start the test case or entry point for test case.

Now if look precisely and compare then we can make out that Lines marked with "Yellow" color are same in both Test case-1 and Test case-2, and after execution of steps of Test Case-1 application is in open state with some state-Y which will serve as entry point for Test case-2.

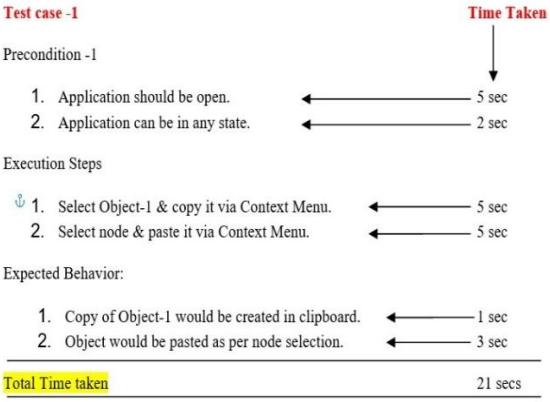
Meaning to say it does not matter for both test cases in which state application is, but what really matters it should be in opened state.

Hence immediately after execution of steps of test case-1, we can execute steps of test case-2.

Let's take an example to understand:



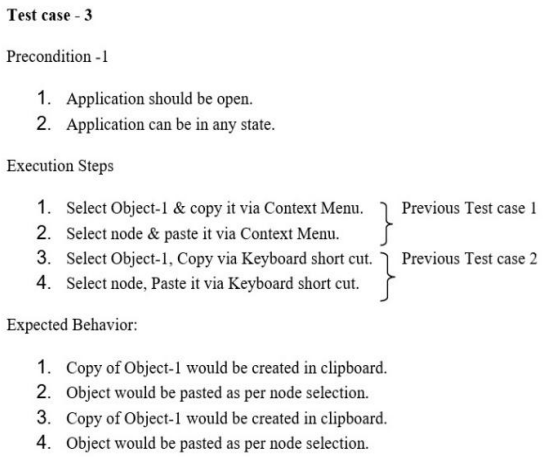
Yellow	Same Precondition
Green	Same Expected Behaviour



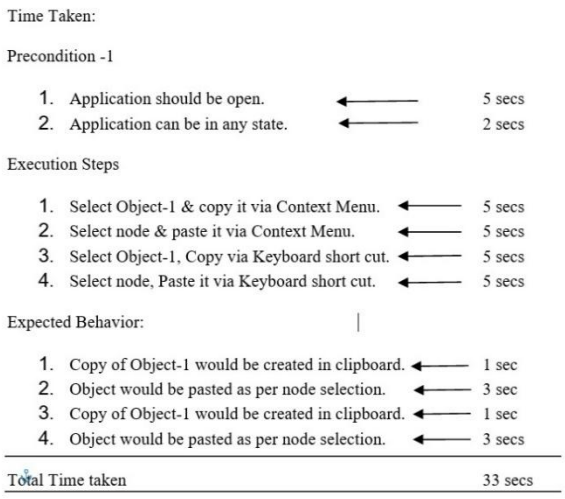
Now to execute test case 2, it will take another 21 secs

Hence to execute both test cases 1 & 2, total it will take 42 secs.

After Merging Test case would be like this:



Time Taken After Merging the Test Case:



Time saving:
TC-1 (21 secs) + TC-2(21 secs) =42 secs
Merged Test case: TC-3 (33 secs) =33 secs
Time Saved =9 secs
In terms of percentage: = 21.42% of time saved

Advantage:

1. The number of test cases just got reduced. Almost by 50%. (Due to merging) two test cases into one.
2. Since test cases number reduced, hence less efforts for maintenance. (for updating test cases)

3. Hence test execution time will be reduced to a maximum of 50%, (in ideal scenario).

IV RESULTS

After applying these techniques and approaches across several software releases, we evaluated the results by tracking the time invested. The findings revealed a substantial reduction in effort and maintenance time. Time saved directly translates to cost savings. Additionally, this approach led to increased efficiency by providing faster feedback in a shorter timeframe.

V CONCLUSION

This case study highlights the necessity of adopting diverse approaches to save time, thereby reducing costs. The techniques described are focused methodologies that enhance the test team's efficiency while addressing the challenges of maintaining extensive test suites. Furthermore, the future integration of AI and ML tools will better align with organizational needs in this area.

VI. ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Siemens Technology & Services Pvt. Ltd. for providing the essential support and resources needed to conduct this study. I am especially thankful to my direct supervisor, Mr. Amol Vinayakrao Birajdar, for his invaluable input, guidance, and motivation throughout the study. Additionally, I extend my heartfelt thanks to my team members and other departmental teams for their support in this endeavor.

REFERENCES

[1] G. J. Myers, C. Sandler, and T. Badgett, "The Art of Software Testing," 3rd ed., Hoboken, NJ: John Wiley & Sons, 2012.
[2] R. Black, "Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional," Indianapolis, IN: Wiley Publishing, 2016.
[3] A. P. Mathur, "Foundations of Software Testing," 2nd ed., Delhi, India: Pearson Education, 2014.
[4] L. Copeland, "A Practitioner's Guide to Software Test Design," Boston, MA: Artech House Publishers, 2004.
[5] P. Ammann and J. Offutt, "Introduction to Software Testing," 2nd ed., Cambridge, UK: Cambridge University Press, 2017.
[6] I. Burnstein, "Practical Software Testing: A Process-Oriented Approach," New York: Springer, 2003.
[7] R. Patton, "Software Testing," 2nd ed., Indianapolis, IN: Sams Publishing, 2005.
[8] M. E. Khan, "Different Approaches to White Box Testing Technique for Finding Errors," International Journal of Software Engineering and Its Applications, vol. 5, no. 3, pp. 1-14, 2011.