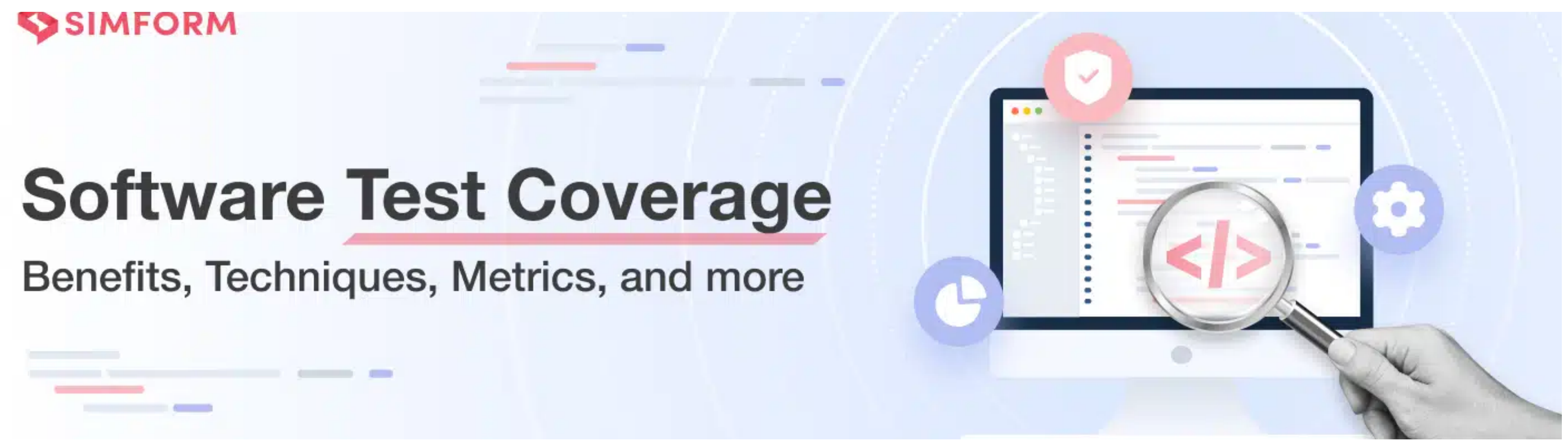
A Detailed Guide on Test Coverage

Learn more about test coverage and how enhancing it might help you while software testing by reading this blog.

Hardik Shah
June 28, 2021 10 mins read Last Updated June 22, 2023



Quick Summary :– *Test coverage is an important indicator in software testing in terms of quality and effectiveness. Read this blog to understand test coverage, its techniques, metrics, matrix and how to improve it.*

Table of Contents

What is Test Coverage?
Benefits of Test Coverage
Code Coverage v/s Test Coverage
Test Coverage Techniques
Test Coverage Metrics
Test Coverage Matrix
What do you Measure Test Coverage Against?
Test Coverage in TDD
How to Improve Test Coverage?

The world has witnessed some of the disastrous events due to the errors prevailing in the software. One such event, which I personally recall, is the opening of Heathrow Terminal 5, the UK in 2008.

Engineers were pretty confident about the working of terminal pertaining to their rigorous testing. Though the baggage handling system couldn’t cope up when it faced some real-life scenarios; which resulted in complete shut down of the system. Over the following 10 days, some 42,000 bags failed to travel with their owners, and over 500 flights were cancelled.

All this due to the failure of engineers to carry out the test coverage of possible real-life scenarios. And it went down as “Disastrous Opening Day for Terminal 5” But we, at Simform, have no room for any mistakes in our test coverage practices.

In this blog, you will discuss all the aspects of test coverage and how it directly affects the production, whether it is custom software development or **software testing**.

professionals to deliver high-quality products that drive customer satisfaction.

What is Test Coverage?

Test coverage is defined as a technique which determines whether our test cases are actually covering the application code and how much code is exercised when we run those test cases.

If there are 10 requirements and 100 tests created and if 90 tests are executed then test coverage is 90%. Now, based on this metric, testers can create additional test cases for remaining tests.

Benefits of Test Coverage

#1. Eliminates defects at early stages

You can identify gaps in requirements, test cases and defects at early stages of your product development life cycle. It saves you from a lot of headaches later.

#2. Better coverage

Test coverage creates more test cases to ensure superior coverage. This leads to fewer defects and work to do at later stages. Moreover, you get to increase customer satisfaction with a refined product.

#3. Removes redundant cases

Test coverage is especially useful in identifying and eliminating test cases that don't make much sense in the current project. Your developers can report these cases to remove them and make the overall code lighter.

#4. Higher ROI

With fewer defects at production stages and lesser user acceptance testing defects, test coverage can have a significant impact on the ROI. All the resources which would've been spent on addressing defects, now translate into your profits.

#5. Discovers uncovered areas

Test coverage helps you unearth areas of a program that have not been covered by a set of test cases. It helps make your program more robust and error-free.

#6. Superior control

Test coverage gives you a better control over the resources during the product development lifecycle. You save time by eliminating defects earlier and faster. The saved time allows you to keep a tab of costs. And most importantly, you get to have a firm grip on the scope of the project.

#7. Smoother testing cycles

You can prevent defect leakage using Test coverage analysis. Test coverage also helps in **Regression testing**, test case prioritization, test suite augmentation and test suite minimization. All this leads to smoother yet efficient testing cycles.

Before we dive deeper into test coverage and it's techniques, let's also discuss another wildly popular testing methodology— code coverage.

What Is Code Coverage And How Is It Different From Test Coverage?

comprehensive.

The primary difference between test coverage and code coverage is that while test coverage is a qualitative measure, code coverage is more about quantitative measurement in terms of testing.

Code coverage helps determine how much of the code is executed while the application is running, whereas test coverage is helpful in describing test cases that can be written against the requirements document.

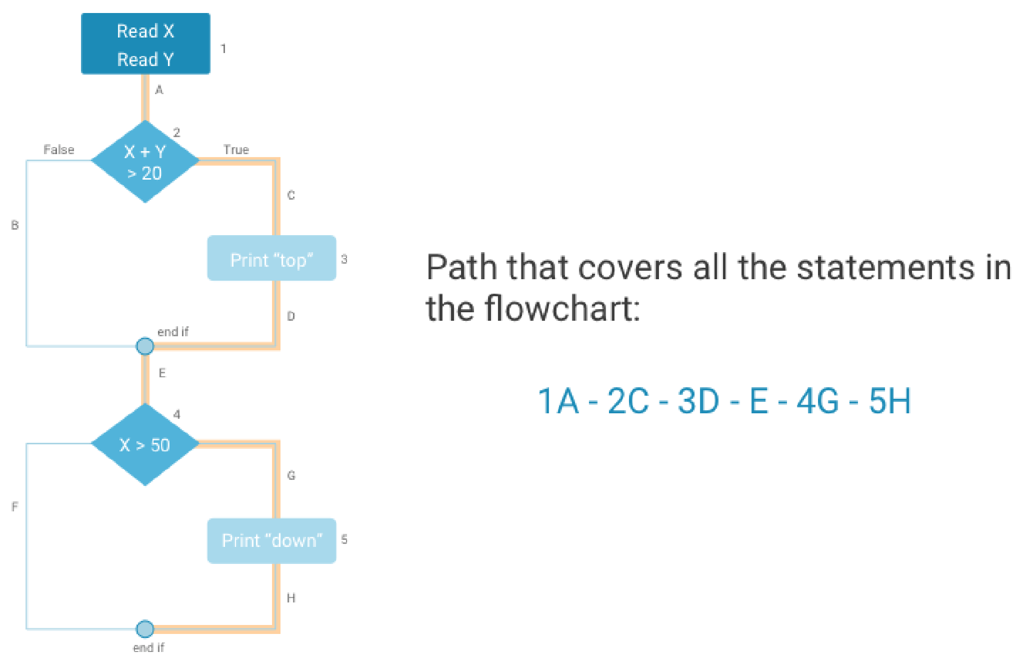
With the code coverage v/s test coverage debate addressed, let's now take a look at various test coverage techniques.

Test Coverage Techniques

#1. Statement Coverage

Statement coverage ensures that all the statements in the source code have been tested at least once. It provides the details of both executed and failed code blocks out of total code blocks.

Let's understand it with the example of the flow diagram. In the given example, this path 1A-2C-3D-E-4G-5H covers all the statements and hence it requires only on a test case to cover all the requirements. One test case means one statement coverage.



In complex code, a single path is not sufficient to cover all the statements. In that case, you need to write multiple test cases to cover all the statements.

Advantages:

It can be applied directly to object code and does not require processing source code.

It verifies what the written code is expected to do and not to do

Disadvantages:

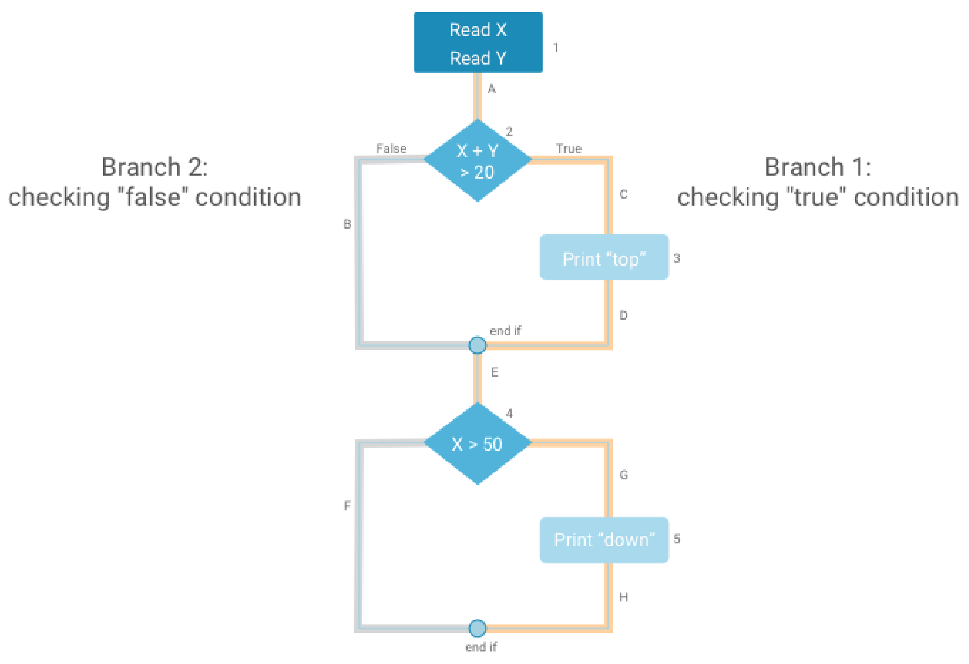
It covers only true conditions of every statement.

Does not report when loop reaches to the terminations.

Statement coverage is completely insensitive to the logical operators (|| and &&).

Statement coverage is the basic coverage and hence does not guarantee 100% coverage.

the code to meet the functionality requirements. Branching in the code is actually a jump from one decision point to another point. Branch Coverage checks every possible path or branch in the code is covered.



Branch coverage can be calculated by finding the minimum number of paths which ensure that all the edges have been covered. In the given example, there is no single path that ensures coverage of all the edges at one go.

For example, if you follow this path 1A-2C-3D-E-4G-5H which covers the maximum number of edges(A, C, D, E, G and H), you will still miss the two edges B and F. You need to follow another path 1A-2B-E-4F to cover the remaining two edges. By combining the above two paths you can ensure of travelling through all the paths. For this test coverage example, our branch coverage is 2 as we are following two paths and it requires two test cases to meet the requirements.

Advantages:

It covers both the true and false conditions unlikely the statement coverage.

It validates if all branches are tested.

Disadvantages:

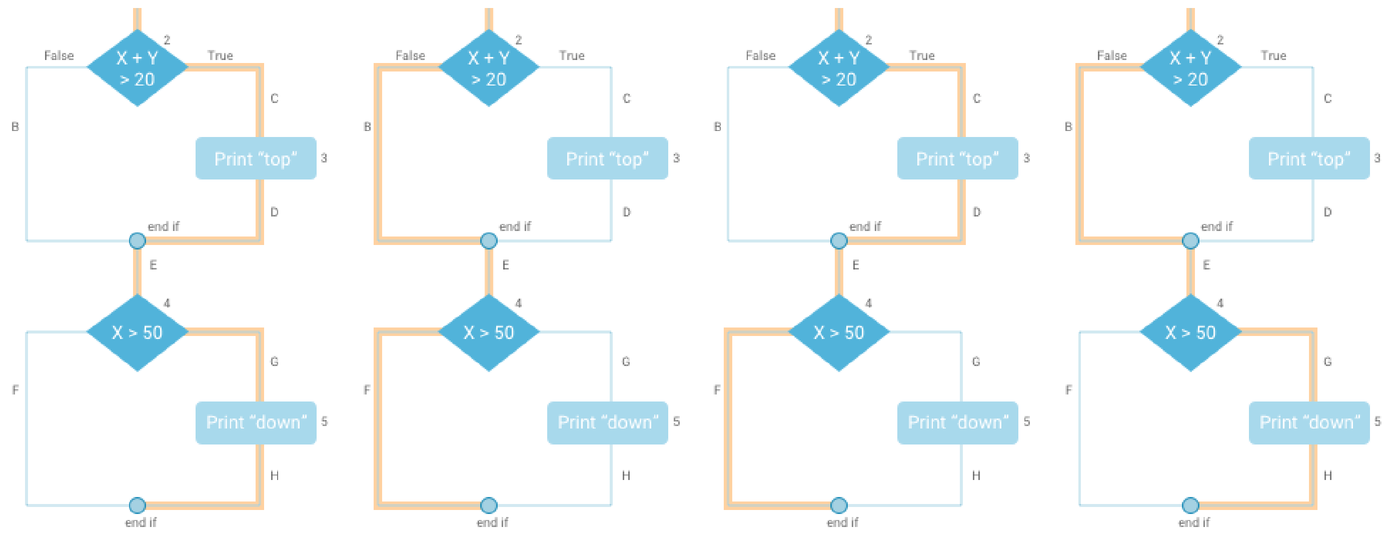
It ignores branches within Boolean expressions which occur due to short-circuit operators.

#3. Path Coverage

Path testing is a structural testing method that involves using the source code of a program in order to find every possible executable path.

Path Coverage ensures coverage of all the paths from start to end. In this example, there are four possible paths:

1. 1A-2B-E-4F
2. 1A-2B-E-4G-5H
3. 1A-2C-3D-E-4G-5H
4. 1A-2C-3D-E-4F



Advantages:

It helps to reduce redundant tests.

Path coverage provides high test coverage because it covers all statements and branches in the code.

Disadvantages:

Testing each path is challenging as well as time-consuming because a number of paths are exponential to the number of branches. For example, a function containing 10 if-statements has 1024 paths to test.

Sometimes many paths are impossible to exercise due to relationships of data.

#4. Condition Coverage

Condition coverage checks if both the outcomes("true" or false") of every condition have been exercised. The outcome of the decision point is only relevant for checking the conditions. It requires two test cases per condition for two outcomes.

Advantages:

Condition coverage measures the conditions independently of each other.

It has better sensitivity to the control flow.

Disadvantages:

Similar to the branch/decision coverage.

#5. Boundary Value Coverage

This coverage metric is very useful for those applications in which error occurred due to input numbers. And these errors occurred at boundary values. In boundary value coverage, test cases are selected at the endpoints of the equivalence classes. For this test coverage example, below are boundary values for an application which requires 3-digit number as an input.

1. 100(Minimum)
2. 99(Just below the minimum boundary value)
3. 999(Maximum)
4. 1000(Just above the maximum boundary value)

Advantages:

It is quite easy to test a small set of data in place of testing a whole lot of data sets.

It can not test dependencies between two inputs.

It can not cover code which contains boolean functions.

#6. Product Coverage

Product coverage refers to the process of testing a software solution from a product perspective and evaluating specific product areas. It involves evaluating the performance of software across various scenarios by

Defining and prioritizing requirements

Developing a checklist

Implementing effective test automation

#7. Risk Coverage

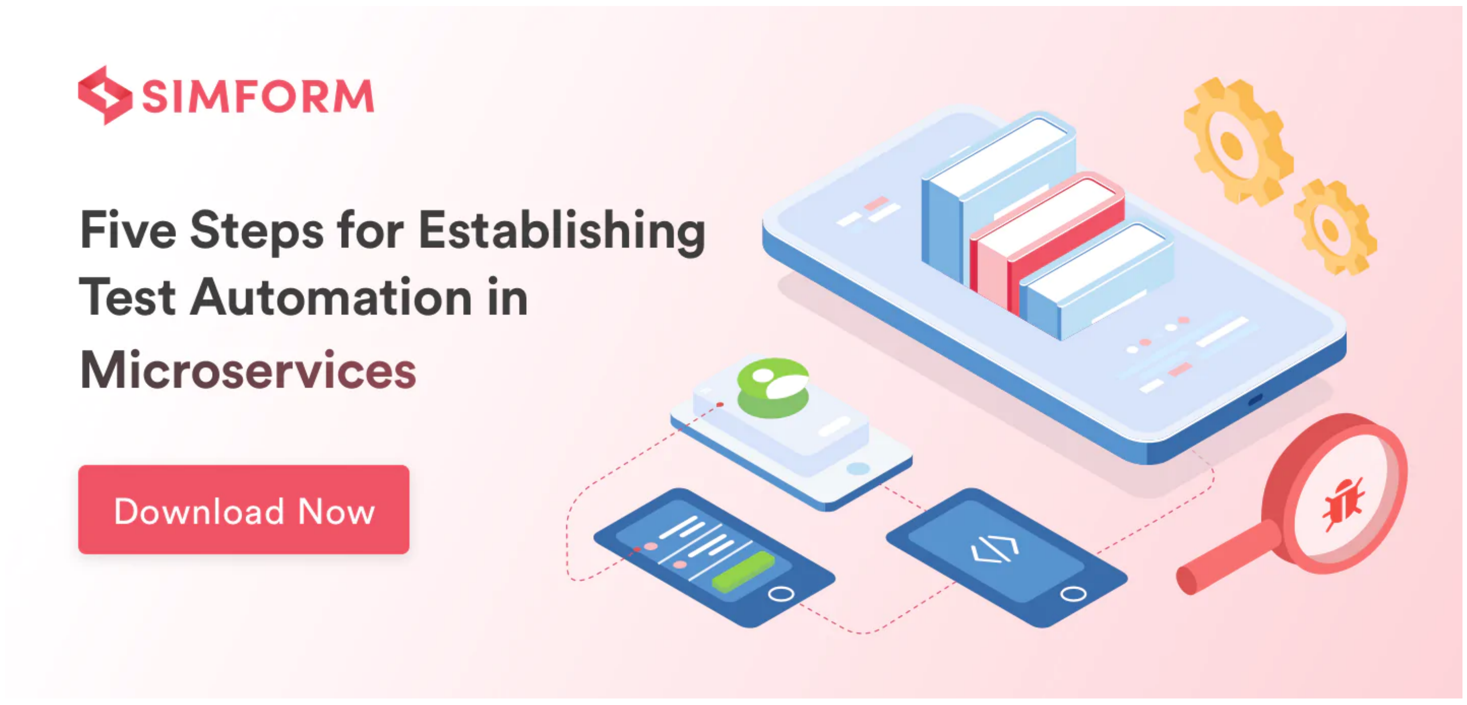
The risk coverage technique is used to comprehensively evaluate all software-related risks and rigorously test the application. It involves identifying and assessing all potential risks and the issues that may arise from them, including low-probability scenarios that could significantly impact the software.

#8. Requirements Coverage

Requirements coverage is used to assess whether the end-product encompasses all the committed functionalities and fulfills the customer’s stipulated requirements. It measures the number of requirements addressed and the types of tests executed to validate them.

#9. Compatability Coverage

Compatibility coverage is a technique used to ensure the software works seamlessly across all browsers, operating systems, and devices without any issues or complications. It involves testing the product’s compatibility-related problems, including mobile testing, hardware testing, browser testing, network testing, and other subtypes.


SIMFORM
Five Steps for Establishing Test Automation in Microservices
Download Now

What are Test Coverage Metrics?

Test coverage metrics measure the test effort and help answer, “How much of the application was tested?” Test coverage metrics can be divided into three parts: code-level metrics, feature testing metrics and application level metrics. There are various test coverage formulas to calculate these results and generate test coverage reports.

Code level metrics

#1. Test Execution Coverage Percentage

It also called as executed tests and it is a percentage of passed/executed tests out of the total number of tests.

$$\text{Executed Tests Or Test Execution Coverage Percentage} = \frac{\text{Number of tests run}}{\text{Total number of tests to be run}} \times 100\%$$

The advantage is that you get an overview of testing progress by counting the number of passed and failed tests.

The disadvantage is that counting test cases passed doesn't say anything about the quality of those tests. For example, some tests might pass because it checks the trivial condition or some error in the test code while it's not functioning as per requirement.

Feature testing metrics

#1. Requirements coverage

Requirements coverage is used to determine how well the test cases cover the software requirements. For that, you simply need to divide the number of requirements covered by the total number of scoped requirements for a sprint, release or project.

$$\text{Requirements Coverage} = \frac{\text{Number of requirements covered}}{\text{Total number of requirements}} \times 100\%$$

In the Requirements module, you create test coverage by linking tests to a requirement. Test coverage assesses the impact of a change in the test or requirement.

Instead of covering each requirement only at the level of the test, you can cover a requirement by test configurations. A test configuration represents a specific use-case of a test. Covering test configurations with requirements provides finer granularity by covering multiple use cases.

#2. Test cases by the requirement

This metric is used to see what features are being tested and the number of tests that aligned with a requirement. Most of the requirements contain multiple test cases. It is very important to know which test cases are failed for a particular requirement to rewrite the test cases for specific requirements.

Request 1	Test Case 1	Pass
Request 2	Test Case 2	Failed
Request 3	Test Case 3	Incomplete

This metric is very important for stakeholders as it shows the progress of the app/software development.

Application level metrics

#1. Defect Density

Defect density is a measure of the total known defects divided by the size of the software entity being measured.

$$\text{Defect Density} = \frac{\text{Number of known defects}}{\text{Size of software entity}}$$

It is used to identify the areas which require automation. If defect density is high for the specific functionality than it requires retesting. To reduce the efforts of retesting, test cases for known defects can be automated.

It is important to consider the priority of the defect (low or high) while evaluating the defects. For example, multiple low priority defects may pass because the acceptance criteria have been satisfied. And on the other hand, only one high priority defect may prevent acceptance criteria from being satisfied. Higher-priority defects are generally weighted more heavily as part of this metric.

#2. Requirements without Test Coverage

After calculating the requirements coverage, you will find some requirements which are not covered. Now, it is important to know about each requirement which has not been covered and what stage the requirement is in.

Request ID	Request Name	Request Status
Request 001	Request A	To Do
Request 002	Request B	Done

This metric helps test engineers and developers to identify and eliminate uncovered requirements from total requirements before they send them to the production phase.

Read more: What is Functional Testing? Explained with Test Cases and Examples

What is Test Coverage Matrix?

To get started with test coverage matrix, there are no prerequisites. You may create your own table, considering the following table as an example.

Feature	Module	Case Type	Test Case
User	Login	Default Case	Valid username, valid password
		Negative	Invalid username, valid password
		Negative	Valid username, invalid password
		Negative	Empty username, empty password
		Boundary	Username & password exceeding the maximum length of 10 characters

What do you Measure Test Coverage Against?

The one thing many QA teams don't consider while measuring test coverage: What Do You Measure Test Coverage Against? Or How to Measure the Test Coverage? Or How to Ensure Test Coverage?

Test coverage measured against lines of code. It is a Test Execution Coverage Percentage which is discussed above. For example, If you have executed 800 lines of code through test cases, out of 1000 lines of code, then your test coverage is 80%.

Now, you need to measure test coverage by requirements to focus on good test cases in your test repository. A good test case traces a requirement (happy and unhappy flows included) to fulfilment. All you have to do to ensure you mainly have good test cases is to establish Requirements Traceability.

You can achieve traceability in your projects by mapping out test scenarios to cover all the requirements in scope for that release, project, iteration, sprint. By establishing Requirements Traceability, you know – at any given point in time – test coverage by requirements.

Test coverage in Test Driven Development (TDD)

Following TDD makes sure each new development has automation test coverage and the behaviour is safeguarded iteration after iteration. Generally, writing tests first is strongly correlated with better coverage, and better coverage is strongly correlated with fewer bugs. It also tells us that writing tests first correlates with better API designs.

Let's take an example to understand how TDD can facilitate better test coverage. At Simform, we follow 2-week sprint as a part of agile methodology in software development.

Sprint – Week 1

In the planning session, our engineering team shortlists the x most important stories on the backlog for delivery. The Quality Assurance team analyses the new sprint backlog and maps out all the test scenarios they would be covering in their test cases. These scenarios are then passed over to the development team for doing unit testing.

Sprint – Week 2

confident about the test coverage.

Sprint – Week 3

Developers have used the test scenarios and test cases to complete unit testing of their codes. This has led to weeding out of easy to find and conspicuous bugs. Further the code is deployed using continuous integration and continuous deployment tools to facilitate end-to-end functional testing. Tester use the detailed test cases to execute end-to-end functional testing and identify critical bugs.

As we get to the end of Sprint, the team is ready for the product release with a good 85%+ test coverage and no critical bugs. Thus **Test driven development** makes a high test coverage possible which leads to robust product and satisfied customer.

However, if you are doing TDD by the book you should always end up with 100% test code coverage because you will never add new functionality to the application, a new branch in code, without first having failing tests for that specific new piece of code.

How to Improve Test Coverage?

#1. Remove Dead Code

Total coverage can be defined as a ratio of code coverage and test coverage (covered_code/total_code). You can increase coverage by decreasing the denominator which is a total code. This can be possible by removing the Dead code or Zombie code. Usually, Dead code can be found in program history in which feature code was added, deleted or disabled, and the debugging code was likewise added and deleted. In this way, you can increase your total code coverage ratio without writing any additional tests.

Dead code can be found easily by manual testing or using automation tools. Before removal of dead code, you need to perform functional testing and if it performs exactly as per requirements than unused can be removed. You can also use static test coverage analysis tools to identify the unused dead code from the source code. The effectiveness of static analysis, however, depends on tools, language and architecture, but it's a good start.

#2. Remove Redundant/Cloned Code

Removing cloned code can improve test coverage ratios in the same way as removing dead code.

The long program contains base code and code blocks which have clones in the programme. If you find these clones and remove it, you can increase test coverage with two improvements, one is size reduction and the second is "free" additional testing. You can increase your test coverage 5-10% by removing the cloned code from your source code.

#4. Increase device coverage

The digital space is complex more than ever. Mobile devices and OS versions are also being introduced to the market in a similar cadence while desktop browsers like Chrome and Firefox are releasing a public beta or GA on a regular basis. While each release not just introduces a test coverage challenge, but also introduces new features and functionalities which require additional testing efforts.

To measure maximum testing coverage, Perfecto Mobile has built a methodology that is based on market usage analytics and looks at the market devices from various angles.

The way to measure mobile testing coverage is to understand through usage and customer analytics. Make a list of top 30 devices that are mostly used in your geography. Also, make sure to include devices from both iOS and Android OS version perspective as well as device ages (New and trending, Most popular, Legacy).