# DEPRECATED: This document has been moved to the [knowledge base](https://episource.atlassian.net/wiki/spaces/ET/pages/150208541/Automated+Testing+Practices) and exists here only for historical purposes

This document is describing the process to be followed to assure the quality of the product through software testing techniques.

[Test Cases](#_o55sx7kcwju5)

[What is smoke testing?](#_6dwm21t827u3)

[What is regression testing?](#_941ccmsheuxm)

[Bug Management](#_i2k3id3niyni)

[Test Management Tool](#_g26qk0plqouz)

[Test Automation](#_je28ho5aeka1)

[Test Design Techniques](#_dm1ro9pmw8j5)

[Boundary Value Analysis](#_jcltt4tshl4f)

[Equivalence testing](#_kepz3aip8zxl)

[Decision table](#_vgho4akjhcoj)

[State transition](#_7h8cuppuglky)

[Negative testing](#_gp294t9ill47)

[Error Guessing](#_o1qc2zf0012f)

[Performance Testing](#_11cs08d9z8qt)

[Unit Testing](#_pfgax8usvdw5)

[Measuring Unit Test Effectiveness](#_n4ni1o4k527h)

[Code Coverage](#_4s9sjcv8w4nq)

[Bug Density](#_9sm60unqxhnd)

[Test Failure Rate](#_xoag6cug5jfj)

[Defect Detection Rate](#_mf6mlvi1hc5w)

[Examples of Unit Tests](#_uce315l2378f)

[Over Reliance on Unit Tests](#_7kfgqwhzkj9x)

Topics:

* Test Cases
* Bug Management
* Test Management Tool
* Test Automation
* Test Design Techniques description
* Performance testing
* BDD/Gherkin
* Unit testing

## Test Cases

Each testable story will have its own set of test cases.

Every test case should be written in detail, meaning that someone, who probably has never used the product before, can follow the steps and get the test results. All test cases must be written in Gherkin language.

A test case must have a short and easy to understand title, a brief description of the test scenario, preconditions (in case needed), test steps, and expected result of each test step.

To create test cases, we need to use the next techniques (description at the end of document)

1. Boundary Value Analysis (BVA)
2. Equivalence Partitioning
3. Decision table
4. State Transitioning
5. Negative testing
6. Error Guessing

With the help of these techniques, we will be able to create quality test cases that will assure the quality of our product.

The goal being less than 5% of defect leakage to production. Being as close to 0% as possible is the ideal.

Test cases created will need to have a label indicating if the test case is applicable for which feature, and if will be considered for Regression testing, or smoke testing.

### What is smoke testing?

Smoke testing is running a small suite of test cases that are usually executed once a new feature is just deployed to check if the environment is completely functional before running any functional testing suite.

### What is regression testing?

Regression testing is running a suite of test cases that will validate that related features that were deployed previously are still working as expected after new deployment.

## Bug Management

If a bug is found in the new feature currently in test. Ticket will be returned to the dev team to fix it. QA will only raise bug tickets for features that are tested and completed before and start to fail due to changes in new features.

Once a bug is found. A ticket will be raised in Shortcut and will be assigned to the dev team.

Bug should have an easy to understand title. As well as a well detailed description.

Including:

* Test steps followed
* Data used
* Environment (qa/uat)
* Evidence

\*Connect with [Risk prioritization document](https://docs.google.com/document/d/1ZaZt_Ch9Omb2fLeXmgNzvcgN4vXKr5Kd0mGhQySLswg/edit#heading=h.67tco1jl1qz2) and [Planning and Execution doc](https://docs.google.com/document/u/0/d/1TSQS1EXLswR2akklOnCQEb0bfn1qrU5AeGeFv9YQm1E/edit)

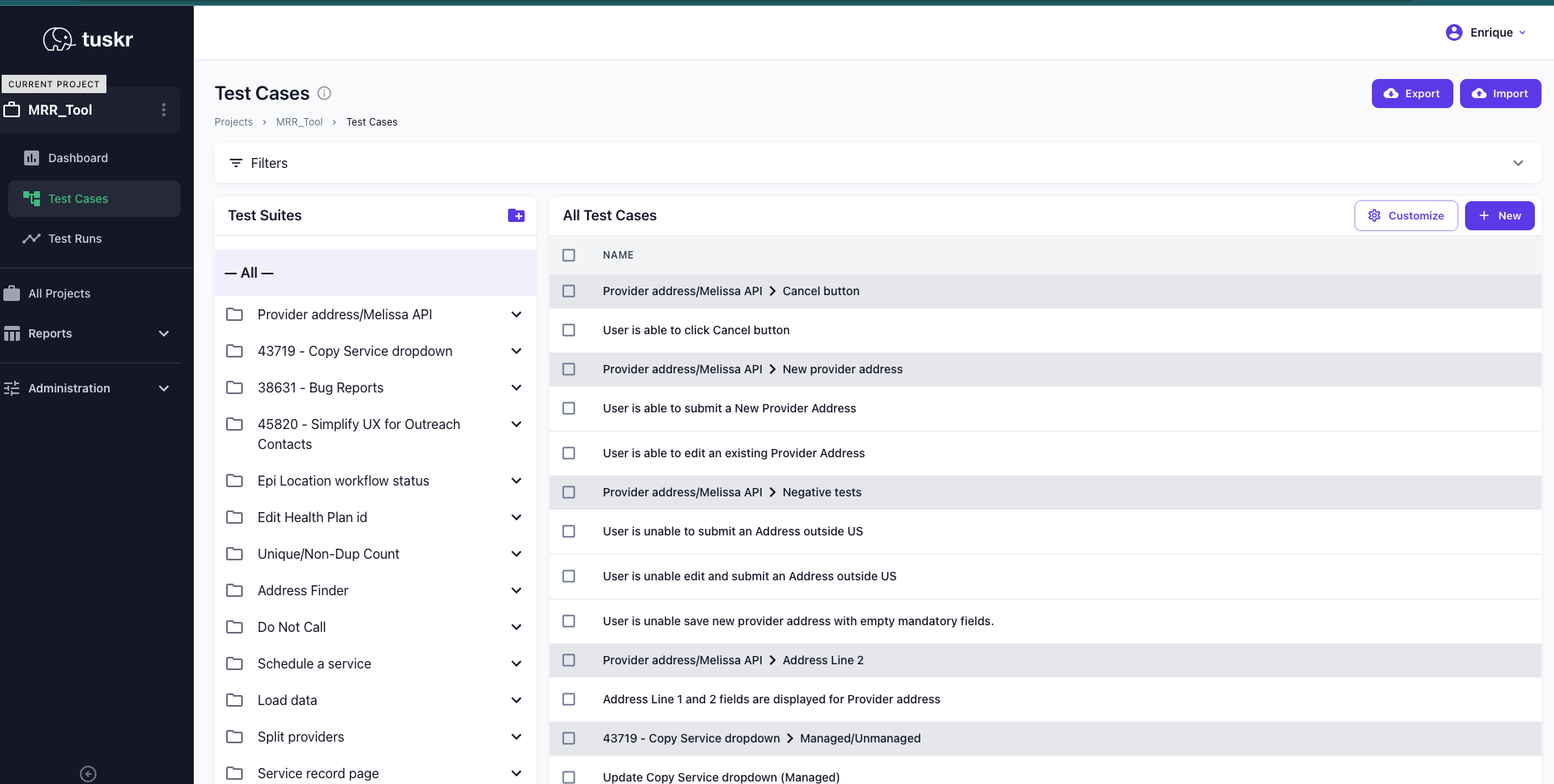
## Test Management Tool

Having a Test Management tool to help gather all the information needed for QA deliverables may be translated into many advantages.

* Data accuracy.
* Easy way to find test cases related to a specific feature (specially when a Regression testing is needed)
* Easy way to build a Test suite for a test run session.
* All historical data for test execution is being saved (including evidence whenever it is appropriate)
* Easy to generate dashboards to report QA test results.
* More time to spend in Test Automation.
* Ability to automatically register the Automated test results into the tool to keep accuracy in the test results history.
* Any person having access to the tool may find any information needed easily.

MCC/MRR teams are using the tool “Tuskr” for help with test management.

In this tool, we are able to write all our test cases and organize them by feature.

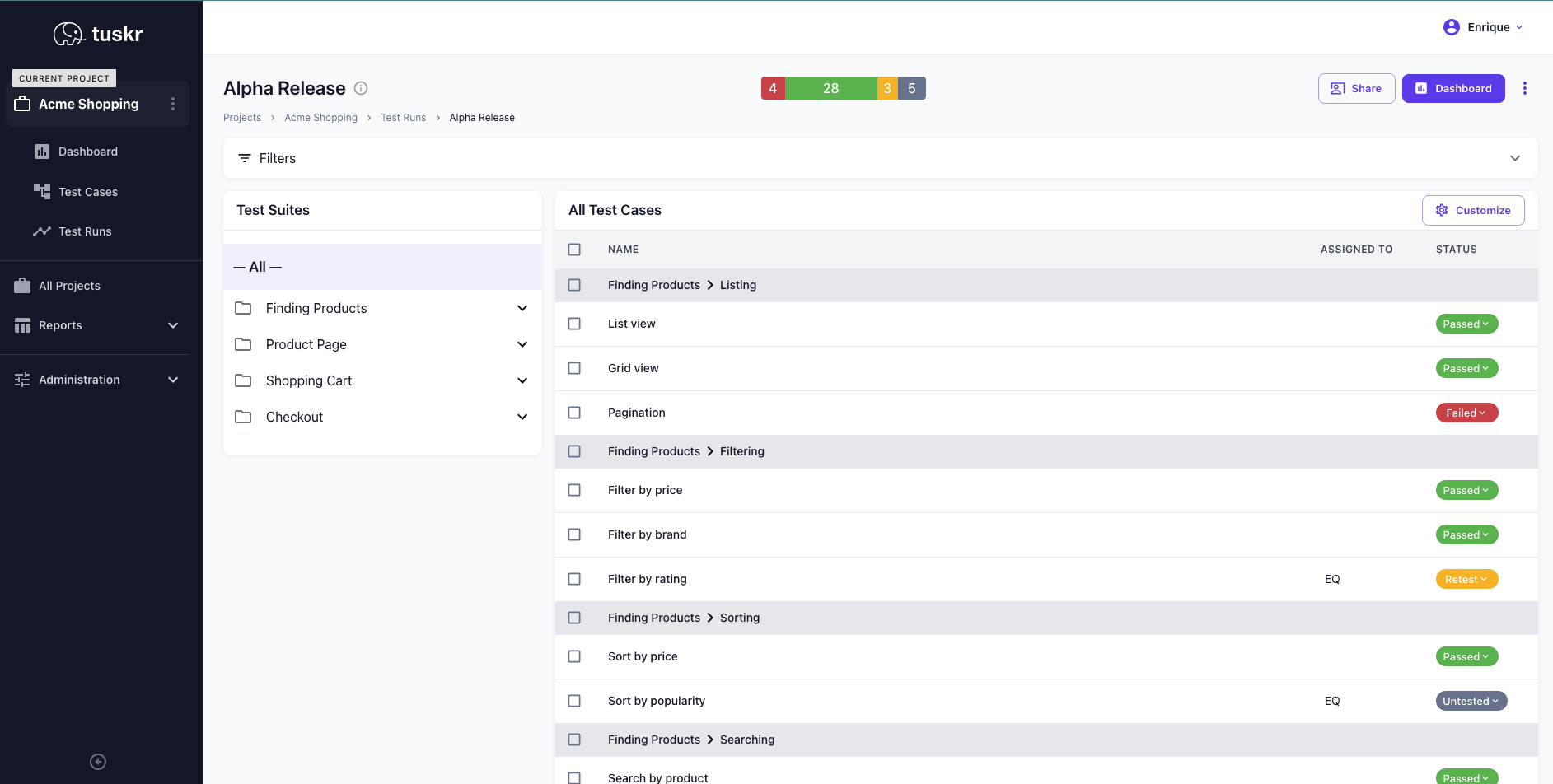


When creating a new test case in Tuskr, this is the information we need to include:

* Name
* Suite (to relate it to the feature to be tested)
* Preconditions (if applicable)
* Test Steps (in Gherkin language)
* Automation status (Not applicable, Pending automation, Automated)
* Shortcut Story Id.

We are also registering the test execution results in Tuskr.

When testing a feature is required, we can pull the suite related to the feature to be tested and easily build a Test run session.



## Test Automation

Tools we have selected for Test Automation  
  
**Selenium:** Open-source tool for automating web applications for testing purposes. It can be used with different programming languages. It supports different browsers and OS. It has a really big community of users that help each other on the web.  
  
**Cucumber:** Open-source tool for test automation using BDD (Behavior-Driven Development) using Gherkin language.  
  
**SikuliX:** Open-source tool for automation based on image recognition. Useful whenever there’s no easy access to GUI’s internals, source code of the application or web page we need to interact with.  
  
To install our Test Automation project - include LINK  
  
For the near future, we are targeting to integrate our test automation project to a CI/CD pipeline.

## Test Design Techniques

### Boundary Value Analysis

Is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.

### Equivalence testing

Allows you to divide a set of test conditions into a partition which should be considered the same. This software testing method divides the input domain of a program into classes of data from which test cases should be designed. The concept behind this technique is that a test case of a representative value of each class is equal to a test of any other value of the same class. It allows you to Identify valid as well as invalid equivalence classes.

### Decision table

This software testing technique is used for functions which respond to a combination of inputs or events. The first task is to identify functionalities where the output depends on a combination of inputs. If there are large input sets of combinations, then divide it into smaller subsets which are helpful for managing a decision table. For every function, you need to create a table and list down all types of combinations of inputs and its respective outputs. This helps to identify a condition that is overlooked by the tester.

### State transition

In State Transition technique changes in input conditions change the state of the Application Under Test (AUT). This testing technique allows the tester to test the behavior of an AUT. The tester can perform this action by entering various input conditions in a sequence. In State transition technique, the testing team provides positive as well as negative input test values for evaluating the system behavior.

### Negative testing

Negative Testing is a software testing type used to check the software application for unexpected input data and conditions. Unexpected data or conditions can be anything from wrong data type to strong hacking attack. The purpose of negative testing is to prevent the software application from crashing due to negative inputs and improve the quality and stability.

### Error Guessing

Error Guessing is a software testing technique based on guessing the error which can prevail in the code. The technique is heavily based on the experience where the test analysts use their experience to guess the problematic part of the testing application. Hence, the test analysts must be skilled and experienced for better error guessing.

The technique counts a list of possible errors or error-prone situations. Then the tester writes a test case to expose those errors. To design test cases based on this software testing technique, the analyst can use the past experiences to identify the conditions.

## Performance Testing

Performance Testing is a software testing process used for testing the speed, response time, stability, reliability, scalability and resource usage of a software application under a particular workload. The main purpose of performance testing is to identify and eliminate the performance bottlenecks in the software application.

* **Speed** – Determines whether the application responds quickly
* **Scalability** – Determines the maximum user load the software application can handle
* **Stability** – Determines if the application is stable under varying loads

To analyze the performance of the web server under test, you should focus on 2 parameters

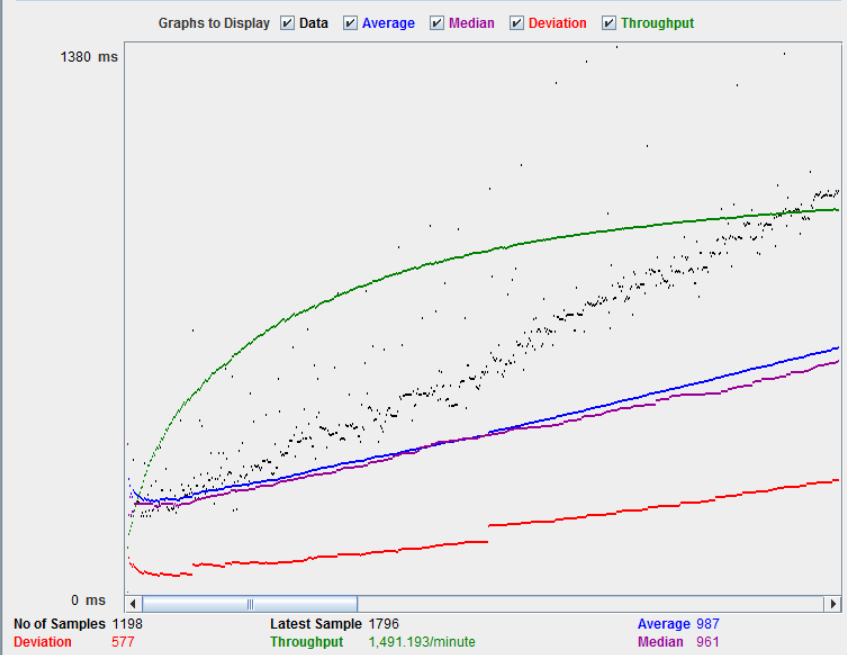
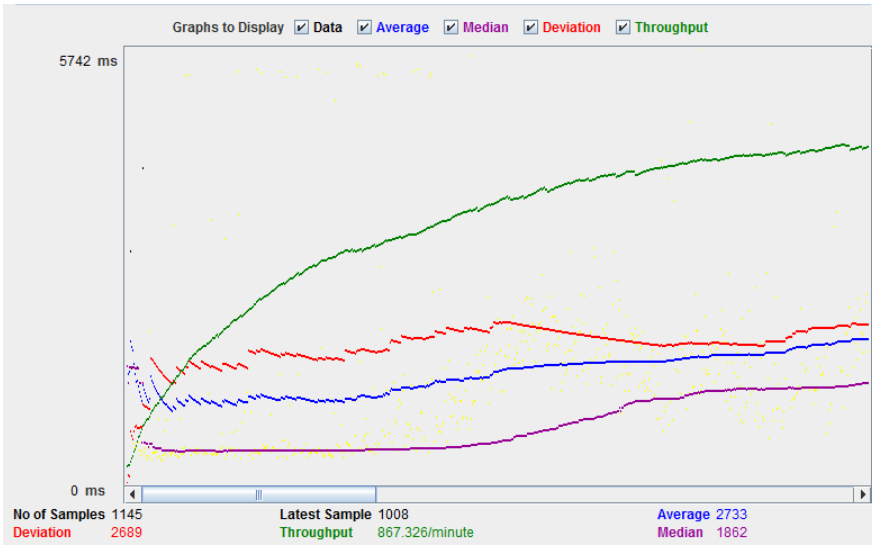
* Throughput
* Deviation

The **Throughput** is the most important parameter. It represents the ability of the server to handle a heavy load. The **higher** the Throughput is, the **better** is the server performance.

The **deviation** is shown in red – it indicates the deviation from the average. The **smaller** the **better**.

Tools we have selected for Performance Testing

**Jmeter:** Apache JMeter is pure Java-based open-source software designed to load test functional behavior and measure performance. You can use JMeter to analyze and measure the performance of web applications or a variety of services.

**Graph Results to analyze the performance between Server A & B**  
**Server A:**  
  
**Server B:**  
  
  
The throughput of a website under test Server B is 867.326/minute. This means this server handles 867.326 requests per minute, lower than Server A(1491.193/minute).  
  
The deviation is 2689, much higher than Server A(577). So we can determine the performance of this Server B is less than Server A.

## Unit Testing

Unit testing is a software testing method in which individual units or components of a software system are tested in isolation from the rest of the system to ensure that they are working as expected. The goal of unit testing is to identify and fix any defects or bugs in the smallest testable units of code, which are often functions or methods.

Unit tests typically involve writing test cases that cover a specific set of inputs and expected outputs for a given function or method. The test cases are executed automatically, often using a testing framework, and the results are compared to the expected outputs to determine if the function or method is behaving correctly.

Unit testing is an important practice in software development because it helps catch errors early in the development process, making them easier and cheaper to fix. It also provides a way to ensure that changes made to the code do not introduce new bugs or break existing functionality.

In general, unit tests should be considered a building block of all new functionality and should accompany all code changes.

### Measuring Unit Test Effectiveness

The effectiveness of unit testing should be measured by its ability to identify defects early in the development process and to prevent defects from being introduced into the code. The following are some methods that can be used to do this, but no one method should be used in isolation.

#### Code Coverage

Code coverage is a metric that measures the percentage of code that is exercised by unit tests. It can be used to identify areas of the code that are not covered by tests and may require additional testing. A high code coverage percentage indicates that most of the code has been tested and is less likely to contain defects.

Teams need to be very cautious about using code coverage alone to measure the effectiveness of unit tests. Code coverage can give a false sense of security that all the possible scenarios are being covered by the tests. Just measuring code coverage is not enough to guarantee the correctness of the program, the quality of the test cases, or the quality of the test code itself. A high code coverage can be achieved with poorly designed test cases. Critically, code coverage only measures the code execution and does not ensure that the tests cover all the business logic and requirements of the application. It is possible to have a high code coverage but still miss some critical scenarios.

**There is no one-size-fits-all answer to what percentage of code coverage strikes a good balance between effort involved in writing tests and the value/assurances those tests provide.** The ideal percentage of code coverage depends on various factors, including the complexity of the codebase, the criticality of the software, the risk tolerance of the organization, and the available resources. However, as a general guideline, it is often recommended to aim for a code coverage of at least 80% for critical parts of the application[[1]](#footnote-0)[[2]](#footnote-1).

#### Bug Density

Bug density measures the number of defects per unit of code. It can be used to track the number of bugs found in the code over time and to identify areas of the code that are more prone to defects. A lower bug density indicates that the code is more reliable and less likely to contain defects.

Bug density is a lagging measure for unit test effectiveness. It is a useful datapoint, but it does not provide immediate feedback to the developer during the development process, which can result in late detection of bugs. Additionally, it does not indicate the source of the bugs, whether they are due to programming errors, requirements errors, or other factors. And finally, it does not assess the quality of the tests or their effectiveness in finding bugs.

#### Test Failure Rate

Test failure rate measures the percentage of tests that fail during a testing cycle. It can be used to track the stability of the code and to identify areas of the code that are more prone to defects. A lower test failure rate indicates that the code is more stable and less likely to contain defects.

Test failure rate needs to be used with caution though, it is possible to have a low test failure rate and still have critical bugs that are not being caught by the tests. It only measures the number of failed tests and does not assess the quality of the tests or their ability to find bugs.

#### Defect Detection Rate

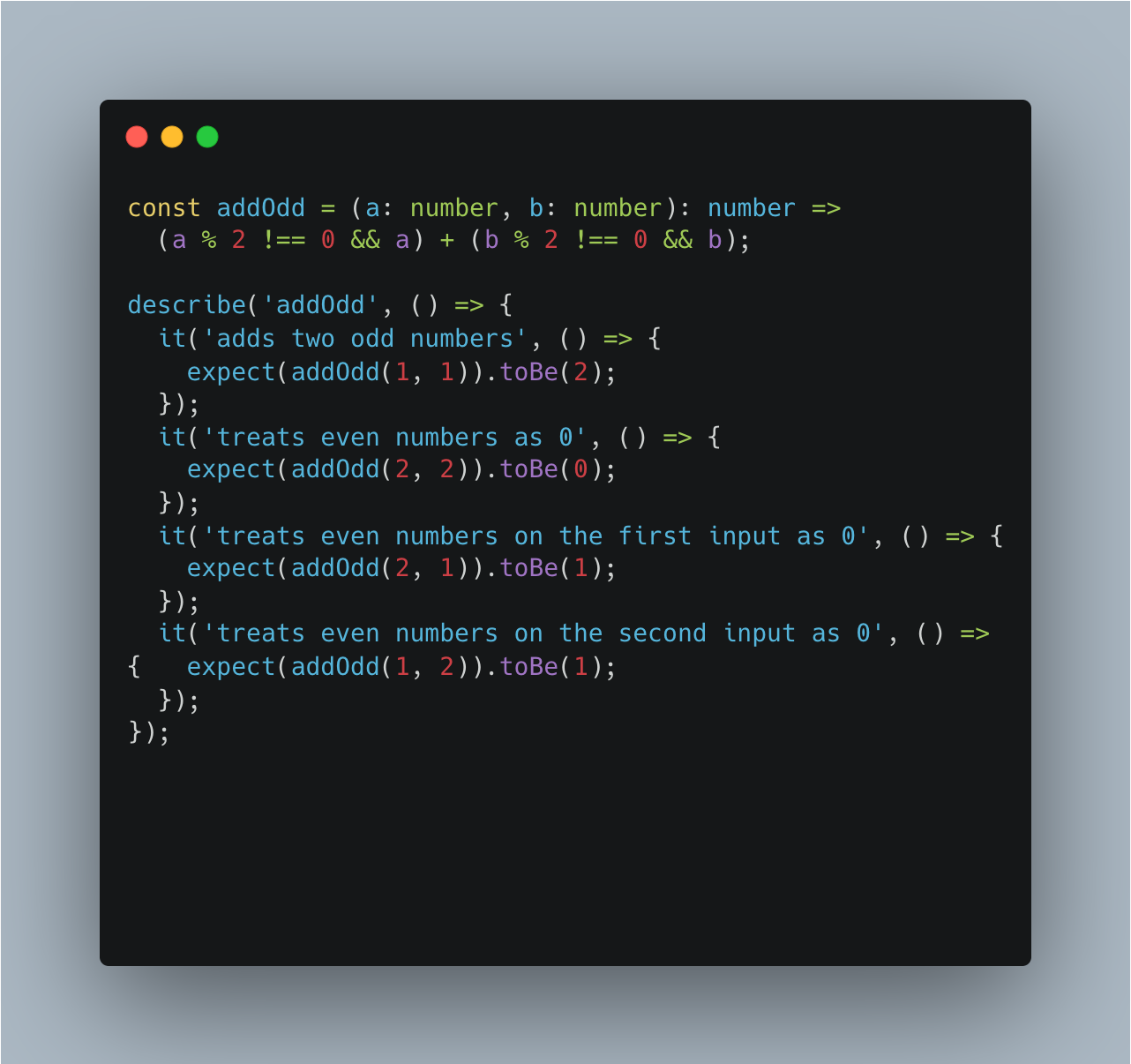
Defect detection rate measures the number of defects that are found and fixed during a testing cycle. It can be used to track the effectiveness of the testing process and to identify areas of the code that require additional testing. A higher defect detection rate indicates that the testing process is more effective at finding defects.

As with bug density, defect detection rate is a lagging measure. The same downsides apply here.

### Examples of Unit Tests

As mentioned above, a good unit test should operate on a method that has a finite set of inputs and outputs. The test should be isolated to a single unit of code, ideally a single method. The test should exercise both ends of a branch, that is both the true and false case of a condition. The test should avoid heavy reliance on mocked methods – this is a “smell” that indicates that the code under test may not be well suited to unit testing.

The following is a good example of a unit test. All branches in the code under test are covered and there’s no external methods being relied on to confirm behavior.



Here’s an example of code that wouldn’t be a good candidate for a unit test. The method simply wraps an f I/O operation and contains no other operations. The underlying methods could be mocked, but that only confirms that the mock works correctly and does not indicate whether or not the code is behaving as designed. In this case, an integration test would be more appropriate.



### Over Reliance on Unit Tests

While unit testing is a fundamental aspect of automated software testing, relying solely on unit tests poses some serious risks to development teams. Unit tests focus on testing individual components or modules of the software in isolation, they do not assess the behavior of the system as a whole and therefore may miss important integration issues or interactions between different components of the system. It’s important to remember that unit tests are written by developers, for developers, and they do not reflect the perspective of the end user. This can result in software that does not meet the needs or expectations of the users.

1. In the book "[Clean Code: A Handbook of Agile Software Craftsmanship](https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882#customerReviews)" by Robert C. Martin, the author recommends that developers aim for a code coverage of 80% or more for critical parts of the code. This guideline is mentioned in several places throughout the book, including chapter 9 on unit testing. Martin emphasizes that code coverage alone is not sufficient to ensure good test coverage, and that developers should focus on writing meaningful, effective tests that cover the most critical parts of the code. He also notes that the exact percentage of code coverage may vary depending on the specific application and that achieving 100% code coverage may not always be practical or necessary. [↑](#footnote-ref-0)
2. In "[Test-Driven Development: By Example](https://www.amazon.com/Test-Driven-Development-Kent-Beck/dp/0321146530)," by Kent Beck, the author emphasizes that the goal of testing should be to find and fix defects, and that the focus should be on writing effective, meaningful tests that cover the most critical parts of the code. In chapter 18, Beck suggests that a good target for code coverage is between 80% and 95%, with the understanding that achieving 100% code coverage is often difficult and not always necessary. He notes that the exact percentage of code coverage may vary depending on the specific application and that developers should use their judgment to determine the appropriate level of code coverage for their project. [↑](#footnote-ref-1)