BY ORDER OF THE COMMANDER
SPACE SYSTEMS COMMAND

SPACE SYSTEMS COMMAND
HANDBOOK 63-150

12 DECEMBER 2024

Acquisition

UNIT TESTING FOR LEGACY CODE

ACCESSIBILITY: Publications and forms are available for downloading or ordering on the e-Publishing website at **www.e-Publishing.af.mil**.

RELEASABILITY: There are no releasability restrictions on this publication.

OPR: HQ SSC/S6

Certified by: SSC/S6
(Dr. Daniel Veit)
Pages: 15

This instruction implements DAFI 63-101/20-101, *Integrated Life Cycle Management*, and provides guidance consistent with DoD Instruction 5000.87, *Operation of the Software Acquisition Pathway*, in terms of the need for CI / CD pipeline implementation, in which unit testing plays a critical role. This handbook discusses the benefits of unit testing and how legacy code can benefit from proper implementation of these modern software practices. This publication applies to all SSC commands, including SSC-gained Air National Guard, Air Force Reserve Command, and United States Air Force members. Refer recommended changes and questions about this publication to the OPR using DAF Form 847, *Recommendation for Change of Publication*; route DAF Forms 847 from the field through the appropriate functional chain of command. Ensure that all records created as a result of processes prescribed in this publication are maintained IAW AFI 33-322, *Records Management and Information Governance Program,* and disposed of IAW Air Force Records Information Management System Records Disposition Schedule. This publication may be supplemented at any level, but all supplements must be routed to the OPR of this publication for coordination prior to certification and approval. The use of the name or mark of any specific manufacture, commercial product, commodity, or service in this publication does not imply endorsement by the DAF.

Chapter 1

DEFINING UNIT TESTING AND HOW IT CAN BE APPLIED

1.1. BLUF. Unit testing is a critical step for software development, and this document provides best practices for how it can be done on both newly developed code and legacy code.

1.2. Abstract. Unit testing is a critical part of the software development process that should be strongly supported by the leadership of a development team. Among other things, it helps to improve code quality, allows for many faults in the code to be found early in the development cycle, and it helps prepare isolated code modules for re-use both within the existing system and in other systems. Unit testing on brand new development is relatively straight forward to do, but most developers are not developing brand new products. They are refining, updating, or extending existing code, also known as legacy code. Unit testing on legacy code is a more challenging process, and this document provides a few techniques for how to accomplish this.

1.3. What Is Unit Testing and What Are Unit Tests?

1.3.1. Unit testing is the process of testing the smallest functional unit of code. This is typically done by the developer who writes the code. Software testing, in general, helps ensure code quality, and unit testing acts as the first phase of verifying a new piece of software works as expected. It is a best practice in software development to write software as small, functional units, then write a unit test for each of these units. A developer can first write unit tests as code, then run that test code automatically every time changes are made to that code. This way, if a test fails, the developer can quickly isolate the area of the code that has the bug or error. Unit testing enforces modular thinking paradigms and improves test coverage.

1.3.2. Some of the primary objectives of unit testing include:

- Verifying the correctness of code.
- Isolating a more precise location of bugs within a piece of code.
- Fixing bugs early in the development cycle.
- Improving code reuse.

1.3.3. Some of the benefits of unit testing include:

- The developer obtains immediate feedback on work (efficient bug discovery).
- It is the only testing that is assuredly white box.
- Code coverage can be more easily measured and optimized at this stage.
- Early testing is more cost effective.
- Unit tests act as a form of documentation.

1.3.4. A *unit test* is a block of code that verifies the accuracy of a small, isolated block of application code, typically a function or method. The unit test is designed to check that the block of code runs as expected, according to the developer's logic behind it. The unit test is

only capable of interacting with the block of code via inputs and captured asserted (true or false) output.[1]

1.3.5. A single block of code may also have a set of multiple unit tests. A complete set of unit test cases (referred to as a *test suite*) cover the full expected behavior of the code block. When a block of code requires other parts of the system to run, a developer cannot use a unit test with that external data. The unit test needs to run in isolation.[2] Other system data, such as databases, objects, or network communication, might be required for the code's functionality. If that is the case, the developer will need to use data stubs, drivers, or mockups instead. Once the unit tests have been created, they should be automated so that they can quickly run on demand each time changes are made to the that block of code.[3]

1.4. What Is the Difference Between Unit Testing and Other Types of Testing?

1.4.1. There are many other types of software test methods alongside unit testing, and they each have specific roles to play in the software development lifecycle. Understanding the purpose behind each type of test will help developers keep the proper context of the test cases they write. It was mentioned that unit tests verify the smallest code function or isolated block of code. From that point, as independently verified blocks of code grow in number and are ready for integration, other levels of testing will be required:

- **Integration testing** checks that different parts of the software system that are designed to interact do so correctly.
- **Functional testing** checks whether the software system passes the software requirements outlined before building.
- **Performance testing** checks whether the software runs to expected performance requirements, such as speed and memory size.
- **Acceptance testing** is when the software is tested manually and/or operationally by stakeholders or user groups to check whether it is working as they anticipate.
- **Security testing** checks the software against known vulnerabilities and threats. This includes analysis of the threat surface, including third-party entry points to the software.

1.4.2. These testing methods usually require specialized tools and independent processes to check the software, and they are typically performed after basic application functionality has been developed. In contrast, unit tests run every time the code builds. They can be written as soon as any of the code they are testing is written and do not require any special tools to run, although there are unit testing frameworks (such as TestNG or JUnit for Java) that can improve unit testing development and automation.

1.5. How Unit Tests Work.

1.5.1. The creation of unit tests consists of four primary phases:

1. Planning and setting up the environment. During this phase, developers will consider which units in the code need to be tested and how to execute all relevant functionality to

[1] (Amazon Web Services, Inc., 2024)
[2] (Feathers, 2004)
[3] (Geeks for Geeks, 2024)

test it effectively.

2. Writing the test cases and scripts. Developers design scenarios that confirm the functionality of the unit and check that edge cases are handled.

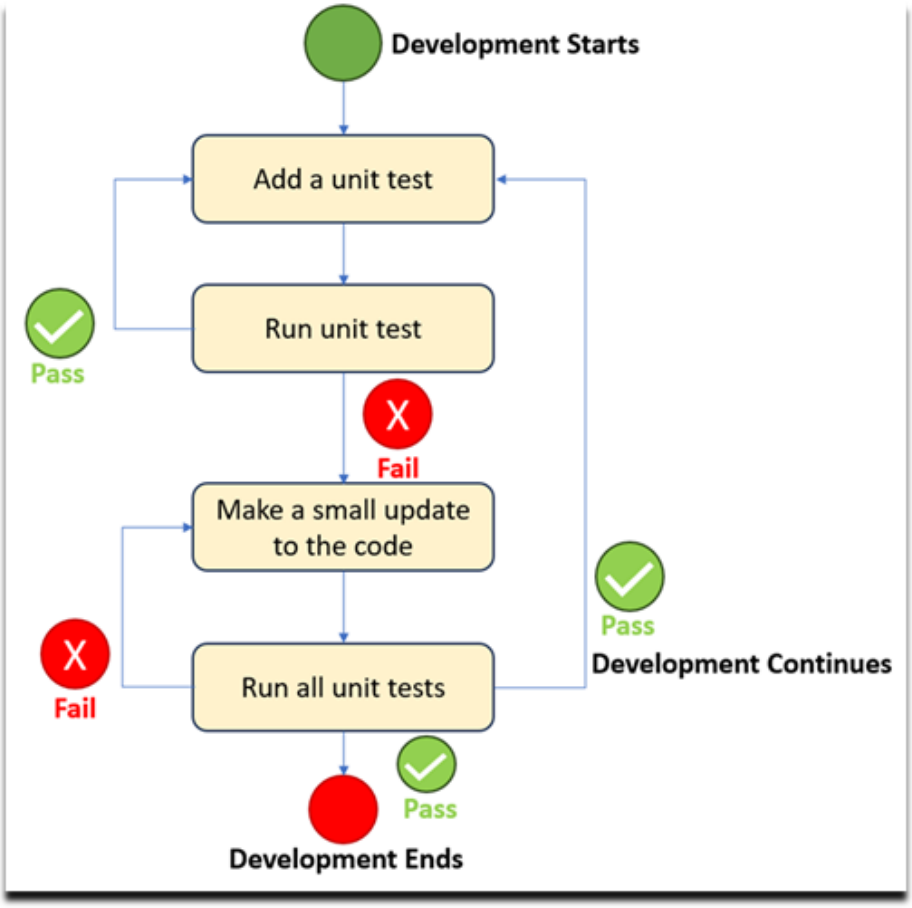
3. Executing the unit tests. Once executed, each test will result in a pass or failure representing the behavior of the unit with the designed scenario.

4. Analyzing the results. Errors or issues with the code can be identified and resolved at this point.

1.5.2. Test-driven development (TDD) is a common coding approach to support unit testing. TDD requires the developer to create the unit test first based on the requirement or user story that has been provided. The unit test is created before the application code exists so, naturally, the initial test will fail. Then the developer adds the functionality to the application until the test passes. This process tends to result in a high-quality codebase.[4]

1.5.3. **Figure 1.1** shows a typical workflow for unit testing using the xUnit testing framework. The xUnit framework is used with .NET development, but the process shown could be used in nearly any development environment.

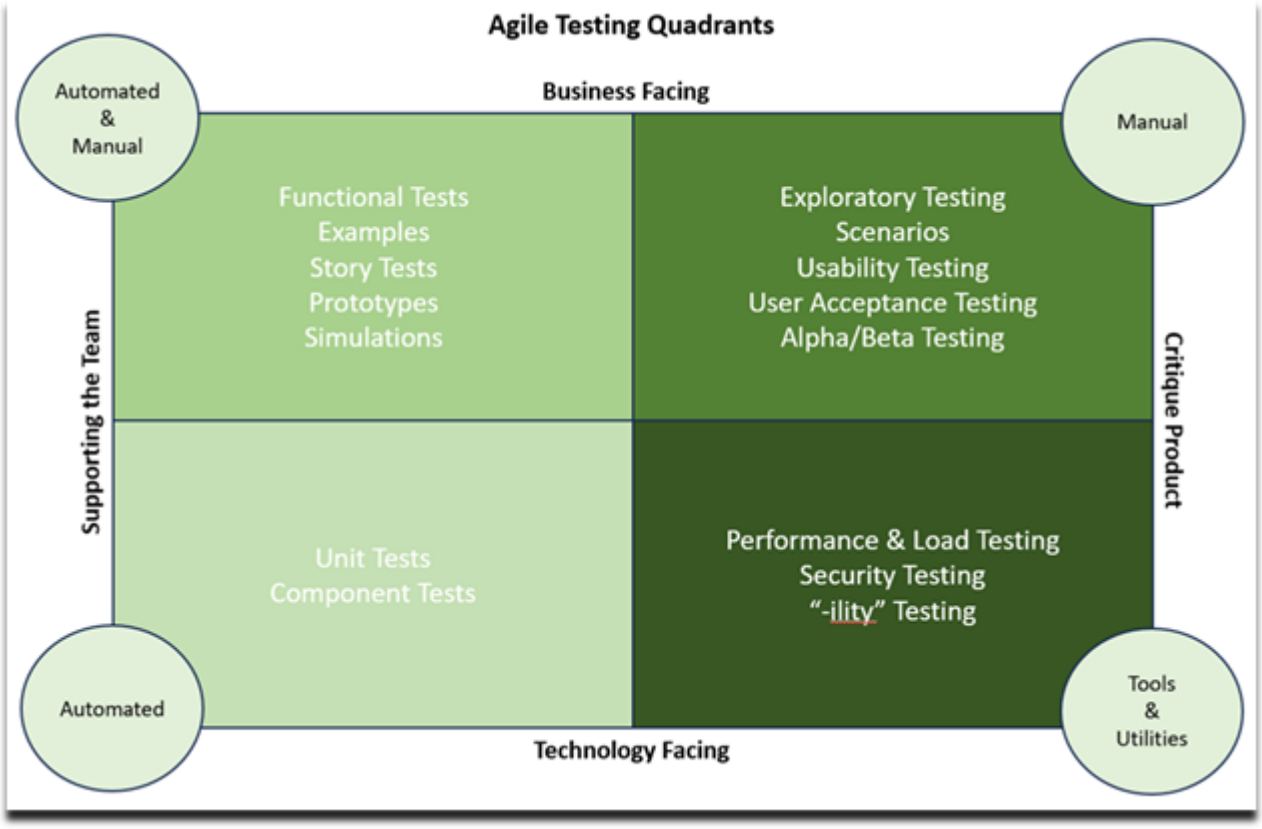
Figure 1.1. Unit Testing Flow Diagram[5].



[4] (Bakharev, 2023)
[5] (Verma, 2017)

1.5.4. Unit testing frameworks are used to create automated test scenarios. The developer implements a framework for automation by incorporating criteria into the test to verify the accuracy of the code. The framework records failures during the execution of the test cases. Depending on the severity of a failure, the framework may stop testing and require immediate resolution to the errors.[6] If the unit tests are embedded within the code itself, developers will comment out the unit tests (or remove them altogether) just prior to deployment.[7] Due to the repetitive and simple nature of unit tests, it is a best practice to automate their use as much as possible. Based on the Agile Test Quadrants diagram developed by Brian Marick and Lisa Crispin (shown in **Figure 1.2** below), teams should strive to fully automate unit tests.[8] As shown in the Figure, unit testing takes place in the bottom left quadrant in an area that is both technology facing and directly supporting the team.

Figure 1.2. Agile Testing Quadrants[9].



1.6. Code Coverage Techniques with Unit Testing.

1.6.1. When designing unit tests, there are five parts of code coverage that developers should take into consideration when making sure their code is fully covered by the unit tests being written. Those five parts are:

[6] (Infotek Solutions, 2023)
[7] (BasuMallick, 2022)
[8] (Crispin, 2011)
[9] (Thomas, 2024)

1. Branch
2. Statement
3. Function
4. Condition
5. Path

1.6.2. **Branch coverage** in unit testing is a metric that measures the percentage of branches (or decision points) in the source code that have been executed during the testing process.[10] It indicates how well the test cases navigate through different possible outcomes of conditional statements (for example, "if/else" statements). A high branch coverage means that most decision paths in the code have been tested.

- *Branch Coverage = (Number of Executed Branches) / (Total Number of Branches) ✕ 100%*

1.6.3. The **statement coverage** technique is used to design test cases for white box testing which includes the execution of all the paths, lines, and statements of source code. It is also used to design test box cases where it will find out the total number of executed statements out of the total statements present in the code. With respect to unit testing, statement coverage gives developers a way to track how much of their unit code is being traversed by their test cases. Keep in mind that even with 100% code coverage at the statement level it does not guarantee there will be no bugs or security issues in the code. The quality of the test cases, the use of syntax, and the combination of inputs creating unique outputs are all examples where even full code coverage can allow for bugs or vulnerabilities to seep into the software.

- *Statement coverage = (Number of executed statements / Total number of statements in source code) ✕100%*

1.6.4. **Function coverage** measures the extent to which the functions present in the source code are covered during testing. All functions that are in the source code have unit tests written for them, and they should be tested for varying values.[11]

- *Function coverage = (Number of Executed Functions) / (Total Number of Functions) ✕ 100%*

1.6.5. In source code where we have a condition, the result would be a Boolean value of either true or false. **Condition coverage** establishes if the unit tests cover both true and false values. In the source code, when each occurring condition is evaluated for both true and false states, then the condition coverage for the code is said to be complete.

- *Condition coverage = (Number of executed operands) / (Total number of operands) ✕ 100%*

1.6.6. **Path coverage** measures the number of full path options covered by testing in each piece of code under test. This takes both the branch view and the condition view into account to fully cover each path sequence from start to finish. Depending on how the path sequence is

[10] (Geeks for Geeks, 2024)
[11] (Sruthy, 2024)

constructed, the time it takes to fully test every path could take too long if there are a high number of potential combinations. For example, some loops will have a wide variety of input options that require the loop to be run so many times that it will slow down the testing process too much while not returning much value. In situations like this, it is best to keep the testing focused on scenarios most representative of how the loops might be executed and only write test cases that have boundary limits.[12] This approach combined with the branch, statement, and condition approaches should provide proper coverage at the unit test level.

- *Path coverage = (Number of paths covered) / (Total number of paths) ⨯100%*

1.7. Challenges with Unit Testing.

1.7.1. There are several common challenges and even disadvantages to having a robust unit testing approach. One such challenge is that the process can be quite time-consuming for writing unit tests. Many developers will want to skip out on fully covering their new code with unit tests to focus on more functionality or to meet a deadline. This challenge should be addressed by planning early to have time for unit test creation baked into the development schedule. Having proper unit tests written can also be a part of the code review process where code cannot be promoted to integrated environments unless there are adequate unit tests included and those tests are passing.

1.7.2. Another challenge for development teams is to understand that unit testing should not be expected to cover all the errors in each module because there is a likely chance of finding new errors in that module while doing integration testing. The expectation here is that unit testing should not be considered as providing sufficient coverage for functional or integration testing. Unit testing has a specific purpose, and it should not be counted on to extend beyond that purpose. This challenge also holds for finding errors in the user interface (UI) where unit tests have limited visibility, as well as covering non-functional testing parameters such as scalability, extendibility, and system performance.

1.7.3. There can also be an over-reliance on test automation results. Over-reliance on automated unit tests can lead to a false sense of security, as automated tests may not uncover all possible issues or bugs. While unit tests should be automated as much as possible, manual code reviews and audits of the unit test cases can be done to make sure appropriate amounts of code coverage are being attained and that critical use cases are not being ignored by the automation suite.

[12] (Gay, 2016)

Chapter 2

UNIT TESTING WITH RESPECT TO LEGACY CODE

2.1. Best Practices.

2.1.1. Many developers will spend more time enhancing, extending, modifying, and fixing existing software applications than they will coding new applications from scratch. Because of this, it is helpful to take what we have learned about unit tests in the first part of this document and see how much we can apply to legacy code (i.e., existing code) that is being updated. In many cases, this legacy code will not have a suite of automated test cases (unit or otherwise) to go along with it. This makes the testing of legacy code that much more challenging. In scenarios where the legacy code is substantially bulky or convoluted, traditional unit testing may not work or may not be worth the return on investment it takes to design and create the test cases.[13] In such cases, it is more beneficial to focus on activities like:

- Get the build and existing tests working
- Run static analysis of code
- Test by functional division

2.1.2. To get the system build and existing tests (if any) working first, a team must evaluate the current state of the codebase. Identify which tests already exist, what code is covered by those tests, and when do those tests run. If certain tests are failing, determine why they are failing. Once all existing tests are fully understood and passing, the team can begin writing new tests. This process should start with very small unit tests covering the changes being made to the legacy code. Attempting to go through and write new unit tests for the unchanged legacy code should not be done.[14] Instead, running static analysis scans (SAST), creating mocks to mimic integration points, and designing higher level functional and integration tests should suffice to cover those parts of legacy code that *are not* being updated.

2.1.3. For those parts of legacy code that *are* being updated, it is a best practice to first look for opportunities to refactor the code. This could include removing outdated or unused pieces of code or splitting up monolithic or tightly coupled actions into separate, decoupled methods or function calls. Activities like this not only make the code more extensible and easier to update in the future, but it can also make the inclusion of unit tests easier and more accurate.[15]

2.1.4. Another approach for validating legacy code is to create a suite of smoke tests to be used to quickly identify those high-level parts of the application that are failing. Smoke testing covers a wide array of primary use cases for an application but will not dig into secondary or edge cases. These suites are just looking to see if the new build is stable. Typically, a build that passes the smoke tests can be promoted to a QA team for more in-depth testing. In our situation, failures in the smoke testing may give the developers clues on where the legacy code should be updated first. Smoke test suites may include some of the existing legacy test cases mentioned earlier as well as brand new test cases. The suite should also include unit tests for all public methods or functions in each area of the application to verify those methods can be

[13] (Eugene S., 2022)
[14] (Eugene S., 2022)
[15] (Vogel, 2021)

called and are returning acceptable values. The goal here is to quickly identify parts of the application that need priority attention (or confirm that everything is running as expected).

2.1.5. A best practice for making changes to and testing legacy code is to start testing from the outside of the code, then refactor from the inside.[16] The "outside of the code" would be at the highest functional level the team needs to begin their verification. This is an easier way to write the initial smoke tests, and it keeps the developer from tying tests too closely to the implementation details. Some experts refer to these types of unit test cases as Approval tests or Snapshot tests, but the end goal is like what we described above with smoke testing. When digging into legacy code, the team needs to quickly build a reliable test suite that checks primary functionality and lets the developers know instantly when and where something has gone wrong.

2.1.6. Once the team has determined enough of the code has been covered by these high level test cases, the code can begin to be refactored. The simplest way to do this is to tackle small chunks of code from the inside. Because the test cases are at a higher level, developers have more flexibility to create new classes, methods, or decision trees as needed. The tests do not get in the way, but they still verify the overall behavior. As code is being refactored, improved unit tests can be written to validate the new pieces.

2.1.7. When none of the above approaches are enough, another tactic for improved unit testing of legacy code is to incorporate mocks. Mocks are objects pre-programmed with expectations which form a specification of the calls they are expected to receive.[17] New unit tests should be written to cover new changes made in the legacy code, and mocks should be created to cover (or simulate) the legacy code that is untouched.[18] This is most effective when the parts of the application the developer does not change still work as expected. The benefit of using mocks in this scenario is that mock objects can help developers separate units of code and test them in isolation without relying on or worrying about their dependencies. The challenge with using mocks in this manner is that they stay as close to the original functionality they are simulating as possible without drifting into something different. It does no good to develop mocks that pass their test cases, but that do not properly simulate the actual integration points being verified.

2.1.8. Each of these methods will help developers improve their ability to validate the changes they make to legacy code. These techniques do require more time up front, and they can initially slow down the development process, so it is important that this extra time be included into any estimates the team is making with respect to timelines or turnaround expectations.

[16] (Carlo, 2024)
[17] (Fowler, 2007)
[18] (Vogel, 2021)

2.2. Conclusion.

2.2.1. Unit testing is a well-known piece of the software development lifecycle and is a relatively straightforward task for developers to accomplish when creating new applications. However, many developers are not creating brand new applications but are instead working with legacy codebases that were probably developed by other teams. In cases such as these, especially when there are no unit tests to rely on, there are alternate methods for making changes to the legacy code while employing best practices for both unit test creation and general code refactoring. This document has attempted to describe some of these best practices at a very high level, but new methods relying on more advanced tool sets are always on the rise. Development teams must prioritize unit test creation appropriately and be open to creating new ways to improve their unit testing especially as it relates to legacy code.

Craig E Frank, Colonel, USSF
SSC/S6 Director of Communications

Attachment 1

GLOSSARY OF REFERENCES AND SUPPORTING INFORMATION

References

AFI 33-322, *Records Management and Information Governance Program*, 23 March 2020

DAFI63-101/20-101, *Integrated Life Cycle Management*, 16 February 2024

DoDI 5000.87, *Operation of the Software Acquisition Pathway*, 2 October 2020

Academic References

Amazon Web Services, Inc. (2024). *What is Unit Testing?* Retrieved from AWS What Is: **https://aws.amazon.com/what-is/unit-testing/**

Bakharev, N. (2023, July 26). *Unit Testing: Definition, Examples, and Critical Best Practices.* Retrieved from Bright Security: **https://brightsec.com/blog/unit-testing/**

BasuMallick, C. (2022, September 20). *What Is Unit Testing? Types, Tools, and Best Practices.* Retrieved from Spice Works: **https://www.spiceworks.com/tech/devops/articles/what-is-unit-testing/**

Carlo, N. (2024). *The best way to start testing untested code.* Retrieved from Understanding Legacy Code: **https://understandlegacycode.com/blog/best-way-to-start-testing-untested-code/**

Crispin, L. (2011, November 8). *Using the Agile Testing Quadrants.* Retrieved from Agile Testing with Lisa Crispin: **https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/**

Eugene S. (2022, July 14). *Testing Legacy Codebase: What is, Common Problems and Best Practices.* Retrieved from ModLogix by Langate: **https://modlogix.com/blog/testing-legacy-codebase-what-is-common-problems-and-best-practices/**

Feathers, M. (2004). *Working Effectively with Legacy Code.* Hoboken: Pearson.

Fowler, M. (2007, January 2). *Mocks Aren't Stubs.* Retrieved from Martin Fowler: **https://martinfowler.com/articles/mocksArentStubs.html**

Gay, G. (2016). *Path Coverage - CSCE 747 (Spring 2016).* Retrieved from The University of Edinburgh: **https://www.inf.ed.ac.uk/teaching/courses/st/2017-18/Path-coverage.pdf**

Geeks for Geeks. (2024, April 23). *Unit Testing - Software Testing.* Retrieved from Geeks for Geeks: **https://www.geeksforgeeks.org/unit-testing-software-testing/**

Geeks for Geeks. (2024, February 29). *What is Branch Coverage in Unit Testing?* Retrieved from Geeks for Geeks: **https://www.geeksforgeeks.org/what-is-branch-coverage-in-unit-testing/**

Infotek Solutions. (2023, August 8). *What is automated approach to perform Unit Testing.* Retrieved from Medium: **https://infoteksolutions.medium.com/what-is-automated-approach-to-perform-unit-testing-ca7df8a34e73**

Sruthy. (2024, March 7). *Code Coverage Tutorial: Branch, Statement, Function Coverage.* Retrieved from Software Testing Help: **https://www.softwaretestinghelp.com/code-coverage-tutorial/**

Stephan, F. (2019, February 12). *What is the Agile Testing Quadrant?* Retrieved from Kaizenko: **https://www.kaizenko.com/what-is-the-agile-testing-quadrant/**

Thomas, A. (2024, April 17). *Agile Testing Quadrants: Concept & How to Use it.* Retrieved from Test Sigma: **https://testsigma.com/blog/agile-testing-quadrants/**

Verma, S. (2017, October 5). *Unit Testing using xUnit.* Retrieved from Systems +: **https://systemsplusgroup.blogspot.com/2017/10/unit-testing-using-xunit.html**

Vogel, P. (2021, September 23). *Unit Testing Legacy Code, Part 1: Creating Maintainable Applications.* Retrieved from Telerik: **https://www.telerik.com/blogs/unit-testing-legacy-code-part-1-creating-maintainable-applications**

Prescribed Forms

None

Adopted Forms

DAF Form 847, *Recommendation for Change of Publication*

Abbreviations and Acronyms

SAST—static analysis scans

SSC—Space Systems Command

TDD—Test-driven development

UI—user interface

Terms

Acceptance Testing—When software is tested manually and/or operationally by stakeholders or user groups to check whether it is working as they anticipate.

Agile (software development)—Refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

Boolean—A variable that can be either true or false.

Branch—A segment of code that begins from a decision point in the code where the execution can take a different path from the previous execution flow.

Bug—A coding mistake that causes unintended output.

Build (a software build)—A version of a program that, as a rule, is a pre-release version and is identified by a build number rather than by a release number. Simply put, a software build is a set of executable code that is ready for use by customers.

Driver—Drivers serve the same purpose as stubs, but drivers are used to replicate the "calling program" used in integration testing.

Edge Cases—A scenario designed to check that irregular inputs do not cause test failures.

Functional Testing—Checks whether the software system passes the software requirements outlined before building.

Integration Testing—Checks that different parts of the software system that are designed to interact do so correctly.

Legacy Code—Existing computer source code that is no longer actively maintained by a development team. Typically, this code may be obsolete, but for the purposes of this document, legacy code can still be used in support of current applications, but only for the benefit of not re-writing working features.

Mock—If a block of code has external dependencies, a mock is an object that can be used to simulate a dependency to isolate the code block.

Performance Testing—Checks whether the software runs to expected performance requirements, such as speed and memory size.

Refactor—Changing the structure of code without changing the behavior, usually to improve simplicity and/or efficiency.

Smoke Testing—Preliminary testing or sanity testing to reveal simple failures severe enough to, for example, reject a prospective software release.

Static Application Security Testing (SAST)—A white-box testing tool that analyzes an application's source code, binary, or byte code to identify security vulnerabilities. SAST is used early in the software development lifecycle (SDLC) to help secure software and remediate security flaws before the code is compiled.

Security Testing—Checks the software against known vulnerabilities and threats. This includes analysis of the threat surface, including third-party entry points to the software.

Statement—A line of code in a computer programming language that instructs a task to be performed.

Stub—Example variables used in testing to simulate scenarios. Stubs are like drivers, but stubs are used to replicate the “called programs” in integration testing.

Test Coverage—How much of a project is tested on by unit tests.

Test-driven Development (TDD)—A software development practice that emphasizes writing tests before writing the actual code. It follows a cyclical process of writing a failing test, writing the minimum code to make the test pass, and then refactoring the code.

Test Suite—A collection of test cases, scripts, and data used to evaluate a software application or system's functionality and performance, typically grouped according to functional or application area.

Unit Testing—A test designed to confirm that a small block of code operates as intended by the developer.

White Box Testing—Testing with the goal of ensuring the correctness of the internal structure of code as opposed to the correctness of the system's functionality (which is referred to as Black Box Testing).