

# METHODS OF ENHANCING UNIT TEST QUALITY FOR RELIABLE CODE

**Nikhil Badwaik**

Software Engineer at NIKE INC Portland OR, USA

## Abstract

In modern software engineering, unit testing plays a central role in maintaining the reliability and quality of the code. In light of this, the article "Methods for Improving Unit Tests to Improve Code Reliability" highlights various strategies and practices aimed at optimizing the unit testing process. Based on extensive research and many years of experience, methods such as improving the independence of tests, increasing their determinism and readability, as well as the introduction of test automation techniques are presented. Important aspects are discussed, including test-driven development (TDD), test coverage analysis, and component isolation. The purpose of the article is to provide readers with a comprehensive set of recommendations for creating faster, clearer and more reliable tests, which ultimately contributes to the development of better software.

**Keywords** Unit tests, tests, code reliability, programming, code writing.

## INTRODUCTION

In the context of software development, unit testing occupies a key position. Incorrectly designed test suites can lead to code instability, unrecognized bugs, and significant debugging time. It also makes supports complexity, scalability, redundancy, uncertainty of results and insufficient test coverage [1].

In turn, the relevance of the topic is due to the increasing demands on software quality as well as the increasing complexity of systems, which requires a deeper approach to testing. The problematic issue is that traditional unit testing methods often do not cover all possible use cases

and boundary cases, which leads to hidden defects and decreased confidence in the system.

### 1. General characterization of unit testing

Unit testing is a key aspect of software development that allows individual components - often called modules, such as functions, procedures, or methods - to be thoroughly tested. This process touches each element of a program to ensure that it performs its intended functions correctly. The goal of unit testing is to detect bugs early, which facilitates their timely correction, preventing possible future complications of problems (Fig.1.).

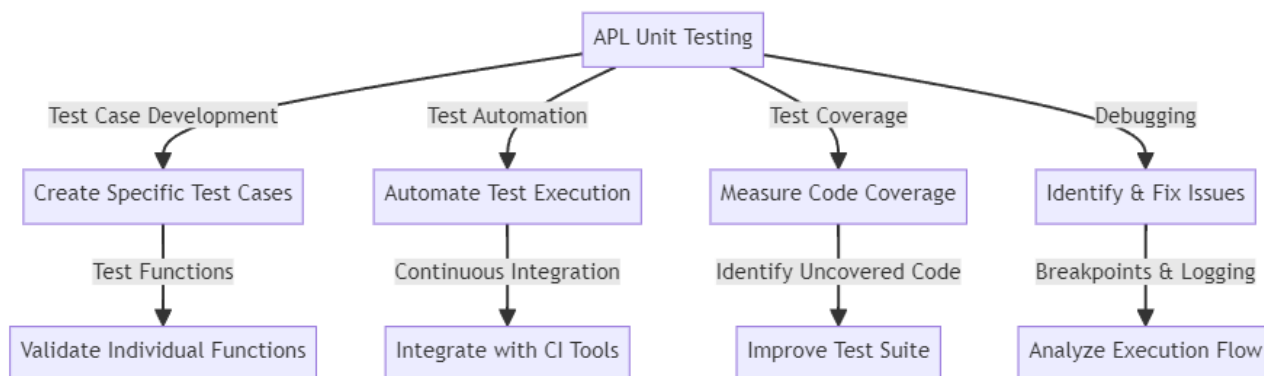


Fig.1. Unit testing [2].

Unit testing in APL programming language characterized by its short syntax is a process of testing separate code components, most often functions or procedures. This method provides independent operability of each part of the code by the declared requirements.

The main element of unit testing in APL is the creation of test functions. These functions are specialized programs designed to test individual features or procedures in the project.

The key stage of unit testing in APL is the development of various test scenarios that simulate various input conditions for analyzing the code functionality.

To illustrate this, we can use a set of test cases for

the addition function, where each case represents a combination of input data and expected results [2].

In the area of unit testing, developers can choose between manual and automated approaches to verify that each module of a software product meets customer requirements. Manual testing, which involves executing tests without automation tools, can become time-consuming, especially in test repetition and preparation. However, complete automation is also unattainable and some level of manual intervention will always be necessary.

Consider the following aspects that improve the efficiency and reliability of unit testing (Table 1.).

**Table 1. Aspects designed to improve the efficiency and reliability of unit testing.**

Aspects of Improving the Efficiency and Reliability of Modular Testing	Description
Isolation of Tested Components	The fundamental principle of modular testing is the strict isolation of the tested component from the rest of the system and external dependencies. This approach allows accurately determining and testing the functionality of each component without external influences, helping to identify and fix errors directly in the

	component itself rather than in its interactions with other elements of the system. To achieve isolation, mocks and stubs imitating the behavior of external systems are widely used.
Repeatability and Stability of Results	It is critically important to ensure that each test run under the same conditions leads to the same results. This repeatability is crucial for regression testing, which checks if changes in one part of the system have caused failures in already verified functions. The stability of test results in different environments (development, testing, production) is also critical for confirming the reliability of the software product.
Testing Small Code Segments	The efficiency of modular testing increases by focusing efforts on small, well-defined sections of code. This allows for detailed verification of each module's functionality before their integration into larger systems. This approach simplifies the search and fixing of errors, making test cases more manageable and understandable.
Test Automation	Automation significantly speeds up the development and quality control process. Automatic execution of modular tests during code changes provides immediate feedback on the impact of changes on overall functionality. The introduction of automation helps maintain the reliability of tested systems and supports a high level of software quality [3].

**2. Practices for effective unit testing**

Unit testing, when properly applied, is a powerful tool for improving software quality. To maximize

its potential, it is necessary to follow a number of best practices that can significantly improve the efficiency of the process (Table 2.).

*Table 2. Effective unit testing practices.*

<b>Practices</b>	<b>Description</b>
Optimization of Modular Tests	Each modular test should be compact and focused, limited to checking a single specific functionality or aspect of behavior. This ensures clarity in understanding the test and simplifies maintenance. It also helps in the precise localization of errors if the test fails.
Independence and Repeatability of Tests	Tests should be designed so that their results do not depend on each other and can be reproduced with the same results in multiple runs. This guarantees the stability and reliability of the testing environment.
Sufficient Test Coverage	It is necessary to strive for maximum possible code coverage with tests, which will help identify potential problems in most parts of the application. While absolute test coverage may be unattainable, a high coverage percentage is a

	desirable goal.
Use of Mocking	Mocking allows the imitation of real object behavior in a controlled test environment. This is ideal for isolating the tested code from external dependencies and side effects, making testing more focused and reliable.
Use of Automated Testing Tools	Test automation using specialized tools can significantly improve the testing function by providing automatic test detection, detailed reports generation, and integration with existing development environments.
Choosing the Right Platform for Modular Testing	Effective modular testing requires the use of reliable platforms such as JUnit for Java, PyTest for Python, and NUnit for .NET. These tools offer advanced capabilities for result verification, test execution, and test scenario management [4,5].

**3. Advanced Python unit testing concepts**

After familiarizing ourselves with the basics of unit testing, let's move on to examine more advanced methods and techniques that can significantly improve the efficiency and reliability of tests, especially in complex or specific situations [6].

1. Testing code that depends on command line

arguments

Code that depends on command-line arguments can be challenging to unit test since these arguments are usually provided directly by the operating system. However, with `unittest.mock.patch`, these inputs can be effectively mimicked in tests. Suppose there is a function that analyzes command line arguments:

```
import argparse

def parse_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument('--path', type=str, required=True)
    args = parser.parse_args()
    return args

def main():
    args = parse_arguments()
    print(f"Path provided is {args.path}")
```

To test this function, use `unittest.mock.patch` to simulate `sys.argv` as shown below:

```
import unittest
from unittest.mock import patch
from your_module import main, parse_arguments
```

```
class TestCommandLineArguments(unittest.TestCase):
    @patch('sys.argv', ['program_name', '--path', 'test_path'])
    def test_parse_arguments(self):
        args = parse_arguments()
        self.assertEqual(args.path, 'test_path')

    @patch('sys.argv', ['program_name', '--path', 'test_path'])
    def test_main(self):
        with patch('builtins.print') as mocked_print:
            main()
            mocked_print.assert_called_with('Path provided is test_path')

if __name__ == '__main__':
    unittest.main()
```

2. Testing code that depends on environment variables
- Environment variables are often used to customize the behavior of applications. To unit test such customizations, use `unittest.mock.patch.dict` to temporarily modify environment variables:

```
import os
import unittest
from unittest.mock import patch

def get_database_url():
    return os.getenv("DATABASE_URL")

class TestEnvironmentVariables(unittest.TestCase):
    @patch.dict(os.environ, {"DATABASE_URL": "test_url"})
    def test_database_url(self):
        url = get_database_url()
        self.assertEqual(url, "test_url")

if __name__ == '__main__':
    unittest.main()
```

3. testing code that outputs information to the console
- Sometimes it is important to test that the application correctly outputs the correct information to the console. This can be tested by

redirecting sys.stdout to a string buffer and analyzing its contents:

```
import io
import sys
import unittest

def greeting():
    print("Hello, world!")

class TestPrintFunction(unittest.TestCase):
    def test_greeting(self):
        captured_output = io.StringIO()
        sys.stdout = captured_output
        greeting()
        sys.stdout = sys.__stdout__
        self.assertEqual(captured_output.getvalue().strip(), "Hello, world!")

if __name__ == '__main__':
    unittest.main()
```

These advanced techniques allow testing different aspects of code more flexibly and accurately, making it more robust and resistant to change.

#### **4. Benefits of unit testing**

Unit testing plays a central role in the software development process, providing many key benefits that improve the quality and reliability of code at various stages of code creation.

Early detection of errors: One of the main advantages of unit testing is the ability to detect errors at the initial stages of development. Testing code immediately after it is written identifies and fixes problems before they get deeper into the system, saving time and money in later stages of the project. It also reduces the risks associated with defects in the product when it reaches end users.

Simplified refactoring and support: Unit testing sets the stage for safe code refactoring because it provides developers with a "safety net" of tests that immediately signal problems caused by changes. This greatly simplifies the process of optimizing

and improving code structure without the risk of compromising functionality. In addition, well-structured unit tests can serve as a form of documentation that helps new and current developers understand and work with the code base more quickly and efficiently.

Promote quality software design: A unit testing approach forces developers to consider the architecture and design of the application from the beginning, encouraging the development of modular, loosely coupled code that is easier to maintain and scale. Applying principles such as single responsibility and separation of interfaces leads to separate, independent units that are easier to test and integrate. The practice of development through testing (TDD) deepens these benefits as tests guide code design, ensuring that code is minimal and focused.

Improved team communication and collaboration: Unit tests serve as a common language for the development team, making it easier to share knowledge about how code should function. They

simplify the process of peer code review by providing clear criteria for evaluating changes. Unit tests also ensure quick adaptation of new team members, reducing the time needed to familiarize them with the project. Unit tests promote quality software design by formalizing requirements and supporting refactoring. They aid in early defect detection and provide living documentation, enhancing code understanding and discussion.

Integration with other types of testing: Although unit testing is the foundation, it is complemented by integration and functional tests, each of which solves its tasks within the testing pyramid. A model proposed by Mike Cohn illustrates how unit tests form the basis of this pyramid (fig.2), emphasizing their numerical predominance and relative ease of execution compared to other types of tests [7].

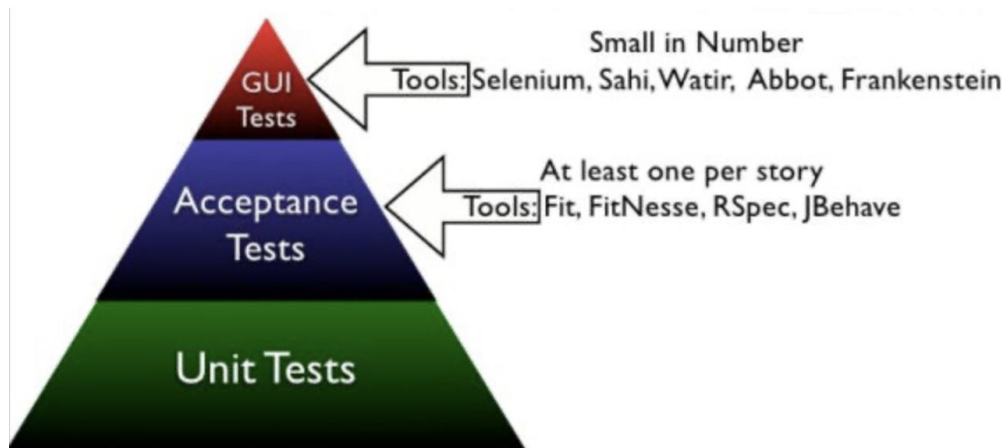


Fig. 2 - Mike Cohn Testing Pyramid

### 5. Types of testing

Integration testing: This type of testing seeks to test the interaction between individual modules by evaluating how they work together. Integration testing is particularly important for identifying problems in data transfer and interfaces between different components of the system. This process makes sure that modules that function correctly in isolation also function correctly when working together.

Functional Testing: Functional testing aims to verify that a software product meets its functional requirements and specifications, usually conducted through black box methods where the internal structure of the system remains unexplored. This type of testing focuses on the behavior of the program from the end user's point of view, confirming that the functional requirements are met.

Unit Testing Tools: To perform unit testing, developers can use various frameworks that make

it easy to create and execute tests:

- JUnit is a widely used framework for Java that offers convenient annotations for defining tests and preparing test scripts.
- NUnit, a .NET application testing platform evolved from JUnit, provides flexible options for creating complex test cases and supports data-driven testing.
- unittest is a standard testing tool in Python, supports automation, and test aggregation, and can run independently of a reporting system.
- Mocha is a JavaScript framework that works in both Node.js and browsers. It supports asynchronous testing and provides detailed reports, making it easy to diagnose bugs and track results [8].

### CONCLUSION

In conclusion, the exploration of unit testing within this study highlights its crucial role in ensuring software reliability and quality. Through the

meticulous examination of its principles, methodologies, and advanced practices, it becomes evident that unit testing is indispensable for early bug detection, which substantially reduces debugging time and associated costs. The implementation of unit testing fosters modular and loosely coupled code, which not only simplifies maintenance and refactoring but also enhances overall software design quality.

The comparative analysis of manual versus automated testing underscores the necessity of balancing both approaches to optimize testing efficiency and effectiveness. The use of mocking and automation tools further solidifies the reliability of test results, providing developers with immediate feedback and facilitating continuous integration practices. Advanced techniques, particularly in handling command-line arguments, environment variables, and console outputs, demonstrate the flexibility and robustness of unit tests in complex scenarios.

Moreover, unit testing serves as the foundation within the testing pyramid, as proposed by Mike Cohn, emphasizing its predominance and ease of execution compared to integration and functional tests. This hierarchical structure ensures comprehensive test coverage, validating that both individual components and their interactions meet the desired functional requirements.

The integration of unit testing with other testing types and tools, such as JUnit, NUnit, unittest, and Mocha, exemplifies its adaptability across various programming languages and environments. This flexibility is pivotal for developers, providing them with the necessary tools to maintain high standards of code quality and reliability.

## REFERENCES

1. The best methods of modular testing in Python for creating additional applications. [Electronic resource] Access mode: <https://pytest-with-eric.com/introduction/python-unit-testing-best-practices/> (accessed 8.05.2024).
2. How to make a Modular change to an application to improve Code efficiency. [Electronic resource] Access mode: <https://marketsplash.com/apl-unit-testing/> (accessed 8.05.2024).
3. 10 Ways to Improve your unit Testing. [Electronic resource] Access mode: <https://www.cleantechloops.com/ways-to-improve-unit-testing/> (accessed 8.05.2024).
4. Effective unit testing: improve code quality and reduce errors in your software. [Electronic resource] Access mode: <https://www.fromdev.com/2023/05/effective-unit-testing-boosting-code-quality-and-reducing-bugs-in-your-software.html> (accessed 8.05.2024).
5. Recommendations for long-term testing in Java. [Electronic resource] Access mode: <https://www.baeldung.com/java-unit-testing-best-practices> (accessed 8.05.2024).
6. The Basic Modular Terms in Python Are: Setup, Disassembly, Apology, And More. [Electronic resource] Access mode: <https://hands-on.cloud/python-unit-tests/> (accessed 8.05.2024).
7. Unit testing: principles, benefits and 6 simple recommendations. [Electronic resource] Access mode: <https://codefresh.io/learn/unit-testing/> (accessed 8.05.2024).
8. Unit testing and coding: Why the code under test is important. [Electronic resource] Access mode: <https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters> (accessed 8.05.2024).