Test4z Primer - DRAFT

UNIT TESTING CAN SAVE TIME. YES, REALLY!

For many developers, testing is viewed as a necessary but time-consuming task. Why is that? It's because testing COBOL applications often involves numerous manual steps or setting up complex input data scenarios. Test4z challenges the notion that high-quality testing is inherently time-consuming by offering a solution that accelerates the development process.

Test4z streamlines testing in two key ways:

1. Test4z automates unit testing through COBOL-written test suites, allowing for a range of testing approaches without needing to recompile the application under test. For example, the unit test can dynamically insert runtime callbacks to inspect and validate application variables directly, streamlining the transition from testing to production – without cluttering the application code with endless DISPLAY statements.
2. Test4z simplifies test environment setup by enabling "mocking" of program calls and middleware calls to data sources like IMS, CICS, DB2 and VSAM. This feature allows developers to test their code in isolation, free from external dependencies and fluctuations in the test environment, thus saving time and reducing setup frustrations.

More specifically, what do we mean by "mocking" programs and middleware calls? Mocking is done in two steps:

1. By editing a few lines of your application's JCL, you run the production-ready code in a live environment and Test4z records the input and output calls to middleware and programs
2. You run your unit test suite, but this time the Test4z runtime intercepts the calls, substituting input and output data captured in the previous recording.

The bottom line? With Test4z, you'll reduce time spent on validating your code, which means more time for delivering new functionality.

TEST4Z WORKSHOP: WHAT YOU'LL LEARN AND DO

In this workshop, we recommend that you start with this primer on unit testing and an introduction to Test4z. Next, you'll dive into a hands-on exercise using a simple application that will be tested by the unit test suite you'll write. You'll learn to create unit tests for production code without a live environment, leveraging Test4z's ability to mock middleware interactions.

In other words, your program will *think* it's reading and writing real files, but it's actually using the Test4z data from a previously recorded live environment that we'll provide. This demonstrates how Test4z ensures your application can interact with system resources in a controlled, isolated test environment, enhancing stability and predictability in your testing process.

PRIMER - WHAT AND WHY OF UNIT TESTING

The term *unit test* is well-known, but if you ask three COBOL developers to define it, you may get three different definitions! Is it code that validates the operation of a single program? Or parts of a COBOL program like a SECTION or PARAGRAPH? It's not the same as integration testing, right?!?

Unit testing in COBOL means validating the smallest testable parts of a program in *isolation*, which is crucial for test automation. And whether it's comparing inputs and outputs, examining internal processes, or manipulating internal program states, Test4z helps developers to thoroughly test their code in a manner that best suits their needs, free from the constraints of a shifting test environment.

When writing unit tests with Test4z, the developer/tester decides what's the smallest testable part and how much the unit test knows about the internal workings of the tested program. As a simple shorthand notation, think of the unit test's level of knowledge of program internals as black, gray, and white box testing:

- **Black box testing**: This tried-and-true approach assesses only externally accessible input-output without any knowledge of the internal workings of the application. It's straightforward and effective for confirming that the program delivers the expected outputs for given inputs, but the unit test doesn't have insights into the underlying process.

- **Gray box testing**: This approach offers a middle ground, allowing the unit test to have some knowledge of the internal structures. It extends beyond merely checking the end results, enabling the unit test to examine intermediate steps and understand how the outputs are produced. This approach can reveal more about the application's behavior and potential points of failure.

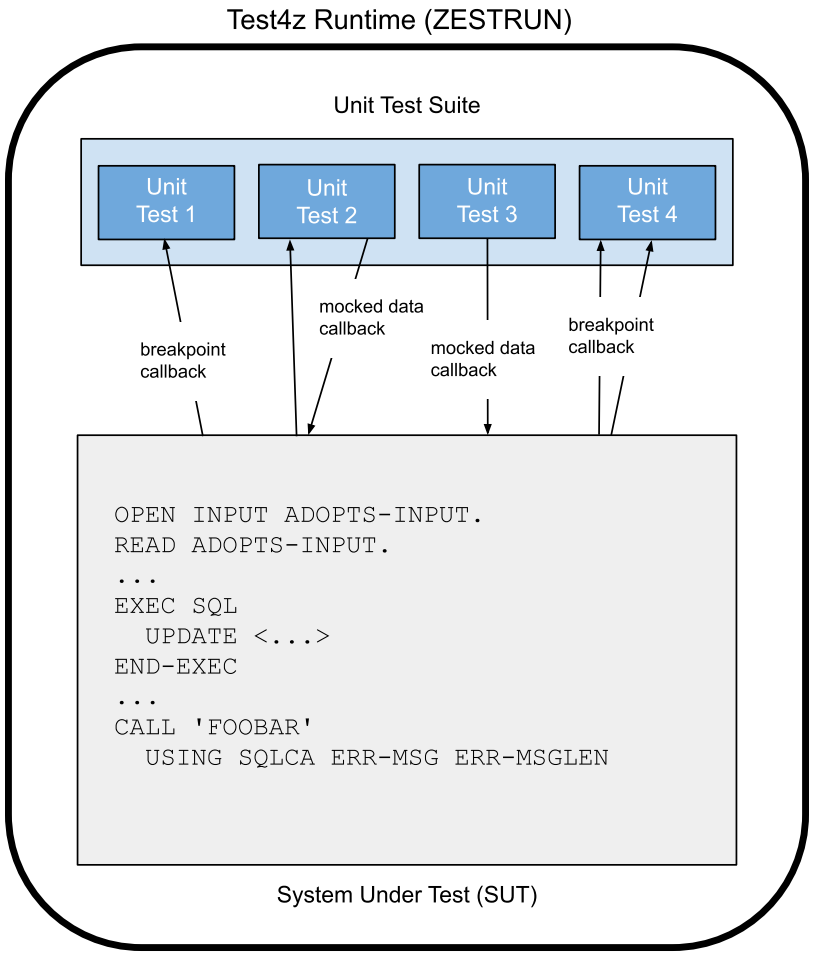
- **White box testing**: Often written by the developers who wrote the application code, this approach exploits complete transparency into the application's internals. The unit test can programmatically manipulate internal program states and monitor the execution flow at a granular level, ensuring that all logic paths are tested. For example, a white box unit test might manipulate the return code of the program's middleware call or change a

program's variable value, forcing the execution along an otherwise incompletely tested "unhappy path".

With Test4z, you can tailor your testing strategy to fit the specific requirements of your application, whether that's validating straightforward input-output relationships, examining internal processes, or thoroughly testing the internal logic and data flow, including exception pathways. This flexibility enhances the depth and breadth of your testing, contributing to more robust and reliable code.

HOW TEST4Z WORKS

How does this "test magic" actually work? The diagram below depicts the main actors in the Test4z runtime:



Your production code is the System Under Test (SUT); it's a load module that can be used as-is, that is, there's no need to recompile it. The SUT is loaded by the Test4z runtime by modifying the `EXEC PGM` statement in your original JCL – don't worry about the details, we'll review example JCL in the exercise.

The new code you'll write is the *test suite*, plus one or more *unit tests* that execute some portion of the SUT. A test suite is a single load module that exists within a load library with multiple unit tests defined by COBOL entry points. Once the load module is ready, the Test4z test runner will execute each of the suite's unit test entry points, recording if they passed or failed.

The Test4z runtime provides an extensive API that you'll call in your unit tests. We'll review those that you'll use most often, such as those that create and manage mocked resources, spies, and section breakpoints.

Let's briefly define them:

- **Mocks**: Create simulated resources within your unit tests, replacing real dependencies with controlled versions to ensure consistent test outcomes. For example, mocking a QSAM file allows the SUT to interact with a predetermined dataset independent of the live environment, facilitating precise, repeatable test conditions conducive to automation.

- **Spies**: Capture and record the interactions with resources, providing insights into how your application behaves with certain inputs and outputs. Spies can track real or mocked resource interactions, offering a detailed view of your application's operation during tests. For example, a file spy would record all the operations against a real resource like a QSAM file. As part of testing validation, your unit test spies can make assertions about the data operations in real-time or just log what they "see".

- **Section breakpoints**: Set specific points in your code where the SUT execution will pause, allowing the unit test to do in-depth inspection or modification of the program state. This feature is particularly useful for validating the execution flow, such as ensuring specific sections or paragraphs of code are triggered under the right conditions.

Now that you have a better understanding of the Test4z components and how they work together, let's put these concepts into action! It's time to start coding with the workshop exercise.