Test4z Tutorial V1.1

Tip #1: The "Try it ✎" icon indicates where input from you, the developer, is required
Tip #2: Short on time? See the "hints" folder in your Visual Studio Code workspace
Tip #3: Select [main menu] > View > Open View… > Outline to quickly traverse COBOL source

Like a lot of developers, you might think of testing as just more work. But with Test4z, you'll see in this exercise how it can actually speed up development! Test4z does this in two ways:

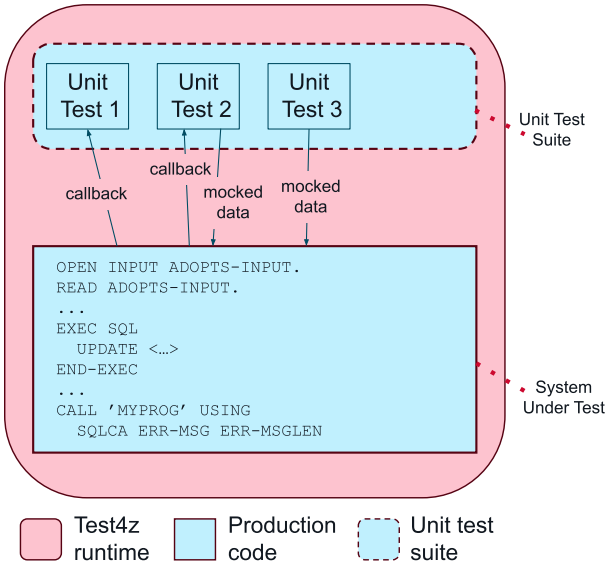
1. By reducing the need for manual testing.
2. By simplifying the setup of a valid and stable test environment using recordings, instead of the actual live environment.

Beyond the setup time savings for developers, with the benefit of automated testing, your organization saves time responding to regressions caused by less-than-complete testing. These two principles–**isolation** and **automation**–are fundamental to the Test4z ethos, *"The key to higher quality code is making it easier to test."*

WHAT IS UNIT TESTING? HOW DOES TEST4Z WORK?

Unit testing in COBOL means validating the smallest testable parts of a program in isolation, which is crucial for test automation. 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.

The diagram below depicts the main actors in the Test4z runtime:



Test4z Tutorial V1.1

Your production code is the System-Under-Test in the lower box; it's a load module that can be used as-is; there's no need to recompile it. The SUT is loaded by the Test4z runtime and then instrumented with callbacks to your unit test, giving it the opportunity to observe and validate the SUT's processing.

The new code you'll write is the *unit test suite* load module, shown in the upper dashed box, plus unit tests that execute portions of the SUT. A test suite contains multiple unit tests defined by COBOL entry points. Once the SUT load module is ready, the Test4z unit 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. In this exercise, we'll focus on two API categories that you'll use most often:

- **Mocks**: Simulated resources within your unit tests, replacing real dependencies with controlled versions to ensure consistent test outcomes
- **Spies**: Code that captures interactions with middleware resources, providing insights into how your application interacts with certain inputs and outputs.

Together, mocks and spies lead to the unit test's most important step:

- **Validation** - Verifying your code works as expected and keeps working as expected!

Now that you have a better understanding of the Test4z components and how they work together, let's look at some code composed of the SUT (often called the "program-under-test") and the unit test (UT) responsible for validation.

CONTENTS:

Test4z Tutorial V1.1

EXERCISE REVIEW

This exercise requires two programs: The tested program ZTPDOGOS and the unit test program ZTTDOGWS.

- ZTTDOGWS - the unit test program that validates the operation of the "dog adoption" report program.

- ZTPDOGOS - this is a simple application that reads from an input file and writes a report to an output file. The input file ADOPTS has this format:

```
01  ADOPTED-DOGS-REC.
     05 INP-DOG-BREED              PIC X(30).
     05 FILLER                     PIC X(25).
     05 INP-ADOPTED-AMOUNT         PIC 9(3).
     05 FILLER                     PIC X(22).
```

  Below are example records written to the OUTREP report file:

```
BREED SHIBA                         WAS ADOPTED 008 TIMES
BREED SCHNAUZER                     WAS ADOPTED 000 TIMES
...
BREED OTHER                         WAS ADOPTED 000 TIMES
```

  Your unit test program ZTTDOGWS will capture the OUTREP file output and compare it against expected results.

  The ZTPDOGOS program keeps a running count of nine breeds in an internal `ACCUMULATOR` variable that is used to produce the report totals:

```
01 ACCUMULATOR.
     05 BREED-ADOPTIONS PIC 9(3) OCCURS 9 TIMES VALUE 0.
```

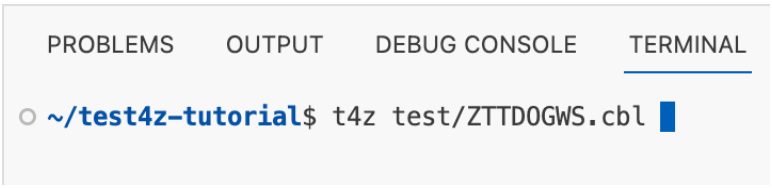
  Your unit test program will access this working storage variable to double-check the SUT's calculations.

The bulk of this exercise will focus on the unit test code in ZTTDOGWS using Visual Studio Code. If you haven't started VS Code, please do so now.

Test4z Tutorial V1.1

HOW TO RUN A TEST4Z TEST SUITE

You can start a unit test multiple ways, such as using the Test4z command line interface (CLI) named `t4z`, a pop-up menu from the VS Code Explorer, or from the dedicated Testing view. The command to run the unit test program ZTTDOGWS that validates ZTPDOGOS is shown below:



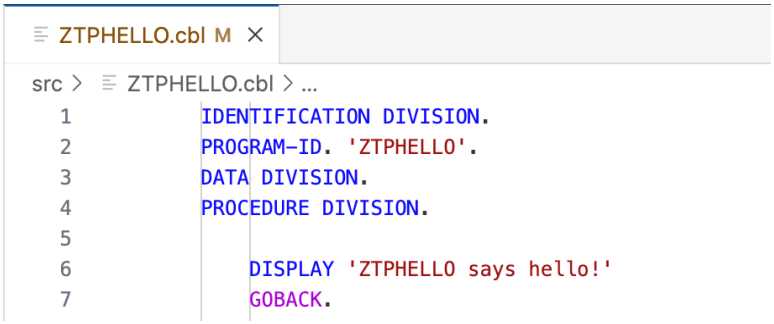
Test4z uses Team Build to compile COBOL source found in the programs-under-test `src` folder and unit test suites in the `test` folder, execute them on the mainframe, then download the results to your VS Code workspace.

TEST4Z APIs AND CODE SNIPPETS

There are many Test4z APIs to build your unit test case. But don't worry about the specifics, since Test4z's Visual Studio Code extension provides "code snippets". All you do is type `t4z` in the VS Code editor followed by the first few letters of the API name.

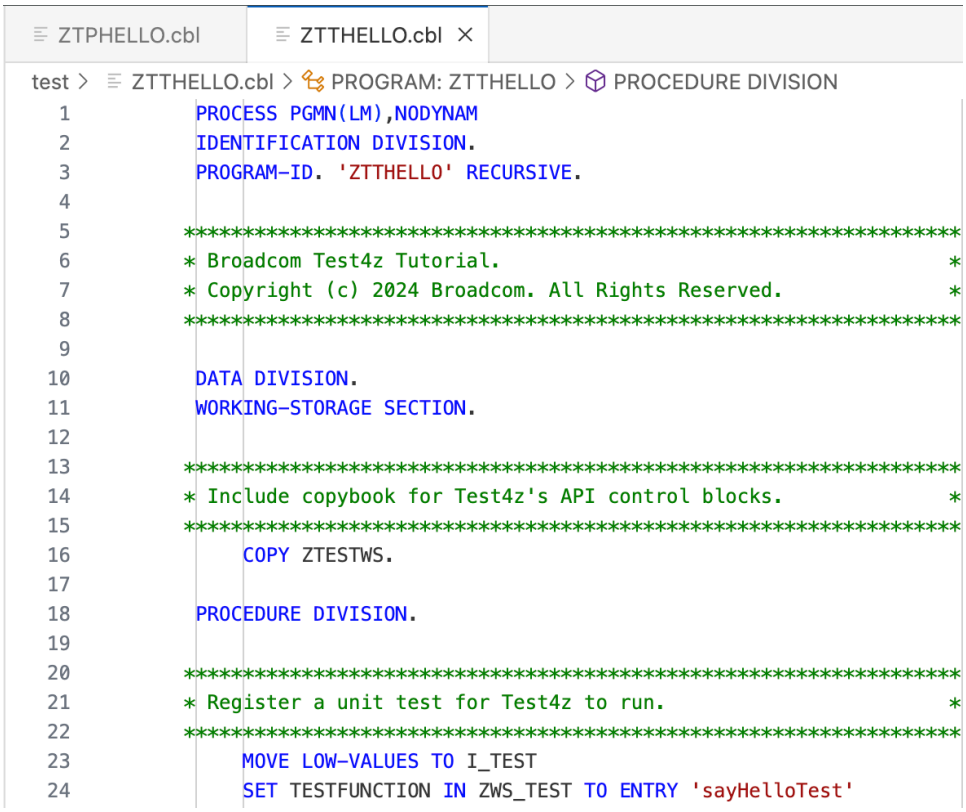
To familiarize yourself with snippets, let's modify a very simple "Hello, World!" program and then run it with Test4z.

Open the SUT program ZTPHELLO.cbl; it's located in the `src` folder and shown below:



As promised, it's a *really* short program! The unit test suite program ZTTHELLO.cbl is a bit longer; look for it in the `test` folder.

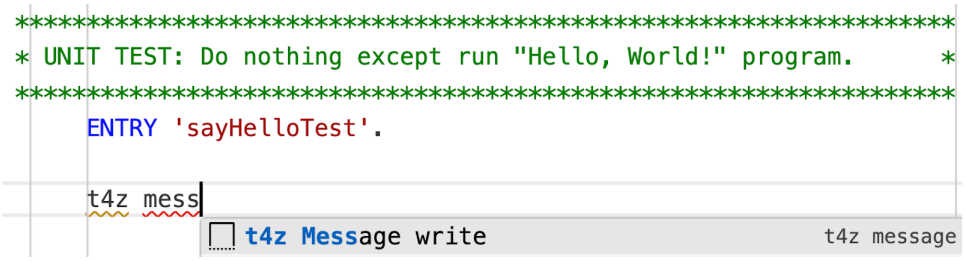
Test4z Tutorial V1.1



Code (0): Try it ✎

Add a new Test4z API call in **ZTT**HELLO.cbl to its unit test—the `sayHelloTest` unit test is defined in an `ENTRY` section. Search for "TUTORIAL" to skip down to the correct section of code.

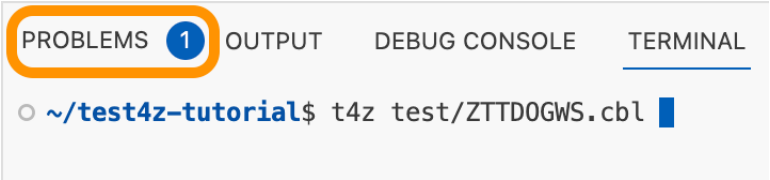
Next, click below `ENTRY` in the editor where you want the snippet to be added and type "t4z mess…" as shown below. It will display matching APIs; select "Message write".



Finally, change the `MESSAGETEXT` value to `'ZTTHELLO you there?'`. For debug purposes, you may wish to add a `DISPLAY` statement. The final code is shown below, including the optional `DISPLAY` statement in blue:

```
MOVE LOW-VALUES TO I_MESSAGE IN ZWS_MESSAGE
MOVE 'ZTTHELLO you there?' TO MESSAGETEXT IN ZWS_MESSAGE
DISPLAY '[SYSOUT] ' MESSAGETEXT IN ZWS_MESSAGE
CALL ZTESTUT USING ZWS_MESSAGE
```

The PROBLEMS tab summarizes any syntax errors that were found.



Double-check there are no errors and then continue.

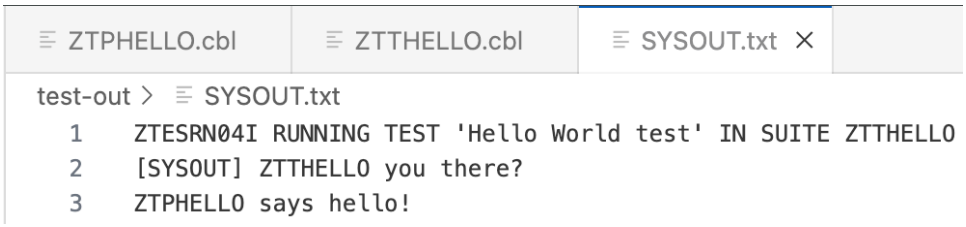
Run (a): Try it ✎

Save your changes to ZTTHELLO.cbl. To execute ZTTHELLO/ZTPHELLO, enter `t4z test/ZTTHELLO.cbl` in the Terminal pane (select *[main menu]* > *Terminal* > *New Terminal* if you need to open a new one). Once Test4z finishes, you should see something like this:

```
 PASS  test/ZTTHELLO.cbl
  ✓ Hello World test (822 ms)
      ZTTHELLO you there?

Tests Suites: 1 passed, 1 total
Tests:        1 passed, 1 total
Time:         1 s
```

In addition to the pass/fails shown in the Terminal pane, `DISPLAY` output is copied from the mainframe SYSOUT data set into the `test-out/SYSOUT.txt` file:



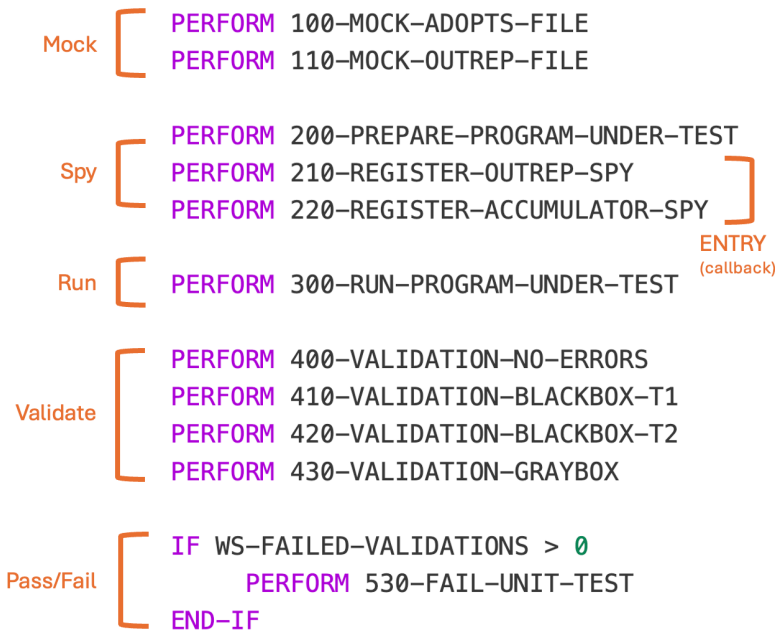
Another file in the same `test-out` folder, `ZLMSG.txt`, includes a summary of what unit tests were run and whether they passed or failed.

THE "BIG PICTURE" OF TEST4Z UNIT TESTING

Now that you've taken a quick look at running a unit test, let's take a step back and consider the bigger picture. The Test4z APIs can be classified into several categories corresponding to their role in unit testing:

1. RECORD - Execute and record middleware operations of the SUT as-is
2. MOCK - Emulate real resources in a test environment
3. SPY - Observe SUT state changes with spies, watches, and breakpoints
4. RUN - Execute UT as a sub-program of Test4z, which then loads/runs the SUT
5. VALIDATE - Perform post-execution validation and pass/fail the unit test

These are represented in the ZTTDOGWS code in the `PROCEDURE DIVISION`:



Test4z Tutorial V1.1

In this exercise, most of the unit test code is provided. In the next sections, you'll complete what's missing using Test4z's MOCK, SPY, and VALIDATE related APIs.

MOCK IT

For this exercise, the unit test suite ZTTDOGWS will emulate the input and output QSAM files that the program-under-test ZTPDOGOS processes. By providing mocked files, the unit test program can monitor how the tested program performs in an environment decoupled from a potentially changing live environment.

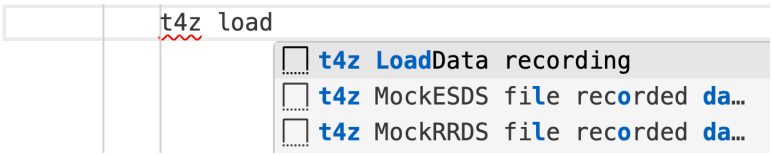
That is, when run under Test4z, the tested program will *behave* as if it's reading a real QSAM file, but it's actually using the Test4z data from a previously recorded live environment stored in `ZTPDOGOS.json`.

Let's finish the code for mocking these two files, ADOPTS and OUTREP.

Establishing the mocked ADOPTS QSAM file starts by loading the previously recorded data provided with this tutorial. If you haven't done so already, please open `test/ZTTDOGWS.cbl` (*not* the tested program `src/ZTPDOGOS.cbl`).

Code (T1): Try it ✎

Search for the paragraph `100-MOCK-ADOPTS-FILE`, then use the snippet "t4z load..." to add a call to the `_LOADDATA` API shown below:



First, change the `MEMBERNAME` parameter value to `'ZTPDOGOS'`. To fix the squiggly line under `LOADOBJECT IN LOAD_DATA`, change `LOAD_DATA` to the already defined working storage variable `WS-ZDATA-RECORDING` (you should also reformat the line and delete the snippet's comment reminder). The remaining default parameters are unchanged.

The updated snippet code is shown below:

```
100-MOCK-ADOPTS-FILE.
```

```
DISPLAY 'ZTTDOGWS 100-MOCK-ADOPTS-FILE'
...
MOVE LOW-VALUES TO I_LOADDATA
MOVE 'ZTPDOGOS' TO MEMBERNAME IN ZWS_LOADDATA
CALL ZTESTUT USING ZWS_LOADDATA,
     LOADOBJECT IN WS-ZDATA-RECORDING
```

Reminder 1 ✎ Specify the recording member name `'ZTPDOGOS'` above, not `'ADOPTS'`, which is a QSAM file whose operations were recorded from a live environment into the `MEMBERNAME` JSON file.

Reminder 2 ✎ If you see "squiggles" under the `CALL` statement above, check that the code isn't past column 72. Add a line break as necessary.

The `MEMBERNAME` refers to the recording stored in the `test/data` directory; it's in JSON format and represents all the middleware operations Test4z has recorded during the live execution of ZTPDOGOS (if you're curious, browse it – it's human-readable). It will be uploaded to the mainframe into the ZLDATA data set.

Code (T2): Try it ✎

To use the recording, the loaded data above is an optional parameter of the `_MockQSAM` API. Uncomment the required `SET LOADOBJECT` statement below `MOVE 'ADOPTS'…`, as shown below:

```
MOVE LOW-VALUES TO I_MOCKQSAM
MOVE 'ADOPTS' TO FILENAME IN ZWS_MOCKQSAM
SET LOADOBJECT IN ZWS_MOCKQSAM
     TO LOADOBJECT IN WS-ZDATA-RECORDING
MOVE 80 TO RECORDSIZE IN ZWS_MOCKQSAM
CALL ZTESTUT USING ZWS_MOCKQSAM,
     QSAMOBJECT IN WS-ZQSAM-ADOPTS-MOCK
```

When you run ZTTDOGWS, the Test4z CLI will upload the recording to the mainframe. The mock for the ADOPTS file accepts this recorded data as input in the `_MockQSAM` API with the `LOADOBJECT` parameter.

Code (T3): Try it ✎

Since the output file OUTREP doesn't require recorded data, creating its mock is even easier. Start by searching for the paragraph `110-MOCK-OUTREP-FILE`.

Test4z Tutorial V1.1

Use the t4z snippet command "t4z mockqsam" to add the template code for the OUTREP mock; be sure to select "MockQSAM *file*" as there are several `_MockQSAM` APIs. Change the mocked file name to `'OUTREP'`. Also change the output data structure to `WS-ZQSAM-OUTREP-MOCK` (it's already defined in the working storage section). This data structure will contain the QSAM mock in the field `QSAMOBJECT`.

The snippet code with two updated parameters is shown below:

```
MOVE LOW-VALUES TO I_MOCKQSAM
MOVE 'OUTREP' TO FILENAME IN ZWS_MOCKQSAM
MOVE 80 TO RECORDSIZE IN ZWS_MOCKQSAM
CALL ZTESTUT USING ZWS_MOCKQSAM,
     QSAMOBJECT IN WS-ZQSAM-OUTREP-MOCK
```

Reminder ✎ Double-check you specified `'WS-ZQSAM-OUTREP-MOCK'`, not the `ADOPTS` mock.

This completes the first part of this exercise: Creating mocks for the input and output data, thereby isolating our unit test from the live environment. Let's see if it works!

Run (b): Try it ✎

As before with the "Hello World" mini exercise, save your work, and then run the unit test from the Terminal pane using the Test4z CLI command `t4z test/ZTTDOGWS.cbl`. The result is shown below:

```
$ t4z test/ZTTDOGWS.cbl
Test4z CLI version: 1.1.101
Uploading source code and test data, building, and executing tests...

 PASS  test/ZTTDOGWS.cbl
   ✓ ZTTDOGWS simple totals test (295 ms)

Tests Suites: 1 passed, 1 total
Tests:        1 passed, 1 total
Time:         0 s
```

The tested program is now accessing your mocked files, but it's only showing `PASS` because there's no validations. Let's add them in the next section by creating a "spy" on the OUTREP output file.

Test4z Tutorial V1.1

SPY IT

As the name suggests, a spy can observe the behavior of different middleware resources; in the case of this exercise, we'll create a spy on the QSAM output file.

Code (T4): Try it ✎

Search for the paragraph `210-REGISTER-OUTREP-SPY`. Use a snippet to add the code, this time with "t4z spyq…". Be sure to select "Spy QSAM file *with callback*" since there's more than one SpyQSAM API. Eelete the snippet's comment reminders and callback example, then update the three parameters and as shown below:

```
MOVE LOW-VALUES TO I_SPYQSAM
SET CALLBACK IN ZWS_SPYQSAM TO ENTRY 'spyCallbackOUTREP'
MOVE 'OUTREP' TO FILENAME IN ZWS_SPYQSAM
CALL ZTESTUT USING ZWS_SPYQSAM,
     QSAMSPYOBJECT IN WS-ZSPQSAM-OUTREP-SPY
EXIT.
```

The `CALLBACK` field specifies the unit test's entry point `spyCallbackOUTREP` that will be invoked whenever an operation against the OUTREP file is requested.

This callback gives the unit test an opportunity to validate the output records. For example, below is an excerpt of the DISPLAY output from this callback:

```
ZTTDOGWS spied - BREED SHIBA     WAS ADOPTED 008 TIMES
ZTPDOGOS wrote - BREED SHIBA     WAS ADOPTED 008 TIMES
ZTTDOGWS spied - BREED SCHNAUZER WAS ADOPTED 000 TIMES
ZTPDOGOS wrote - BREED SCHNAUZER WAS ADOPTED 000 TIMES
```

Notice how the DISPLAYs from the spy (red) *precede* the DISPLAYs from the program-under-test (blue). This demonstrates how a Test4z spy is notified of middleware operations *before* the SUT receives a response; this enables the spy to observe, log, and optionally modify the response.

VALIDATE IT (PART 1)

In the case of the OUTREP file spy, validation is handled by checking if the `COMMAND` [file operation] is `WRITE`. If that's the case, the unit test increments the valid write count so it can be compared against the expected total once the SUT ends.

Code (T5): Try it ✎

In the `spyCallbackOUTREP` entry section, find the `IF` statement excerpt below; it's checking the file operation (command) and status code. Insert the code to increment `WS-ACTUAL-OUTREP-WRITES`:

```
            IF COMMAND IN ZLS_QSAM_RECORD = 'WRITE' AND
                  STATUSCODE IN ZLS_QSAM_RECORD = '00'

      *-------------------------------------------------------
      * TUTORIAL(5) - Record count of actual WRITEs to OUTREP.
      *-------------------------------------------------------
              ADD 1 TO WS-ACTUAL-OUTREP-WRITES

              SET ADDRESS OF LS-OUTREP-RECORD
                   TO PTR IN RECORD_ IN ZLS_QSAM_RECORD
              DISPLAY 'ZTTDOGWS spied - ' LS-OUTREP-RECORD
         END-IF
```

Since the ADOPTS file is mocked, we know how many writes to expect. We'll verify it in the next validation step of this exercise.

VALIDATE IT (PART 2)

That was the setup of a QSAM spy on OUTREP. Now, how to use it? This short exercise has two QSAM spy validations:

- Black box validation #1 - number of WRITEs to OUTREP
- Black box validation #2 - comparison of expected versus actually written records.

The exercise also includes one variable spy:

- Gray box validation - compare expected versus actual totals from the SUT.

The difference between the first two black box validations and the last gray box validation is a matter of "insider knowledge". A black box validation only considers what's accessible outside the SUT; a gray box validation takes advantage of implementation details and possibly internal access.

Let's complete the code for the first black box validation above and then review the other two validations.

Code (T6): Try it ✎

Remember the QSAM file spy you coded earlier that kept track of writes? Once the SUT ends, it's time to validate them. Recall from the `PROCEDURE DIVISION`:

```
PERFORM 300-RUN-PROGRAM-UNDER-TEST
PERFORM 400-VALIDATION-NO-ERRORS
PERFORM 410-VALIDATION-BLACKBOX-T1
PERFORM 420-VALIDATION-BLACKBOX-T2
PERFORM 430-VALIDATION-GRAYBOX
```

The 400 level validations (blue) above are done *after* the SUT has run (red), i.e., ZTPDOGOS has terminated and its memory freed. The spies you created, however, exist in the unit test suite's memory, so their working storage and the SUT states it captured are available for post-execution validation.

[If this point is unclear, please review the diagram in the section WHAT IS UNIT TESTING? HOW DOES TEST4Z WORK?]

Once the SUT ends, we can compare the runtime tally with the expected total. It makes sense to compare the expected number of writes first, because if they don't match, the actual written records won't match either.

Search for the paragraph `410-VALIDATION-BLACKBOX-T1` and add the code below:

```
IF WS-ACTUAL-OUTREP-WRITES NOT = WS-EXPECTED-OUTREP-WRITES
      PERFORM 500-REPORT-COUNT-MISMATCH
END-IF
```

Recall the first variable (`WS-ACTUAL…`) is the write count captured by the QSAM spy; the second variable (`WS-EXPECTED…`) is the write count that was calculated based on the recorded input for ADOPTS.

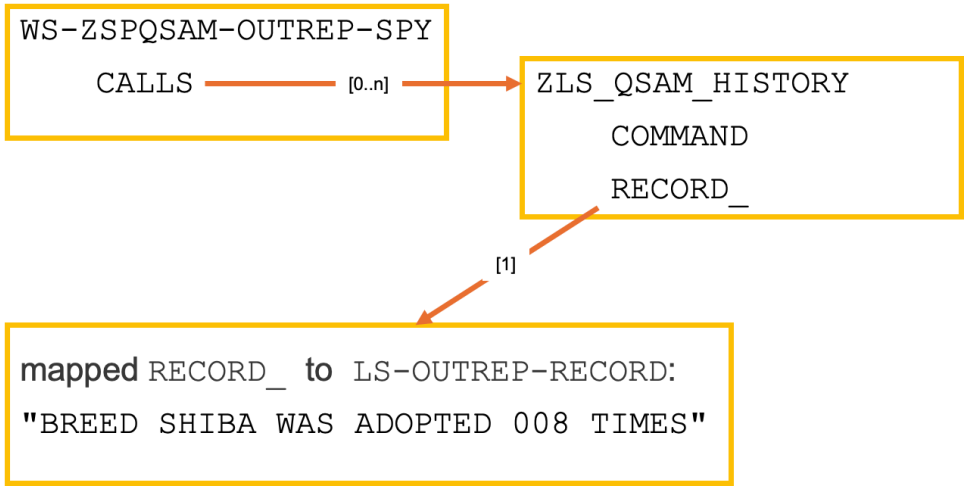
Save your update and then try running it. Were any mismatches reported?

Code (T7): Try it ✎

Test4z Tutorial V1.1

The second black box validation looks at the actual captured history of OUTREP records that can be compared against the expected records. The latter is defined in the hardcoded working storage table `WS-EXPECTED-OUTREP-RECORD(n).`

The code in `420-VALIDATION-BLACKBOX-T2` loops through the expected records and compares them against the captured records. Review this code and refer to the diagram below; it represents the key data structures that are accessed:



Admittedly, this code *may* look a little arcane. We'll blame that on COBOL's pointer implementation. 😉

To address the actual records from the captured data, the validation code maps addresses from the QSAM spy into the linkage storage section variable `ZLS_QSAM_HISTORY` (defined by Test4z) and then the individual record definition `LS-OUTREP-RECORD` (defined by ZTTDOGWS).

Despite the helpful diagram above, sometimes it's better to see what's going on via `DISPLAY` statements. Add the `DISPLAY` shown below inside the loop traversing the QSAM spy's call history:

```
           PERFORM VARYING I FROM 1 BY 1
                     UNTIL I > SIZE_ IN CALLS IN WS-ZSPQSAM-OUTREP-SPY
                DISPLAY 'ZTTDOGWS filename='
                     FILENAME IN ZLS_QSAM_HISTORY(I)
                     ' status=' STATUSCODE IN ZLS_QSAM_HISTORY(I)
                     ' command=' COMMAND IN ZLS_QSAM_HISTORY(I)
```

This will show the file operations that were intercepted and stored by the Test4z spy. Tip: To save typing the above, see the `hints/copy-paste.txt` file.

Run (c): Try it ✎

Save your changes, then run your unit test. Once it finishes, open the `test-out/SYSOUT.txt` file. Review the output; it's excerpted below for easy reference:

```
ZTTDOGWS 420-VALIDATION-BLACKBOX-T2
ZTTDOGWS filename=OUTREP   status=00 command=OPEN OUTPUT
ZTTDOGWS filename=OUTREP   status=00 command=WRITE
ZTTDOGWS filename=OUTREP   status=00 command=WRITE
ZTTDOGWS filename=OUTREP   status=00 command=WRITE
ZTTDOGWS filename=OUTREP   status=00 command=WRITE
ZTTDOGWS filename=OUTREP   status=00 command=WRITE
ZTTDOGWS filename=OUTREP   status=00 command=WRITE
ZTTDOGWS filename=OUTREP   status=00 command=WRITE
ZTTDOGWS filename=OUTREP   status=00 command=WRITE
ZTTDOGWS filename=OUTREP   status=00 command=WRITE
ZTTDOGWS filename=OUTREP   status=00 command=CLOSE
ZTTDOGWS 430-VALIDATION-GRAYBOX
```

Notice that the command/operations include not just WRITEs (blue), but also the other file-related operations like OPEN and CLOSE (red). This history enables your spy to have a complete picture of what the SUT was doing from start-to-finish.

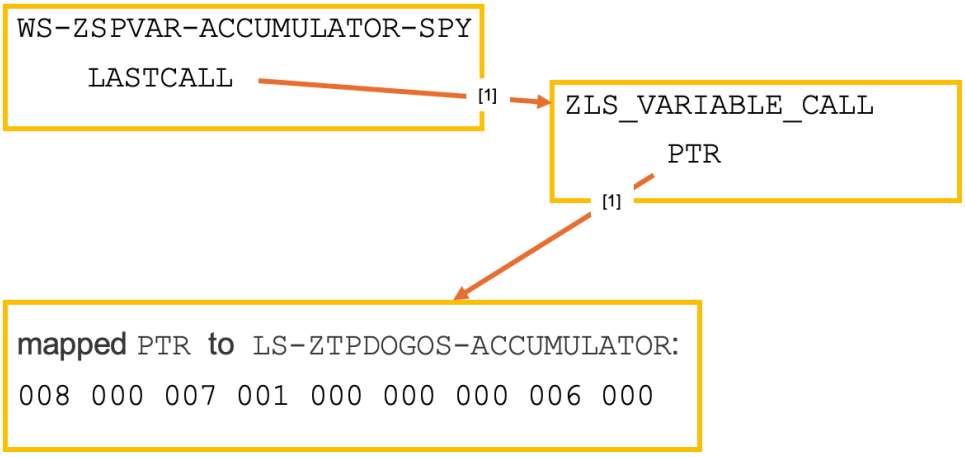
The final code is for the gray box validation. Let's review it.

Code (T8): Review it ✎

Search for the paragraph `430-VALIDATION-GRAYBOX`.

The gray box validation takes advantage of *internal* knowledge of variables in the working storage section of ZTPDOGOS. Test4z's variable spy you registered earlier captures a history of `ACCUMULATOR` changes. Like the QSAM spy history, these changes can be validated post-execution.

This diagram below represents the key data structures of the code in `420-VALIDATION-GRAYBOX`:



The final captured variable value can conveniently be compared to the expected result via the variable spy's `LASTCALL` reference. It's excerpted and highlighted below from the example code from the `420-VALIDATION-GRAYBOX` paragraph:

```
SET ADDRESS OF ZLS_VARIABLE_CALL
     TO LASTCALL IN WS-ZSPVAR-ACCUMULATOR-SPY
...
SET ADDRESS OF LS-ZTPDOGOS-ACCUMULATOR
     TO PTR IN ZLS_VARIABLE_CALL
MOVE LS-ZTPDOGOS-ACCUMULATOR TO WS-FINAL-ACCUMULATOR

IF WS-FINAL-ACCUMULATOR NOT = WS-EXPECTED-ACCUMULATOR
     PERFORM 520-REPORT-TOTALS-MISMATCH
END-IF
```

By comparing the write count, record content, and even the internal totals, we're certain the unit test has validated the correct operation of ZTPDOGOS.

PASS OR FAIL IT

If you have time, modify the unit test expected values to make the unit test fail.

Code (T9): Try it ✎

For example, modify one of the unit test's expected values defined in the `WORKING STORAGE` section of ZTTDOGWS:

- Change `WS-EXPECTED-OUTREP-WRITES` to 6 (was 9), or

- Change `OUTREP-7` from `'BREED BULLDOG WAS ADOPTED 3 TIMES'` to `'BREED LABRADOR WAS ADOPTED 3 TIMES'`, or
- Change `BREED-ADOPTIONS-9-OTHER` to 3 (was 0).

Then run the test suite again. It should fail with an assert error and the details in `SYSOUT.txt`:

```
FAIL  test/ZTTDOGWS.cbl
  × ZTTDOGWS simple totals test (325 ms)
      Assertion error: Failed 03 validations
      SYSOUT:
      ZTTDOGWS 100-MOCK-ADOPTS-FILE
      ZTTDOGWS 110-MOCK-OUTREP-FILE
      …
```

Did you finish early? Excellent! Let's see if any code wasn't unit tested in the next section.

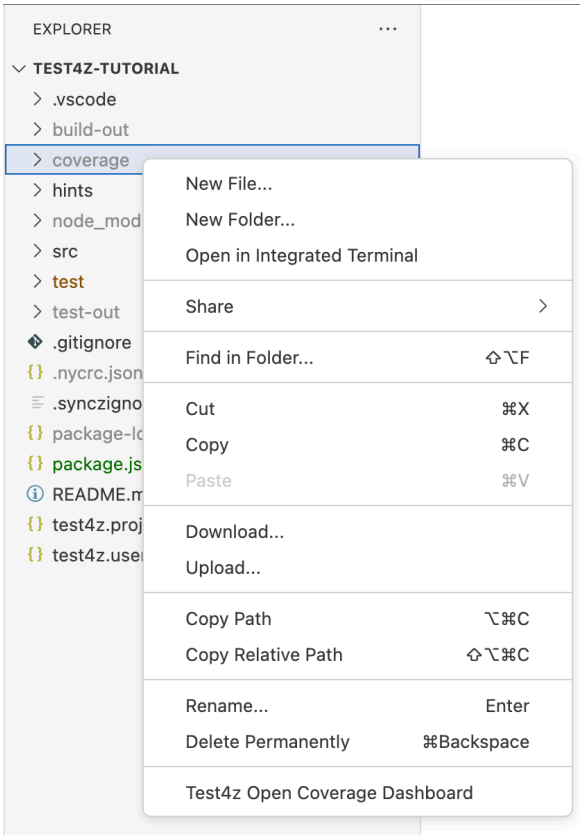
BONUS! TEST4Z CODE COVERAGE

Code coverage answers the question, "How well does my unit test suite exercise the program-under-test?" If you have extra time, check out how well ZTTDOGWS does.

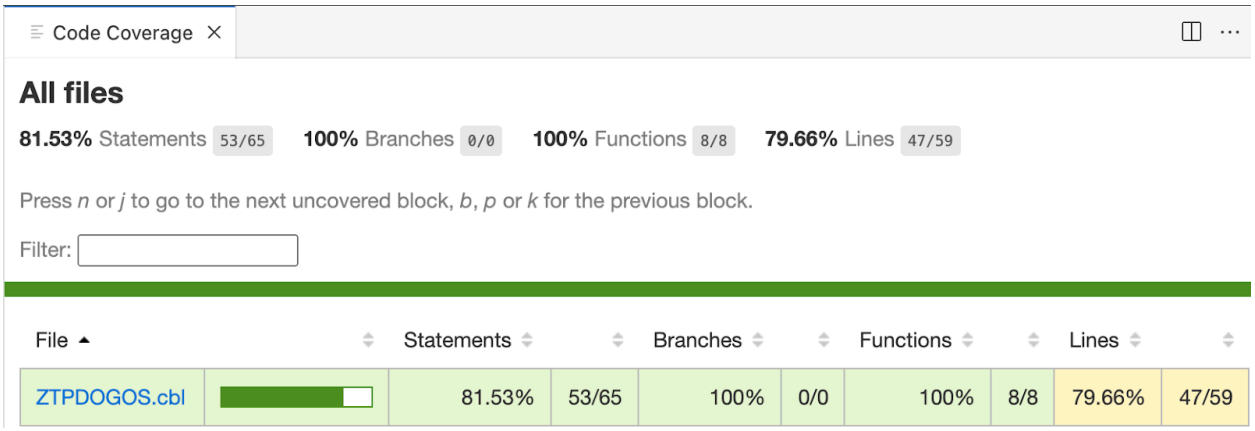
Run (d): Try it ✎

Generating code coverage is simple: Just add `--cov` to the end of the t4z command, for example, `t4z test/ZTTDOGWS --cov`. This will display a summary of code coverage in the Terminal pane.

To get more details, open the Coverage Dashboard by right clicking the `coverage` folder to display its pop-up menu and then select "Test4z Open Coverage Dashboard":



This view shows the percentage of statements, conditional branches like `IF` and `EVALUATE`, and functions (sections/paragraphs) that were executed:

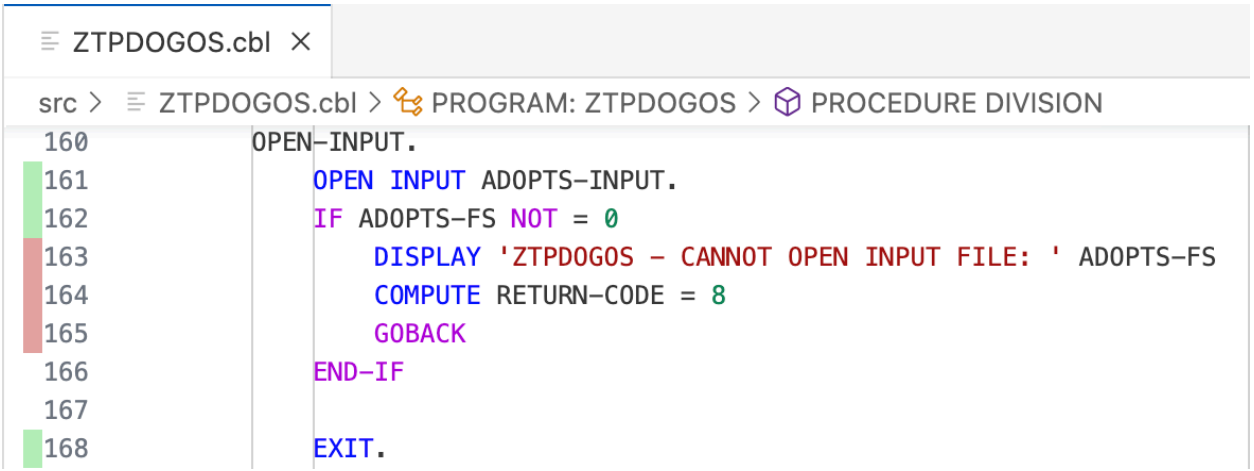


You can see more detail by clicking the source file name in the dashboard.

The Code Coverage dashboard is a static snapshot saved as an HTML file. However, once code coverage is run, the associated source files are also annotated with green/red gutters in the VS Code editor.

Try it ✎

Open src/ZTPDOGOS.cbl as shown below:



The annotations indicate which statements were executed (green) and which were not (red).

WHAT'S NEXT?

This is just a brief overview of using Test4z. While the exercise might be simplified, the overriding Test4z principles–**isolation and automation**–can be applied to any code under test, reducing the need for manual testing and increasing the effectiveness of your test efforts.