Test4z Workshop - DRAFT

WHAT YOU'LL LEARN AND DO

In this exercise, you'll take on the role of a developer testing the code of a simple application, ZTPDOGOS.

As mentioned in the primer, you won't have to set up a live test environment, thanks to Test4z's middleware mocks. In other words, in this exercise, your ZTPDOGOS program will *think* it's reading and writing QSAM files, but it's actually using the Test4z data from a previously recorded live environment that we'll provide. You're completely isolated from changes in the infrastructure. Neat, eh?

HOW TEST4Z FITS INTO THE DEVELOPMENT/TEST LIFECYCLE

In the primer, we asserted that Test4z can actually *speed up* development. Now it's time to prove it!

Test4z does this two ways: (1) by reducing the need for manual testing and (2) by simplifying the setup of a valid, stable development/test environment using recordings instead of an actual live environment.

Beyond the setup time savings for developers, with the benefit of automated testing via unit tests, 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."

For this exercise, you'll follow these top-level steps:

- Part 1: Write/verify unit test using the recording (mock API)
- Part 2: Add validation logic to the unit test (spy API)
- Part 3: Add pass/fail checks to the unit test (assert API)
- Part 4: Run final test (optionally make it fail)
- Addendum: Create a recording in a live environment (JCL)

The following sections introduce the two programs for this exercise, ZTPDOGOS and ZTTDOGWS, which were based on the samples included with Test4z; the ZTP* programs are the programs under test and the ZTT* programs are the associated unit test suites.

Test4z Workshop - DRAFT

OVERVIEW OF TESTED PROGRAM AND UNIT TEST PROGRAM

Let's look more closely at the two programs for this exercise:

- ZTPDOGOS - the simple application that reads from an input file, ADOPTS, calculates totals, and then writes the results to an output file, OUTREP. The ADOPTS input file 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).
```

  The output report recognizes nine breeds including SHIBA, SCHNAUZER, CORGI, and so on. All unrecognized breeds are totaled under OTHER. Below are example records written to OUTREP:

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

  As an example of black box testing, your ZTTDOGWS unit test will create a QSAM spy on the OUTREP file. This will enable you to validate that ZTTDOGOS has written the expected number of output records.

  The tested program ZTPDOGOS keeps a running count of each breed 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 000.
```

  This is an example of gray box testing where your ZTTDOGWS unit test peeks into ZTTDOGOS' internal running count `ACCUMULATOR` to double-check its calculations.

- ZTTDOGWS - the unit test program that validates the operation of ZTPDOGOS. As noted above, your unit test will have assertions for two metrics, namely (1) the output record count and (2) the breed totals.

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

EXERCISE TIP! *We've included "hint" files you can copy/paste. If you want to move onto the next part, look for the file ZTTDOGWS_P1.txt, ZTTDOGWS_P2.txt, ZTTDOGWS_P3.txt, or ZTTDOGWS_P4.txt in the hints folder. These correspond to the starting point of the given section.*

OVERVIEW OF UNIT TEST PROGRAM: ZTTDOGWS

Unit test development begins with Test4z recording of middleware and program calls; the recording can then be used in an immutable unit test environment. To save time, we've already done this step for you using the ZTPDOGOS program-under-test, run directly from JCL (these edits are documented in the addendum of the exercise),

The recorded data was captured from the live environment and saved in your Visual Studio Code workspace in the test/data folder as ZTPDOGOS.json. Although it's human-readable, we expect other tools will want to take advantage of this captured data to create useful insights into how your code *really* works.

The basic structure of the exercise's ZTTDOGWS unit test program has been provided to you, so Part 1-4 will focus on coding the mocks, spies, and asserts with Test4z's COBOL APIs. Let's briefly review the startup code and then get to the actual code modifications.

Once the Test4z test suite runner (ZESTRUN) loads your test program, it sequentially executes the unit tests you've registered. Below is an excerpt of the unit test for ZTTDOGWS:

```
*****************************************************************
* Unit test called by Test4z test suite runner.
*****************************************************************
           entry 'outrepTotalsUnitTest'

           perform mockADOPTSFile
           perform mockOUTREPFile
           perform registerOUTREPFileSpy
           perform runZTPDOGOS

           goback.
```

You may be thinking *"Hey, wait a minute! That looks like standard COBOL code. Where are the Test4z APIs?"* Indeed, the unit test entry point `outrepTotalsTest` won't be called unless it's registered. That's done with the Test API. Below is an excerpt of the unit test registration for ZTTDOGWS:

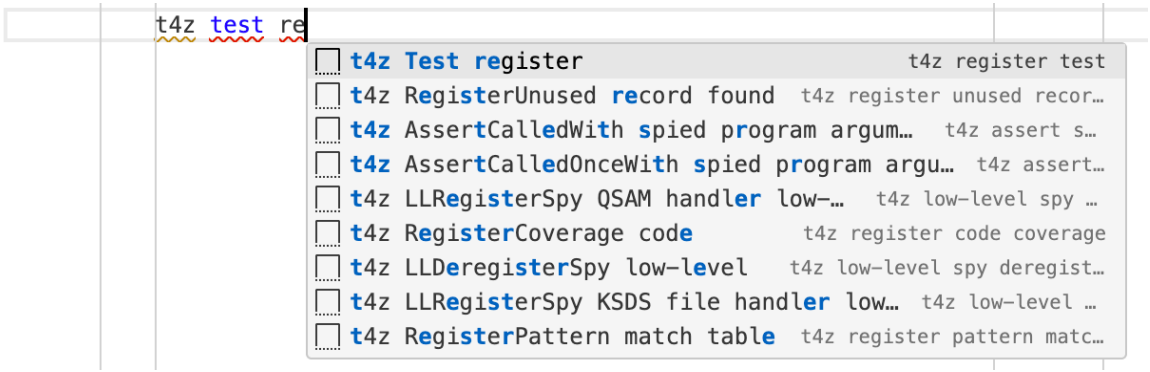
```
      PROCEDURE DIVISION.

*****************************************************************
* Register test to be run (only one in this simple example).
*****************************************************************
           move low-values to I_Test
           set testFunction in ZWS_Test to entry 'outrepTotalsUnitTest'
           move 'ZTTDOGWS simple totals test' to testName in ZWS_Test
           call ZTESTUT using ZWS_Test
```

It's really that simple to register a unit test as part of a test suite: You declare an entry point in your test suite and call Test4z to register it. The Test4z test suite runner takes it from there.

TEST4Z APIs AND CODE SNIPPETS

The code pattern for other Test4z COBOL APIs is essentially the same as shown above. Fortunately, Test4z's Visual Studio Code extension adds "code snippets" that generate them for you – all you have to do is type `t4z` followed by the start of the API name. For example:



Although the code snippets are great timesavers, it's worth understanding the pattern for the Test4z COBOL API invocations. For each API, there's an associated control block. Consider the prior Test API invocation above as a model of this pattern:

```
move low-values to I_Test
set <subfield in ZWS_Test> to <value>
move <text> to <text field in ZWS_Test>
call ZTESTUT using ZWS_Test
```

The control block `ZWS_Test` has fields, like `testFunction`, qualified by `I_Test` (where `I` is for input). The statement `move low-values` clears the control block of previous values; next, you

assign other fields in the control block. If there's output parameters, those are appended to the call to ZTESTUT.

Let's look at one more example, this time for a fictitious "Abc" API with output parameters in **bold** below. The pattern would be:

```
move low-values to I_<ZWS_Abc input>
set <subfield in ZWS_Abc/I_Abc input> to <value>
move <text> to <text_field in <text field in ZWS_Abc input>
call ZTESTUT using ZWS_Abc,
         <output field> in <return structure for ZWS_Abc>
```

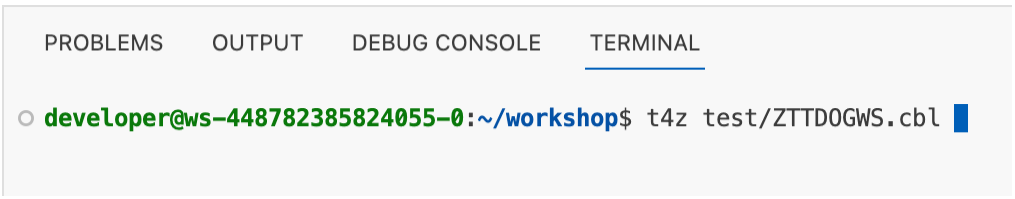
As with the prior example, the control block input area is cleared, subfields are set, and then ZTESTUT is called. Just as with the Test4z API copybook that defines the ZWS_* input control blocks, Test4z provides copybooks for the APIs that return values (e.g., ZWS_LoadData is qualified by I_LoadData and includes a copybook for its output parameters, ZDATA).

Don't worry if this seems like a lot of details to remember. The Test4z snippet includes a comment reminder to add the required output parameter to your WORKING-STORAGE section. As you'll see in the exercise, thanks to the snippet templates, calling the Test4z COBOL APIs is almost fill-in-the-blanks coding.

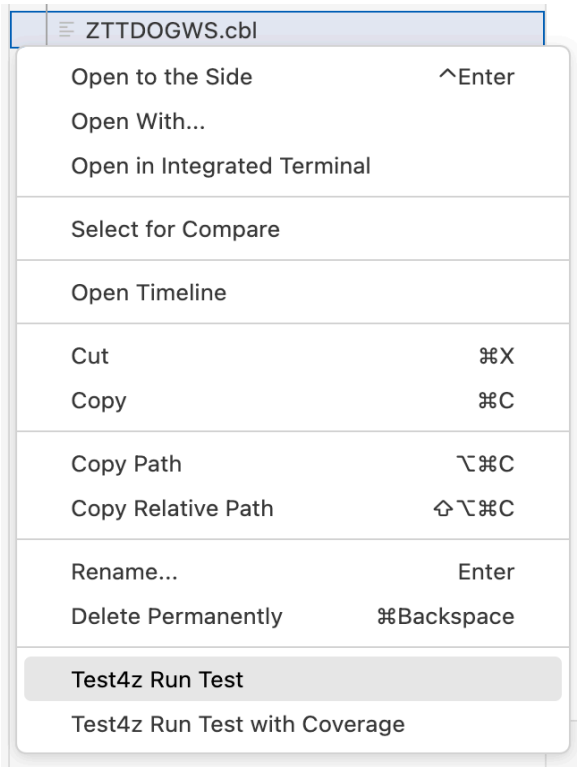
HOW TO RUN A TEST4Z TEST SUITE

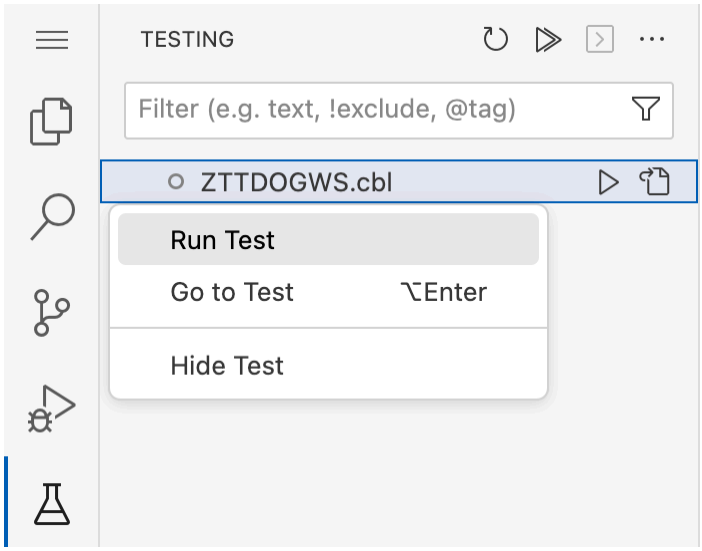
There's one last step before you start coding: Run the skeletal ZTTDOGWS unit test suite.

The program doesn't actually do anything except register a unit test and run. You can start a 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. These three options are depicted below:



Test4z Workshop - DRAFT





Pick one of the above and run it. Test4z uses Team Build to compile COBOL source found in the src/* and test/* folders, execute them on the mainframe, then download the results to your VS Code workspace.

Test4z shows its progress in TERMINAL for the first two choices and in TEST RESULTS of the Testing view. The terminal output will look something like this:

```
$ t4z test/ZTTDOGWS.cbl
Test4z CLI version: 1.0.xxx-develop
Executing Endevor Team Build on xxx.broadcom.net to build and run tests...

 PASS  test/ZTTDOGWS.cbl
  ✓ ZTTDOGWS simple totals test (770 ms)
      ZTSTQF01W UNMOCKED FILE ADOPTS DOES NOT HAVE A DD. AN 0C4 ABEND IN THE
USER APPLICATION IS LIKELY.

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

Running tests completed successfully. For more details see files
'test-out/ZLMSG.txt' and 'test-out/SYSOUT.txt'.
```

The "ZTSTQF01W UNMOCKED FILE ADOPTS DOES NOT HAVE A DD" message is displayed because the skeletal version of ZTTDOGOS includes file operations for ADOPTS, but there's no corresponding DD for it in the JCL and the file is not mocked. We'll take care of that in Part 1.

TEST4Z WORKSPACE STRUCTURE

The output of the test is stored in the workspace's test-out folder; it includes DISPLAY messages in SYSOUT.txt and the unit test pass/fail summary in ZLMSG.txt. Below is a summary of the default Test4z folder structure:

- workshop/src - COBOL programs under test
- workshop/test - COBOL unit test programs
- workshop/data - recorded data files in JSON format
- workshop/test-out - output from unit test execution
- workshop/coverage - output from code coverage (generated by `t4z --cov`)
- workshop/build-out - program compiler listings

For easy reference, we've also included an exercise "hints" folder:

- workshop/hints - copy/paste friendly COBOL source code

Okay, enough with the introductions! Open the unit test suite ZTTDOGWS.cbl and let's start coding.

EXERCISE TIP! *We've included "hint" files you can copy/paste. If you want to move onto the next part, look for the file ZTTDOGWS_P1.txt, ZTTDOGWS_P2.txt, ZTTDOGWS_P3.txt, or ZTTDOGWS_P4.txt in the hints folder. These correspond to the starting point of the given section.*

Test4z Workshop - DRAFT

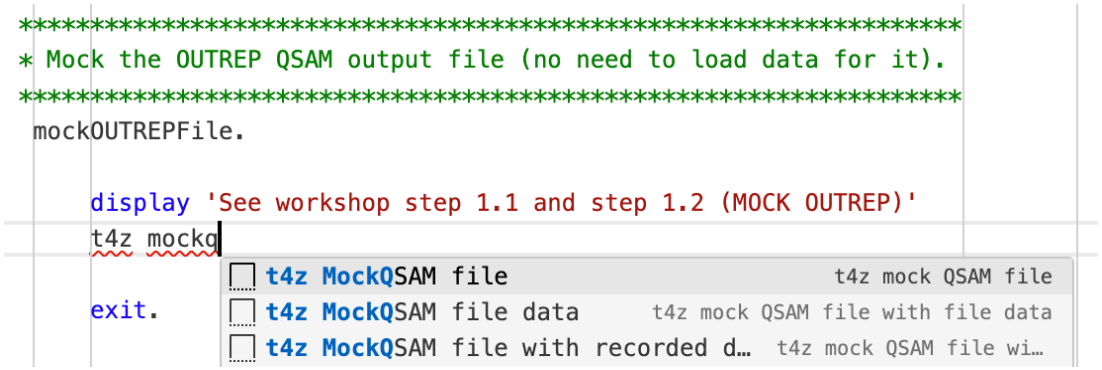
PART 1 - WRITE/VERIFY UNIT TEST (MOCK)

In this section, you'll add code to mock the input file ADOPTS and output file OUTREP. If you haven't already opened ZTTDOGWS.cbl, do so now.

Important! Open the test suite program (ZTT*), *not* the program under test (ZTP*).

MOCK OUTREP

1.1). Search for the paragraph `mockOUTREPFile` (or "step 1.1") then insert a call to MockQSAM using the snippet "t4z mockq…".



There's quite a few mock APIs, so be sure to select "MockQSAM file". This will insert the following:

```
move low-values to I_MockQSAM
move 'QSAMFILE' to fileName in ZWS_MockQSAM
move 80 to recordSize in ZWS_MockQSAM
call ZTESTUT using ZWS_MockQSAM, qsamObject in QSAM_Data
```

1.2). Change the `fileName` parameter 'QSAMFILE' to 'OUTREP' and the output parameter `QSAM_Data` to `MOCK_OUTREP`; the other default parameters are unchanged:

```
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 MOCK_OUTREP.
```

Since OUTREP is an output file, there's no need to load it with recorded data. Of course, ADOPTS is a mocked input file, so unlike OUTREP, it will need recorded data. Let's take care of this in the next step.

LOAD RECORDED DATA

1.3). Search for the paragraph `mockADOPTSFile` (or "step 1.3"). Since this is an input file, the mock will need recorded data. You can load it with the LoadData API. Use the snippet template "t4z LoadData recording"; it will insert the following:

```
move low-values to I_LoadData
move 'MYMEMBER' to memberName in ZWS_LoadData
call ZTESTUT using ZWS_LoadData, loadObject in LOAD_Data
```

Change the `memberName` parameter 'MYMEMBER' to 'ZTPDOGOS' (note: `memberName` refers to the file/member that contains all the recorded operations, *not* the file 'ADOPTS').

```
move low-values to I_LoadData
move 'ZTPDOGOS' to memberName in ZWS_LoadData
call ZTESTUT using ZWS_LoadData, loadObject in LOAD_Data
```

This tells the Test4z runtime to search for the recorded data in the data set xxx.WORKSHOP.**ZLDATA**.ZTPDOGOS that the Test4z CLI uploaded from **test**/**data**/ZTPDOGOS.json in your VS Code workspace.

MOCK ADOPTS

1.4). The prior step loaded the recorded data, but now you need a place to store it. So add a call to MockQSAM after the prior call to LoadData (search on "step 1.4"). You can use the snippet "t4z MockQSAM with recorded data".

Important! Be sure to select the snippet ending with "...*recorded data*" since there's more than one MockQSAM API. Then update the `fileName` parameter from 'QSAMFILE' to 'ADOPTS' and the output parameter `QSAM_Data` to `MOCK_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 LOAD_Data
move 80 to recordSize in ZWS_MockQSAM
call ZTESTUT using ZWS_MockQSAM, qsamObject in MOCK_ADOPTS.
```

Note the `set loadObject` statement in the *italicized* code above. When the tested program ZTPDOGOS reads the ADOPTS file, the mocked file will return the recorded response from the `loadObject` that you created in step 1.3.

SMOKE TEST

1.5). Run the test suite and confirm it compiles and executes successfully. For example, open a new Terminal window with the VS Code pulldown choice Terminal > New Terminal and enter the CLI command:

```
t4z test/ZTTDOGWS.cbl
```

The unit test test is now loading recorded data and using it to mock the reads for the ADOPTS file – that resolved the "UNMOCKED FILE DOES NOT HAVE A DD" warning you saw earlier. But it is not validating the output yet; we'll do that in Part 2.

If you have any errors, please correct them before continuing.

EXERCISE TIP #1 *If you receive the runtime error "UNMOCKED FILE xxx DOES NOT HAVE A DD", check the fileName parameter in the MockQSAM call. Test4z's runtime is indicating there's no mock or DD statement for input/output file "xxx".*

EXERCISE TIP #2 *We've included "hint" files you can copy/paste. If you want to move onto the next part, look for the file ZTTDOGWS_P1.txt, ZTTDOGWS_P2.txt, ZTTDOGWS_P3.txt, or ZTTDOGWS_P4.txt in the hints folder. These correspond to the starting point of the given section.*

PART 2 - ADD VALIDATION LOGIC TO UNIT TEST (SPY)

For the validation logic of this unit test, you will "spy" on the WRITE and CLOSE operations against the OUTREP file. First, you register your spy callback – that's a COBOL entry point that will be called for each OUTREP operation. Next, you add validation code to the callback.

REGISTER SPY

2.1). Search for the paragraph `registerOUTREPFileSpy` (or "step 2.1"). Add the code to register a spy on the QSAM file. As before, start with a snippet, in this case, "SpyQSAM file with callback":

```
move low-values to I_SpyQSAM
set callback in ZWS_SpyQSAM to entry 'my_QSAMSpy'
move 'QSAMFILE' to fileName in ZWS_SpyQSAM
call ZTESTUT using ZWS_SpyQSAM, qsamSpyObject in QSAM_Data
```

Important! Be sure to select the snippet that ends with "*...with callback*" since there's more than one SpyQSAM API.

Notice that the squiggles underneath `qsamSpyObject in QSAM_Data`. Update `QSAM_Data` to the correct output parameter, `OUTREP_SPY`; we already defined that for you in the WORKING-STORAGE section:

```
1 OUTREP_SPY.
  COPY ZSPQSAM.
```

2.2). The callback parameter indicates which COBOL entry point should be invoked; change the `entry` parameter 'my_QSAMSpy' to your callback entry point 'spyCallbackOUTREP'; also change the `fileName` parameter 'QSAMFILE' to 'OUTREP'. The final result is 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 OUTREP_SPY.
```

If you want to double-check your work, run the unit test suite again. Because the validations aren't complete, the unit test will fail with this message:

```
FAIL  test/ZTTDOGWS.cbl
 × ZTTDOGWS simple totals test (860 ms)
     Invalid OUTREP count from ZTTDOGWS
```

Tip: If you only want to compile the source to save time, use the CLI command below:

```
t4z test/ZTTDOGWS.cbl --skip-test-execution
```

The final step is adding validation code to the callback that you registered.

EXERCISE TIP! *We've included "hint" files you can copy/paste. If you want to move onto Part 4, look for the file ZTTDOGW4.txt. It includes all the source modifications for Parts 1-3.*

PART 3 - ADD PASS/FAIL CHECKS TO UNIT TEST (ASSERT)

Final coding step! For this simple exercise, your QSAM file spy will drive two validations. It does this by recording how many write operations were performed. Once the OUTREP file is closed, the spy validates:

1. The number of written records is correct
2. The total for each breed is correct.

If either comparison doesn't match the expected values, a failed assertion is signaled.

VALIDATE OUTREP

3.1). Search for the entry point `spyCallbackOUTREP` (or "step 3.1"). This entry point is called when an operation is performed against OUTREP; the callback's incoming parameter indicates what operation was performed and includes a table of all spied calls so far in the `calls` parameter.

For convenience, the incoming parameter to the callback includes the last invocation in `lastCall`. The code at the beginning of `spyCallbackOUTREP` maps it into an addressable structure:

```
set address of ZLS_QSAM_Record to
     lastCall in SPY_CALLBACK_OUTREP
```

Once this is mapped to the callback details `ZLS_QSAM_Record`, your spy callback can determine what operation was performed, by which module, what file was affected, the status code, and so on:

```
01 ZLS_QSAM_RECORD.
  03 COMMAND PIC X(32).
  03 MODULENAME PIC X(8).
  03 FILENAME PIC X(8).
  03 ITERATION PIC S9(9) USAGE COMP-5.
  03 STATUSCODE PIC X(2).
  03 RECORD_.
```

3.2). Complete the IF statement that checks the `command` for a valid WRITE (search on "step 3.2"). If it's status code is valid, you should increment the counter `OUTREP_SPY_WRITE_COUNT`, as shown in the bolded code below:

```
 if command in ZLS_QSAM_Record = 'WRITE' and
     statusCode in ZLS_QSAM_Record = '00'

     add 1 to OUTREP_SPY_WRITE_COUNT

  end-if
```

The program ZTPDOGOS has 9 breeds and each bread has one output record, so the expected value of `OUTREP_SPY_WRITE_COUNT` is 9. We'll validate that total in the next step.

3.3). Scroll down and review the IF statement that checks the command for CLOSE. This will be called once and only after all the records have been written; that's where the unit test can validate the number of output records:

```
if command in ZLS_QSAM_RECORD = 'CLOSE'
```

```
           perform validateOUTREPFile
         end-if
```

Search for the paragraph `validateOUTREPFile` (or "step 3.3"). Its validation code has two tests. First, the "black box" test confirms the total number of WRITEs observed by the OUTREP spy matches the number of breed types (9):

```
     validateOUTREPFile.

         if OUTREP_SPY_WRITE_COUNT not = 9 then
           perform failOutrepWriteCount
         end-if
```

VALIDATE ACCUMULATOR

3.4). The second validation is a "gray box" test because it access the internals of the tested program using Test4z's GetVariable API:

```
         move low-values to I_GetVariable
         move 'ACCUMULATOR' to variableName in ZWS_GetVariable
         call ZTESTUT using ZWS_GetVariable,
                  address of ZTPDOGOS-ACCUMULATOR
```

This retrieves the runtime address of ZTTDOGOS' accumulator and maps it into ZTTDOGWS' linkage section with the `ZTPDOGOS-ACCUMULATOR` variable. Given access to the `ACCUMULATOR` variable in the program under test, the unit test ZTTDOGWS can directly validate ZTPDOGOS' internal totals:

```
         if BREED-ADOPTIONS(1) not = 8 or
                   BREED-ADOPTIONS(2) not = 0 or
                   BREED-ADOPTIONS(3) not = 7 or
                   BREED-ADOPTIONS(4) not = 1 or
                   BREED-ADOPTIONS(5) not = 0 or
                   BREED-ADOPTIONS(6) not = 0 or
                   BREED-ADOPTIONS(7) not = 0 or
                   BREED-ADOPTIONS(8) not = 6 or
                   BREED-ADOPTIONS(9) not = 0 then
              perform failInternalAccumulator
         end-if
```

These correspond to the expected output record totals for each breed:

```
BREED SHIBA                          WAS ADOPTED 008 TIMES
BREED SCHNAUZER                      WAS ADOPTED 000 TIMES
BREED CORGI                          WAS ADOPTED 007 TIMES
BREED CHI                            WAS ADOPTED 001 TIMES
BREED POODLE                         WAS ADOPTED 000 TIMES
```

```
BREED POMERANIAN                    WAS ADOPTED 000 TIMES
BREED BULLDOG                       WAS ADOPTED 000 TIMES
BREED JINGO                         WAS ADOPTED 006 TIMES
BREED OTHER                         WAS ADOPTED 000 TIMES
```

Note: We acknowledge the above validations should be more robust; it was simplified for illustrative purposes.

If either of these comparisons are incorrect, the unit test invokes `call ZTESTUT using ZWS_Assert` to signal the unit test failed (see `failInternalAccumulator` or `failOutrepWriteCount` for details).

EXERCISE TIP! *We've included "hint" files you can copy/paste. If you want to move onto the next part, look for the file ZTTDOGWS_P1.txt, ZTTDOGWS_P2.txt, ZTTDOGWS_P3.txt, or ZTTDOGWS_P4.txt in the hints folder. These correspond to the starting point of the given section.*

PART 4 - FINAL TEST

Try running your unit test suite as you did before. Does it pass?

```
$ t4z test test/ZTTDOGWS.cbl
Test4z CLI version: xxx
Executing Endevor Team Build on xxx.broadcom.net to build and run tests...

 PASS  test/ZTTDOGWS.cbl
  ✓ ZTTDOGWS simple totals test (570 ms)

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

Running tests completed successfully. For more details see files
'test-out/ZLMSG.txt' and 'test-out/SYSOUT.txt'.
```

If not, check the ZLMSG.txt and SYSOUT.txt files in the test-out folder to help diagnose the problem.

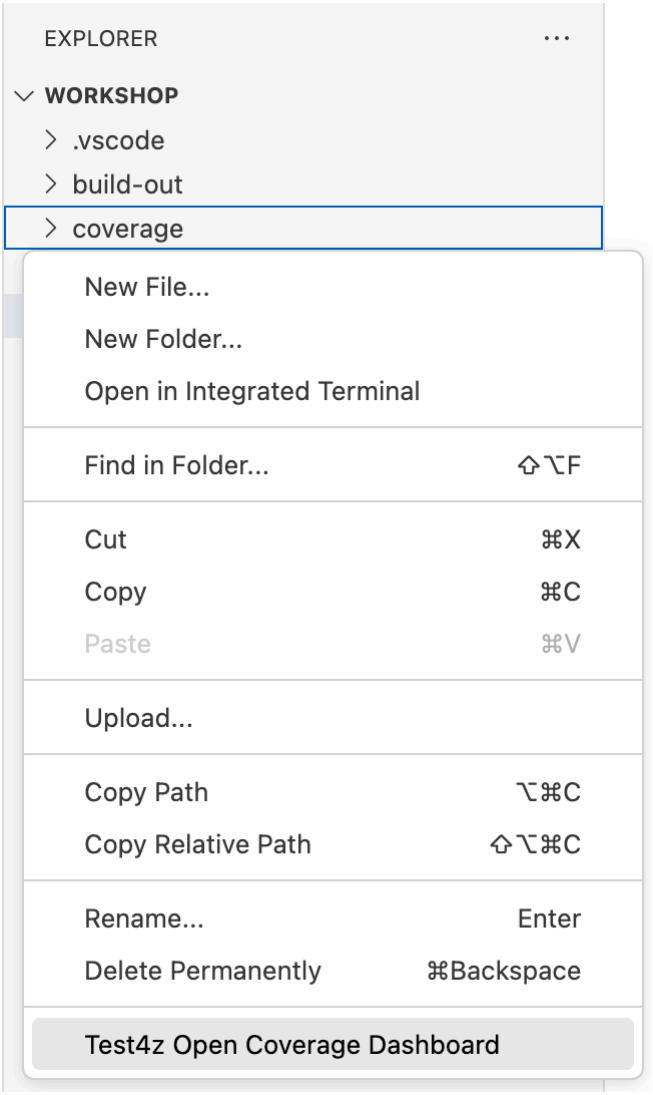
Before wrapping up, let's make the unit test intentionally fail. Modify one of the tests in `validateOUTREPFile`, e.g., double the number of expected records recorded in `OUTREP_SPY_WRITE_COUNT` or change one of the `if BREED-ADOPTIONS...` tests. You should see something like this:

```
 FAIL  test/ZTTDOGWS.cbl
  × ZTTDOGWS simple totals test (119 ms)
      Invalid accumulator value from ZTPDOGOS
```

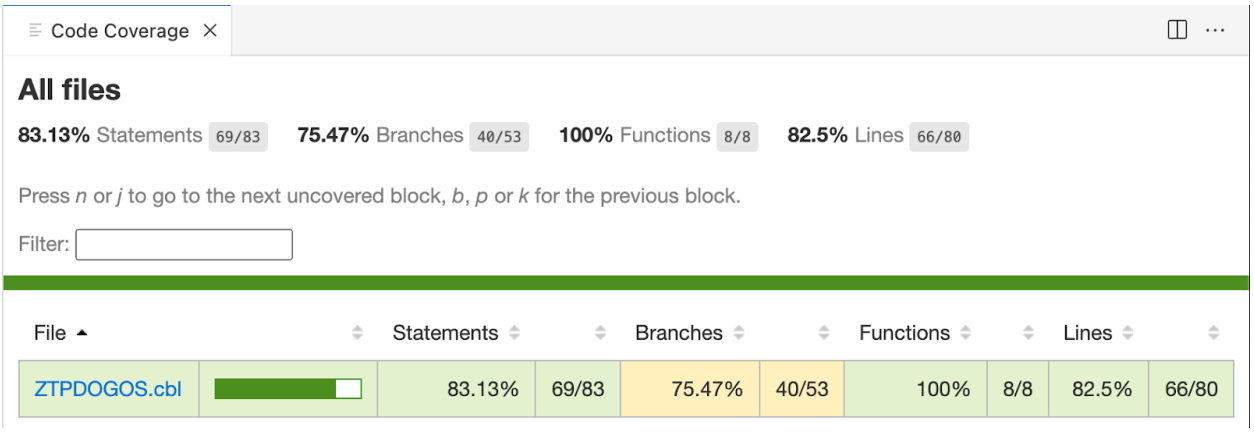
If you have extra time, try running the same test with the CLI's code coverage parameter:

```
t4z test/ZTTDOGWS.cbl --cov
```

Then open the Coverage Dashboard:



How well did your test cover the code in ZTPDOGOS? Double-click ZTPDOGOS.cbl in the coverage dashboard to see more details.



Hint: the ADOPTS file doesn't represent all the possible breeds and the unit test doesn't force ZTPDOGOS along the exception "unhappy paths". How would you test it more thoroughly? That's a more advanced Test4z topic and another workshop!

ADDENDUM - CREATE A RECORDING (JCL)

Unit testing with Test4z starts by recording the middleware and program calls of the program under test; the recording can then be used in an immutable unit test environment. All that's needed is a lightly modified version of the JCL you normally use to run your program.

For example, the JCL used to run the program-under-test ZTPDOGOS is shown below:

```
//ZTPDOGOS EXEC PGM=ZTPDOGOS
//STEPLIB  DD DISP=SHR,DSN=xxx.WORKSHOP.LOAD
//ADOPTS   DD DISP=SHR,DSN=xxx.DOGOS.INPUT(ADOPTS)
//OUTREP   DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
```

Below are the modifications to create the recording:

```
//ZTPDOGOS EXEC PGM=ZTESTEXE
//STEPLIB  DD DISP=SHR,DSN=xxx.WORKSHOP.LOAD
//         DD DISP=SHR,DSN=xxx.WORKSHOP.CT4ZLOAD
//ADOPTS   DD DISP=SHR,DSN=xxx.WORKSHOP.INPUT(ADOPTS)
//OUTREP   DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//ZLMSG    DD SYSOUT=*
```

```
//ZLDATA   DD DISP=SHR,DSN=xxx.DOGOS.ZLDATA
//ZLOPTS   DD *
  RUN(ZTPDOGOS)
/*
//CEEOPTS  DD *
  TRAP(ON,NOSPIE)
/*
```

The changes are highlighted in bold.

For your existing execution JCL, it boils down to changing the PGM parameter to ZTESTEXE, adding the data sets ZLDATA and ZLMSG, and specifying the Test4z options in ZLOPTS. The ZLOPTS RUN(...) parameter should specify the program that appeared in the original EXEC PGM statement.

With these changes, the Test4z runtime will load your program, start recording its middleware and program calls, and store the recording in the ZLDATA data set. Refer to the Test4z documentation for the data set attributes required for ZLDATA and other runtime options that can control how the unit test is executed.

HELPFUL WORKSHOP TIPS

- If "out of space" errors are reported during the build, try running the command `t4z --clean`. This will delete extraneous build files in your workspace and the copies on the mainframe/USS.
- If the workspace's test-out folder is cluttered with ABEND files, you can delete the entire folder. The build output files will be downloaded by the Test4z CLI during the next test.
- If you receive the runtime error "UNMOCKED FILE xxx DOES NOT HAVE A DD", check the `fileName` parameter in the MockQSAM call. Test4z's runtime is indicating it detected a file operation but (a) there's no DD statement for the file "xxx" in the JCL, or (b) the recording wasn't found in the test/data directory ↔ ZLDATA data set.