DYALOG

Elsinore 2023

TP1: Testing APL Systems

Michael Baas, Morten Kromberg, Stefan Kruger

Goals

- Review what we know about existing tools and frameworks for testing
- Present some techniques that Dyalog is actually using
- Share our collective experience
- Discuss requirements for potential future frameworks or tools that Dyalog (or the community) might develop

13:30-14:30 (ish, hopefully a bit less)

Session 1: Introduction (Morten)

- Define Terminology
- Review Some Existing Frameworks & Actual Tests

Exercise 1:
Use "Tester" package to write a test

14:45-15:45

Session 2: DTest (Michael)

- Basics
- Demo
- DIY
- Bonus: Automation
- Bonus: Code Coverage

Exercise: Write a test with DTest for coolStat's "Count" function

16:00-17:00

Session 3: Automation (Stefan)

- The case for automation
- Testing on the command line
- Running tests in Docker
- Automation with GitHub Actions

Exercise: Deploy test automation to GitHub

Terminology & Techniques

Types of Testing

- Unit
- Regression
- Integration
- Data Driven
- Code Coverage

Techniques

- Test-driven Development
- Mocking (fakes & stubs)
- Continuous Integration
- GUI Testing (Selenium)

Unit testing

Article Talk

From Wikipedia, the free encyclopedia

In computer programming, **unit testing** is a software testing method by which individual units of source code—sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures—are tested to determine whether they are fit for use.[1] It is a standard step in development and implementation approaches such as Agile.

Procedural programming [edit]

In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure.

Object-oriented programming [edit]

In object-oriented programming, a unit is often an entire interface, such as a class, or an individual method.[5] By writing tests first for the smallest testable units, then the compound behaviors between those, one can build up comprehensive tests for complex applications.[4]

Regression testing

Article Talk

From Wikipedia, the free encyclopedia

This article is about software development. For the statistical analysis process, see Regression analysis.

Regression testing (rarely, *non-regression testing*[1]) is re-running functional and non-functional tests to ensure that previously developed and tested software still performs as expected after a change.[2] If not, that would be called a *regression*.

Changes that may require regression testing include bug fixes, software enhancements, configuration changes, and even substitution of electronic components (hardware).[3] As regression test suites tend to grow with each found defect, test automation is frequently involved. The evident exception is the GUIs regression testing, which normally must be executed manually. Sometimes a change impact analysis is performed to determine an appropriate subset of tests (*non-regression analysis*[4]).

Integration testing (sometimes called **integration and testing**, abbreviated **I&T**) is the phase in software testing in which the whole software module is tested or if it consists of multiple software modules they are combined and then tested as a group. Integration testing is conducted to evaluate the compliance of a system or component with specified functional requirements.[1] It occurs after unit testing and before system testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.[2]

Data-driven testing

3 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

Data-driven testing (**DDT**), also known as **table-driven testing** or **parameterized testing**, is a software testing methodology that is used in the testing of computer software to describe testing done using a table of conditions directly as test inputs and verifiable outputs as well as the process where test environment settings and control are not hard-coded.[1][2] In the simplest form the tester supplies the inputs from a row in the table and expects the outputs which occur in the same row. The table typically contains values which correspond to boundary or partition input spaces. In the control methodology, test configuration is "read" from a database.

Introduction [edit]

In the testing of software or programs, several methodologies are available for implementing this testing. Each of these methods co-exist because they differ in the effort required to create and subsequently maintain. The advantage of Data-driven testing is the ease to add additional inputs to the table when new partitions are discovered or added to the product or system under test. Also, in the data-driven testing process, the test environment settings and control are not hard-coded. The cost aspect makes DDT cheap for automation but expensive for manual testing.

Test-driven development

Article Talk

From Wikipedia, the free encyclopedia

Test-driven development (**TDD**) is a software development process relying on software requirements being converted to test cases before software is fully developed, and tracking all software development by repeatedly testing the software against all test cases. This is as opposed to software being developed first and test cases created later.

Software engineer Kent Beck, who is credited with having developed or "rediscovered"[1] the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.[2]

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999,[3] but more recently has created more general interest in its own right.[4]

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.[5]

Test-driven development cycle [edit]

The following sequence is based on the book *Test-Driven Development by Example*:[2]

A graphical representation of the test-driven development lifecycle

1. Add a test

The adding of a new feature begins by writing a test that passes iff the feature's specifications are met. The developer can discover these specifications by asking about use cases and user stories. A key benefit of test-driven development is that it makes the developer focus on requirements *before* writing code. This is in contrast with the usual practice, where unit tests are only written *after* code.

2. Run all tests. The new test *should fail* for expected reasons

This shows that new code is actually needed for the desired feature. It validates that the test harness is working correctly It rules out the possibility that the new test is flawed and will always pass.

3. Write the simplest code that passes the new test

Inelegant or hard code is acceptable, as long as it passes the test. The code will be honed anyway in Step 5. No code should be added beyond the tested functionality.

4. All tests should now pass

If any fail, the new code must be revised until they pass. This ensures the new code meets the test requirements and does not break existing features.

5. Refactor as needed, using tests after each refactor to ensure that functionality is preserved

Code is refactored for readability and maintainability. In particular, hard-coded test data should be removed. Running the test suite after each refactor helps ensure that no existing functionality is broken.

Mock object

15 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

In object-oriented programming, **mock objects** are simulated objects that mimic the behaviour of real objects in controlled ways, most often as part of a software testing initiative. A programmer typically creates a mock object to test the behaviour of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behaviour of a human in vehicle impacts. The technique is also applicable in generic programming.

Motivation [edit]

In a unit test, mock objects can simulate the behavior of complex, real objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test. If an object has any of the following characteristics, it may be useful to use a mock object in its place:

- the object supplies non-deterministic results (e.g. the current time or the current temperature);
- it has states that are difficult to create or reproduce (e.g. a network error);
- it is slow (e.g. a complete database, which would have to be prepared before the test);
- it does not yet exist or may change behavior;
- it would have to include information and methods exclusively for testing purposes (and not for its actual task).

For example, an alarm clock program which causes a bell to ring at a certain time might get the current time from a time service. To test this, the test must wait until the alarm time to know whether it has rung the bell correctly. If a mock time service is used in place of the real time service, it can be programmed to provide the bell-ringing time (or any other time) regardless of the real time, so that the alarm clock program can be tested in isolation.

In software engineering, **continuous integration** (**CI**) is the practice of merging all developers' working copies to a shared mainline several times a day.[1] Nowadays it is typically implemented in such a way that it triggers an automated build with testing. Grady Booch first proposed the term CI in his 1991 method,[2] although he did not advocate integrating several times a day. Extreme programming (XP) adopted the concept of CI and did advocate integrating more than once per day – perhaps as many as tens of times per day.[3]

In software engineering, **code coverage** is a percentage measure of the degree to which the source code of a program is executed when a particular test suite is run. A program with high test coverage has more of its source code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to a program with low test coverage.[1][2] Many different metrics can be used to calculate test coverage. Some of the most basic are the percentage of program subroutines and the percentage of program statements called during execution of the test suite.

Test coverage was among the first methods invented for systematic software testing. The first published reference was by Miller and Maloney in *Communications of the ACM*, in 1963.[3]

Coverage criteria [edit]

To measure what percentage of code has been executed by a test suite, one or more *coverage criteria* are used. These are usually defined as rules or requirements, which a test suite must satisfy.[4]

Basic coverage criteria [edit]

There are a number of coverage criteria, but the main ones are:[5]

- **Function coverage** – has each function (or subroutine) in the program been called?
- **Statement coverage** – has each statement in the program been executed?
- **Edge coverage** – has every edge in the control-flow graph been executed?
  - **Branch coverage** – has each branch (also called the DD-path) of each control structure (such as in *if* and *case* statements) been executed? For example, given an *if* statement, have both the *true* and *false* branches been executed? (This is a subset of edge coverage.)
- **Condition coverage** – has each Boolean sub-expression evaluated both to true and false? (Also called predicate coverage.)

What about primitives with switches "built in"?

x←|÷y

Test with y positive, negative and zero?

Code coverage is necessary but NOT sufficient.

Test Frameworks for APL

"Unit Test" Frameworks

- https://github.com/**Gianfrancoalongi/APLUnit**
  - A "classical" Unit Test framework, inspired by non-APL frameworks
- https://**xpqz**.github.io/**learnapl**/testing.html
  - A more pragmatic and APL-friendly approach.

Other Test Frameworks

- **DTest** (DyalogTest) – an internal tool used at Dyalog, that is included with Dyalog APL
- **davin-Tester** – A Tatin Package by Davin Church
- **aplteam-Tester2** – Tatin Package by Kai Jaeger, used to test many of Kai's tools
- **aplteam-CodeCoverage** – Tatin package for measuring code coverage

Do you/we know of others?

https://github.com/Gianfrancoalongi/APLUnit

```
:NameSpace Demo_tests

∇ Z←count_comments_TEST;input
  input←0 1 2/⊂'int a;'
  input,←1 2/⊂'// this is a comment'
  input,←⊂'  // comment with leading blank'
  #.UT.expect←0 0 0,1 2,1
  Z←#.Demo.count_comments⍤1⊢input
∇

:EndNameSpace
```

Code | Issues 7 | Pull requests | Actions | Projects | Wiki | Security | Insights

Files

master

Go to file

- Pages
- .gitignore
- Demo.dyalog
- Demo_tests.dyalog
- Empty_tests.dyalog
- README.md
- UT.dyalog
- UTFile.dyalog
- UTTests2_tests.dyalog
- UT_tests.dyalog
- requirement_specificati...

APLUnit / Demo_tests.dyalog

Gianfrancoalongi Added demo files for the coverage of a full file

Code | Blame | 33 lines (26 loc) · 792 Bytes

```
1  :NameSpace Demo_tests
2
3  ∇ Z ← count_zero_comments_from_no_input_TEST
4    #.UT.expect ← 0
5    Z ← #.Demo.count_comments 0
6  ∇
7
8  ∇ Z ← count_zero_comments_from_single_normal_line_TEST
9    #.UT.expect ← 0
10   Z ← #.Demo.count_comments 'int a;'
11 ∇
12
13 ∇ Z ← count_one_comment_from_single_line_TEST
14   #.UT.expect ← 1
15   Z ← #.Demo.count_comments '//this is a comment'
16 ∇
17
18 ∇ Z ← count_one_comment_which_has_leading_whitespace_TEST
19   #.UT.expect ← 1
20   Z ← #.Demo.count_comments ' //this is a comment'
21 ∇
22
23 ∇ Z ← count_no_comments_for_a_blank_line_TEST
24   #.UT.expect ← 0
25   Z ← #.Demo.count_comments '   '
26 ∇
27
28 ∇ Z ← count_multiple_comments_amongst_lines_TEST
29   #.UT.expect ← 2
30   Z ← #.Demo.count_comments '//first comment'  '//second comment'
31 ∇
32
33 :EndNameSpace
```

https://xpqz.github.io/learnapl/testing.html

```
:Namespace unittest
    ⎕IO ← 0
    run←{
        tests ← 'test_.+'⎕S'&'⎕NL ¯3
        0=≢tests: 'no tests found'
        ↑{⍺,('.'/⍨30-≢⍺),⍵⊃'[FAIL]' '[OK]'}⌿↑tests (⍷¨tests,¨⊂' ⍬')
    }
:EndNamespace
```

```
unittest.test_upper←{'FOO'≡#.upper 'foo'}
```

https://xpqz.github.io/learnapl/testing.html

... also contains a "framework" for data-driven testing:

```
:Namespace datatest
    ⎕IO ← 0
    _test←{(⊢/↑ω)≡⍤0 0⊢⍺⍺/¯1↓⍤1⊢↑ω}
    run←{ ⍝ ω -- ns containing functions to be tested
        params ← '_'(≠⊆⊢)¨'[^_]+_testdata'⎕S'&'⎕NL¯2.1    ⍝ https://aplcart.info/?q=%E2%8E%
        0=≢params: 'no test parameter sets found'
        funs ← ⊃¨↓¯1↓⍤1↑params                             ⍝ Corresponding functions defined
        testable ← funs/⍨funs∈ω.⎕NL¯3
        result←ω⍎¨{(⍺.⍎ω)_test ⍎ω,'_testdata'}¨testable    ⍝ Run the tests
        ↑{⍺,('.'/⍨30-≢⍺),'[',(⍕⍵+/ω),'/',(⍕≢ω),']'}⌿↑testable result ⍝ Format
    }
:EndNamespace
```

```
datatest.split_testdata←(' ' ('hello world') ('hello' 'world')) (',' ('hello,world') ('hell
datatest.isupper_testdata←(⍬ ('FOO') 1) (⍬ ('Foo') 0) (⍬ (,'F') 1) ⍝ monadic function
```

Some Recent QA we have written...

- Ullu: Testing APL primitives
- Kamila's tests
- Link Testing
- Selenium

(Michael will show some examples based on DTest in the next section)

https://github.com/Dyalog/ullu

readme.md

license MIT

ullu

> A test suite to test APL Primitives.

What is ullu?

Ullu is a QA for DyalogAPL (can be used to test any APL) which tests specifically the functionality of primitives.

Coverage

Available Tests

- floor (monadic ⌊)
- index of (dyadic ⍳)
- magnitude (monadic ⌈)
- membership (dyadic ∈)
- residue (dyadic ⌈)

Ullu

Id & Comment

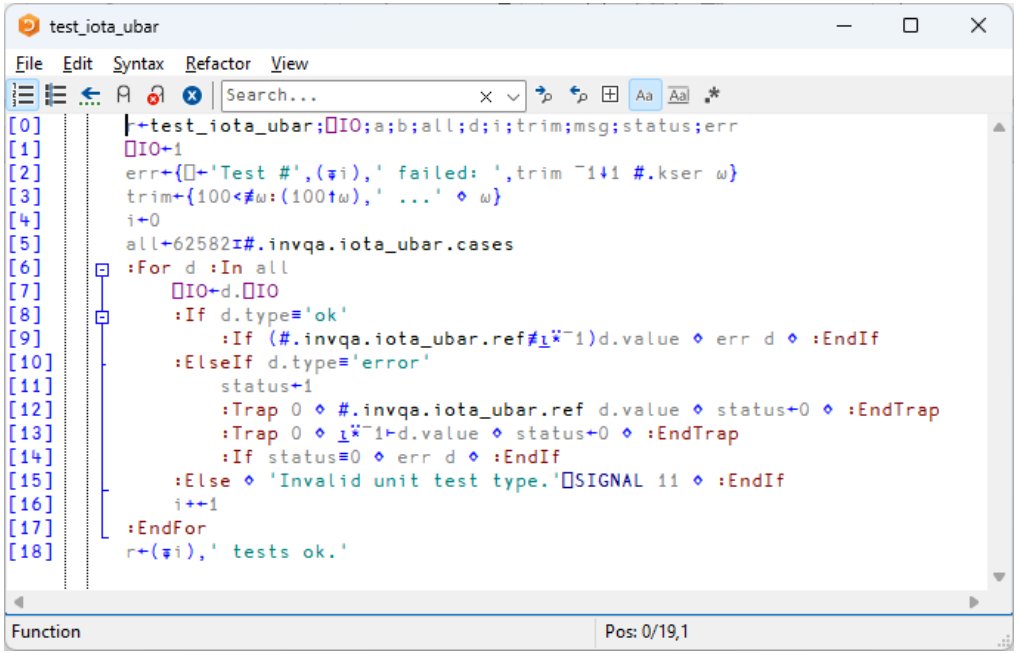
```
:Namespace unittest
    GetTests←{ ⍝ ω is a ref to a namespace containing functions called test_*
        tests←'test_.+'⎕S'&'⍠'ω.⎕NL ¯3
        tests←('.', ⍕⍎⍵)∘, ¨tests
        tests
    }

    FAIL_OK←'[FAIL]' '[OK]' ⍝ 1+bool will give fail and ok on 0 and 1

    ⍝ Pretty print test result
    PPTestResult←{⍵[2], ⍵[3], ': ', FAIL_OK[1+⍵[1]]}

    ∇ r←tData Assert r ;r;tID;tCmt ⍝ to output result of tests
      (tID tCmt)←tData
      ⎕RL←rl ⍝ Reset ⎕RL after each test
      :If (~r)∧stop
          PPTestResult r tID tCmt
          'Stopping on failure of:' ⎕SIGNAL 500
      :EndIf
      :If verbose
          PPTestResult r tID tCmt
      :ElseIf ~r∨stop
          PPTestResult r tID tCmt
      :EndIf
    ∇
```

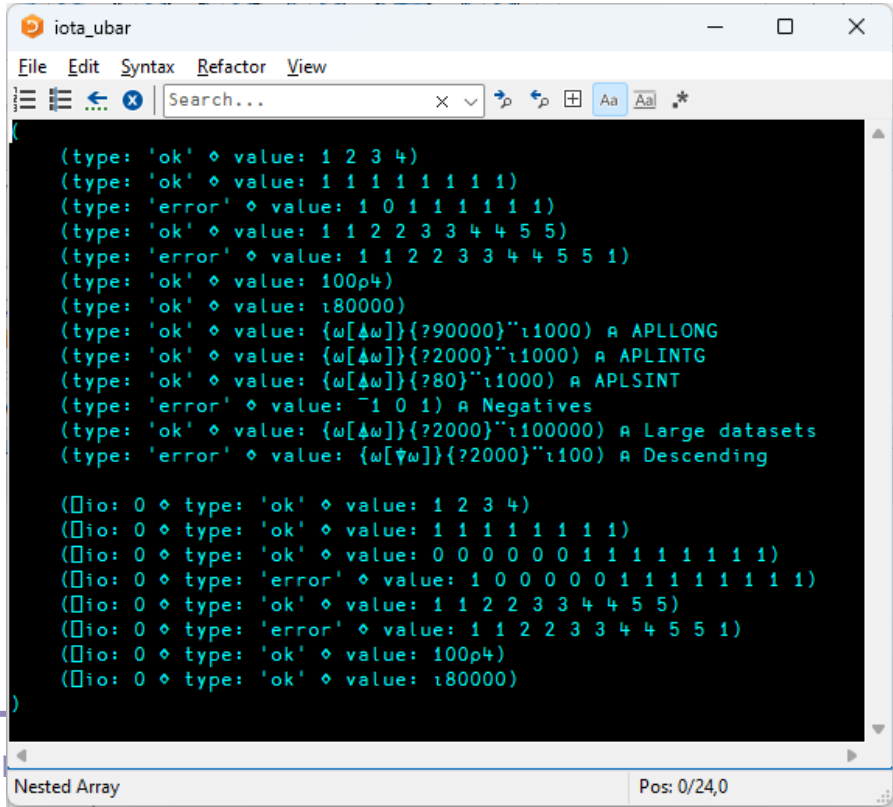
ullu / tests / magnitude.apln

sloorush fix typo in magnitude

Code | Blame | 116 lines (99 loc) · 5.46 KB

```
1   ⍝ This Namespace includes tests for the function Magnitude which is represented by Monadic stile(|)
2   ⍝
3   ⍝ Magnitude:
4   ⍝ Y may be any numeric array. R is numeric composed of the absolute (unsigned) values of Y.
5   ⍝ Note that the magnitude of a complex number a+ιb is defined to be √(a^2+b^2).
6   :Namespace magnitude
7       Assert←#.unittest.Assert
8
9       ⍝ Run Variations of each test with normal, empty and multiple shaped data
10      ∇ tRes←tData RunVariations exp ;actualR;actualRE;expectedR;left;right;res;tID;tCmt;p;shape;shapeW0;actualRS
11          (expectedR p)←exp
12          (tID tCmt)←tData
13          tRes←0
14
15          ⍝ normal
16          actualR←|p
17          tRes,←tData Assert expectedR≡actualR
18
19          ⍝ empty
20          actualRE←|(0pp) ⍝ 0 in the shape means we have no elements in the array, i.e. it's empty.
21          tRes,←('Empty',tID) tCmt Assert 0≡actualRE
22
23          ⍝ different shapes
24          shape←?(?4)/4
25          actualRS←|(shapepactualR)
26          tRes,←('Multiple',tID) tCmt Assert (shapepexpectedR)≡actualRS
```



test_iota_ubar
File Edit Syntax Refactor View
Search...
[0] ⍝←test_iota_ubar;⎕IO;a;b;all;d;i;trim;msg;status;err
[1] ⎕IO←1
[2] err←{⍺←'Test #',(⍕i),' failed: ',trim ¯1↓1 #.kser ⍵}
[3] trim←{100<≢⍵:(100↑⍵),' ...' ◇ ⍵}
[4] i←0
[5] all←62582⌶#.invqa.iota_ubar.cases
[6] :For d :In all
[7] ⎕IO←d.⎕IO
[8] :If d.type≡'ok'
[9] :If (#.invqa.iota_ubar.ref≠⍳⍤¯1)d.value ◇ err d ◇ :EndIf
[10] :ElseIf d.type≡'error'
[11] status←1
[12] :Trap 0 ◇ #.invqa.iota_ubar.ref d.value ◇ status←0 ◇ :EndTrap
[13] :Trap 0 ◇ ⍳⍤¯1⊢d.value ◇ status←0 ◇ :EndTrap
[14] :If status≡0 ◇ err d ◇ :EndIf
[15] :Else ◇ 'Invalid unit test type.'⎕SIGNAL 11 ◇ :EndIf
[16] i+←1
[17] :EndFor
[18] r←(⍕i),' tests ok.'
Function
Pos: 0/19,1


iota_ubar
File Edit Syntax Refactor View
Search...
(type: 'ok' ◇ value: 1 2 3 4)
(type: 'ok' ◇ value: 1 1 1 1 1 1 1 1)
(type: 'error' ◇ value: 1 0 1 1 1 1 1 1)
(type: 'ok' ◇ value: 1 1 2 2 3 3 4 4 5 5)
(type: 'error' ◇ value: 1 1 2 2 3 3 4 4 5 5 1)
(type: 'ok' ◇ value: 100⍴4)
(type: 'ok' ◇ value: ⍳80000)
(type: 'ok' ◇ value: {⍵[⍋⍵]}{?90000}¨⍳1000) ⍝ APLLONG
(type: 'ok' ◇ value: {⍵[⍋⍵]}{?2000}¨⍳1000) ⍝ APLINTG
(type: 'ok' ◇ value: {⍵[⍋⍵]}{?80}¨⍳1000) ⍝ APLSINT
(type: 'error' ◇ value: ¯1 0 1) ⍝ Negatives
(type: 'ok' ◇ value: {⍵[⍋⍵]}{?2000}¨⍳100000) ⍝ Large datasets
(type: 'error' ◇ value: {⍵[⍒⍵]}{?2000}¨⍳100) ⍝ Descending
(⎕io: 0 ◇ type: 'ok' ◇ value: 1 2 3 4)
(⎕io: 0 ◇ type: 'ok' ◇ value: 1 1 1 1 1 1 1 1)
(⎕io: 0 ◇ type: 'ok' ◇ value: 0 0 0 0 0 0 1 1 1 1 1 1 1 1)
(⎕io: 0 ◇ type: 'error' ◇ value: 1 0 0 0 0 0 1 1 1 1 1 1 1 1)
(⎕io: 0 ◇ type: 'ok' ◇ value: 1 1 2 2 3 3 4 4 5 5)
(⎕io: 0 ◇ type: 'error' ◇ value: 1 1 2 2 3 3 4 4 5 5 1)
(⎕io: 0 ◇ type: 'ok' ◇ value: 100⍴4)
(⎕io: 0 ◇ type: 'ok' ◇ value: ⍳80000)
Nested Array
Pos: 0/24,0

test_invqa_pot

File Edit Syntax Refactor View

Search...

```
[0]   r←test_invqa_pot;⎕IO;e1;e2;a;b;all;d;i;trim;msg;status;err
[1]   ⎕IO←1
[2]   err←{⎕←'Test #',(⍕i),' failed: ',trim ¯1↓1 #.kser ⍵}
[3]   trim←{100<≢⍵:(100↑⍵),' ...' ◇ ⍵}
[4]   i←0
[5]   all←62582⌶#.invqa.pot.cases
[6]   :For d :In all
[7]       ⎕IO←d.⎕IO
[8]       :If (⊂,'l')≡d.⎕NL-⍳9
[9]           e1←d.l(d.a)d.r
[10]          e2←d.l(d.b)d.r
[11]          :If e1≢e2 ◇ err d ◇ :EndIf
[12]      :Else
[13]          e1←(d.a)d.r
[14]          e2←(d.b)d.r
[15]          :If e1≢e2 ◇ err d ◇ :EndIf
[16]      :EndIf
[17]      i+←1
[18]  :EndFor
[19]  r←(⍕i),' tests ok.'
```

Function | Pos: 0/20,1

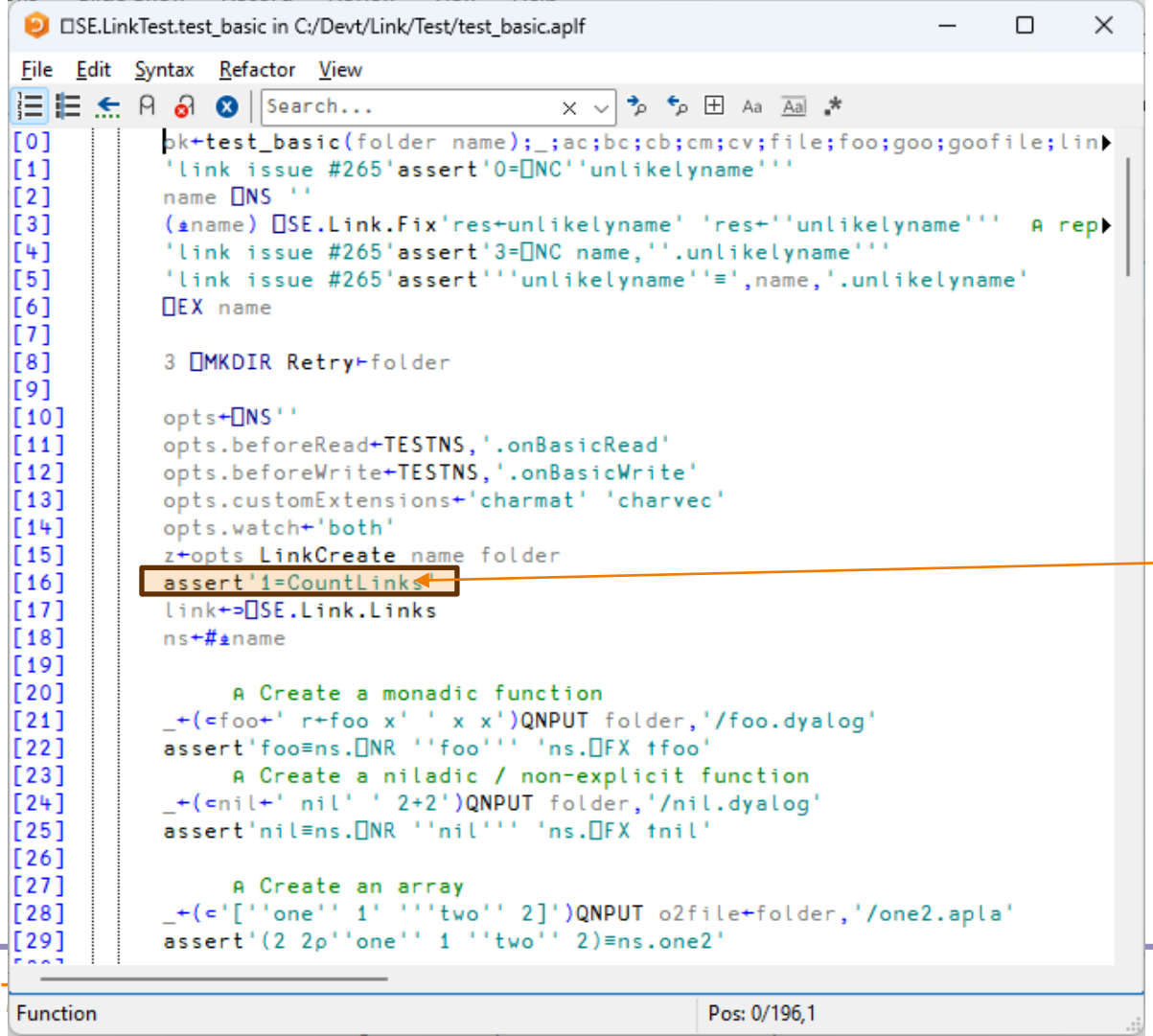
invqa_pot

File Edit Syntax Refactor View

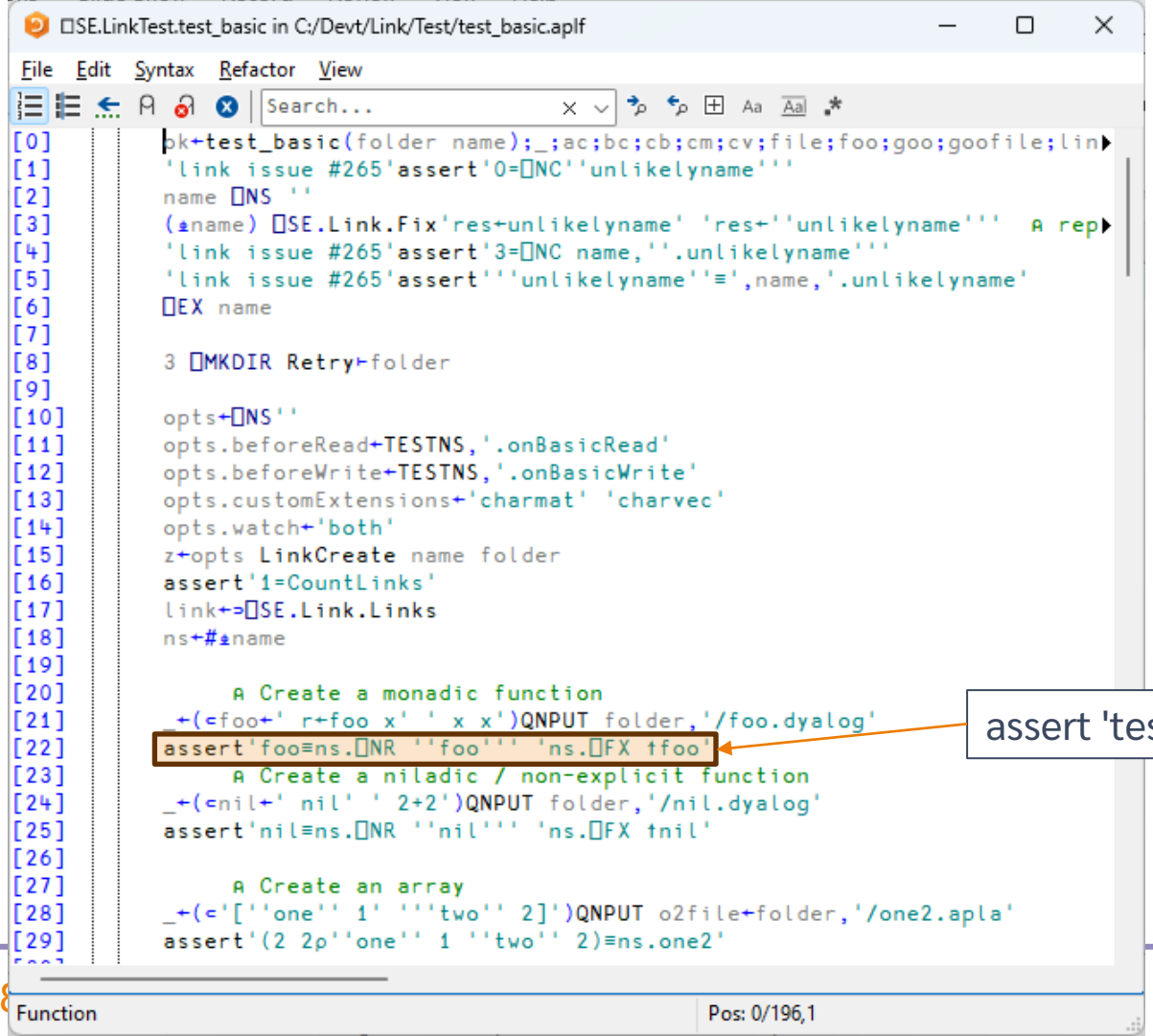
Search...

```
(
    ⍝ Reverse Compose
    (l: 3 ¯1 4 ◇ r: ¯2 ¯7 1
     a: ×∘.×∘⌽ ◇ b: {(×⍺)×⌽⍵})
    (l: 3 ◇ r: 2
     a: ×⍨∘.- ◇ b: {(⍺×⍺)-⍵})
    (l: 2 ◇ r: 4
     a: ÷∘.,- ◇ b: {(÷(⍺-⍵)),(⍺-⍵)})
    ⍝ Sort (≤, ≥)
    (r: 1 8 6 9 2 7 3
     a: ≤ ◇ b: {(⊂⍋⍵)⌷⍵})
    (r: 1 8 6 9 2 7 3
     a: ≥ ◇ b: {(⊂⍒⍵)⌷⍵})
    (r: 0
     a: ≥ ◇ b: {(⊂⍒⍵)⌷⍵})
    (r: 1e¯20+⍳10
     a: ≥ ◇ b: {(⊂⍒⍵)⌷⍵})
    (r: 1e¯10+⍳10
     a: ≤ ◇ b: {(⊂⍋⍵)⌷⍵})
    ⍝ Obverse
    (r: 5 ◇ a: × ⍥⍣OBVERSE - ◇ b: ×)
    (r: 5 ◇ a: (× ⍥⍣OBVERSE -)⍣¯1 ◇ b: -)
    (r: 5 ◇ a: (÷⍟) ⍥⍣OBVERSE (⍟÷) ◇ b: ÷⍟)
    (r: 5 ◇ a: ((÷⍟) ⍥⍣OBVERSE (⍟÷))⍣¯1 ◇ b: ⍟÷)
    (r: 1+3 3⍴⍳9 ◇ a: (÷∘⍟) ⍥⍣OBVERSE (⍟∘÷) ◇ b: ÷⍟)
    (r: 1+3 3⍴⍳9 ◇ a: ((÷∘⍟) ⍥⍣OBVERSE (⍟∘÷))⍣¯1 ◇ b: ⍟÷)
    (r: 1+3 3⍴⍳9 ◇ a: (÷+⍟) ⍥⍣OBVERSE (⍟+÷) ◇ b: ÷+⍟)
    (r: 1+3 3⍴⍳9 ◇ a: ((÷+⍟) ⍥⍣OBVERSE (⍟+÷))⍣¯1 ◇ b: ⍟+÷)
    ⍝ Monadic ⍤: Rank
```

Link Testing


SE.LinkTest.test_basic in C:/Devt/Link/Test/test_basic.aplf
File Edit Syntax Refactor View
Search...
[0] ⍝k←test_basic(folder name);_;ac;bc;cb;cm;cv;file;foo;goo;goofile;lin
[1] 'link issue #265'assert'0=⎕NC''unlikelyname'''
[2] name ⎕NS ''
[3] (⍎name) ⎕SE.Link.Fix'res←unlikelyname' 'res←''unlikelyname''' ⍝ rep
[4] 'link issue #265'assert'3=⎕NC name,''.unlikelyname'''
[5] 'link issue #265'assert'''unlikelyname''≡',name,'.unlikelyname'
[6] ⎕EX name
[7]
[8] 3 ⎕MKDIR Retry⊢folder
[9]
[10] opts←⎕NS''
[11] opts.beforeRead←TESTNS,'.onBasicRead'
[12] opts.beforeWrite←TESTNS,'.onBasicWrite'
[13] opts.customExtensions←'charmat' 'charvec'
[14] opts.watch←'both'
[15] z←opts LinkCreate name folder
[16] assert'1=CountLinks'
[17] link←⎕SE.Link.Links
[18] ns←#⍎name
[19]
[20] ⍝ Create a monadic function
[21] _←(⊂foo←' r←foo x' ' x x')⎕NPUT folder,'/foo.dyalog'
[22] assert'foo≡ns.⎕NR ''foo''' 'ns.⎕FX ⍕foo'
[23] ⍝ Create a niladic / non-explicit function
[24] _←(⊂nil←' nil' ' 2+2')⎕NPUT folder,'/nil.dyalog'
[25] assert'nil≡ns.⎕NR ''nil''' 'ns.⎕FX ⍕nil'
[26]
[27] ⍝ Create an array
[28] _←(⊂'[''one'' 1' '''two'' 2]')⎕NPUT o2file←folder,'/one2.apla'
[29] assert'(2 2⍴''one'' 1 ''two'' 2)≡ns.one2'
Function
Pos: 0/196,1

assert 'test'



assert 'test' 'recovery-expression'

Test for expected errors

```
      ⍝ test failing creations
assert'{6::1 ⋄ 0=CountLinks}⍬'
3 ⎕NDELETE folder ⋄ ⎕EX name ⋄ opts.source←'dir'
assertError'opts LinkCreate name folder' 'Source directory not found'
```

Expression to run

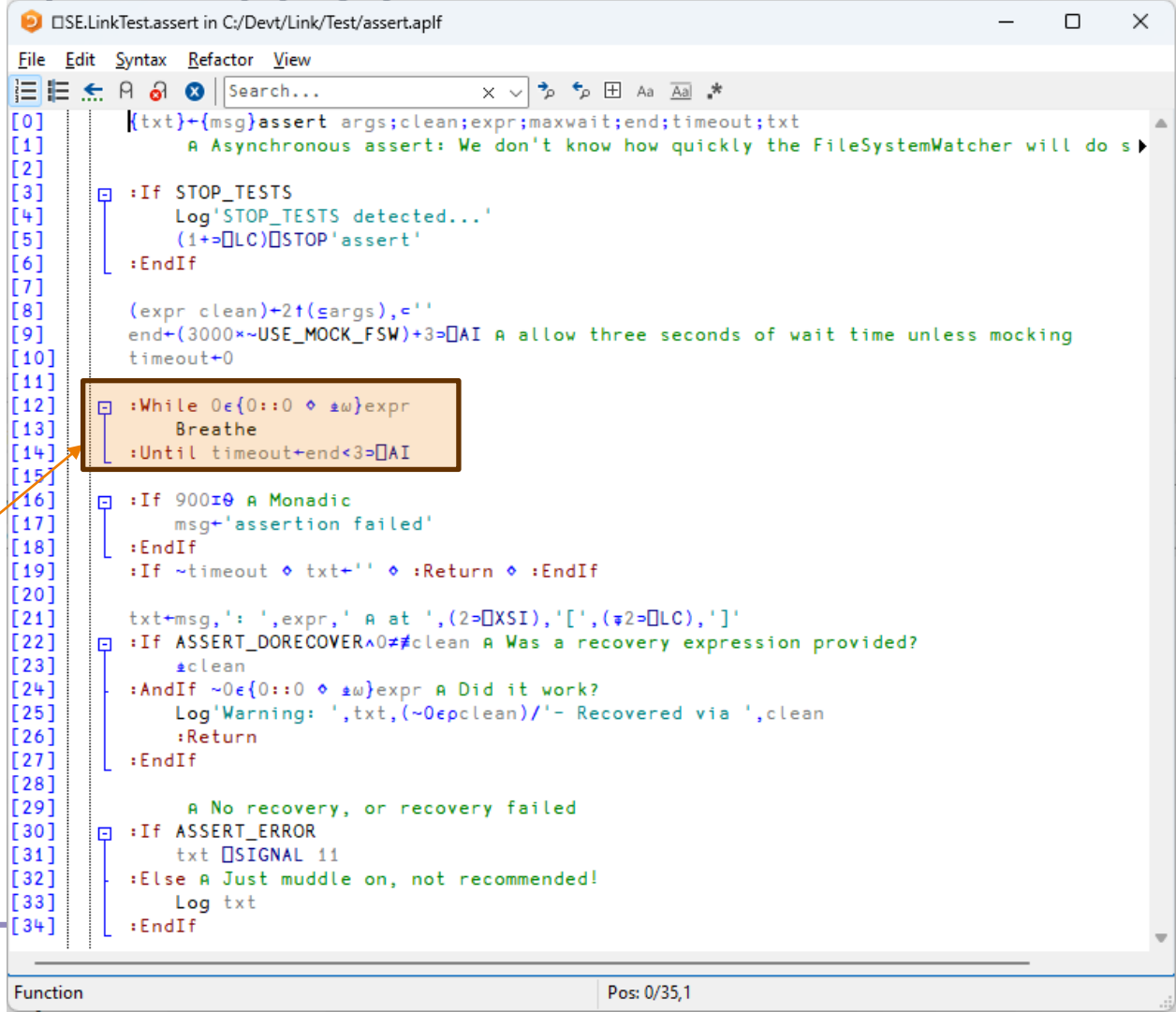
Text to find in ⎕DM

Mocking

- The Link QA needs to test Link's responses to notifications of additions, deletions, and changes to files
- File System Watcher cause callbacks to APL from .NET. These are:
  - Not processed until the end of the current thread time slice (so if QA script keeps running, it may be some time before the callback runs)
  - Potentially simultaneous: If one takes more than one time slice to process, the next callback may start running before the previous one is completed
- This is usually not a problem for the normal use case of editing or moving a small number of files outside APL "by hand"
- However, for a QA that makes hundreds or thousands of additions, deletions, moves and copies, it leads to intermittent, unpredictable failures

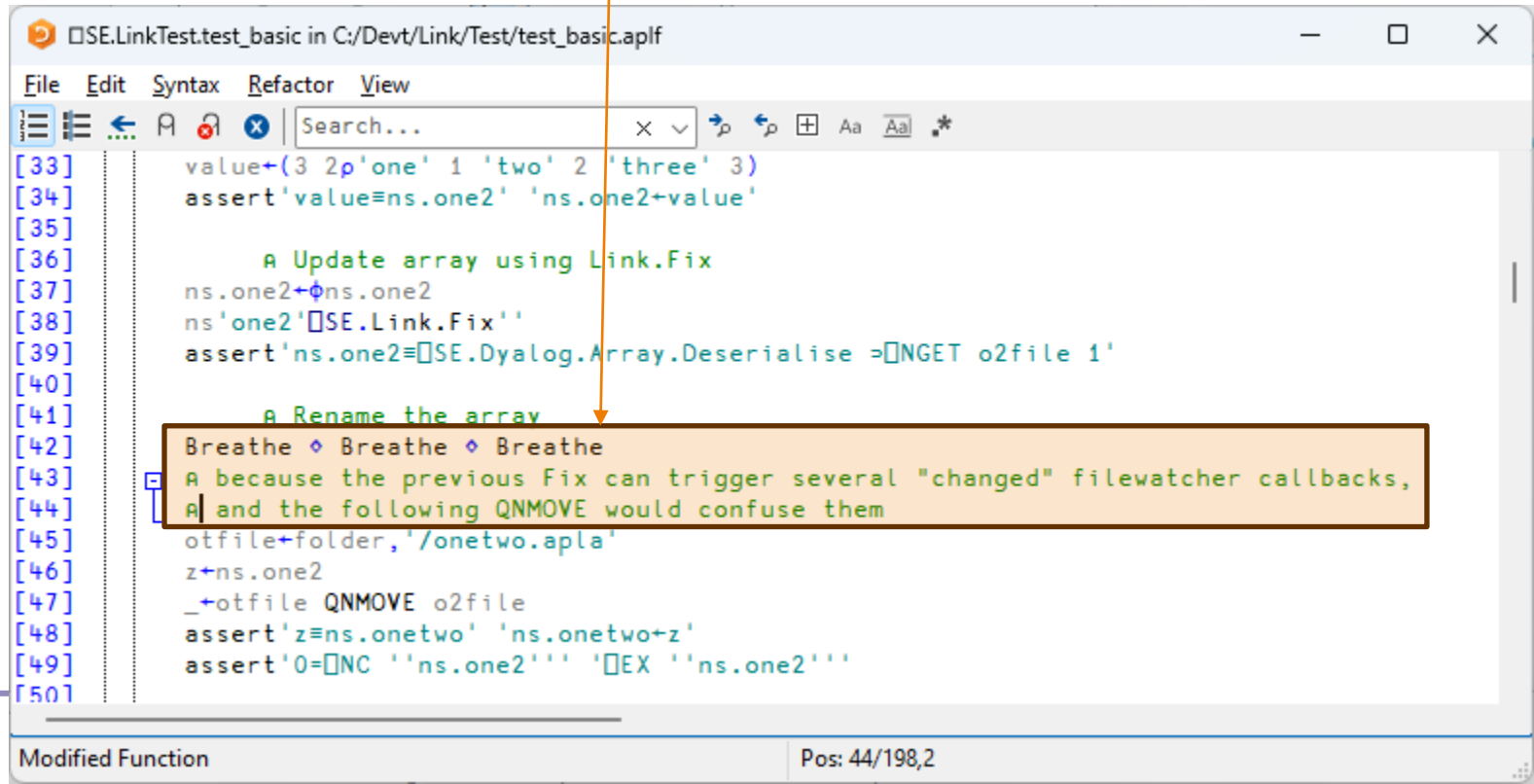
Keep trying until Event arrives and is processed.

Hence the ⍢



Mocking

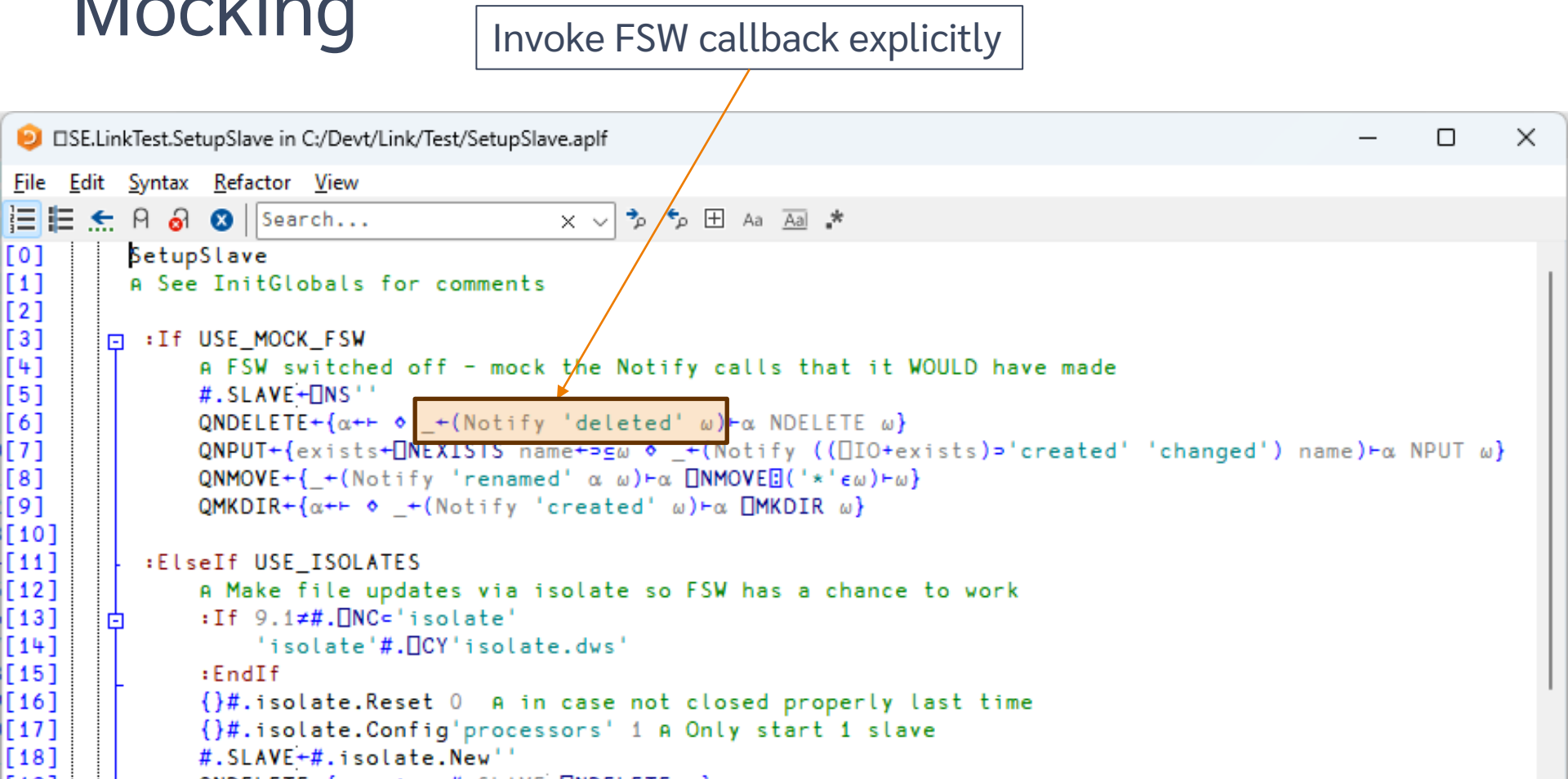
This also helped a bit



Mocking

- The out-of-order processing meant that delaying was not enough
  - Create-Update-Delete notifications might not arrive in that order
- It was ultimately impossible to get the Link QA to run reliably when using a real File System Watcher
- The solution was to "Mock" the FSW by covering all file system operations and call the FSW callback function immediately.
- This simulated a "synchronous" FSW and finally made the tests deterministic (after three years of messing about)

Mocking


Invoke FSW callback explicitly
⎕SE.LinkTest.SetupSlave in C:/Devt/Link/Test/SetupSlave.aplf
File Edit Syntax Refactor View
Search...
[0] SetupSlave
[1] ⍝ See InitGlobals for comments
[2]
[3] :If USE_MOCK_FSW
[4] ⍝ FSW switched off - mock the Notify calls that it WOULD have made
[5] #.SLAVE←⎕NS''
[6] QNDELETE←{α←⊢ ⋄ _←(Notify 'deleted' ω)⊢α NDELETE ω}
[7] QNPUT←{exists←⎕NEXISTS name←⊃⍵ ⋄ _←(Notify ((⎕IO+exists)⊃'created' 'changed') name)⊢α NPUT ω}
[8] QNMOVE←{_←(Notify 'renamed' α ω)⊢α ⎕NMOVE⍠('*'∊ω)⊢ω}
[9] QMKDIR←{α←⊢ ⋄ _←(Notify 'created' ω)⊢α ⎕MKDIR ω}
[10]
[11] :ElseIf USE_ISOLATES
[12] ⍝ Make file updates via isolate so FSW has a chance to work
[13] :If 9.1≠#.⎕NC⊂'isolate'
[14] 'isolate'#.⎕CY'isolate.dws'
[15] :EndIf
[16] {}#.isolate.Reset 0 ⍝ in case not closed properly last time
[17] {}#.isolate.Config'processors' 1 ⍝ Only start 1 slave
[18] #.SLAVE←#.isolate.New''

Asynchronous Effects / GUI Testing

- Sometimes, a test will trigger an effect which will take time to materialise
- We have seen how Link "assert" waited in a loop
- Automated GUI testing will nearly always exhibit this behaviour

DUI/MS3/QA/Examples/DC/Butt

https://github.com/Dyalog/DUI/blob/master/MS3/QA/Examples/DC/ButtonSimple.dyalog

Product Solutions Open Source Pricing

Search or jump to... /

Sign in Sign up

Dyalog / DUI Public

Notifications Fork 0 Star 6

Code Issues 3 Pull requests 2 Actions Projects Security Insights

Files

master

Go to file

- Examples
- IndexData
- Logs
- QA/Examples
  - Applications
  - DC
    - ASimple.dyalog
    - AudioSimple.dyalog

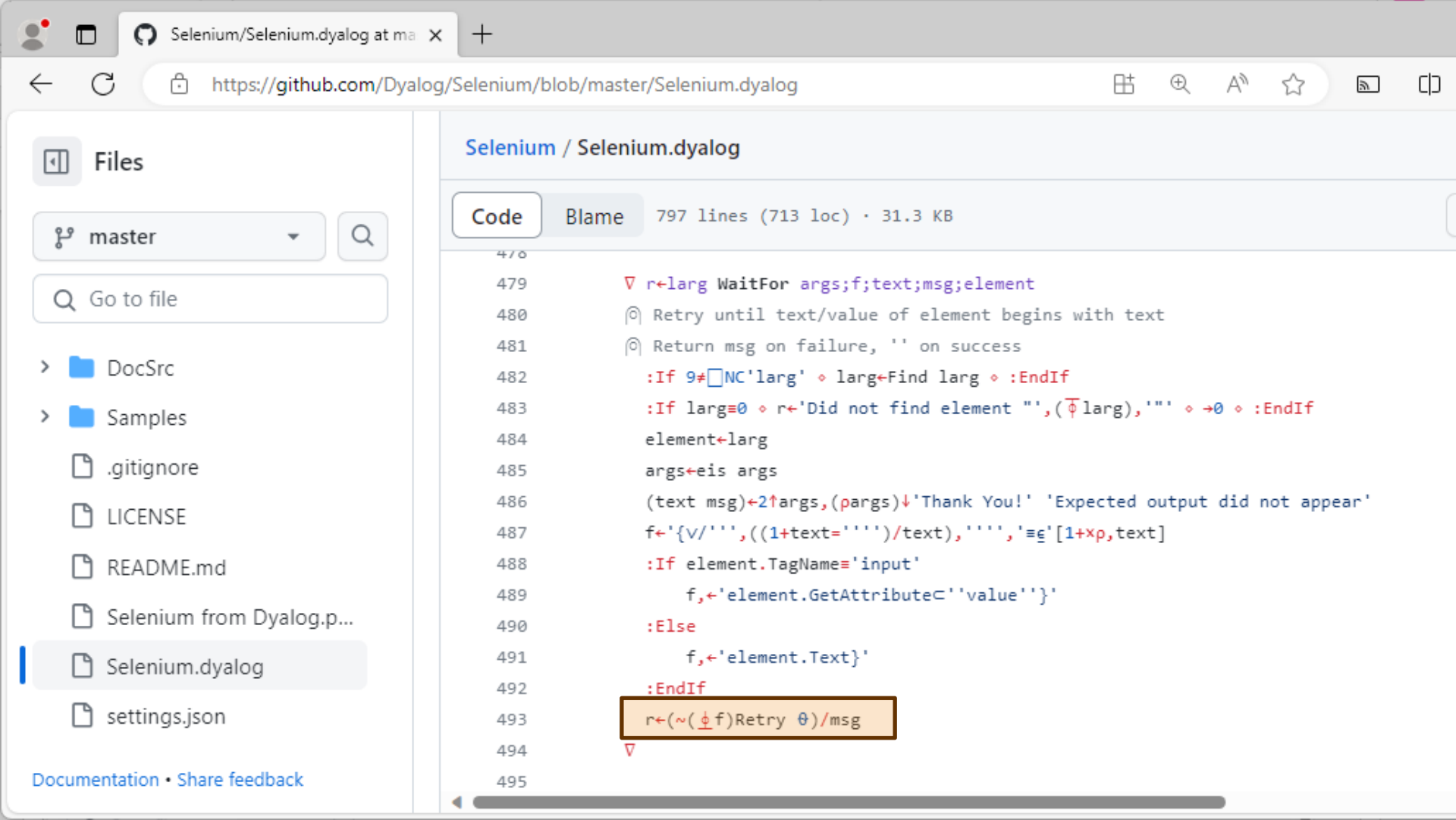
DUI / MS3 / QA / Examples / DC / ButtonSimple.dyalog

bpbecker Initial commit c4e0a9d · 5 years ago History

Code Blame 3 lines (3 loc) · 76 Bytes Raw

```
1  msg←Test dummy
2  Click'btnPressMe'
3  msg←'output' WaitFor 'Thank You!'
```

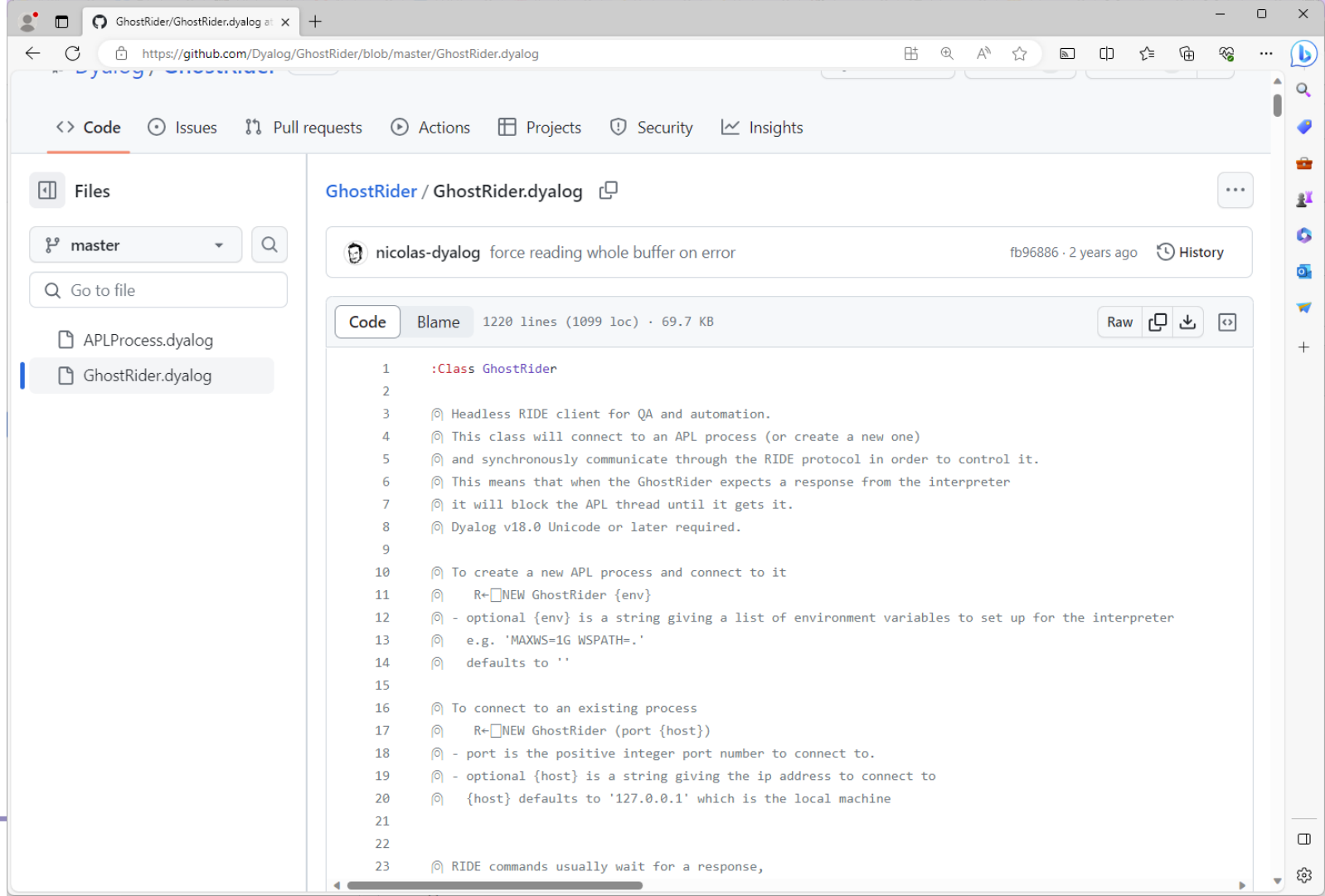
```
msg←Test dummy;data
⍝ Test /Examples/DC/InputGridSimple

data←'Morten' 'Kromberg'(⍕1↓⍵'/',¨⍕3↑⎕TS) ⍝ It's my birthday every day!
'fname' 'lname' 'bdate' SendKeys¨data
Click'ClickMe'
msg←'output'WaitFor'Hi Morten Kromberg. Happy Birthday!'
```


Selenium/Selenium.dyalog at ma
https://github.com/Dyalog/Selenium/blob/master/Selenium.dyalog
Files
master
Go to file
DocSrc
Samples
.gitignore
LICENSE
README.md
Selenium from Dyalog.p...
Selenium.dyalog
settings.json
Documentation • Share feedback
Selenium / Selenium.dyalog
Code
Blame
797 lines (713 loc) · 31.3 KB
479 ∇ r←larg WaitFor args;f;text;msg;element
480 ⍝ Retry until text/value of element begins with text
481 ⍝ Return msg on failure, '' on success
482 :If 9≠⎕NC'larg' ⋄ larg←Find larg ⋄ :EndIf
483 :If larg≡0 ⋄ r←'Did not find element "',(⍕larg),'"' ⋄ →0 ⋄ :EndIf
484 element←larg
485 args←eis args
486 (text msg)←2↑args,(⍴args)↓'Thank You!' 'Expected output did not appear'
487 f←'{⍵/''',((1+text='''')/text),'''','≡⍷'[1+×⍴,text]
488 :If element.TagName≡'input'
489 f,←'element.GetAttribute⊂''value''}'
490 :Else
491 f,←'element.Text}'
492 :EndIf
493 r←(~(⍎f)Retry ⍬)/msg
494 ∇
495

□WC – No idea how to test automatically

Driving Dyalog IDE

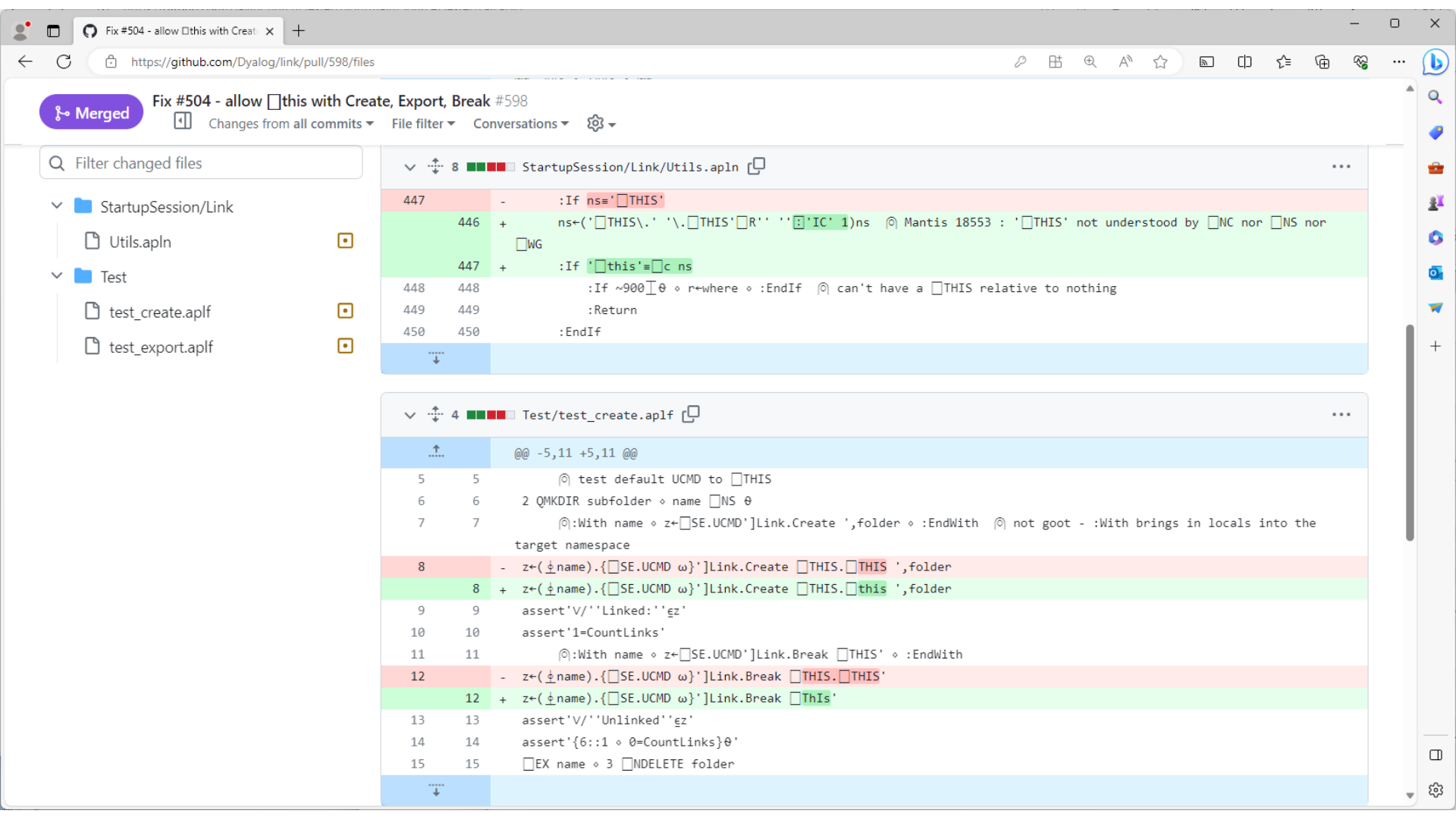
Drivin


GhostRider/GhostRider.dyalog
https://github.com/Dyalog/GhostRider/blob/master/GhostRider.dyalog
Code
Issues
Pull requests
Actions
Projects
Security
Insights
Files
master
Go to file
APLProcess.dyalog
GhostRider.dyalog
GhostRider / GhostRider.dyalog
nicolas-dyalog force reading whole buffer on error
fb96886 · 2 years ago
History
Code
Blame
1220 lines (1099 loc) · 69.7 KB
Raw
:Class GhostRider
⍝ Headless RIDE client for QA and automation.
⍝ This class will connect to an APL process (or create a new one)
⍝ and synchronously communicate through the RIDE protocol in order to control it.
⍝ This means that when the GhostRider expects a response from the interpreter
⍝ it will block the APL thread until it gets it.
⍝ Dyalog v18.0 Unicode or later required.
⍝ To create a new APL process and connect to it
⍝ R←⎕NEW GhostRider {env}
⍝ - optional {env} is a string giving a list of environment variables to set up for the interpreter
⍝ e.g. 'MAXWS=1G WSPATH=.'
⍝ defaults to ''
⍝ To connect to an existing process
⍝ R←⎕NEW GhostRider (port {host})
⍝ - port is the positive integer port number to connect to.
⍝ - optional {host} is a string giving the ip address to connect to
⍝ {host} defaults to '127.0.0.1' which is the local machine
⍝ RIDE commands usually wait for a response,

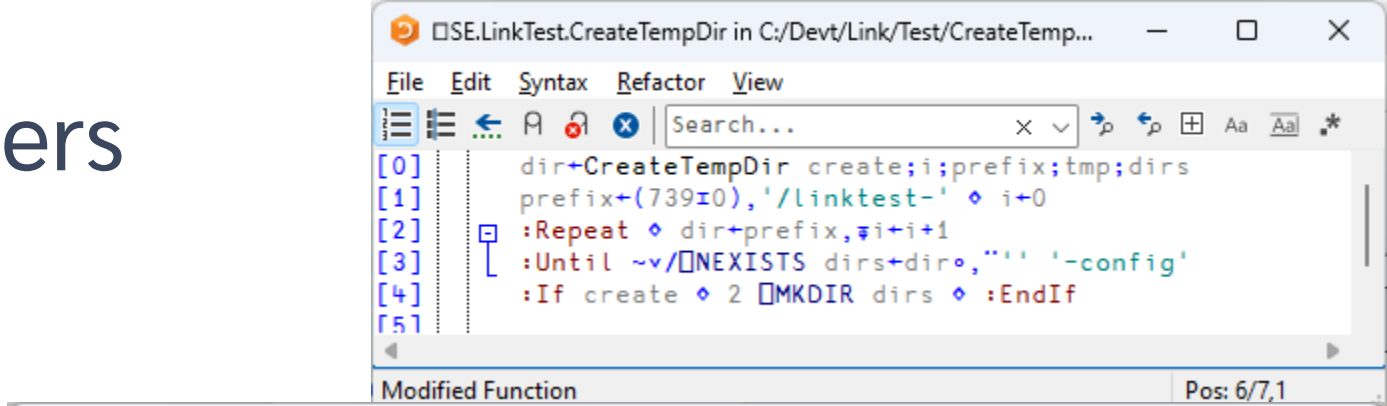
Test Driven Development

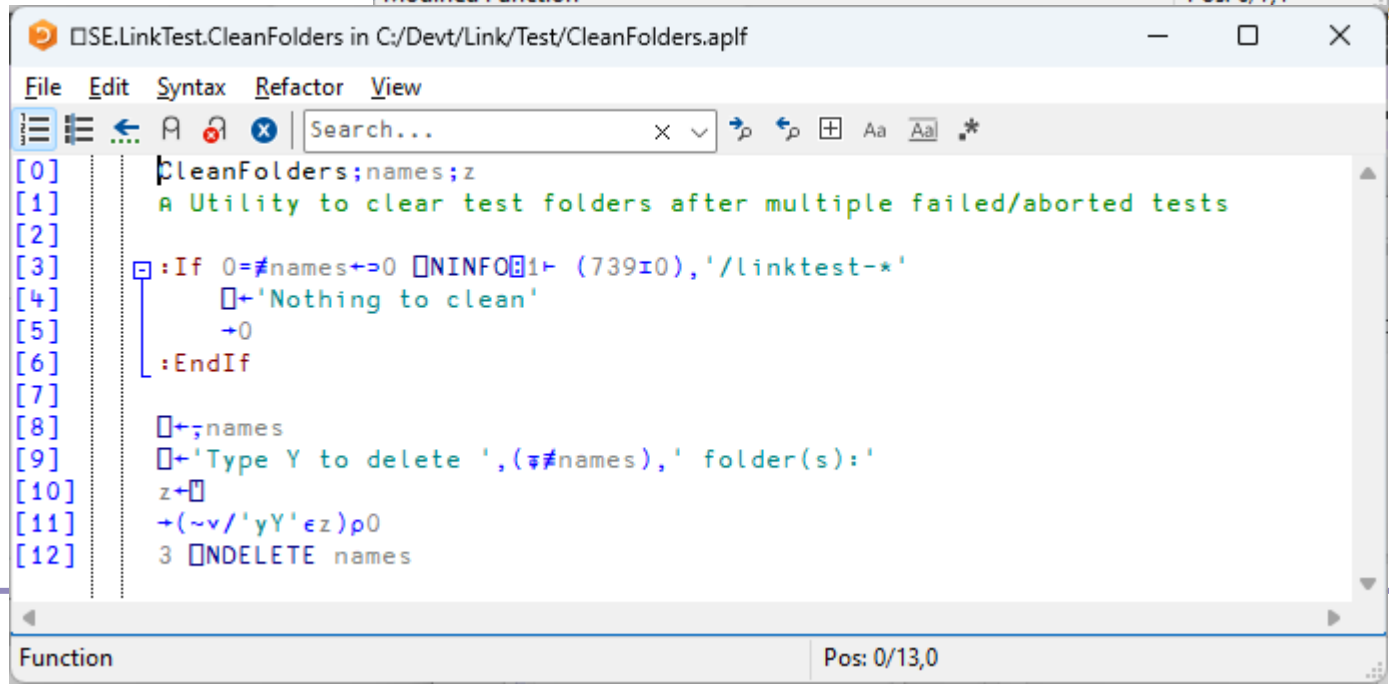
- Write tests BEFORE fixing the problem or adding the new functionality

- … or at least before you make the commits ☺


Fix #504 - allow ⎕this with Creat
https://github.com/Dyalog/link/pull/598/files
Merged
Fix #504 - allow ⎕this with Create, Export, Break #598
Changes from all commits
File filter
Conversations
Filter changed files
StartupSession/Link
Utils.apln
Test
test_create.aplf
test_export.aplf
8 StartupSession/Link/Utils.apln
447 - :If ns='⎕THIS'
446 + ns←('⎕THIS\.' '\.⎕THIS'⎕R'' ''⍠'IC' 1)ns ⍝ Mantis 18553 : '⎕THIS' not understood by ⎕NC nor ⎕NS nor ⎕WG
447 + :If '⎕this'≡⎕c ns
448 448 :If ~900⌶⍬ ⋄ r←where ⋄ :EndIf ⍝ can't have a ⎕THIS relative to nothing
449 449 :Return
450 450 :EndIf
4 Test/test_create.aplf
@@ -5,11 +5,11 @@
5 5 ⍝ test default UCMD to ⎕THIS
6 6 2 QMKDIR subfolder ⋄ name ⎕NS ⍬
7 7 ⍝:With name ⋄ z←⎕SE.UCMD']Link.Create ',folder ⋄ :EndWith ⍝ not goot - :With brings in locals into the target namespace
8 - z←(⍎name).{⎕SE.UCMD ⍵}']Link.Create ⎕THIS.⎕THIS ',folder
8 + z←(⍎name).{⎕SE.UCMD ⍵}']Link.Create ⎕THIS.⎕this ',folder
9 9 assert'V/''Linked:''⊆z'
10 10 assert'1=CountLinks'
11 11 ⍝:With name ⋄ z←⎕SE.UCMD']Link.Break ⎕THIS' ⋄ :EndWith
12 - z←(⍎name).{⎕SE.UCMD ⍵}']Link.Break ⎕THIS.⎕THIS'
12 + z←(⍎name).{⎕SE.UCMD ⍵}']Link.Break ⎕ThIs'
13 13 assert'V/''Unlinked''⊆z'
14 14 assert'{6::1 ⋄ 0=CountLinks}⍬'
15 15 ⎕EX name ⋄ 3 ⎕NDELETE folder

Temp Folders





Observed APL Practices

(Small) Unit Testing is expensive	APL functions are more like complete modules in other languages
Data Driven Regression Testing is common	Generating lots of test data in APL is easy
Continuous Integration	On the rise in APL!

Framework Requirement Spec

- Assert
  - Bool rarg of built-in ≡
  - How to identify failing test
  - Async capability?
- Expect Specific Error
  - EN or DM text
- Logging levels
  - Error / Warning
  - Verbose / Quiet
- Stopping behaviour
- Record Random Seed
  - Log/report it on failure
- Temporary folder creation
  - … And cleanup?
- Code coverage

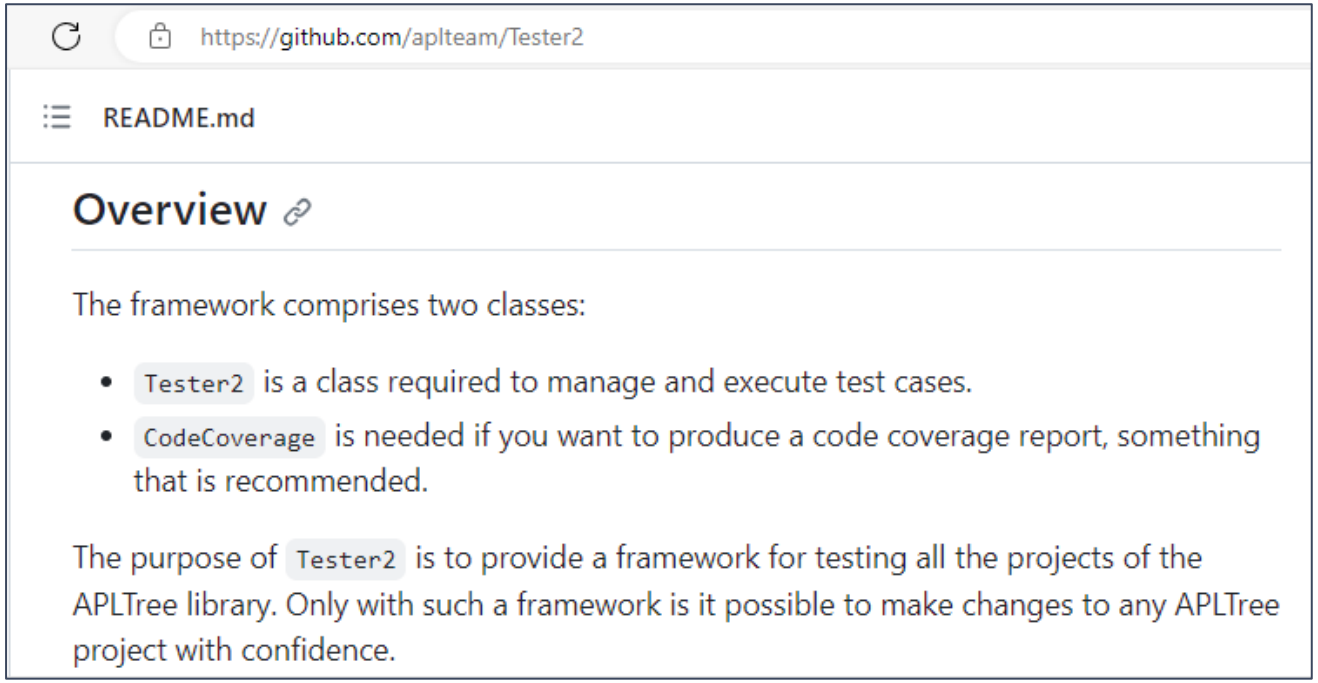
Recommendations

- Design application to allow
  - Unit Testing
  - Mocking
- Write tests before ~~coding~~ commit
- (more to come)

A couple of Tatin Packages

- aplteam-Tester2
  - Kai Jaeger's own test framework for testing his own tools / packages
- davin-Tester
  - A very simple test framework

Tester2


https://github.com/aplteam/Tester2
README.md
Overview
The framework comprises two classes:
Tester2 is a class required to manage and execute test cases.
CodeCoverage is needed if you want to produce a code coverage report, something that is recommended.
The purpose of Tester2 is to provide a framework for testing all the projects of the APLTree library. Only with such a framework is it possible to make changes to any APLTree project with confidence.

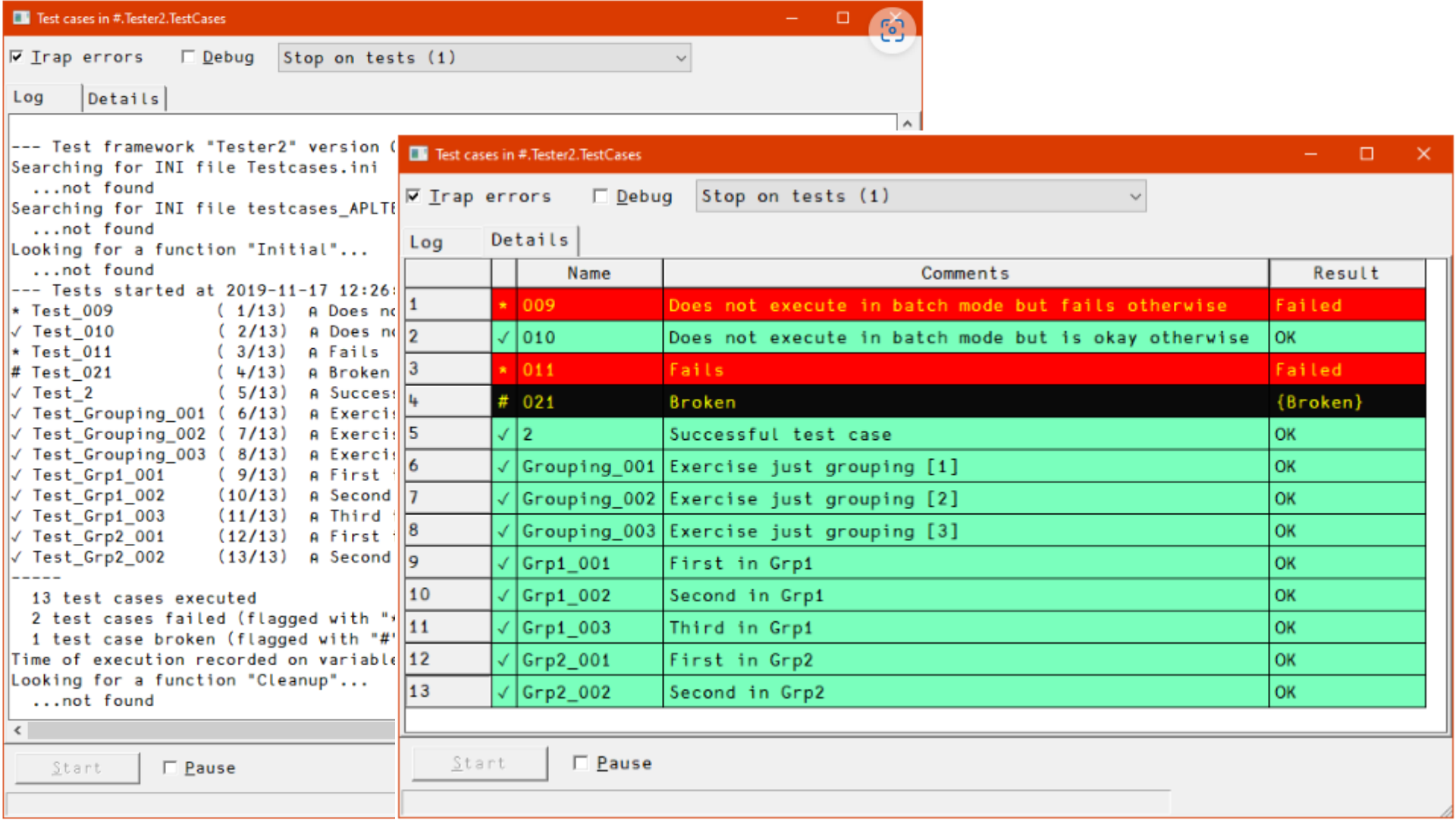
Test cases in #.Tester2.TestCases

Trap errors Debug Stop on tests (1)

Log | Details

```
--- Test framework "Tester2" version 0.4.0 from 2019-11-16 ----
Searching for INI file Testcases.ini
  ...not found
Searching for INI file testcases_APLTEAM2.ini
  ...not found
Looking for a function "Initial"...
  ...not found
--- Tests started at 2019-11-17 12:26:11 on #.Tester2.TestCases.TestCasesSimu ---
* Test_009           ( 1/13)  ⍝ Does not execute in batch mode but fails otherwise
✓ Test_010           ( 2/13)  ⍝ Does not execute in batch mode but is okay otherwise
* Test_011           ( 3/13)  ⍝ Fails
# Test_021           ( 4/13)  ⍝ Broken
✓ Test_2             ( 5/13)  ⍝ Successful test case
✓ Test_Grouping_001  ( 6/13)  ⍝ Exercise just grouping [1]
✓ Test_Grouping_002  ( 7/13)  ⍝ Exercise just grouping [2]
✓ Test_Grouping_003  ( 8/13)  ⍝ Exercise just grouping [3]
✓ Test_Grp1_001      ( 9/13)  ⍝ First in Grp1
✓ Test_Grp1_002      (10/13)  ⍝ Second in Grp1
✓ Test_Grp1_003      (11/13)  ⍝ Third in Grp1
✓ Test_Grp2_001      (12/13)  ⍝ First in Grp2
✓ Test_Grp2_002      (13/13)  ⍝ Second in Grp2
-----
  13 test cases executed
  2 test cases failed (flagged with "*")
  1 test case broken (flagged with "#")
Time of execution recorded on variable #.Tester2.TestCases.TestCasesSimu.TestCasesExec
Looking for a function "Cleanup"...
  ...not found
```

Start Pause


Test cases in #.Tester2.TestCases
Trap errors
Debug
Stop on tests (1)
Log
Details
--- Test framework "Tester2" version (
Searching for INI file Testcases.ini
...not found
Searching for INI file testcases_APLTE
...not found
Looking for a function "Initial"...
...not found
--- Tests started at 2019-11-17 12:26:
* Test_009 (1/13) A Does nc
✓ Test_010 (2/13) A Does nc
* Test_011 (3/13) A Fails
# Test_021 (4/13) A Broken
✓ Test_2 (5/13) A Succes
✓ Test_Grouping_001 (6/13) A Exerci
✓ Test_Grouping_002 (7/13) A Exerci
✓ Test_Grouping_003 (8/13) A Exerci
✓ Test_Grp1_001 (9/13) A First
✓ Test_Grp1_002 (10/13) A Second
✓ Test_Grp1_003 (11/13) A Third
✓ Test_Grp2_001 (12/13) A First
✓ Test_Grp2_002 (13/13) A Second
-----
13 test cases executed
2 test cases failed (flagged with "
1 test case broken (flagged with "#"
Time of execution recorded on variable
Looking for a function "Cleanup"...
...not found
Start
Pause
Test cases in #.Tester2.TestCases
Trap errors
Debug
Stop on tests (1)
Log
Details
Name
Comments
Result
1 * 009 Does not execute in batch mode but fails otherwise Failed
2 ✓ 010 Does not execute in batch mode but is okay otherwise OK
3 * 011 Fails Failed
4 # 021 Broken {Broken}
5 ✓ 2 Successful test case OK
6 ✓ Grouping_001 Exercise just grouping [1] OK
7 ✓ Grouping_002 Exercise just grouping [2] OK
8 ✓ Grouping_003 Exercise just grouping [3] OK
9 ✓ Grp1_001 First in Grp1 OK
10 ✓ Grp1_002 Second in Grp1 OK
11 ✓ Grp1_003 Third in Grp1 OK
12 ✓ Grp2_001 First in Grp2 OK
13 ✓ Grp2_002 Second in Grp2 OK
Start
Pause

davin-Tester

Tatin Registry

List of packages

test

Package name	Description	Major Versions	Project URL	OS	UC	Tags
aplteam-CodeCoverage	Monitors which parts of an application got actually executed	1	github.com	Lin, Mac, Win		code-coverage,test-framework,unit-tests
aplteam-Latest	Lists APL objects by change date/time	1	github.com	Lin, Mac, Win	Yes	list-apl-objects
aplteam-Tester2	Dyalog APL test framework	1	github.com	Lin, Mac, Win		test,test-framework
davin-Tester	Simplified function-level testing of programs	1	github.com	Lin, Mac, Win		function,testing,tester,validation

Tatin Registry

Details of <davin-Tester-1.0.1>


```
{
  api: "",
  assets: "",
  date: 20220913.024216,
  description: "Simplified function-level testing of programs",
  documentation: "https://github.com/DavinChurch/Tester/blob/main/README.md",
  files: "",
  group: "davin"
  io: 1,
  license: "MIT",
  lx: "",
  maintainer: "Davin Church <davinchurch@gmail.com>",
  minimumAplVersion: "18.0",
  ml: 1,
  name: "Tester",
  os_lin: 1,
  os_mac: 1,
  os_win: 1,
  project_url: "https://github.com/DavinChurch/Tester/blob/main/README.md",
  source: "Source/Tester",
  tags: "function,testing,tester,validation",
  version: "1.0.1",
}
```

Tester

This is a small set of utility programs to assist with function-level testing of applications using a very simple syntax and programming interface. Copy any or all of these routines into a namespace containing test case functions (or cross-referenced with them). These completely stand-alone testing-management programs (implemented as programmed operators) are then called from the application's test case function(s) to perform the call-and-return test(s) as written. Executing the test case function(s) will then perform all the coded testing and result validations.

If multiple functions are to be used to perform testing, the included `Test` cover function may be used to call them all in sequence. It is invoked with a list of function names (in almost any reasonable structure and format) as a right argument, the matching function names in the namespace will be executed. These names may include an `*` wild-card character, so `Test '*'` will execute all the functions in the workspace. An optional left argument may be specified to temporarily override the global `StopOnError` setting *(see below)*. `Test` will return a completion message unless errors are being counted, in which case it will return that count.

This is a member of the APLTree project and is also available via the Tatin package manager.

Testing engine

The testing engine consists of three independent, stand-alone APL operators. These may be used individually for simple argument/result testing anywhere. The left operand of each operator is the function to be tested. The right operand is the expected result. The derived function uses the provided left and right arguments and passes them directly to the function being tested. The three routines are:

Testing engine

The testing engine consists of three independent, stand-alone APL operators. These may be used individually for simple argument/result testing anywhere. The left operand of each operator is the function to be tested. The right operand is the expected result. The derived function uses the provided left and right arguments and passes them directly to the function being tested. The three routines are:

Tester	Used to...
`Pass`	Make sure the tested function returns the expected result, which is provided as the right operand value *(if a value is specified)*. Alternatively a boolean function may be specified as the right operand which will be called monadically with the result to verify that the result is correct.
`Pass_`	Make sure the tested function does NOT return an explicit result in this case. The right operand must be a boolean function to determine if the tested function produced proper side-effects, or `{1}` or `(1≡)` is sufficient if no explicit verification is to be performed.
`Fail`	Make sure the tested function exits with a `⎕SIGNAL` as validated by the right operand. The right operand may be text to match `⎕DM`, a numeric (array) for `⎕EN` to be a member of, or a boolean function (provided up to both of these values) to validate that the failure was as expected.

Error handling during testing

These routines all respect the setting of an optional namespace-global variable named `StopOnError`, which may be set to any of the following values:

`StopOnError`	Function
`0`	Do not stop, just report invalid test results.
`1`	Stop in the testing function on the line that did not validate. ***[Default]***
`2`	Stop in the tested function at the original error without any error trapping.
`¯1`	Do not stop, and increment global variable "Errors" if it exists.

This error handling is performed as described if an APL error occurs during execution of the test or if validation fails.

Stopping during testing

These routines also respect the setting of an optional namespace-global variable named `StopOnTest` which may be used to place a `⎕STOP` breakpoint in the code being tested. It should consist of a simple character vector (or a nested vector of such vectors to specify several stop points) that contains the name of the testing function (e.g. `TestFoo`) that is calling one of the above routines (not the name of the function actually being tested), followed by the desired line number in square brackets.

For instance, if testing function `TestFoo` runs 3 different tests on function `Foo` from its lines 1, 2, and 3, then you may tell the testing to pause for the test on line 2 by specifying `StopOnTest←'TestFoo[2]'`. The stop actually occurs on `Foo[1]` but only when it is being called from `TestFoo[2]`.

If you wish to specify a particular line of the tested code on which to stop (instead of `[1]`), extend the `StopOnTest` breakpoint notation to include an `@` followed by the function name and line number where the stop is to be placed. For instance, `StopOnTest←'TestFoo[2]@Foo[17]'` will cause the stop to occur on line `[17]` of `Foo` when it is called from line `[2]` of `TestFoo`. This method can also be used to stop on any other subroutine instead by specifying its name after the `@`. Any tested function not in the current namespace should be listed with an appropriate full or relative dotted name.

Remember that any D-fn must have multiple lines in order for it to accept a `⎕STOP` setting.

Writing application testing functions

Create one or more functions with any desired names (e.g. `TestFoo`) that uses these operators for each function call to be tested. For instance, if the `Plus` function is to be tested with:

```
      3 Plus 4
7
```

Include in your testing function (e.g. `TestFoo`) the simple line:

```
      3 (Plus Pass 7) 4
```

This means that `3 Plus 4` will pass the test if it returns `7` for a result.

Testing function notes

- These arbitrary testing routines may include any other code as needed to prepare the tests to be performed (and clean up afterwards), initialize testing arguments, loop through multiple tests, call subroutines, or perform any other desired actions that APL allows.
- A niladic function may be tested by enclosing it in a D-fn and passing a dummy right argument.
- Since these routines are actually operators rather than functions, remember to use parentheses around the operator and its operands or use another mechanism to separate the operands from the tested function's arguments.
- Also remember that when invoking operators, the right operand has short scope and probably needs to be enclosed in parentheses itself whenever an expression is being used as the right operand rather than a simple value.
- The `⊢` function may be used with Pass to perform a simple value assertion test, such as in `(⊢Pass 7) 3+4`, or a named function may be assigned to perform a logical assertion check with `Assert←⊢Pass 1`.

Our Application

https://github.com/Dyalog-Training/DTest/coolStat/src/coolStat.apln

```
:Namespace coolStat
⍝ A next generation statistics package
⍝ ©2023 Humble Author
⍝ in memoriam Ken Iverson, Kurt Gödel, Abu Dscha'far Muhammad ibn Musa al-Chwārizmī (أبو جعفر محمد بن موسى الخوارزمي)
⍝ with gratitude to Adám Brudzewsky and in respect of Steve Mansour
⍝ dedicated to my lovely wife


      Avg←{
          ⍵⌸=⍤⊂⍵
          0=≢⍵:0
      }

      Median←{
          (2÷⍨1⊥⊢⌷⍨⍤⊂⍋⌷⍨⍤⊂⌈2÷⍨0 1+≢)⍵
      }

      Count←{⍝ counts the number of elements in a vector
          ⍴⍵
      }

:EndNamespace
```

Exercise 1

- Write a test for one or more coolStat functions using davin-Tester

```
]tatin.loadpackages Tester

… or …

tester←'https://github.com/DavinChurch/Tester/blob/main/Source/Tester/'
{⎕SE.UCMD 'get ',tester,' ',⍵}¨'Fail.aplo' 'Pass.aplo' 'Pass_.aplo' 'Test.aplf'
```

- … Or use code scraped from https://xpqz.github.io/learnapl/testing.html (or slide #18)

DYALOG

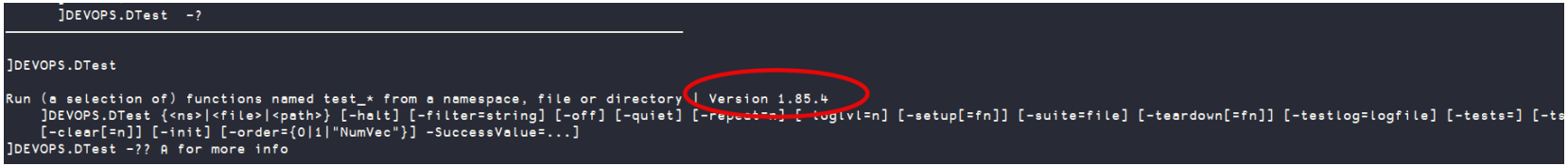
Elsinore 2023

]DTest

Michael Baas

Are you ready?

- Start Dyalog
- Same version?



- :If not ◊ :Andif v18 ◊ :Then

```
]set cmddir ",[USERPROFILE]\Documents\My UCMDs" -p
```

Scope of the workshop

- Unit testing with DTest

  ...verify the functionality
  of a specific section of code...

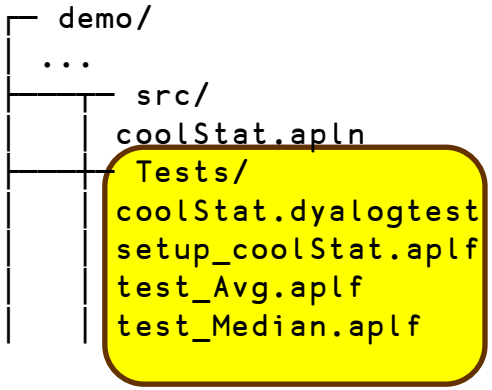
  (for APLers: "a function")

- there's more...
- Leave inspired! 😉

Organisation of files & tests

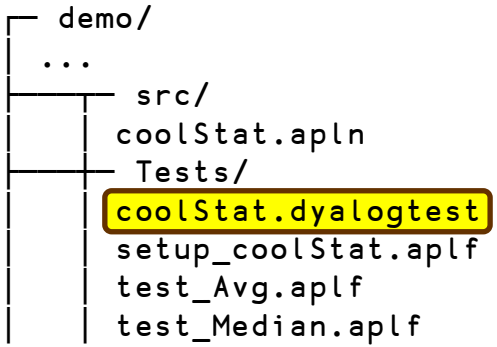
```
┌─ demo/
│  ...
├────┬─ src/
│    │  coolStat.apln
├────┼─ Tests/
│    │  coolStat.dyalogtest
│    │  setup_coolStat.aplf
│    │  test_Avg.aplf
│    │  test_Median.aplf
```

Organisation of files & tests



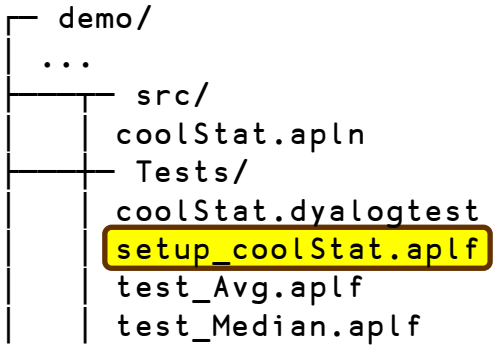
- tests live in a dedicated folder

Organisation of files & tests



- tests live in a dedicated folder
- optional .dyalogtest files define a "test suite" and are advantegous when you have multiple test suites ("basic " and "overnight") etc. or additional parameters (CodeCoverage or SuccessValue)

Organisation of files & tests



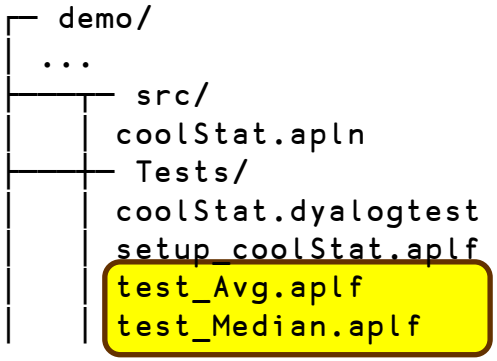
- tests live in a dedicated folder
- optional .dyalogtest files define a "test suite" and are advantegous when you have multiple test suites ("basic " and "overnight") etc. or additional parameters (CodeCoverage or SuccessValue)
- files with prefix setup_ define setups that set the stage

Organisation of files & tests

```
demo/
...
    src/
        coolStat.apln
    Tests/
        coolStat.dyalogtest
        setup_coolStat.aplf
        test_Avg.aplf
        test_Median.aplf
```

- tests live in a dedicated folder
- optional .dyalogtest files define a "test suite" and are advantegous when you have multiple test suites ("basic " and "overnight") etc. or additional parameters (CodeCoverage or SuccessValue)
- files with prefix setup_ define setups that set the stage
- the files with prefix test_ do the real work...

Organisation of files & tests



- tests live in a dedicated folder
- optional .dyalogtest files define a "test suite" and are advantegous when you have multiple test suites ("basic " and "overnight") etc. or additional parameters (CodeCoverage or SuccessValue)
- files with prefix setup_ define setups that set the stage
- the files with prefix test_ do the real work...
- and you can also have teardown_ fn that remove ~~the mess that the test created~~ any leftovers

Organisation of files & tests

```
demo/
...
    src/
        coolStat.apln
    Tests/
        coolStat.dyalogtest
        setup_coolStat.aplf
        test_Avg.aplf
        test_Median.aplf
```

- tests live in a dedicated folder
- optional .dyalogtest files define a "test suite" and are advantegous when you have multiple test suites ("basic " and "overnight") etc. or additional parameters (CodeCoverage or SuccessValue)
- files with prefix setup_ define setups that set the stage
- the files with prefix test_ do the real work...
- and you can also have teardown_ fn that remove ~~the mess that the test created~~ any leftovers

Writing tests

dfn/ test_foo1.aplf

```
test_foo1←{

    x←argL MyFn argR
```

Writing tests

dfn/ test_foo1.aplf

```
test_foo1←{

    x←argL MyFn argR

    xpct Assert x: ⍺ bla
```

{res}←a Assert b

a≡b: returns 0
a≢b: returns 1, **logs failed Assertion**

comment can also be in separate line
or
`var ⊢ a Assert b`
"var" has explanation of failure

Writing tests

dfn/ test_foo1.aplf

```
test_foo1←{

    x←argL MyFn argR

    xpct Assert x: ⍺ bla

    ''
}
```

{res}←a Assert b

a≡b: returns 0
a≢b: returns 1, **logs failed Assertion**

comment can also be in separate line
or
`var ⊢ a Assert b`
"var" has explanation of failure

Writing tests

{res}←a Check b

a≡b: returns 0
a≢b: returns 1

a←a Because b
returns a and appends b to global r

tradfn / test_foo2.dyalog

```
∇ r←test_foo2 sink

x←argL MyFn argR
r←''

:if xpct Check x
  →0 Because'test failed'
:endif
```

Writing tests

{res}←a Check b

a≡b: returns 0
a≢b: returns 1

a←a Because b
returns a and appends b to global r

tradfn / test_foo2.dyalog

```
∇ r←test_foo2 sink

x←argL MyFn argR
r←''

:if xpct Check x
  →0 Because'test failed'
:endif

2 Assert 1+1  ⍝ doc doc
```

Writing tests

{res}←a Check b

a≡b: returns 0
a≢b: returns 1

a←a Because b
returns a and appends b to global r

tradfn / test_foo2.dyalog

```
∇ r←test_foo2 sink

x←argL MyFn argR
r←''

:if xpct Check x
  →0 Because'test failed'
:endif

2 Assert 1+1  ⍝ doc doc
∇
```

]DTest

-]demo

Writing tests

.dyalogtest:

```
DyalogTest: 1.84
[SuccessValue: …]
[Setup: …]
Test: test_1
Test: test_foo
...
[Teardown: …]
```

Test function

```
∇ r←mytest sink
  ⍝ test stuff
  x←testSubj arg
  expct Assert x  ⍝ doc
∇
      OR
{
   y←testSubj arg
   [MsgVar]⊢expct Assert y: ⍝
   ...
}
```

Test DSL

`[docvar]⊢x Assert y OR x Check y`
Returns 1 if the assertion that x=y is wrong, 0 otherwise.
If –halt modifier is set, halts execution if check fails.
Additional comments on line or immediately before or after. If comments are computed, use `docvar⊢x Assert y`

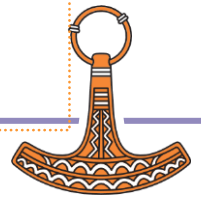
`x IsNotElement y`
test `~x∊y` and halts execution if it isn't.

`x Because y`
concatenates y to global "r" and returns x.
=> "Syntax sugar" to enable statements like:
`:if 1 Check 2 ⋄ →0 Because'1≠2!' ⋄ :endif`

`n←[id] ##.RandomVal x [y]`
generates y (default=1) random values identified by „id“ (like `[y]?x`).

`('Type' 'I|W|E')Log txt`
Adds txt to specified log (Info / Warning / Error)

Test functions need to return an empty string to indicate success. If you want to use 0 or other values, have a look at the "`SuccessValue`" modifier or add it to the .dyalogtest suite.



Running tests

]DTest {.dyalogtest | .aplf | .dyalog | path} -modifiers

Modifiers:

`-halt`: halts execution when Check or Assert fails (so that you can examine the ws)

`-trace:` trace into setup(s) and tests()

`-verbose:` show text logged with Log. (test fns should access `##.verbose` if they want to support this for ⎕←..output!)

`-quiet[=0|1]`: only shows error messages (1) or all messages (0)

`-filter=`*`aaa`*: select tests to execute (supports `*` and `?`)

`-loglvl=`*`n`*: controls the log files DTest creates. Value is a sum of the values.

1={base.log} - Errors
2={base}.warn.log - Warnings
4={base}.info.log - Informations
8={base}.session.log - Session log
16={base}.session.log - Session log ONLY for failing tests
32={base}.log.json - machine-readable results ("rc"=20: Success, 21=Failure)

`-off[=0|1]`: do (1) or do not (0) exit APL after running tests (also writes logfiles if required)

`-order[=0|1|"numvec"]`: order of tests. (0=random, 1=alphabetical, numvec specifies alternate order)

`-SuccessValue=nnn`: the value that successful tests need to return

Excercise

- Implement a test for the coolStat.Count function!
- Bonus points if you find a way to improve the implementation.
  (Is there a way to improve this (is that even possible?))

Test automation

-]demo

Automating tests

- Classic or Unicode?
- Unicode
  - LX="⎕SE.DTest ...."
  - LOAD=".../Tests" with Run.[aplf|dyalog]
- Classic
  - needs a .dws to start things
  - keep it small: ⎕LX←'⎕FIX"file:...Run.aplf"'
- loglvl=32 to get a .log.json

Code Coverage

- Careful: 100% Coverage does not mean 100% Correctness!
- 100% Coverage means that all code was executed, all possible branches were excuted.
- So IF your test cases were designed to be be wide and general (and cover ALL requirements), chances are that your code is good ;)
-]demo

DYALOG

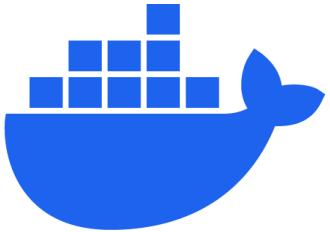
Elsinore 2023

Part 3: Testing Dyalog with Docker and GitHub Actions

Stefan Krüger

git

GitHub



Docker

Aims

- Learn how to run your unit tests from the shell.
- Learn how to use a Docker container to run your application's unit tests
- Learn how to deploy your Docker container as a GitHub Action to run your tests automatically on each commit

Aims

- Learn how to run your unit tests from the shell.
- Learn how to use a Docker container to run your application's unit tests
- Learn how to deploy your Docker container as a GitHub Action to run your tests automatically on each commit

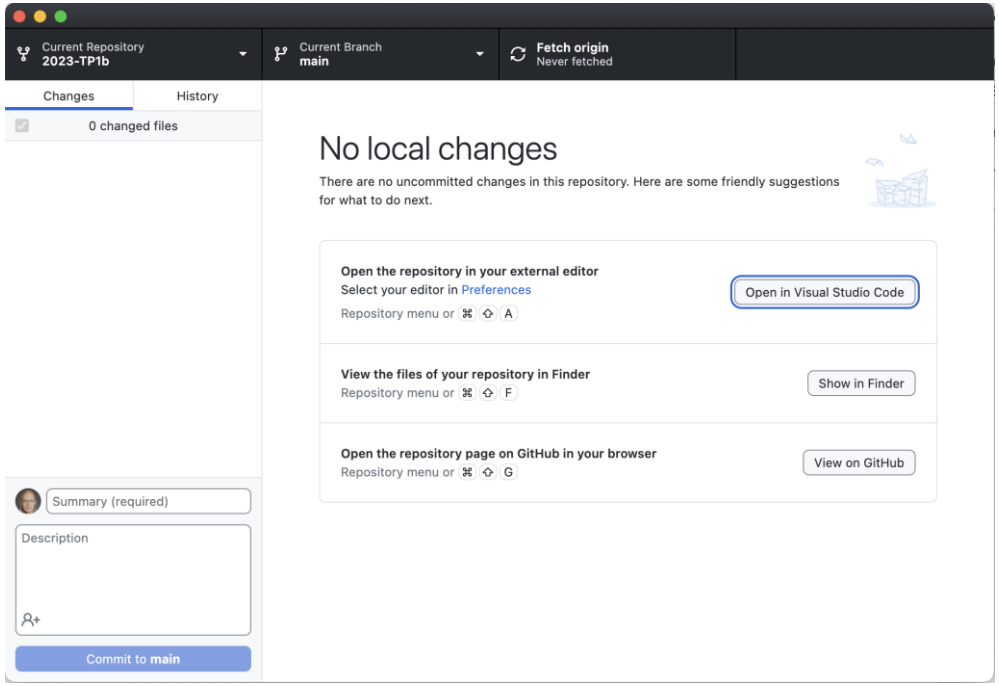
Aims

- Learn how to run your unit tests from the shell.
- Learn how to use a Docker container to run your application's unit tests
- Learn how to deploy your Docker container as a GitHub Action to run your tests automatically on each commit

Pre-requisites

- Docker installed
- `git` installed -- or GitHub Desktop
- A GitHub account
- Dyalog v18.2 + current(ish) `DTest`

GitHub Desktop



DEMO

Task: Fork 'n Clone

- To obtain a working copy, and not having to type along, fork and clone this repository

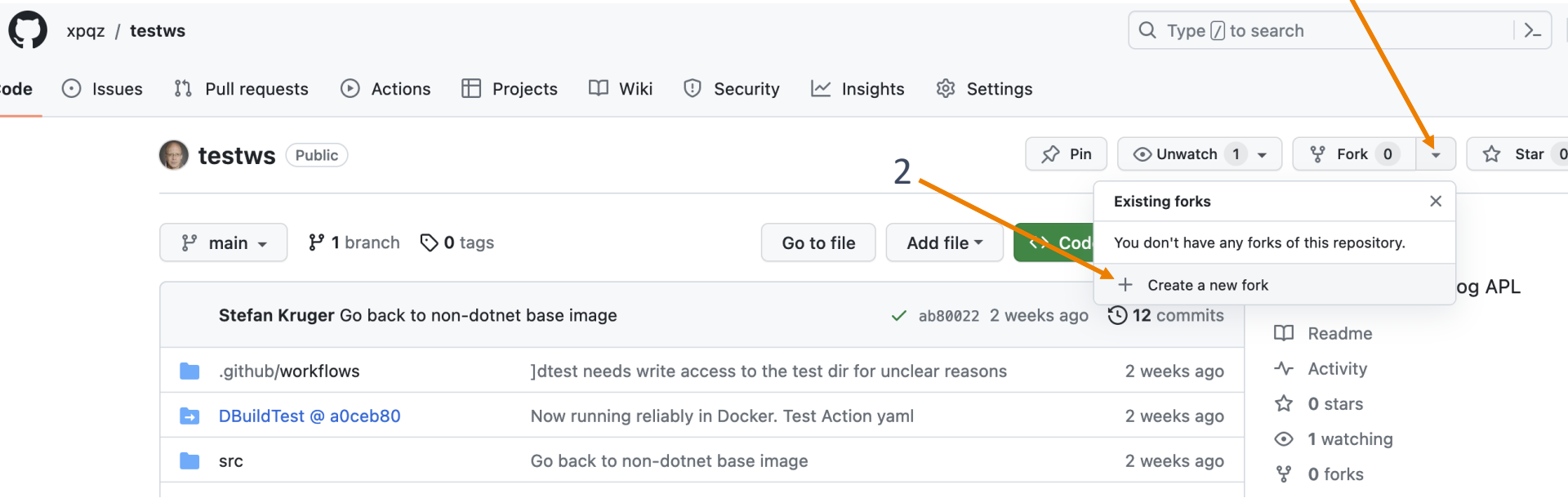
```
https://github.com/dyalog-training/2023-TP1b
```

Task: Fork 'n Clone

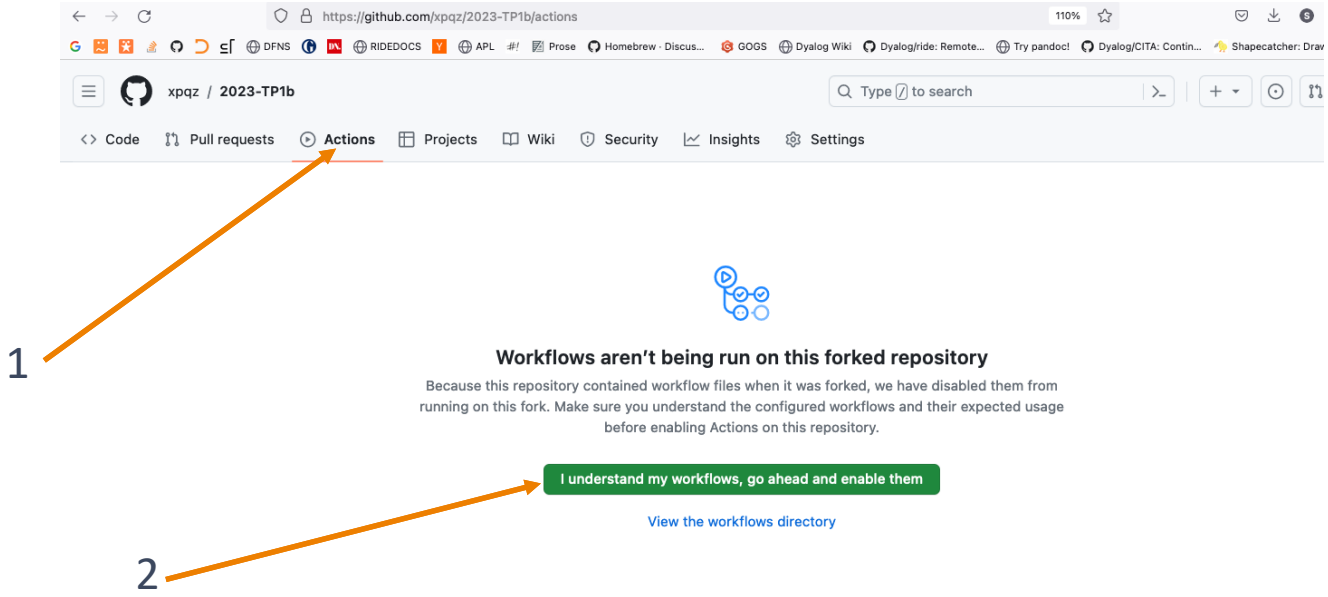
- To obtain a working copy, and not having to type along, fork and clone this repository

`https://is.gd/dytest`

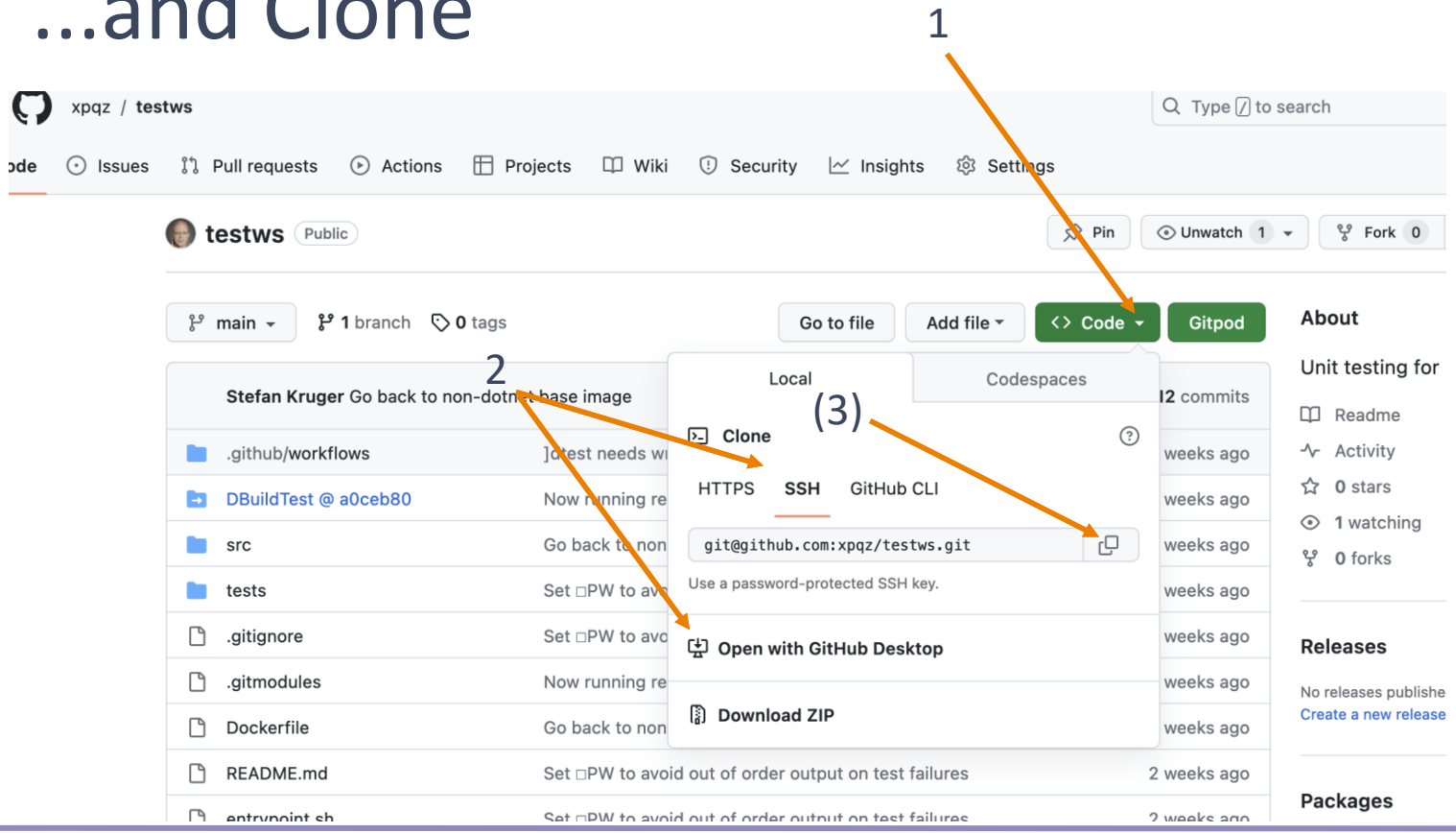
Fork...


1
2
xpqz / testws
Type / to search
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
testws
Public
Pin
Unwatch 1
Fork 0
Star 0
Existing forks
You don't have any forks of this repository.
Create a new fork
main
1 branch
0 tags
Go to file
Add file
Stefan Kruger Go back to non-dotnet base image
ab80022 2 weeks ago
12 commits
.github/workflows
]dtest needs write access to the test dir for unclear reasons
2 weeks ago
DBuildTest @ a0ceb80
Now running reliably in Docker. Test Action yaml
2 weeks ago
src
Go back to non-dotnet base image
2 weeks ago
Readme
Activity
0 stars
1 watching
0 forks

Enable workflows...



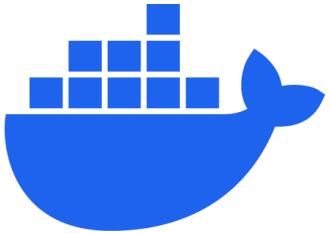
...and Clone


1
2
(3)
xpqz / testws
Type / to search
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
testws
Public
Pin
Unwatch 1
Fork 0
main
1 branch
0 tags
Go to file
Add file
Code
Gitpod
Local
Codespaces
Clone
HTTPS
SSH
GitHub CLI
git@github.com:xpqz/testws.git
Use a password-protected SSH key.
Open with GitHub Desktop
Download ZIP
Stefan Kruger Go back to non-dotnet base image
12 commits
.github/workflows
DBuildTest @ a0ceb80
src
tests
.gitignore
.gitmodules
Dockerfile
README.md
Set □PW to avoid out of order output on test failures
2 weeks ago
About
Unit testing for
Readme
Activity
0 stars
1 watching
0 forks
Releases
No releases publishe
Create a new release
Packages

Console jocks:

```
git clone --recursive git@github.com:{ACCT}/2023-TP1b.git
git clone --recursive https://github.com/{ACCT}/2023-TP1b.git
```

Actions



Docker

GitHub Actions: Continuous integration

- Code changes are automatically built, tested, and integrated into the existing codebase on a frequent basis
- GitHub has a light-weight built-in CI framework called "Actions".
- Combining Docker and Actions, we can test our Dyalog code automatically.

Continuous Integration?

- Code changes are automatically built, tested, and integrated into the existing codebase on a frequent basis
- GitHub has a light-weight built-in CI framework called "Actions".
- Combining Docker and Actions, we can test our Dyalog code automatically.

Continuous Integration?

- Code changes are automatically built, tested, and integrated into the existing codebase on a frequent basis
- GitHub has a light-weight built-in CI framework called "Actions".
- Combining Docker and Actions, we can test our Dyalog code automatically.

Docker?

- Containerisation: lightweight form of virtualisation.
- Containers share the host system's OS and isolate software dependencies.
- Efficient, easy to set up, and compatible across different computing environments.
- Useful for running tests in CI-environments.

Docker?

- Containerisation: lightweight form of virtualisation.
- Containers share the host system's OS and isolate software dependencies.
- Efficient, easy to set up, and compatible across different computing environments.
- Useful for running tests in CI-environments.

Docker?

- Containerisation: lightweight form of virtualisation.
- Containers share the host system's OS and isolate software dependencies.
- Efficient, easy to set up, and compatible across different computing environments.
- Useful for running tests in CI-environments.

Docker?

- Containerisation: lightweight form of virtualisation.
- Containers share the host system's OS and isolate software dependencies.
- Efficient, easy to set up, and compatible across different computing environments.
- Useful for running tests in CI-environments.

Project Layout

```
.
├── src
│   ├── mysum.aplf
│   └── run.aplf
└── tests
    ├── test_assert.aplf
    └── test_mysum.aplf
```

A DTest Function

```
test_mysum ← {
     '1+1 should equal 2'⊢2 Assert 1 #.mysum 1:
     ''
}
```

Task: Run `]dtest` manually

```
]link.create # /{path}/2023-TP1b/src
]dtest /{path}/2023-TP1b/tests
```

Running tests from the command line

- Utilise the `LOAD` parameter
- On start-up, Dyalog will `]link` the folder given
- If it finds a function called `Run`, it will run it

Task: Run tests from shell


```
% dyalog -b -s LOAD=src
Linked: # ↔ /Users/stefan/work/testws/src
All tests passed
```

Task: Make a test fail!


```
% dyalog -b -s LOAD=src
Linked: # ↔ /Users/stefan/work/testws/src
    *** Errors logged
        test_assert: ... = "Assertion failed: 1+1 should equal 2"
         left arg = "3", ⎕DR=83, rho=
        right arg = "2", ⎕DR=83, rho=
 Time spent:  0.0s
 -order="2 1"
```

The `Run` function

- `Run` locates the tests, and executes `]dtest`, and then □OFFs appropriately
- The GitHub Action expects a command to return 0 on success, and non-zero otherwise: □OFF 0 means success.

Run

Called with ⊂DIR by dyalog when encountering LOAD

```
Run dir;testdir;results

testdir ← 'tests',⍕⊃1⎕NPARTS'[/\\]$'⎕R''⊃dir
:If ~(⎕NEXISTS⊃1) testdir,'/test_*'
    ⎕←'No tests found'
    ⎕OFF 0
:EndIf
⎕PW ← 32767
results ← ⎕SE.UCMD'DTest ',testdir, ' -quiet'
:If 0=≢results
    ⎕←'All tests passed'
    ⎕OFF 0
:Else
    ⎕←results
    ⎕OFF 11
:EndIf
```

Find our tests dir

Abandon play if we can't find any tests

Call]dtest

]dtest returns empty vector in quiet mode

Return value 0 for success

FAIL!

Docker

```
[1]  FROM dyalog/dyalog

[2]  ARG DYALOG_RELEASE=18.2

[3]  USER root

[4]  RUN mkdir -p /home/dyalog/MyUCMDs

[5]  RUN chmod 777 /home/dyalog/MyUCMDs && chown dyalog:dyalog /home/dyalog/MyUCMDs

[6]  RUN mkdir /src /tests

[7]  RUN chown dyalog:dyalog /src /tests

[8]  COPY entrypoint.sh /entrypoint
[9]  RUN chmod +x /entrypoint

[10] RUN sed -i "s/{{DYALOG_RELEASE}}/${DYALOG_RELEASE}/" /entrypoint

[11] USER dyalog

[12] ENV LOAD "/src"

[13] ENTRYPOINT ["/entrypoint"]
```

```
[1]  FROM dyalog/dyalog

[2]  ARG DYALOG_RELEASE=18.2

[3]  USER root

[4]  RUN mkdir -p /home/dyalog/MyUCMDs

[5]  RUN chmod 777 /home/dyalog/MyUCMDs && chown dyalog:dyalog /home/dyalog/MyUCMDs

[6]  RUN mkdir /src /tests

[7]  RUN chown dyalog:dyalog /src /tests

[8]  COPY entrypoint.sh /entrypoint
[9]  RUN chmod +x /entrypoint

[10] RUN sed -i "s/{{DYALOG_RELEASE}}/${DYALOG_RELEASE}/" /entrypoint

[11] USER dyalog

[12] ENV LOAD "/src"

[13] ENTRYPOINT ["/entrypoint"]
```

```
[1]  FROM dyalog/dyalog

[2]  ARG DYALOG_RELEASE=18.2

[3]  USER root

[4]  RUN mkdir -p /home/dyalog/MyUCMDs

[5]  RUN chmod 777 /home/dyalog/MyUCMDs && chown dyalog:dyalog /home/dyalog/MyUCMDs

[6]  RUN mkdir /src /tests

[7]  RUN chown dyalog:dyalog /src /tests

[8]  COPY entrypoint.sh /entrypoint
[9]  RUN chmod +x /entrypoint

[10] RUN sed -i "s/{{DYALOG_RELEASE}}/${DYALOG_RELEASE}/" /entrypoint

[11] USER dyalog

[12] ENV LOAD "/src"

[13] ENTRYPOINT ["/entrypoint"]
```

```
[1]  FROM dyalog/dyalog

[2]  ARG DYALOG_RELEASE=18.2

[3]  USER root

[4]  RUN mkdir -p /home/dyalog/MyUCMDs

[5]  RUN chmod 777 /home/dyalog/MyUCMDs && chown dyalog:dyalog /home/dyalog/MyUCMDs

[6]  RUN mkdir /src /tests

[7]  RUN chown dyalog:dyalog /src /tests

[8]  COPY entrypoint.sh /entrypoint
[9]  RUN chmod +x /entrypoint

[10] RUN sed -i "s/{{DYALOG_RELEASE}}/${DYALOG_RELEASE}/" /entrypoint

[11] USER dyalog

[12] ENV LOAD "/src"

[13] ENTRYPOINT ["/entrypoint"]
```

```
[1]  FROM dyalog/dyalog

[2]  ARG DYALOG_RELEASE=18.2

[3]  USER root

[4]  RUN mkdir -p /home/dyalog/MyUCMDs

[5]  RUN chmod 777 /home/dyalog/MyUCMDs && chown dyalog:dyalog /home/dyalog/MyUCMDs

[6]  RUN mkdir /src /tests

[7]  RUN chown dyalog:dyalog /src /tests

[8]  COPY entrypoint.sh /entrypoint
[9]  RUN chmod +x /entrypoint

[10] RUN sed -i "s/{{DYALOG_RELEASE}}/${DYALOG_RELEASE}/" /entrypoint

[11] USER dyalog

[12] ENV LOAD "/src"

[13] ENTRYPOINT ["/entrypoint"]
```

```
#!/bin/bash

export DYALOG=/opt/mdyalog/{{DYALOG_RELEASE}}/64/unicode/
export LD_LIBRARY_PATH="${DYALOG}:${LD_LIBRARY_PATH}"
export WSPATH=$WSPATH:${DYALOG}/ws
export TERM=dumb
export APL_TEXTINAPLCORE=${APL_TEXTINAPLCORE-1}
export TRACE_ON_ERROR=0
export SESSION_FILE="${SESSION_FILE-$DYALOG/default.dse}"

$DYALOG/dyalog -b -s
```

Task 3: Build container locally

```
docker build -t dytest .
```

Task 4: Run it (on macOS or Linux)

```
docker run --rm \

  {FILE SYSTEM MOUNTS}

  dytest
```

Remove container on exit

The messy bits in the middle

Container name

Task 4: Run it (on macOS or Linux)

```
docker run --rm \
  -v
"$(pwd)/DBuildTest/DyalogBuild.dyalog:/home/dyalog
MyUCMDs/DyalogBuild.dyalog" \
  -v "$(pwd)/src:/src" \
  -v "$(pwd)/tests:/tests" \
  dytest
```

Task 4: Run it (on macOS or Linux)

```
docker run --rm \
  -v
"$(pwd)/DBuildTest/DyalogBuild.dyalog:/home/dyalog
MyUCMDs/DyalogBuild.dyalog" \
    -v "$(pwd)/src:/src" \
    -v "$(pwd)/tests:/tests" \
    dytest
```

Task 4: Run it (on macOS or Linux)

```
docker run --rm \
  -v
"$(pwd)/DBuildTest/DyalogBuild.dyalog:/home/dyalog
MyUCMDs/DyalogBuild.dyalog" \
  -v "$(pwd)/src:/src" \
  -v "$(pwd)/tests:/tests" \
  dytest
```

Task 4: Run it (Windows PowerShell)

```
docker run --rm `
 -v
"${PWD}/DBuildTest/DyalogBuild.dyalog:/home/dyalog
MyUCMDs/DyalogBuild.dyalog" `
  -v "${PWD}/src:/src" `
  -v "${PWD}/tests:/tests" `
  dytest
```

```
% docker run --rm \
  -v "$(pwd)/DBuildTest/ {∘∘∘} /DyalogBuild.dyalog" \
  -v "$(pwd)/src:/src" \
  -v "$(pwd)/tests:/tests" \
  dytest
Link Warning: ⎕SE.Link.Create: .NET or .NetCore not availabl
- watch defaults to 'ns'
Linked: # → /src

Rebuilding user command cache... done
All tests passed
```

GitHub Actions

```yaml
name: Run Dyalog APL Unit Tests

on:
  push:
    branches:
      - main

jobs:
  run-dyalog:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2
      with:
        submodules: 'recursive'
        fetch-depth: 0
```

Which events trigger the Action?

...pushes to the main branch

What kind of O/S should the Action runner use?

Check out our repository

Use GitHub's "checkout" action

...including submodules

Build container

```
    - name: Build custom Docker image
      run: docker build -t dytest .

    # ]dtest requires write access to /tests
    - name: Set permissions for /tests
      run: chmod 777 tests

    - name: Run unit tests
      run: |
        docker run --rm \
          -v "${{ github.workspace
}}/DBuildTest/DyalogBuild.dyalog:/home/dyalog/MyUCMDs/DyalogBuild.dyalog"
\
          -v "${{ github.workspace }}/src:/src" \
          -v "${{ github.workspace }}/tests:/tests" \
          dytest
```

Tweak permissions

Run container

Task 5: Action!

Git cheat-sheet

```
git add src/mysum.aplf

git commit -m 'Make a test fail'

git push origin main
```

Summary

- We executed our tests from the shell using LOAD
- We ran our tests in a Docker container
- We deployed our Docker container using a GitHub Action
- Now our every commit triggers a full test run.

Summary

- We executed our tests from the shell using LOAD
- We ran our tests in a Docker container
- We deployed our Docker container using a GitHub Action
- Now our every commit triggers a full test run.

Summary

- We executed our tests from the shell using LOAD
- We ran our tests in a Docker container
- We deployed our Docker container using a GitHub Action
- Now our every commit triggers a full test run.

Summary

- We executed our tests from the shell using LOAD
- We ran our tests in a Docker container
- We deployed our Docker container using a GitHub Action
- Now our every commit triggers a full test run.