



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



Session 2: DTest (Michael)

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

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



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]

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

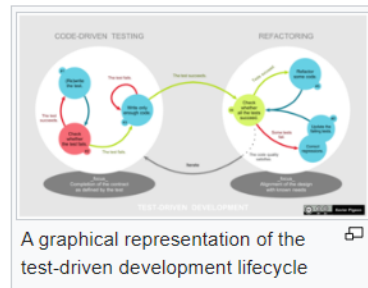
Inelegant or *hard code* is acceptable, as long as it passes the test. The code will be honed anyway in Step 5. No code should be added beyond the tested functionality.

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.



A graphical representation of the test-driven development lifecycle



≡ 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"?

$$x \leftarrow | \div y$$

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

<> 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
```

```
:Namespace Demo_tests
▽ Z←count_comments_TEST;input
input←0 1 2/'c'int a;'
input,+1 2/'c'// this is a comment'
input,+c' // comment with leading blank'
#.UT.expect←0 0 0,1 2,1
Z←#.Demo.count_comments"input
:EndNamespace
```

[https://xpqz.github.io/
learnapl/testing.html](https://xpqz.github.io/learnapl/testing.html)

```
:Namespace unittest
  ⍺IO ← 0
  run←{
    tests ← 'test_.'+⍺S'&'⍺NL ⍵3
    0=≠tests: 'no tests found'
    ↑{α,('.'/~30-≠α),ω>'[FAIL]' '[OK]'}≠↑tests (⊥`tests,`c' ⍺)
  }
: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⊢αα/⊖1↑01⊢↑ω}
  run←{ A ω -- ns containing functions to be tested
    params ← '_(≠⊢)' '[^_]+_testdata'⍬S'&'⍬NL~2.1 A https://aplcart.info/?q=%E2%8E%
    0≠params: 'no test parameter sets found'
    funs ← ⍵↑⊖1↑01⊢↑params A Corresponding functions defined
    testable ← funs/⍵funs∈ω.⍬NL~3
    result←ω∘{(α.≡ω)_test ≡ω, '_testdata'}⍵testable A Run the tests
    ↑{α, ('./'~30-≠α), '[', (≠+/ω), '/', (≠≠ω), '']}↑testable result A Format
  }
:EndNamespace
```

```
datatest.split_testdata←(' ' ('hello world') ('hello' 'world')) (',' ('hello,world') ('hell
datatest.isupper_testdata←(⊕ ('FOO') 1) (⊕ ('Foo') 0) (⊕ (, 'F') 1) A 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)



readme.md



ullu

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 ⌊)



Id & Comment

:Namespace unittest

```

GetTests←{ ⍵ is a ref to a namespace containing functions called test_*
  tests←'test_.'∘S'&'⍵.⍋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←r1 ⍵ 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 ~rVstop

```

```

  PPTestResult r tID tCmt

```

```

:EndIf

```

▽

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+ib is defined to be  $\sqrt{a^2+b^2}$ .
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 (shapeexpectedR)=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 ω)
[3] |trim←{100<#ω:(100†ω),' ...' ⌈ ω}
[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#1~1)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 ⌈ 1~1-d.value ⌈ status+0 ⌈ :EndTrap
[14] |       :If status=0 ⌈ err d ⌈ :EndIf
[15] |   :Else ⌈ 'Invalid unit test type.'⌈ SIGNAL 11 ⌈ :EndIf
[16] |   i←i+1
[17] |:EndFor
[18] |r←(⌈i),' tests ok.'
```

```
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: 100p4)
  (type: 'ok' ⌈ value: 180000)
  (type: 'ok' ⌈ value: {ω[4ω]}{?90000}~1000) A APLLONG
  (type: 'ok' ⌈ value: {ω[4ω]}{?2000}~1000) A APLINTG
  (type: 'ok' ⌈ value: {ω[4ω]}{?80}~1000) A APLSINT
  (type: 'error' ⌈ value: ~1 0 1) A Negatives
  (type: 'ok' ⌈ value: {ω[4ω]}{?2000}~100000) A Large datasets
  (type: 'error' ⌈ value: {ω[ψω]}{?2000}~100) A 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)
  (⌈io: 0 ⌈ type: 'error' ⌈ value: 1 0 0 0 0 0 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: 100p4)
  (⌈io: 0 ⌈ type: 'ok' ⌈ value: 180000)
)

Nested Array
Pos: 0/24,0
```


Link Testing



```
[0] pk+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'' A 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] A Create a monadic function
[21] _+(<foo+' r+foo x' ' x x')QNPOT folder,'/foo.dyalog'
[22] assert'foo=ns.NR 'foo'' 'ns.FX tfoo'
[23] A Create a niladic / non-explicit function
[24] _+(<nil+' nil' ' 2+2')QNPOT folder,'/nil.dyalog'
[25] assert'nil=ns.NR 'nil'' 'ns.FX tnil'
[26]
[27] A Create an array
[28] _+(<['one' 1 'two' 2])QNPOT o2file+folder,'/one2.apla'
[29] assert'(2 2p'one' 1 'two' 2)=ns.one2'
```

assert 'test'



```
[0] pk+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'' A 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] A Create a monadic function
[21] +(cfoo+' r+foo x' ' x x')QNPUT folder,'/foo.dyalog'
[22] assert'foo=ns.NR 'foo'' 'ns.FX tfoo'
[23] A Create a niladic / non-explicit function
[24] _+(cnil+' nil' ' 2+2')QNPUT folder,'/nil.dyalog'
[25] assert'nil=ns.NR 'nil'' 'ns.FX tnil'
[26]
[27] A Create an array
[28] _+(c['one' 1 'two' 2])QNPUT o2file+folder,'/one2.apla'
[29] assert'(2 2p'one' 1 'two' 2)=ns.one2'
```

assert 'test' 'recovery-expression'



Test for expected errors

```
A test failing creations  
assert'{6::1 ♦ 0=CountLinks}0'  
3 □DELETE 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



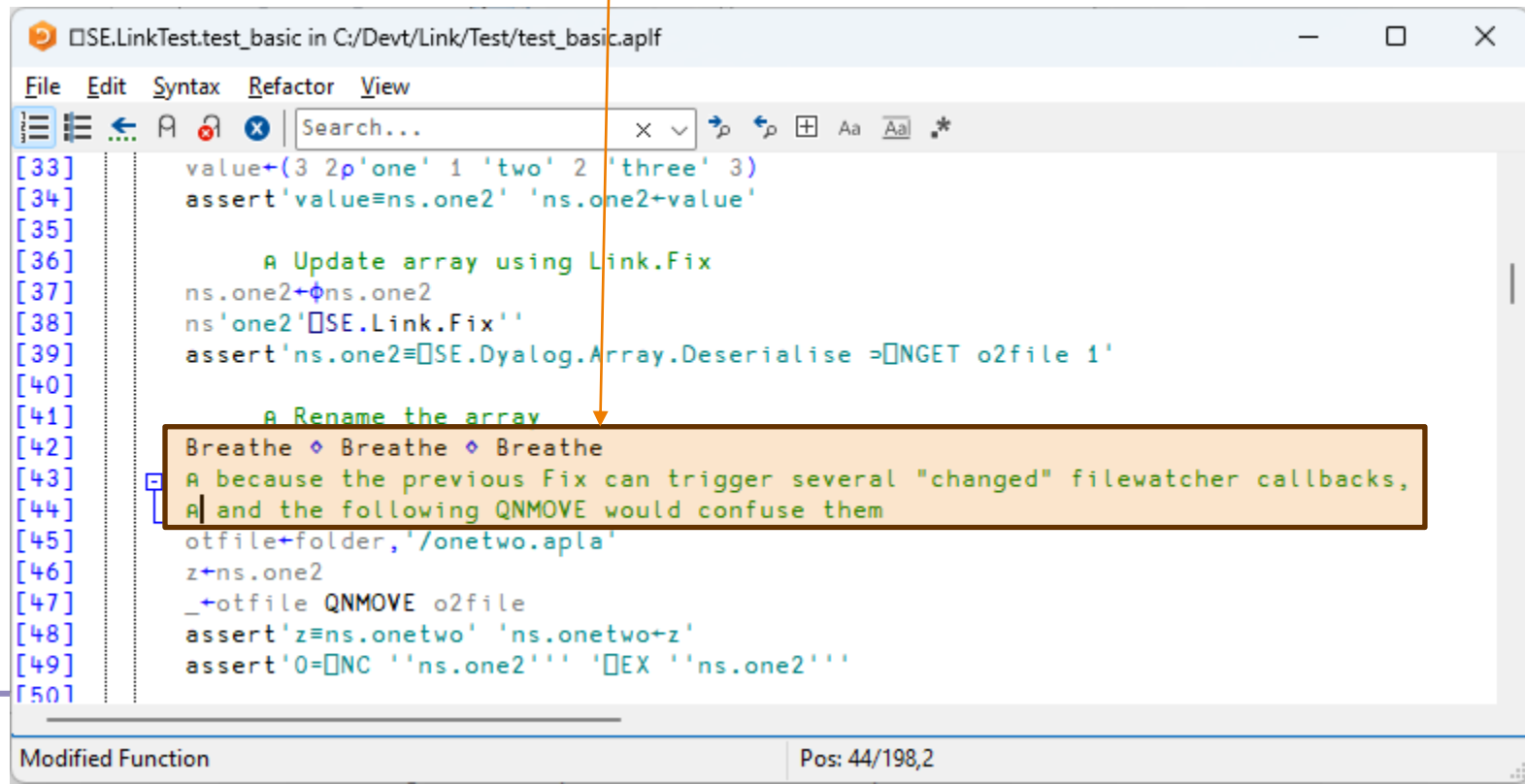
Keep trying until
Event arrives and
is processed.

Hence the `⌞`

```
OSE.LinkTest.assert in C:/Dev/Link/Test/assert.aplf
File Edit Syntax Refactor View
Search...
[0] | {txt}+{msg}assert args;clean;expr;maxwait;end;timeout;txt
[1] |   A Asynchronous assert: We don't know how quickly the FileSystemWatcher will do s
[2] |
[3] |   :If STOP_TESTS
[4] |     Log 'STOP_TESTS detected...'
[5] |     (1+>⌞LC)⌞STOP 'assert'
[6] |   :EndIf
[7] |
[8] |   (expr clean)+2⌞(⌞args),c''
[9] |   end+(3000*~USE MOCK_FSW)+3>⌞AI A allow three seconds of wait time unless mocking
[10] |   timeout+0
[11] |
[12] |   :While 0<{0::0 ⌞ ⌞w}expr
[13] |     Breathe
[14] |   :Until timeout+end<3>⌞AI
[15] |
[16] |   :If 900<0 A Monadic
[17] |     msg+'assertion failed'
[18] |   :EndIf
[19] |   :If ~timeout ⌞ txt+' ' ⌞ :Return ⌞ :EndIf
[20] |
[21] |   txt+msg,' ' ,expr,' A at ',(2>⌞XSI),'[',(⌞2>⌞LC),' '
[22] |   :If ASSERT_DORECOVER^0≠#clean A Was a recovery expression provided?
[23] |     ⌞clean
[24] |     :AndIf ~0<{0::0 ⌞ ⌞w}expr A Did it work?
[25] |       Log 'Warning: ',txt,(~0<⌞clean)/'- Recovered via ',clean
[26] |       :Return
[27] |     :EndIf
[28] |
[29] |     A No recovery, or recovery failed
[30] |   :If ASSERT_ERROR
[31] |     txt ⌞SIGNAL 11
[32] |   :Else A Just muddle on, not recommended!
[33] |     Log txt
[34] |   :EndIf
```


Mocking

This also helped a bit



```
SE.LinkTest.test_basic in C:/Dev/Link/Test/test_basic.aplf
File Edit Syntax Refactor View
Search...
[33] value+=(3 2p'one' 1 'two' 2 'three' 3)
[34] assert'value==ns.one2' 'ns.one2+value'
[35]
[36] A Update array using Link.Fix
[37] ns.one2+=ns.one2
[38] ns'one2'=SE.Link.Fix''
[39] assert'ns.one2==SE.Dyalog.Array.Deserialise =>NGET o2file 1'
[40]
[41] A Rename the array
[42] Breathe Breathe Breathe
[43] A because the previous Fix can trigger several "changed" filewatcher callbacks,
[44] A and the following QNMOVE would confuse them
[45] o2file+=folder,'/onetwo.apla'
[46] z+=ns.one2
[47] _+=o2file QNMOVE o2file
[48] assert'z==ns.onetwo' 'ns.onetwo+z'
[49] assert'0=NC 'ns.one2' 'EX 'ns.one2'
[50]
```

Modified Function Pos: 44/198,2

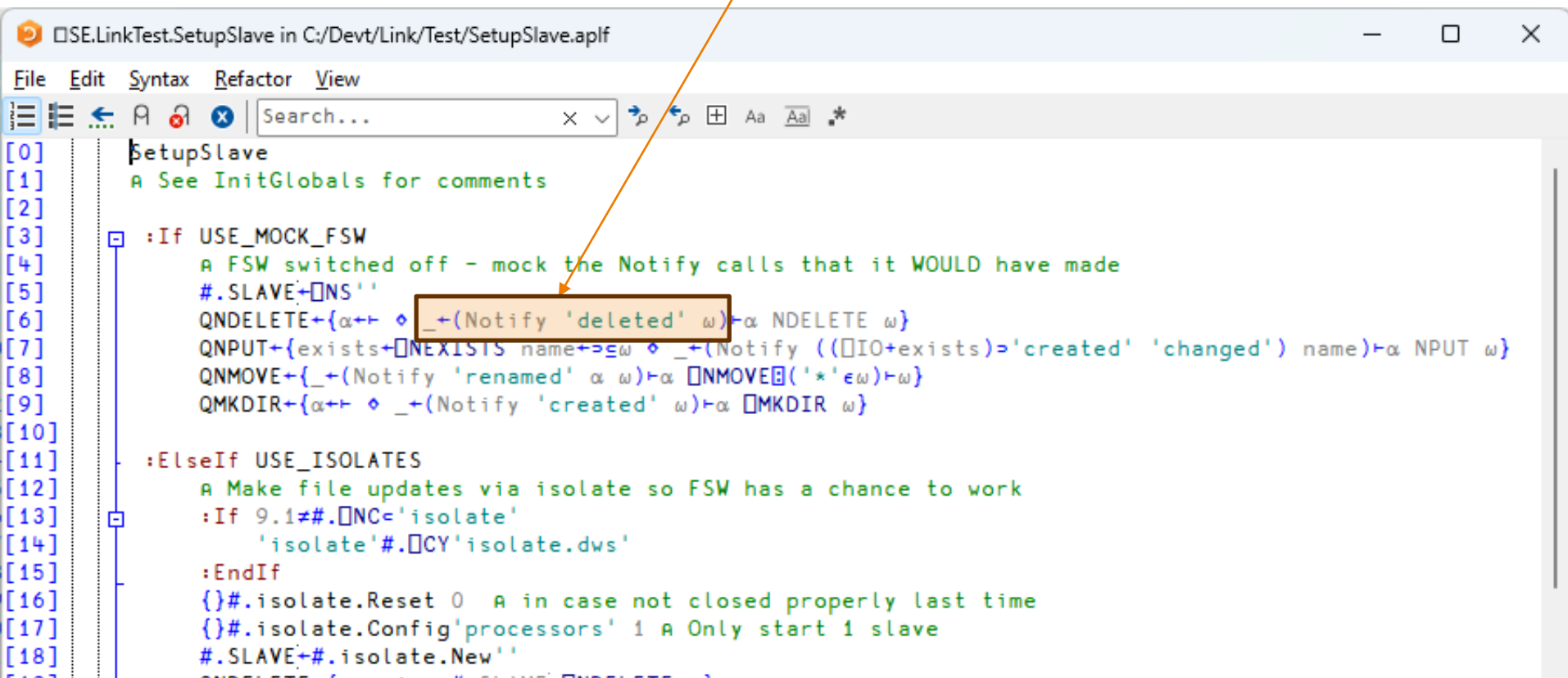
Mocking

- 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



The screenshot shows a code editor window titled 'OSE.LinkTest.SetupSlave in C:/Dev/Link/Test/SetupSlave.aplf'. The code is written in a language that uses a mix of plain text and a specific syntax for operations and comments. A callout box with an orange arrow points to the line `QDELETE+{α+⋄ _+(Notify 'deleted' ω)⋄α NDELETE ω}`, which is highlighted with a brown border. The code includes conditional blocks for `USE MOCK_FSW` and `USE_ISOLATES`, with various operations like `QDELETE`, `QINPUT`, `QMOVE`, and `QMKDIR` being defined or used. Comments are preceded by `A`.

```
[0] SetupSlave
[1] A See InitGlobals for comments
[2]
[3] :If USE MOCK_FSW
[4] A FSW switched off - mock the Notify calls that it WOULD have made
[5] #.SLAVE+⋄NS''
[6] QDELETE+{α+⋄ _+(Notify 'deleted' ω)⋄α NDELETE ω}
[7] QINPUT+{exists+⋄NEXISTS name+⋄ω ⋄ _+(Notify ((⋄IO+exists)⋄'created' 'changed') name)⋄α NPUT ω}
[8] QMOVE+{ _+(Notify 'renamed' α ω)⋄α ⋄NMOVE⋄(' '*εω)⋄ω}
[9] QMKDIR+{α+⋄ _+(Notify 'created' ω)⋄α ⋄MKDIR ω}
[10]
[11] :ElseIf USE_ISOLATES
[12] A 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 A in case not closed properly last time
[17] {}#.isolate.Config'processors' 1 A Only start 1 slave
[18] #.SLAVE+#.isolate.New''
[19] QDELETE+{α+⋄ _+(Notify 'deleted' ω)⋄α NDELETE ω}
```

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/Buttc

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

A Test /Examples/DC/InputGridSimple

data←'Morten' 'Kromberg' (⌈1+⌈/','⌈⌈⌈TS) A It's my birthday every day!

'fname' 'lname' 'bdate' SendKeys"data

Click'ClickMe'

msg←'output'WaitFor'Hi Morten Kromberg. Happy Birthday!'

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

```

478
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,(pargs)↓'Thank You!' 'Expected output did not appear'
487 f←{v/''',(1+text='')/text),'','≡'[1+xp;text]
488 :If element.TagName='input'
489     f,←'element.GetAttributeC' 'value''}
490 :Else
491     f,←'element.Text'}
492 :EndIf
493 r←(~(Ⓜ f)Retry 0)/msg
494
495

```

□ WC – No idea
how to test
automatically



Driving Dyalog IDE



Drivin

The screenshot shows a GitHub repository for `GhostRider.dyalog`. The left sidebar displays the file structure with `master` selected. The main content area shows the code for `GhostRider.dyalog`, which is a Dyalog APL class definition. The code is as follows:

```
1  :Class GhostRider
2
3  ⑆ Headless RIDE client for QA and automation.
4  ⑆ This class will connect to an APL process (or create a new one)
5  ⑆ and synchronously communicate through the RIDE protocol in order to control it.
6  ⑆ This means that when the GhostRider expects a response from the interpreter
7  ⑆ it will block the APL thread until it gets it.
8  ⑆ Dyalog v18.0 Unicode or later required.
9
10 ⑆ To create a new APL process and connect to it
11 ⑆   R←NEW GhostRider {env}
12 ⑆ - optional {env} is a string giving a list of environment variables to set up for the interpreter
13 ⑆   e.g. 'MAXWS=1G WSPATH=.'
14 ⑆   defaults to ''
15
16 ⑆ To connect to an existing process
17 ⑆   R←NEW GhostRider (port {host})
18 ⑆ - port is the positive integer port number to connect to.
19 ⑆ - optional {host} is a string giving the ip address to connect to
20 ⑆   {host} defaults to '127.0.0.1' which is the local machine
21
22
23 ⑆ 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 Create, Export, Break #598

Merged

Changes from all commits

File filter

Conversations

Filter changed files

StartupSession/Link

Utils.apln

Test

test_create.aplrf

test_export.aplrf

StartupSession/Link/Utils.apln

8

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

Test/test_create.aplrf

4

@@ -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:''gz'

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:''gz'

14

14

assert'{6:1 @ @=CountLinks}@'

15

15

[]EX name @ 3 []NDELETE folder

Temp Folders

```
OSE.LinkTest.CreateTempDir in C:/Dev/Link/Test/CreateTemp...
File Edit Syntax Refactor View
[0] dir+CreateTempDir create;i;prefix;tmp;dirs
[1] prefix+(739i0),'/linktest-' i+0
[2] :Repeat dir+prefix,i+i+1
[3] :Until ~v/[]NEXISTS dirs+dir,'"'" '-config'
[4] :If create 2 []MKDIR dirs :EndIf
[5]

Modified Function Pos: 6/7,1
```

```
OSE.LinkTest.CleanFolders in C:/Dev/Link/Test/CleanFolders.aplf
File Edit Syntax Refactor View
[0] CleanFolders;names;z
[1] A Utility to clear test folders after multiple failed/aborted tests
[2]
[3] :If 0=#names+>0 []NINFO[] (739i0),'/linktest-*'
[4] []+'Nothing to clean'
[5] +0
[6] :EndIf
[7]
[8] []+;names
[9] []+'Type Y to delete ',(v#names),' folder(s):'
[10] z+[]
[11] +(~v/'yY'εz)p0
[12] 3 []NDELETE names

Function Pos: 0/13,0
```

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 \equiv
 - ✧ 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)

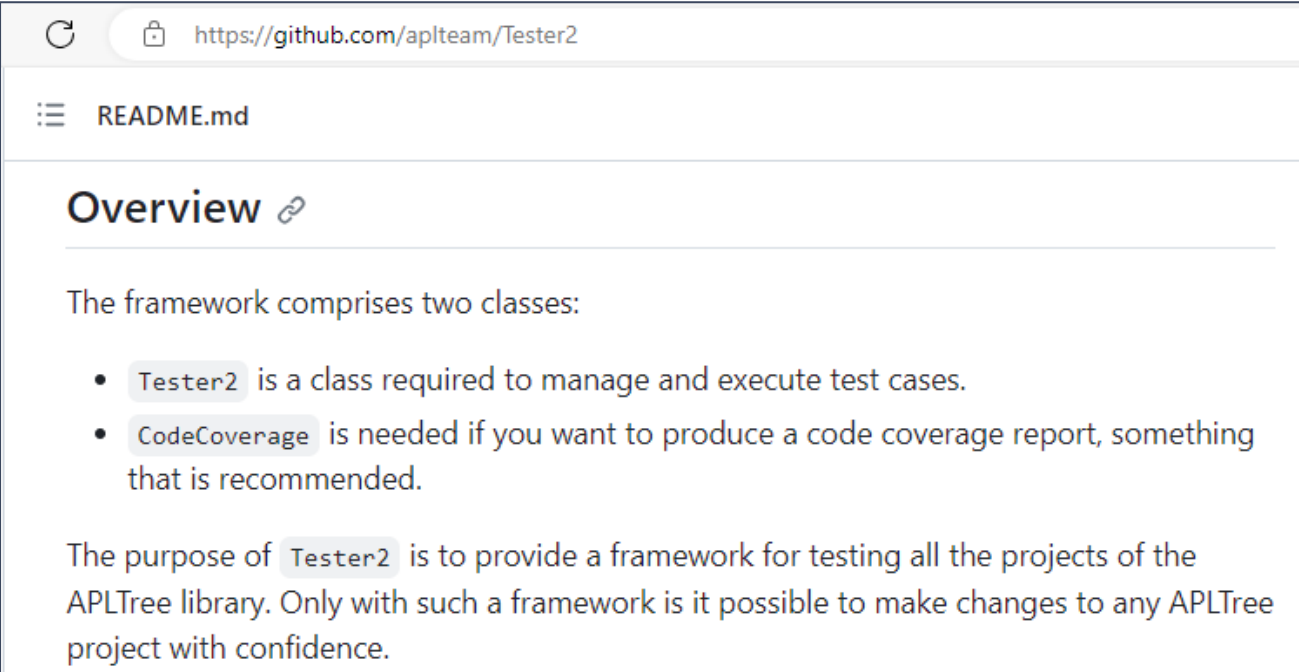


A couple of Tatin Packages

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



Tester2

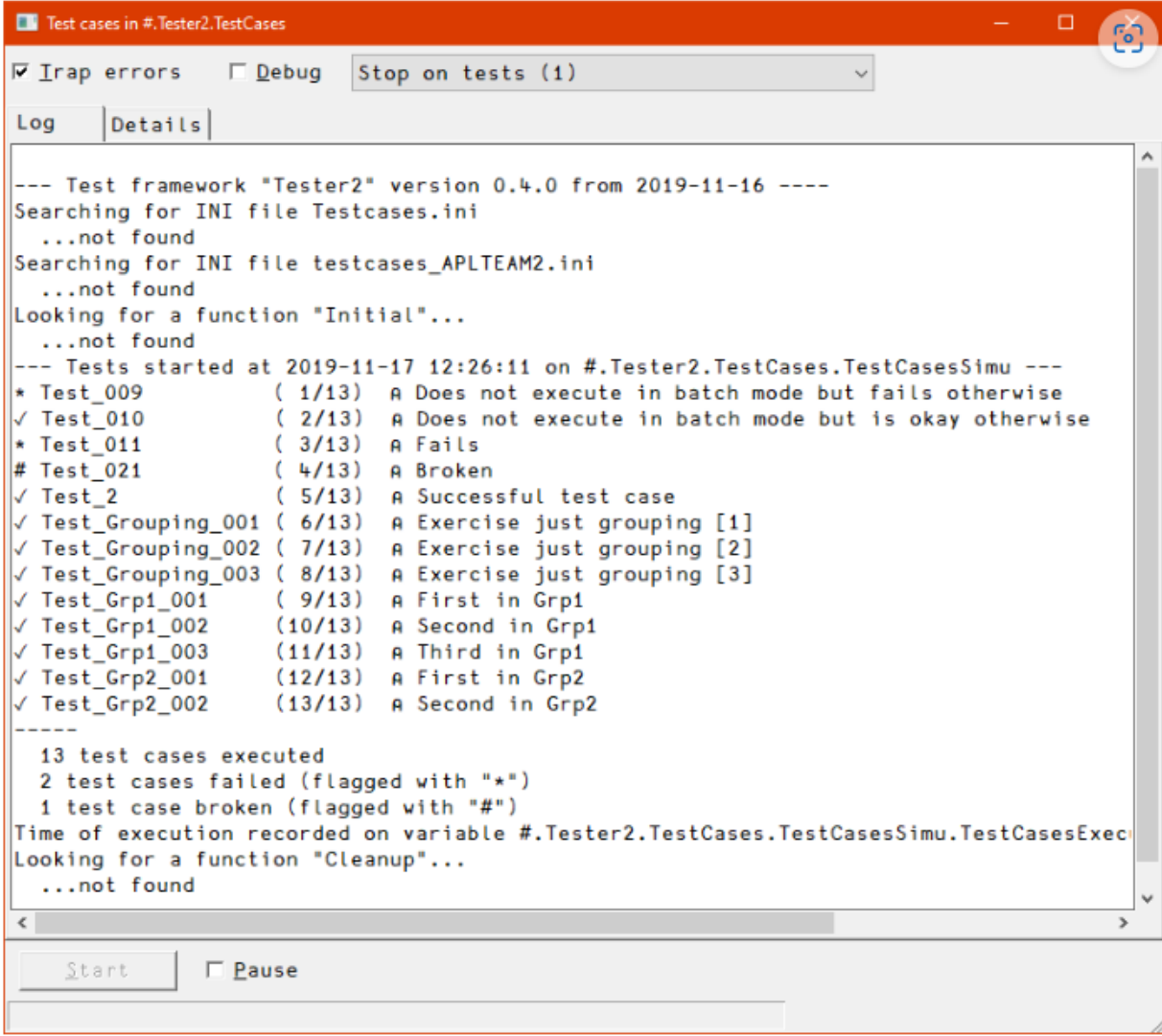


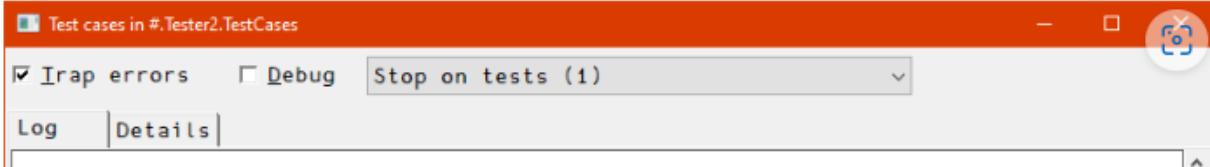
A screenshot of a web browser displaying the GitHub repository page for `aplteam/Tester2`. The address bar shows the URL `https://github.com/aplteam/Tester2`. Below the address bar, there is a navigation menu with a hamburger icon and the text `README.md`. The main content area features a section titled **Overview** with a link icon. Below the title, a paragraph states: "The framework comprises two classes:". This is followed by a bulleted list:

- `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.

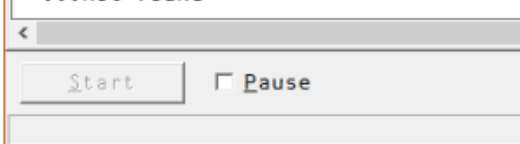
Below the list, another paragraph states: "The purpose of `Tester2` is to provide a framework for testing all the projects of the APLTree library. Only with such a framework is it possible to make changes to any APLTree project with confidence."



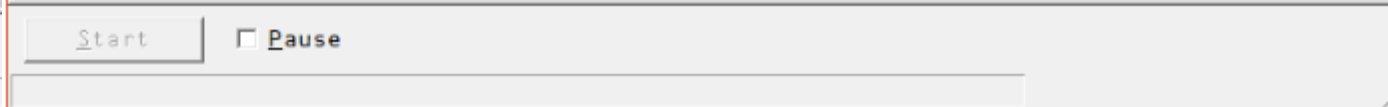




```
--- Test framework "Tester2" version 0.1.0
Searching for INI file Testcases.ini
...not found
Searching for INI file testcases_APLT
...not found
Looking for a function "Initial"...
...not found
--- Tests started at 2019-11-17 12:26:
* Test_009 ( 1/13) A Does not
✓ Test_010 ( 2/13) A Does not
* Test_011 ( 3/13) A Fails
# Test_021 ( 4/13) A Broken
✓ Test_2 ( 5/13) A Success
✓ Test_Grouping_001 ( 6/13) A Exercise
✓ Test_Grouping_002 ( 7/13) A Exercise
✓ Test_Grouping_003 ( 8/13) A Exercise
✓ 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
```



Test cases in #.Tester2.TestCases				
<div><input checked="" type="checkbox"/> Irap errors <input type="checkbox"/> Debug Stop on tests (1)</div> <div>Log Details</div>				
		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



davin-Tester





Tatin Registry

List of packages

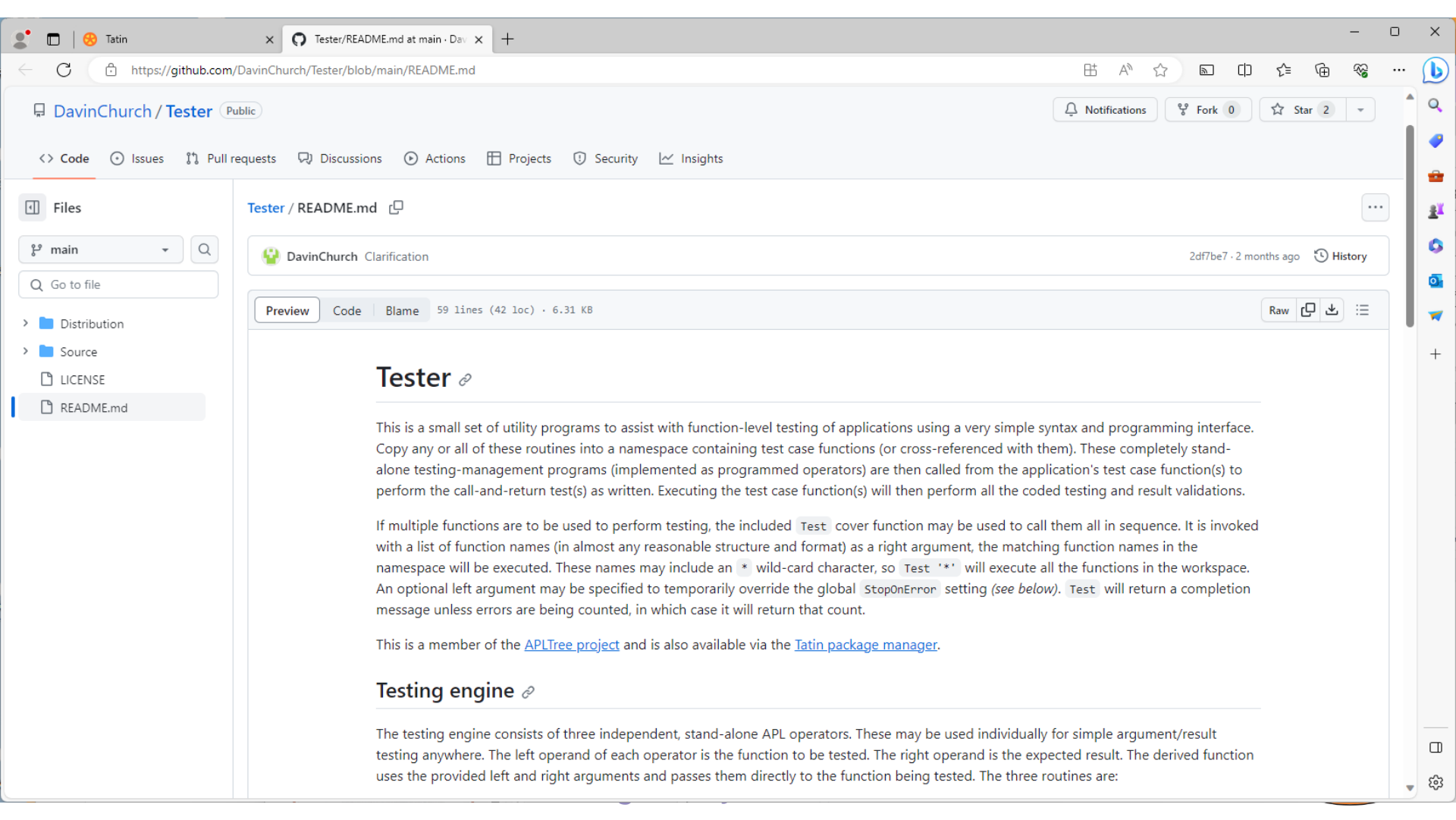
Package name	Description	Major Versions	Project URL	OS	UC	Tags
aplteam-CodeCoverage	Monitors which parts of an application got actually executed	1	github.com	Lin, Mac, Win		code-coverage,test-framework,unit-tests
aplteam-Latest	Lists APL objects by change date/time	1	github.com	Lin, Mac, Win	Yes	list-apl-objects
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>",
  minimumApiVersion: "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",
}
```



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...
<code>Pass</code>	Make sure the tested function returns the expected result, which is provided as the right operand value (<i>if a value is specified</i>). 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.
<code>Pass_</code>	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 <code>{1}</code> or <code>(1~)</code> is sufficient if no explicit verification is to be performed.
<code>Fail</code>	Make sure the tested function exits with a <code>⌊SIGNAL</code> as validated by the right operand. The right operand may be text to match <code>⌊DM</code> , a numeric (array) for <code>⌊EN</code> 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. <i>[Default]</i>
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.



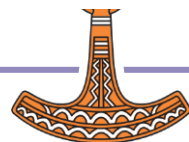
Stopping during testing

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

A in memoriam Ken Iverson, Kurt Gödel, Abu Dscha'far Muhammad ibn Musa al-Chwārizmī (أبو جعفر محمد بن موسى الخوارزمي)

A with gratitude to Adám Brudzewsky and in respect of Steve Mansour

A dedicated to my lovely wife

$$Avg \leftarrow \{$$
$$(w) \frac{d}{dt} = \frac{d}{dt} (w)$$
$$0 \neq \omega : 0$$

4

Median+{

$$(2 \div \sim 1 \perp \vdash \neg \sim \circ \in \Delta \neg \sim \circ \in \circ [2 \div \sim 0 \quad 1 + \neq])_{\omega}$$

4

Count+{A counts the number of elements in a vector

QW

}

: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,' ',ω}''Fail.aplo' 'Pass.aplo' 'Pass_.aplo' 'Test.aplf'
```

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

