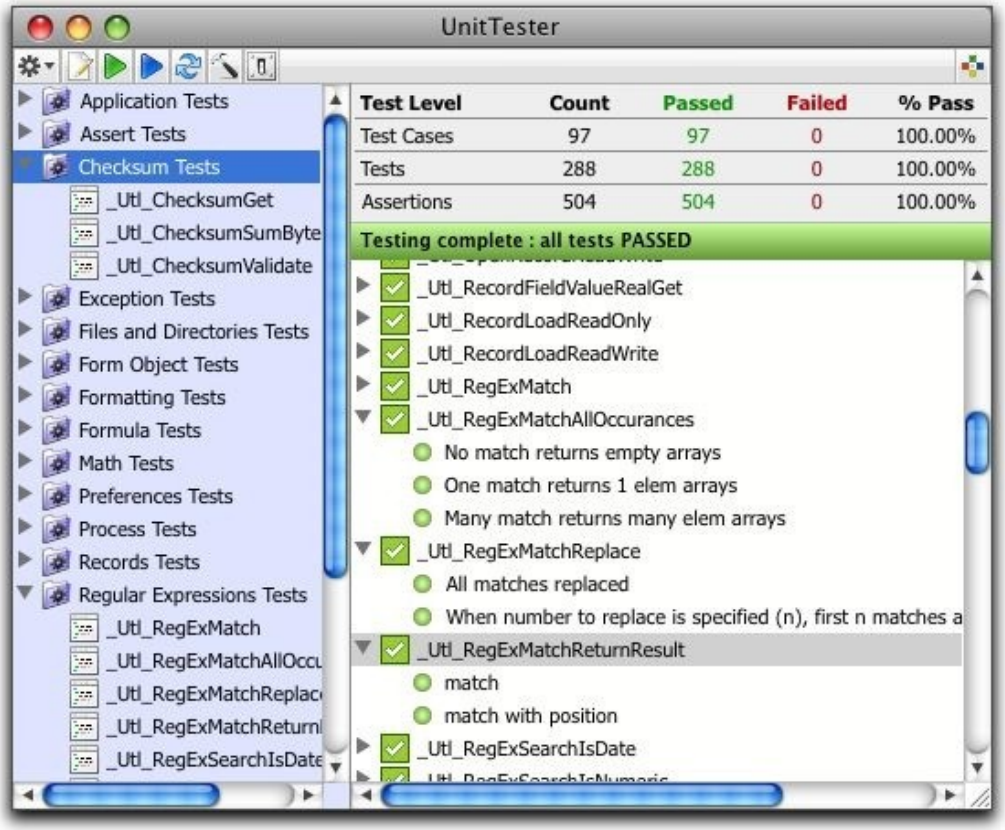
UnitTester Reference



Schaake Consulting, LLC

UnitTester version 1.6

Unit testing and test-driven development for 4D v11

Introduction

Thank you for trying UnitTester! I developed UnitTester after having spent a few months working with Java and using the JUnit testing framework. I realized the power of unit testing and how it can greatly reduce the stresses of software development. It killed me to think about going back to my 4D projects and not being able to use a nice unit testing framework. So, I decided to roll my own framework based on the xUnit framework standard. The result is UnitTester.

I know that unit testing is not an easy-to-grasp concept. I will try to provide a basic overview of why unit testing is a valuable and worthwhile addition to your development workflow. If you would like a more detailed discussion of unit testing, there are quite a few books written entirely on the subject. Though not written specifically about unit testing, I recommend *The Pragmatic Programmer* by Andrew Hunt and David Thomas. I think they do a nice job of concisely explaining unit testing and the value of doing so.

Thanks again for trying UnitTester. I hope you find, as I have, that adopting unit testing into your regular workflow makes you write better code while enjoying a considerable stress reduction. In short, unit testing makes programming more fun!

Sincerely,

Mark Schaake
Schaake Consulting, LLC

Installation

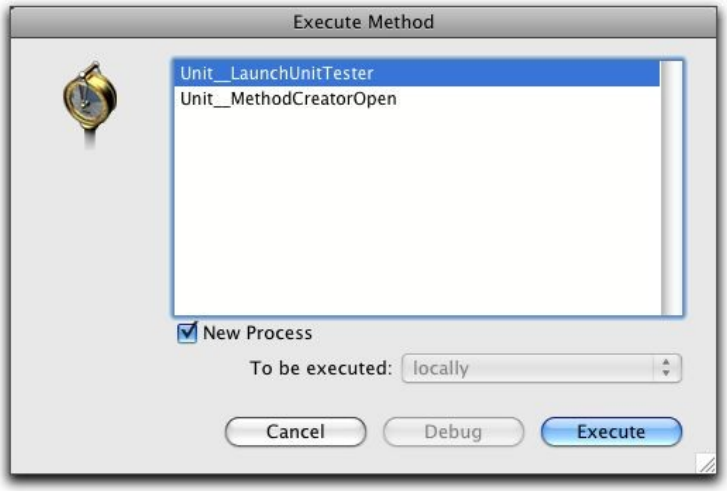
UnitTester consists of a 4D component and relies on the 4D Pack plugin and Pluggers Software's (Rob Laveaux) API Pack plugin.

Installing the Component

Copy the UnitTester component to your application's "Components" folder located next to the 4D structure file. If the "Components" folder does not exist, create it and then copy UnitTester into it.

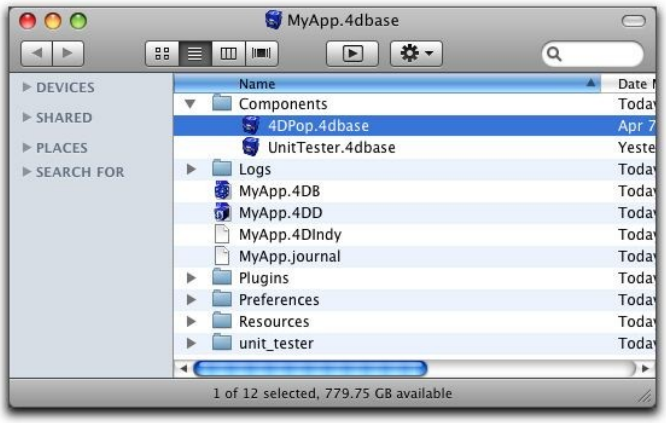
Installing the Plugins

UnitTester requires the 4D Pack and API Pack plugins to make interaction between UnitTester and the design environment possible. The first time you launch UnitTester by executing the method Unit__LaunchUnitTester (see screenshots below), it will detect whether or not the plugins are installed. If one or both are not installed, UnitTester will prompt you to let it install the plugins for you and restart the database. You should only have to do this once per database.



Launching UnitTester with 4D Pop

UnitTester works with the 4D Pop developer tool component freely available at http://www.4d.com/products/4dpop.html. If 4D Pop is installed and initialized, you will find a button for UnitTetser in the 4D Pop palette. Simply click the UnitTester button in the palette to launch it. This is an alternative to launching UnitTester via the “Execute Method” dialog (see Installing the Plugin).



The unit_tester Folder

When UnitTester is launched for the first time, it creates a folder containing preferences and test case method data next to the database structure file. Do not move, delete, or modify it or any of its contents. There are cases, however, in which you may wish to add files to this folder (see Customizing the Production Method Template).

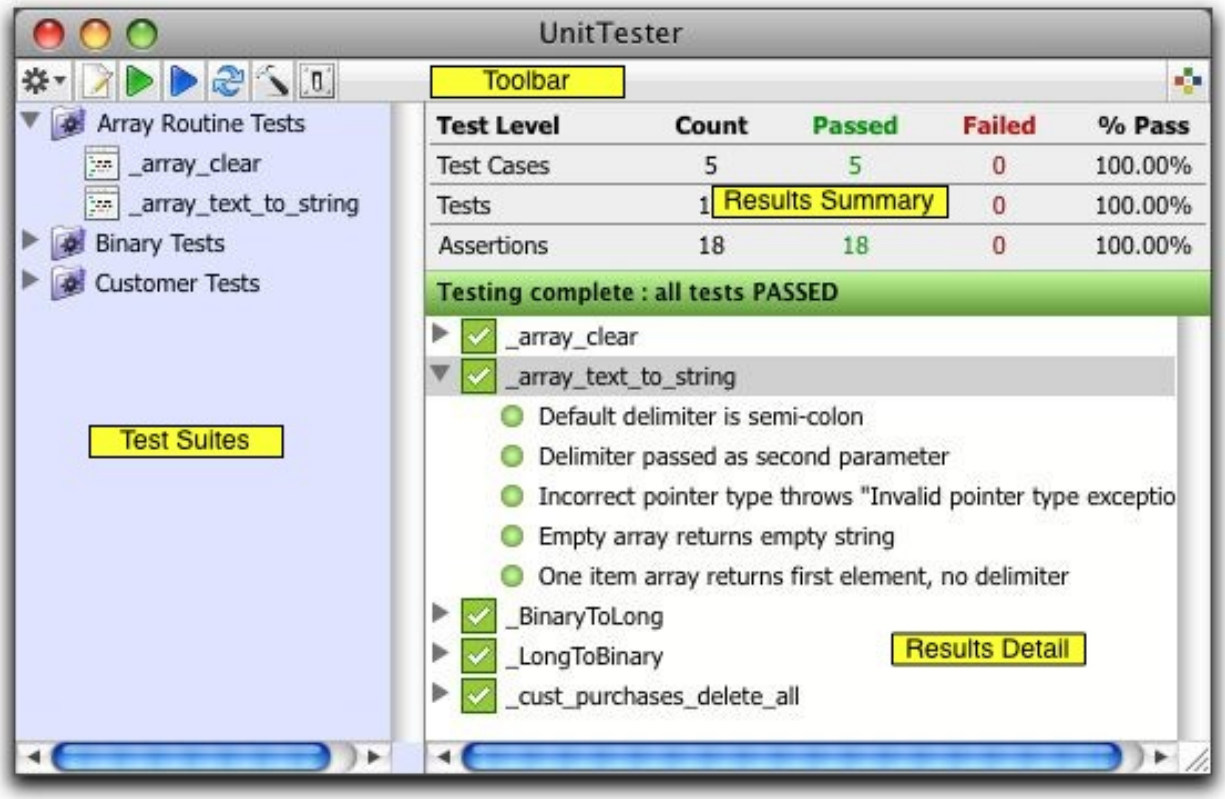
Methods Created in the Host Database

When launched for the first time, UnitTester will create a method in the host database for error handling purposes. This method is called:

_unit_error_handler

Note: You can have UnitTester generate wrapper methods for all UnitTester shared methods so that you can safely deploy without including the UnitTester component. Unfortunately, if you try to launch a compiled application that contains tokenized method calls to UnitTester methods, the application will not be able to launch. Therefore, you must make sure that no calls to UnitTester methods are tokenized. The way to do this is to generate wrapper methods in the host database that make calls to EXECUTE METHOD("Unit_<some_method>"). See the Macros section for more information.

The UnitTester GUI



The UnitTester GUI is composed of three main parts and a toolbar. The parts include:

- **Test Suites** - this is where you can manage test suites and select those you'd like to run
- **Results Summary** - the summary of test run results
- **Results Detail** - detail of test run results
- **Toolbar** - see the following section

The UnitTester Toolbar

UnitTester's toolbar contains the following buttons (from left to right):

- **Actions** - allows you to add and delete test suites and test cases (see The Test Suite Input Tool in the "Unit Testing Existing Applications" section for more info)
- **Create method** (⇧⌘M) – launches the Method Creator tool
- **Run all tests** (⇧⌘A) – runs all tests
- **Run selected tests** (⇧⌘S) – runs selected tests and suites of tests
- **Get updates from server** (⇧⌘U) – if in demo mode or with a team development license, will pull updates from the server that other developers have made since launching UnitTester
- **Preferences** (⇧⌘P) – opens the preferences dialog (see Preferences section)
- **Build application** (⇧⌘B) – if you've setup the build preferences in UnitTester's preferences (see Preferences), this will launch the build application process
- **Schaake Consulting** – provides access to license management, about box, and feedback

Keyboard Shortcuts

I recommend you try and use the keyboard shortcuts. When designing the GUI, I kept the buttons small to minimize the screen footprint of UnitTester and because I personally like using shortcuts in place of clicking buttons wherever possible for efficiency's sake. For Windows, substitute references to ⌘ with ^ (control key).

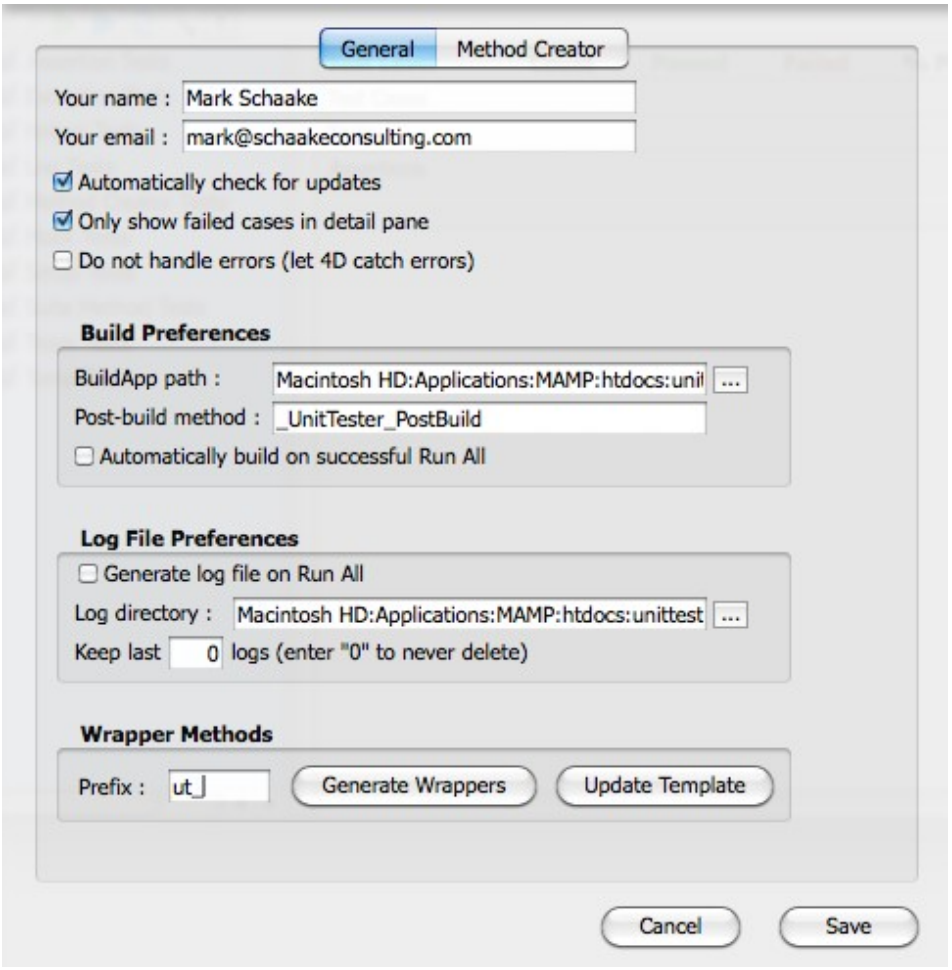
Actions

The "Actions" button when clicked brings up a contextual menu. This menu can also be brought up by right-clicking (or control clicking) on the Test Suites area.

Preferences

General Preferences

The General preferences include setting your name and email for feedback purposes (when you submit bugs or feature requests to Schaake Consulting), automatically checking for updates (once per day), the ability to have UnitTester show only failed test cases in the detail pane, and an option to turn off UnitTester's error handler so that 4D will display errors as opposed to error info being displayed in the detail pane.



Build Preferences

The Build preferences are useful if you are building an application and would like UnitTester to automate the process for you. In the BuildApp path setting you put the path to the 4D build xml document. For more information, see 4D's document "XML Keys BuildApplication" available at http://www.4d.com/support/documentation.html. In the Post-build method setting, you can specify a method to be executed once the build has completed successfully. You could, for ex-

ample, have certain documents copied into the built application after the build, and then have the built application be zipped into an archive for deployment. You can also set UnitTester to automatically start a build in the event you've done a Run All and all tests passed.

Log File Preferences

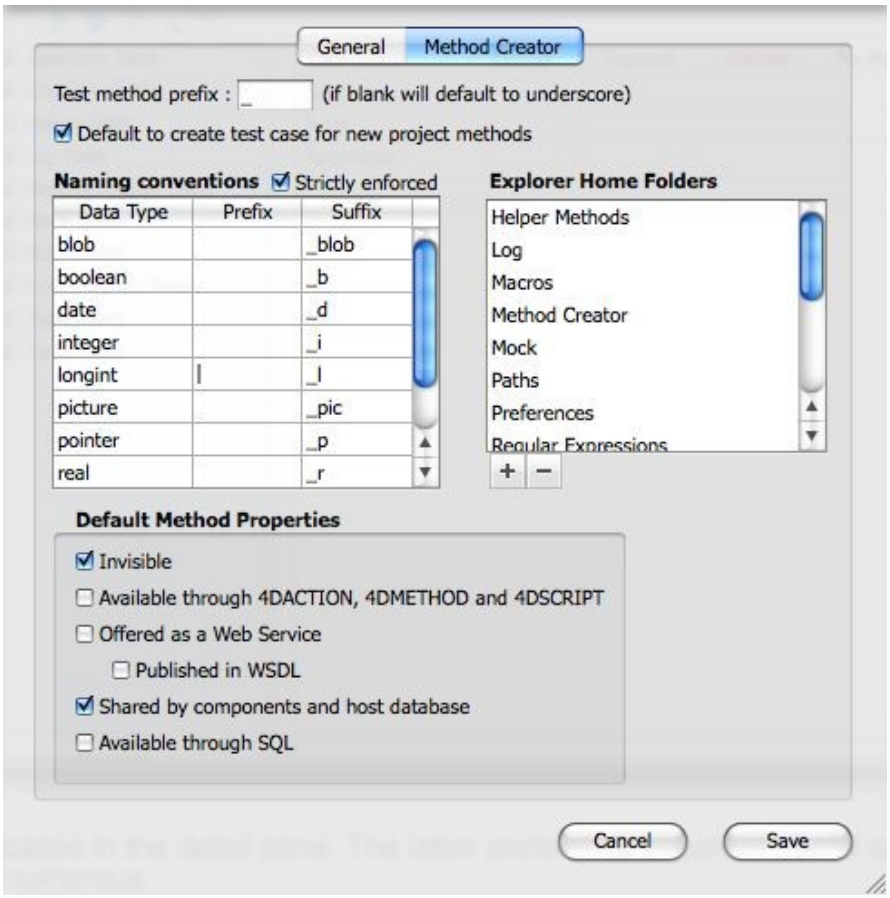
The Log File preferences allow you to automatically generate both XML and HTML log files each time a “run all tests” is performed. You specify the location for the log files to be written, and the number of log files to keep (0 never deletes).

Wrapper Methods Preferences

The Wrapper Methods Preferences is where you can generate wrapper methods on the host to make deployment without UnitTester possible. Simply set a prefix, then click "Generate Wrappers" and all UnitTester shared methods will be wrapped into methods on the host. The "Update Template" button automatically updates the UnitTemplate.txt file with the correct wrapper methods.

Method Creator Preferences

The Method Creator preferences are related to the Method Creator tool. Settings include the test case method prefix, default to create a test case method when creating a production method via the Method Creator, and coding conventions. By setting your naming conventions, Method Creator will detect the appropriate data type based on the names you give the return variable and parameters. You'll find this is a useful productivity enhancement.



Naming Conventions Discussion

Naming conventions are rules or guidelines used for writing code. Adopting and abiding by conventions can make your code more readable while helping you to write code faster. This is because with established conventions, you don’t have to think as much about naming your variables and methods. You just follow the convention rules. Some example conventions could be:

- Methods should be camel case (e.g. camelCase)
- Variables should be lower case and words delimited by underscores (e.g. my_variable_t)
- Variables should be suffixed with the data type indicator (_blob for BLOB, _l for LONGINT, etc.)

Explorer Home Folders

The Explorer Home Folders list is managed automatically by the Method Creator tool. Its purpose is to “remember” the folder names you have input in the Method Creator tool form when indicating the folder in which to generate the production method. You have the ability to edit this list in the preferences, which may be useful if you rename or delete a Home Folder in the Explorer or delete.

Unit Testing

Unit testing is the cornerstone of Agile software development methodologies. If you have not heard of Agile software development methodologies such as Scrum or Extreme Programming, I recommend starting with the Wikipedia entry (just do a Google search on "Agile software development"). Basically, the goal of Agile methods is to maximize productivity while minimizing wasteful activities. Think "The Toyota Way" for software development.

Before we start talking testing theory, it might help to establish some testing terminology. For the purposes of UnitTester, the three terms we'll focus on will be unit testing, integration testing, and regression testing. There are quite a few other testing terms that we won't discuss, such as system testing, GUI testing, and stress testing. Let's establish definitions to the three testing terms UnitTester is concerned with.

Testing Terms

Unit Testing	Testing to make sure the smallest parts of a program (the units) work individually exactly as intended. An example would be a test that makes sure an ArraySum method produces the expected results under various criteria.
Integration Testing	Testing to make sure the smallest parts of a program (the units) work together in modules exactly as intended.
Regression Testing	Testing to make sure changes to a program haven’t broken previously working parts of the program.

Why Unit Test?

For me, the most important benefit to unit testing is the ability to do regression testing. A close second is the improvement in my coding quality and consistency as a result of committing to test-driven development.

Many people are intimidated by the prospect of incorporating unit testing into a project. The reason, I think, is that if you are going to do it right, you will end up writing nearly as much code for unit tests as you do for production. Sounds exhausting and wasteful, doesn't it? Once you get in the unit testing groove, however, you'll find that your overall productivity increases as you start to approach writing production code differently in such a way that you force yourself into writing cleaner, smaller, more modular project methods that result in an easier to read and much more maintainable code base.

Back to the testing terms... I find it helpful to think of unit testing as something that facilitates both integration testing and regression testing. The latter two tests are the real payoff to unit testing. Once you understand unit tests, you'll see that writing unit tests are simply a means to running reliable integration tests, which in turn are a means to running reliable regression tests, which when done well can become an indispensable part of your development process. For me personally, committing to unit testing (and specifically test-driven development) has resulted in two revolutionary improvements in my development work: 1) I write better, much more maintainable code the first time, and 2) I have much less stress due to the reduced risk of introducing uncaught new bugs with new feature development.

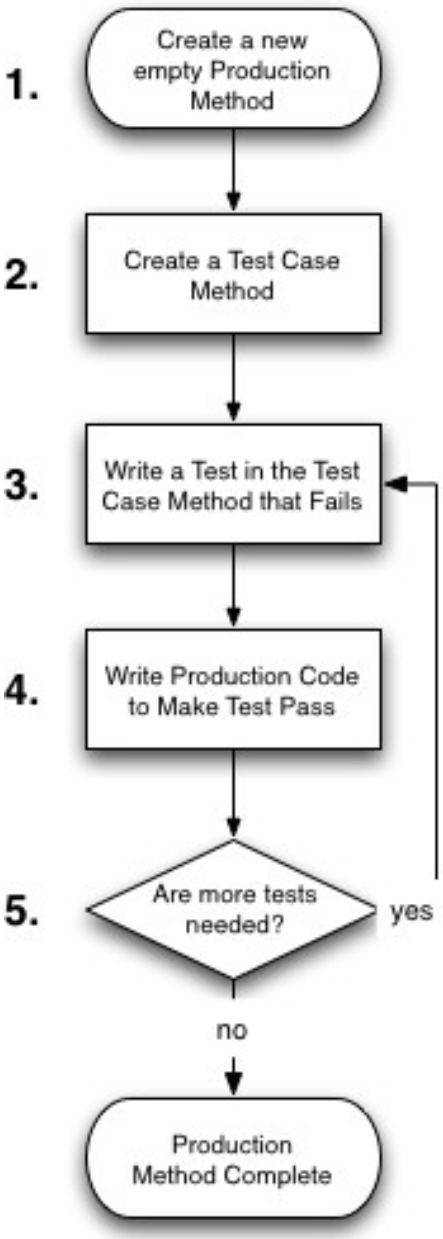
Finally, there are many books / articles / websites devoted to the discussion of unit testing that I would recommend you check out. I've posted links to websites on the UnitTester page at http://www.unittester4d.com.

Some good books that discuss unit testing include:

- *The Pragmatic Programmer* by Andrew Hunt and David Thomas
- *The Productive Programmer* by Neal Ford
- *Lean Software Development: An Agile Toolkit* by Mary and Tom Poppendieck

Test-Driven Development (TDD)

The Method Creator tool included with UnitTester is provided to faciliate test-driven development (TDD). The basic idea is that you write your tests prior to writing the production code necessary to make the tests pass. The process works like this:

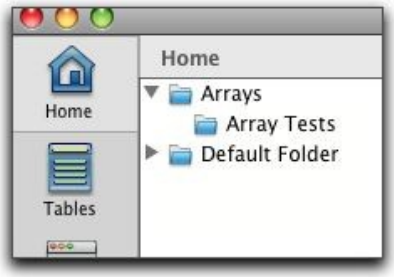


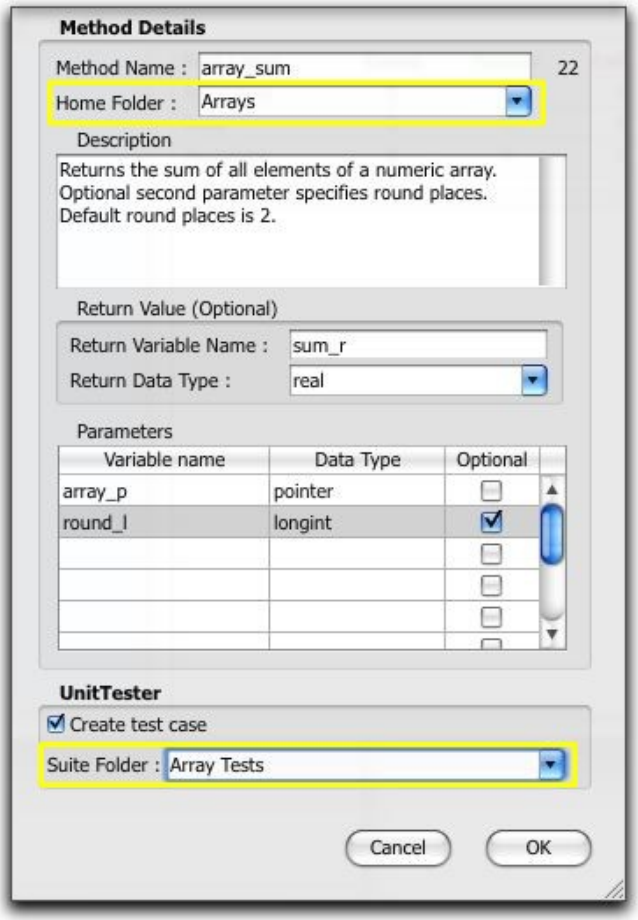
The Method Creator - a Tool for TDD

The Method Creator tool provides many conveniences for quickly creating both a production method and test case method simultaneously. Some of the features include:

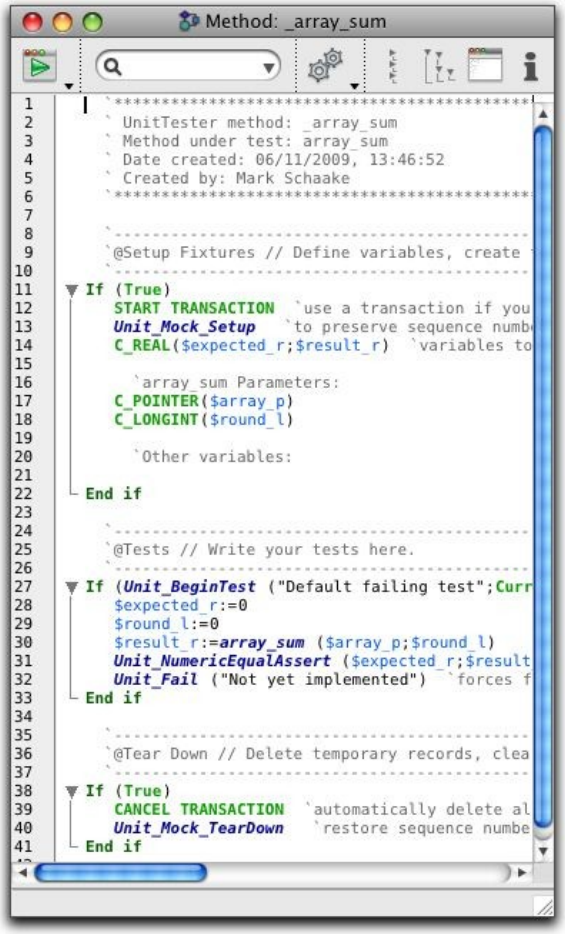
- Automatic data type detection based on your naming conventions defined in the UnitTester preferences
- Type-ahead fields to speed up data entry
- If your production method returns a value and you let Method Creator generate a test case method for you, a first test will be automatically generated

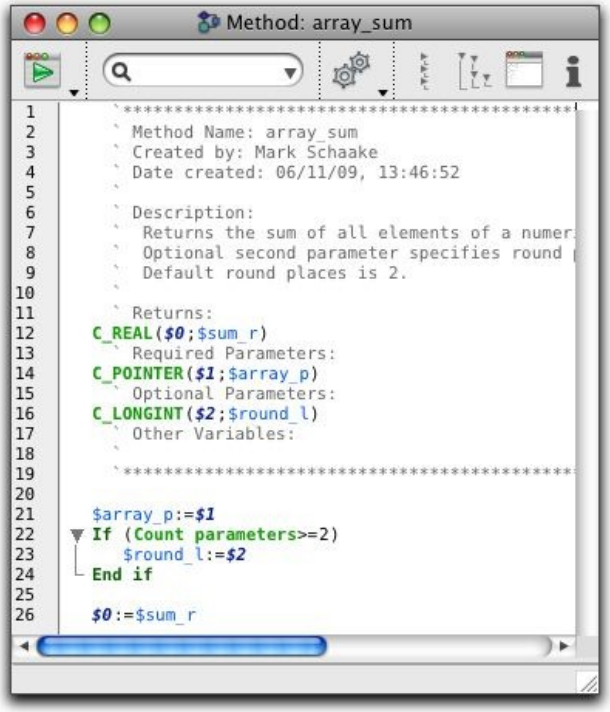
The Method Creator tool utilizes folders you have set up in the Explorer -> Home view in the Design environment. When creating methods with the Method Creator, you specify which Home Folder in which the production method should be created, as well as the Suite Folder in which the test case method should be created. In the example below, we have set up two folders in the Home view: “Arrays” and “Array Tests.” In this case, our production method ***array_sum*** will be created in the “Arrays” folder, and our test case method ***_array_sum*** will be created in the “Array Tests” folder.

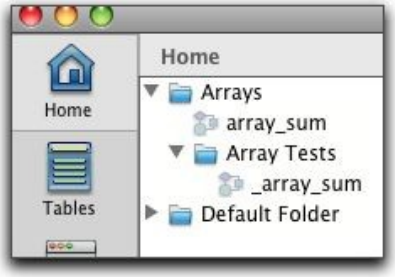




Following are screenshots of the resulting production and test case methods after clicking “OK”:





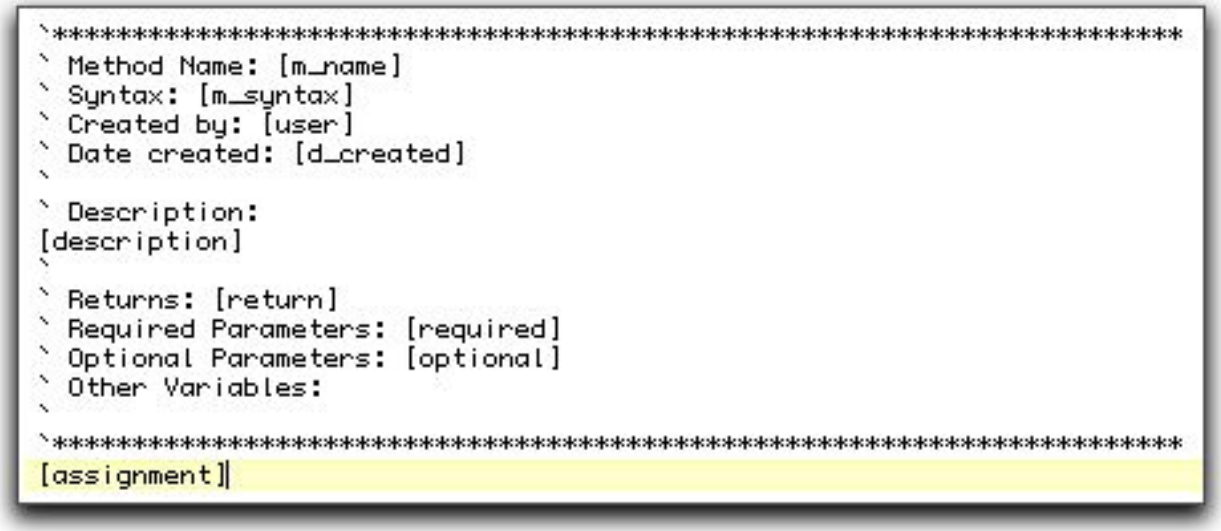


Customizing the Production Method Template

UnitTester comes with a default method template used when creating a method via the Method Creator. You can override this template by providing your own and saving it in the "unit_tester" folder next to your structure file. It must be named "MethodHeaderTemplate.txt." If you would like to create your own template, I recommend starting with the template UnitTester ships with. You can find it inside the UnitTester component's "Resources" folder (not the host database's "Resources" folder), called "MethodHeaderTemplate.txt." You will find the following tags in the template that should be in your custom template as well:

- **[m_name]** - the method's name
- **[m_syntax]** – the method's syntax e.g. MyMethod(required {; optional}) → string
- **[user]** - the user's name
- **[d_created]** - timestamp of the method's creation
- **[description]** - the description of the method
- **[return]** - return variable declaration
- **[required]** - required parameters declaration
- **[optional]** - optional parameters declarations
- **[assignment]** - assignment of return and parameters to local variables

Screenshot of the MethodHeaderTemplate.txt document contents:



Unit Testing Existing Applications

Unit testing is most closely associated with test-driven development. But what if you have an existing application? Can you realize benefits of unit testing when you have already written code without unit tests? The major advantage to beginning a project with test-driven development is the development of comprehensive regression tests that cover most of the production code. With large applications built without unit tests, it probably doesn't make sense to try and write unit tests for all of that pre-existing and untested code. However, it is possible and very valuable to begin integrating test-driven development into an existing application in the following way:

- Commit to test-driven development for all new production methods
- When a bug is found, first write a test that fails because of the bug, then fix your production code so that the test passes
- When re-factoring, write tests to ensure code still functions as expected

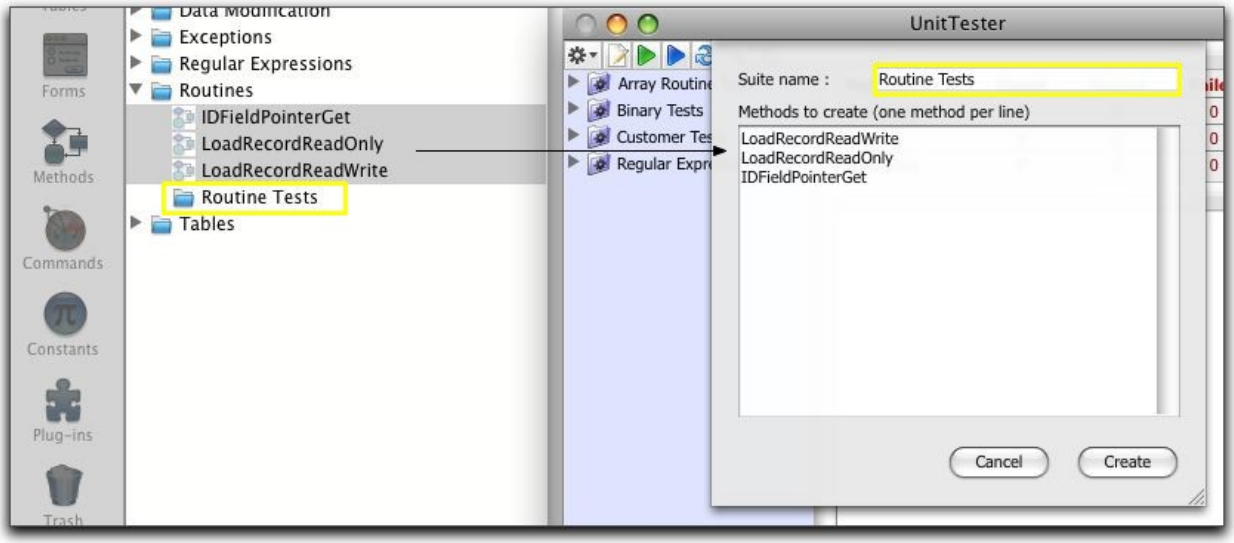
By committing to unit testing you will start to grow a set of regression tests and find that the quality of the code you write will improve.

The Test Suite Input Tool

If you'd like to unit test existing methods, I recommend using the Test Suite Input tool by selecting "New Test Suite" from the Actions contextual menu. This tool facilitates testing existing methods by allowing yo to create a test suite and generate test case methods for many production methods all at once. You can select a number of methods from the Explorer and drag and drop the selection into the "Methods to create" field. Once you click "Create," a test case method will be generated for each method and given the appropriate prefix set in the UnitTester preferences. Prior to clicking "Create," you should create a folder in the Home view of the Explorer with the same name as the Suite name. This is the folder in which the test case methods will be created.

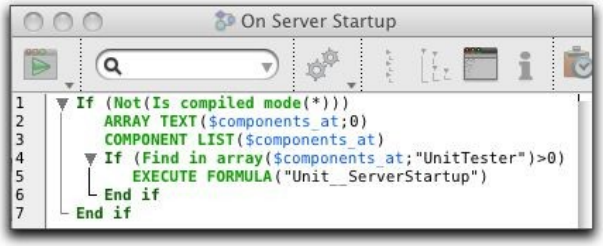
You can add test case methods in batches to existing test suites by double-clicking the suite in UnitTester to re-open the Test Suite Input Tool. This way, you can spread out the development of test case methods by (for example) adding three test cases at a time. This way, you won't ge overwhelmed with 30 automatically generated (and failing) test cases.

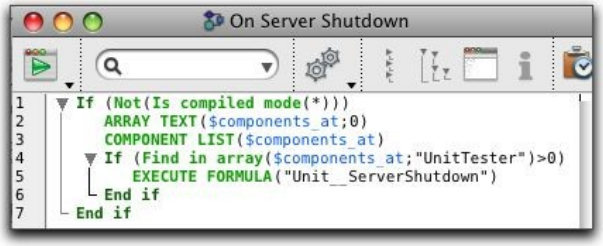
The Test Suite Input / Editor



Team Development

UnitTester is designed to work in both single user and client-server (team) development environments. In demo mode (no license), both single user and team functionalities are available but limited by the number of tests and assertions you can run at a time. If you intend to use UnitTester in a client-server development environment, you will need a team license and to make calls to Unit__ServerStartup and Unit__ServerShutdown in the *On server startup* and *On server shutdown* database methods, respectively.





UnitTester maintains the synchronization of test case methods in a server process. When a client launches UnitTester and adds a test case method, other clients can pull updates from the server process by clicking the synchronize toolbar button in the Unit-Tester GUI.

UnitTester ships with macros for facilitating setting up the client-server environment. See the "Macros" section for specifics on these macros.

Tutorial

Introduction

For this tutorial, I will be constructing an email address validation routine called ***email_address_is_valid***. Our goal is to practice test-driven development (TDD) to construct our production method. Let's begin by deciding on the format of our production method.

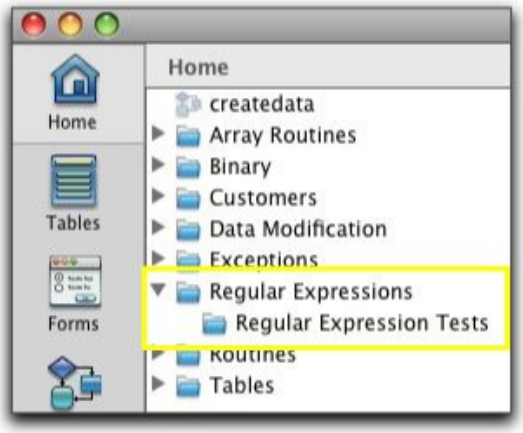
email_address_is_valid(address) ⟶ boolean true/false

Our production method will take an email address string as a parameter and return true if the email address is well formed, and false if not.

Preparing the Environment

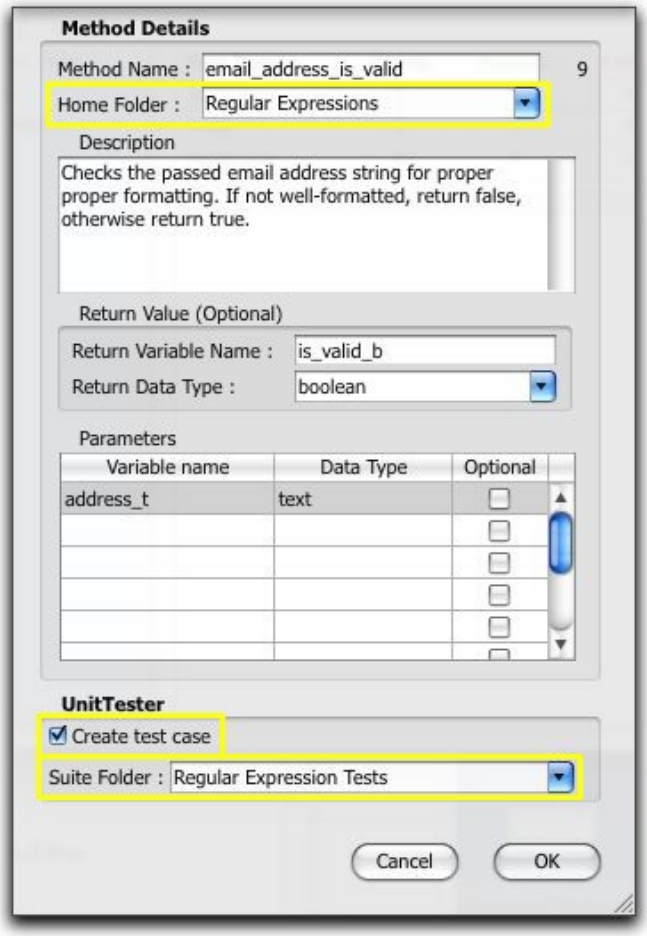
Since we are practicing TDD, I will use the Method Creator tool to generate both our production method and a test case method simultaneously. In preparation, I will first set up two folders in our Explorer -> Home view. Since I know I will be using regular expressions to help validate emails, I have created two folders to hold routines for regular expressions:

- Regular Expressions
- Regular Expression Tests



Using the Method Creator

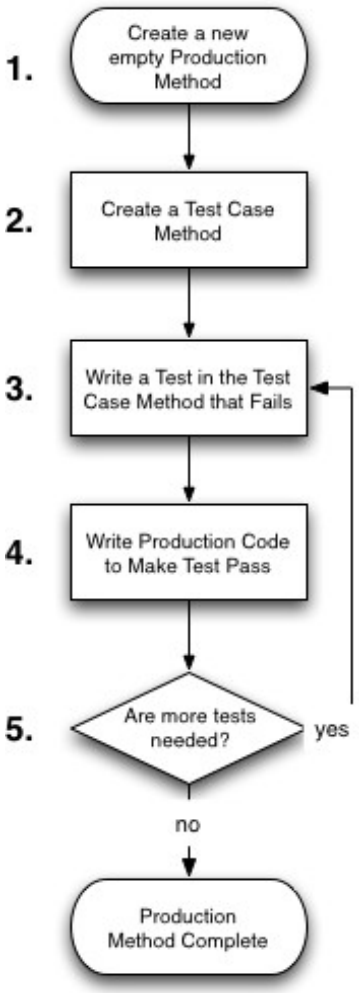
Now that we have our Home Folder structure prepared for the Method Creator, it is time to launch it. After filling out the fields in the Method Creator, we have the following:



Notice we have entered the names of the folders we just created in the Explorer. The Method Creator will remember these entries so the next time you want to create a method in the same folder, type-ahead will make entry simple. Also, notice that I've opted to "Create test case." This tells Method Creator to generate a test case method as well as the production method. Using our preferences, the test case method will have the same name as our production method but prefixed by an underscore:

```
_email_address_is_valid
```

Once we click "OK," Method Creator generates the two project methods and we have begun the TDD cycle for our production method. As a review, following is a chart illustrating the TDD cycle:



Planning our Tests

At this point, Method Creator has performed steps 1 and 2 for us. Now we need to start thinking about the functionality of our production method and how we can test it. We will construct a test plan for our test case method. Since we are practicing TDD, we will write one test at a time, and not write the next test until the first one passes in UnitTester.

After some thinking, I have come up with this test plan which is simply a list of scenarios our production method should be able to handle:

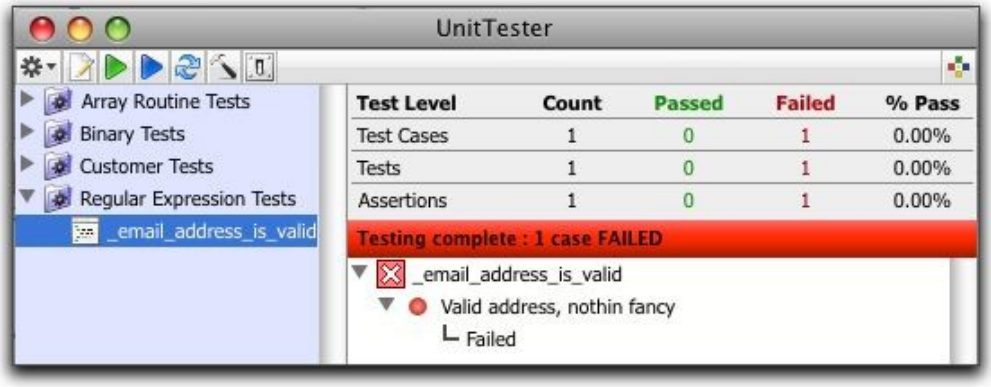
- A plain old valid address (of course!)
- A valid address that contains a period or some other punctuation prior to the “@” sign like john.smith@gmail.com or john_smith@gmail.com
- An invalid address that contains an illegal character before the “@” sign like john[smith@gmail.com
- An invalid address due to a suffix that is too short like johnsmith@gmail.c
- An invalid address due to having a suffix that is too long like johnsmith@gmail.comm

Writing the First Test

Now that we have our test plan in place, we can write our first test. In this case, we want to have our production method return true when passed a plain valid email address. The Method Creator automatically generates a “Default failing test” in our test case method. I will replace it with our first test. Here is what I’ve come up with in the test case method *_email_address_is_valid*:

```
`----------------------------------------------------------------------
`@Tests // Write your tests here.
`----------------------------------------------------------------------
If (Unit_BeginTest ("Valid address, nothin fancy";Current method name))
   $address_t:="johnsmith@gmail.com"
   $result_b:=email_address_is_valid ($address_t)
   Unit_TrueAssert ($result_b)
End if
```

At this point, we should run our test even though we haven’t yet written any production code. We find that our test fails:

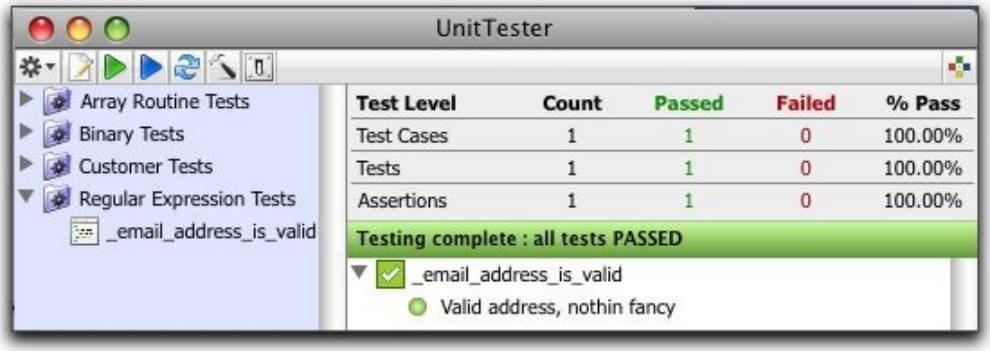


This is what we expect. At this point, we will write some production code, but only enough to make this first test pass. Here is what I’ve come up with:

email_address_is_valid:

```
20    $address_t:=$1
21
22    $regEx_t:="\\w+@\\w+\\.\\w+"
23    $is_valid_b:=Match regex($regEx_t;$address_t)
24
25    $0:=$is_valid_b
```

I know that this regular expression is not going to cut it for some of our other tests, but all I'm trying to do at this point is get the first test to pass as easily as possible, which it does with this pattern.



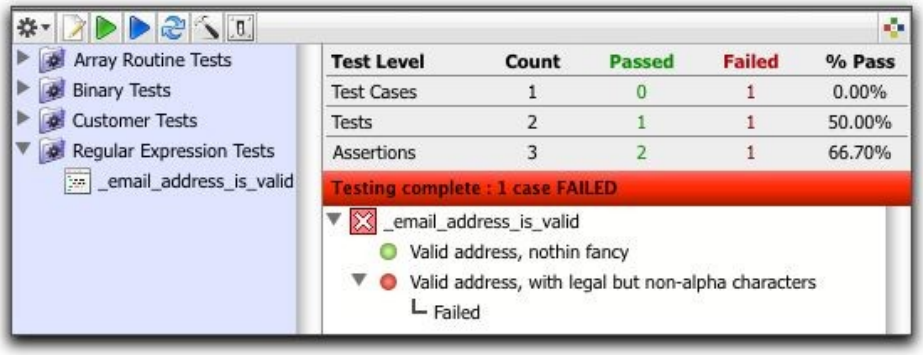
Writing the Second Test

Next, we write our second test to handle the second scenario: an email with a legal but non-alpha character in the first part. So we add another test to our test case method and now have:

```
   `------------------------------------------------------------------------
   `@Tests // Write your tests here.
   `------------------------------------------------------------------------
If (Unit_BeginTest ("Valid address, nothin fancy";Current method name))
   $address_t:="johnsmith@gmail.com"
   $result_b:=email_address_is_valid ($address_t)
   Unit_TrueAssert ($result_b)
End if

If (Unit_BeginTest ("Valid address, with legal but non-alpha characters";Current method name))
      `first try period
   $address_t:="john.smith@gmail.com"
   $result_b:=email_address_is_valid ($address_t)
   Unit_TrueAssert ($result_b)
      `next try underscore
   $address_t:="john_smith@gmail.com"
   $result_b:=email_address_is_valid ($address_t)
   Unit_TrueAssert ($result_b)
End if
```

As you can see, in the second test I've written two assertions: one for an address with a period, and one for an address with an underscore. You may want to write more assertions for the other legal characters. When we run the test case again in UnitTester, we find that it fails due to our second test we've just written (step 3 in the TDD cycle):



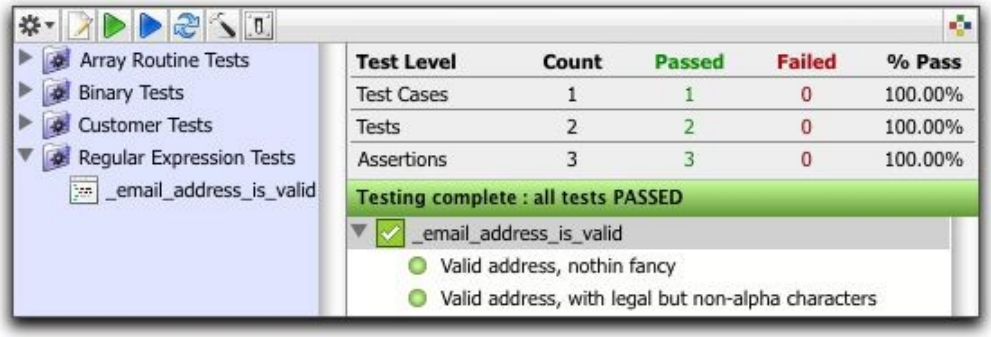
At this point, we have to modify our production method so that both tests pass which will make our test case pass. Here is the regular expression I've come up with that accomplishes this goal:

```
$address_t:=$1

$regEx_t:="[a-zA-Z0-9._-]+@\\w+\\.\\w+"
$is_valid_b:=Match regex($regEx_t;$address_t)

$0:=$is_valid_b
```

This change to the regular expression in our production method allowed both of our first two tests to pass.



TDD Discussion

Hopefully at this point you can see the TDD cycle at work, and see how you can get into a rhythm of write test - write code - write test - write code - etc. Sometimes you will find that you write a test that passes without having to write or modify the production code at all. In these instances, it is still good to write the tests because they are verifying the production code is doing what it is supposed to do. The result is you build confidence in your code by having the ability to run automated tests to verify it does what it is supposed to do. A side benefit is that well-named tests can provide use case documentation for your production methods.

Instead of going through the remaining tests one-by-one, I will instead list the final test case method. Note that there may still be bugs in this method, but when these bugs are caught, we would first update our tests to catch them (and thus fail) prior to modifying the production code to eliminate the bugs.

One major benefit of practicing TDD is time spent in the 4D Debugger is greatly reduced. Instead of tracing values, you can rely on UnitTester to inform of you of the discrepency between expected and actual results. Don't just take my word for it:

"You will spend much less time in the 4D debugger if you adopt TDD" - Walt Nelson (Seattle)

The Final Test Case Method

```
   `@Tests // Write your tests here.
   `-----------------------------------------------------------------
If (Unit_BeginTest ("Valid address, nothin fancy";Current method name))
   $address_t:="johnsmith@gmail.com"
   $result_b:=email_address_is_valid ($address_t)
   Unit_TrueAssert ($result_b)
End if

If (Unit_BeginTest ("Valid address, with legal but non-alpha characters";Current method name))
     `first try period
   $address_t:="john.smith@gmail.com"
   $result_b:=email_address_is_valid ($address_t)
   Unit_TrueAssert ($result_b)
     `next try underscore
   $address_t:="john_smith@gmail.com"
   $result_b:=email_address_is_valid ($address_t)
   Unit_TrueAssert ($result_b)
End if

If (Unit_BeginTest ("Invalid address, illegal character in local part";Current method name))
   $address_t:="john[smith@gmail.com"
   $result_b:=email_address_is_valid ($address_t)
   Unit_FalseAssert ($result_b)
End if

If (Unit_BeginTest ("Invalid address due to suffix smaller than 2 characters";Current method name))
   $address_t:="johnsmith@gmail.c"
   $result_b:=email_address_is_valid ($address_t)
   Unit_FalseAssert ($result_b)
End if

If (Unit_BeginTest ("Invalid address due to suffix greater than 4 characters";Current method name))
   $address_t:="johnsmith@gmail.comm"
   $result_b:=email_address_is_valid ($address_t)
   Unit_FalseAssert ($result_b)
End if
```

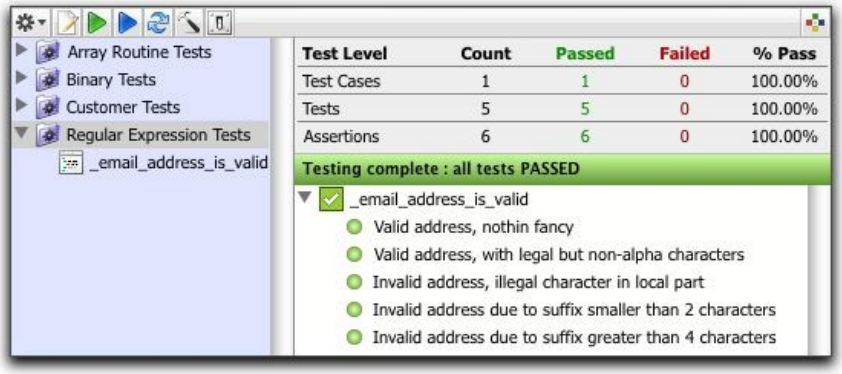
The Final Production Method

```
$address_t:=$1

$regEx_t:="([a-zA-Z0-9._\\-!#?%\\&`*$+/='{|}\\^~]){1,64}@([a-zA-Z0-9\\-.]){1,255}\\.(\\w){2,4}"
$is_valid_b:=Match regex($regEx_t;$address_t)

$0:=$is_valid_b
```

All our tests pass!


Array Routine Tests
Binary Tests
Customer Tests
Regular Expression Tests
_email_address_is_valid
Test Level
Count
Passed
Failed
% Pass
Test Cases
1
1
0
100.00%
Tests
5
5
0
100.00%
Assertions
6
6
0
100.00%
Testing complete : all tests PASSED
_email_address_is_valid
Valid address, nothin fancy
Valid address, with legal but non-alpha characters
Invalid address, illegal character in local part
Invalid address due to suffix smaller than 2 characters
Invalid address due to suffix greater than 4 characters

Assertion Methods

Assertion methods are the heart and soul of unit tests. Each unit test must have at least one call to an assertion method in order for it to be a test. In general, the format of assertions are as follows:

Unit_<DataType>EqualAssert`(expected; actual{; message})`

However, there are exceptions:

- The two boolean assertion methods ***Unit_TrueAssert*** and ***Unit_FalseAssert*** take only one required parameter: a boolean expression.
- The two Numeric assertion methods ***Unit_NumericEqualAssert*** and ***Unit_Numeric-NotEqualAssert*** take an additional precision tolerance parameter.

Unit_ArrayEqualAssert

Unit_ArrayEqualAssert(expected; actual{; message})

Parameter	Type		Description
expected	Pointer	⟶	Pointer to an array containing the expected values.
actual	Pointer	⟶	Pointer to an array containing the actual values.
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two arrays and asserts equality. Optional third parameter message provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`Test to make sure our production method TextArrayAppend works as expected
`@Setup
ARRAY TEXT($actualNames_at;0)
ARRAY TEXT($expectedNames_at;4)
`@Tests
If(Unit_BeginTest("appending to text array";Current method name))
     $expectedNames_at{1}:="Mark"
     $expectedNames_at{2}:="Mike"
     $expectedNames_at{3}:="Matt"
     $expectedNames_at{4}:="Mitch"
     ArrayClear(->$actualNames_at) ` make sure we start with an empty array
     TextArrayAppend(->$actualNames_at;"Mark";"Mike";"Matt";"Mitch")
     Unit_ArrayEqualAssert(->$expectedNames_at;->$actualNames_at)
End if
```

Unit_ArrayNotEqualAssert

Unit_ArrayNotEqualAssert(unexpected; actual{; message})

Parameter	Type		Description
unexpected	Pointer	⟶	Pointer to an array containing the unexpected values.
actual	Pointer	⟶	Pointer to an array containing the actual values.
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two arrays and asserts inequality. Optional third parameter message provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`Test to make sure our production method TextArrayAppend works as expected
`@Setup
ARRAY TEXT($actualNames_at;0)
ARRAY TEXT($expectedNames_at;4)
`@Tests
If(Unit_BeginTest("appending to text array";Current method name))
     $expectedNames_at{1}:="Mark"
     $expectedNames_at{2}:="Mike"
     $expectedNames_at{3}:="Matt"
     $expectedNames_at{4}:="Mitch"
     ArrayClear(->$actualNames_at) ` make sure we start with an empty array
     TextArrayAppend(->$actualNames_at;"Mark";"Mike";"Matt";"Mitch")
     Unit_ArrayEqualAssert(->$expectedNames_at;->$actualNames_at)
End if
```

Unit_BLOBEqualAssert

Unit_BLOBEqualAssert(expected; actual{; message})

Parameter	Type		Description
expected	BLOB	⟶	Expected BLOB
actual	BLOB	⟶	Actual BLOB
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two BLOBs and asserts equality. Optional third parameter message provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`Test to make sure our production method DuplicateBLOB works as expected
`@Setup
C_BLOB($expectedBlob;$blob)
TEXT TO BLOB("some text to store";$expectedBlob)

`@Tests
If(Unit_BeginTest("a test description";Current method name))
     `make sure blobs are not already equal
     Unit_BLOBNotEqualAssert($expectedBlob;$blob;"blobs already equal")
     $blob:=DuplicateBLOB($expectedBlob)
     Unit_BLOBEqualAssert($expectedBlob;$blob)
End if
```

Unit_BLOBNotEqualAssert

Unit_BLOBNotEqualAssert(unexpected; actual{; message})

Parameter	Type		Description
unexpected	BLOB	⟶	Unexpected BLOB
actual	BLOB	⟶	Actual BLOB
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two BLOBs and asserts *inequality*. Optional third parameter message provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`Test to make sure our production method DuplicateBLOB works as expected
`@Setup
C_BLOB($expectedBlob;$blob)
TEXT TO BLOB("some text to store";$expectedBlob)

`@Tests
If(Unit_BeginTest("a test description";Current method name))
     `make sure blobs are not already equal
     Unit_BLOBNotEqualAssert($expectedBlob;$blob;"blobs already equal")
     $blob:=DuplicateBLOB($expectedBlob)
     Unit_BLOBEqualAssert($expectedBlob;$blob)
End if
```

Unit_DateEqualAssert

Unit_DateEqualAssert(expected; actual{; message})

Parameter	Type		Description
expected	Date	⟶	Expected date
actual	Date	⟶	Actual date
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two dates and asserts equality. Optional third parameter message provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`@Setup
C_DATE($expectedDate;$actualDate)
`@Tests
If(Unit_BeginTest("Y2K returns 01/01/2000";Current method name))
     $expectedDate:=!01/01/2000!
     $actualDate:=Dates_Y2K
     Unit_DateEqualAssert($expectedDate;$actualDate)
End if
```

Unit_DateNotEqualAssert

Unit_DateNotEqualAssert(unexpected; actual{; message})

Parameter	Type		Description
unexpected	Date	⟶	Unexpected date
actual	Date	⟶	Actual date
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two dates and asserts *inequality*. Optional third parameter message provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`@Setup
C_DATE($expectedDate;$actualDate)
`@Tests
If(Unit_BeginTest("Y2K returns 01/01/2000";Current method name))
     $expectedDate:=!01/01/2000!
     $actualDate:=Dates_Y2K
     Unit_DateEqualAssert($expectedDate;$actualDate)
End if
```

Unit_IntegerEqualAssert

Unit_IntegerEqualAssert(expected; actual{; message})

Parameter	Type		Description
expected	Integer	→	Expected integer
actual	Integer	→	Actual integer
message	String	→	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two integers and asserts equality. Optional third parameter message provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`@Setup
ARRAY TEXT($arr_t;5) `initialize an array to size 5
`@Tests
If(Unit_BeginTest("sets size of array to zero";Current method name))
     Unit_IntegerNotEqualAssert(0;Size of array($arr_t))
     ArrayClear(->$arr_t)
     Unit_IntegerEqualAssert(0;Size of array($arr_t))
End if
```

Unit_IntegerNotEqualAssert

Unit_IntegerNotEqualAssert(unexpected; actual{; message})

Parameter	Type		Description
unexpected	Integer	⟶	Unexpected integer
actual	Integer	⟶	Actual integer
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two integers and asserts *inequality*. Optional third parameter message provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`@Setup
ARRAY TEXT($arr_t;5) `initialize an array to size 5
`@Tests
If(Unit_BeginTest("sets size of array to zero";Current method name))
     Unit_IntegerNotEqualAssert(0;Size of array($arr_t))
     ArrayClear(->$arr_t)
     Unit_IntegerEqualAssert(0;Size of array($arr_t))
End if
```

Unit_NumericEqualAssert

Unit_NumericEqualAssert(expected; actual{; tolerance {; failure})

Parameter	Type		Description
expected	Numeric	⟶	Expected numeric value
actual	Numeric	⟶	Actual numeric value
tolerance	Numeric	⟶	Allowable difference between expected and actual
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two numerics and asserts equality. Optional (and recommended) third parameter tolerance defines the allowable difference between unexpected and expected to be considered equal. Optional fourth parameter failure provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`Test case method to test the Cust_SalarySet and Cust_SalaryGet methods
`@Setup
START TRANSACTION
Unit_Mock_Setup
C_LONGINT($customerId)
C_REAL($wage;$salary;$expectedSalary)
$wage:=10 `$/hour
$customerId:=Unit_Mock_RecordCreate(->[Customers])
`@Tests
If(Unit_BeginTest("cusotmer salary set to 100.00";Current method name))
     ` call our production method under test
     Cust_SalarySet($customerId;Round(2000*$wage;2))
     $expectedSalary:=20000
     $salary:=Cust_SalaryGet($customerId)
     Unit_NumericEqualAssert($expectedSalary; $salary; 0.01)
     `tolerance = 0.01, so $salary must be 19999.99 to 20000.01
End if
```

```
`@TearDown
CANCEL TRANSACTION
Unit_Mock_TearDown
```

Unit_NumericNotEqualAssert

Unit_NumericNotEqualAssert(expected; actual{; tolerance{; message}})

Parameter	Type		Description
unexpected	Numeric	⟶	Unexpected numeric value
actual	Numeric	⟶	Actual numeric value
tolerance	Numeric	⟶	Allowable difference between expected and actual
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two numerics and asserts *inequality*. Optional (and recommended) third parameter defines the allowable difference between unexpected and expected to be considered equal. Optional fourth parameter failure provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`Test case method to test the Cust_SalarySet and Cust_SalaryGet methods
`@Setup
START TRANSACTION
Unit_Mock_Setup
C_LONGINT($customerId)
C_REAL($wage;$salary;$expectedSalary)
$wage:=10 `$/hour
$customerId:=Unit_Mock_RecordCreate(->[Customers])
`@Tests
If(Unit_BeginTest("cusotmer salary set to 100.00";Current method name))
     ` call our production method under test
     Cust_SalarySet($customerId;Round(2000*$wage;2))
     $expectedSalary:=20000
     $salary:=Cust_SalaryGet($customerId)
     Unit_NumericEqualAssert($expectedSalary; $salary; 0.01)
     `tolerance = 0.01, so $salary must be 19999.99 to 20000.01
End if
```

```
`@TearDown
CANCEL TRANSACTION
Unit_Mock_TearDown
```

Unit_RecordEqualAssert

Unit_RecordEqualAssert(table; rec1; rec2{; ignoreFields{; message}})

Parameter	Type		Description
table	Pointer	→	Pointer to table that contains records to compare
rec1	Long	→	Record number of expected duplicate
rec2	Long	→	Record number of expected duplicate
IgnoreFields	Pointer	→	Pointer to an array of field pointers that will be ignored when comparing records
message	String	→	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares two records and asserts *inequality*. Optional fourth parameter defines fields that are allowed to be different while the records continue to be considered equal (e.g. ID fields). Optional fifth parameter failure provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

Unit_EqualAssert

Unit_EqualAssert(expected; actual{; message})

Parameter	Type		Description
expected	Pointer	⟶	Pointer to the expected value
actual	Pointer	⟶	Pointer to the actual value
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares pointers and asserts equality of referenced variables. Can be used with any data type. This method is in fact called by all typed equal-assertion methods. If possible, you should use a typed assertion method instead of this one as typed assertions may provide extended functionality. For example, Unit_NumericEqualAssert takes an additional parameter for specifying precision tolerance. Optional third parameter message provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`Test to make sure our production method TextArrayAppend works as expected
`@Setup
ARRAY TEXT($actualNames_at;0)
ARRAY TEXT($expectedNames_at;4)
`@Tests
If(Unit_BeginTest("appending to text array";Current method name))
     $expectedNames_at{1}:="Mark"
     $expectedNames_at{2}:="Mike"
     $expectedNames_at{3}:="Matt"
     $expectedNames_at{4}:="Mitch"
     ` make sure we are starting with an empty array
     ArrayClear(->$actualNames_at)
     TextArrayAppend(->$actualNames_at;"Mark";"Mike";"Matt";"Mitch")
     Unit_EqualAssert(->$expectedNames_at;->$actualNames_at)
End if
```

Unit_NotEqualAssert

Unit_NotEqualAssert(unexpected; actual{; message})

Parameter	Type		Description
unexpected	Pointer	⟶	Unexpected pointed-to value
actual	Pointer	⟶	Actual pointed to value
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Compares pointers and asserts inequality of referenced variables. Can be used with any data type. This method is in fact called by all typed not-equal-assertion methods. If possible, you should use a typed assertion method instead of this one as typed assertions may provide extended functionality. For example, Unit_NumericNotEqualAssert takes an additional parameter for specifying precision tolerance. Optional third parameter message provides a custom message that will be displayed in the UnitTester detail pane in case of assertion failure.

Example

```
`Test to make sure our production method TextArrayAppend works as expected
`@Setup
ARRAY TEXT($actualNames_at;0)
ARRAY TEXT($expectedNames_at;4)
`@Tests
If(Unit_BeginTest("appending to text array";Current method name))
     $expectedNames_at{1}:="Mark"
     $expectedNames_at{2}:="Mike"
     $expectedNames_at{3}:="Matt"
     $expectedNames_at{4}:="Mitch"
     ` make sure we are starting with an empty array
     ArrayClear(->$actualNames_at)
     TextArrayAppend(->$actualNames_at;"Mark";"Mike";"Matt";"Mitch")
     Unit_EqualAssert(->$expectedNames_at;->$actualNames_at)
End if
```

Unit_TrueAssert

Unit_TrueAssert(bool {; message})

Parameter	Type		Description
bool	Boolean	⟶	Value asserted to be true
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Asserts a boolean expression to be true. Optional second parameter message provides custom printout notes in case of assertion failure.

Example

```
`Test case method to test the CustomerIsGood and CustomerIsBad methods
`@Setup
START TRANSACTION
Unit_Mock_Setup
C_REAL($good;$bad)
C_LONGINT($goodCustomerId;$badCustomerId)
$good:=1000 ` a good customer has purchased $1000 or more
$bad:=100 ` a bad customer has purchased $100 or less
$goodCustomerId:=Unit_Mock_RecordCreate(
  ->[Customers];->[Customers]total_sales;->$good)
$badCustomerId:=Unit_Mock_RecordCreate(
  ->[Customers];->[Customers]total_sales;->$bad)
`@Tests
`First test will make sure a good customer gets the appropriate results
If(Unit_BeginTest("Good customer";Current method name))
     Unit_TrueAssert(CustomerIsGood($goodCustomerId);"good")
     Unit_FalseAssert(CustomerIsBad($goodCustomerId);"bad")
End if
`Second test will make sure a bad customer gets the appropriate results
If(Unit_BeginTest("Bad customer";Current method name))
     Unit_FalseAssert(CustomerIsGood($badCustomerId);"good")
```

```
    Unit_TrueAssert(CustomerIsBad($badCustomerId);"bad")
End if
`@TearDown
CANCEL TRANSACTION
Unit_Mock_TearDown
```

Unit_FalseAssert

Unit_FalseAssert(bool {; message})

Parameter	Type		Description
bool	Boolean	⟶	Value asserted to be false
message	String	⟶	Additional message to print to the UnitTester detail pane when the assertion fails.

Description
Asserts a boolean expression to be false. Optional second parameter message provides custom printout notes in case of assertion failure.

Example

```
`Test case method to test the CustomerIsGood and CustomerIsBad methods
`@Setup
START TRANSACTION
Unit_Mock_Setup
C_REAL($good;$bad)
C_LONGINT($goodCustomerId;$badCustomerId)
$good:=1000 ` a good customer has purchased $1000 or more
$bad:=100 ` a bad customer has purchased $100 or less
$goodCustomerId:=Unit_Mock_RecordCreate(
  ->[Customers];->[Customers]total_sales;->$good)
$badCustomerId:=Unit_Mock_RecordCreate(
  ->[Customers];->[Customers]total_sales;->$bad)
`@Tests
`First test will make sure a good customer gets the appropriate results
If(Unit_BeginTest("Good customer";Current method name))
     Unit_TrueAssert(CustomerIsGood($goodCustomerId);"good")
     Unit_FalseAssert(CustomerIsBad($goodCustomerId);"bad")
End if
`Second test will make sure a bad customer gets the appropriate results
If(Unit_BeginTest("Bad customer";Current method name))
     Unit_FalseAssert(CustomerIsGood($badCustomerId);"good")
```

```
    Unit_TrueAssert(CustomerIsBad($badCustomerId);"bad")
End if
`@TearDown
CANCEL TRANSACTION
Unit_Mock_TearDown
```

Other Methods

The methods in this section include utility and application support methods. In future releases, I may separate out a “Utilities Methods” section, which would include the ***Unit_Mock...*** methods that you will find in this section.

What are “Mock” Records?

In unit testing lingo, the term “mock” refers to objects that are instantiated purely for testing purposes. A mock record, therefore, would be a record that is created for testing purposes only. If we are creating records for tests, we certainly don’t want those records to remain once the tests are finished. In this version of UnitTester, the way to accomplish creating mock records without risking corruption of the database (corruption meaning adding records that shouldn’t be there), we use transactions and the ***Unit_Mock...*** routines. The basic structure is as follows:

```
`@Setup
START TRANSACTION
Unit_Mock_Setup
`create some temporary “mock” records for use in tests
C_LONGINT($id)
C_TEXT($value)
$value:=”mock value”
$id:=Unit_Mock_RecordCreate(->[MyTable];->[MyTable]myField;->$value)

`@Tests
`write your tests here. You may use the [MyTable] record referenced by $id
...

`@TearDown
CANCEL TRANSACTION
Unit_Mock_TearDown `restore sequence numbers
```

Unit_BeginTest

Unit_BeginTest(description; Current method name) ⟶ True

Parameter	Type		Description
description	String	⟶	Description of the test
method name	String	⟶	The 4D command **Current method name**
return	Boolean	⟵	Always returns True

Description
This method defines the start of a unit test. It always returns True so that it may be used as a block element in the method editor. Pass the name of the test in the first parameter. You must pass the 4D function "Current method name" as the second parameter in order for UnitTester to be able to properly parse the test. Hint: a type-ahead macro is provided that makes test creation simple. Type "_unit_new_test" and hit enter or tab to generate a new test block.

Example

```
If(Unit_BeginTest("customer in good standing";Current method name))
    Unit_TrueAssert(CustomerIsGood)
End if
```

Unit_CustomLogTextSet

Unit_CustomLogTextSet(text; Current method name)

Parameter	Type		Description
text	String	→	Custom information to be output in the log file
method name	String	→	The 4D command **Current method name**

Description
This method provides a hook for adding your own custom text to the log file output for the test case method in which it is called. You must pass the 4D function "Current method name" as the second parameter. Hint: a type-ahead macro is provided that makes test creation simple. Type "_unit_custom_log_text" and hit enter or tab to generate code.

Example

```
`@Setup
Unit_CustomLogTextSet("Case ID: 123456";Current method name)
```

Unit_Mock_RecordCreate

Unit_Mock_RecordCreate(tablePtr{; fieldPtr; valPtr{;...}}) ⟶ Integer

Parameter	Type		Description
tablePtr	Pointer	⟶	Pointer to the table for which to create a record
fieldPtr	Pointer	⟶	Pointer to a field for which to set a value with the following value pointer parameter
valuePtr	Pointer	⟶	Pointer to a value to which to set the preceding field pointer
return	Integer	⟵	Newly created record ID generated by calling **Sequence number**

Description
This method is for use within test case methods that require temporary records to be created. Pass the tablePtr as the first parameter, and optional pairs of fieldPtr;valuePtr thereafter (no limit). Use this method to ensure table sequence numbers are returned to the values prior to running the test case method. Note: if another process increments the tablePtr's sequence number before the test case method calls Unit_Mock_TearDown, the sequence number will not be reset as a safety precaution.

Example

```
`methods under test: MyTable_SetDescription, MyTable_GetDescription
`@Setup
If(True)
    START TRANSACTION
     Unit_Mock_Setup `intend to create records with Unit_Mock_RecordCreate
     C_LONGINT($id_l)
End if

'@Tests
If(Unit_BeginTest("good customer";Current method name))
     $id_l:=Unit_Mock_RecordCreate(->[MyTable]) `sequence number incremented
     MyTable_SetDescription($id_l;$expected_t)
     Unit_TextEqualAssert($expected_t;MyTable_GetDescription)
End if
```

```
`@Tear Down
If(True)
     Unit_Mock_TearDown `sequence numbers reset to previous values
End if
```

Unit_Mock_Setup

Unit_Mock_Setup

Description
Call this method in the Setup section of a test case method. Its purpose is to initialize UnitTester variables to keep track of records created using the Unit_Mock_RecordCreate method. In the Tear Down section, you should call the Unit_Mock_TearDown method to restore sequence numbers. See Unit_Mock_RecordCreate and Unit_Mock_TearDown.

Example

```
`methods under test: MyTable_SetDescription, MyTable_GetDescription
`@Setup
If(True)
     START TRANSACTION
     Unit_Mock_Setup `intend to create records with Unit_Mock_RecordCreate
     C_LONGINT($id_l)
End if

'@Tests
If(Unit_BeginTest("good customer";Current method name))
     $id_l:=Unit_Mock_RecordCreate(->[MyTable]) `sequence number incremented
     MyTable_SetDescription($id_l;$expected_t)
     Unit_TextEqualAssert($expected_t;MyTable_GetDescription)
End if

`@Tear Down
If(True)
     Unit_Mock_TearDown `sequence numbers reset to previous values
End if
```

Unit_Mock_TearDown

Unit_Mock_TearDown

Description
Call this method in the @Tear Down section of a test case method after having called Unit_Mock_Setup in the @Setup section. This method returns tables' sequence numbers to the values previous to the Unit_Mock_Setup call. Only those tables that have had records added by Unit_Mock_RecordCreate are affected. Note: if another process increments the tablePtr's sequence number before the test case method calls Unit_Mock_TearDown, the sequence number will not be reset as a safety precaution.

Example

```
`methods under test: MyTable_SetDescription, MyTable_GetDescription
`@Setup
If(True)
     START TRANSACTION
     Unit_Mock_Setup `intend to create records with Unit_Mock_RecordCreate
     C_LONGINT($id_l)
End if

'@Tests
If(Unit_BeginTest("good customer";Current method name))
     $id_l:=Unit_Mock_RecordCreate(->[MyTable]) `sequence number incremented
     MyTable_SetDescription($id_l;$expected_t)
     Unit_TextEqualAssert($expected_t;MyTable_GetDescription)
End if

`@Tear Down
If(True)
     Unit_Mock_TearDown `sequence numbers reset to previous values
End if
```

Unit__LaunchUnitTester

Unit__LaunchUnitTester

Description
Execute Unit__LaunchUnitTester to open the UnitTester GUI. This method is set to be "visible" which allows it to be executed from the "Run Method" dialog. Note: a macro is provided to generate the code necessary to safely call this method whether the component is installed or not. See the "Macros" section. Methods with double-underscore (Unit__...) are not to be used within a test case method. You may execute this method from the Execute Method dialog.

Example

```
If (Not(Is compiled mode(*)))
     ARRAY TEXT($components_at;0)
     COMPONENT LIST($components_at)
     If (Find in array($components_at;"UnitTester")>0)
               EXECUTE FORMULA("Unit__LaunchUnitTester")
     End if
End if
```

Unit__MethodCreatorOpen

Unit__MethodCreatorOpen

Description
Call this method to open UnitTester's Method Creator tool separately from the UnitTester GUI. Methods with double-underscore (Unit__...) are not to be used within a test case method. You may execute this method from the Execute Method dialog.

Example

```
If (Not(Is compiled mode(*)))
     ARRAY TEXT($components_at;0)
     COMPONENT LIST($components_at)
     If (Find in array($components_at;"UnitTester")>0)
               EXECUTE FORMULA("Unit__MethodCreatorOpen")
     End if
End if
```

Unit__ServerStartup

Unit__ServerStartup

Description
Call this method in the On Server Startup database method. Provides synchronization between multiple clients of unit tests. Note: a macro is provided to generate the code necessary to safely call this method whether the component is installed or not. See the "Macros" section for more information. Methods with double-underscore (Unit__...) are not to be used within a test case method. You may execute this method from the Execute Method dialog.

Example

```
`In the On server startup database method:
If (Not(Is compiled mode(*)))
     ARRAY TEXT($components_at;0)
     COMPONENT LIST($components_at)
     If (Find in array($components_at;"UnitTester")>0)
               EXECUTE FORMULA("Unit__ServerStartup")
     End if
End if
```

Unit__ServerShutdown

Unit__ServerShutdown

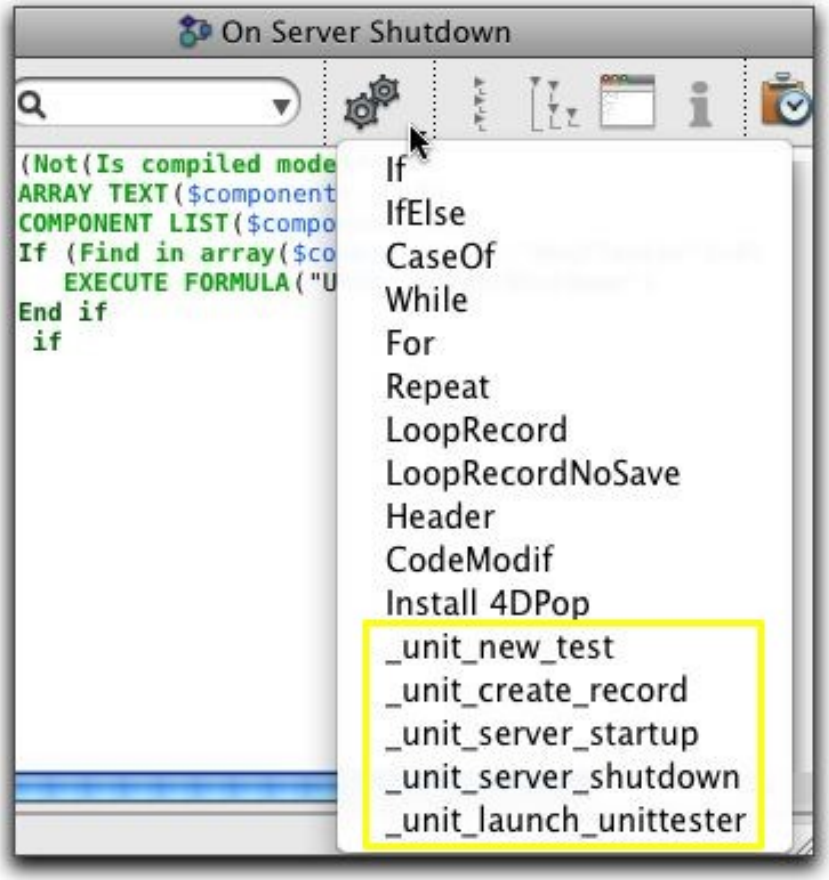
Description
Call this method in the On Server Shutdown database method. Provides synchronization between multiple clients of unit tests. Note: a macro is provided to generate the code necessary to safely call this method whether the component is installed or not. See the "Macros" section for more information. Methods with double-underscore (Unit__...) are not to be used within a test case method. You may execute this method from the Execute Method dialog.

Example

```
`In the On server shutdown database method:
If (Not(Is compiled mode(*)))
     ARRAY TEXT($components_at;0)
     COMPONENT LIST($components_at)
     If (Find in array($components_at;"UnitTester")>0)
               EXECUTE FORMULA("Unit__ServerShutdown")
     End if
End if
```

Macros

UnitTester ships with the following macros to facilitate its use.



_unit_new_test

Description
Use this macro to quickly create a new test block in a test case method. This macro is setup for type-ahead, so you can type “_unit_new” followed by a tab and have the following code generated for you:

Generates

```
If(Unit_BeginTest(“”;Current method name))


End if
```

_unit_custom_log_text

Description
Use this macro to quickly generate the code to set custom text to the log file for a test case method. This macro is setup for type-ahead, so you can type “_unit_cust” followed by a tab and have the following code generated for you:

Generates

```
Unit_CustomLogTextSet(“”;Current method name)
```

_unit_create_record

Description
Use this macro when you have a table selected in the method editor. Automatically generates code that creates a record for the selected table and prepares each field for an assignment. This is useful in test case methods when you need to create temporary records (mock records).

Example Output:

```
CREATE RECORD([Projects])
[Projects]p_Id:=
[Projects]Name:=
[Projects]Introduction:=
[Projects]PDFPath:=
SAVE RECORD([Projects])
```

_unit_launch_unittester

Description
Provides a safe way to launch UnitTester in your production code. The idea is that when you deploy your application, you can safely exclude the UnitTester component and plugin. This macro is accessible via the Macros menu in the method editor.

Generates

```
If (Not(Is compiled mode(*)))
     ARRAY TEXT($components_at;0)
     COMPONENT LIST($components_at)
     If (Find in array($components_at;"UnitTester")>0)
               EXECUTE FORMULA("Unit__LaunchUnitTester")
     End if
End if
```

_unit_server_startup

Description
UnitTester provides two macros for making safe calls to Unit__ServerStartup and Unit__ServerShutdown. The idea is that when you deploy your application, you can safely exclude the UnitTester component and plugin. Both macros are accessible via the Macros menu in the method editor.

Generates

```
If (Not(Is compiled mode(*)))
     ARRAY TEXT($components_at;0)
     COMPONENT LIST($components_at)
     If (Find in array($components_at;"UnitTester")>0)
               EXECUTE FORMULA("Unit__ServerStartup")
     End if
End if
```

_unit_server_shutdown

Description
UnitTester provides two macros for making safe calls to Unit__ServerStartup and Unit__ServerShutdown. The idea is that when you deploy your application, you can safely exclude the UnitTester component and plugin. Both macros are accessible via the Macros menu in the method editor.

Generates

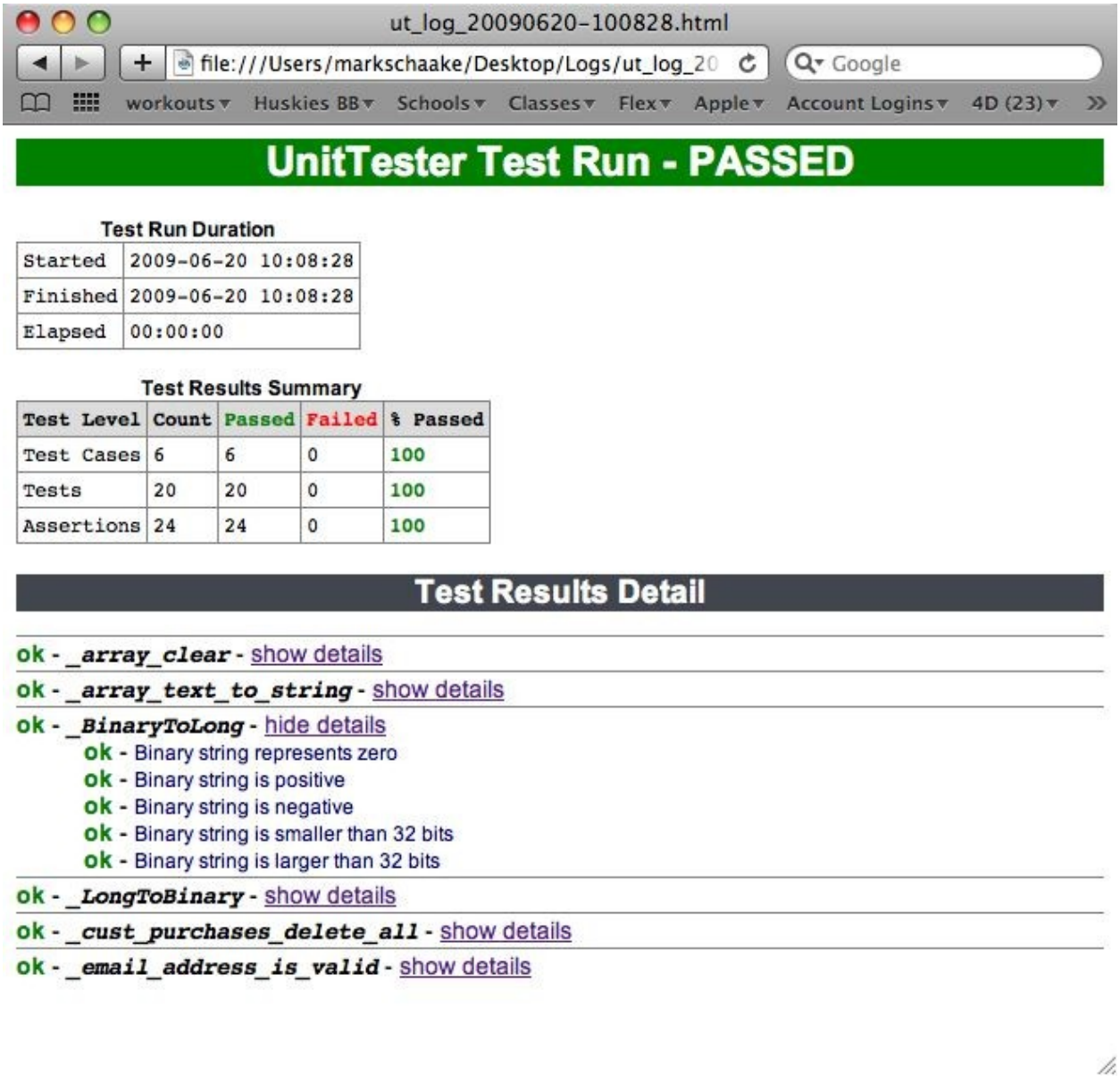
```
If (Not(Is compiled mode(*)))
     ARRAY TEXT($components_at;0)
     COMPONENT LIST($components_at)
     If (Find in array($components_at;"UnitTester")>0)
               EXECUTE FORMULA("Unit__ServerShutdown")
     End if
End if
```

Log Files

UnitTester generates two files each time all tests run if the preference to do so is set (see Preferences section). One file is an XML document, the other is a styled HTML document.

HTML Log File

Following is a screenshot of a HTML log file generated by UnitTester.



XML Log File

For many, the HTML document will likely be sufficient. However, you may need to parse the XML document from another application – your company's system, for example. Following is the XML structure of UnitTester log files (you can also open a UnitTester XML log file to view its structure).

```
<test_run pass="{true/false}">
  <summaries start="{timestamp}" finish="{timestamp}" elapsed="{HH:MM:SS}">
    <summary name="{case/test/assertion}" count="{#}" failed="{#}" passed="{#}"
percent_passed="{#}"/>
    ...
    <summary.../>
  </summaries>
  <test_cases>
      <test_case name="{name}" pass="{true/false}">
      <user_data>{text from call to Unit_TestCaseCustomDataSet}</user_data>
      <tests>
        <test name="{name}" pass="{true/false}">
          <assertions>
            <assertion>{description}</assertion>
            ...
            <assertion>...</assertion>
          </assertions>
        </test>
        ...
        <test...>...</test>
      </tests>
    </test_case>
    ...
    <test_case...>...</test_case>
  </test_cases>
</test_run>
```

Closing Remarks

UnitTester has been under development since November 2008. There have been many revisions, and I expect there will be many more. However, the core of UnitTester – the assertion API – has been stable since December 2008, and I suspect will not see many changes going forward.

I am committed to keeping UnitTester alive and well for the long run. If you have any feature requests or bugs to report, please do so via the UnitTester interface by clicking the Schaake Consulting button in the upper-right hand corner and selecting the appropriate option. Other responsibilities permitting, I expect to be responsive to requests.

By the way, yes, UnitTester itself is unit tested and developed using TDD.

Happy unit testing!

Mark Schaake
Schaake Consulting, LLC