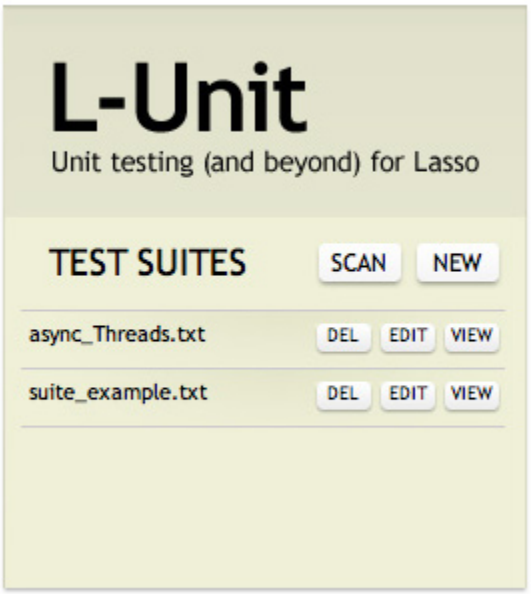
What is L-Unit

L-Unit is a unit testing frame work for Lasso. It provides a set of tools that enables automated testing of any Lasso Type or Tag.



What is Unit Testing

Unit testing is a method to test if the smallest components of a program are functioning correctly. These components a referred to as "Units", in our world of Lasso a Unit would generally refer to a Types->Member Tag or more rarely a stand alone Custom Tag. The goal of unit testing is to verify that each small unit of code preforms as expected when given a known set of parameters. By automatically verifying the results of each unit test we can ensure that the unit is functioning as it should.

Unit testing is designed to test the smallest units of code possible. This is in contrast to integration testing which tests higher levels of an application's behaviour where multiple code layers or systems are triggered to interact, and the results of the whole are verified.

Benefits

Whack-a-Mole bug hunting. We've all played it. You're writing code, checking things by running your app, and all is going well. Then, all of a sudden, stuff starts to break. Every fix you add breaks something else. Pretty soon you're afraid to change anything. Enter unit testing.

Imagine a shopping basket object with an ->addItem method (member tag in Lasso-speak). If a user of your website clicks an Add Item button twice, do you want two line items for the same thing added to the basket? Probably not. You might want the second add to be ignored, or maybe you want to first see if the quantity was changed and modify the quantity field of the item in the basket. So, you write some code to do that. How do you test it? You probably launch your app, and pretend to be a user and click the add item button twice, then go inspect the basket to see what happens. Fair enough. Now, let's say it's three months later, and you modify the basket code for some reason. How do you know you didn't break this double-add handling? Unless you test everything all over again, you don't.

This presents us with a couple of issues. First, we could end up spending a lot of time doing this manual testing. Now, eventually such testing has to be done, but the problem with testing like this is that it is not reusable. The time spent role play testing today is useless tomorrow because to verify that recent code additions haven't broken something, that manual testing has to be done all over again. It would be advantageous if something could do the testing grunt work for us. Automatically.

Second, each time we test how can we be sure that the exact same test was performed? Does it matter that you logged into the user preference settings before clicking Add Item yesterday, but today you didn't log in first? So, we also want something that will perform testing identically every time. For scenarios like this, and for many other reasons, Unit Testing was created. And, to help Lasso developers keep up with the Joneses, L-Unit exists.

Unit Testing Terminology

It's been mentioned already, but it is worth repeating: the goal of unit testing is to verify that a certain small unit of code given a known starting state, and given a known set of inputs or influences, will generate a known ending state. That chunk of code and its verifications form a unit test. Unit testing is designed to test the smallest units of code possible. This is in contrast to integration testing which tests higher levels of an application's behaviour where multiple code layers or systems are triggered to interact, and the results of the whole are verified.

So, as we cover the discussion of testing, be sure to keep in mind that what we're testing are discrete functions of the software one at a time. This does not replace fully interactive user testing, but it does come before that kind of testing gets done.

Automated unit testing can be looked at in two levels. First is the test writing, and second is the automated running of the tests. If we break down the level of detail involved in automated unit testing, we come across the following terms:

- **task** — a task is the smallest element of a test. It is a piece of code that evaluates to a true or false result. It verifies that one specific piece of data
- **test method** — a test method is a method (i.e. a method of a class, or a member tag of a custom type in Lasso-speak) which tests a correlating method of an application class. A test method is comprised of many tasks which help to verify that the method as a whole appears to be fully functional as expected and handles invalid inputs gracefully as well.
- **test case** — a test case (or test class) is a collection of test methods within a single class which together ensure that an entire application class appears to function as desired. A test case is not always an exact mirror of an application class as there are extra methods needed for test automation, and there may be some class methods that are too simple to bother testing.
- **test suite** — a test suite is a collection (usually just a list) of test cases which should be run together simply because they are related in some manner. "Run together" does not mean integrated together, it just means each test case is run in sequence as a group. So, perhaps the test cases that cover the classes involved in a purchase order's definition are tested together. Whereas classes for a shopping basket would likely kept in a separate test suite. It's really just a topical organisation.
- **test fixture** — or test harness is code that in itself is neither a test nor an application class, but is a utility to help perform tests. In L-Unit, there are some standard custom types that get used as the parents for the test code that the developer writes. These could be considered fixtures. Additionally, you might need some code that creates a test database (and destroys it when the tests are done). Rather that rewrite that in each test case, you would create a separate piece of code that each test case can use. Such a code piece would be a test fixture.
- **test application** — this is the appliance that helps organise test suites, automates the running of the tests, and provides output indicating which tests passed, which ones failed, and some general metrics about the tests.

L-Unit is a test application, as well as a protocol for developing test cases in such a manner that allows the test application to run those test cases.

L-Unit Testing and OOP

It should be said before we go much further than unit testing is very dependent on object oriented programming. For those of you not yet developing using the OOP paradigm, you might find the concepts of TDD and unit testing a compelling reason to get started.

To get the most benefit out of L-Unit (and Lasso) you really want to be working with custom types. L-Unit can test custom tags just as well (those that use a return statement), but unit testing is just about useless for testing includes. The lack of input and output constraints on an include make it very difficult to test, which in itself is a telling statement.

Test Driven Development

Myth: you only need to unit test.

Unit testing is a component of a larger philosophy of test driven development (TDD). I'm certainly no expert in the TDD culture (just getting started myself), so it is probably best that you research TDD on the web and learn what it's about and why it's an important development paradigm (and even find some anti-TDD articles as well), but I'll try to summarize it here.

If we look at the term test driven development, we can assume that it means development driven by testing. This implies that testing has to come before development. This very notion is quite the reverse of traditional development practices where functional code is written, then it is tested. With TDD, or more accurately, at this level of thought, test first development (TFD) the tests are actually written before the functional code. Now, that may sound ridiculous, but it's not without a parallel in even in manufacturing. Many products are manufactured by first constructing a test, or a gauge to which the product will be measured. Why? Because maybe the details of the product don't matter, but as along as it passes a few tests, we're happy. So, if that's the case, then we would build the tests first.

If we want a container to hold 1 gallon of water, but we didn't really care what the container was shaped like, then we could build a test fixture that establishes our 1 gallon volume. Then we could build a container, fill it with water, and empty it into our test container. It we run short, we could modify the container to be a little bigger. We could keep making various containers, but as long as they hold 1 gallon, we're happy.

Code can be the same way. We know there's nearly an infinite number of ways to accomplish the same task in programming. In many respects the details don't matter. Does it do the job we need? If it will do the job, we're happy. Like our water container, there are many aesthetics to consider, but if any one of them is important enough, then we can add a new test.

With code, if we have a certain task to accomplish, a data structure to manipulate, a calculation to perform, then what matters is the input and output. In effect, every routine can be viewed as a black box. As far as the functionality of the program is concerned the details don't matter, but the I/O does matter. So, if that's what matters, then we can write a test for that. And, the fact is we generally know what matters before we know how we're going to do it. So, if we know what we want from an I/O standpoint, we can actually author a test skeleton that would verify that. If that test were run, it would fail, of course, because there is no functional code yet to run it against. But, now, when we write the functional code, we have a test already to see if it works.

In a way, writing tests is like writing a specification. If a test is written which provides inputs X and Y and verifies that it gets the desired answer Z, then that is a very clear specification. If it comes up in conversation that the code should also deal with inputs X and Y and W, and still produce Z, then, a new test can be added to verify that scenario. However, a test is better than just a specification because it not only documents the requirement, it also provides a means to prove the code meets the requirement. And, as we established earlier, that test is available to reuse over and over to prove that new code hasn't broken the requirements we've already programmed to.

Therefore, writing a test which at first glance might seem to be a wasteful task, proves to be a productive system for creating and sustaining application functionality and reliability.

As Scott Ambler's article (see sidebar) points out, as we write a simple test, then write enough code to pass that test, then add another test requirement, and write more code to pass that new test, and keep doing that, then we have morphed into test driven development. It is development driven by the satisfying of tests.

By the way, having said that tests are better than specifications does not imply there is no need for specifications. A unit test is not a 100% substitute for written specs. First, unit testing can't define many aspects of an application's behavior or interface. Second, designing a test is a response to an expressed requirement for application functionality. So, clearly, some requirement specifications have to exist to start with. However, at some point in the defining of an application, it is not a value added exercise to spell out implementation details in advance that may turn out to be irrelevant. This is where unit tests as a form of specification can fill some gaps. Where the specs provided to the original developer can't (and shouldn't) define implementation details in advance, unit tests can provide those details for the development team to build to, and a maintenance team to test to.

TDD References

Kent Beck is the author of the first unit testing framework SUnit written for SmallTalk. He recently wrote a book on TDD (using Java as the code examples). This is probably the classic reference for test driven development and unit testing:
http://www.bookpool.com/sm/0321146530

This article by Scott Ambler provides a nice overview of TDD and the essential philosophies and practices from a generic discussion view:
http://www.agiledata.org/essays/tdd.html

The material on testing in the classic Pragmatic Programmer is also quite good:
http://www.bookpool.com/sm/020161622X

The SUnit project home has quite a few good links to unit testing references from that site.
http://sunit.sourceforge.net/

How do we know we're defining the right tests to start with? Well, that has more to do with communicating what the purpose and desires are for the application to the developers, and that's a topic for another article. Hint: look at Agile development (like Extreme Programming or Scrum) and the concept of stories.

How do we know that the tests are adding up to the actual functionality we need from the application? That's where integration testing and traditional user testing participate in the total picture of TDD. TDD is not just unit testing.

Enough theory. Let's get busy.

Assertions

Assertions are the molecules of test driven development.
They provide the basic building blocks of tests.

Before we dig into how to write tests and use the L-Unit test application, we have to cover assertions. An assertion is an expression that evaluates code as having the result of either true or false. Generically, we could call this an assertion:

```
$price->type == 'decimal'
```

If we look at that expression as a statement of fact, and not as a conditional, then we assert that the statement has a result of either true or false. That statement is therefore referred to as an assertion.

A programming language is said to support assertions if it has built-in support for evaluating an expression against an expected result. So, rather than having to write code like this:

```
if: $price->type == 'decimal';
   $result = true;
else;
   $result = false;
/if;
```

We would prefer something more efficient, and some thing that generates the boolean result on its own. Something more like this:

```
assert_isTrue: $price->type == 'decimal';
```

If we imagined this command assert_isTrue, it would evaluate what comes after it to be an expression of code to be evaluated, and we would expect the result of that code to be equal to what is asserted which in this case is *true*. Another example could be:

```
assert_isDecimal: $price->type;
```

With that case we would not need to write code for the evaluation of the data type, that is explicitly identified in the assertion tag. In fact, we could further assume that the assertion tag knows that we want a data type of decimal, so the code could simply be:

```
assert_isTypeDecimal: $price;
```

Given this functionality of an assertion, an assertions library can be built to have a number of assertions with the expected result defined as part of the assertion command, and some some extra ones for general purpose flexibility.

Assertions and Lasso

Alas, Lasso does not have assertions, so L-Unit includes one in the form of a custom data type called unit_assertions.

The assertions library could have been implemented as a collection of custom tags (one tag per assertion), but considering the real purpose of having assertions is to facilitate testing, it turned out to be more efficient to write the library as a single custom type. Assertions aren't really a data type of course, so it's awkward to use Lasso's chosen term for custom types. Throughout the rest of this document, I'll be using the more conventional OOP terms of *class* (custom type) and *method* (member tag). *Instance variable* works pretty well for most languages, and *custom tag* is a structure fairly unique to Lasso that we don't need to translate into an OOP equivalent because there really isn't one (although I think PHP and ColdFusion may use them similar to how Lasso does).

L-Unit Assertions Syntax

The unit_assertion class was created to best serve as a superclass (a parent) for the unit_testCase class that we'll look at shortly, but it can be used independently as well. In this mode, the syntax looks a bit awkward, but when we get to writing real test code, you'll see how it works out better. To minimize the awkwardness, we're going to use a simple variable of x for the object. So, the syntax for the unit_assertion class looks like this:

```
var:'x' = unit_assertions;
$x->(assert_isTrue:
-expr = "$price->type == 'decimal'");
```

You'll notice right away, that the expression is flagged with a typical Lasso parameter name and that the expression is quoted. What's going to happen inside the assert_isTrue method is that the code passed via the -expr parameter is going to be evaluated using the [process] tag. In some assertions, the expressions will actually be modified a little by adding some code. So, in order for the method code to do all that, the expression code has to be submitted as a string which can be used by [process].

It is important to notice also that the code must be submitted as LassoScript syntax. The assertion method will wrap the code in brackets and perhaps make other modifications, so the format of the code must be LassoScript. Whether you use colon or paren syntax should not matter.

Following the hypothetical examples, another assertion example is:

```
$x->(assert_isTypeDecimal: -expr="$price");
```

Some fancier examples include:

```
$x->(assert_isTypeArray:
-expr="$recordsList");

$x->(assert_isIntegerMax:
-expr = "$movieCredits",
-max = 3);

$x->(assert_isArraySizeInRange:
-expr = "$passengerList",
-min = 1,
-max = 7);

$x->(assert_isMapContaining:
-expr = "$purchaseOrder",
-contains = "shipDate");
```

There are numerous assert_is methods available, and they're likely to be expanded. Some languages have explicit assertions like this, and some have only a few generic assertion commands. The advantage of the explicit commands is the readability. The disadvantage is that there can be too many to remember. So, the
L-Unit assertions plays both ways. There are some generic ones like assert_isTrue, assert_isFalse, assert_isSizeOf, assert_isTypeOf where you can write out the longer version of the code, but can remember fewer assertion methods.

The appendix defines the available assertions as of the version of this document.

Assertion Results

I mentioned earlier that an assertion should automatically return true or false. Here again, I've taken advantage of the case that unit_assertions was developed to serve as a superclass for the unit_testCase class. Each assertion stores it's result internally (the pedants out there will likely be quick to say "aha! so unit_assertions *is* a data type"). This is done so that results can be more easily processed and provide more information in their context of the unit testing automation application.

Each assertion stores a map which consists of the assertion name (just a regurgitation of the assertion method name), the result which will be true, false, or null, a message which will be either "passed" or a description of why the assertion failed, and finally the time in milliseconds required to process the assertion.

Time isn't too important for an assertion as most will be even less than 1 millisecond, but the data structure is consistent with doing custom tests inside the unit_testCase class where we may need to find out which tests are getting too time consuming. We'll cover more about test times later.
So, a result for a failed _isIntegerMax assertion might look like this:

```
name = assert_isIntegerMax
result = false
message = $movieCredits == 4
which is not <= 3
time = 0
```

Results in this format provide useful data for displaying the results of automated tests.

How Assertions Relate to Testing

We're not ready to dive into writing test code just yet, but before a few of the finer points of writing assertions can be covered, you should understand the basic relationship of how assertions are used as part of testing.

Assertions are used to verify that code executed as expected. Therefore, a test is comprised of some application code which does some things, and then some assertions that check that the app code generated the expected results.

For example, here's a no-brainer:

```
var:'array' = array;
$array->(insert:'red');

$x->(assert_isArraySizeOf:
-expr = "$array",
-size = 1);
```

After inserting an element into an empty array, did the array have a size of 1? OK, so that one is a bit boring, but the point is simply to show that first some code gets executed, then assertions are used to verify results.

Here's some pseudo code to show a more complex example. Let's say you have a shopping basket class, and it has an ->addItem method. Part of that addItem's job is to prevent adding duplicate items into the basket, but when a user tries to do that, the quantity is checked to see if it has been changed. So, to test this, we would create a shopping basket item. Then we'd manually insert some data into the basket to prep it. Then we'd add a duplicate item with a different quantity. Finally, we'd use assertions to see what happened.

```
// test setup
var:'basket' = shoppingBasket;

$basket->(addItem:
              -sku = 'abc123',
              -desc = 'yadda',
              -qty = 1);

$basket->(addItem:
               -sku = 'xyz456',
               -desc = 'blabla',
               -qty = 1);

// test action
$basket->(addItem:
               -sku = 'abc123',
               -desc = 'yadda',
               -qty = 3);
// verify
assert_isIntegerEqualTo:
         -expr = "$basket->size",
         -value = 2;

assert_isIntegerEqualTo:
         -expr = "$basket->(getQtyForSKU: 'abc123')",
         -value = 3;
```

These two assertions made sure that we did not increase the number of basket items, and that we did in fact replace the quantity value. So, first there may be some setup code, then some actual scenario code, then the assertions. This brings us to a few finer points about writing assertions.

Writing Good Assertions

The first point to understand about assertions is that the expression you submit along with the assertion method used must evaluate to a true or false result.

The next point to understand is that the expressions you test should be as simple as possible, and test only a single critical data point of the application. An assertion should not be written in such a way that it requires several things to all work as expected in order to succeed. If it fails, you won't know where the failure is.

Assertions, and unit testing in general is supposed to help you very rapidly identify the point of failure. If an assertion expression includes 5 conditional evaluations, you won't know which one failed. Write 5 individual assertions instead.

Don't do this:

```
assert_isTrue:
-expr = "$array->size <= 3
&& $map->find:'color'=='red'
&& $starsAreAligned == true;"
```

If that assertion fails, you don't know which conditional caused the failure. If you don't know that, you don't know what part of the code you were testing didn't behave as expected.

Assert Like This:

```
assert_isArraySizeMax:
        -expr = "$array",
        -max = 3;

assert_isTrue:
        -expr = "$map->find:'color'=='red'";

assert_isTrue:
        -expr = "$starsAreAligned"
```

If you're analying what all this means at this stage, you might be wondering what some reasonable boundaries are. We could get real picky about what testing a "single data point" means. If we go back to the shopping basket example, the two assertions assumed we still had a `$basket` object (i.e. the `addItem` code didn't kill the basket), and in the second assertion we assumed that `->getQtyForSKU` would even return a result to start with. So, we could have started with assertions that verified we did have a basket, and that indeed we could extract a `qty` value.

Where you draw the line as to how many assumptions you make impacts a balancing act of how long it takes to run tests vs. complete test coverage. We'll talk more about that later, but I thought it worth recognizing that such boundary questions will arise.

Test Cases

Test cases are the real meat of unit testing. This is where you will spend the vast majority of your test development time.

A test case as it has been modeled in the L-Unit framework is a Lasso custom type (a class). It is intended to mirror an application class. It's not an exact mirror, but the methods of the application class which are to be tested will be represented by correlating methods in the test case class. Continuing to use our shopping basket example, if there is a method ->addItem in the application class, then there will be a method named ->test_addItem in the test case class.

So, just to get a flavor of the overall look of a test case, let's draw some parallels of an application class and the test case class that will be used to test it.

```
// application class structure

define_type: 'shoppingBasket';

define_tag:'addItem';
/define_tag;

define_tag:'removeItem';
/define_tag;

define_tag:'size';
/define_tag;

/define_type;

// test case class structure

define_type: 'test_shoppingBasket','unit_testCase';

define_tag:'test_addItem';
/define_tag;

define_tag:'test_removeItem';
/define_tag;

define_tag:'setup';
/define_tag;

define_tag:'teardown';
/define_tag;

/define_type;
```

To answer the first question, yes, the test_ prefix is required, and yes the test methods must be named exactly as the application class methods with the test_ prefix. This is a critical part of the automation capabilities as it allows L-Unit to automatically find the test methods to run which will test the corresponding application method.

You'll notice there's no test_size method in the test case. That's to point out that application methods which are really simple don't necessarily have to have a test. In this case the ->size method might be as simple as:

```
return: (self->'basketItems')->size;
```

where basketItems is simply an array of maps. It's not likely that such a simple line would ever fail, so it might be deemed unnecessary to build a test.

So, before we go into more details, this first part was just to establish that a test case is a class, and it will be comprised of methods that mirror the application class methods that will be tested.

The unit_testCase Class

The L-Unit framework includes a class called unit_testCase which is used only as a parent to the test case classes that the developer writes. The developer test cases all inherit from unit_testCase which includes the following important features:

- ->'classPath' — an instance var that defines a standardized (though optional) path in the L-Unit application folder where application class files are stored.
- ->setup — a stub method which has no code, but is required to be provided by the developer's test case to run any code which is needed to prepare conditions (create objects, databases, etc) for running the test methods.
- ->teardown — a stub method which has no code, but is required to be provided by the developer's test case to run code which reverses or destroys what was created by setup.
- ->run — a method which gets called by the automation application to trigger the running of the tests. This method uses introspection (the ability of a class to determine what it's properties are on the fly) to find all method in the test case that begin with test_ and then invoke them. The developer doesn't have to do anything regarding this tag, and it does not get replaced by the test case code.
- ->storeResult — a method which stores the results of individual test tasks (assertions or custom tasks). These results are automatically consolidated with other test cases by the L-Unit application to calculate metrics on the tests that were run, and also to display detailed results of each test task.

There are a few other tags and variables in the unit_testCase class, but they're all for internal use only. If L-Unit were exclusive to Lasso 8.5, they could be made private.

For the most part the developer really doesn't need to know too much about the unit_testCase class, and most of the working instructions apply to the developer's own test case class code.

Starting a Test Case Class

When writing a test case, the define_type statement needs to indicate the use of the unit_testCase type as a parent like this:

```
define_type:'test_whatever', 'unit_testCase';
```

which is a typical Lasso inheritance declaration. After that, the two mandatory requirements are that a setup and teardown method be defined.

Test Case Setup

Each test case class has to create on its own any resources it needs in order to conduct the test tasks of every method in the test case. If certain objects are required, they must be created. At a minimum, this is going to involve instantiating the application class, and may involve such elaborate tasks as creating a database and populating it with test data.

With our now familiar shopping basket example, before a shopping basket object can be tested, it has to be created.

```
define_tag:'setup';
    library:'app_shoppingBasket.ctyp';
    var:'testBasket' = shoppingBasket;
/define_tag;
```

You'll notice that the custom type code for the shopping basket class had to be loaded first. While that file might exist in /LassoStartup during production running and integration testing of the application, for unit testing, it is best if the files can be loaded manually. I'll cover this detail again in the L-Unit Installation section, but it is best if L-Unit runs in a Lasso site that has no files in LassoStartup, and no files in the on-demand Libraries folder either.

If the application class code has to be loaded, where should it be loaded from? There's two options: one is in the /L-Unit/appClasses folder, and the other is wherever you want. A standard appClasses folder is defined in the L-Unit application configuration, and this path is passed to every test case through the parent class as the instance variable classPath. If you store application class files there, you can use a line like the following to load the file:

```
library:
    (self->'classPath') +
    'app_shoppingBasket.ctyp';
```

Otherwise, feel free to use any hard coded path you prefer. In either case, you'll want to be sure that those files will be found there any time the tests need to run.
What if you have do something as drastic as setup an entire database and populate it with test data, and you want to reuse that code for other test cases as well? When you have setup code that should be shared and reused, then go ahead and create a reusable file for that by creating a ctype or a ctag to do that. We call that a test fixture.

Whatever you do though, be sure the code is as self-contained as possible. Don't use page variables, use locals. You don't want any chance of that reusable code contaminating the test environment. For this reason, I strongly discourage the use of includes for writing reusable test fixtures.

As you write test methods, there can be some question whether to perform some setup tasks in the setup method or in the test method itself. Use basic logic. If the setup is needed for more than one test method, put it in the setup method, otherwise it can go inside the test method it is needed for. just be sure to follow the same guidelines that apply to the teardown method.

Test Case Teardown

Whatever gets built by the setup code, has to be destroyed by the teardown code. It sounds extreme, but just like was mentioned that test fixture code should be as self-contained as possible, so should test case code.

Many test cases will be run as part of a test suite, and we don't want environmental conditions of one test case contaminating another. If a database gets created in a setup for one test, and the test methods alter that database, the next test case should start with that same original database. What's the best way to guarantee that in an automated environment? Kill it, and build it again. Remember at the very beginning I mentioned that we want tests to be repeated in exactly the same manner every time? This setup and teardown process is part of ensuring that.

So, the teardown code should reverse anything the setup code did. This includes eliminating page variables used to create objects. It's not a common need in Lasso to do this, but it is possible:

```
define_tag:'teardown';
    vars->remove:'testBasket';
/define_tag;
```

That will eliminate the variable from the page scope completely.
With the setup and teardown essentials out of the way, the test case code can now focus on writing tests in the test methods.

The Setup and Teardown Cycle

The `->setup` message is sent before each test method is called, and the `->teardown` method is called after each test method. So, if there's 5 test methods, then there are 5 cycles of `->setup`, unique test method, and `->teardown`. Clearly this will impact how you write the `->setup` method.

Thinking about the need to create a test database, if it is going to be created and destroyed 5 times, clearly we want it to be a quick operation. If testing the class only requires 3 tables out of the 15 that your application uses, then create only those 3 tables. Also, unit testing is not performance testing. You don't need to run queries on databases with thousands of records. You probably only need a handful or records or maybe even one record. So, keep the setup (and teardown) tasks as efficient as possible.

Writing Test Methods

Rule #1 is that a test method name must start with `test_`, and the remainder of the name must be exactly the name of a method in the application class that is being tested by this test method. An example of that was given at the introduction to this section. No inputs are passed to a test method, so the `define_tag` declaration is a straight forward statement with no `-required` or `-optional` parameters.

Just to make sure there's no confusion, a test method does not imitate the application class for which it is named, it tests the application class method for which it is named. The test method creates various conditions under which the application method can be invoked and verifies the results of the application method are what is expected. The testing should include cases that create positive results, and cases that create failures on purpose so we can be sure the application method will behave gracefully when it doesn't run under optimum conditions.

Test Tasks

The body of a test method is a series of test tasks made up of assertions where possible, or custom code where an assertion can't be used.
Let's look at an example test method with some simple assertions.

```
define_tag:'test_addItem';

$testBasket->(addItem:
                -sku = 'abc123',
                -description = 'Lasso Unit Testing',
                -price = 24.99);

self->(assert_isArray:
        -expr = "$testBasket->'lineItems'");

self->(assert_isArraySizeOf:
        -expr = "$testBasket->'lineItems'",
        -size = 1);

self->(assert_isPair:
        -expr =
        "($testBasket->'lineItems')->get:1");

self->(assert_isTrue:
        -expr =
        "((($testBasket->'lineItems')->get:1")->find:'price') == 24.99;

/define_tag;
```

The basket object $testBasket was created as part of the ->setup method. So, we head straight into sending the ->addItem message and passing some typical data. Having done that, we know the intent is for that data to be stored in an instance variable named 'lineItems'. We now use some assertions to verify that the data indeed got into the instance variable, and that it got there in the correct format. The 'lineItems' data structure should look like this:

```
lineItems = (array:
sku = (map:
'description' = string,
'price' = decimal));
```

Some of the things we can verify is that 'lineItems' is an array, that the array has one element, and that the first element is a pair data type. We could also use some assertions to verify that indeed, the sku, description, and price of the item is exactly what we supplied.

Each of these assertions verifies one specific detail. If the price was correct, but the description did not come back matched, we have a very specific functional point to investigate what went wrong.

Custom Test Tasks

There's going to be some cases where the assertions library doesn't have a good tool to use for a certain test you want to do. Perhaps this indicates the assertions library needs updated, but let's assume we need to deal with it anyway, and write a custom task without an existing assertion.

Technically anything can be done with an assertion because of the generic assertions like assert_isTrue. The main reason I suggest that an existing assertion may not be suitable is for the error message that is generated. An error message which can provide a specific clue as to why a test failed is an important part of unit testing. So it may be necessary to write a custom task which can provide a meaningful error message.

That custom task code must still ultimately evaluate to a true or false result, and it must focus on testing a single critical data point as was discussed in the Assertions section. Consider your custom task code a custom assertion. (If it something you find you use over and over, then you may want to suggest an extension to the assertions library).

Writing a custom task requires taking several steps that the assertions library normally does for you. This mostly has to do with handling the result data.

The test case parent code has an internal data structure for a test result that looks like this:

```
&amp;nbsp;   map:
    'name' = string,
    'results' = boolean,
    'message' = string,
    'time' = integer (milliseconds)
```

When you write a custom task, you need to provide the data to fill that result map. Additionally, there's some housekeeping steps to perform. The following code is an outline that should be followed when creating custom tasks:

```
    self->startTask;

    // your test code goes here
    // it ultimately has to populate
    // (self->'taskPassed') with a true or false
    // (self->'taskMessage') with a string

    self->storeResult: map(
      'name' = 'ENTER_A_NAME_FOR_THIS_TASK',
      'result' = self->'taskPassed',
      'message' = self->'taskMessage',
      'time' = self->endTask
    );
```

The ->startTask method ensures that internal resources are cleared, and it also starts a task timer. The ->endTask method stops the timer, and resets internal resources as well. They're mini setup and teardown steps for custom tasks.

Let's look at an example of a custom task using the above skeleton by considering a timestamp element in the shopping basket. The idea is that the basket stores a price because that price is honored for a period of time such as 24 hours. At checkout we want to validate that all prices in the

basket are less than 24 hours old, and if not we would retrieve the latest price. So, now our basket data strcture looks like this:

```
lineItems = (array:
sku = (map:
'description' = string,
'price' = decimal,
'timestamp' = YYYY-MM-DD HH:MM:SS));
```

The ->addItem message is not changed because the timestamp is populated automatically internal to the ->addItem method code.

Let's add a test that verifies that in fact, a timestamp was made.

```
self->startTask;

if: valid_date:
      ((((($testBasket->'lineItems')->get:1)->second)->find:'timestamp'),
      -format='%Q %T';

   self->'taskPassed' = true;
   self->'taskMessage' = 'passed';

else;

   self->'taskPassed' = false;
   self->'taskMessage' = 'The time stamp of ';
   self->'taskMessage' += ((((($testBasket
        ->'lineItems')->get:1)->second)
         ->find:'timestamp');

   self->'taskMessage' = ' does not meet the formatting requirements of %Q %T';

/if;

self->storeResult: (map:
   'name' = 'custom_timeIsFormatQT',
   'result' = self->'taskPassed',
   'message' = self->'taskMessage',
   'time' = self->endTask);
```

In testing valid timestamps we'd also want to be sure the timestamp was not all zeros, and possibly a few other conditions as well, but the objective here was just to show the structure of a custom task and introduce the taskPassed and taskMessage variables.

Cleaning Up

There's no specific method for this because there can't be, but just a reminder that if the test method has created any page variables, or other resources, then it should also destroy those resources when the tasks have completed.

Also, just to clarify, the developer does not have to do anything to consolidate test task results. The L-Unit application does that work automatically.

Test Suites

A test suite is a collection of test cases that are connected topically in some manner, and that you want tested at the same time.

The developer doesn't really have to do anything with respect to test suites except to define a text file that is a simple list of the test cases to be executed. This can be done manually or through the L-Unit web interface. The details of the latter will be covered in the next chapter Working with L-Unit.

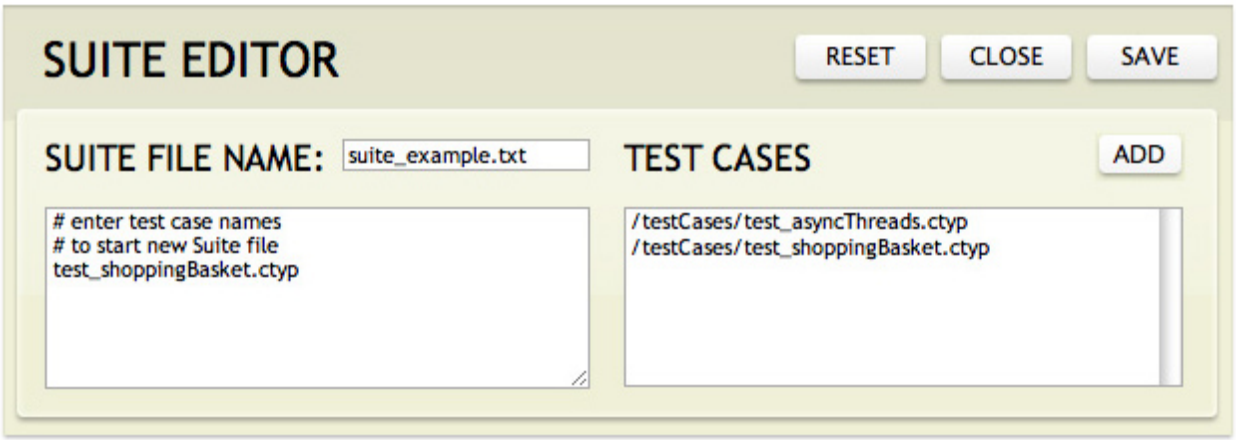
Working with L-Unit

The L-Unit application organises, executes, and displays the results of the developer's test cases and test suites.

The L-Unit application runs in a web browser on the developer's local machine or on a test server on the LAN. (It can run on a remote server, but the app isn't security hardened, and it's not recommended to run it across the internet).

The application has 3 workspaces: the test suites list, the suites editor, and the result displays (tests & metrics). When the application is launched, the suites list is presented which displays all test suite files found in the /testSuites/ folder. A test suite file is a simple text file which lists the test case source code files which are to be run.

From this panel, a new test suite can be created, an existing suite edited or deleted, or a test can be viewed prior to running.



Creating / Editing Test Suites

Clicking the New button or an Edit button for a particular test suite will open the test suite editor above the test suite list.

The left panel displays the test suite name and file contents. The right panel presents a list of test cases as found in the L-Unit /testCases/ folder and any test cases found within the /appClasses folder and sub directories. To add a test case to the test suite, select a name from the test case list, and click the Add button. The names can also be added manually by typing in the test suite file contents field.

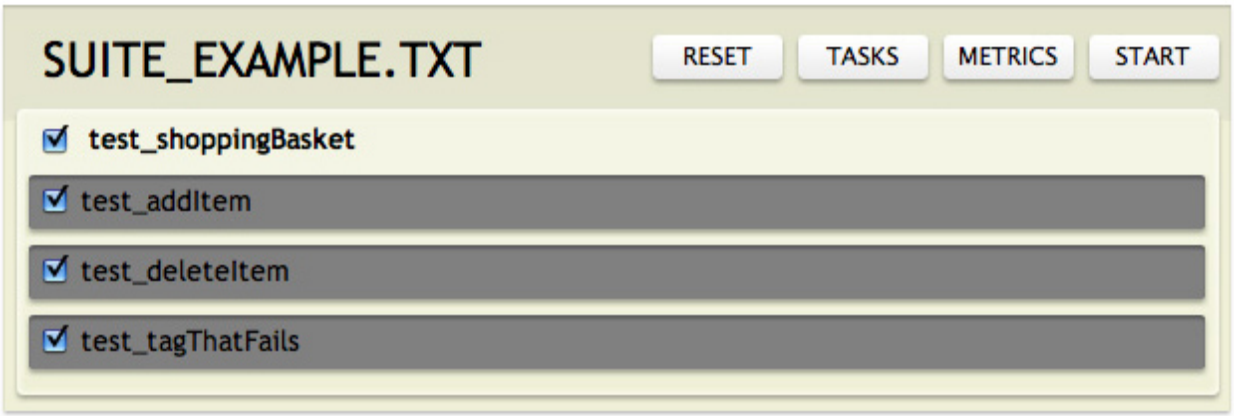
The Reset button will clear the editor fields, and the Close button will hide the editor panels. Clicking Save will write the suite file to disk in the /testSuites/ folder, and add the file to the suite list displayed. The list forms two even columns automatically to benefit page layout. Technically there are currently no restrictions on file extensions for test suites, but I suggest using either .txt or .cnfg as there may be future enhancements to the editor that requires known file extensions.

Running Test Suites

Clicking the View button of any test suite will show you an overview of the test cases and test methods that will be run. Select which tests you would like to run (useful for test development) and click start to begin testing. If you have javascript enabled the test results will be populated as they are returned. You can view the results on one of two tabs: metrics and tests. Metrics provides and overview of test information. Tests shows the results of each test task.

Concurrent Tests

Due to the nature of how the tests are run (each test runs in an asynchronous process for stability) only one instance each test suite can be running at the same time. Due to this, the view button may display a previously completed or even a currently running test. This allows for multiple developers on multiple machines to observe the same or previous tests, and ensures that running tests do not interfere with one and another (particularly database construction & destruction). Tests can be reset (all results cleared) by click the reset but and aborted (tests are stopped at current point) by clicking the abort button.



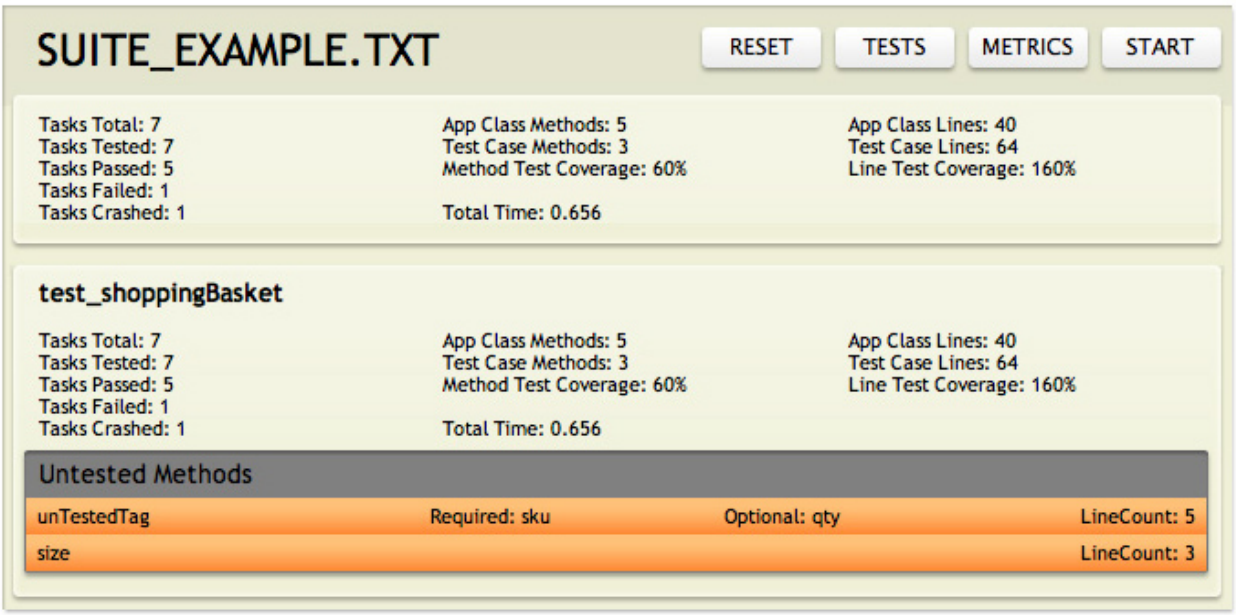
Test Metrics

The Metrics section presents a series of values derived from counting the number of test cases, test methods, and individual tasks performed, and summaries how many passed, failed or crashed. The top section summarises all test cases run, each subsequent section summarises the relevant test case.

The Metrics section also performs some analysis on the application class and test case source code. The application class file is analysed to determine how many methods it has, and approximately how many lines of code each method has (empty lines and comment lines are not counted). Then the test case class is analysed to determine how many test methods correspond to the application class methods. With that value in hand, L-Unit can calculate what percentage of the methods are covered, and what percentage of total code lines are covered. The coverage calculations are totals for the entire test suite.

These metrics provide an indicator as to how much of the application code is covered by tests. While a project is young and being developed by people new to the application, a high coverage percentage with no failures will yield the confidence that development is producing reliable code. As a project matures, that value might be relaxed a little by allowing very simple methods to not be tested. It has been expressed to me that initial line coverage should be close to 90%, and that it should be relaxed to no less than 70% as a project matures.

Any untested Methods will be displayed in within the relevant test case along with any -required or -optional parameters. Untested methods are determined by attempting to match the test methods their corresponding app methods (test_addItem = addItem). To make use of this feature it's important that test methods are accurately named after their counter parts (with the "test_" prefix).

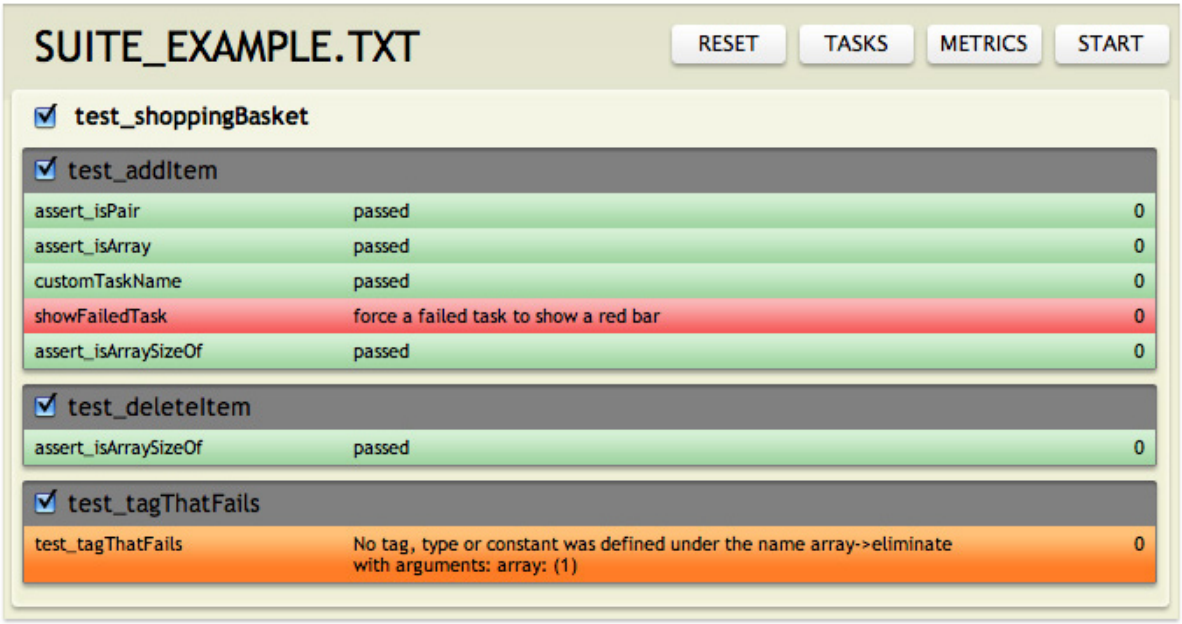


Test Tasks Results

The Tasks section provides detailed output of individual tasks that were performed in each test method of each test case. Green bars indicate tasks that passed, red bars indicate tasks that failed, and orange bars indicate tasks where either the task or the application code crashed.

L-Unit should catch application code crashes and continue testing with the next test method. the task result bar should provide the code that Lasso generates about the crash.

Each task bar also provides an indicator of the time required to process. Test suites should be able to run in only a few minutes, and if any one test is starting to take a long time, it may need to be rethought, or separated into a separate suite.



Organizing Application Classes and Test Cases

The application class code and the test case code to be run by L-Unit must be on the same machine as the L-Unit application.L-Unit provides a directory structure to store application classes and test case classes, but they can be located anywhere on the system's local drives (anywhere that the Lasso Site has been configured to access files).

It's recommended that test cases (test_myType.ctyp) are stored in the same location the respective class file (myType.ctyp). This keeps things simple a code management perspective (tests in the same location and version control system as source) and allows for easy access to update the test case.

A few options for getting the source files to where L-Unit can run them:

Subversion / Version Control

Automated exporting is on the agenda in the meantime you can:

1. Export your appClass folders to the L-Unit appClass folder before testing
2. Configure svn:externals on the appClass folder within your L-Unit project / repository folder and run an update before testing. Example: myAppFolder svn://localhost/Trunk/myApp

Symbolic links

Create symlinks of the application project folders that contain classes to be tested. Locate the linked folders inside the L-Unit folder

Rsync / Automatic

Configure a script / automated tasks that regularly (every few minutes) updates the appClass files

Copy

Brute force copy the files to the L-Unit folder — It will atleast get you going for the time being...

Assertions Reference

All assertions require an input named -expr which is a string form of Laso code that can [process]ed. Write the code in LassoScript syntax, and do not enclose it in brackets. If the expression is a single line, do not terminate it with a semicolon. (The library does need to be updated to handle these details automatically, but as of this release it doesn't do that).

The following assertions require no additional parameters

- assert_isTrue
- assert_isFalse
- assert_isNotEmpty
- assert_isNotNull
- assert_isBoolean
- assert_isInteger
- assert_isDecimal
- assert_isString
- assert_isBytes
- assert_isPair
- assert_isArray
- assert_isMap
- assert_isSet
- assert_isList
- assert_isStack
- assert_isQueue
- assert_isPriorityQueue
- assert_isSeries
- assert_isTreeMap
- assert_isFile
- assert_isImage
- assert_isXML

The following assertions require additional parameters

assert_isTypeOf

Requires a -type param which a string value of the type that is expected:

```
assert_isTypeOf:
-expr = "$myObject",
-type = 'purchaseOrder';

assert_isTypeOf:
-expr = "$myObject",
-type = 'array';
```

assert_isDecimalInRange
assert_isIntegerInRange
assert_isArraySizeInRange
assert_isSizeInRange

These assertions verify that a value is within a specified range.
Requires a -min param and a -max param, both of which are decimals for the decimal assertion, and integers for the others.

```
assert_isIntegerInRange:
   -expr = "$guitarVolume",
   -min = 8,
   -max = 11;
```

assert_isDecimalMin
assert_isIntegerMin
assert_isArraySizeMin
assert_isSizeMin

These assertions verify that a value meets at least a minimum setting.
Requires a -min param which is a decimal for the decimal assertion, and an integer for the others.

```
assert_isDecimalMin:
    -expr = "$hourlyWage",
    -min = 5;
```

assert_isDecimalMax
assert_isIntegerMax
assert_isArraySizeMax
assert_isSizeMax

These assertions verify that a value does not exceed a maximum setting.
Requires a -max param which is a decimal for the decimal assertion, and integers for the others.

```
assert_isSizeMax:
    -expr = "$pressColors",
    -max = 5;
```

assert_isArraySizeOf
assert_isSizeOf

These assertions verify that a data type has an exact size.
Requires a -sizeOf param which is an integer.

```
assert_isArraySizeOf:
   -expr = "$dateParts",
   -sizeOf = 3
```

assert_isStringContaining
assert_isArrayContaining
assert_isSetContaining
assert_isListContaining
assert_isMapContaining
assert_isTreeMapContaining

These assertions verify that a data type contains a certain string value.
Requires a -contains param which is a string

```
assert_isMapContaining:
   -expr = "$dateParts",
   -containing = 'year';
```