Astemes LMock

Astemes - Anton Sundqvist

Anton Sundqvist

Table of contents

1. Introduction to LMock

This document introduces the basic features and components of LMock. For a more detailed discussion on how to write mock-based tests using LMock, please see this page.

1.1 Typical Use Case

LMock is used to generate test doubles to replace concrete classes in unit and integration tests. A LabVIEW interface must be used to define the VI:s which are to be mocked. The system under test would typically depend on the abstract Interface and the concrete class would be used in the actual application. One typically starts with a first version of the Interface and crete a mock class for the interface. This class would be used in tests to verify expected behavior and simulate return values during test. The generated mock can later be updated when the mocked Interface changes.

1.2 The Mock Class

LMock is designed for mocking LabVIEW Interfaces. A mock may be generated from any Interface using the right click menu in the LabVIEW Project Explorer. The generated mock class will inherit from the LMock `Mock.lvclass` and implement all the Dynamic Dispatch VI:s of the mocked Interface. It will also provide a When VI for each Dynamic Dispatch VI. When VI:s are used to declare return values *when* the VI is called. If the mocked Interface changes, the mock may be updated using the right-click menu in the project explorer.

The `Mock.lvclass` implements the necessary boilerplate code for handling expectations, VI calls and return values. The implementation uses LabVIEW queues, which is a very performant structure with little overhead. This makes the performance orders of magnitude better than using something like VI server.

1.3 API

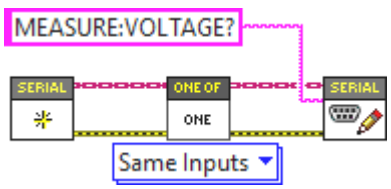
The LMock API can be found in the LMock Palette after installing the package.

It consists of three expectation VIs `Never.vi`, `One.vi` and `One or More.vi` as well as the `Verify.vi`.

A typical use case would be to first create a mock using its constructor, then configure expectations on the mock using the expectation API. Once the expectations are configured the code under test would bwe exercised and the `Verify.vi` would finally be called. This API method verifies that the expectations are met and generates the result description message. Return values from each VI call may optionally be declared using the When API.

1.4 Expectations

The expectation API is designed for readability. An expectation is declared using one of the provided API VIs together with a call to the expected Dynamic Dispatch VI. The Dynamic Dispatch VI is implemented by the mock class and generated through scripting when creating/updating the mock. An example is given below.



There is always a pair of VI:s for each declared expectation which would read as "expect *one of* vi_name.vi with *same inputs*". The following expectations may be used

- Never - Passes only if the declared VI call was never made
- One - Passes only if the declared VI call is made exactly once
- One or More - Passes if the declared VI call is made one or more times
- Exactly - Passes only if the VI call is made exactly a given number of times
- At Least - Passes if the VI Call is made at least a given number of times

All the expectations are polymorphic and offers multiple comparison options used to declare *how* VI calls should be compared when validating a mock.

1.5 Comparisons

Comparisons define *how* two VI calls should be compared and if they should be regarded as matching. The available comparators are listed below.

- Identical Inputs - Requires all inputs to be equal for the VI calls to match
- Anthing at Inputs - The inputs are ignored and the VI calls will match as long as the same VI is called
- Matching String Inputs - Any string input will be regarded as a regular expression when the mock is configured, and matches if the expression matches the string value when the VI is called.
- Tolerant Numeric Inputs - Any numeric input will need to be within the given tolerance when the VI is called for the VI call to be considered equal.

1.6 Return Values

LMock provides an API for queueing up return values from a VI call through the When API. Using the generated `When vi_name.vi` VI, return values for calls to the `vi_name.vi` are enqueued. Each call to `vi_name.vi` will dequeue the next return values, using LabVIEW type-specific defaults if the queue is empty.

The When API is slightly unconventional, as it reverses the direction of data flow to be right-to-left for return values. Each indicator on the actual VI is replaced with a control on the corresponding When VI, which makes the code read nicely in a unit test.

1.7 Failure descriptions

One important feature of LMock is that it provides fluent failure descriptions when mocks are verified. The descriptions explains, in plain English, the expected number of VI calls, how the VI:s are compared and lists calls made to the expected VI. An example would be the following result description of a passing test

```
Write to Log.vi Called Once with String Inputs Matching Expectation
Call 1: Text to Write: "Test" found in "22/07/2023 21:18:50 READ:   Test", error in (no error): No Error(Cluster) == No Error(Cluster)
```

2. Writing Tests using Mocks

This document walks through the basic workflow using LMock to test LabVIEW code using mock objects. All the code introduced in this document is available as example code installed with the toolkit package. The example code is found at `C:\Program Files (x86)\National Instruments\LabVIEW 202X\examples\Astemes\LMock`

2.1 Prerequisites

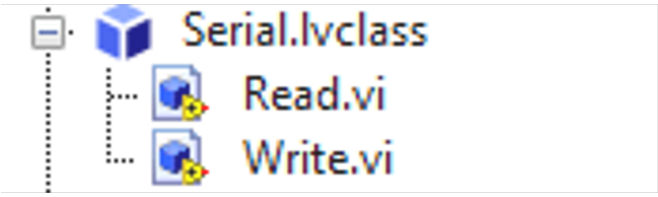
To follow along with the instructions on this page you will need to have LabVIEW version 2020 or later installed as well as the LUnit unit testing framework and the LMock mocking toolkit. The complete code is available as in the LabVIEW examples directory after installation of LMock.

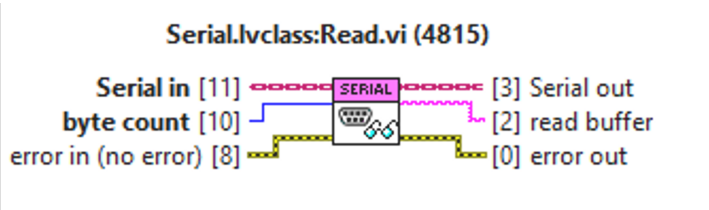
2.2 Using LMock

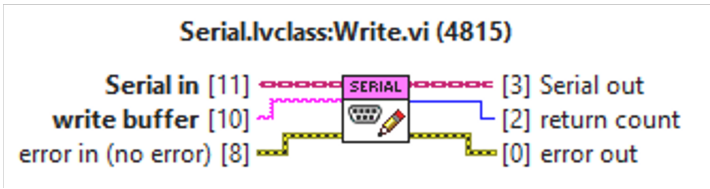
Below is a brief introduction to basic usage of LMock. There is a rich syntax provided in the API, for more details, please see the API documentation. In addition to introducing the API of LMock, we will also consider and highlight some of the benefits of test-first development.

To make this section a bit less abstract, let us continue by looking at a simple example. The source code is available in the repository and installed as example code.

We will consider a typical Test Driven Development workflow for developing a driver for some instrument. The details of the instrument are not of interest for now. We will want to verify what is sent to the instrument by the driver, and how the responses are interpreted. We would typically start by defining an interface, which wraps the calls to the communication bus. Note that we do not need to consider what the physical interface actually is (RS-232, TCP/IP, etc.). We only care that we can write to and read from the bus and that the data is interpreted as a string. Initially, our interface could look something like below, as we know we will later want to wrap NI VISA calls in this interface.



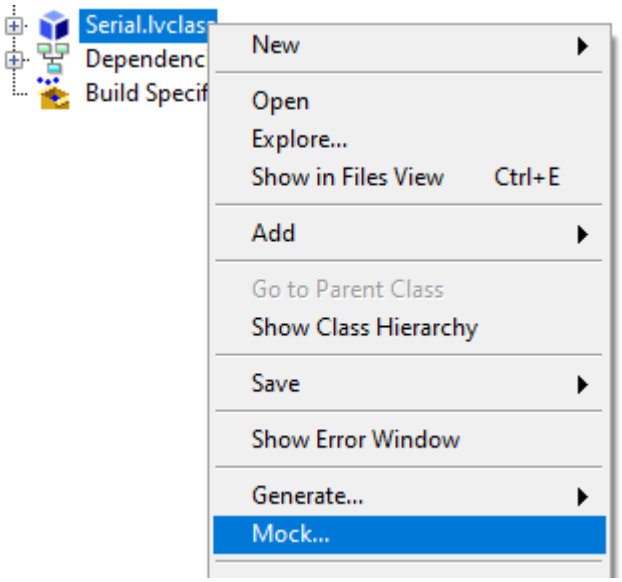




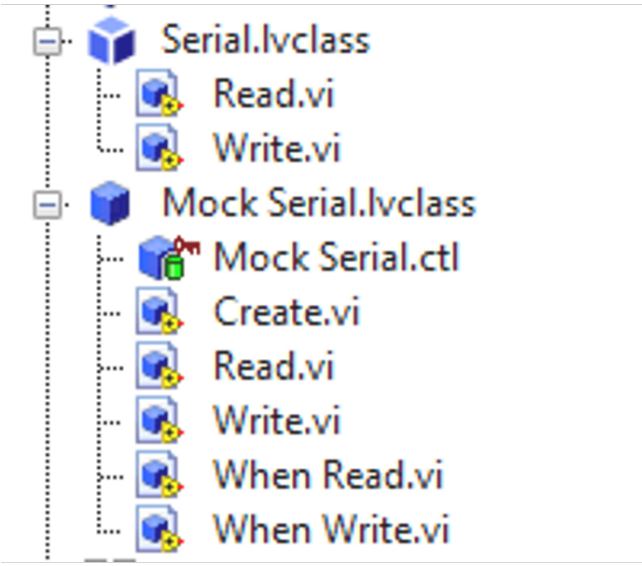
As we only need the read and write methods for now, we will not bloat the interface with methods we might need in the future. We can always extend it later when the need arises and our design has settled.

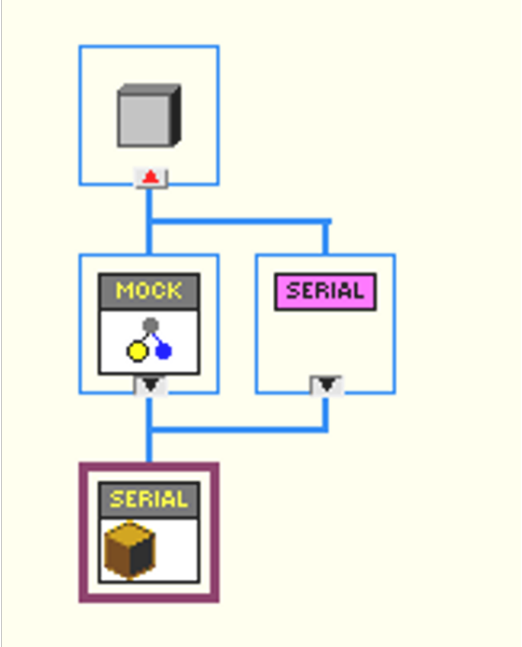
2.2.1 Mocking an Interface

LMock does not allow for mocking concrete classes, only abstract interfaces may be mocked. To generate a Mock for an interface in LabVIEW, simply right-click it in the LabVIEW Project Explorer and select `Mock...`.



This will generate a new class which implements the selected interface and inherits from the LMock `Mock.lvclass`. All dynamic dispatch VIs are overridden by the mock class, and a special `When` VI is generated for each dynamic dispatch VI. The `When` API is used for defining outputs of the mock class when it is called.



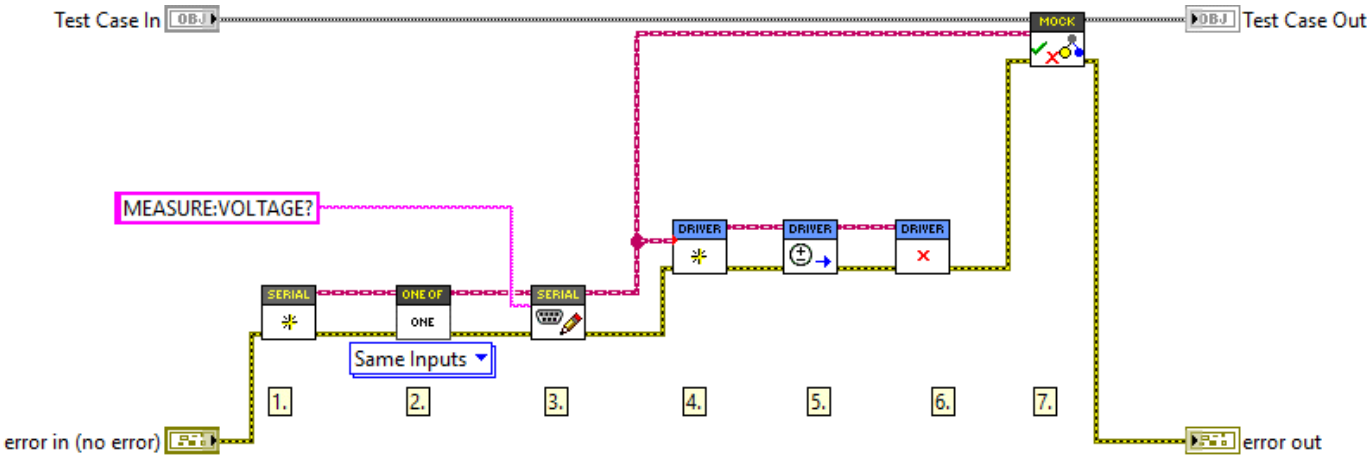


The generated mock class can be used as a test double, replacing the concrete implementation of the interface when writing a test. We now have the tools we need to start writing our first unit test. To get going, we create an LUnit Test Case and a `Driver.lvclass` class for the driver we are about to develop, which is our system under test. If test-driven development is unfamiliar, please do not get scared by the amount of classes we are creating. Our test will tend to push the classes to become loosely coupled and highly cohesive. It is actually less painful to maintain a large amount of small, cohesive, classes than a small amount of large classes.

2.2.2 Setting up Expectations

We now have an empty class called `Driver.lvclass` and an LUnit Test Case class called `Driver Test.lvclass`. Next, we create a test case VI from the static test template and start filling it out. The first thing we are going to test is that the driver sends the correct query string to the instrument when we try to read a voltage.

We will now use the LMock syntax to set up expectations on the mock. This is a central concept for mock-based testing. The expectation API is designed so that we need to set up what we expect to happen *before* calling the code which causes the events we are testing for. The test case looks as below.



We first create an instance of `Serial Mock` **(1.)**. Using `One.vi` **(2.)** and the `Write.vi` **(3.)**, we declare that we expect the `Write.vi` method to be called exactly once during the test with the given inputs. The `One.vi` is an LMock API VI used to configure an expectation. The next VI called on the mock, after `One.vi`, specifies the call we are expecting together with the desired inputs.

The polymorphic VI selector showing `Same Inputs` configures the expectation to match only when all inputs to the `Write.vi` are identical to the ones used at **(3.)**. So in this case, the test will pass if the `Write.vi` is called during the test with the string input of `MEASURE:VOLTAGE?` and no error at the Error In control.

2.2.3 Verifying Behavior

Now the Mock has been configured, and we continue by creating an instance of our system under test, the `Driver.lvclass`, and inject our configured `Mock Serial` to the constructor **(4.)**. Next, we exercise the system under test by calling the `Read Voltage.vi` **(5.)**, which is what should trigger the mock to be called. We clean up **(6.)** and call the `Verify.vi` API method with the mock as input, which does the work of verifying the mock and generating the result description message. The value of a clear failure description should not be underestimated.

As we have been creating the required VIs for our `Driver.lvclass` while writing the tests, the block diagrams are still empty, so our test case should fail. And indeed it does, with the following failure message:

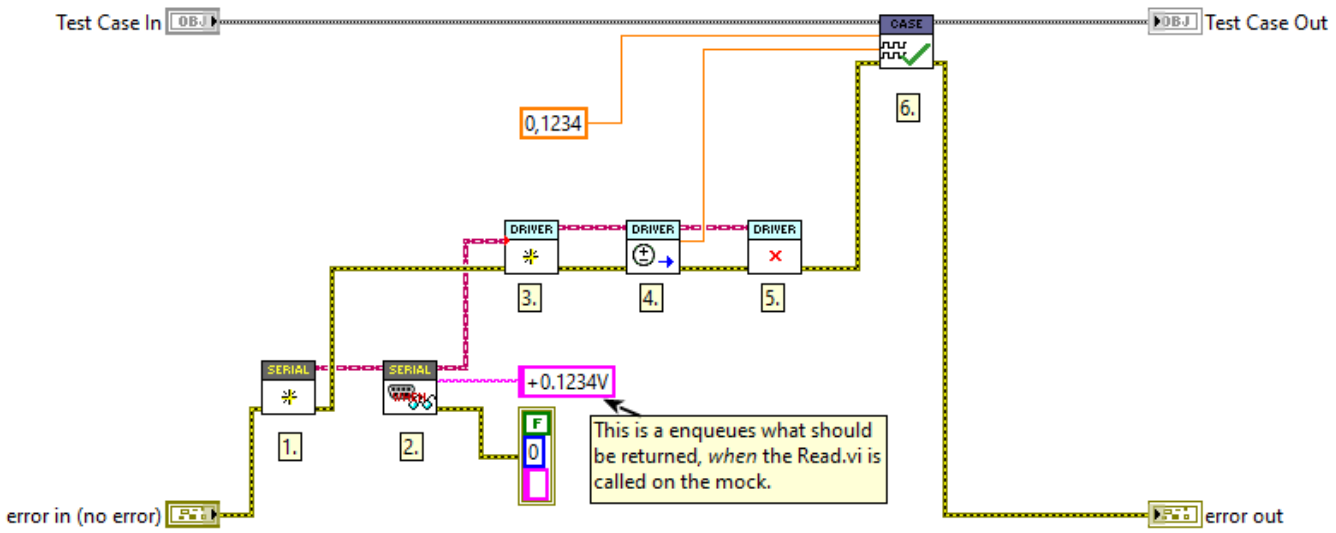
```
Write.vi Expected Once but Never Called with Expected Inputs
```

The implementation for solving this is trivial. After filling in the blanks, the test passes, and we get the gratifying green bar and the following message:

```
Write.vi Called Once with Expected Inputs
Call 1: error in (no error): No Error(Cluster) == No Error(Cluster), write buffer: MEASURE:VOLTAGE?(String) == MEASURE:VOLTAGE?(String)
```

2.2.4 Defining Stubbed Behavior

Now we have verified that our system under test sends the required message for reading a voltage from the instrument. Next, we need to consider parsing the response from the instrument. By working one test at a time, we are able to break down the task and focus on one thing at a time. To continue, we will need to use the `When Read.vi` generated by the mocking framework. The next test case looks as follows.



As before, we create a new mock instance **(1.)**, but now we do not configure any expectations for the mock, as we are not going to verify calls made to the mock. Instead, we use the `When Read.vi` **(2.)** to define what is returned by the `Read.vi` when it is called. As seen in the figure, the LMock When API has inputs where the interface API has outputs, so the direction of the data flow is reversed. This is intentional, as it results in intuitively readable test code when one gets used to the syntax.

Similar to before, the system under test is initialized **(3.)**, exercised **(4.)**, and cleared **(5.)**. As we do not need to verify calls to the mock, we use the regular test verification methods of LUnit **(6.)** to verify that the response is properly parsed.

2.3 Adding a New Feature

The classes developed above are highly cohesive and the `Driver.lvclass` is weakly coupled to the serial interface through the `Serial.lvclass` interface. There is a clear separation of concerns, and we have a clear architectural boundary introduced by the abstract interface. This gives a lot of flexibility as we may **without making any changes to our `Driver.lvclass`**:

- change the concrete implementation of the `Serial.lvclass` interface to use a different bus or calling the bus in a different way
- change how the data is sent to the bus, *e.g.* adding end of line characters
- and more

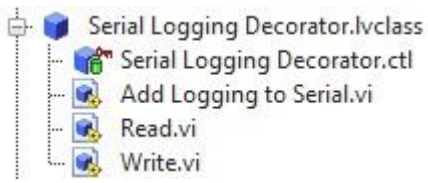
Let us next explore how we can implement a new, test-driven, feature and how this plugs into our design.

When working with hardware, it is often valuable to be able to probe the data sent on the bus during debugging and integration of the whole system. The logging feature might then need to be turned off in production to minimize overhead and prevent excessive logs from accumulating. A well-designed system lets us toggle the logging on or off and allows for redirecting the output to different locations *e.g.* a file or a debug view.

2.3.1 Identifying a Design Pattern

In the spirit of test-driven development, we will start by creating a test case. This test will need to verify that data sent to, and later read from, the instrument is also sent to a log. This starts to feel like the common decorator pattern, which basically means that we create a wrapper for our interface that "decorates" calls made to the interface by adding logging capabilities.

We could define a decorator class with the following structure to start filling in our fresh test case.



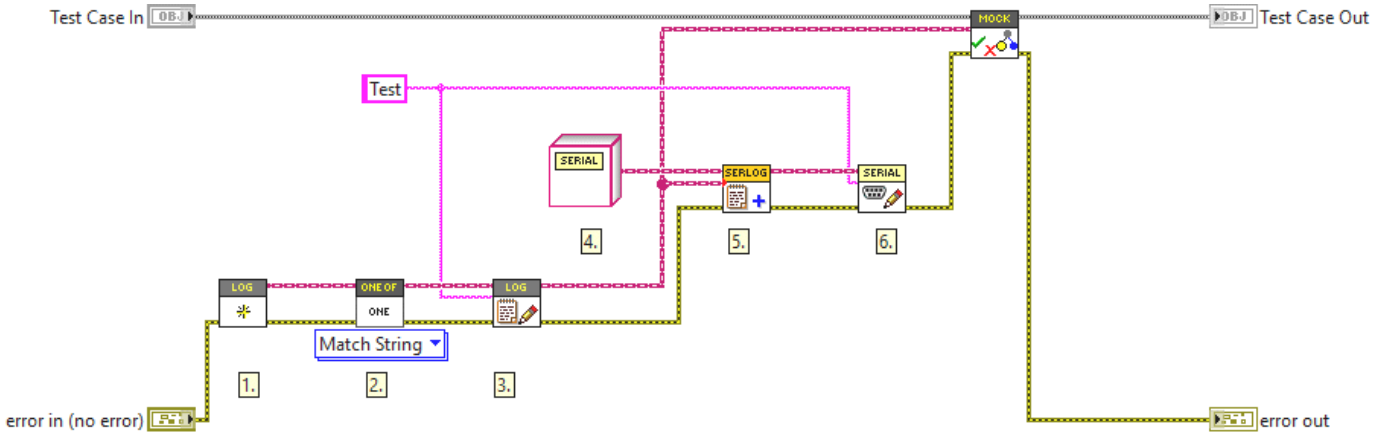
The `Read.vi` and `Write.vi` are the overrides of the `Serial.lvclass` interface (which we are about to decorate). The `Add Logging to Serial.vi` is the VI we are going to use to attach the decorator to any class implementing the `Serial.lvclass` interface (including our mock from before).

The implementation of the log is not really important at this point, we just need an interface representing the log to which we can write. The concrete log may then be implemented as a file, an indicator, an email message, or whatever we may need later in our project. Let us simply define the an interface for the log with a `Write to Log.vi` method as below.

You can probably guess what the abstract write VI looks like; it only has one string control in addition to the default control and indicator pairs.

2.3.2 Using a Match String Comparator

We can now finish our first test case for the decorator which looks as follows.



This looks very familiar to what we had before, we create a mock **(1.)** and declare an expectation that the string literal "Test" will be matched to the log **(2.)** and **(3.)**. In this case, we use the polymorphic VI selector to use the `Match String` version of the comparator. This works similarly to the native LabVIEW `Match Pattern.vi` for any string input, *i.e.* the test will pass if the string "Test" is matched with what is written to the log. This is useful, as we might want to add more information to the log *e.g.* a timestamp.

Note that we use the abstract `Serial.lvclass` interface **(4.)** to which we attach the decorator **(5.)**. This is because we are not really interested in what the Serial interface does, we now only care about what is written to the log. Using the abstract interface makes the test easier to interpret (and also shaves off some microseconds from the test time).

Finally, we call the `Write.vi` on our new decorator, which should trigger the written data to be sent to the log. As all the block diagrams are empty, this fails with the result:

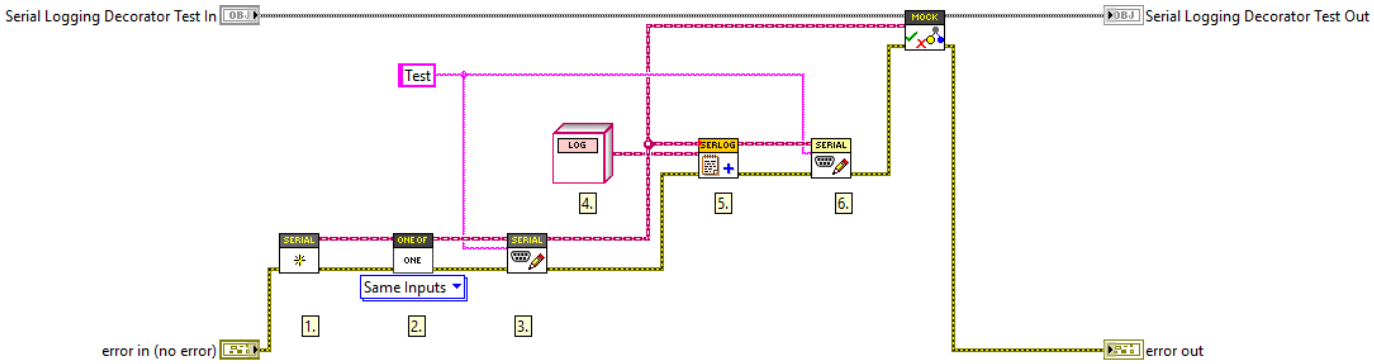
```
Write to Log.vi Expected Once but Never Called with String Inputs Matching Expectation
```

Again, as is often the case during test-driven development, the implementation is rather trivial. We can now take this one step further and add the timestamp and "WRITE" string to indicate the action being logged, as alluded to earlier, while still keeping the test passing. Now the result message reads as:

```
Write to Log.vi Called Once with String Inputs Matching Expectation
Call 1: Text to Write: "Test" found in "21/07/2023 21:17:42 WRITE: Test", error in (no error): No Error(Cluster) == No Error(Cluster)
```

2.3.3 Forwarding Calls to Decorated Object

The purpose of the decorator design pattern is to add behavior to a class, while not changing what the decorated class does. Currently, our code fails to do this as the `Write.vi` call is never forwarded to the decorated class. Let's add a test to fix this.

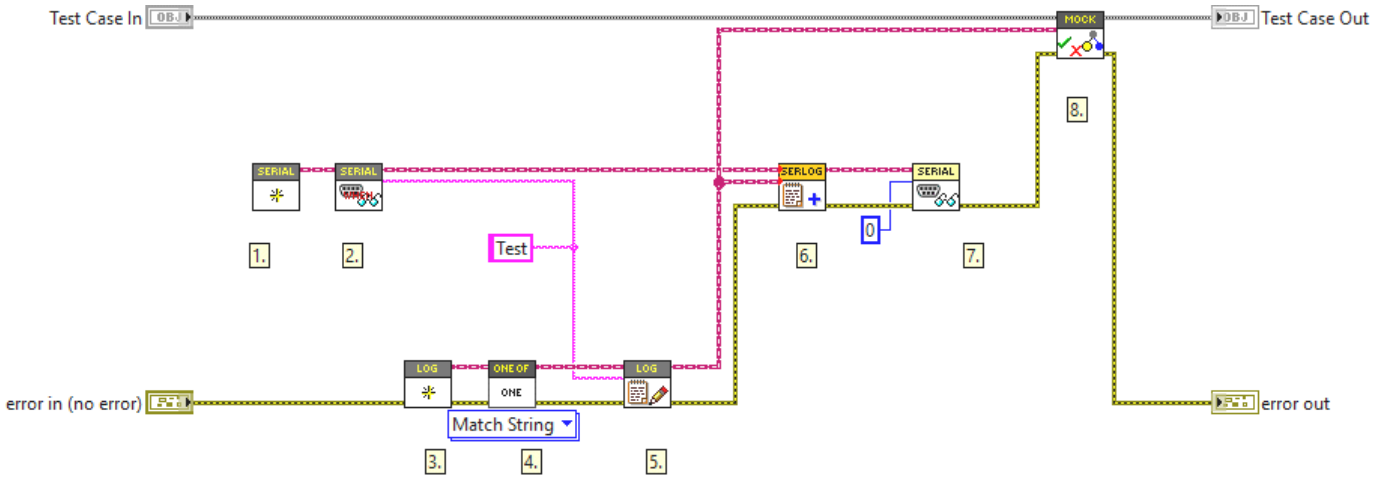


We now create **(1.)** the mock and configure our expectation **(2.)** that it should be called with the string literal "Test" **(3.)**. We now use the abstract `Log.lvclass` interface **(4.)** as input to the `Add Logging to Serial.vi` **(5.)** to remove any ambiguity as to what we are testing. We then call the `Write.vi` method **(6.)**, which is the call that must be forwarded to the decorated class. As expected, this test first fails, and after a very straightforward implementation, the result reads:

```
Write.vi Called Once with Expected Inputs
Call 1: error in (no error): No Error(Cluster) == No Error(Cluster), write buffer: Test(String) == Test(String)
```

2.3.4 Using Multiple Mocks in One Test

Next, we will need to implement the `Read.vi` method of the `Serial.lvclass` interface. This will be very similar to what we have seen before, except now we will need to use both our mocks in the same test. We will need to use our `Mock Serial.lvclass` to fake what is read from the bus and our `Mock Log.lvclass` to verify that this data is actually forwarded to the log. The test will look as below.



This should look familiar at this point. We create a `Mock Serial` **(1.)** and configure our expectation **(2.)** that it should return the string literal "Test" when the `Read.vi` method is called. We continue to create a `Mock Log` **(3.)** which we configure to expect one call to `Log.vi` with the same string literal as before **(4-5)**. We then attach our logging decorator to the `Mock Serial` and direct the output to the `Mock Log`. Finally, we call the `Read.vi` **(7.)** and verify our expectation **(8.)**.

After implementing the `Read.vi` and adding the timestamp and "READ" string, we get the result:

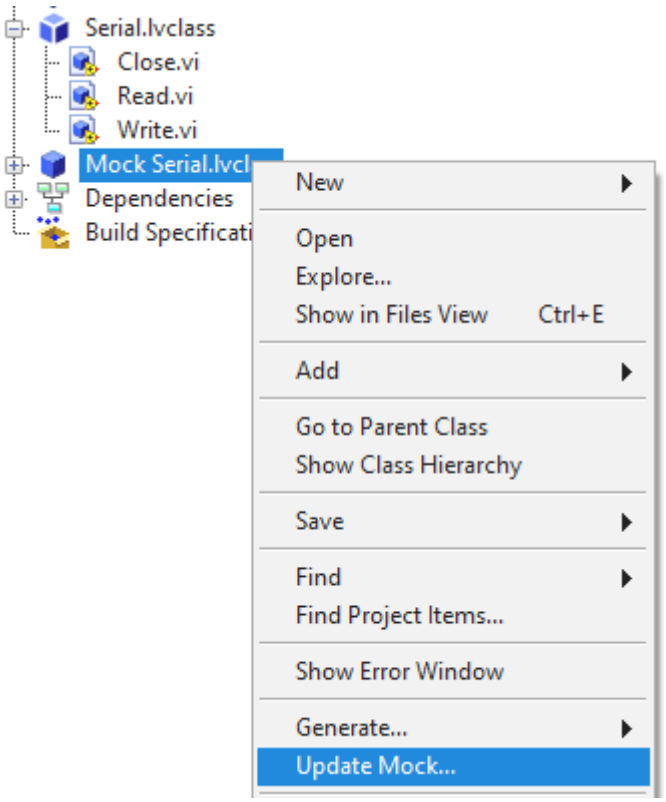
```
Write to Log.vi Called Once with String Inputs Matching Expectation
Call 1: Text to Write: "Test" found in "21/07/2023 21:48:31 READ: Test", error in (no error): No Error(Cluster) == No Error(Cluster)
```

We should now write one more test to verify that the output of the decorated `Read.vi` method is returned by the decorator. Writing this final test is left as an exercise. (As a footnote, I first skipped this test as I thought it was too trivial, which caused a silly bug and pushed me to add it anyway. In my experience, it is more common than not that I make a mistake when I cheat and skip writing a test. But maybe I am just a sub-average developer.)

2.4 Updating a Mock Class

As we work with the code, we might (with very high probability) need to change or update our design. Let us now explore such a scenario and assume that we now need to make sure that the bus is closed when we are done with it. At this point, we need to add a new method to our abstract interface. After this, our mock class will be broken because it does not implement our new method.

LMock offers a feature for updating existing Mocks by right-clicking a mock in the LabVIEW project explorer and selecting `Update Mock...`.



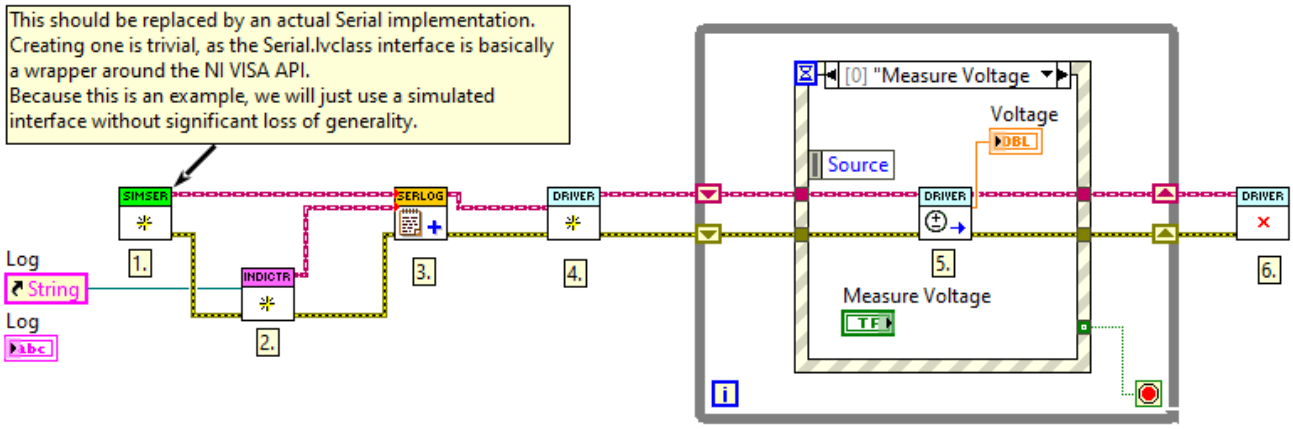
This will replace the existing mock with a new mock re-generated from the mocked interface. All existing code is relinked to this updated mock class. We can now continue writing tests using the new API VIs to verify that `Close.vi` is called appropriately..

2.5 Putting it All Together

Let us now conclude by looking at how we would use the classes and interfaces developed above in an actual application. We will need concrete classes implementing the `Serial.lvclass` and `Log.lvclass` interfaces.

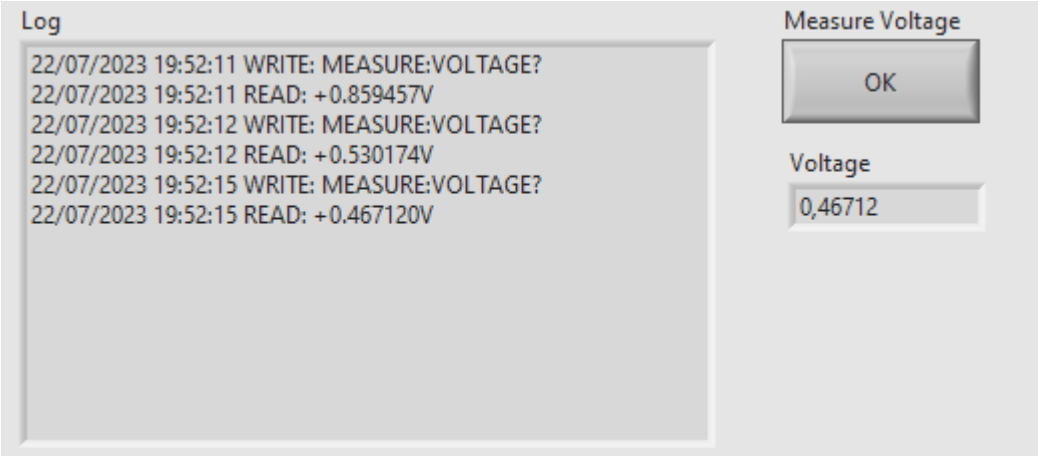
We will now introduce two new classes. First, we have created a `Simulated Instrument` which gives a random reading using the same syntax as the actual instrument when read. Second, we add a `String Indicator Logger` class implementing the `Log` interface, which concatenates the data written to the log and writes it to a String Indicator. The indicator is written through by reference, and the reference is configured in the constructor.

As a minimal demonstration, let us consider the following VI.



We initialize the `Simulated Instrument` **(1.)** and a `String Indicator Logger` **(2.)**. We attach the logger to the simulated instrument using the logging decorator **(3.)**.

Now we wire the decorated serial interface to the constructor of our driver **(4.)**. Once the `Driver` instance has been constructed, it may be used through its API *e.g.* by calling the `Read Voltage.vi` **(5.)**. In addition to returning the voltage from the API call, the calls are written to the front panel indicator using our logging decorator.



2.6 Discussion

The example above shows how driving the design by tests, enabled by mocks, puts a pressure on the design. One might argue that two concrete classses, an interface, a mock, and a test case is overkill for the simple driver developent shown above. It is important to keep in mind that a real world application would not be considered done at this point and would grow in complexity considerably before being put into production. And then the importance of design increases significantly.

Already when we wanted to add logging to the driver, we were able to leverage the loose coupling between the driver and the actual hardware calls. Note that when I first started writing this example, I had no intention or plan to add logging, it was added as pure afterthought. Because of the loose coupling the new feature did not necessitate any changes to the existing driver, which is a charactheristic of good design.

Using test driven development with mocks tends to pull our concrete dependencies to the edges of our modules, as this makes testing simpler. Injecting dependencies, as shown above, allows for highly flexible architectures. The dependencies are declared and wired up on the topmost level, the main VI, and the classes are linked through abstract interfaces resulting in a weak coupling between the classses through the interface. By using factory methods or classes on the top level, we may change how the application is composed in run time and the concrete implementations may be maintained as plugins.

3. LMock Framework Architecture

The following introduces the design of the LMock framework. This is not necessary to understand at a deep level when using LMock, and is intended for curious readers and potential contributors. Details are omitted from this document, as they tend to change and would quickly get outdated. The best source of truth is to read the unit tests for LMock.

3.1 The Mock Class

The purpose of the mock class is to provide the boilerplate needed to track expectations, VI calls, and return values. The class maintains references to all configured expectations and a reference to a map of queues of return values.

The class has two protected methods. The `Mock Call.vi` is called in each dynamic dispatch method in the concrete mock class. This VI registers a call to the VI with the given input parameters, which is later forwarded to all configured expectations to determine if the expectation is fulfilled. The `Mock Call.vi` also returns a map of return values which the VI call should use. The mapping to the front panel controls and indicators is managed through scripting.

3.2 VI Call

The `VI Call.lvclass` is an interface that describes a call to a VI. Each call should have a VI Name and a map of input parameters. VI Calls are compared by Expectations using Comparators to determine the result of a test.

3.3 Comparators

Comparators inherit from the `Comparator.lvclass` interface and are orthogonal to Expectations. This implies that any Expectations should work with any comparator. The purpose of the comparator is to determine when two VI calls are to be considered as matching. Additionally, the comparator should provide a description of the comparison used in the test result description.

3.4 Expectations

An expectation is a class inheriting from the `Expectation.lvclass` interface. Expectations are and must be by-value classes.

The expectations use the visitor pattern to compare VI calls. This implies that the constructor of each expectation takes a comparator, which it should use for checking the VI calls. This enables reuse of the comparator and expectation code, as well as result message generation.

The `Expectation.lvclass` interface declares two methods, `Record Call.vi` and `Evaluate.vi`. The record call is called whenever a VI (any VI) is called on the mock. The LMock API convention states that the first VI call recorded by an expectation should define the expectation. Any subsequent calls are calls to be checked against this expected VI call using the configured comparator. Note that there is no way of knowing when the configuration ends and the actual test begins.

The `Evaluate.vi` is called by the `Verify.vi` method of the framework when the mock is verified and returns a pass/fail-result together with a description string. The expectation should keep track of all calls to it to be able to determine the result.

The expectation API is polymorphic, and each polymorphic variant instantiates an expectation with a different comparator.

4. License

4.1 Astemes LMock

4.2 Astemes LUnit