Testing VICI

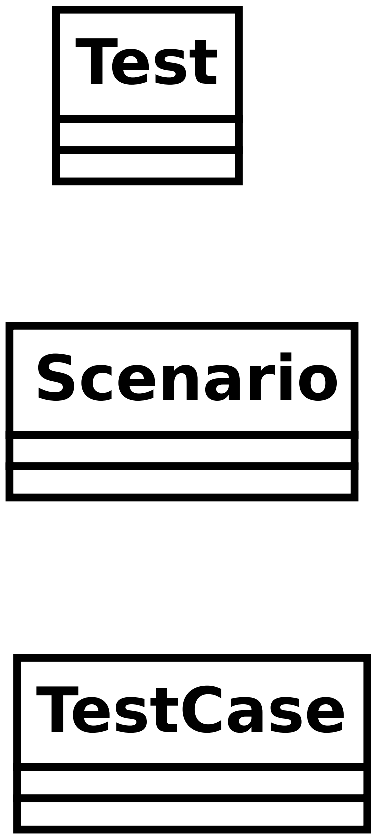
- Unit Tests
- Asynchronous Unit Tests
- Module Testing
- Program Testing
- GUI Program Testing

Unit Testing

Subdivide the Testing

- One Test Object
- Several Scenario Objects
- Many Test Case Objects

Test Classes



Unit Testing

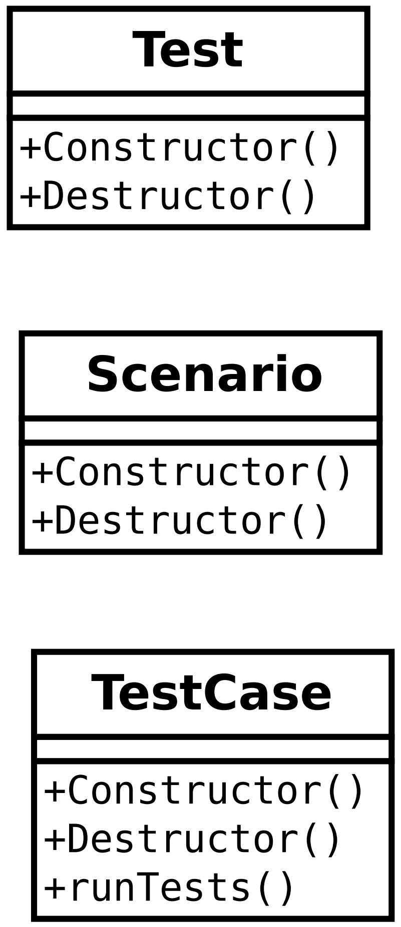
Resource Acquisition Is Initialization

The Constructor acquires the resources required for the test.

The body of the test case objects perform the tests.

The Destructor releases the resources.

Test Classes

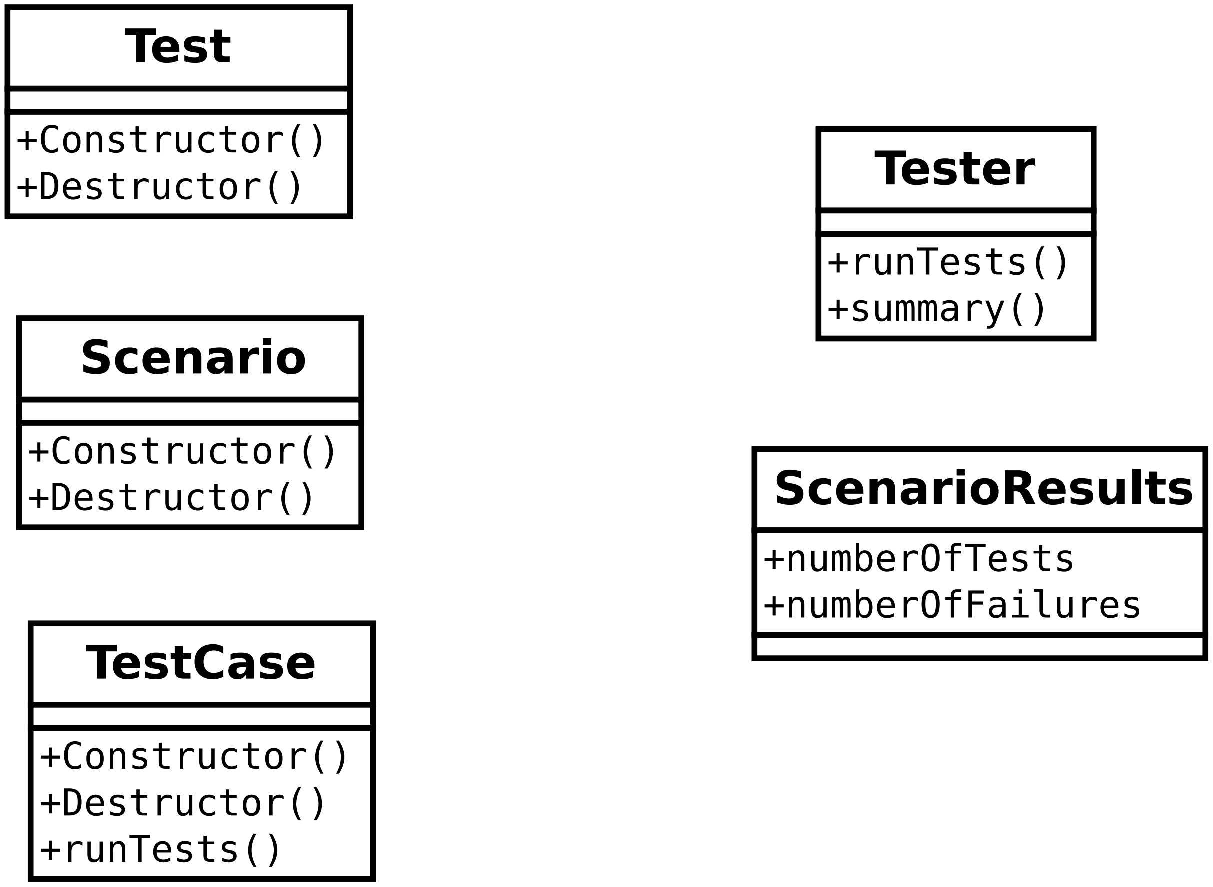


Unit Testing

Test Library

- Singleton Tester class manages the testing.
- Responsible for instantiating the test objects.
- Responsible for collecting and reporting the test results.

Test Classes



Unit Testing Library

Some Problems

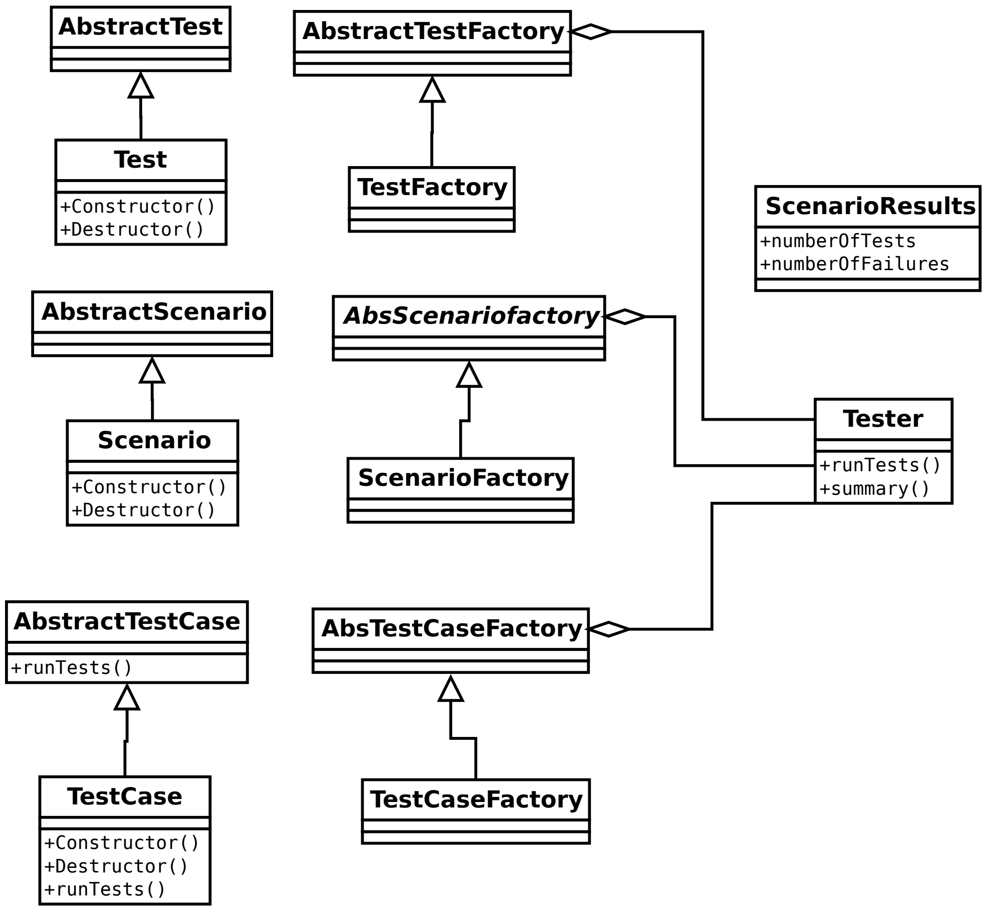
A reusable library cannot know about the test objects as they are specific to the test program.

The test program cannot create the test objects and pass them to the test harness since creating them would start them acquiring the resources.

Unit Testing Library

- Factory objects – responsible for creating the test objects.
- Testing library provides abstract versions that the test program derives from for the concrete scenarios and test cases.
- The factory objects can be created by the test program and handed to the Tester for safe keeping until they are needed.

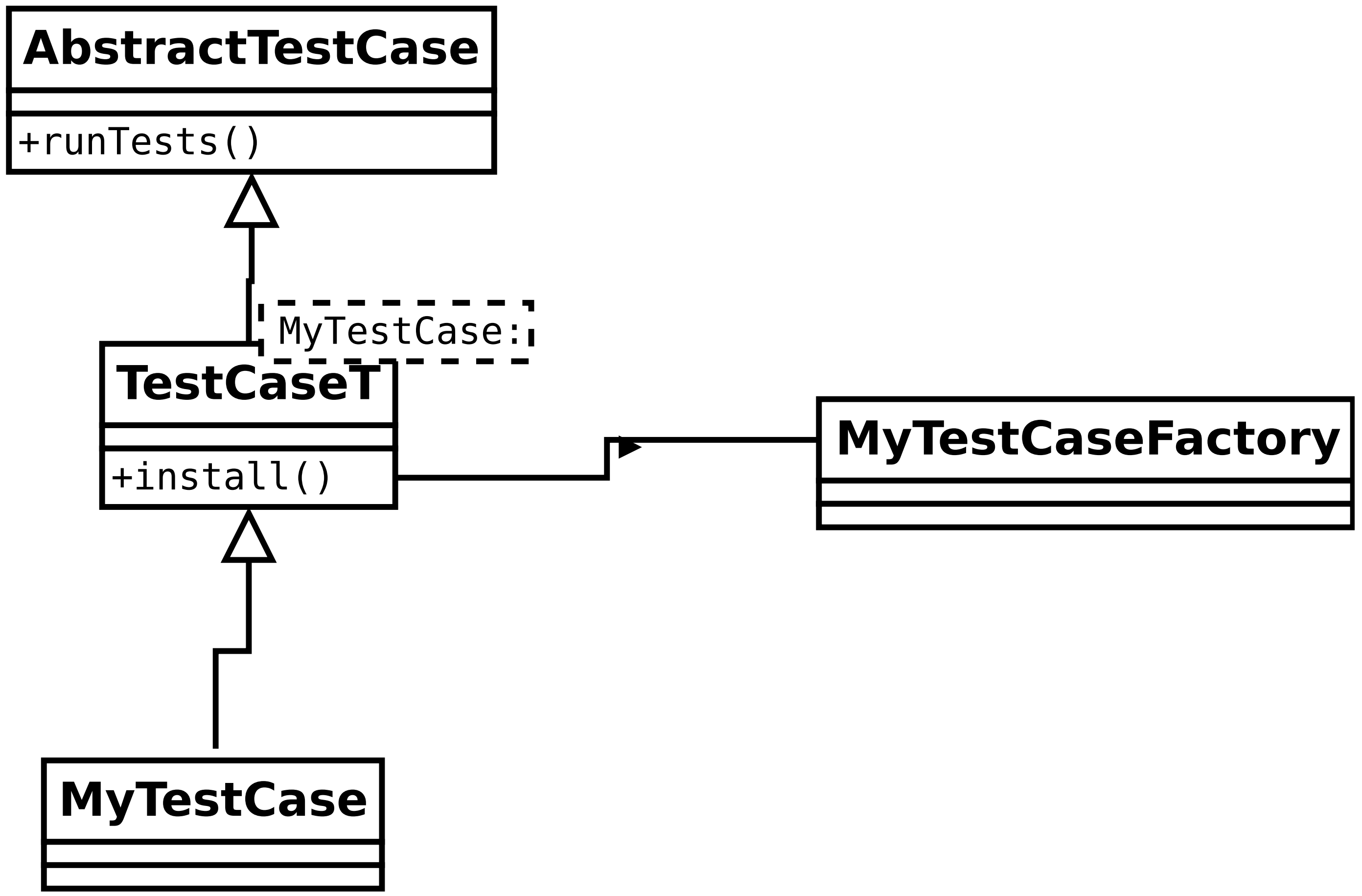
Test Classes



Unit Test Library

- Tedious to have to create factory classes for each scenario and test case.
- Templates used to automate the process.
- Use the “Curiously Recurring Template Pattern”.
- Create a static “install” method for each test case and scenario that creates the factory and passes it to the Tester

Test Classes



Asynchronous Testing

- VICI uses threads and separate processes.
- Functions just initiate something on another thread or process, and some callback function is notified when the results come in.
- Don't know how long it will take.
- Callback may be in a different thread.

Async Test Case

Body of test case divided into three functions:

- InitTest() - which triggers the processing.
- HandleEvent() - which responds to events when they occur.
- TimeOut() - when they don't.

Async Test Case

The function being tested needs to be instrumented to generate the events for the test harness.

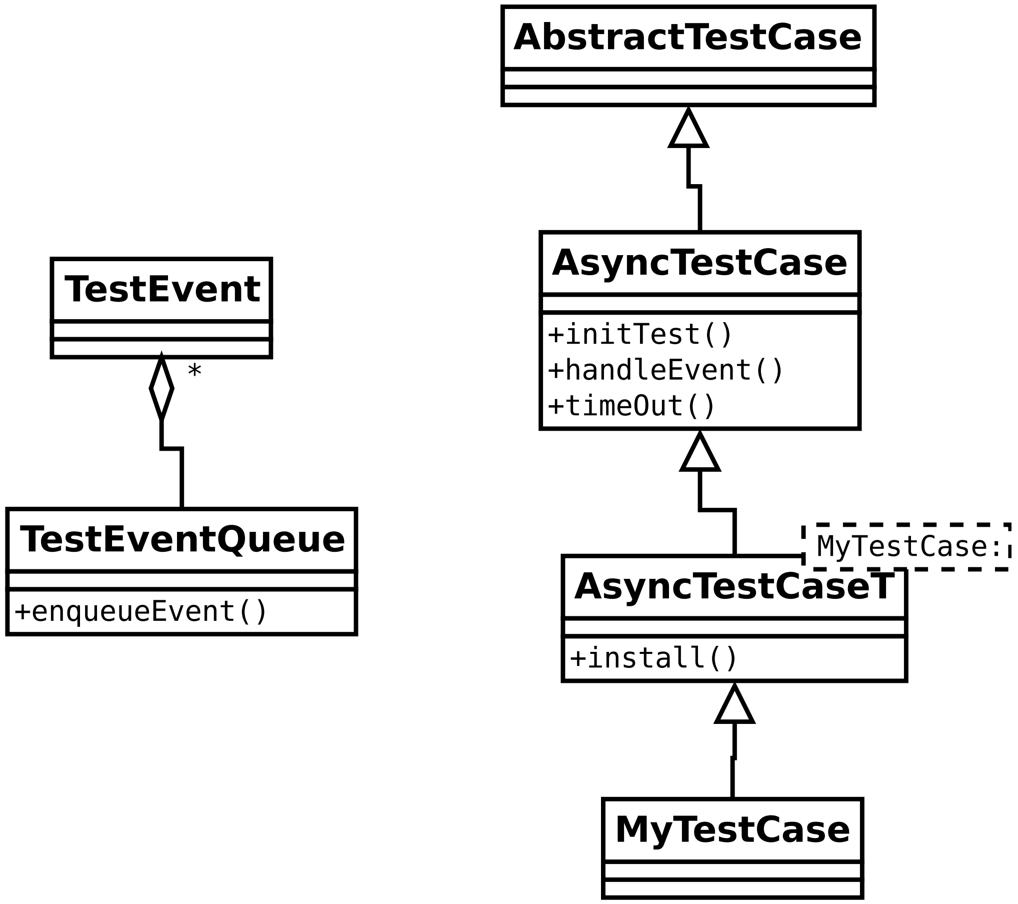
This is done by including a pointer to a function (*AsyncTestEventFn)()

This pointer is normally aimed at an empty function, but during testing it is reset to point to a function which queues an event and then waits.

The test harness runs a separate thread which waits for these events and calls the HandleEvent method in the current test case.

When the HandleEvent method completes the function is allowed to proceed.

Test Classes



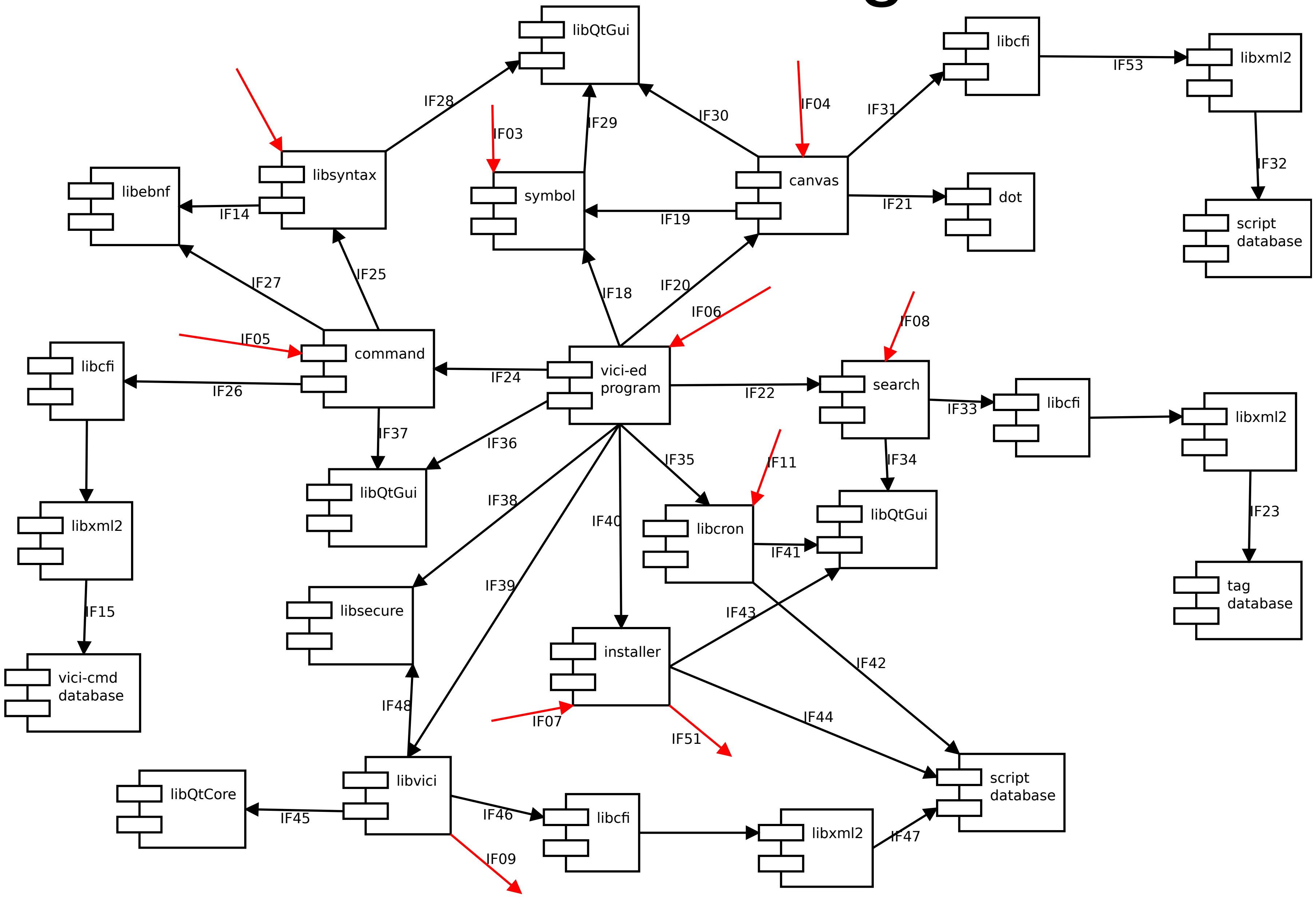
Module Testing

One of the aims of VICI is to demonstrate building large systems from independently developed components.

VICI has the following modules:

- Common Infrastructure and Testing modules.
- Interfaces and Configuration modules.
- Application modules.

Module Testing



Module Testing

There is a basic conflict -

- Want to develop modules independently.
- Modules depend on each other.

Module Testing

- Configuration library provides a FactoryFactory class that supplies Factory classes that provide the modules.
- Default Factories provide a stub library that is sufficient to compile.
- Searches the linked libraries for installed modules.
- Allows test code to substitute test versions of modules.

Program Testing

- Testing the completed programs.
- Ability to test internal state, not just external responses.
- Don't clutter up the program with test code that can be a vector for nefarious activities.
- Testing must be on the delivered program, not something that was compiled for testing.

Program Testing

Plug-in Libraries

VICI infrastructure includes support for plug-ins:

- Load and stay resident – for extensions to the program selected by the user or configuration.
- Load on demand – for functionality that is only needed briefly, such as a system handling hundreds of different types of forms.
- Autorun – load and run on start-up, and then unload. This is just what we need for testing.

GUI Program Testing

As for Program Testing with the additional requirement of simulating user interaction.

Need to have unchanged tests for a variety of distributions, desktop environments, monitor resolutions and languages.

This makes it impossible to simply record mouse positions and replay the clicks and key strokes.

GUI Testing

- Test harness will need to interact with the widgets either by simulating key strokes or directly calling their methods.
- Widgets are often private objects, inaccessible to the test code.
- Widgets may be created dynamically as the program executes.
- Hence the program needs to tell the test harness about the widgets.
- The test harness may not be loaded. Result is link errors.

GUI Testing

- Use an intermediary – WidgetMgr
- Windows register their existence with WidgetMgr
- Test harness queries it to get list of windows.
- Test harness queries each window for its widgets.
- Test harness passes a pointer to a function to the window to avoid link errors.

GUI Testing

Adaptors

- Each widget is given a type and a name.
- The type is used to select an Adaptor object.
- The Adaptor object adapts a command and a list of parameters to calls on the widget.
- The Adaptor provides a common interface for all widgets (and a few other objects).

GUI Testing

Scripting

- It can be fiddly getting the sequence of events set up correctly for a test.
- By reading the commands and parameters from a script file we avoid having to recompile the test code for every minor change.
- Original design used a bash script and could do arbitrary processing to provide intelligent control of the application.

GUI Testing

Qt runs a second event loop for modal dialogs.

This causes the test harness code to get stuck as the call to display the dialog doesn't return until the user interacts with dialog buttons.

GUI Testing

Event Driven Design

- Use focus change events to detect when a new window has focus.

- Start a new thread for each window. Doesn't matter if test code gets stuck – the new thread will be able to interact with new window.

GUI Testing

GUIs and threads don't mix.

You can only interact with a GUI from the thread that you started it in.

All of the Adaptor objects are being called from other threads.

GUI Testing

- Qt provides a “queued connection” that can make a function call from a thread to the GUI thread.
- A queued connection is asynchronous and cannot pass results back to the caller.
- Adaptor may have methods that need to run in the GUI thread and others that need to run in the calling thread.

GUI Testing

- Adaptor checks to see which thread its running in.
- If its the calling thread it does commands that get values from the widgets and returns results.
- If the command changes the widget the Adaptor makes a call to the GUI thread, passing its details.
- The Adaptor is re-run in the GUI thread.

GUI Testing

Scripting

- Can associate with the constructor and destructor of each test case a list of actions.
- Each action consists of:
  - A label.
  - A delay so that the tests don't happen too fast
  - A widget name
  - A command
  - A list of parameters for the command.

GUI Testing

Still need some way of having the testing respond to the program.

- Jump to a label.
- Variables
- LUA Scripting

GUI Testing

Label Jumps

- Each action may have a label.
- Result of action processed by up to two regular expressions (RE) each of which has an associated jump label.
- If the RE matches processing skips to the action with the corresponding jump label.

GUI Testing

Variables

- Each action may have a RE and a list of variable names.
- The RE is applied to the result of the action.
- The values of the sub-expressions are assigned to the names as variables.
- The parameters to other commands may refer to the variables.
- The test code may also refer to the variables.

GUI Scripting

Still need some way of doing more complex logic, such as performing an action based on the values of several widgets or what happened at some previous point in the testing.

- LUA can provide the logic.
- It is designed to be integrated into other programs.
- It handles the threads without issue.

GUI Scripting

- Each window can have a chunk of LUA code.

- The LUA scripts will be accessed via actions as if they were just another widget.

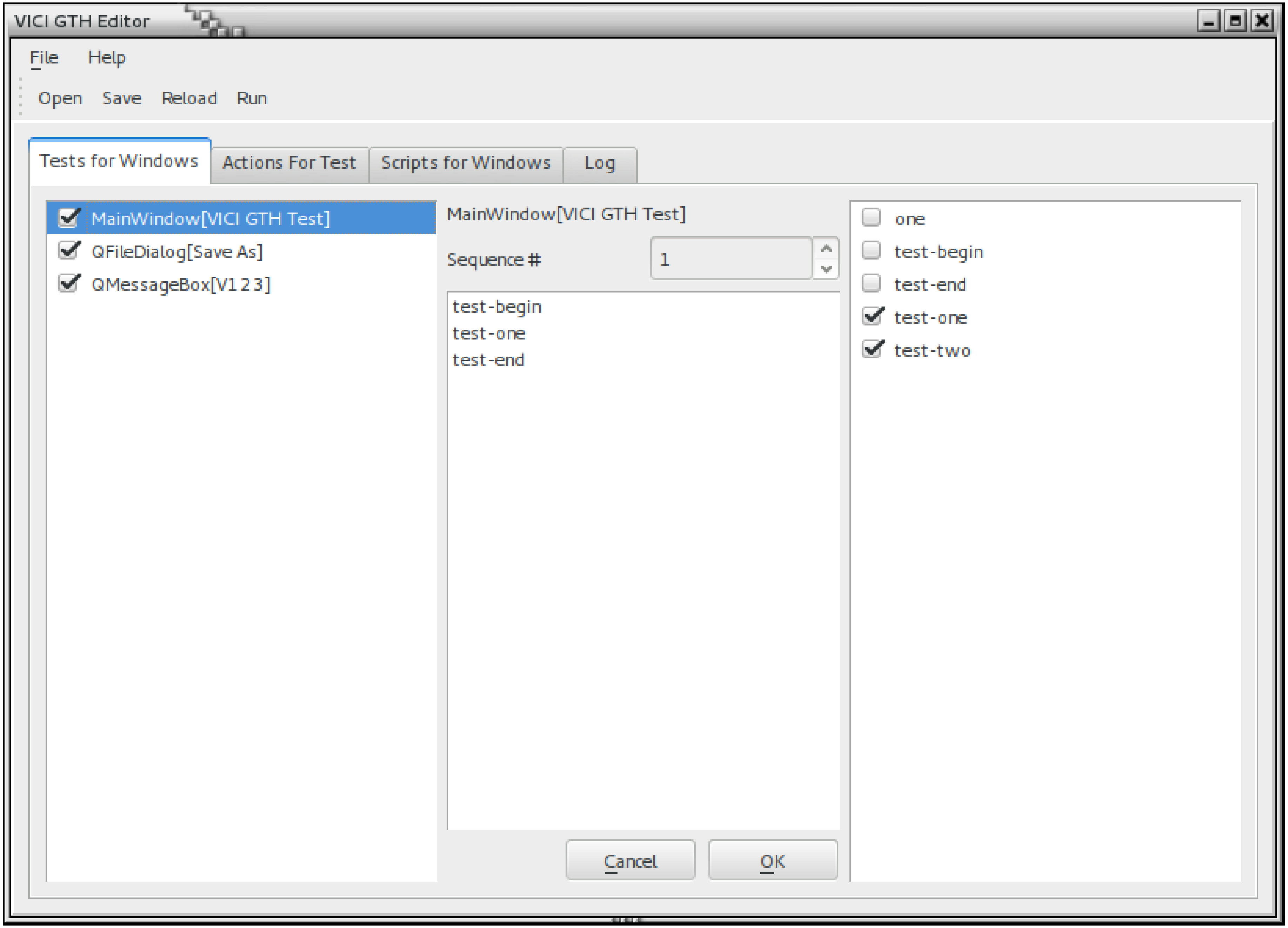
GUI Script Editor

The script is an XML file.

The test harness writes to the script file with details of the windows, widgets and adaptors.

A script editor:

- Assigns test cases to windows.
- Assigns actions to test cases.
- Creates the LUA script.



VICI GTH Editor

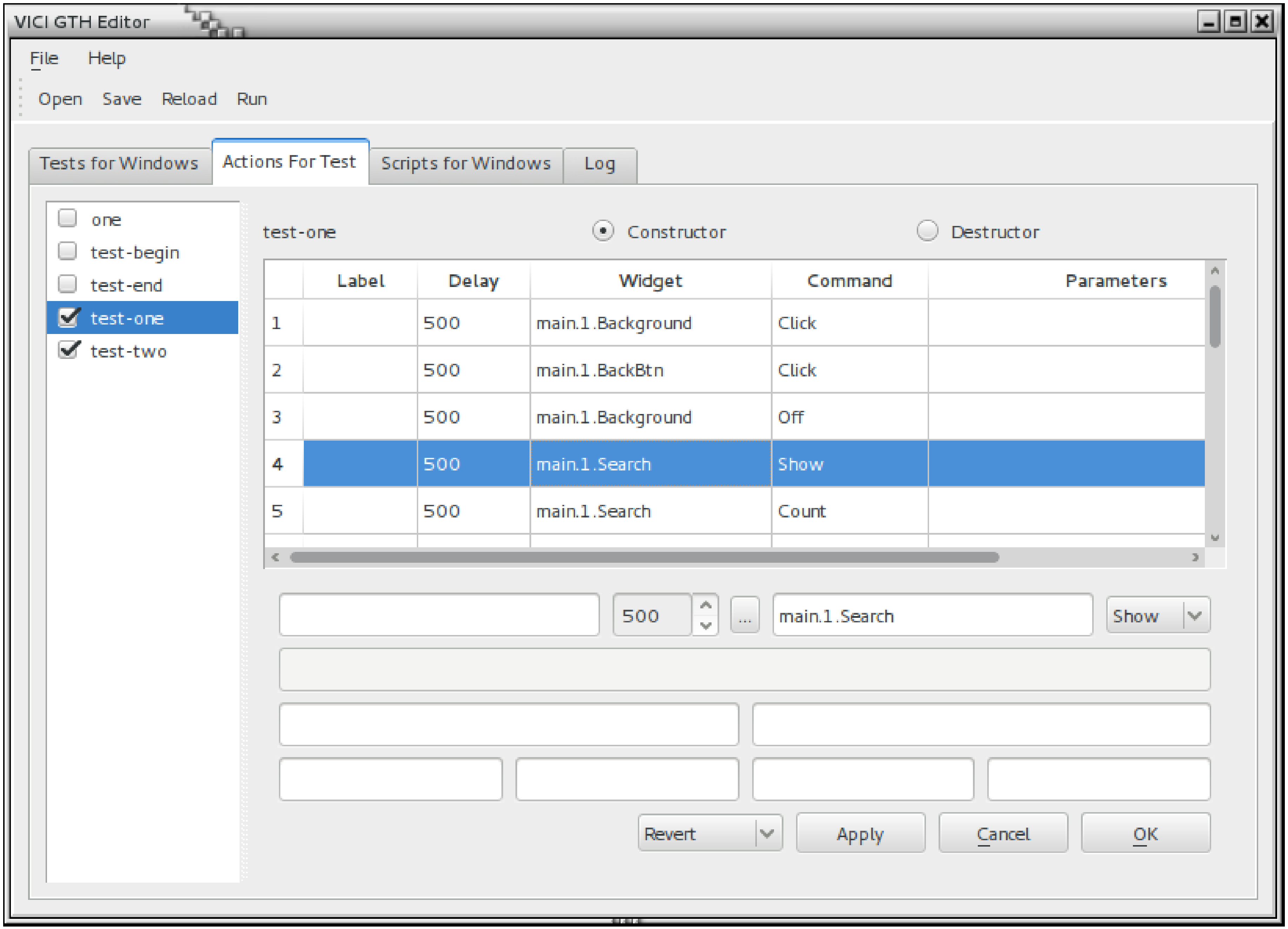
- [✓] MainWindow[VICI GTH Test]
- [✓] QFileDialog[Save As]
- [✓] QMessageBox[V1 2 3]

MainWindow[VICI GTH Test]

Sequence # 1

test-begin
test-one
test-end

- [ ] one
- [ ] test-begin
- [ ] test-end
- [✓] test-one
- [✓] test-two

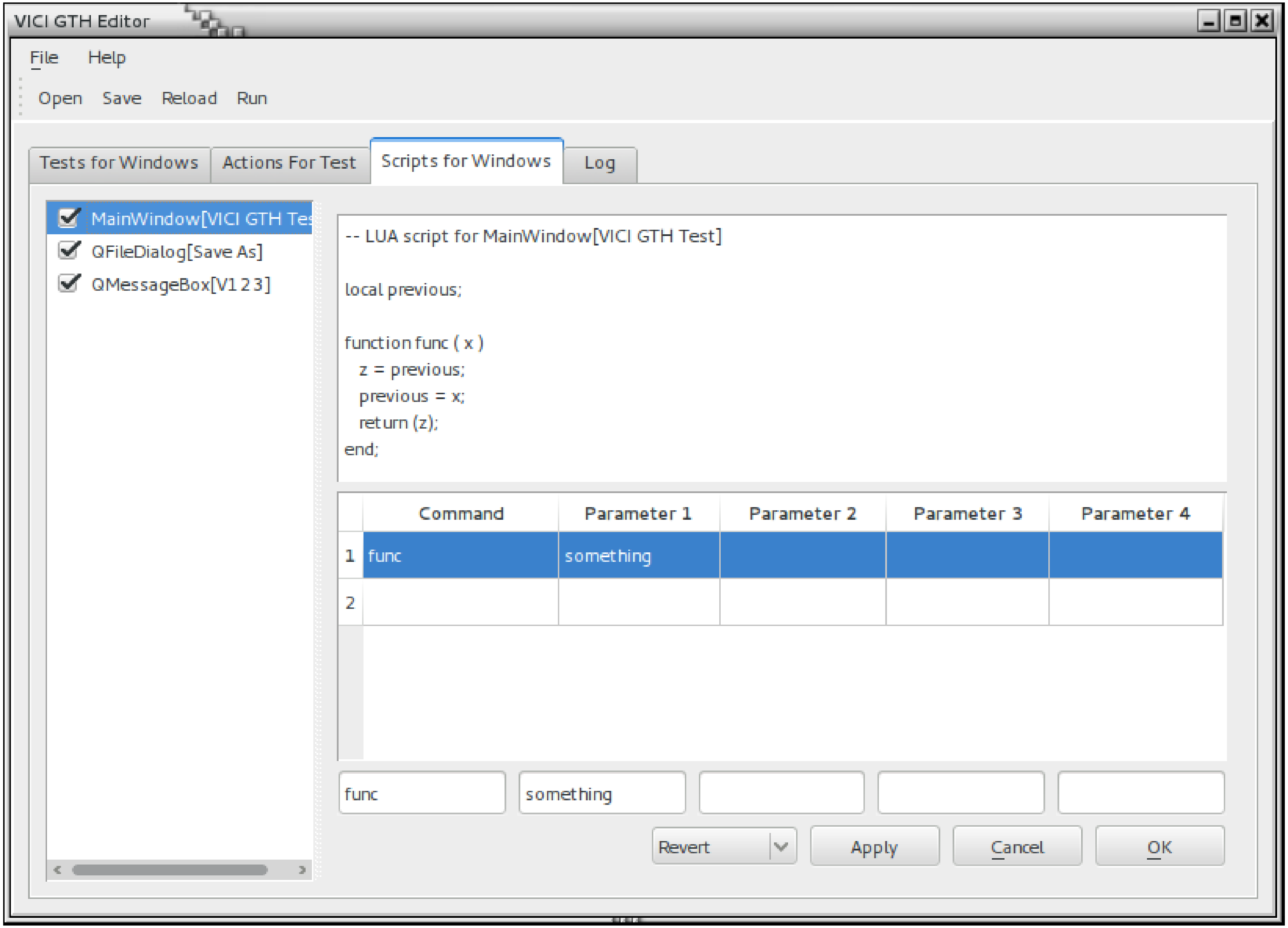


- [ ] one
- [ ] test-begin
- [ ] test-end
- [✓] test-one
- [✓] test-two

test-one [✓ Constructor] Destructor

	Label	Delay	Widget	Command	Parameters
1		500	main.1.Background	Click	
2		500	main.1.BackBtn	Click	
3		500	main.1.Background	Off	
4		500	main.1.Search	Show	
5		500	main.1.Search	Count	

500 main.1.Search Show



- ✓ MainWindow[VICI GTH Tes
- ✓ QFileDialog[Save As]
- ✓ QMessageBox[V1 2 3]

```
-- LUA script for MainWindow[VICI GTH Test]

local previous;

function func ( x )
   z = previous;
   previous = x;
   return (z);
end;
```

	Command	Parameter 1	Parameter 2	Parameter 3	Parameter 4
1	func	something			
2					

func | something