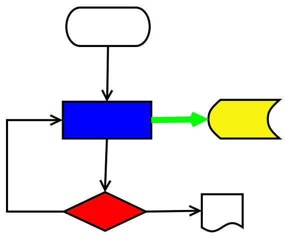
***VICI***



# VISUAL CHART INTERPRETER

Test Plan

## Publication History

| Date | Who | What Changes |
|---|---|---|
| 1 June 2014 | Brenton Ross | Initial version. |
| 20 June 2014 | Brenton Ross | Test Support Design |
| 17 January 2016 | Brenton Ross | Revised for new test harness design |
| 31 January 2016 | Brenton Ross | Added description of module testing support. |
| 4 February 2017 | Brenton Ross | Added new GUI test harness section. |

## Table of Contents

# 1 Introduction

This is the test plan for the VICI project.

## *1.1 Scope*

The test plan describes how the software will be tested.

The document covers unit tests of classes and components, system tests of the entire system, and integration tests where it is installed and used.

## *1.2 Audience*

The test plan is intended to be used by those responsible for testing the component.

# 2 Test Strategy

This is the high level design for testing the VICI software.

An important aspect of the modern testing methodology is that the testing should be automated as far as possible. Often it necessary to repeat the tests several times and if this is done manually then it becomes very easy to skip tests or fail to notice variations in the results. Thus the first task we have is to design the way in which the testing can be automated.

The unit testing forms the basis for all the testing. If the functions and classes are good then the higher level testing can concentrate on interfaces and other wider issues. The aim with all testing is to find problems – a successful test is one that finds a problem – which is a hard thing for a programmer to do. The testing needs to be automated, and preferably the test cases need to be set up before the code is written. Most unit testing can be done from test programs that include the object files for the main program.

The design for classes must provide the ability to test the functions. This may mean splitting out some functions from the constructor or destructor for example. The test case classes will also often need to be friends of the application classes so that testing can validate the internal state of the objects. Some classes will need additional methods defined to support testing.

The testing of each component will attempt to find problems with interfaces to the other modules. We will use the defined use cases (that are applicable to the module) and the module's responsibilities to define the test cases. We can extend the interface stubs to provide test cases to test the interfaces to other modules. For those modules that have a user interface we need a mechanism to activate the GUI controls from some form of script. Fortunately the Qt library includes methods for triggering widgets so this should be possible.

The final testing is to demonstrate that the system works in the environment that it is designed for. In the case of VICI we need to test that it works for a range of distributions and desktop environments. To show this we will run through the use cases that have been defined in a set of virtual machines set up for the tests. These tests can be automated using the same mechanism as the user interface tests for the modules.

The testing for the final programs will need some support code that is of no use otherwise. This can be handled by loading a shared library at run time and executing the test code therein if the program is run in test mode. Using this technique we can fully test the program in its final form with very little overhead imposed on normal usage.

To build useful tests it is often important to understand the flow of control within a program. This can be difficult in an object oriented program that uses mutliple libraries. A function call tracing capability will allow us to observe the flow of control.

## *2.1 Test Architecture*

### 2.1.1 Unit Tests

The **`make check`** build target will create and run a test program that runs the tests using the Tester and related classes and the object files for the application, or the library.

These tests will run test cases for all non-trivial methods with the aim of finding problems.

The test harness programs will be retained as a regression test library so that any future changes to the code can be made and tested to ensure no errors have been introduced.

### 2.1.2 Module Tests

For modules that are libraries there will be a test program that will instantiate test versions of the interface stubs and use those interfaces to set up the test cases and record the results.

For modules that are programs a test library will be dynamically loaded and the tests run when the program is run in test mode.

For components with a GUI the test harness will include the ability to run a script that will trigger and record the widgets.

Modules should be run with function tracing enabled so that the flow of control can be examined. The trc2dot program can be used to convert the trace log into the input of GraphViz dot program for the production of collaboration diagrams.

The tests will include all the uses cases specified for the module and all the responsibilities allocated to the module.

### 2.1.3 Program Tests

The programs will be tested using the GUI script test harness which will be dynamically loaded when the program is run in test mode.

The tests will cover all the use cases.

# 3 Unit Test Support Design

This section describes the code that will be used to support the unit testing. Three main components have been identified: A test manager for managing, recording and summarizing the test results, a tracing component for documenting the flow of control and a logging component for recording the results.

## 3.1 Logging

For a description of the logging classes please refer to the Programmer's Guide.

## 3.2 The Tracing Classes

The trace.h file defines some classes and macros to assist in tracing the behaviour of the system. The trace log messages have a higher time resolution than the normal log messages which can be useful in checking for timing related issues. The trace system provides a mechanism to filter the messages by debug level.

The “Trace” element in the configuration file should have two child elements, one to specify the name of the trace log, which should then be specified using a TraceLogger as described above; and a second to specify the default level of debugging.

A typical problem that arises with tracing is that too much or too little output is produced. This is addressed by setting a trace-level integer constant in each source file, typically such that its value indicates how deep the code is nested in the program structure. Trace output is then only generated if the default level, defined in the configuration file, is less (or equal) than the file's trace-level. The default level can also be set on a source file basis by setting a level in the configuration file for one or more source files – this can either enable or disable tracing for the file according to its value compared to the trace-level coded into the source file.

A typical tracing section of the configuration file:

```
<Trace>
      <logName>TRACE-LOG</logName>
      <default>10</default>
      <lex.cpp>5</lex.cpp>
</Trace>
```

### 3.2.1 The Trace Class

The Trace class is used, typically, to log messages indicating the progress of the execution and the paths taken. It uses stream semantics to enable easy message construction.

A macro form is provided so that the source file name and line number can be included automatically. It is used as follows:

```
TRACE(TraceLvl) << "we are here";
```

Another macro, _VICI_TRACING, is used to enable or disable the code generation associated with this class. However, there remains a small overhead even when disabled, so its best to comment out these trace statements when they are not in use.

### 3.2.2 The CallTrace Class

The CallTrace class is used to trace the call graph of the executing program. A macro form is also supplied and the normal use is place the call at the start of each function or method:

```
void AClass::AMethod ()
{
      FN_TRACE( TraceLvl, "AClass::AMethod" );
      ...
}
```

The destructor for the CallTrace object automatically traces the end of the function.

The _VICI_FN_TRACING macro is used to enable or disable the code generation associated with this class. When disabled there is no overhead, so these can safely be left in the program, and this is encouraged.

### 3.2.3 The trc2dot Program

The output of the CallTrace class can be processed by a program called trc2dot, and then by the GraphViz dot program to create a collaboration diagram showing the call graph of the test.

This program expects that the text that was passed to FN_TRACE was a class name and method name separated by "::".

A script, trace.xsl, can be used to turn an SVG version of the collaboration diagram into an animated version.

## 3.3 The Test Manager

The testmgr.h file defines a singleton Tester class. This object manages the testing process.

A test run consists of an overall Test in which there are Scenarios and within these Test Cases. The intention is that each of these will use its constructor to set up test conditions and perform a clean up in its destructor. Each one should leave the system in the state that it found it.

The Test Cases run a test method that records the results of the test against its enclosing Scenario.

Since these objects must be created (and destroyed) during the test run the Tester holds a set of factory objects for constructing them as they are needed. These factory objects are created by a template parent class using a static method called “install”.

Default versions of the Scenario and Test classes are provided and installed automatically if the testing doesn't need its own version of them.

The Tester object gets its configuration from the XINI configuration based on a name passed to its constructor. This name should be an element name within “//Test”, and in turn should have at least a child element called “Report” with a child element “filename” containing the path for the test report.

### 3.3.1 Asynchronous Tests

There are quite a few functions within Vici that are asynchronous in nature. The call to the function may return immediately, but it initiates some additional action that fulfils the purpose of the function. In order to test these we cannot simply make a call and then check the results.

A common approach is to sleep for a while and then test to see if the results have been achieved. This has two problems: how long to wait is indeterminate, and it may be necessary to have the test performed at a very specific moment while various temporary variables remain in scope.

The AsyncTestCase solves this problem  We add a macro to the code under test that expands to a function that queues an event and then pauses the code. The test object can then perform the required tests while the code under test is in a well defined state. The macro can expand to an empty statement when the testing has been completed.

### 3.3.2 Graphical Tests

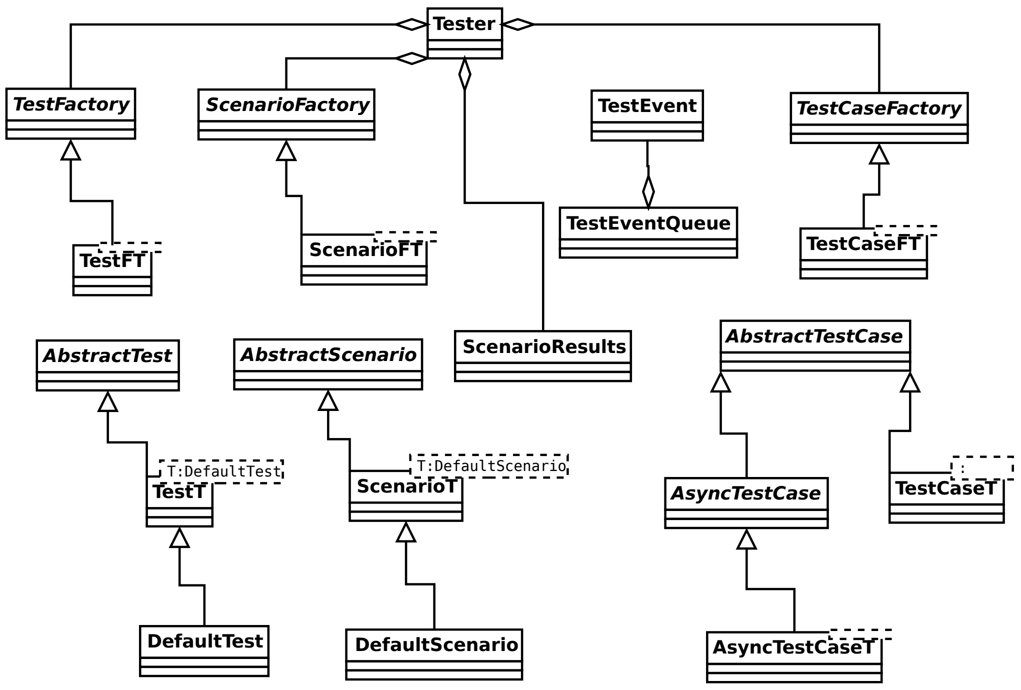
The libgth.h file defines an extension to the Test class that executes a script and uses its output to trigger events within widgets. The constructors and destructors for the Test, Scenario and Test Case objects can call on functions in the script which applies commands to the widgets.

A set of helper objects, Adaptors, are used to interpret the commands, one per type of widget. These helper objects have a link to a particular widget and are associated with a name that precedes the command from the script.

The only part of this which is required in the code under test is a function to register the widgets, and this is only called when the program is being tested.

## 3.4 Test Support Classes

The following diagram shows the relationships between the classes used to support testing.



### 3.4.1 Tester Class

This is a singleton object that manages the testing. It has factories for constructing the test objects and accumulates the test results for each scenario.

```
class Tester
{
protected:
      Tester();
      std::string nameOfTest;
      std::string logName;
      bool failed;
      void title();
      void installDefaults();
      std::map< std::string, std::map< std::string, TestCaseFactory *> > tests;
      std::map< std::string, ScenarioFactory * > scenarios;
      std::map< std::string, ScenarioResults * > results;
      TestFactory *theTest;
      AbstractTest *absTest;
      void testScenario( csr scenario, ScenarioFactory *scn );
public:
      void configure ( csr testName );
      csr getTestName() { return nameOfTest; }
      static Tester & instance();
      virtual ~Tester(){}
      cfi::logstream & log();
      void addTestCase( csr name, csr scenario, TestCaseFactory *tcf );
      void addScenario( csr scenario, ScenarioFactory *sf);
      void setTest( TestFactory *t) { theTest = t; }
      AbstractTest * getTest() { return absTest; }
      virtual void runTests();
      bool summary();
};
```

### 3.4.2 TestFactory Class

This is an abstract type used by Tester to create a Test object when the testing begins.

```
class TestFactory
{
public:
      virtual ~TestFactory(){}
      virtual AbstractTest * make() = 0;
};
```

### 3.4.3 TestFT Template

This template is instantiated for a specific test class and is responsible for constructing the test object. It is created automatically by the install method of the TestT template.

```
template < class T >
class TestFT : public TestFactory
{
public:
	AbstractTest * make() { return new T; }
};
```

### 3.4.4 AbstractTest Class

This is an abstract base type for the test classes.

```
class AbstractTest
{
public:
	virtual ~AbstractTest(){}
};
```

### 3.4.5 TestT Template

This template is instantiated for a specific test class which is also derived from it. An instance of the “Curiously Recurring Template Pattern”.

```
template < class T >
class TestT : public AbstractTest
{
public:
	static void install()
	{
		Tester::instance().setTest(new TestFT<T> );
	}
};
```

### 3.4.6 DefaultTest Class

For simple test there may not be a need for a special Test object. If no Test object is installed, then this default one will be installed prior to running the tests.

```
class DefaultTest : public TestT<DefaultTest>
{
public:
	DefaultTest(){}
	~DefaultTest(){}
};
```

### 3.4.7 ScenarioFactory Class

This is an abstract type used Tester to hold a list of scenario factories.

```
class ScenarioFactory
{
public:
	ScenarioFactory(){}
	virtual ~ScenarioFactory(){}
	virtual AbstractScenario * make() = 0;
};
```

### 3.4.8 ScenarioFT Template

This template is instantiated for a specific scenario and is responsible for constructing the scenario object. It is created automatically by the install method of the ScenarioT template.

```
template < class T >
class ScenarioFT : public ScenarioFactory
{
public:
	AbstractScenario * make() { return new T; }
};
```

### 3.4.9 ScenarioResults Structure

The tester holds a set of these for accumulating the test results prior to printing the summary.

```
struct ScenarioResults
{
	int mNumTests;
	int mNumErrors;
	bool failure;
	ScenarioResults(): mNumTests(0), mNumErrors(0), failure(false) {}
};
```

### 3.4.10 AbstractScenario Class

This is an abstract base type for the scenario objects.

```
class AbstractScenario
{
public:
	virtual ~AbstractScenario(){}
};
```

### 3.4.11 ScenarioT Template

This template is instantiated for a specific scenario class which is also derived from it.

```
template < class T >
class ScenarioT : public AbstractScenario
{
public:
	static void install( csr nm )
	{
		Tester::instance().addScenario(nm, new ScenarioFT<T>);
	}
};
```

### 3.4.12 DefaultScenario Class

For simple tests there may not be a need for a special Scenario object. If no Scenario object is installed, then this default one will be installed prior to running the tests.

```
class DefaultScenario : public ScenarioT<DefaultScenario>
{
public:
	DefaultScenario(){}
	~DefaultScenario(){}
};
```

### 3.4.13 TestCaseFactory Class

This is an abstract base type used by the Tester to hold the tests case factories.

```
class TestCaseFactory
{
public:
	virtual ~TestCaseFactory(){}
	virtual AbstractTestCase * make(csr nm) = 0;
};
```

### 3.4.14 TestCaseFT Template

This template is instantiated for a specific test case and is responsible for constructing the TestCase object. It is created automatically by the install method of the TestCaseT template.

```
template < class T >
class TestCaseFT : public TestCaseFactory
{
public:
	AbstractTestCase * make(csr nm) { return new T(nm); }
};
```

### 3.4.15 AbstractTestCase Class

This is an abstract base type for the test case objects. The “expected” parameter of test() should only be used when testing the test harness itself.

```
class AbstractTestCase
{
protected:
	std::string name;
	bool test( bool cond, csr testMsg, bool expected = false );
	virtual bool runTest() = 0;
	ScenarioResults *scenario;	// reference
public:
	explicit AbstractTestCase(csr nm) : name(nm), scenario(0) {}
	virtual ~AbstractTestCase(){}
	virtual bool operator()(ScenarioResults *);
	static const bool EXPECTED = true;
};
```

### 3.4.16 TestCaseT Template

This template is instantiated for a specific test case class which is also derived from it.

```
template < class T >
class TestCaseT : public AbstractTestCase
{
public:
	TestCaseT(csr nm) : AbstractTestCase(nm) {}
	static void install( csr nm, csr scn )
	{
		Tester::instance().addTestCase(nm, scn, new TestCaseFT<T> );
	}
};
```

### 3.4.17 AsyncTestCase Class

This is the base type for test cases that handle queued events.

```
class AsyncTestCase : public AbstractTestCase
{
protected:
	virtual void initTest() = 0;
	virtual void handleEvent( TestEvent *ev ) = 0;
	virtual void timedOut() = 0;
	bool done;
	std::chrono::steady_clock::time_point startTime;
	std::chrono::steady_clock::duration timeOut;
public:
	AsyncTestCase(csr name);
};
```

### 3.4.18 AsyncTestCaseT Template

This template is instantiated for a specific test case class which is also derived from it.

```
template < class T >
class AsyncTestCaseT : public AsyncTestCase
{
public:
	AsyncTestCaseT(csr nm) : AsyncTestCase(nm) {}
	static void install( csr nm, csr scn )
	{
		Tester::instance().addTestCase(nm, scn, new TestCaseFT<T> );
	}
};
```

### 3.4.19 TestEvent Class

The global VICI::event() method creates these and places them on the test event queue.

```
class TestEvent
{
public:
	TestEvent( const std::string & s) : mId(s) {}
	const std::string & id() { return mId; }
};
```

### 3.4.20 TestEventQueue Class

Test events are queued here until handled by an AsyncTestCase.

```
class TestEventQueue
{
protected:
	TestEventQueue();

public:
	std::mutex eventMx;
	std::list< TestEvent * > events;
	std::set< std::string > runningEvents;
	std::mutex functionsMx;
	std::condition_variable eventCV;
public:
	static TestEventQueue & instance();
	void event( csr id );
	void enqueueEvent( TestEvent * );
};
```

# 4 GUI Test Harness

The GUI test harness is responsible for simulating user interactions with GUI programs.

## 4.1 Analysis

There are two common approaches to this problem. One is to simulate the mouse movements and clicks and keyboard events. In practice this doesn't work very well as GUIs tend to change the size and shape of the widgets according to the language, version, Desktop theme, etc. The more stable approach is to have the tests call methods on the widgets.

## 4.2 Architecture

I would like to be able to reuse VICI's test harness for driving the tests. It provides well tested code for structured testing as well as a standard report format.

I don't want the test harness and test cases to be built into the program as that is likely to be a vector for nefarious activity – test code often has various short-cuts that may make it vulnerable to attack.

This leads to demand loaded plug-in libraries to hold the test code. The program can be placed into test mode causing it to load and run the tests. VICI's libcfi library has support for this.

Setting up GUI tests can be rather fiddly and if you have to do a compile after each change it can be frustrating and time consuming. The solution is to have a script that is interpreted by the test harness.

Minor changes can thus be made and immediately tried out.

Thus we have a high level design consisting of a test harness that is dynamically loaded and which interprets a script to trigger methods on the widgets.

## 4.3 Detailed Design

### 4.3.1 Registering Widgets

The first problem is that the test harness needs to call methods on the widgets. Since we are building a general purpose library that will be used in several programs it is necessary for the programs to tell the test harness about the widgets. However, the test harness may not be loaded so we cannot directly reference the test harness from the program code either.

One option is to use a global function that must be implemented by each program. The function is called by the test harness when it starts and passes a pointer to a function that registers the widgets. The program then uses this function to tell the test harness about each widget - a pointer, its type, and a name that will be used by the script.

This solution has a couple of problems - global functions are best avoided, and it doesn't handle the case where the widgets are dynamically constructed (and destroyed) at later points in the program's execution.

A better idea is to have a singleton object act as the intermediary (WidgetMgr class). Windows will register themselves with it and the test harness can also add itself as a client. When a window registers itself that is passed to all the clients and when a client adds itself it is told about all the windows. The test harness can then ask the window to register its widgets using the function pointer. (The function pointer allows the code to link without the actual function being loaded.)

This leads to the next problem. Qt's window widgets know nothing of VICI so I will need to wrap them in a VICI class that has the code for interfacing to the WidgetMgr. Normally the registration would be done in the constructor, but at that point the window probably won't have instantiated its widgets. The registration has to be done after the widgets have been added. This can be automated by overriding the showEvent() method which is called when a widget is about to be displayed.

### 4.3.2 Threads for Windows

The next problem concerns the way Qt runs dialogs. This mostly concerns the QMessageBox, but applies to all dialogs. They can be run two different ways - the exec() method which does not return until the dialog is closed, or the open() method which returns immediately.

Normally exec() is used with dialogs that are created on the stack, while open() is for dialogs that are created on the heap. The exec() method returns the result of the dialog, while open() requires setting up connections to additional functions for getting the results.

Obviously using exec() is far more convenient for reporting errors to the user so it would be annoying to have to avoid it just to accommodate automated testing. However, since it doesn't return until its finished it makes it difficult for the test harness to interact with it!

The test harness will hook into Qt's QApplication class to get events that indicate a change of focus. If the active window has changed to a new window then the currently running test case will be stopped and a new thread started that will run the test cases associated with the window.

If the active window has changed to a currently paused window the currently running test cases will be either paused or abandoned depending on the state of the window. The paused test cases will then be resumed.

The test cases will check a flag to see if they should pause or abandon, and wait on a condition variable until notified if paused.

### 4.3.3 Actions

The actions to perform on the widgets will be associated with the constructor and destructor of each test case.

Each action will have a syntax something like:

```
optional-label: delay: widget-name command param ... :
result-assignment : regex label, regex label, ... ;
```

The optional-label allows us to conditionally skip some actions.

The delay is a time in milliseconds to wait. This gives the test a more realistic performance.

The "widget-name command params" is passed to the interpreter for each widget type (as is already implemented in GTH). The action will return a string - for example the current label on a button, or the text of a message box.

The "result-assignment" will use regular expression sub-expression matches to assign values to variables. These can then be used by the body of the test case to test for expected values.

The "regex label" pairs check the returned string and jump to the action with the label, thus providing conditional (or even looping) processing.

The widget name is used to lookup the address of the widget and its widget type from the information stored when the application registered the details of its widgets.

The widget type is used to get an Adaptor object. Much of the Adaptor code is done with templates as they are all essentially very similar. However each one has its own "describe" and "action" methods. The describe method is used to provide descriptions of the commands for use in the gth-editor. Each action method is a mini-interpreter that takes the command and parameters and does things to the widget.

The commands in an Adaptor can either change the state of the widget, or get some property of the widget. (I am hopeful we will never need a command that does both at once.) The commands that get a property simply extract the value and return it as a string. However, we are currently running a function that was originally called by a thread object, and it is not possible to cause a widget to change its visual state from anything other than the main GUI thread.

Fortunately Qt provides a "queued connection" that can effectively make a function call between threads. If the action method needs to change a widget it passes the widget name, command and parameters to a method called "jumpThread" which sends to a function that runs in the main thread, which goes through the whole process of finding an Adaptor and executing the command, but this time it is in the main GUI thread and can update the state of widgets.

At first it may seem odd that the Adaptor is run in both threads – it would be more efficient to just run it once? The problem is that the queued connection does not provide a way of returning the state information. Thus the update methods have to run in GUI thread while the access methods have to run in the test window thread.

### 4.3.4 Smart Processing

**Label Jumps.**

Let's suppose there is a label widget that has a status message, such as "Success" or "Fail" depending on what happened in the program. We can use an action to interrogate the widget to get its label text. Let's also suppose we don't want to perform the next few actions if the text included the word "Fail". We add to the action a regular expression (RE), probably just "[Ff]ail" and an associated jump label that is the label for an action further on in the action list for the test case.

The action processing applies the RE to the result string and if it matches it adjusts its pointer to point to the action with corresponding label.

**Variables**

The second use for regular expressions is for extracting values to be stored into variables.

Let's suppose a message box is displayed and we want to use part of the text displayed to insert into another widget later. We use an action to get the text from the message box. Then we use a RE with sub-expressions to pick out the values we want to save.

For example, lets say the text we get contains a fragment of XML:

"XML tag: <tag-name>the value</tag-name>."

We can set up a regular expression to search this:

"<(.*)>(.*)</(\\1)>"

The parentheses define the sub expressions. This one says that the first and last sub-expressions must be the same (so we get a valid element from the XML).

The action processing collects up the contents that match each sub-expression ("tag-name", "the value", "tag-name") and assigns then to variables which are named in the action.

These values can then be retrieved and used in other actions anywhere in the test scenario.

**LUA Script**

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.

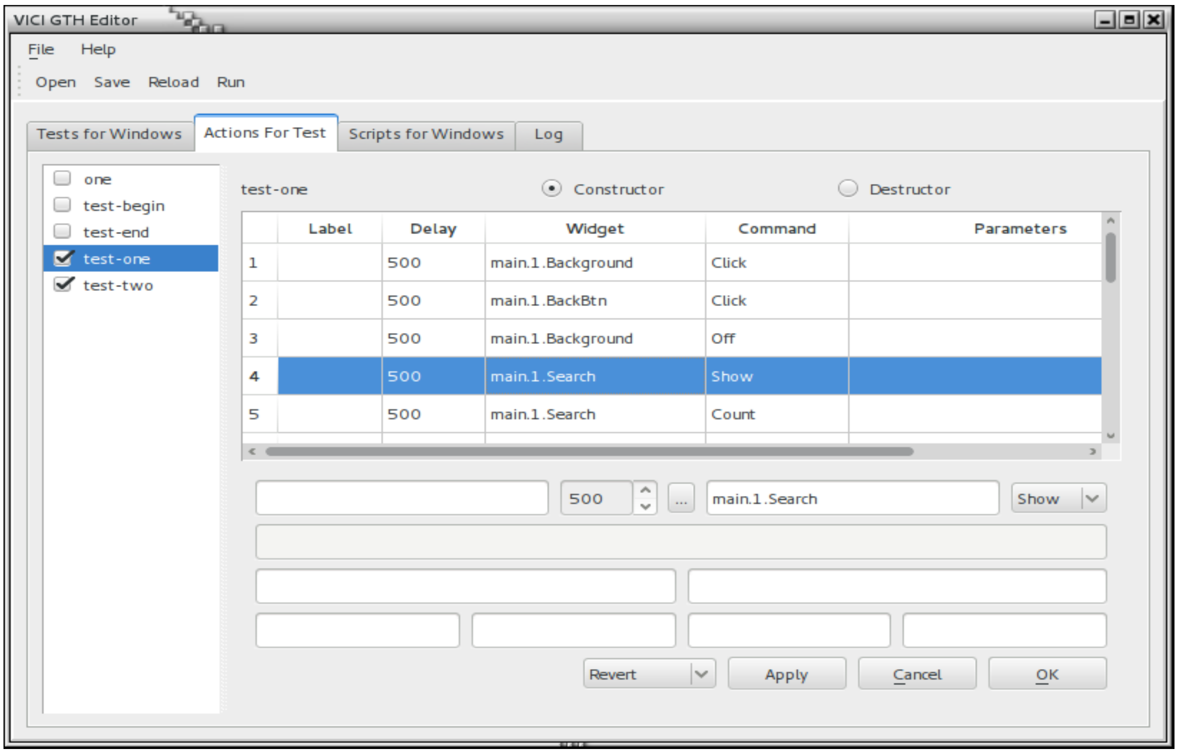
- LUA can provide the logic.
- It is designed to be integrated into other programs.
- It handles the threads without issue.
- Each window can have a chunk of LUA code.
- The LUA scripts will be accessed via actions as if they were just another widget.

### 4.3.5 XML Script

The XML script contains actions that are to be applied to the GUI during the constructor and destructor of each test case.

The script is initialised by the test harness which stores the widget types and the actions that can be applied to each, as well as the registered widgets.

The script can be edited using the gth-editor program to assign test cases to windows, actions to test cases and LUA script to windows.



☐ one
☐ test-begin
☐ test-end
☑ test-one
☑ test-two

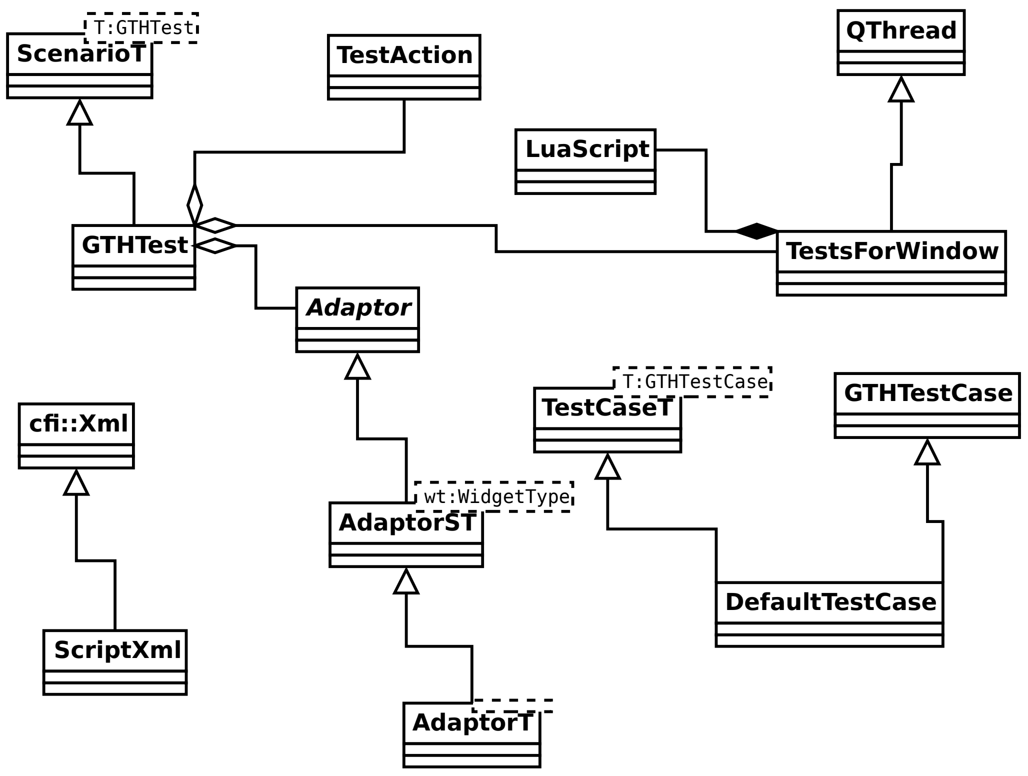
test-one (● Constructor) (○ Destructor)

| | 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

## 4.4 Class Design

The following diagram shows the static relationships between the classes of the GUI Test Harness.



### 4.4.1 GTHTest Class

This is a specialisation of the Scenario class that loads and executes a script for interacting with GUI widgets.

The GTHTest class takes over the task of selecting the next test case to run from Tester. This is necessary since the order of the test cases is determined by which window has focus.

The class runs its own event loop which runs until the tests for the main window have been completed.

The class is also responsible for processing the output of the commands to set variables and jump over some actions.

```
class GTHTest : public VICI::cdi::TestT<GTHTest>
{
public:
  GTHTest();
  ~GTHTest();
  csr getName();
  virtual bool willRunTests();
  virtual void runTests(csr scenarioName, cdi::ScenarioResults *);
  cdi::ScenarioResults *results;
  void endMainWindow();
  void runActions( csr testCase, ConDes );
  void assignVariables( const TestAction &, csr result );
  std::string getJump( const TestAction &, csr result);
  csr getVar( csr varName ) const;
  void substVars( const ParamList &, ParamList &);
  virtual void windowHasRegistered( GTHWindowWidget * );
  virtual void windowHasDeregistered( GTHWindowWidget * );
  void regWidget( void *w, VICI::gth::WidgetType t, csr n);
};
```

### 4.4.2 GTHTestCase Mixin Class

This class should be inherited by test cases that need to run a GUI script.

Its constructor and destructor call GTHTest::runActions.

```
class GTHTestCase
{
protected:
      GTHTestCase(csr name);
      ~GTHTestCase()
};
```

### 4.4.3 DefaultTestCase

This test case is installed for tests that only have test actions but no actual test case object.

### 4.4.4 Adaptor Class

This is an abstract type that wraps widgets to provide a simple command interpreter.

```
class Adaptor
{
public:
  virtual ~Adaptor(){}
  virtual std::string action( csr cmnd, const std::vector< std::string >  &params ) = 0;
  virtual VICI::gth::WidgetType getType() = 0;
};
```

### 4.4.5 AdaptorST Template

This provides the static method for describing the adaptor. This description is used by the gth-editor.

### 4.4.6 AdaptorT Template

This template provides the common constructor for the adaptors.

```
template < VICI::gth::WidgetType wt, class T >
class AdaptorT : public Adaptor
{
private:
      T *w;
      static const VICI::gth::WidgetType wType = wt;
public:
      AdaptorT( void *p) : w( reinterpret_cast<T *>(p)) {}
      VICI::gth::WidgetType getType() { return wType; }
      virtual std::string action( csr cmnd, const std::vector< std::string >
& params );
};
```

### 4.4.7 TestAction Structure

This holds the details of each action. It is stored by the GTHTest class in a map indexed by the test case name and constructor/destructor.

### 4.4.8 TestsForWindow Class

This object is derived from QThread and holds the test cases for each window in the GUI.

### 4.4.9 LuaScript Class

This is responsible for running the Lua interpreter on the chunk of code that may be associated with a window.

### 4.4.10 ScriptXml Class

This is responsible for interfacing to the XML script file. It has methods for getting test details and for storing widget and window details.

# 5 Module Test Support Design

This section describes the design of the components that support the testing of complete modules.

The support consists of two components, a stubs library that provides a minimal implementation of each module that can be linked into the test version of the module to simulate the remainder of the system, and a configuration library that has its build controlled by configure options so that it generates test or stub versions of modules.

## 5.1 The Stubs Library

The header file, vici.h, provides the declarations for the classes that constitute the public APIs of all the modules that make up the project. This is effectively the foundations of the design that shapes the entire suite of VICI applications.

The stubs library, libstubs.so, provides a minimal implementation of the classes defined in vici.h. An additional library, libcc.so, provides an implementation of those classes which are used to pass information between modules – these are called the “currency classes” since they are passed around like money.

Each class has a corresponding factory class that is used to construct the objects.

## 5.2 The Config Library

There is also the master FactoryFactory class that is responsible for instantiating the factory classes. This enables us to provide different implementations depending on whether we want the real object, a test harness stub,or just a minimal stub.

The config library, libconfig.so, provides an implementation (and associated header file) that can be built in different configurations for testing each module, a sub-system, or the entire suite for testing or production.