GUI Unit Testing (TUG) Framework

Testing Library and Project Wizard for Qt Panels

Pedro Mateo
pedromateo@um.es
Cátedra SAES

November 27, 2015

Contents

1 Motivation

TUG — "GUI Unit Testing" — "Testing Unitario GUI", in Spanish.

TUG project [1] was born with the main purpose of providing a unit testing framework for graphical user interfaces. The main goal was providing developers with a method to easily create a battery of tests for Qt-based applications. The tests had to simulate, as far as possible, users interaction with the interface.

With this purpose, the TUG project is divided into two main components:

- **TUG Wizard**: a wizard-like application that helps developers to create and configure, step by step, a test project aimed at testing a Qt based panel as well as the underlying model and communication classes (if they exists). It generates a new panel inheriting the original one. This new panel includes customized methods to simulate users interaction with the widgets composing the panel. It can also generate a full, standalone test project including testsuites and empty test methods ready for being filled with testing code.
- **TUG Base Library**: a library aimed at supporting the tests generated by TUG Wizard, as well as test projects created manually by developers. It provides a way to structure test suites, as well as a set of methods to support the definition of GUI tests, all around the Qt Test framework.

Along this document it is described how to properly use these components to test your Qt-based projects.

[1] The TUG Project is an initiative of Cátedra SAES (http://www.catedrasaes.org) funded by the SAES company (http://www.electronica-submarina.com). This project and all its components have been designed and developed at University of Murcia (Spain).

2 Package Content

When uncompressing the TUG package you can find the following folders and files:

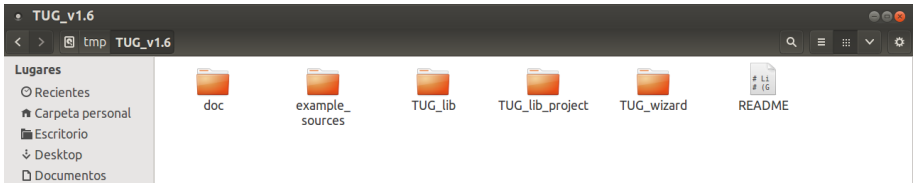
- `doc`: this folder includes this guide.
- `libTUG_project`: this folder includes the C++ project of libTUG (GUI Unit Testing) library. Go to Section 4 for further information.
- `TUG_wizard`: this folder includes the wizard application to create test projects (requires Java RE). Go to Section 3 for further information.
- `install.sh`: this file can be used to install libTUG into a specific directory. Installation includes also the documentation and a tools folder including TUG Wizard. Type "`./install.sh -help`" to show further information. Maybe you need to provide execution permissions as follows: "`sudo chmod a+x install.sh`".
- `README`: includes brief information about the package content.

3 TUG Wizard Usage Guide

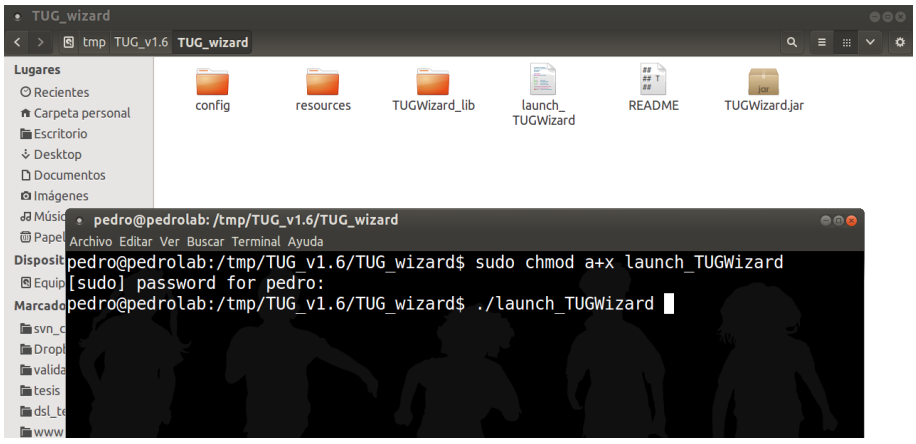
3.1 Running TUG Wizard

This section describes how to deploy TUG Wizard. Please, follow the steps described in the following.

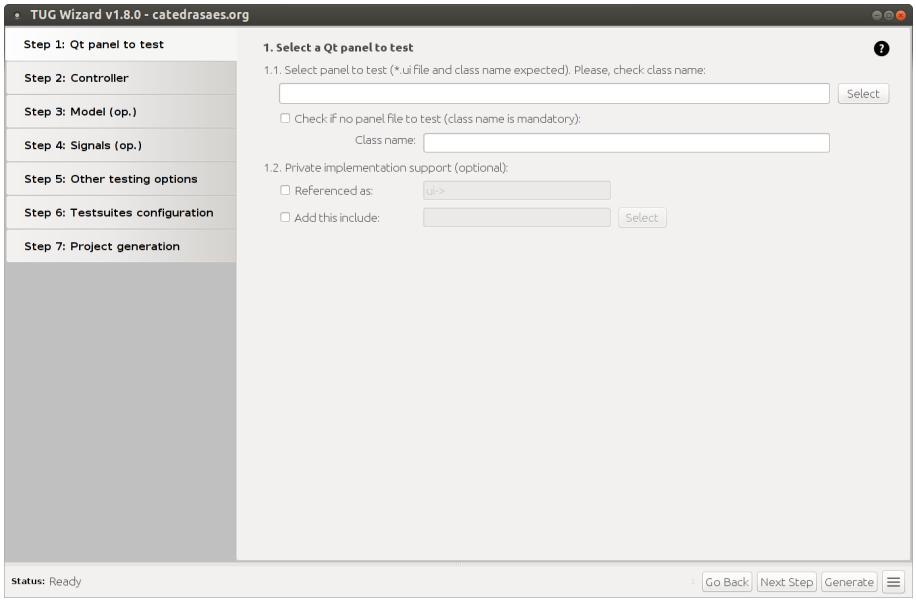
1. **Step 1: Unpack TUG package.**
   `TUG Wizard` as well as `TUGLib` are packed into a file named `TUG_vXX.tar.gz`. Unpack this package into a folder. As a result you will get a set of files as depicted in the figure below.



2. **Step 2: Launch TUG Wizard.**
   Go to `TUG_Wizard` folder.
   Add execution permission to the script file `launch_TUGWizard` as depicted in the figure below.
   Execute `launch_TUGWizard` to launch the wizard prompt.



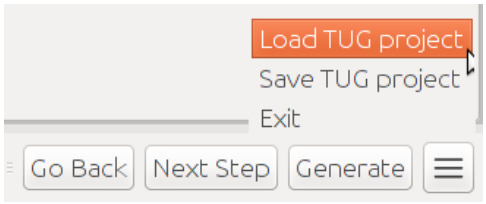
3. **Step 3: TUG Wizard ready.**
   After the execution of the two steps described above, TUG Wizard will be prompted to start the configuration and generation of a testing project, as described in next section.

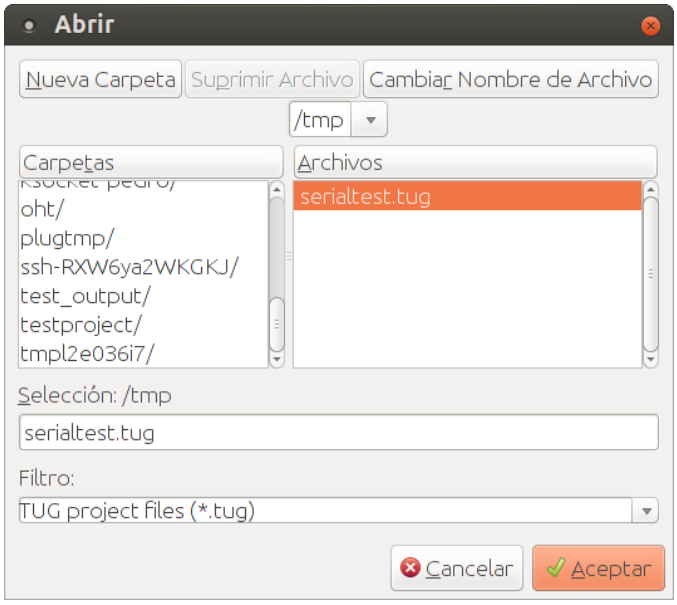


3.2 Test Projects in TUG Wizard

TUG Wizard uses test projects that can be loaded from and saved to files with extension `*.tug`. Please, follow the steps described in the following to load/save a TUG test project.

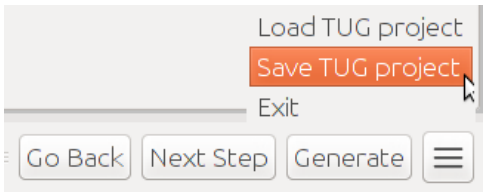
1. **Load a TUG project.**
   Click ☰ to deploy the pop-up menu and select `Load TUG project`. Select a `*.tug` file and click `OK`. Data will be restored into TUG Wizard fields.

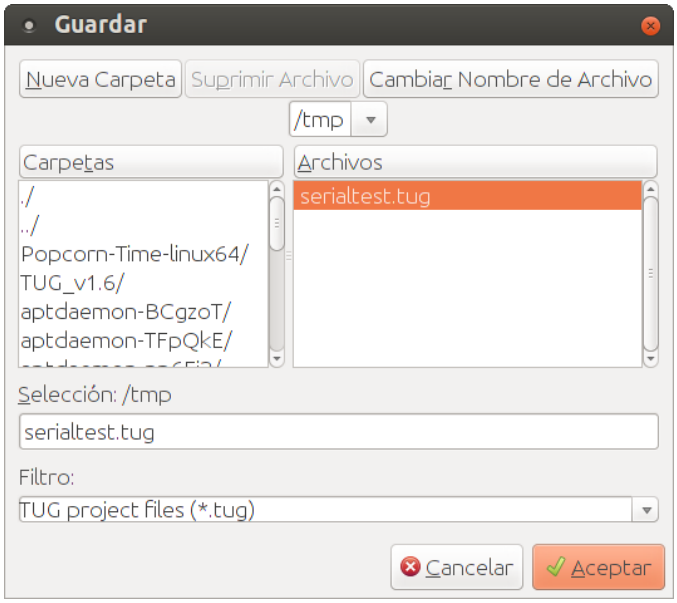




2. **Save a TUG project.**

   Click ☰ to deploy the pop-up menu and select `Save TUG project`.
   Select a `*.tug` file or write a new name for the project. Click `OK`.
   Data will be restored into TUG Wizard fields.

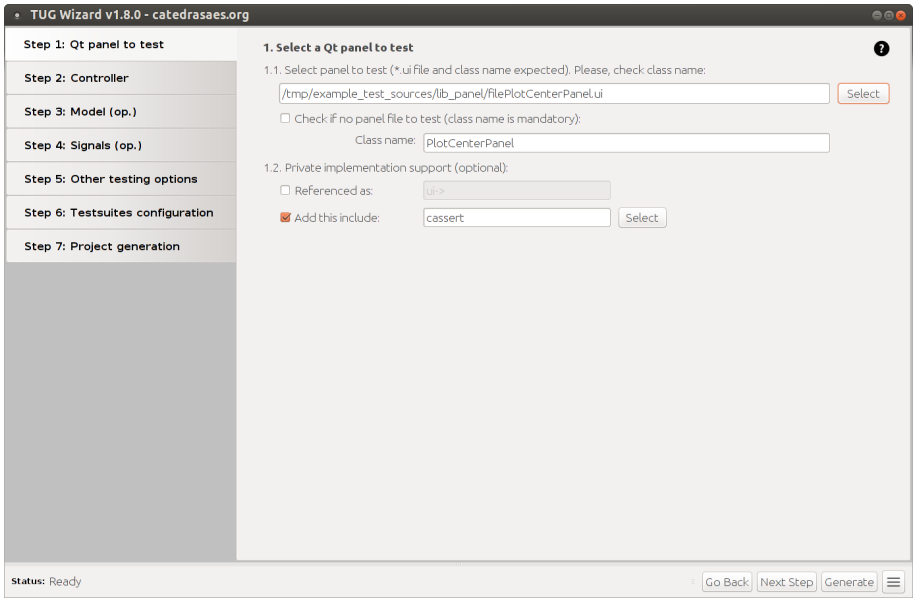




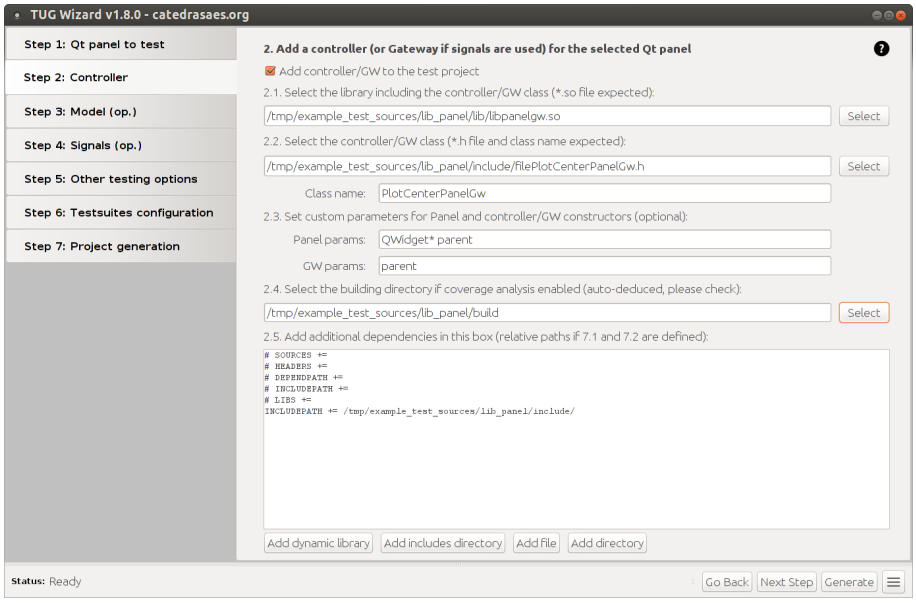
3.3 Test Project Generation

Creating a project to test a Qt-based GUI is really easy by using TUG Wizard. Please, follow the steps described in the following.

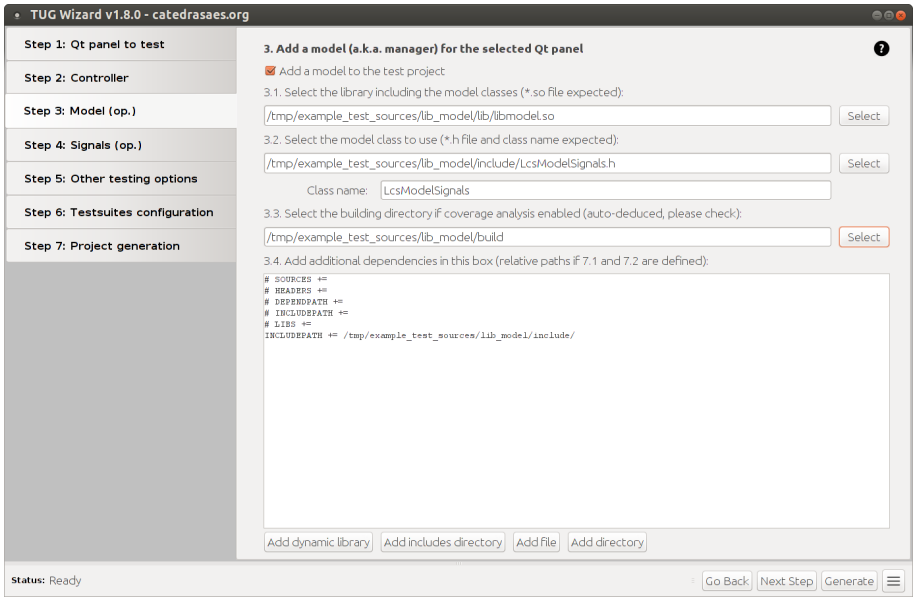
1. **Step 1: Select a panel.**
   Panel is the window to test.
   Click `1.1 Select` to select a Qt *.ui file. `Class name` will be auto-filled based on the panel name found in the selected file. Please, check it.
   A test project can be also generated without a *.ui file. For this, select `There is no panel file to test` and fill `Class name` manually with a name for the panel.
   Within a Qt GUI, widgets can be referenced in many different ways (e.g., by using a private implementation). By default, widgets are referenced using `ui->`. It can be changed selecting `Referenced as` and filling this field.
   After this change, a new include could be needed. If so, select `Add this include` and click `1.2 Select` to select a file to include.
   Click `Next Step` at the bottom of the wizard.



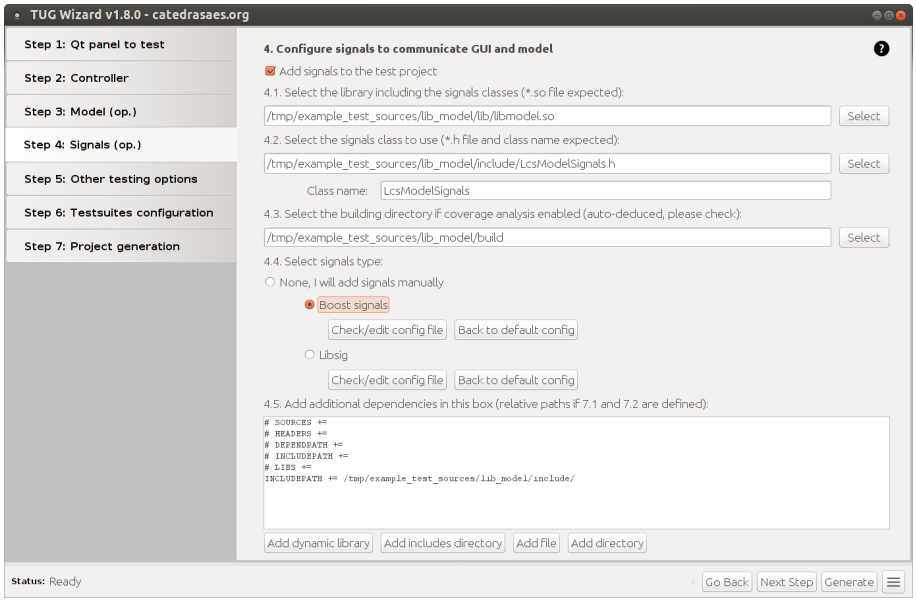
2. **Step 2: Select Gateway class.**
   Gateway class (GW) is the class implementing communication with the manager (M).
   Click `2.1 Select` to choose the library containing the GW class.
   Click `2.2 Select` to select the file defining the GW class. `Class name` will be auto-filled based on the class name found in the selected file. Please, check it.
   Click `2.3 Select` to select the directory in which GW library is built. It is needed if coverage analysis is enabled.
   In `2.4` you can add additional dependencies for the GW library. There are some buttons to help you adding dependencies: `Add dynamic library` and `Add includes directory` include `qmake` tags; `Add file` and `Add directory` can be used to add new paths.
   Click `Next Step` at the bottom of the wizard.



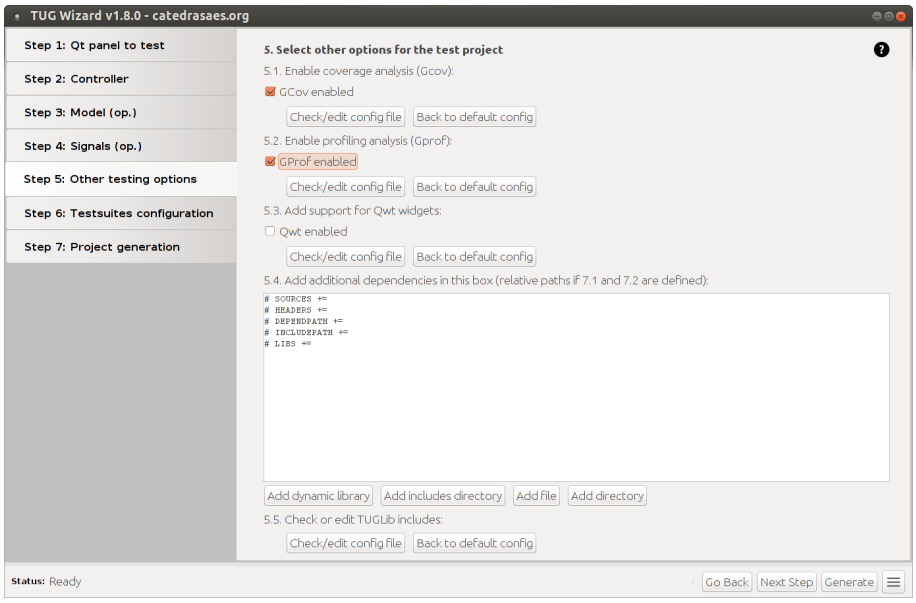
3. **Step 3: Select a Manager/model (optional).**
   You can select a manager class (M) with which the panel is connected.
   Check `Include a manager into the testsuites` to enable this option.
   Click `3.1 Select` to choose the library containing the M class.
   Click `3.2 Select` to select the file defining the M class. `Class name` will be auto-filled based on the class name found in the selected file. Please, check it.
   Click `3.3 Select` to select the directory in which M library is built. It is needed if coverage analysis is enabled.
   In `3.4` you can add additional dependencies for the M library.
   Click `Next Step` at the bottom of the wizard.



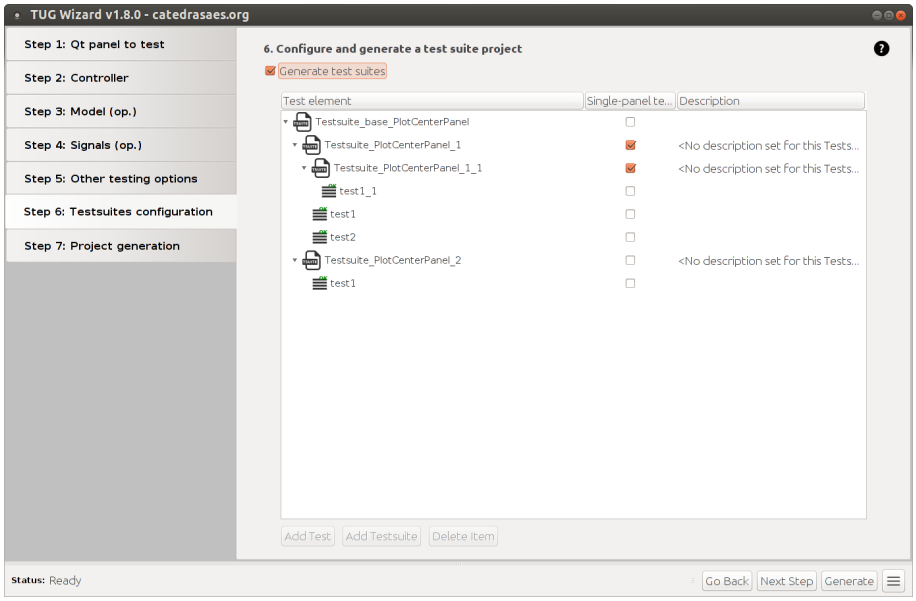
4. **Step 4: Select a Signals class (optional).**
   You can select a class including those signals (S) used for the communication between GW and M.
   Check `Include a signals class into the testsuites` to enable this option.
   Click `4.1 Select` to choose the library containing the S class.
   Click `4.2 Select` to select the file defining the S class. `Class name` will be auto-filled based on the class name found in the selected file. Please, check it.
   Click `4.3 Select` to select the directory in which S library is built. It is needed if coverage analysis is enabled.
   In `4.4` you can select if the signals used are based on `Boost` or on `Libsig`. You can select `None` to add it manually. `Check/edit config file` can be used to check and edit the dependencies to be incorporated into the test project. `Back to default config` can be used to go back to default configuration of such dependencies.
   In `4.5` you can add additional dependencies for the S library.
   Click `Next Step` at the bottom of the wizard.



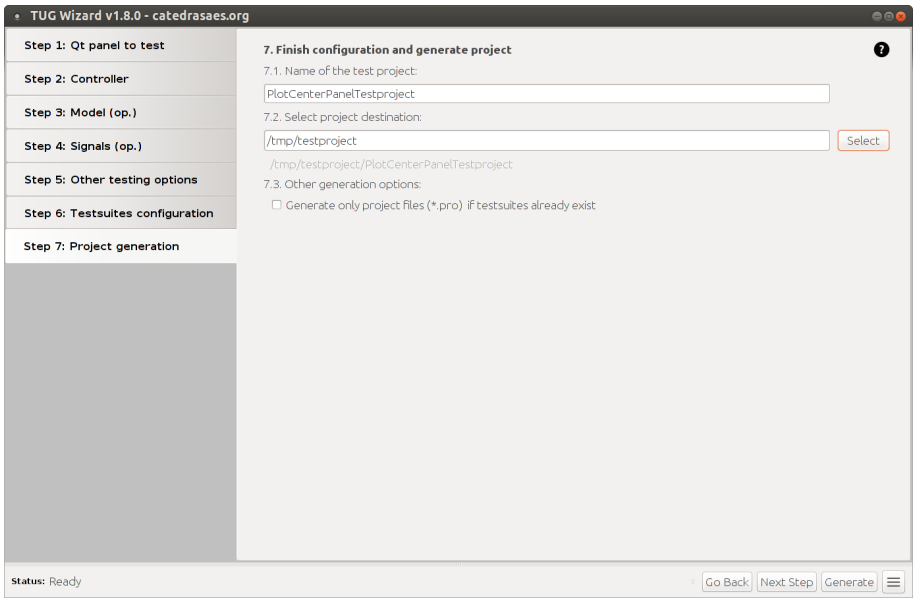
5. **Step 5: Select other options for the test project.**
   In this step we can choose and configure options like coverage and profiling analysis, as well as add additional dependencies to the project. In this step we can also check the includes for **TUGLib**, the library running under test projects generated by TUG Wizard.
   Check `GCov enabled` to enable coverage analysis.
   Check `GProf enabled` to enable profiling.
   Check `Qwt enabled` to enable support for Qwt widgets.
   `Check/edit config file` can be used to check and edit the dependencies to be incorporated into the test project for each option. `Back to default config` can be used to go back to default configuration of such dependencies.
   In `5.4` you can add additional dependencies to the project.
   In `5.5` you can check, edit, and restore dependencies related to **TUGLib** library.
   Click `Next Step` at the bottom of the wizard.



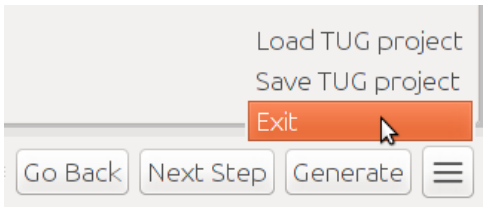
6. **Step 6: Configure a testsuites structure (recommended).**
   In this step we can create the structure of a test project. As a result after the generation process, a set of testsuites will be auto-generated according to the test project configured in this step.
   Check `Generate test suites` to enable this option. A pre-configured test project is provided as starting point. A test project is composed of a base testsuite (often used to configure and clean the test scenario) and a set of child testsuites that include some extra configuration (if needed) and the tests to be executed. We can modify this test project as follows.
   Click `Add Test` to add a new test to a testsuite. Please, note that every testsuite (except base testsuite) must include, at least, one test.
   Click `Add Testsuite` to add a new child testsuite to a parent testsuite. Please, note that only level-2 testsuites are allowed (level-0 is base testsuite).
   Click `Delete Item` to delete selected item (either a test or a testsuite).
   Once the test project is created, click `Next Step` at the bottom of the wizard.
   TUG Wizard implements **roundtrip** between different versions of a test project. If a project is generated in the same directory it was generated before, then the code of old tests will be kept in the new version if they still exist.


TUG Wizard v1.8.0 - catedrasaes.org
Step 1: Qt panel to test
Step 2: Controller
Step 3: Model (op.)
Step 4: Signals (op.)
Step 5: Other testing options
Step 6: Testsuites configuration
Step 7: Project generation
6. Configure and generate a test suite project
Generate test suites
Test element
Single-panel te...
Description
Testsuite_base_PlotCenterPanel
Testsuite_PlotCenterPanel_1
<No description set for this Tests...
Testsuite_PlotCenterPanel_1_1
<No description set for this Tests...
test1_1
test1
test2
Testsuite_PlotCenterPanel_2
<No description set for this Tests...
test1
Add Test
Add Testsuite
Delete Item
Status: Ready
Go Back
Next Step
Generate

7. **Step 7: Name the project and select a destination.**
   In `7.1` we have to add a name for our project. This name will be used to create a folder into which generate the project. It will be also used to identify the project within the output report.
   Click `7.2 Select` to select a directory into which generate the test project. Once selected, the label below this field will show the final destination of the test project.
   Click `Generate` at the bottom of the wizard to start the generation process. A console will appear showing generation process and result.



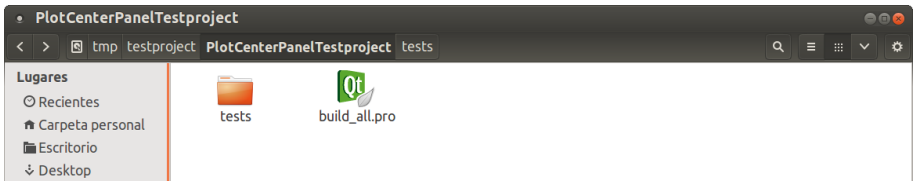
8. **Step 8: Exit the wizard.**
   Click ☰ to deploy the pop-up menu and select `Exit`.



3.4 Test Project Compilation and Execution

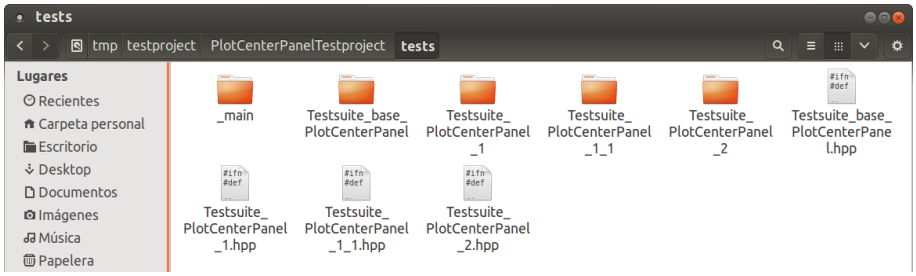
Once we have generated a test project using TUG Wizard, in this section we are going to describe how to compile and execute the project, as well as how to check the results obtained from its execution. This guide assumes that Qt Creator (`https://qt-project.org/wiki/Category:Tools::QtCreator`) is being used to open, compile, and deploy Qt-based projects.

1. **Step 0: Project structure.**
   The generated test project is composed of:
   - a main project (represented by `build_all.pro` file in the figure below) that includes all test projects, and from which they can be compiled.
   - a folder including all test subprojects (`tests` folder in the figure below).

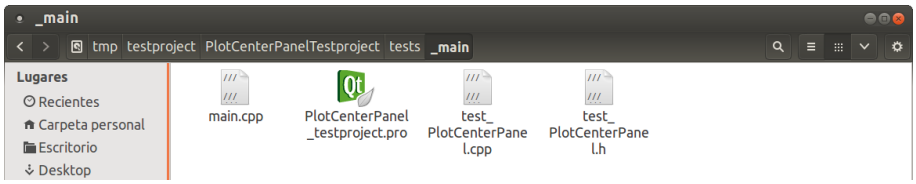


Once opened the `tests` folder, we can find:

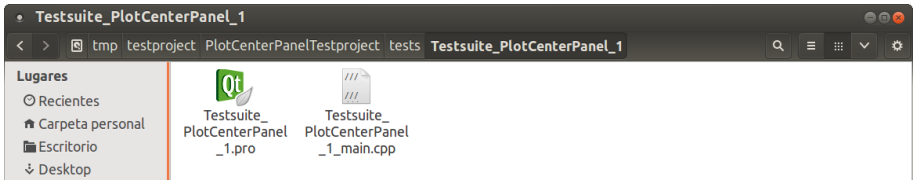
- `_main` subproject including a class that inherits from the panel to test, and that includes a set of methods aimed at supporting the interaction with panel widgets.
- all testsuites previously configured in TUG Wizard...
- ...as well as the test projects from which they can be compiled and executed individually.



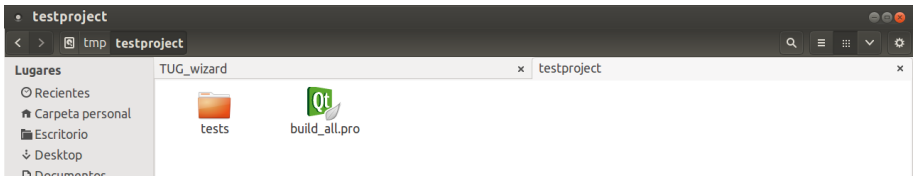
An snapshot of the _main subproject is included below. The `.pro` file can be opened using QtCreator to modify/compile/run the project.



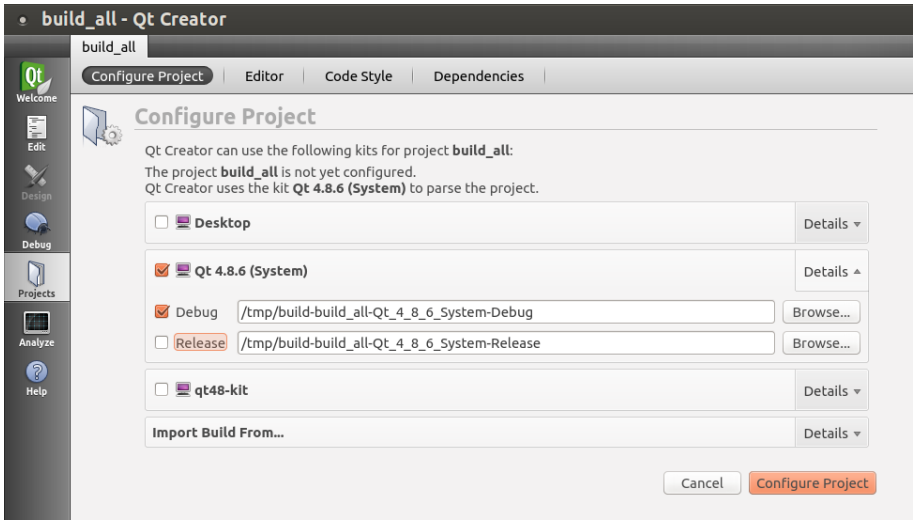
An snapshot of a testsuite subproject is included below.



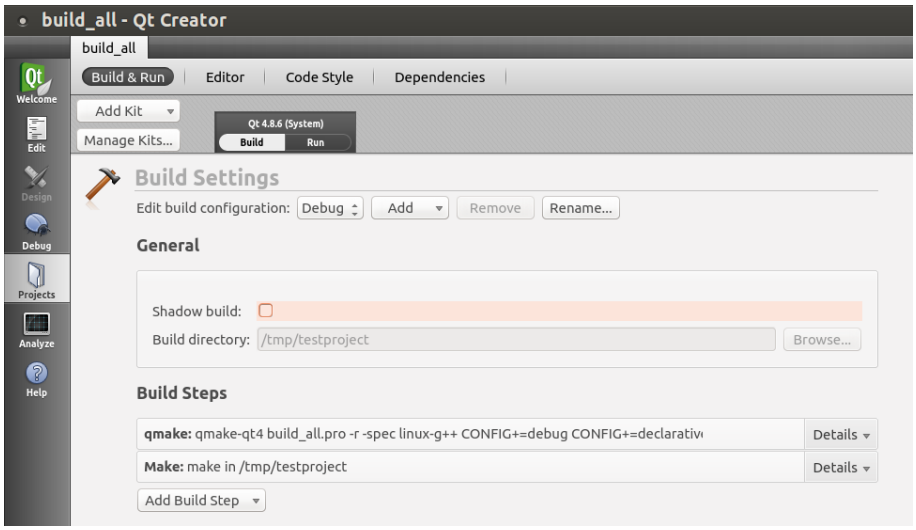
2. **Step 1: Open project.**
   Go to project folder.
   Double click `build_all.pro` file to open project in Qt Creator.



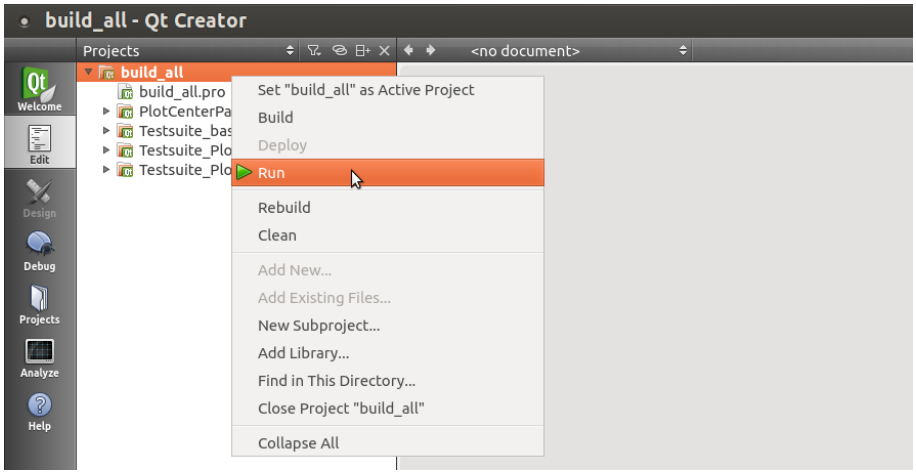
3. **Step 2: Configure project (i).**
   Once in Qt Creator, in `Configure Project` window, check only `Debug` version and click `Configure Project`.



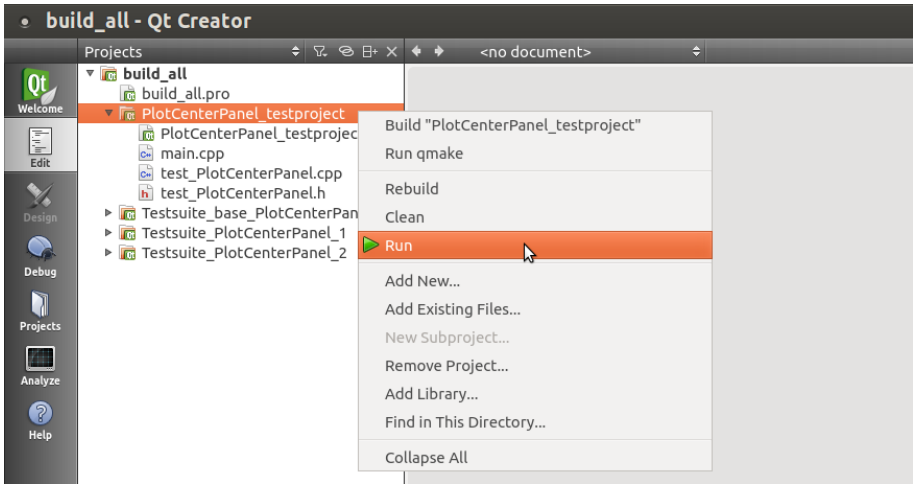
4. **Step 3: Configure project (ii).**
   Go to `Projects` section in the tools bar at left.
   Uncheck `Shadow Build`.



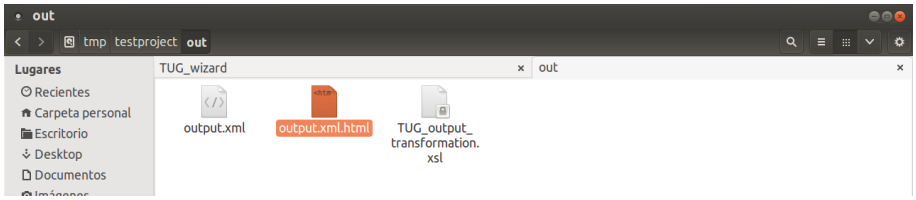
5. **Step 4: Project compilation.**
   Go to `Edit` section in the tools bar at left.
   Right click on `build_all.pro` at the top of the projects list and then click `Rebuild`.



6. **Step 5: Project execution.**
   Go to main project (it is placed the first in the project list), right click on it and then click `Run`.



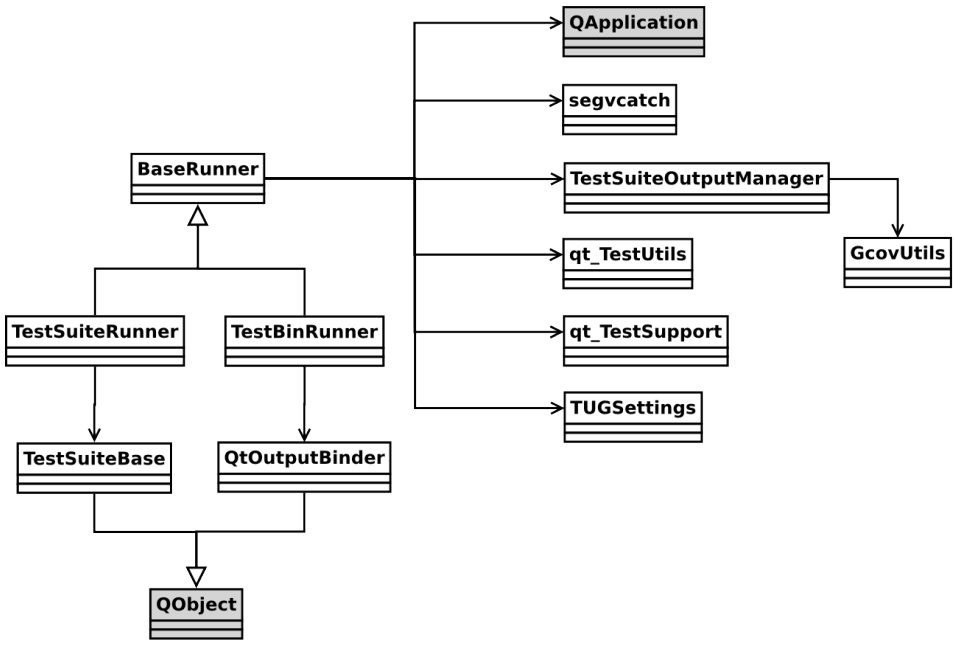
7. **Step 6: Check project output.**
   In the project folder, go to `out` folder and open `output.xml.html` file to see a summary of execution results.



4 TUG Base Library

4.1 Library Design

TUG Base Library is the supporting library used by test projects generated using TUG Wizard. Its architecture design is depicted in the figure below.



Main elements of this architecture are described in the following:

- `TestSuiteBase`
- `BaseRunner`
- `TestSuiteRunner`
- `TestBinRunner`
- `qt_TestUtils`
- `qt_TestSupport`

`TestSuiteBase` is the base class for any testsuite. It implements the basic functionality of a test suite, and includes also some virtual methods to be implemented in subclasses. It extends `QObject` class to be integrated in Qt Test framework.

The code snippet below shows how new testsuites have to be defined:

```
#include "TestSuiteBase.h"

class Testsuite_base_Panel : public TestSuiteBase {

    Q_OBJECT /// Our test suite has to execute Q_OBJECT macros

    (...)
```

Testsuites based on `TestSuiteBase` can be created as object and executed standalone by using the following methods:

```
//
static int
 launch_standalone (TestSuiteBase* tsb, int argc=0, char *argv[]=0);
//
static int
 launch (TestSuiteBase* tsb, QApplication* app);
//
static int
 launch (TestSuiteBase* tsb, QApplication* app, int argc,char** argv);
```

One or more `TestSuiteBase` objects can be also be launched sequentially using `TestSuiteRunner`, as described below.

`BaseRunner` includes the basic functionality to run tests. Tests can be run from a testsuite class file using `TestSuiteRunner`, or from a testsuite binary using `TestBinRunner`. The former has the advantage that no previously compiled binaries are needed, and that all tests are encapsulated into a single binary file. The latter has the advantage that tests are executed as independent binary files, allowing thus to handle segmentation faults. This is the option used in TUG Wizard.

All relevant methods and options supported by `BaseRunner` are described in the following:

- `BaseRunner& add_timestamp_to_output_filename()`: adds a timestamp to the name of the generated output files.
- `BaseRunner& coverage_on_dir(const std::string& dir)`: includes directory `dir` into coverage analysis.
- `BaseRunner& coverage_on_file(const std::string& filepath)`: includes file `filepath` into coverage analysis.
- `BaseRunner& output_***()`: selects output level between `silent`, `verbose`, `extended` or `all`.
- `BaseRunner& pause_between(int ms)`: sets idle time between the execution of a test and the following one.
- `BaseRunner& project_name(const std::string name)`: sets a name for the test project. This name is used if a report is generated.
- `void reset()`: resets runner values.
- `int run()`: run the tests previously added to the runner.

`TestSuiteRunner` can be used to run testsuite files. It provides the method `add_testsuite()` to add a new testsuite object to the runner. Testsuites objects are run sequentially. If a test causes segmentation fault or another similar error, next text will not be executed.

The code snippet below shows how `TestSuiteRunner` has to be defined and properly configured. In the example, a project name is given, output is configured, and coverage targets are set. A new testsuite object is created and added to the runner. Finally, the tests are run.

```
1  #include <TestSuiteRunner.h>
2  #include "testsuite_Panel_1.hpp"
3
4  int main(int argc, char *argv[])
5  {
6      /// 1. create a TestSuiteRunner and configure...
7      TestSuiteRunner trunner(argc,argv);
8      trunner.project_name("PanelTestproject");
9
10     // output options
11     trunner.output_verbose();
12
13     // coverage options
14     trunner.coverage_on_file("/adir/afile.cpp")
15            .coverage_on_dir("/adir/");
16
17     /// 2. add testsuites to the runner
18     testsuite_Panel_1 ts1;
19     trunner.add_testsuite(&ts1);
20
21     /// 3. run testsuites
22     return trunner.pause_between(1000).run_testsuites();
23 }
```

`TestBinRunner` can be used to run testsuite binaries (i.e., testsuite projects already compiled). It provides the method `add_testbin()` to add a new testsuite binary to the runner. Testsuites binaries are run sequentially. If a test causes segmentation fault or another similar error, runner tries to recover output data and then executes next test.

The code snippet below shows how `TestBinRunner` has to be defined and properly configured. In the example, a project name is given, output is configured, and coverage targets are set. A new testsuite binary is added to the runner using a relative path. Finally, the tests are run.

```
1  #include <TestBinRunner.h>
2
3  int main(int argc, char *argv[])
4  {
5      /// 1. create a test runner and configure...
6      TestBinRunner trunner(argc,argv);
7      trunner.project_name("PanelTestproject");
8
9      // output options
10     trunner.output_verbose();
11
12     // coverage options
13     trunner.coverage_on_file("/adir/afile.cpp")
14            .coverage_on_dir("/adir/");
15
16     /// 2. add test binaries to the runner
17     trunner.add_testbin("./Testsuite_PlotCenterPanel_1_1");
18
19     /// 3. run tests
20     return trunner.pause_between(1000).run();
21 }
```

`qt_TestUtils` includes a set of methods aimed at supporting the definition of new tests (e.g., launch or repaint a panel, sleep some milliseconds, assert values, generate random variables, simulate segmentation faults, send data to logs, range macros, etc).

`qt_TestSupport` includes a set of methods to manipulate Qt GUI widgets. These methods try to simulate user interaction.
Relevant methods in these classes are further described in next subsection.

4.2 Library Utils for Writing Tests

As said above, `qt_TestUtils` includes a set of methods aimed at supporting the definition of new tests. These methods can be accessed through the namespace `tug::`. Some of these methods are described next.

Launch and destroy panels:

```
1  // launches a panel/window object
2  void panel_launch(QWidget* panel)
3  void panel_launch(QWidget& panel)
4
5  // deletes a panel/window object
6  void panel_destroy(QWidget* panel)
7  void panel_destroy(QWidget& panel)
```

```
1  Example:
2  ------------------------------------------------------------------------
3  test_panel panel;
4  tug::panel_launch(panel);
```

Assertions:

```
1  // checks a boolean expression
2  void assert(bool expr)
3
4  // checks a boolean expression and displays 'msg' if error
5  void assert_msg(bool expr, const char* msg)
6
7  // prints a warning message in a test
8  void warning(const char* msg)
9
10 // simulates an error in a test
11 void fail(const char* msg)
```

```
1  Example:
2  ------------------------------------------------------------------------
3  tug::assert(1 == true);
4  tug::assert_msg(1 == false,``This will never be true'');
5  tug::warning(``This is a warning message'');
6  tug::fail(``Simulating a failure'');
```

Update/repaint panels:

```
1  // repaints a panel
2  void panel_repaint(QWidget* panel)
3  void panel_repaint(QWidget& panel)
4
5  // hides and shows a panel. It forces repaint and update.
6  void panel_blink(QWidget* panel)
7  void panel_blink(QWidget& panel)
```

```
Example:
-------------------------------------------------------------------------
test_panel panel; tug::panel_launch(panel);
// do something here...
tug::panel_repaint(panel); //this repaints the user interface
// do something more...
tug::panel_blink(panel); //this forces a panel update
```

Sleeps:

```
// sleeps 'ms' milliseconds
void sleep(int ms)

// sleeps 1 second
void sleep1()

// sleeps 2 second
void sleep2()

// sleeps 3 second
void sleep3()

// sleeps 5 second
void sleep5()
```

```
Example:
-------------------------------------------------------------------------
tug::sleep2();
```

Random values:

```
// resets random numbers generator
void random_reset()

// generates a random number between 0 and 'n'
int random(int n)

// generates a random number between 'low' and 'high'
int random_in_range(int low, int high)

// generates true or false randomly
bool random_bool()
```

```
Example:
-------------------------------------------------------------------------
tug::random_reset();
panel->amethod_expecting_int(tug::random(10));
panel->amethod_expecting_percentage(tug::random(0,100));
panel->amethod_expecting_bool(tug::random_bool());
```

Simulation of segmentation faults:

```
// simulates a segmentation fault
void segfault()
```

```
Example:
-------------------------------------------------------------------------
tug::segfault(); //at this point a testsuite ends its execution
```

Timers:

```
/// starts timer
void timer_start()

/// returns milliseconds elapsed from last call to timer_start
long timer_elapsed_ms()
```

```
Example:
-------------------------------------------------------------------------
tug::timer_start();

tug::sleep1();
tug::log() << "timer 1: " << tug::timer_elapsed_ms();
tug::sleep2();
tug::log() << "timer 2: " << tug::timer_elapsed_ms();

tug::timer_start(); //timer reset

tug::sleep2();
tug::log() << "timer 3: " << tug::timer_elapsed_ms();
```

Ranges:

```
/// ForRange
template <typename T>
class ForRange(T l, T u)

/// ForNestedRange
template <typename T>
class ForNestedRange(T l, T u, T li, T ui)
```

```
Example:
-------------------------------------------------------------------------
tug::ForRange<double>(-10,70)
   .call<test_panel>(_panel, &test_panel::set_sbLatMin)
   .call_void<test_panel>(_panel, &test_panel::doClick_btApplyCenter)
   .repaint(_panel) //optional
   .sleep_ms(100)   //optional - default is 0
   .increment(2)    //optional - default is 1
   .run();

tug::ForNestedRange<int>(-100,100,-200,200)
   .call<test_panel>(_panel, &test_panel::set_sbLatDegrees)
```

```
13     .call_inner<test_panel>(_panel, &test_panel::set_sbLongDegrees)
14     .call_void<test_panel>(_panel, &test_panel::doClick_btApplyCenter)
15     .repaint(_panel) //optional
16     .increment(5)    //optional - default is 1
17     .run();
```

Log:

```
1  // log adds a log line to a test
2  void log(const char* s)
```

```
1  Example:
2  ------------------------------------------------------------------------
3  tug::log(``log a sentence...'');
4  tug::log() << ``or use it as `` << 1 << `` stream.'';
```

Macros for value ranges (out of `tug::` namespace):

```
1  // This macro repeates a code 'n' times.
2  // Additionally, values from 0 to 'n-1' are generated.
3  // 'value' is the name of the variable to be used.
4  tug__REPEAT(n)
5
6  // This macro simulates an integer range between 'min' and 'max', both
7  // included.
8  // 'value' is the name of the variable to be used.
9  tug__INT_RANGE(min,max)
10
11 // This macro simulates an integer range between 'min' and 'max', both
12 // included, using 'inc' as increment.
13 // 'value' is the name of the variable to be used.
14 tug__INT_RANGE_INC(min,max,inc)
15
16 // This macro simulates a float range between 'min' and 'max', both
17 // included, using 'inc' as increment.
18 // 'value' is the name of the variable to be used.
19 tug__FLOAT_RANGE(min,max,inc)
20
21 // This macro executes a code 'n' iterations.
22 // At each iteration 'value' holds a random value between 0
23 // and 'random_limit'.
24 // 'value' is the name of the variable to be used.
25 tug__RANDOM_INT_SET(n,random_limit)
26
27 // This macro executes a code 'n' iterations.
28 // At each iteration 'value' holds a random value true or false.
29 // 'value' is the name of the variable to be used.
30 tug__RANDOM_BOOL_SET(n)
```

4.3 Library Configuration

Some options of TUG Base Library can be configured in `settings` file. For changes in this file to take effect, it is needed to recompile the library after saving it. Some of the options that can be changed are briefly described in the following.

Commands used to gather to coverage and profiling information:

```
### gcov/lcov/gprof commands

gcov_pre_command = "gcov"
gcov_pre_options = "-p -r -n -o ##FILEPATH## ##FILEPATH##;"
gcov_post_command = ""
gcov_post_options = ""
gcov_clean_command = "rm *.gcda *.gcov;"

gprof_pre_command = "gprof"
gprof_pre_options = "-p -b ##BINNAME## gmon.out;"
gprof_post_command = ""
gprof_post_options = ""
gprof_clean_command = "rm gmon.out;"

lcov_pre_command = "lcov"
lcov_pre_options = "-z --directory ##SOURCEDIR##;"
lcov_post_command = "lcov"
lcov_post_options = "--capture --ignore-errors graph --directory ##
    SOURCEDIR## --output-file /tmp/coverage.info; genhtml /tmp/
    coverage.info --output-directory ##DESTDIR##;"
lcov_clean_command = "rm /tmp/coverage.info;"
```

Directories in which include generated artifacts:

```
### main directories

# dir paths relatives to "maintest"
dir_bin = "../bin"
dir_coverage = "../coverage"
dir_output = "../out"
dir_tests = "../bin"
```

File names used during output management:

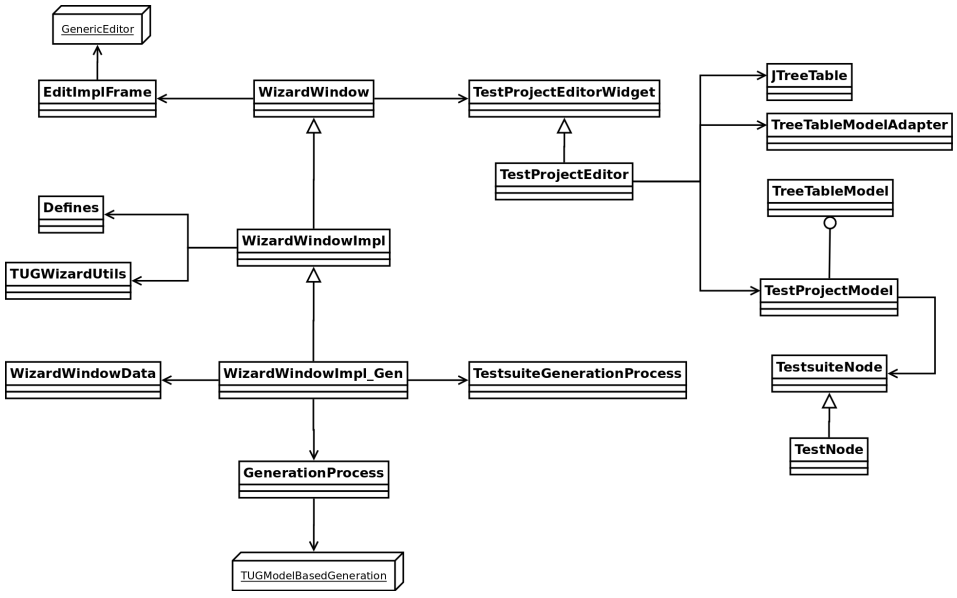
```
### output file

output_style_filename = "TUG_output_transformation.xsl"
output_temp_file = "/tmp/___tug_test_output";
output_final_filename = "output.xml";
```

5 TUG Wizard Design and Configuration

5.1 Wizard Design

TUG Wizard is a wizard-like application that helps developers create and configure test projects to test Qt based panels and their related classes. Its architecture design is depicted in the figure below.



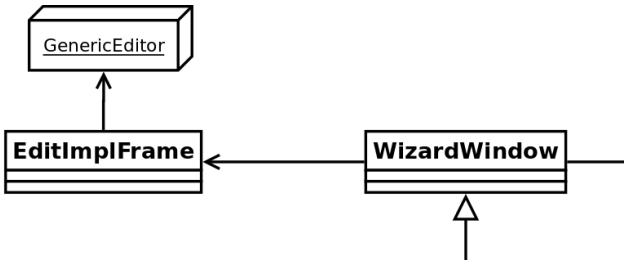
This architecture is divided into three levels. Inheritance is used to divide the implementation of `WizardWindow` into the following three classes:

- `WizardWindow`: includes configuration and deployment of TUG Wizard user interface.
- `WizardWindowImpl`: includes the implementation under the TUG Wizard user interface, excluding generation functionality.
- `WizardWindowImpl_Gen`: includes the implementation of the test project generation processes.

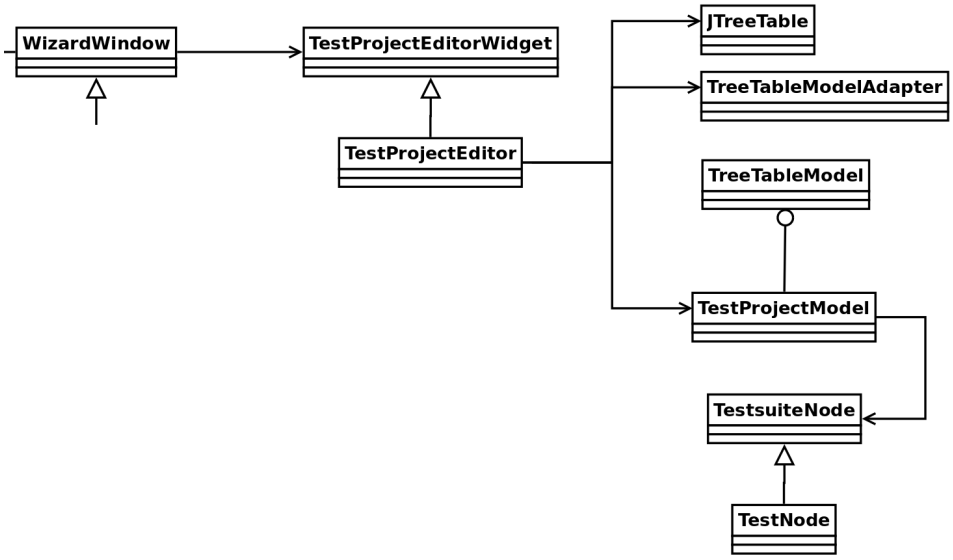
The most relevant classes included at each of these levels are further described in the following.

As said above, `WizardWindow` implements the configuration and deployment of TUG Wizard user interface. It uses two supporting classes, `EditImplFrame` and `TestProjectEditorWidget`, used to allow developers to edit configuration files and to configure a test project structure, respectively.

`EditImplFrame` is the main class of `GenericEditor`, an external component used in TUG Wizard in Steps 4 and 5 to allow the modification of configuration files. This component is not further described because it is out of the scope of this guide.

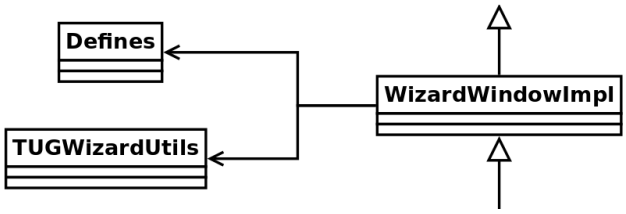


`TestProjectEditorWidget` represents the component to configure a test project structure composed of a set of testsuites and tests (Step 6 in TUG Wizard). The editor is based on a `JTreeTable` object and uses an underlying model described by `TestProjectModel`.

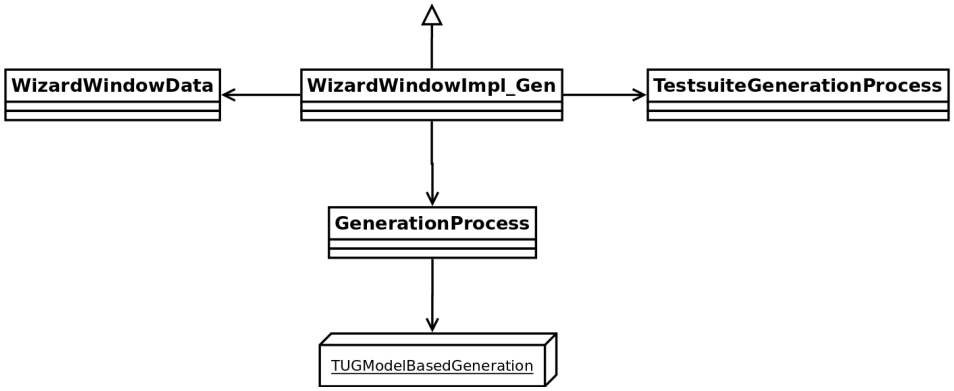


`WizardWindowImpl` includes all the implementation under the TUG Wizard user interface. It implements all steps in the process, excluding Step 7 in which the generation process is carried out.

`TUGWizardUtils` implements supporting methods to simplify the wizard processes. The `Defines` class defines some constant values used during the wizard and generation processes.



`WizardWindowImpl_Gen` includes all the implementation of the TUG Wizard related to the test project generation processes.



The generation processes are based on the data encapsulated within `WizardWindowData`. This object includes all configuration data provided by developers during the wizard steps.

`GenerationProcess` encapsulates the generation of a new test panel project. This process uses the Qt panel and the dependencies introduced by developers in the wizard steps to generate a new panel to be tested. This new panel inherits the original one and includes methods to simulate users interaction with the widgets composing the panel. These methods are aimed at being

used in tests definition. It is also generated a Qt project to compile and run this panel standalone.

The `TestsuiteGenerationProcess` class includes methods supporting the generation of a test project structure. A test project is composed of:

- a main project called `build_all.pro` from which everything can be compiled.
- the test panel project described above. It is included in `_main` folder.
- a set of testsuites. Each testsuite includes a set of empty methods (i.e., the tests) to be filled by developers with the desired functionality. Methods from the test panel and from `qt_TestUtils` class can be used to support the tests implementation.
- a Qt project for each testsuite. This project can be used to compile and run the testsuites standalone.

5.2 Wizard Configuration

TUG Wizard includes some files that allow developers to configure the generation process. Some of these files can be modified from the TUG Wizard user interface. Others can only be modified using an external text editor. These files can be found in `config/` folder, and are briefly described in the following.

`config/generation_templates`: set of templates used during the generation of testsuites and test projects.

- `testsuitebase_template`: testsuite base template.
- `mp_testsuite_template`: template for testsuites in which a new panel is launched for each test.
- `mp_test_template`: template for tests in which a new panel is launched for each test.
- `op_testsuite_template`: template for testsuites in which only one panel is used to execute all tests (for leaf nodes in the test projects structure).
- `op_testsuite_internal_template`: template for testsuites in which only one panel is used to execute all tests (for internal nodes in the test projects structure).
- `op_test_template`: template for tests in which only one panel is used to execute all tests.
- `testsuite_project_main`: template for main file launching a testsuite.
- `testsuite_project_pro`: template for Qt project file compiling a testsuite.

`config/includes`: set of files in which the includes related to different options selected in TUG Wizard are defined. These files can be modified also from TUG Wizard user interface.

- `boost_signals_include`: defines the lines to include Boost signals into a test project.
- `libsig_signals_include`: defines the lines to include Libsig signals into a test project.
- `gcov_include`: defines the lines to include Gcov into a test project.
- `gprof_include`: defines the lines to include Gprof into a test project.
- `tuglib_include`: defines the lines to include TUGLib library into a test project.