Quantum GIS (QGIS)

Developers guide for QGIS

Contents

1 QGIS Coding Standards

These standards should be followed by all QGIS developers.

1.1 Classes

1.1.1 Names

Class in QGIS begin with Qgs and are formed using mixed case.

Listing

```
Examples:
  QgsPoint
  QgsMapCanvas
  QgsRasterLayer
```

1.1.2 Members

Class member names begin with a lower case m and are formed using mixed case.

Listing

```
  mMapCanvas
  mCurrentExtent
```

All class members should be private. **Public class members are STRONGLY discouraged**

1.1.3 Accessor Functions

Class member values should be obtained through accesssor functions. The function should be named without a *get* prefix. Accessor functions for the two private members above would be:

Listing

```
  mapCanvas()
  currentExtent()
```


1.1.4 Functions

Function names begin with a lowercase letter and are formed using mixed case. The function name should convey something about the purpose of the function.

Listing

```
updateMapExtent()
setUserOptions()
```

1.2 Qt Designer

1.2.1 Generated Classes

QGIS classes that are generated from Qt Designer (ui) files should have a *Base* suffix. This identifies the class as a generated base class.

Listing

```
Examples:
  QgsPluginManagerBase
  QgsUserOptionsBase
```

1.2.2 Dialogs

All dialogs should implement the following: * Tooltip help for all toolbar icons and other relevant widgets * WhatsThis help for **all** widgets on the dialog * An optional (though highly recommended) context sensitive *Help* button that directs the user to the appropriate help page by launching their web browser

1.3 C++ Files

1.3.1 Names

C++ implementation and header files should be have a .cpp and .h extension respectively. Filename should be all lowercase and, in the case of classes, match the class name.

Listing

```
Example:
  Class QgsFeatureAttribute source files are
    qgsfeatureattribute.cpp and qgsfeatureattribute.h
```


/!\ **Note:** in case it is not clear from the statement above, for a filename to match a class name it implicitly means that each class should be declared and implemented in its own file. This makes it much easier for newcomers to identify where the code is relating to specific class.

1.3.2 Standard Header and License

Each source file should contain a header section patterned after the following example:

Listing

```
/***************************************************************************
    qgsfield.cpp - Describes a field in a layer or table
     --------------------------------------
    Date                 : 01-Jan-2004
    Copyright            : (C) 2004 by Gary E.Sherman
    Email                : sherman at mrcc.com
/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/
```

1.3.3 Keyword Substitution

In the days of SVN we used to require that each source file should contain the Id keyword. Keyword substitution is not supported by GIT and so should no longer be used.

1.4 Variable Names

Variable names begin with a lower case letter and are formed using mixed case.

Listing

```
Examples:
  mapCanvas
  currentExtent
```

1.5 Enumerated Types

Enumerated types should be named in CamelCase with a leading capital e.g.:

Listing

```
enum UnitType
{
  Meters,
  Feet,
  Degrees,
  UnknownUnit
} ;
```

Do not use generic type names that will conflict with other types. e.g. use "UnkownUnit" rather than "Unknown"

1.6 Global Constants

Global constants should be written in upper case underscore separated e.g.:

Listing

```
const long GEOCRS_ID = 3344;
```

1.7 Editing

Any text editor/IDE can be used to edit QGIS code, providing the following requirements are met.

1.7.1 Tabs

Set your editor to emulate tabs with spaces. Tab spacing should be set to 2 spaces.

1.7.2 Indentation

Source code should be indented to improve readability. There is a .indent.pro file in the QGIS src directory that contains the switches to be used when indenting code using the GNU indent program. If you don't use GNU indent, you should emulate these settings.

1.7.3 Braces

Braces should start on the line following the expression:

Listing

```
if(foo == 1)
{
  // do stuff
  ...
 }else
{
  // do something else
  ...
}
```

There is a scripts/prepare-commit.sh that looks up the changed files and reindents them using astyle. This should be run before committing.

As newer versions of astyle indent differently than the version used to do a complete reindentation of the source, the script uses an old astyle version, that we include in our repository.

1.8 API Compatibility

From QGIS 1.0 we will provide a stable, backwards compatible API. This will provide a stable basis for people to develop against, knowing their code will work against any of the 1.x QGIS releases (although recompiling may be required).Cleanups to the API should be done in a manner similar to the Trolltech developers e.g.

Listing

```
class Foo
{
  public:
    /** This method will be deprecated, you are encouraged to use
      doSomethingBetter() rather.
      @see doSomethingBetter()
     */
    bool doSomething();

    /** Does something a better way.
      @note This method was introduced in QGIS version 1.1
     */
    bool doSomethingBetter();

}
```

1.9 Coding Style

Here are described some programming hints and tips that will hopefully reduce errors, development time, and maintenance.

1.9.1 Where-ever Possible Generalize Code

Listing

```
If you are cut-n-pasting code, or otherwise writing the same thing more than
once, consider consolidating the code into a single function.
```

This will:

- allow changes to be made in one location instead of in multiple places
- help prevent code bloat
- make it more difficult for multiple copies to evolve differences over time, thus making it harder to understand and maintain for others

1.9.2 Prefer Having Constants First in Predicates

Prefer to put constants first in predicates.

Listing

```
"0 == value" instead of "value == 0"
```

This will help prevent programmers from accidentally using "=" when they meant to use "==", which can introduce very subtle logic bugs. The compiler will generate an error if you accidentally use "=" instead of "==" for comparisons since constants inherently cannot be assigned values.

1.9.3 Whitespace Can Be Your Friend

Adding spaces between operators, statements, and functions makes it easier for humans to parse code.

Which is easier to read, this:

Listing

```
if (!a&&b)
```

or this:

Listing

```
if ( ! a && b )
```


1.9.4 Add Trailing Identifying Comments

Adding comments at the end of function, struct and class implementations makes it easier to find them later.

Consider that you're at the bottom of a source file and need to find a very long function – without these kinds of trailing comments you will have to page up past the body of the function to find its name. Of course this is ok if you wanted to find the beginning of the function; but what if you were interested at code near its end? You'd have to page up and then back down again to the desired part.

e.g.,

Listing

```
void foo::bar()
{
    // ... imagine a lot of code here
} // foo::bar()
```

1.9.5 Use Braces Even for Single Line Statements

Using braces for code in if/then blocks or similar code structures even for single line statements means that adding another statement is less likely to generate broken code.

Consider:

Listing

```
 if (foo)
   bar();
else
   baz();
```

Adding code after bar() or baz() without adding enclosing braces would create broken code. Though most programmers would naturally do that, some may forget to do so in haste.

So, prefer this:

Listing

```
if (foo)
{
  bar();
}
else
{
   baz();
```


```
}
```

1.9.6 Book recommendations

- Effective C++, Scott Meyers
- More Effective C++, Scott Meyers
- Effective STL, Scott Meyers
- Design Patterns, GoF

You should also really read this article from Qt Quarterly on http://doc.trolltech.com/qq/qq13-apis.html designing Qt style

2 GIT Access

This section describes how to get started using the QGIS GIT repository. Before you can do this, you need to first have a git client installed on your system. Debian based distro users can do:

Listing

```
sudo apt-get install git
```

Windows users can obtain msys git.

2.1 Accessing the Repository

To clone QGIS master:

Listing

```
git://github.com/qgis/Quantum-GIS.git
```

2.2 Check out a branch

To check out a branch, for example the release 1.7.0 branch do:

Listing

```
cd Quantum-GIS
git fetch
git branch --track origin release-1_7_0
git checkout release-1_7_0
```

To check out the master branch:

Listing

```
cd Quantum-GIS
git checkout master
```

/!\ **Note:** In QGIS we keep our most stable code in the current release branch. Master contains code for the so called 'unstable' release series. Periodically we will branch a release off master, and then continue stabilisation and selective incorporation of new features into master.

See the INSTALL file in the source tree for specific instructions on building development versions.

2.3 QGIS documentation sources

If you're interested in checking out Quantum GIS documentation sources:

Listing

```
svn co https://svn.osgeo.org/qgis/docs/trunk qgis_docs
```

/!\ **Note:** This url will change to a git URL in the near future.

You can also take a look at DocumentationWritersCorner for more information.

2.4 GIT Documentation

See the following sites for information on becoming a GIT master.

http://gitref.org http://progit.org http://gitready.com

2.5 Development in branches

2.5.1 Purpose

The complexity of the QGIS source code has increased considerably during the last years. Therefore it is hard to anticipate the side effects that the addition of a feature will have. In the past, the QGIS project had very long release cycles because it was a lot of work to reetablish the stability of the software system after new features were added. To overcome these problems, QGIS switched to a development model where new features are coded in GIT branches first and merged to master (the main branch) when they are finished and stable. This section describes the procedure for branching and merging in the QGIS project.

2.5.2 Procedure

- **Initial announcement on mailing list:** Before starting, make an announcement on the developer mailing list to see if another developer is already working on the same feature. Also contact the technical advisor of the project steering committee (PSC). If the new feature requires any changes to the QGIS architecture, a request for comment (RFC) is needed.

  Create a branch: Create a new GIT branch for the development of the new feature.

Listing

```
git branch newfeature
git checkout newfeature
```

Now you can start developing. If you plan to do extensive on that branch, would like to share the work with other developers, and have write access to the upstream repo, you can push your repo up to the QGIS official repo by doing:

Listing

```
git push origin newfeature
```

Note: if the branch already exists your changes will be pushed into it.

Merge from master regularly: It is recommended to merge the changes in master to the branch on a regular basis. This makes it easier to merge the branch back to master later.

Listing

```
git merge master
```

Documentation on wiki: It is also recommended to document the intended changes and the current status of the work on a wiki page.

Testing before merging back to master: When you are finished with the new feature and happy with the stability, make an announcement on the developer list. Before merging back, the changes will be tested by developers and users. Binary packages (especially for OsX and Windows) will be generated to also involve non-developers. In trac, a new Component will be opened to file tickets against. Once there are no remaining issues left, the technical advisor of the PSC merges the changes into master.

2.6 Submitting Patches

There are a few guidelines that will help you to get your patches into QGIS easily, and help us deal with the patches that are sent to use easily.

2.6.1 Patch file naming

If the patch is a fix for a specific bug, please name the file with the bug number in it e.g. **bug777fix.patch**, and attach it to the original bug report in trac (https://trac.osgeo.org/qgis/).

If the bug is an enhancement or new feature, its usually a good idea to create a ticket in trac (https://trac.osgeo.org/qgis/) first and then attach you

2.6.2 Create your patch in the top level QGIS source dir

This makes it easier for us to apply the patches since we don't need to navigate to a specific place in the source tree to apply the patch. Also when I receive patches I usually evaluate them using merge, and having the patch from the top level dir makes this much easier. Below is an example of how you can include multiple changed files into your patch from the top level directory:

Listing

```
cd Quantum-GIS
git checkout master
git pull origin master
git checkout newfeature
git format-patch master --stdout > bug777fix.patch
```

This will make sure your master branch is in sync with the upstream repository, and then generate a patch which contains the delta between your feature branch and what is in the master branch.

2.6.3 Getting your patch noticed

QGIS developers are busy folk. We do scan the incoming patches on bug reports but sometimes we miss things. Don't be offended or alarmed. Try to identify a developer to help you - using the Project Organigram and contact them asking them if they can look at your patch. If you don't get any response, you can escalate your query to one of the Project Steering Committee members (contact details also available on the Project Organigram).

2.6.4 Due Diligence

QGIS is licensed under the GPL. You should make every effort to ensure you only submit patches which are unencumbered by conflicting intellectual property rights. Also do not submit code that you are not happy to have made available under the GPL.

2.7 Obtaining GIT Write Access

Write access to QGIS source tree is by invitation. Typically when a person submits several (there is no fixed number here) substantial patches that demonstrate basic competence and understanding of C++ and QGIS coding conventions, one of the PSC members or other existing developers can nominate that person to the PSC for granting of write access. The nominator should give a basic promotional paragraph of why they think that person should gain write access. In some cases we will grant write access to non C++ developers e.g. for translators and documentors. In these cases, the person should still have demonstrated ability to submit patches and should ideally have submitted several substantial patches that demonstrate their understanding of modifying the code base without breaking things, etc.

Note: Since moving to GIT, we are less likely to grant write access to new developers since it is trivial to share code within github by forking QGIS and then issuing pull requests.

Always check that everything compiles before making any commits / pull requests. Try to be aware of possible breakages your commits may cause for people building on other platforms and with older / newer versions of libraries.

When making a commit, your editor (as defined in $EDITOR environment variable) will appear and you should make a comment at the top of the file (above the area that says 'don't change this'. Put a descriptive comment and rather do several small commits if the changes across a number of files are unrelated. Conversely we prefer you to group related changes into a single commit.

3 Unit Testing

As of November 2007 we require all new features going into master to be accompanied with a unit test. Initially we have limited this requirement to **qgis_core**, and we will extend this requirement to other parts of the code base once people are familiar with the procedures for unit testing explained in the sections that follow.

3.1 The QGIS testing framework - an overview

Unit testing is carried out using a combination of QTestLib (the Qt testing library) and CTest (a framework for compiling and running tests as part of the CMake build process). Lets take an overview of the process before I delve into the details:

- **There is some code you want to test**, e.g. a class or function. Extreme programming advocates suggest that the code should not even be written yet when you start building your tests, and then as you implement your code you can immediately validate each new functional part you add with your test. In practive you will probably need to write tests for pre-existing code in QGIS since we are starting with a testing framework well after much application logic has already been implemented.
- **You create a unit test.** This happens under <QGIS Source Dir>/tests/src/core in the case of the core lib. The test is basically a client that creates an instance of a class and calls some methods on that class. It will check the return from each method to make sure it matches the expected value. If any one of the calls fails, the unit will fail.
- **You include QtTestLib macros in your test class.** This macro is processed by the Qt meta object compiler (moc) and expands your test class into a runnable application.
- **You add a section to the CMakeLists.txt** in your tests directory that will build your test.
- **You ensure you have ENABLE_TESTING enabled in ccmake / cmake-setup.** This will ensure your tests actually get compiled when you type make.
- **You optionally add test data to <QGIS Source Dir>/tests/testdata** if your test is data driven (e.g. needs to load a shapefile). These test data should be as small as possible and wherever possible you should use the existing datasets already there. Your tests should never modify this data in situ, but rather may a temporary copy somewhere if needed.
- **You compile your sources and install.** Do this using normal make && (sudo)

make install procedure.

- **You run your tests.** This is normally done simply by doing **make test** after the make install step, though I will explain other aproaches that offer more fine grained control over running tests.

Right with that overview in mind, I will delve into a bit of detail. I've already done much of the configuration for you in CMake and other places in the source tree so all you need to do are the easy bits - writing unit tests!

3.2 Creating a unit test

Creating a unit test is easy - typically you will do this by just creating a single .cpp file (not .h file is used) and implement all your test methods as public methods that return void. I'll use a simple test class for QgsRasterLayer throughout the section that follows to illustrate. By convention we will name our test with the same name as the class they are testing but prefixed with 'Test'. So our test implementation goes in a file called testqgsrasterlayer.cpp and the class itself will be TestQgsRasterLayer. First we add our standard copyright banner:

Listing

```
/***************************************************************************
     testqgsvectorfilewriter.cpp
     ---------------------------------------
    Date                 : Frida  Nov 23  2007
    Copyright            : (C) 2007 by Tim Sutton
    Email                : tim@linfiniti.com
 ***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/
```

Next we use start our includes needed for the tests we plan to run. There is one special include all tests should have:

Listing

```
#include <QtTest>
```

Note that we use the new style Qt4 includes - i.e. QtTest is included not qttest.h

Beyond that you just continue implementing your class as per normal, pulling in whatever headers you may need:

Listing

```
//Qt includes...
#include <QObject>
#include <QString>
#include <QObject>
#include <QApplication>
#include <QFileInfo>
#include <QDir>

//qgis includes...
#include <qgsrasterlayer.h>
#include <qgsrasterbandstats.h>
#include <qgsapplication.h>
```

Since we are combining both class declaration and implementation in a single file the class declaration comes next. We start with our doxygen documentation. Every test case should be properly documented. We use the doxygen **ingroup** directive so that all the UnitTests appear as a module in the generated Doxygen documentation. After that comes a short description of the unit test:

Listing

```
/** \ingroup UnitTests
 * This is a unit test for the QgsRasterLayer class.
 */
```

The class **must** inherit from QObject and include the Q_OBJECT macro.

Listing

```
class TestQgsRasterLayer: public QObject
{
  Q_OBJECT;
```

All our test methods are implemented as **private slots**. The QtTest framework will sequentially call each private slot method in the test class. There are four 'special' methods which if implemented will be called at the start of the unit test (**initTestCase**), at the end of the unit test (**cleanupTestCase**). Before each test method is called, the **init()** method will be called and after each test method is called the **cleanup()** method is called. These methods are handy in that they allow you to allocate and cleanup resources prior to running each test, and the test unit as a whole.

Listing

```
private slots:
  // will be called before the first testfunction is executed.
  void initTestCase();
  // will be called after the last testfunction was executed.
  void cleanupTestCase(){};
  // will be called before each testfunction is executed.
  void init(){};
```


```
    // will be called after every testfunction.
    void cleanup();
```

Then come your test methods, all of which should take **no parameters** and should **return void**. The methods will be called in order of declaration. I am implementing two methods here which illustrates to types of testing. In the first case I want to generally test the various parts of the class are working, I can use a **functional testing** approach. Once again, extreme programmers would advocate writing these tests **before** implementing the class. Then as you work your way through your class implementation you iteratively run your unit tests. More and more test functions should complete sucessfully as your class implementation work progresses, and when the whole unit test passes, your new class is done and is now complete with a repeatable way to validate it.

Typically your unit tests would only cover the **public** API of your class, and normally you do not need to write tests for accessors and mutators. If it should happen that an acccessor or mutator is not working as expected you would normally implement a **regression** test to check for this (see lower down).

Listing

```
    //
    // Functional Testing
    //

    /** Check if a raster is valid. */
    void isValid();

    // more functional tests here ...
```

Next we implement our **regression tests**. Regression tests should be implemented to replicate the conditions of a particular bug. For example I recently received a report by email that the cell count by rasters was off by 1, throwing off all the statistics for the raster bands. I opened a bug (ticket #832) and then created a regression test that replicated the bug using a small test dataset (a 10x10 raster). Then I ran the test and ran it, verifying that it did indeed fail (the cell count was 99 instead of 100). Then I went to fix the bug and reran the unit test and the regression test passed. I committed the regression test along with the bug fix. Now if anybody breakes this in the source code again in the future, we can immediatly identify that the code has regressed. Better yet before committing any changes in the future, running our tests will ensure our changes don't have unexpected side effects - like breaking existing functionality.

There is one more benifit to regression tests - they can save you time. If you ever fixed a bug that involved making changes to the source, and then running the application and performing a series of convoluted steps to replicate the issue, it will be immediately apparent that simply implementing your regression test **before** fixing the bug will let you automate the testing for bug resolution in an efficient manner.

To implement your regression test, you should follow the naming convention of regression<TicketID> for your test functions. If no trac ticket exists for the regression, you should create one first. Using this approach allows the person running a failed regression test easily go and find out more information.

Listing

```
  //
  // Regression Testing
  //

  /** This is our second test case...to check if a raster
   reports its dimensions properly. It is a regression test
   for ticket #832 which was fixed with change r7650.
   */
  void regression832();

  // more regression tests go here ...
```

Finally in our test class declaration you can declare privately any data members and helper methods your unit test may need. In our case I will declare a QgsRasterLayer * which can be used by any of our test methods. The raster layer will be created in the initTestCase() function which is run before any other tests, and then destroyed using cleanupTestCase() which is run after all tests. By declaring helper methods (which may be called by various test functions) privately, you can ensure that they wont be automatically run by the QTest executeable that is created when we compile our test.

Listing

```
  private:
    // Here we have any data structures that may need to
    // be used in many test cases.
    QgsRasterLayer * mpLayer;
};
```

That ends our class declaration. The implementation is simply inlined in the same file lower down. First our init and cleanup functions:

Listing

```
void TestQgsRasterLayer::initTestCase()
{
  // init QGIS's paths - true means that all path will be inited from prefix
  QString qgisPath = QCoreApplication::applicationDirPath ();
  QgsApplication::setPrefixPath(qgisPath, TRUE);
#ifdef Q_OS_LINUX
  QgsApplication::setPkgDataPath(qgisPath + "/../share/qgis");
#endif
  //create some objects that will be used in all tests...

  std::cout << "Prefix  PATH: " << QgsApplication::prefixPath().toLocal8Bit().data() << std::endl;
  std::cout << "Plugin  PATH: " << QgsApplication::pluginPath().toLocal8Bit().data() << std::endl;
  std::cout << "PkgData PATH: " << QgsApplication::pkgDataPath().toLocal8Bit().data() << std::endl;
  std::cout << "User DB PATH: " << QgsApplication::qgisUserDbFilePath().toLocal8Bit().data() << std::endl;
```


```
  //create a raster layer that will be used in all tests...
  QString myFileName (TEST_DATA_DIR); //defined in CmakeLists.txt
  myFileName = myFileName + QDir::separator() + "tenbytenraster.asc";
  QFileInfo myRasterFileInfo ( myFileName );
  mpLayer = new QgsRasterLayer ( myRasterFileInfo.filePath(),
            myRasterFileInfo.completeBaseName() );
}

void TestQgsRasterLayer::cleanupTestCase()
{
  delete mpLayer;
}
```

The above init function illustrates a couple of interesting things.

1. I needed to manually set the QGIS application data path so that resources such as srs.db can be found properly. 2. Secondly, this is a data driven test so we needed to provide a way to generically locate the 'tenbytenraster.asc file. This was achieved by using the compiler define **TEST_DATA_PATH**. The define is created in the CMakeLists.txt configuration file under <QGIS Source Root>/tests/CMakeLists.txt and is available to all QGIS unit tests. If you need test data for your test, commit it under <QGIS Source Root>/tests/testdata. You should only commit very small datasets here. If your test needs to modify the test data, it should make a copy of if first.

Qt also provides some other interesting mechanisms for data driven testing, so if you are interested to know more on the topic, consult the Qt documentation.

Next lets look at our functional test. The isValid() test simply checks the raster layer was correctly loaded in the initTestCase. QVERIFY is a Qt macro that you can use to evaluate a test condition. There are a few other use macros Qt provide for use in your tests including:

Listing

```
QCOMPARE ( actual, expected )
QEXPECT_FAIL ( dataIndex, comment, mode )
QFAIL ( message )
QFETCH ( type, name )
QSKIP ( description, mode )
QTEST ( actual, testElement )
QTEST_APPLESS_MAIN ( TestClass )
QTEST_MAIN ( TestClass )
QTEST_NOOP_MAIN ()
QVERIFY2 ( condition, message )
QVERIFY ( condition )
QWARN ( message )
```

Some of these macros are useful only when using the Qt framework for data driven testing (see the Qt docs for more detail).

Listing

```
void TestQgsRasterLayer::isValid()
{
  QVERIFY ( mpLayer->isValid() );
}
```

Normally your functional tests would cover all the range of functionality of your classes public API where feasible. With our functional tests out the way, we can look at our regression test example.

Since the issue in bug #832 is a misreported cell count, writing our test if simply a matter of using QVERIFY to check that the cell count meets the expected value:

Listing

```
void TestQgsRasterLayer::regression832()
{
   QVERIFY ( mpLayer->getRasterXDim() == 10 );
   QVERIFY ( mpLayer->getRasterYDim() == 10 );
   // regression check for ticket #832
   // note getRasterBandStats call is base 1
   QVERIFY ( mpLayer->getRasterBandStats(1).elementCountInt == 100 );
}
```

With all the unit test functions implemented, there one final thing we need to add to our test class:

Listing

```
QTEST_MAIN(TestQgsRasterLayer)
#include "moc_testqgsrasterlayer.cxx"
```

The purpose of these two lines is to signal to Qt's moc that his is a QtTest (it will generate a main method that in turn calls each test funtion. The last line is the include for the MOC generated sources. You should replace 'testqgsrasterlayer' with the name of your class in lower case.

3.3 Adding your unit test to CMakeLists.txt

Adding your unit test to the build system is simply a matter of editing the CMakeLists.txt in the test directory, cloning one of the existing test blocks, and then replacing your test class name into it. For example:

Listing

```
# QgsRasterLayer test
ADD_QGIS_TEST(rasterlayertest testqgsrasterlayer.cpp)
```


3.4 The ADD_QGIS_TEST macro explained

I'll run through these lines briefly to explain what they do, but if you are not interested, just do the step explained in the above section and section.

Listing

```
MACRO (ADD_QGIS_TEST testname testsrc)
  SET(qgis_${testname}_SRCS ${testsrc} ${util_SRCS})
  SET(qgis_${testname}_MOC_CPPS ${testsrc})
  QT4_WRAP_CPP(qgis_${testname}_MOC_SRCS ${qgis_${testname}_MOC_CPPS})
  ADD_CUSTOM_TARGET(qgis_${testname}moc ALL DEPENDS ${qgis_${testname}_MOC_SRCS})
  ADD_EXECUTABLE(qgis_${testname} ${qgis_${testname}_SRCS})
  ADD_DEPENDENCIES(qgis_${testname} qgis_${testname}moc)
  TARGET_LINK_LIBRARIES(qgis_${testname} ${QT_LIBRARIES} qgis_core)
  SET_TARGET_PROPERTIES(qgis_${testname}
    PROPERTIES
    # skip the full RPATH for the build tree
    SKIP_BUILD_RPATH  TRUE
    # when building, use the install RPATH already
    # (so it doesn't need to relink when installing)
    BUILD_WITH_INSTALL_RPATH TRUE
    # the RPATH to be used when installing
    INSTALL_RPATH ${QGIS_LIB_DIR}
    # add the automatically determined parts of the RPATH
    # which point to directories outside the build tree to the install RPATH
    INSTALL_RPATH_USE_LINK_PATH true)
  IF (APPLE)
    # For Mac OS X, the executable must be at the root of the bundle's executable folder
    INSTALL(TARGETS qgis_${testname} RUNTIME DESTINATION ${CMAKE_INSTALL_PREFIX})
    ADD_TEST(qgis_${testname} ${CMAKE_INSTALL_PREFIX}/qgis_${testname})
  ELSE (APPLE)
    INSTALL(TARGETS qgis_${testname} RUNTIME DESTINATION ${CMAKE_INSTALL_PREFIX}/bin)
    ADD_TEST(qgis_${testname} ${CMAKE_INSTALL_PREFIX}/bin/qgis_${testname})
  ENDIF (APPLE)
ENDMACRO (ADD_QGIS_TEST)
```

Lets look a little more in detail at the individual lines. First we define the list of sources for our test. Since we have only one source file (following the methodology I described above where class declaration and definition are in the same file) its a simple statement:

Listing

```
SET(qgis_${testname}_SRCS ${testsrc} ${util_SRCS})
```

Since our test class needs to be run through the Qt meta object compiler (moc) we need to provide a couple of lines to make that happen too:

Listing

```
SET(qgis_${testname}_MOC_CPPS ${testsrc})
QT4_WRAP_CPP(qgis_${testname}_MOC_SRCS ${qgis_${testname}_MOC_CPPS})
ADD_CUSTOM_TARGET(qgis_${testname}moc ALL DEPENDS ${qgis_${testname}_MOC_SRCS})
```


Next we tell cmake that it must make an executeable from the test class. Remember in the previous section on the last line of the class implementation I included the moc outputs directly into our test class, so that will give it (among other things) a main method so the class can be compiled as an executeable:

Listing

```
ADD_EXECUTABLE(qgis_${testname} ${qgis_${testname}_SRCS})
ADD_DEPENDENCIES(qgis_${testname} qgis_${testname}moc)
```

Next we need to specify any library dependencies. At the moment classes have been implemented with a catch-all QT_LIBRARIES dependency, but I will be working to replace that with the specific Qt libraries that each class needs only. Of course you also need to link to the relevant qgis libraries as required by your unit test.

Listing

```
TARGET_LINK_LIBRARIES(qgis_${testname} ${QT_LIBRARIES} qgis_core)
```

Next I tell cmake to install the tests to the same place as the qgis binaries itself. This is something I plan to remove in the future so that the tests can run directly from inside the source tree.

Listing

```
SET_TARGET_PROPERTIES(qgis_${testname}
  PROPERTIES
  # skip the full RPATH for the build tree
  SKIP_BUILD_RPATH  TRUE
  # when building, use the install RPATH already
  # (so it doesn't need to relink when installing)
  BUILD_WITH_INSTALL_RPATH TRUE
  # the RPATH to be used when installing
  INSTALL_RPATH ${QGIS_LIB_DIR}
  # add the automatically determined parts of the RPATH
  # which point to directories outside the build tree to the install RPATH
  INSTALL_RPATH_USE_LINK_PATH true)
IF (APPLE)
  # For Mac OS X, the executable must be at the root of the bundle's executable folder
  INSTALL(TARGETS qgis_${testname} RUNTIME DESTINATION ${CMAKE_INSTALL_PREFIX})
  ADD_TEST(qgis_${testname} ${CMAKE_INSTALL_PREFIX}/qgis_${testname})
ELSE (APPLE)
  INSTALL(TARGETS qgis_${testname} RUNTIME DESTINATION ${CMAKE_INSTALL_PREFIX}/bin)
  ADD_TEST(qgis_${testname} ${CMAKE_INSTALL_PREFIX}/bin/qgis_${testname})
ENDIF (APPLE)
```

Finally the above uses ADD_TEST to register the test with cmake / ctest . Here is where the best magic happens - we register the class with ctest. If you recall in the overview I gave in the beginning of this section we are using both QtTest and CTest together. To recap, **QtTest** adds a main method to your test unit and handles calling your test methods within the class. It also provides some macros like QVERIFY that you can use as to test for failure of the tests using conditions. The output from a QtTest

unit test is an executeable which you can run from the command line. However when you have a suite of tests and you want to run each executeable in turn, and better yet integrate running tests into the build process, the **CTest** is what we use.

3.5 Building your unit test

To build the unit test you need only to make sure that ENABLE_TESTS=true in the cmake configuration. There are two ways to do this:

1. Run ccmake .. (cmakesetup .. under windows) and interactively set the ENABLE_TESTS flag to ON. 1. Add a command line flag to cmake e.g. cmake -DENABLE_TESTS=true ..

Other than that, just build QGIS as per normal and the tests should build too.

3.6 Run your tests

The simplest way to run the tests is as part of your normal build process:

Listing

```
make && make install && make test
```

The make test command will invoke CTest which will run each test that was registered using the ADD_TEST CMake directive described above. Typical output from make test will look like this:

Listing

```
Running tests...
Start processing tests
Test project /Users/tim/dev/cpp/qgis/build
1/  3 Testing qgis_applicationtest          ***Exception: Other
2/  3 Testing qgis_filewritertest           *** Passed
3/  3 Testing qgis_rasterlayertest          *** Passed

0% tests passed, 3 tests failed out of 3

  The following tests FAILED:
  1 - qgis_applicationtest (OTHER_FAULT)
  Errors while running CTest
  make: *** [test] Error 8
```

If a test fails, you can use the ctest command to examine more closely why it failed. User the -R option to specify a regex for which tests you want to run and -V to get verbose output:

Listing

```
[build] ctest -R appl -V
Start processing tests
Test project /Users/tim/dev/cpp/qgis/build
Constructing a list of tests
Done constructing a list of tests
Changing directory into /Users/tim/dev/cpp/qgis/build/tests/src/core
1/  3 Testing qgis_applicationtest
Test command: /Users/tim/dev/cpp/qgis/build/tests/src/core/qgis_applicationtest
********* Start testing of TestQgsApplication *********
  Config: Using QTest library 4.3.0, Qt 4.3.0
PASS   : TestQgsApplication::initTestCase()
  Prefix  PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/../
  Plugin  PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//lib/qgis
  PkgData PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//share/qgis
  User DB PATH: /Users/tim/.qgis/qgis.db
PASS   : TestQgsApplication::getPaths()
  Prefix  PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/../
  Plugin  PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//lib/qgis
  PkgData PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//share/qgis
  User DB PATH: /Users/tim/.qgis/qgis.db
  QDEBUG : TestQgsApplication::checkTheme() Checking if a theme icon exists:
  QDEBUG : TestQgsApplication::checkTheme()
  /Users/tim/dev/cpp/qgis/build/tests/src/core/..//share/qgis/themes/default//mIconProjectionDisabled.png
  FAIL!  : TestQgsApplication::checkTheme() '!myPixmap.isNull()' returned FALSE. ()
  Loc: [/Users/tim/dev/cpp/qgis/tests/src/core/testqgsapplication.cpp(59)]
PASS   : TestQgsApplication::cleanupTestCase()
  Totals: 3 passed, 1 failed, 0 skipped
  ********* Finished testing of TestQgsApplication *********
  -- Process completed
  ***Failed

  0% tests passed, 1 tests failed out of 1

  The following tests FAILED:
1 - qgis_applicationtest (Failed)
  Errors while running CTest
```

Well that concludes this section on writing unit tests in QGIS. We hope you will get into the habit of writing test to test new functionality and to check for regressions. Some aspects of the test system (in particular the CMakeLists.txt parts) are still being worked on so that the testing framework works in a truly platform way. I will update this document as things progress.

4 HIG (Human Interface Guidelines)

In order for all graphical user interface elements to appear consistant and to all the user to instinctively use dialogs, it is important that the following guidelines are followed in layout and design of GUIs.

1. Group related elements using group boxes: Try to identify elements that can be grouped together and then use group boxes with a label to identify the topic of that group. Avoid using group boxes with only a single widget / item inside.
2. Capitalise first letter only in labels: Labels (and group box labels) should be written as a phrase with leading capital letter, and all remaing words written with lower case first letters
3. Do not end labels for widgets or group boxes with a colon: Adding a colon causes visual noise and does not impart additional meaning, so don't use them. An exception to this rule is when you have two labels next to each other e.g.: Label1 Plugin Label2 [/path/to/plugins]
4. Keep harmful actions away from harmless ones: If you have actions for 'delete', 'remove' etc, try to impose adequate space between the harmful action and innocuous actions so that the users is less likely to inadvertantly click on the harmful action.
5. Always use a QButtonBox for 'OK', 'Cancel' etc buttons: Using a button box will ensure that the order of 'OK' and 'Cancel' etc, buttons is consistent with the operating system / locale / desktop environment that the user is using.
6. Tabs should not be nested. If you use tabs, follow the style of the tabs used in QgsVectorLayerProperties / QgsProjectProperties etc. i.e. tabs at top with icons at 22x22.
7. Widget stacks should be avoided if at all possible. They cause problems with layouts and inexplicable (to the user) resizing of dialogs to accommodate widgets that are not visible.
8. Try to avoid technical terms and rather use a laymans equivalent e.g. use the word 'Transparency' rather than 'Alpha Channel' (contrived example), 'Text' instead of 'String' and so on.
9. Use consistent iconography. If you need an icon or icon elements, please contact Robert Szczepanek on the mailing list for assistance.
10. Place long lists of widgets into scroll boxes. No dialog should exceed 580 pixels in height and 1000 pixels in width.
11. Separate advanced options from basic ones. Novice users should be able to quickly access the items needed for basic activities without needing to concern themselves with complexity of advanced features. Advanced features should either be located below a dividing line, or placed onto a separate tab.
12. Don't add options for the sake of having lots of options. Strive to keep the user interface minimalistic and use sensible defaults.
13. If clicking a button will spawn a new dialog, an ellipsis (...) should be suffixed to the button text.

5 Authors

- Tim Sutton (author and editor)
- Gary Sherman
- Marco Hugentobler

Original pages from wiki to deprecate:

- http://wiki.qgis.org/qgiswiki/CodingGuidelines (./)
- http://wiki.qgis.org/qgiswiki/CodingStandards (./)
- http://wiki.qgis.org/qgiswiki/UsingSubversion (./)
- http://www.qgis.org/wiki/How_to_debug_QGIS_Plugins
- http://wiki.qgis.org/qgiswiki/DevelopmentInBranches (./)
- http://wiki.qgis.org/qgiswiki/SubmittingPatchesAndSvnAccess (./)