QGIS Developers Guide

Release 3.4

QGIS Project

15.03.2020

Contents

Welcome to the QGIS Development pages. Here you'll find rules, tools and steps to easily and efficiently contribute to QGIS code.

CHAPTER 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 camel case.

Examples:

- `QgsPoint`
- `QgsMapCanvas`
- `QgsRasterLayer`

1.1.2 Members

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

- `mMapCanvas`
- `mCurrentExtent`

All class members should be private. Public class members are STRONGLY discouraged. Protected members should be avoided when the member may need to be accessed from Python subclasses, since protected members cannot be used from the Python bindings.

Mutable static class member names should begin with a lower case `s`, but constant static class member names should be all caps:

- `sRefCounter`
- `DEFAULT_QUEUE_SIZE`

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:

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

Ensure that accessors are correctly marked with `const`. Where appropriate, this may require that cached value type member variables are marked with `mutable`.

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.

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

For consistency with the existing QGIS API and with the Qt API, abbreviations should be avoided. E.g. `setDestinationSize` instead of `setDestSize`, `setMaximumValue` instead of `setMaxVal`.

Acronyms should also be camel cased for consistency. E.g. `setXml` instead of `setXML`.

1.1.5 Function Arguments

Function arguments should use descriptive names. Do not use single letter arguments (e.g. `setColor( const QColor& color )` instead of `setColor( const QColor& c )`).

Pay careful attention to when arguments should be passed by reference. Unless argument objects are small and trivially copied (such as QPoint objects), they should be passed by const reference. For consistency with the Qt API, even implicitly shared objects are passed by const reference (e.g. `setTitle( const QString& title )` instead of `setTitle( QString title )`.

1.1.6 Function Return Values

Return small and trivially copied objects as values. Larger objects should be returned by const reference. The one exception to this is implicitly shared objects, which are always returned by value. Return `QObject` or subclassed objects as pointers.

- `int maximumValue() const`
- `const LayerSet& layers() const`
- `QString title() const` (`QString` is implicitly shared)
- `QList< QgsMapLayer* > layers() const` (`QList` is implicitly shared)
- `QgsVectorLayer *layer() const;` (`QgsVectorLayer` inherits `QObject`)
- `QgsAbstractGeometry *geometry() const;` (`QgsAbstractGeometry` is abstract and will probably need to be casted)

1.2 API Documentation

It is required to write API documentation for every class, method, enum and other code that is available in the public API.

QGIS uses Doxygen for documentation. Write descriptive and meaningful comments that give a reader information about what to expect, what happens in edge cases and give hints about other interfaces he could be looking for, best best practice and code samples.

1.2.1 Methods

Method descriptions should be written in a descriptive form, using the 3rd person. Methods require a `\since` tag that defines when they have been introduced. You should add additional `\since` tags for important changes that were introduced later on.

```
/**
 * Cleans the laundry by using water and fast rotation.
 * It will use the provided \a detergent during the washing programme.
 *
 * \returns True if everything was successful. If false is returned, use
 * \link error() \endlink to get more information.
 *
 * \note Make sure to manually call dry() after this method.
 *
 * \since QGIS 3.0
 * \see dry()
 */
```

1.2.2 Members Variables

Member variables should normally be in the `private` section and made available via getters and setters. One exception to this is for data containers like for error reporting. In such cases do not prefix the member with an `m`.

```
/**
 * \ingroup core
 * Represents points on the way along the journey to a destination.
 *
 * \since QGIS 2.20
 */
class QgsWaypoint
{
  /**
   * Holds information about results of an operation on a QgsWaypoint.
   *
   * \since QGIS 3.0
   */
  struct OperationResult
  {
    QgsWaypoint::ResultCode resultCode; //!< Indicates if the operation completed␣
↪successfully.
    QString message; //!< A human readable localized error message. Only set if␣
↪the resultCode is not QgsWaypoint::Success.
    QVariant result; //!< The result of the operation. The content depends on the␣
↪method that returned it. \since QGIS 3.2
  };
};
```

1.3 Qt Designer

1.3.1 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.

Examples:

- `QgsPluginManagerBase`
- `QgsUserOptionsBase`

1.3.2 Dialogs

All dialogs should implement tooltip help for all toolbar icons and other relevant widgets. Tooltips add greatly to feature discoverability for both new and experienced users.

Ensure that the tab order for widgets is updated whenever the layout of a dialog changes.

1.4 C++ Files

1.4.1 Names

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

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

Muista: 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.

1.4.2 Standard Header and License

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

```
/***************************************************************************
  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.
 *
 ***************************************************************************/
```

Muista: There is a template for Qt Creator in git. To use it, copy it from `doc/qt_creator_license_template` to a local location, adjust the mail address and - if required - the name and configure QtCreator to use it: *Tools* → *Options* → *C++* → *File Naming*.

1.5 Variable Names

Local variable names begin with a lower case letter and are formed using mixed case. Do not use prefixes like `my` or `the`.

Examples:

- `mapCanvas`
- `currentExtent`

1.6 Enumerated Types

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

```
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`

1.7 Global Constants & Macros

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

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

1.8 Comments

Comments to class methods should use a third person indicative style instead of the imperative style:

```
/**
 * Creates a new QgsFeatureFilterModel, optionally specifying a \a parent.
 */
explicit QgsFeatureFilterModel( QObject *parent = nullptr );
~QgsFeatureFilterModel() override;
```

1.9 Qt Signals and Slots

All signal/slot connects should be made using the "new style" connects available in Qt5. Futher information on this requirement is available in QEP #77.

Avoid use of Qt auto connect slots (i.e. those named `void on_mSpinBox_valueChanged`). Auto connect slots are fragile and prone to breakage without warning if dialogs are refactored.

1.10 Editing

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

1.10.1 Tabs

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

Muista: In vim this is done with `set expandtab ts=2`

1.10.2 Indentation

Source code should be indented to improve readability. There is a `scripts/prepare-commit.sh` that looks up the changed files and reindents them using astyle. This should be run before committing. You can also use `scripts/astyle.sh` to indent individual files.

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 (enable `WITH_ASTYLE` in cmake to include it in the build).

1.10.3 Braces

Braces should start on the line following the expression:

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

1.11 API Compatibility

There is API documentation for C++.

We try to keep the API stable and backwards compatible. Cleanups to the API should be done in a manner similar to the Qt sourcecode e.g.

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

```
    Q_DECL_DEPRECATED bool doSomething();

    /**
     * Does something a better way.
     * \note added in 1.1
     */
    bool doSomethingBetter();

  signals:
    /**
     * This signal will is deprecated, you are encouraged to
     * connect to somethingHappenedBetter() rather.
     * \deprecated use somethingHappenedBetter()
     */
#ifndef Q_MOC_RUN
    Q_DECL_DEPRECATED
#endif
    bool somethingHappened();

    /**
     * Something happened
     * \note added in 1.1
     */
    bool somethingHappenedBetter();
}
```

1.12 SIP Bindings

Some of the SIP files are automatically generated using a dedicated script.

1.12.1 Header pre-processing

All the information to properly build the SIP file must be found in the C++ header file. Some macros are available for such definition:

- Use `#ifdef SIP_RUN` to generate code only in SIP files or `#ifndef SIP_RUN` for C++ code only. `#else` statements are handled in both cases.
- Use `SIP_SKIP` to discard a line
- The following annotations are handled:
  - `SIP_FACTORY`: `/Factory/`
  - `SIP_OUT`: `/Out/`
  - `SIP_INOUT`: `/In,Out/`
  - `SIP_TRANSFER`: `/Transfer/`
  - `SIP_PYNAME(name)`: `/PyName=name/`
  - `SIP_KEEPREFERENCE`: `/KeepReference/`
  - `SIP_TRANSFERTHIS`: `/TransferThis/`
  - `SIP_TRANSFERBACK`: `/TransferBack/`
- `private` sections are not displayed, except if you use a `#ifdef SIP_RUN` statement in this block.
- `SIP_PYDEFAULTVALUE(value)` can be used to define an alternative default value of the python method. If the default value contains a comma `,`, the value should be surrounded by single quotes `'`

- `SIP_PYTYPE(type)` can be used to define an alternative type for an argument of the python method. If the type contains a comma `,`, the type should be surrounded by single quotes `'`

A demo file can be found in `tests/scripts/sipifyheader.h`.

1.12.2 Generating the SIP file

The SIP file can be generated using a dedicated script. For instance:

```
scripts/sipify.pl src/core/qgsvectorlayer.h > python/core/qgsvectorlayer.sip
```

As soon as a SIP file is added to one of the source file (`python/core/core.sip`, `python/gui/gui.sip` or `python/analysis/analysis.sip`), it will be considered as generated automatically. A test on Travis will ensure that this file is up to date with its corresponding header.

Older files for which the automatic creation is not enabled yet are listed in `python/auto_sip.blacklist`.

1.12.3 Improving sipify script

If some improvements are required for sipify script, please add the missing bits to the demo file `tests/scripts/sipifyheader.h` and create the expected header `tests/scripts/sipifyheader.expected.si`. This will also be automatically tested on Travis as a unit test of the script itself.

1.13 Coding Style

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

1.13.1 Where-ever Possible Generalize Code

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

1.13.2 Prefer Having Constants First in Predicates

Prefer to put constants first in predicates.

`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.

1.13.3 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:

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

or this:

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

Muista: `scripts/prepare-commit.sh` will take care of this.

1.13.4 Put commands on separate lines

When reading code it's easy to miss commands, if they are not at the beginning of the line. When quickly reading through code, it's common to skip lines if they don't look like what you are looking for in the first few characters. It's also common to expect a command after a conditional like `if`.

Consider:

```
if (foo) bar();

baz(); bar();
```

It's very easy to miss part of what the flow of control. Instead use

```
if (foo)
  bar();

baz();
bar();
```

1.13.5 Indent access modifiers

Access modifiers structure a class into sections of public API, protected API and private API. Access modifiers themselves group the code into this structure. Indent the access modifier and declarations.

```
class QgsStructure
{
  public:
    /**
     * Constructor
     */
     explicit QgsStructure();
}
```

1.13.6 Book recommendations

- Effective Modern 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 designing Qt style (APIs)

1.14 Credits for contributions

Contributors of new functions are encouraged to let people know about their contribution by:

- adding a note to the changelog for the first version where the code has been incorporated, of the type:

```
This feature was funded by: Olmiomland https://olmiomland.ol
This feature was developed by: Chuck Norris https://chucknorris.kr
```

- writing an article about the new feature on a blog, and add it to the QGIS planet https://plugins.qgis.org/planet/
- adding their name to:
  - https://github.com/qgis/QGIS/blob/release-3_4/doc/CONTRIBUTORS
  - https://github.com/qgis/QGIS/blob/release-3_4/doc/AUTHORS

CHAPTER 2

HIG (Human Interface Guidelines)

In order for all graphical user interface elements to appear consistent 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, tool tips, descriptive text, and other non-heading or title text: These should be written as a phrase with leading capital letter, and all remaining words written with lower case first letters, unless they are nouns
3. Capitalize all words in Titles (group box, tab, list view columns, and so on), Functions (menu items, buttons), and other selectable items (combobox items, listbox items, tree list items, and so on): Capitalize all words, except prepositions that are shorter than five letters (for example, 'with' but 'Without'), conjunctions (for example, and, or, but), and articles (a, an, the). However, always capitalize the first and last word.
4. 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 (Path:) Label2 [/path/to/plugins]
5. 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 inadvertently click on the harmful action.
6. 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.
7. 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.
8. 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.
9. Try to avoid technical terms and rather use a laymans equivalent e.g. use the word 'Opacity' rather than 'Alpha Channel' (contrived example), 'Text' instead of 'String' and so on.
10. Use consistent iconography. If you need an icon or icon elements, please contact Robert Szczepanek on the mailing list for assistance.
11. Place long lists of widgets into scroll boxes. No dialog should exceed 580 pixels in height and 1000 pixels in width.

12. 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.
13. Don't add options for the sake of having lots of options. Strive to keep the user interface minimalistic and use sensible defaults.
14. If clicking a button will spawn a new dialog, an ellipsis char (…) should be suffixed to the button text. Note, make sure to use the U+2026 Horizontal Ellipsis char instead of three periods.

2.1 Authors

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

CHAPTER 3

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.

3.1 Installation

3.1.1 Install git for GNU/Linux

Debian based distro users can do:

```
sudo apt install git
```

3.1.2 Install git for Windows

Windows users can obtain msys git or use git distributed with cygwin.

3.1.3 Install git for OSX

The git project has a downloadable build of git. Make sure to get the package matching your processor (x86_64 most likely, only the first Intel Macs need the i386 package).

Once downloaded open the disk image and run the installer.

PPC/source note

The git site does not offer PPC builds. If you need a PPC build, or you just want a little more control over the installation, you need to compile it yourself.

Download the source from https://git-scm.com/. Unzip it, and in a Terminal cd to the source folder, then:

```
make prefix=/usr/local
sudo make prefix=/usr/local install
```

If you don't need any of the extras, Perl, Python or TclTk (GUI), you can disable them before running make with:

```
export NO_PERL=
export NO_TCLTK=
export NO_PYTHON=
```

3.2 Accessing the Repository

To clone QGIS master:

```
git clone git://github.com/qgis/QGIS.git
```

3.3 Check out a branch

To check out a branch, for example the release 2.6.1 branch do:

```
cd QGIS
git fetch
git branch --track origin release-2_6_1
git checkout release-2_6_1
```

To check out the master branch:

```
cd QGIS
git checkout master
```

Muista: 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.

3.4 QGIS documentation sources

If you're interested in checking out QGIS documentation sources:

```
git clone git@github.com:qgis/QGIS-Documentation.git
```

You can also take a look at the readme included with the documentation repo for more information.

3.5 QGIS website sources

If you're interested in checking out QGIS website sources:

```
git clone git@github.com:qgis/QGIS-Website.git
```

You can also take a look at the readme included with the website repo for more information.

3.6 GIT Documentation

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

- https://services.github.com/
- https://progit.org
- http://gitready.com

3.7 Development in branches

3.7.1 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 reestablish 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.

3.7.2 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.

```
git checkout -b 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:

```
git push origin newfeature
```

Muista: If the branch already exists your changes will be pushed into it.

Rebase to master regularly: It is recommended to rebase to incorporate the changes in master to the branch on a regular basis. This makes it easier to merge the branch back to master later. After a rebase you need to `git push -f` to your forked repo.

Muista: Never `git push -f` to the origin repository! Only use this for your working branch.

```
git rebase master
```

3.7.3 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.

3.8 Submitting Patches and Pull Requests

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

3.8.1 Pull Requests

In general it is easier for developers if you submit GitHub pull requests. We do not describe Pull Requests here, but rather refer you to the GitHub pull request documentation.

If you make a pull request we ask that you please merge master to your PR branch regularly so that your PR is always mergeable to the upstream master branch.

If you are a developer and wish to evaluate the pull request queue, there is a very nice tool that lets you do this from the command line

Please see the section below on 'getting your patch noticed'. In general when you submit a PR you should take the responsibility to follow it through to completion - respond to queries posted by other developers, seek out a 'champion' for your feature and give them a gentle reminder occasionally if you see that your PR is not being acted on. Please bear in mind that the QGIS project is driven by volunteer effort and people may not be able to attend to your PR instantaneously. If you feel the PR is not receiving the attention it deserves your options to accelerate it should be (in order of priority):

- Send a message to the mailing list 'marketing' your PR and how wonderful it will be to have it included in the code base.
- Send a message to the person your PR has been assigned to in the PR queue.
- Send a message to Marco Hugentobler (who manages the PR queue).
- Send a message to the project steering committee asking them to help see your PR incorporated into the code base.

Best practice for creating a pull request

- Always start a feature branch from current master.
- If you are coding a feature branch, don't "merge" anything into that branch, rather rebase as described in the next point to keep your history clean.
- Before you create a pull request do `git fetch origin` and `git rebase origin/master` (given origin is the remote for upstream and not your own remote, check your `.git/config` or do `git remote -v | grep github.com/qgis`).
- You may do a git rebase like in the last line repeatedly without doing any damage (as long as the only purpose of your branch is to get merged into master).
- Attention: After a rebase you need to `git push -f` to your forked repo. **CORE DEVS: DO NOT DO THIS ON THE QGIS PUBLIC REPOSITORY!**

Special labels to notify documentors

Besides common tags you can add to classify your PR, there are special ones you can use to automatically generate issue reports in QGIS-Documentation repository as soon as your pull request is merged:

- `[needs-docs]` to instruct doc writers to please add some extra documentation after a fix or addition to an already existing functionality.
- `[feature]` in case of new functionality. Filling a good description in your PR will be a good start.

Please devs use these labels (case insensitive) so doc writers have issues to work on and have an overview of things to do. BUT please also take time to add some text: either in the commit OR in the docs itself.

For merging a pull request

Option A:

- click the merge button (Creates a non-fast-forward merge)

Option B:

- Checkout the pull request
- Test (Also required for option A, obviously)
- checkout master, git merge pr/1234
- Optional: `git pull --rebase`: Creates a fast-forward, no "merge commit" is made. Cleaner history, but it is harder to revert the merge.
- `git push` (NEVER EVER use the -f option here)

3.9 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 GitHub.

If the bug is an enhancement or new feature, it's usually a good idea to create a ticket in GitHub first and then attach your patch.

3.10 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:

```
cd QGIS
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.

3.10.1 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 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 in the Technical Resources).

3.10.2 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.

3.11 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.

Muista: 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.

CHAPTER 4

Getting up and running with QtCreator and QGIS

QtCreator is a newish IDE from the makers of the Qt library. With QtCreator you can build any C++ project, but it's really optimised for people working on Qt(4) based applications (including mobile apps). Everything I describe below assumes you are running Ubuntu 11.04 'Natty'.

4.1 Installing QtCreator

This part is easy:

```
sudo apt-get install qtcreator qtcreator-doc
```

After installing you should find it in your gnome menu.

4.2 Setting up your project

I'm assuming you have already got a local QGIS clone containing the source code, and have installed all needed build dependencies etc. There are detailed instructions for *git access* and dependency installation.

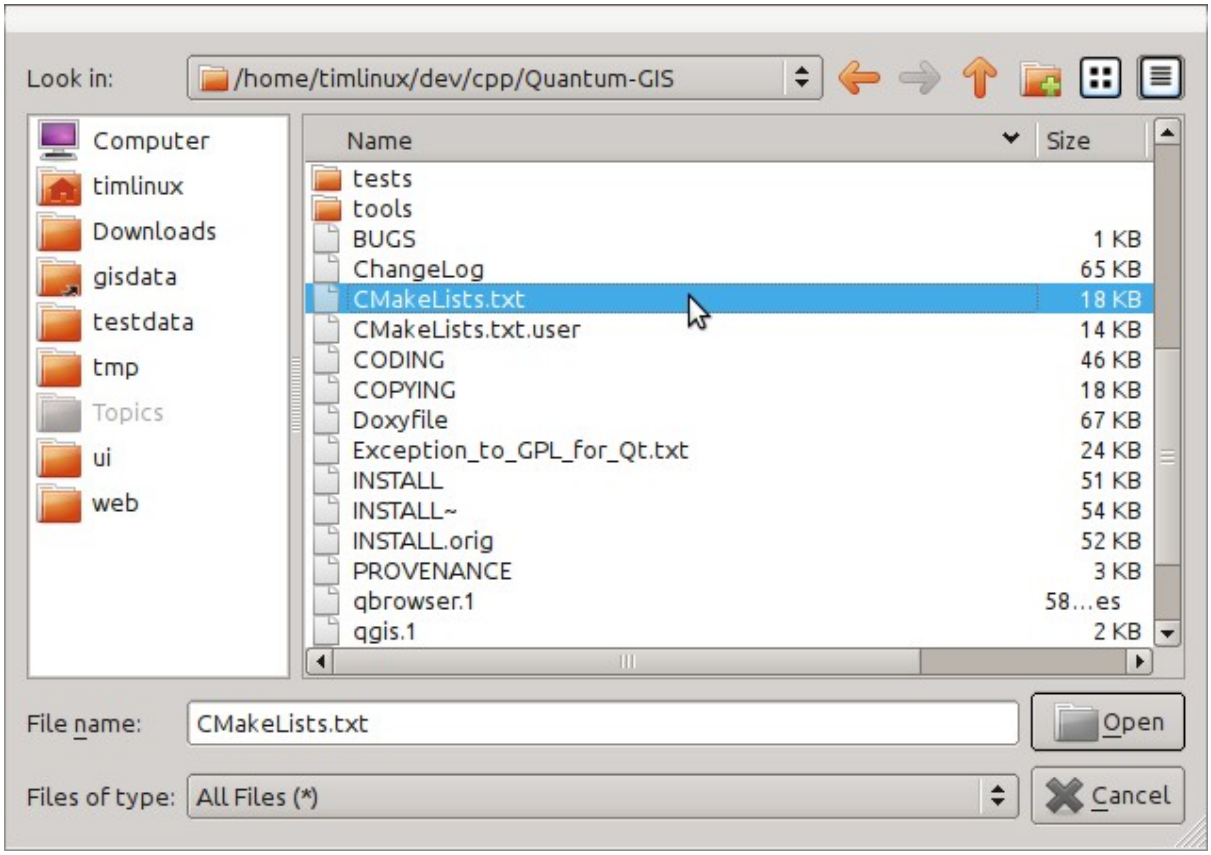
On my system I have checked out the code into `$HOME/dev/cpp/QGIS` and the rest of the article is written assuming that, you should update these paths as appropriate for your local system.

On launching QtCreator do:

File -> Open File or Project

Then use the resulting file selection dialog to browse to and open this file:

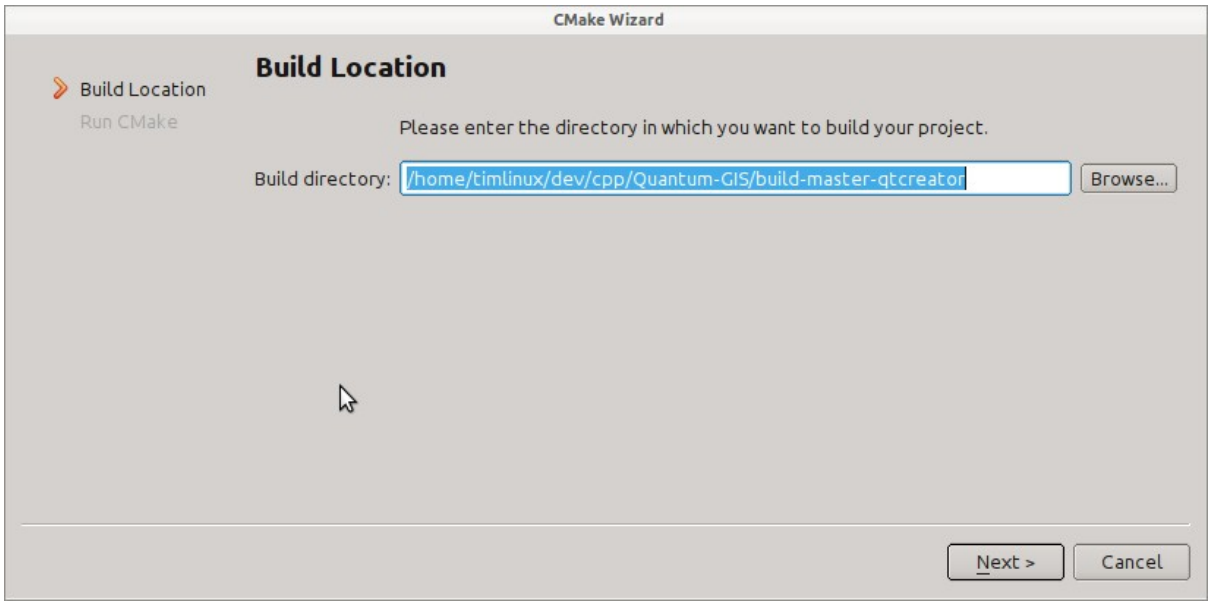
```
$HOME/dev/cpp/QGIS/CMakeLists.txt
```



Next you will be prompted for a build location. I create a specific build dir for QtCreator to work in under:

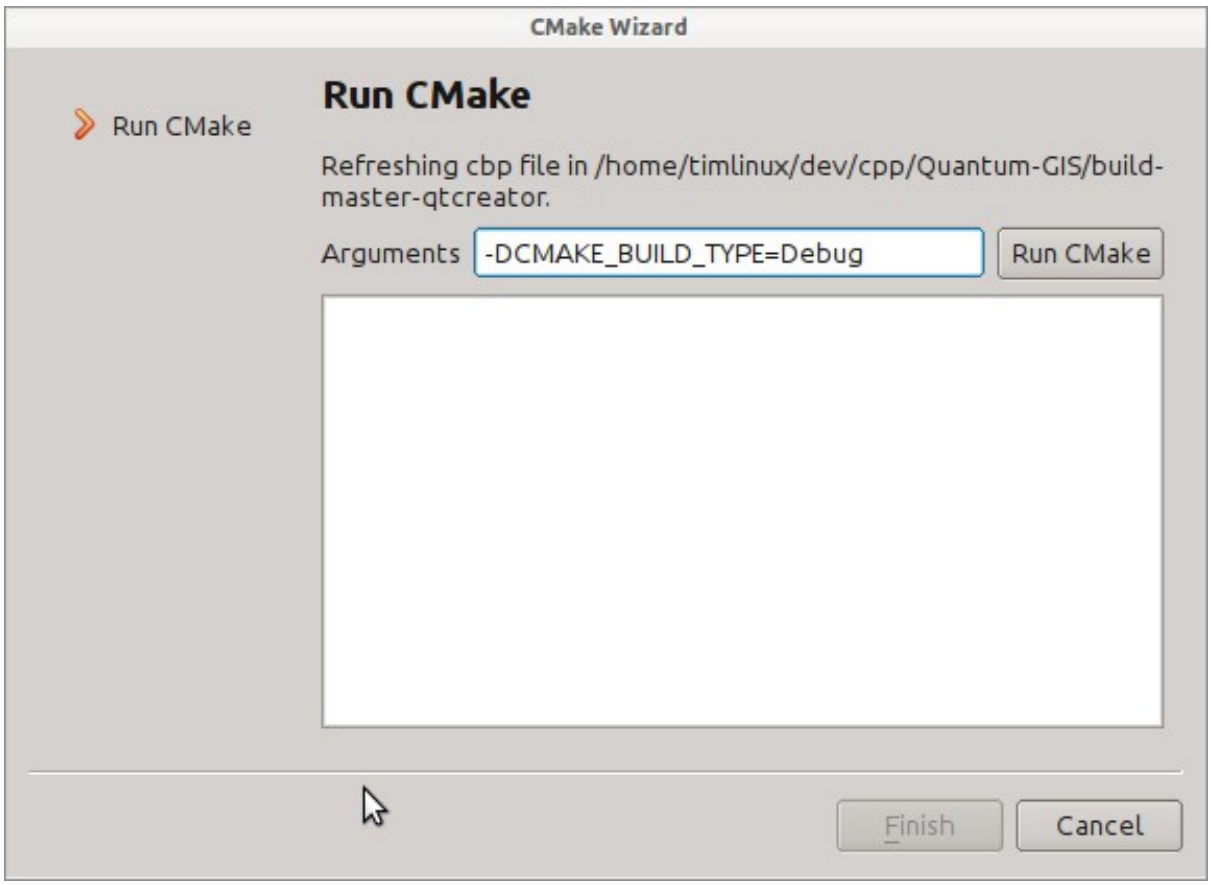
```
$HOME/dev/cpp/QGIS/build-master-qtcreator
```

Its probably a good idea to create separate build directories for different branches if you can afford the disk space.



Next you will be asked if you have any CMake build options to pass to CMake. We will tell CMake that we want a debug build by adding this option:

```
-DCMAKE_BUILD_TYPE=Debug
```

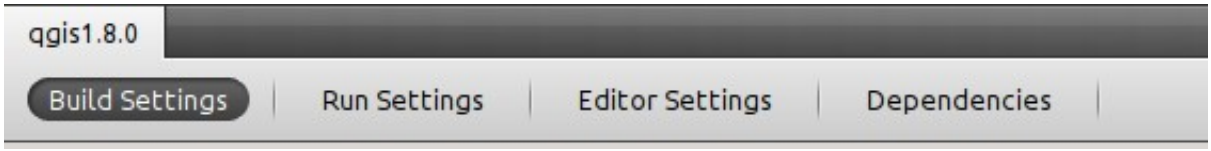


That's the basics of it. When you complete the Wizard, QtCreator will start scanning the source tree for autocompletion support and do some other housekeeping stuff in the background. We want to tweak a few things before we start to build though.

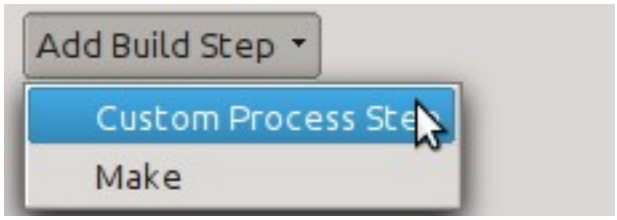
4.3 Setting up your build environment

Click on the 'Projects' icon on the left of the QtCreator window.

Select the build settings tab (normally active by default).



We now want to add a custom process step. Why? Because QGIS can currently only run from an install directory, not its build directory, so we need to ensure that it is installed whenever we build it. Under 'Build Steps', click on the 'Add BuildStep' combo button and choose 'Custom Process Step'.



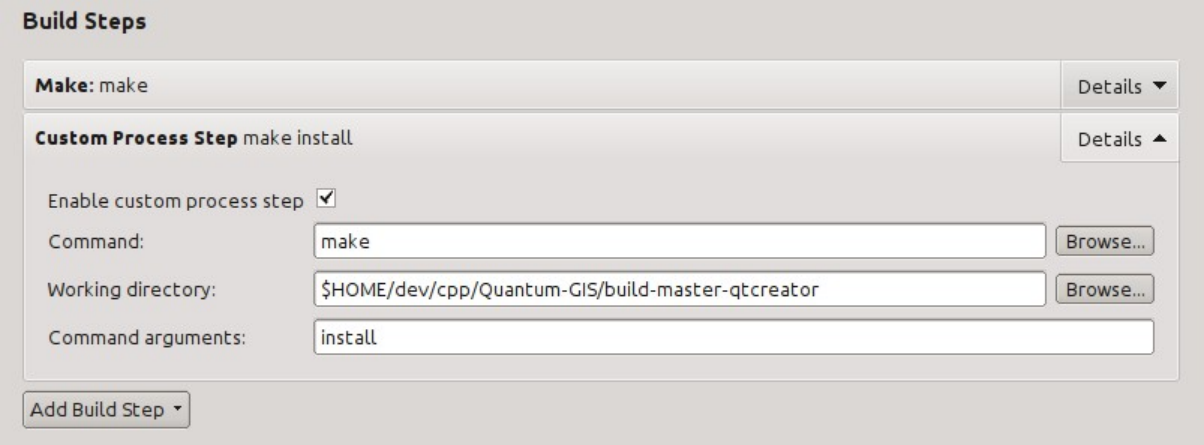
Now we set the following details:

Enable custom process step: [yes]

Command: make

Working directory: $HOME/dev/cpp/QGIS/build-master-qtcreator

Command arguments: install



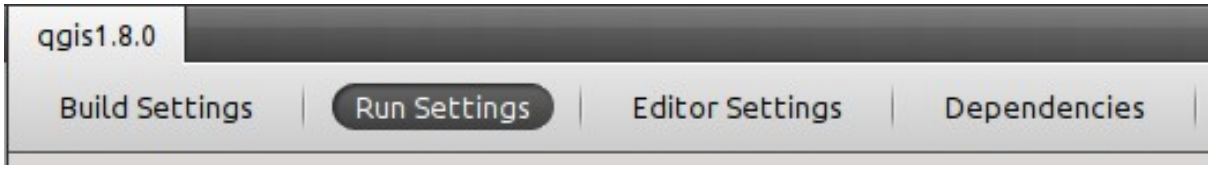
You are almost ready to build. Just one note: QtCreator will need write permissions on the install prefix. By default (which I am using here) QGIS is going to get installed to `/usr/local/`. For my purposes on my development machine, I just gave myself write permissions to the /usr/local directory.

To start the build, click that big hammer icon on the bottom left of the window.

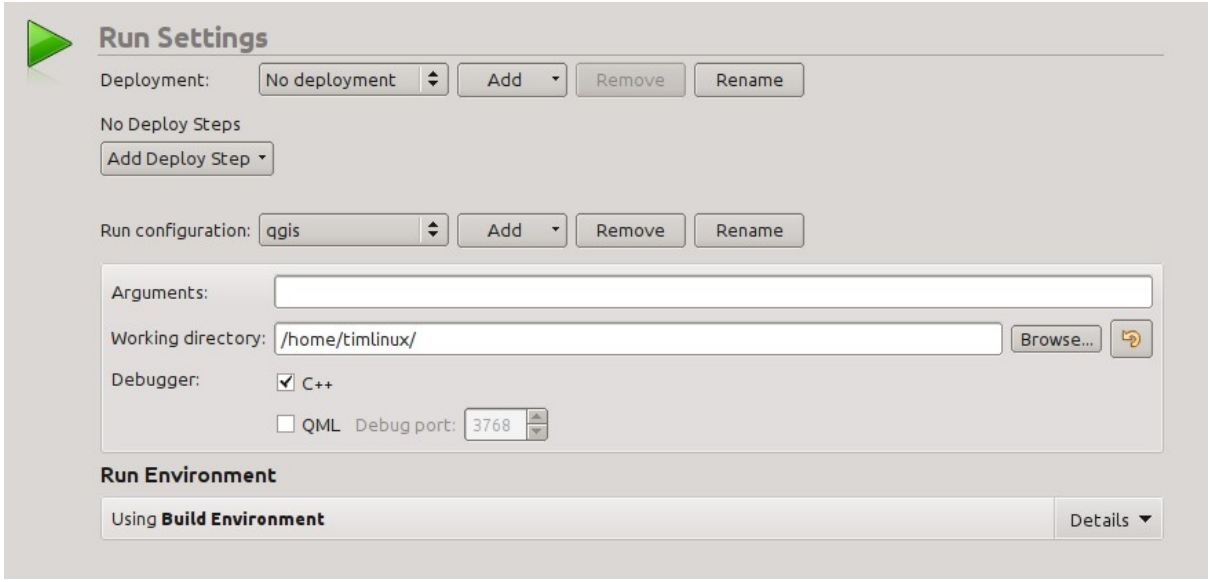
4.4 Setting your run environment

As mentioned above, we cannot run QGIS from directly in the build directly, so we need to create a custom run target to tell QtCreator to run QGIS from the install dir (in my case `/usr/local/`). To do that, return to the projects configuration screen.

Now select the 'Run Settings' tab



We need to update the default run settings from using the 'qgis' run configuration to using a custom one.



Do do that, click the 'Add v' combo button next to the Run configuration combo and choose 'Custom Executable' from the top of the list.

Now in the properties area set the following details:

Executable: /usr/local/bin/qgis

Arguments :

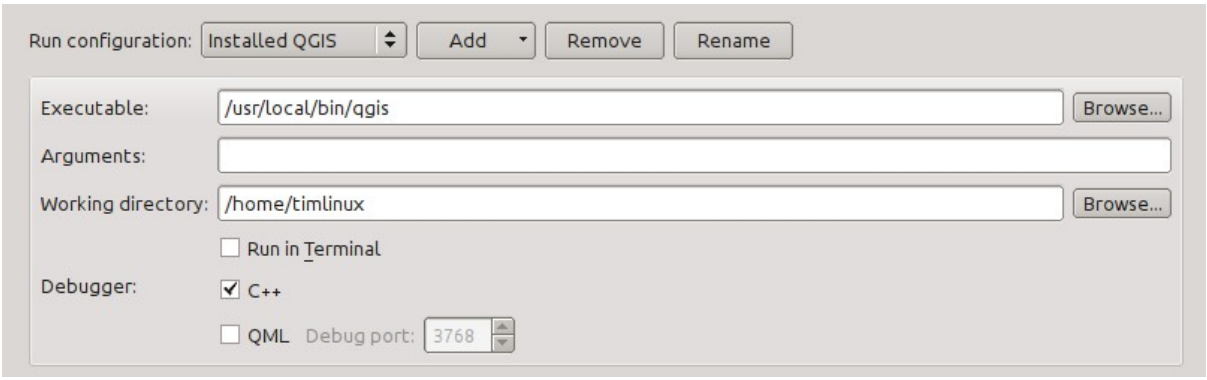
Working directory: $HOME

Run in terminal: [no]

Debugger: C++ [yes]

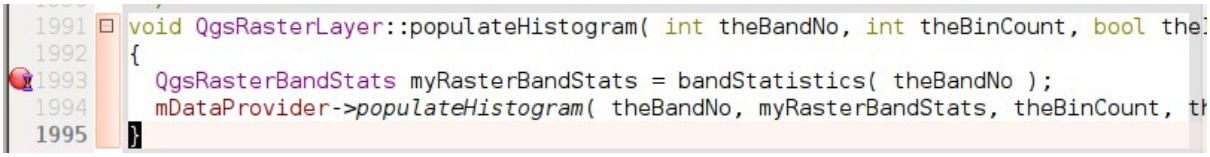
Qml [no]

Then click the 'Rename' button and give your custom executable a meaningful name e.g. 'Installed QGIS'

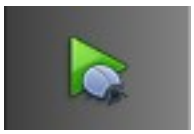


4.5 Running and debugging

Now you are ready to run and debug QGIS. To set a break point, simply open a source file and click in the left column.



Now launch QGIS under the debugger by clicking the icon with a bug on it in the bottom left of the window.



CHAPTER 5

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.

5.1 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 we delve into the details:

1. 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 practice 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.
2. 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.

3. 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.
4. You add a section to the CMakeLists.txt in your tests directory that will build your test.
5. You ensure you have `ENABLE_TESTING` enabled in ccmake / cmakesetup. This will ensure your tests actually get compiled when you type make.
6. 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 make a temporary copy somewhere if needed.
7. You compile your sources and install. Do this using normal `make && (sudo) make install` procedure.
8. You run your tests. This is normally done simply by doing `make test` after the `make install` step, though we will explain other approaches that offer more fine grained control over running tests.

Right with that overview in mind, we will delve into a bit of detail. We'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!

5.2 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. We'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:

```
/***************************************************************************
 testqgsvectorfilewriter.cpp
 ---------------------------------------
  Date : Friday, Jan 27, 2015
  Copyright: (C) 2015 by Tim Sutton
  Email: tim@kartoza.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 start our includes needed for the tests we plan to run. There is one special include all tests should have:

```
#include <QtTest/QtTest>
```

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

```
//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 and the class must inherit from QObject and include the Q_OBJECT macro.

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

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.

```
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. We are implementing two methods here which illustrate two types of testing.

In the first case we want to generally test if the various parts of the class are working, We 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 successfully 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 accessor or mutator is not working as expected you would normally implement a *regression test* to check for this.

```
//
// Functional Testing
//

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

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

5.2.1 Implementing a regression test

Next we implement our regression tests. Regression tests should be implemented to replicate the conditions of a particular bug. For example:

1. We received a report by email that the cell count by rasters was off by 1, throwing off all the statistics for the raster bands.
2. We opened a bug report (ticket #832)
3. We created a regression test that replicated the bug using a small test dataset (a 10x10 raster).
4. We ran the test, verifying that it did indeed fail (the cell count was 99 instead of 100).
5. Then we went to fix the bug and reran the unit test and the regression test passed. We committed the regression test along with the bug fix. Now if anybody breakes this in the source code again in the future, we can immediately 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 benefit 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 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.

```
//
// 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 your test class declaration you can declare privately any data members and helper methods your unit test may need. In our case we 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 won't be automatically run by the QTest executable that is created when we compile our test.

```
  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:

```
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 << "PrefixPATH: " << QgsApplication::prefixPath().toLocal8Bit().data()
↪<< std::endl;
  std::cout << "PluginPATH: " << 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. We 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 it 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:

- 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).

```
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 is simply a matter of using QVERIFY to check that the cell count meets the expected value:

```
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's one final thing we need to add to our test class:

```
QTEST_MAIN(TestQgsRasterLayer)
#include "testqgsrasterlayer.moc"
```

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

5.3 Comparing images for rendering tests

Rendering images on different environments can produce subtle differences due to platform-specific implementations (e.g. different font rendering and antialiasing algorithms), to the fonts available on the system and for other obscure reasons.

When a rendering test runs on Travis and fails, look for the dash link at the very bottom of the Travis log. This link will take you to a cdash page where you can see the rendered vs expected images, along with a "difference" image which highlights in red any pixels which did not match the reference image.

The QGIS unit test system has support for adding "mask" images, which are used to indicate when a rendered image may differ from the reference image. A mask image is an image (with the same name as the reference image, but including a **_mask.png** suffix), and should be the same dimensions as the reference image. In a mask image the pixel values indicate how much that individual pixel can differ from the reference image, so a black pixel indicates that the pixel in the rendered image must exactly match the same pixel in the reference image. A pixel with RGB 2, 2, 2 means that the rendered image can vary by up to 2 in its RGB values from the reference image, and a fully white pixel (255, 255, 255) means that the pixel is effectively ignored when comparing the expected and rendered images.

A utility script to generate mask images is available as `scripts/generate_test_mask_image.py`. This script is used by passing it the path of a reference image (e.g. `tests/testdata/control_images/annotations/expected_annotation_fillstyle/expected_annotation_fillstyle.png`) and the path to your rendered image.

E.g.

```
scripts/generate_test_mask_image.py tests/testdata/control_images/annotations/
↪expected_annotation_fillstyle/expected_annotation_fillstyle.png /tmp/path_to_
↪rendered_image.png
```

You can shortcut the path to the reference image by passing a partial part of the test name instead, e.g.

```
scripts/generate_test_mask_image.py annotation_fillstyle /tmp/path_to_rendered_
↪image.png
```

(This shortcut only works if a single matching reference image is found. If multiple matches are found you will need to provide the full path to the reference image.)

The script also accepts http urls for the rendered image, so you can directly copy a rendered image url from the cdash results page and pass it to the script.

Be careful when generating mask images - you should always view the generated mask image and review any white areas in the image. Since these pixels are ignored, make sure that these white images do not cover any important portions of the reference image – otherwise your unit test will be meaningless!

Similarly, you can manually "white out" portions of the mask if you deliberately want to exclude them from the test. This can be useful e.g. for tests which mix symbol and text rendering (such as legend tests), where the unit test is not designed to test the rendered text and you don't want the test to be subject to cross-platform text rendering differences.

To compare images in QGIS unit tests you should use the class `QgsMultiRenderChecker` or one of its subclasses.

To improve tests robustness here are few tips:

1. Disable antialiasing if you can, as this minimizes cross-platform rendering differences.
2. Make sure your reference images are "chunky"... i.e. don't have 1 px wide lines or other fine features, and use large, bold fonts (14 points or more is recommended).
3. Sometimes tests generate slightly different sized images (e.g. legend rendering tests, where the image size is dependent on font rendering size - which is subject to cross-platform differences). To account for this, use `QgsMultiRenderChecker::setSizeTolerance()` and specify the maximum number of pixels that the rendered image width and height differ from the reference image.
4. Don't use transparent backgrounds in reference images (CDash does not support them). Instead, use `QgsMultiRenderChecker::drawBackground()` to draw a checkboard pattern for the reference image background.
5. When fonts are required, use the font specified in `QgsFontUtils::standardTestFontFamily()` ("QGIS Vera Sans").

5.4 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:

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

5.4.1 The ADD_QGIS_TEST macro explained

We'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.

```
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_RPATHTRUE
# 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)
```

Let's 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 described above where class declaration and definition are in the same file) its a simple statement:

```
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:

```
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 executable from the test class. Remember in the previous section on the last line of the class implementation we 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 executable:

```
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 we 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.

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

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

```
SET_TARGET_PROPERTIES(qgis_${testname}
PROPERTIES
# skip the full RPATH for the build tree
SKIP_BUILD_RPATHTRUE
# 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 we 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 executable which you can run from the command line. However when you have a suite of tests and you want to run each executable in turn, and better yet integrate running tests into the build process, the CTest is what we use.

5.5 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 ..` (or `cmakesetup ..` under windows) and interactively set the `ENABLE_TESTS` flag to `ON`.
2. 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.

5.6 Run your tests

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

```
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:

```
Running tests...
Start processing tests
Test project /Users/tim/dev/cpp/qgis/build
## 13 Testing qgis_applicationtest***Exception: Other
## 23 Testing qgis_filewritertest *** Passed
## 33 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. Use the `-R` option to specify a regex for which tests you want to run and `-V` to get verbose output:

```
$ 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
## 13 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()
PrefixPATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/../
PluginPATH: /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()
PrefixPATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/../
PluginPATH: /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
```

5.6.1 Debugging unit tests

For C++ unit tests, QtCreator automatically adds run targets, so you can start them in the debugger.

It's also possible to start Python unit tests from QtCreator with GDB. For this, you need to go to *Projects* and choose *Run* under *Build & Run*. Then add a new `Run configuration` with the executable `/usr/bin/python3` and the Command line arguments set to the path of the unit test python file, e.g. `/home/user/dev/qgis/QGIS/tests/src/python/test_qgsattributeformeditorwidget.py`.

Now also change the `Run Environment` and add 3 new variables:

Variable	Value
PYTHONPATH	[build]/output/python/:[build]/output/python/plugins:[source]/tests/src/python
QGIS_PREFIX_PATH	[build]/output
LD_LIBRARY_PATH	[build]/output/lib

Replace `[build]` with your build directory and `[source]` with your source directory.

5.6.2 Have fun

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 independent way.

CHAPTER 6

Processing Algorithms Testing

6.1 Algorithm tests

Muista: The original version of these instructions is available at https://github.com/qgis/QGIS/blob/release-3_4/python/plugins/processing/tests/README.md

QGIS provides several algorithms under the Processing framework. You can extend this list with algorithms of your own and, like any new feature, adding tests is required.

To test algorithms you can add entries into `testdata/qgis_algorithm_tests.yaml` or `testdata/gdal_algorithm_tests.yaml` as appropriate.

This file is structured with yaml syntax.

A basic test appears under the toplevel key `tests` and looks like this:

```
- name: centroid
  algorithm: qgis:polygoncentroids
  params:
    - type: vector
      name: polys.gml
  results:
    OUTPUT_LAYER:
      type: vector
      name: expected/polys_centroid.gml
```

6.1.1 How To

To add a new test please follow these steps:

1. Run the algorithm you want to test in QGIS from the processing toolbox. If the result is a vector layer prefer GML, with its XSD, as output for its support of mixed geometry types and good readability. Redirect output to `python/plugins/processing/tests/testdata/expected`. For input layers prefer to use what's already there in the folder `testdata`. If you need extra data, put it into `testdata/custom`.
2. When you have run the algorithm, go to *Processing* → *History* and find the algorithm which you have just run.
3. Right click the algorithm and click *Create Test*. A new window will open with a text definition.
4. Open the file `python/plugins/processing/tests/testdata/algorithm_tests.yaml`, copy the text definition there.

The first string from the command goes to the key `algorithm`, the subsequent ones to `params` and the last one(s) to `results`.

The above translates to

```
- name: densify
  algorithm: qgis:densifygeometriesgivenaninterval
  params:
    - type: vector
      name: polys.gml
    - 2 # Interval
  results:
    OUTPUT:
      type: vector
      name: expected/polys_densify.gml
```

It is also possible to create tests for Processing scripts. Scripts should be placed in the `scripts` subdirectory in the test data directory `python/plugins/processing/tests/testdata/`. The script file name should match the script algorithm name.

6.1.2 Parameters and results

Trivial type parameters

Parameters and results are specified as lists or dictionaries:

```
params:
  INTERVAL: 5
  INTERPOLATE: True
  NAME: A processing test
```

or

```
params:
  - 2
  - string
  - another param
```

Layer type parameters

You will often need to specify layers as parameters. To specify a layer you will need to specify:

- the type, ie `vector` or `raster`
- a name, with a relative path like `expected/polys_centroid.gml`

This is what it looks like in action:

```
params:
  PAR: 2
  STR: string
  LAYER:
    type: vector
    name: polys.gml
  OTHER: another param
```

File type parameters

If you need an external file for the algorithm test, you need to specify the 'file' type and the (relative) path to the file in its 'name':

```
params:
  PAR: 2
  STR: string
  EXTFILE:
    type: file
    name: custom/grass7/extfile.txt
  OTHER: another param
```

Results

Results are specified very similarly.

Basic vector files

It couldn't be more trivial

```
OUTPUT:
 name: expected/qgis_intersection.gml
 type: vector
```

Add the expected GML and XSD files in the folder.

Vector with tolerance

Sometimes different platforms create slightly different results which are still acceptable. In this case (but only then) you may also use additional properties to define how a layer is compared.

To deal with a certain tolerance for output values you can specify a `compare` property for an output. The compare property can contain sub-properties for `fields`. This contains information about how precisely a certain field is compared (`precision`) or a field can even entirely be `skip``ed. There is a special field name ``__all__` which will apply a certain tolerance to all fields. There is another property `geometry` which also accepts a `precision` which is applied to each vertex.

```
OUTPUT:
 type: vector
 name: expected/abcd.gml
 compare:
   fields:
     __all__:
       precision: 5 # compare to a precision of .00001 on all fields
     A: skip # skip field A
   geometry:
     precision: 5 # compare coordinates with a precision of 5 digits
```

Raster files

Raster files are compared with a hash checksum. This is calculated when you create a test from the processing history.

```
OUTPUT:
 type: rasterhash
 hash: f1fedeb6782f9389cf43590d4c85ada9155ab61fef6dc285aaeb54d6
```

Files

You can compare the content of an output file to an expected result reference file

```
OUTPUT_HTML_FILE:
 name: expected/basic_statistics_string.html
 type: file
```

Or you can use one or more regular expressions that will be matched against the file content

```
OUTPUT:
 name: layer_info.html
 type: regex
 rules:
   - 'Extent: \(-1.000000, -3.000000\) - \(11.000000, 5.000000\)'
   - 'Geometry: Line String'
   - 'Feature Count: 6'
```

Directories

You can compare the content of an output directory with an expected result reference directory

```
OUTPUT_DIR:
 name: expected/tiles_xyz/test_1
 type: directory
```

6.1.3 Algorithm Context

There are a few more definitions that can modify the context of the algorithm - these can be specified at the top level of test:

- `project` - will load a specified QGIS project file before running the algorithm. If not specified, the algorithm will run with an empty project
- `project_crs` - overrides the default project CRS - e.g. `EPSG:27700`
- `ellipsoid` - overrides the default project ellipsoid used for measurements, e.g. `GRS80`

6.1.4 Running tests locally

```
ctest -V -R ProcessingQgisAlgorithmsTest
```

or one of the following values listed in the CMakelists.txt

CHAPTER 7

OGC Conformance Testing

The Open Geospatial Consortium (OGC) provides tests which can be run free of charge to make sure a server is compliant with a certain specification. This chapter provides a quick tutorial to setup the WMS tests on an Ubuntu system. A detailed documentation can be found at the OGC website.

7.1 Setup of WMS 1.3 and WMS 1.1.1 conformance tests

```
sudo apt install openjdk-8-jdk maven
cd ~/src
git clone https://github.com/opengeospatial/teamengine.git
cd teamengine
mvn install
mkdir ~/TE_BASE
export TE_BASE=~/TE_BASE
unzip -o ./teamengine-console/target/teamengine-console-4.11-SNAPSHOT-base.zip -d
↪$TE_BASE
mkdir ~/te-install
unzip -o ./teamengine-console/target/teamengine-console-4.11-SNAPSHOT-bin.zip -d ~/
↪te-install
```

Download and install WMS 1.3.0 test

```
cd ~/src
git clone https://github.com/opengeospatial/ets-wms13.git
cd ets-wms13
mvn install
```

Download and install WMS 1.1.1 test

```
cd ~/src
git clone https://github.com/opengeospatial/ets-wms11.git
cd ets-wms11
mvn install
```

7.2 Test project

For the WMS tests, data can be downloaded and loaded into a QGIS project:

```
wget https://cite.opengeospatial.org/teamengine/about/wms/1.3.0/site/data-wms-1.3.
↪0.zip
unzip data-wms-1.3.0.zip
```

Then create a QGIS project according to the description in https://cite.opengeospatial.org/teamengine/about/wms/1.3.0/site/. To run the tests, we need to provide the GetCapabilities URL of the service later.

7.3 Running the WMS 1.3.0 test

```
export PATH=/usr/lib/jvm/java-8-openjdk-amd64/bin:$PATH
export TE_BASE=$HOME/TE_BASE
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
cd ~/te-install
./bin/unix/test.sh -source=$HOME/src/ets-wms13/src/main/scripts/ctl/main.xml
```

7.4 Running the WMS 1.1.1 test

```
export PATH=/usr/lib/jvm/java-8-openjdk-amd64/bin:$PATH
export TE_BASE=$HOME/TE_BASE
export ETS_SRC=$HOME/ets-resources
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
cd ~/te-install
./bin/unix/test.sh -source=$HOME/src/ets-wms11/src/main/scripts/ctl/wms.xml
```