"

QGIS Developers Guide

Publicación 2.18

QGIS Project

08 de April de 2019

Contents

1 Estándares de codificación de QGIS **3**
- 1.1 Clases . . . 3
- 1.2 Diseñador Qt . . . 5
- 1.3 Archivos C++ . . . 5
- 1.4 Nombres de variables . . . 6
- 1.5 Tipos Enumerados . . . 6
- 1.6 Constantes & Macros Globales . . . 6
- 1.7 Qt Signals y Slots . . . 7
- 1.8 Editando . . . 7
- 1.9 Compatibilidad API . . . 7
- 1.10 Estilo de Codificación . . . 8
- 1.11 Créditos para contribuciones . . . 10

2 Acceso GIT **11**
- 2.1 Instalación . . . 11
- 2.2 Acceda al repositorio . . . 12
- 2.3 Revisar branch . . . 12
- 2.4 Fuentes de documentación QGIS . . . 12
- 2.5 Fuentes del sitio web de QGIS . . . 13
- 2.6 Documentación GIT . . . 13
- 2.7 Desarrollo en branches . . . 13
- 2.8 Envío de parches y solicitudes de extracción . . . 14
- 2.9 Nombramiento de archivos de parche . . . 15
- 2.10 Cree su parche en el directorio fuente QGIS de nivel superior . . . 15
- 2.11 Obtención de acceso de escritura GIT . . . 16

3 Prueba de Unidad **17**
- 3.1 La infraestructura de prueba de QGIS - una visión general . . . 17
- 3.2 Adding your unit test to CMakeLists.txt . . . 21
- 3.3 The ADD_QGIS_TEST macro explained . . . 22
- 3.4 Building your unit test . . . 23
- 3.5 Ejecutar las pruebas . . . 23

4 Iniciando y ejecutando con QtCreator y QGIS **25**
- 4.1 Instalando QtCreator . . . 25
- 4.2 Estableciendo su proyecto . . . 25
- 4.3 Setting up your build environment . . . 27
- 4.4 Estableciendo su ambiente de ejecución . . . 29
- 4.5 Ejecución y depuración . . . 31

5 GIH (Guías de Interfaz Humano) **33**
- 5.1 Autores . . . 34

6 Pruebas de conformidad OGC **35**
6.1 Configuración de las pruebas de conformidad WMS 1.3 y WMS 1.1.1 35
6.2 Proyecto de prueba 35
6.3 Ejecución de la prueba WMS 1.3.0 36
6.4 Ejecución de la prueba WMS 1.1.1 36

Índice **37**

Bienvenido/a a las páginas de desarrollo QGIS. Aquí encontrará normas, herramientas y pasos para contribuir al código QGIS de manera fácil y eficientemente.

CHAPTER 1

Estándares de codificación de QGIS

- Clases
  - Nombres
  - Miembros
  - Funciones del accesor
  - Funciones
  - Argumentos de la función
  - Función que regresa valores
- Diseñador Qt
  - Clases Generadas
  - Diálogos
- Archivos C++
  - Nombres
  - Encabezado y Licencia Estándar
- Nombres de variables
- Tipos Enumerados
- Constantes & Macros Globales
- Qt Signals y Slots
- Editando
  - Tabulaciones
  - Indentación
  - Llaves
- Compatibilidad API
- Estilo de Codificación
  - Siempre que sea Posible Generalizar Código
  - Preffiere tener primero las constantes en los predicados
  - espacio en blanco puede ser tu amigo
  - Ponga los comandos en líneas separadas
  - Idente modificadores de acceso
  - Recomendaciones de libros
- Créditos para contribuciones

Todos los desarrolladores de QGIS deberían seguir estos estándares.

1.1 Clases

1.1.1 Nombres

Las clases en QGIS comienzan con Qgs y están formadas usando la notación Camel Case

Ejemplos:

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

1.1.2 Miembros

Los nombres de los miembros de las clases comienzan con una m minúscula y se forman con mayúsculas y minúsculas.

- `mMapCanvas`
- `mCurrentExtent`

Todos los miembros de la clase deben ser privados. Se recomienda encarecidamente no usar miembros públicos en las clases. Los miembros protegidos deben evitarse cuando pueda ser necesario acceder al miembro desde las subclases de Python, ya que los miembros protegidos no se pueden usar desde los enlaces de Python.

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 Funciones del accesor

Los valores de los miembros de la Clase deben obtenerse a través de las funciones del accesor. La función debe ser nombrada sin un prefijo get. Las funciones de acceso para los dos miembros privados anteriores serían:

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

Asegúrese de que los accesores estén correctamente marcados con `const`. Cuando sea apropiado, esto puede requerir que el valor en caché de las variables del tipo miembro estén marcadas con `mutable`.

1.1.4 Funciones

Los nombres de las funciones comienzan con una letra minúscula y se forman mezclando mayúsculas y minúsculas. El nombre de la función debe indicar algo acerca de su propósito

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

Para guardar la consistencia con la API disponible de QGIS y con la API de Qt se deben evitar las abreviaciones, por ejemplo, `setDestinationSize` en lugar de ``setDestSize` o `setMaximumValue` en lugar de `setMaxVal`.

Los acrónimos también deben ser usados con CamelCase por consistencia. Por ejemplo, `setXml` en lugar de `setXML`.

1.1.5 Argumentos de la función

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

Preste atención especial a cuándo se deben pasar los argumentos por referencia. A menos que los objetos de argumento sean pequeños y se copien trivialmente (como los objetos QPoint), se deben pasar por referencia const. Para mantener coherencia con la API Qt, incluso los objetos compartidos implícitamente pasan por referencia const (por ejemplo, ‘‘ setTitle (const QString & title) ‘‘ en lugar de ‘‘ setTitle (QString title) ‘‘.

1.1.6 Función que regresa valores

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.

- `int maximumValue() const`
- `const LayerSet& layers() const`
- `QString title() const` (QString is implicitly shared)
- `QList< QgsMapLayer* > layers() const` (QList is implicitly shared)

1.2 Diseñador Qt

1.2.1 Clases Generadas

Las clases QGIS que se se generan desde archivos Qt Designer (ui) deberían tener un sufijo de Base. Esto identifica la clase como una clase base generada.

Ejemplos:

- `QgsPluginManagerBase`
- `QgsUserOptionsBase`

1.2.2 Diálogos

Todos los cuadros de diálogo deben implementar ayuda con información sobre herramientas para todos los iconos de la barra de herramientas y otros widgets relevantes. La información sobre herramientas agrega mucho a la detección de características para usuarios nuevos y experimentados.

Asegúrese de que el orden de tabulación para widgets se actualice cada vez que cambie el diseño de un diálogo.

1.3 Archivos C++

1.3.1 Nombres

La implementación de C++ y los archivos del título deben tener una extensión .cpp y .h respectivamente. Los nombres de los archivos deberán estar todos en minúsculas y, en el respecto a las clases, deberán coincidir con el nombre de la clase.

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

Nota: En caso de que no esté claro en la declaración anterior, para que un nombre de archivo coincida con un nombre de clase, implícitamente significa que cada clase debe declararse e implementarse en su propio archivo. Esto hace que sea mucho más fácil para los usuarios nuevos identificar dónde se relaciona el código con una clase específica.

1.3.2 Encabezado y Licencia Estándar

Cada archivo de origen debería contener una sección de encabezado que siga el siguiente patrón de ejemplo:

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

Nota: Hay una plantilla para Qt Creator en git. `doc/qt_creator_license_template` uselo, copielo de la ubicación local, ajuste la dirección de correo electrónico y -si es necesario- el nombre y configure QtCreator para usarlo: *Herramientas→ Opciones → C++ → Nombre de Archivo*.

1.4 Nombres de variables

Los nombres de las variables locales inician con minúscula y se forman utilizando mayúsculas y minúsculas. No utilice prefijos como `mi` o `el`.

Ejemplos:

- `mapCanvas`
- `currentExtent`

1.5 Tipos Enumerados

Los tipos enumerados deben nombrarse en CamelCase con una mayúscula al inicio, p. ej:

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

No utilice nombres de tipos genéricos que entren en conflicto con otros tipos. p.ej. use `UnidadDesconocidaUnit` en lugar de `Desconocido`

1.6 Constantes & Macros Globales

Constantes y macros globales deberían escribirse en mayúscula separada por guión bajo e.g.:

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

1.7 Qt Signals y Slots

Todos los espacios/señales conectados deben hacerse utilizando el conector "nuevo estilo" disponible en Qt5. Más información sobre este requisito está disponible en QEP #77.

Evite utilizar de Qt Ranuras de conexión automática (p. ej. los nombrados `void on_mSpinBox_valueChanged`). Ranuras de conexión automática son frágiles y propensos a romperse sin previo aviso si los diálogos se refactan.

1.8 Editando

Cualquier editor de texto/IDE puede ser usado para editar código QGIS, siempre que los siguientes requerimientos sean atendidos.

1.8.1 Tabulaciones

Defina su editor para emular tabulaciones con espacios. El espaciado de tabulación debería establecerse en 2 espacios.

Nota: En vim esto se hace con `set expandtab ts=2`

1.8.2 Indentación

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).

1.8.3 Llaves

Llaves deberían iniciar la línea que sigue a la expresión:

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

1.9 Compatibilidad API

There is API documentation for C++.

Tratamos de mantener la API estable y compatible con versiones anteriores. Los ajustes en la API deben hacerse de modo similar al codigo fuente de Qt, por ejemplo

```
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 be 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();
}
```

1.10 Estilo de Codificación

Aquí se describen algunas pistas y consejos de programación que podrán reducir errores, el tiempo de desarrollo y el mantenimiento.

1.10.1 Siempre que sea Posible Generalizar Código

Si usted está cortando y pegando código, o escribiendo la misma cosa más de una vez, considere consolidar el código en una sola función.

Esto hará:

- permite hacer cambios en una ubicacion en lugar de en multiples ubicaciones
- ayuda a prevenir codigo innnecesariamente largo o lento
- dificulta para multiples copias la evolucion de diferencias en el tiempo, lo cual dificulta a otros entender y mantener

1.10.2 Preffiere tener primero las constantes en los predicados

Es preferible poner constantes primero en predicados.

`0 == valor` en vez de `valor == 0`

Esto ayudara a los programadores a prevenir el uso accidental de "=" cuando intentaban usar "==", lo cual podria introducir sutiles bugs logicos. El compilador generara un error si accidentalmente usas "=" en lugar de "==" para comparaciones dado que las constantes inherentemente no pueden ser asignadas

1.10.3 espacio en blanco puede ser tu amigo

Agregar espacios entre operadores, sentencias y funciones facilita a los humanos analizar el codigo por partes.

lo cual es facil de leer, esto:

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

o este:

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

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

1.10.4 Ponga los comandos en líneas separadas

Cuando se lee codigo, es facil omitir comandos si estos no estan al comienzo de la linea. Cuando se lee rapidamente a lo largo del código, es comun saltarse lineas si estas no lucen como lo que se esta buscando en los primeros caracteres. Es tambien commun esperar un comando despues de una sentencia condicional como "if"

Considere:

```
if (foo) bar();

baz(); bar();
```

Es muy facil perder parte de lo que es el flujo de control. En lugar use

```
if (foo)
  bar();

baz();
bar();
```

1.10.5 Idente modificadores de acceso

Los modificadores de acceso estructuran una clase en secciones de API pública, API protegida y API privada. Los modificadores de acceso a ellos mismos agrupan el código en esta estructura. Sangrar el modificador de acceso y las declaraciones.

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

1.10.6 Recomendaciones de libros

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

Debería también leer este artículo de Qt Quarterly sobre designing Qt style (APIs)

1.11 Créditos para contribuciones

Se anima a los que contribuyen nuevas funciones a hacer conocer a la gente acerca de sus contribuciones mediante:

- añadir una nota al registro de cambios para la primer versión donde el código ha sido incorporado, del tipo:

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

- writing an article about the new feature on a blog, and add it to the QGIS planet http://plugins.qgis.org/planet/
- agregando su nombre a:
  - https://github.com/qgis/QGIS/blob/master/doc/CONTRIBUTORS
  - https://github.com/qgis/QGIS/blob/master/doc/AUTHORS
  - https://github.com/qgis/QGIS/blob/master/doc/contributors.json

CHAPTER 2

Acceso GIT

- Instalación
  - Instalar git para GNU/Linux
  - Instale git para Windows
  - Instale git para OSX
- Acceda al repositorio
- Revisar branch
- Fuentes de documentación QGIS
- Fuentes del sitio web de QGIS
- Documentación GIT
- Desarrollo en branches
  - Propósito
  - Procedimiento
  - Documentation on wiki
  - Pruebe antes de volver a fusionar al master
- Envío de parches y solicitudes de extracción
  - Pull Requests
    - La mejor practica para crear una solicitud de extracción
    - Etiquetas especiales para notificar a los documentadores
    - Para fusionar una solicitud de extracción
- Nombramiento de archivos de parche
- Cree su parche en el directorio fuente QGIS de nivel superior
  - Haga que se note su parche
  - Debida diligencia
- Obtención de acceso de escritura GIT

Esta seccion describe como empeza a usar el repositorio GIT de QGIS. Antes de que puedas hacer esto, primero necesitas tener un cliente git instalado en tu sistema.

2.1 Instalación

2.1.1 Instalar git para GNU/Linux

Los usuarios de distro basadas en Debian pueden hacer:

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

2.1.2 Instale git para Windows

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

2.1.3 Instale git para 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).

Una vez descargado, abra la imagen del disco y ejecute el instalador

nota PPC/fuente

El sitio de git no ofrece compilaciones PPC. Si se necesita una, o solo quiere más control sobre la instalación, es necesario que lo compile usted mismo.

Download the source from http://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
```

Si no se necesitan ninguno de los extras, Perl, Python o TclTk (GUI), puede deshabilitarlos antes de ejecutar make:

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

2.2 Acceda al repositorio

Para clonar QGIS maestro:

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

2.3 Revisar branch

Para validar un branch, por ejemplo, el branch de lanzamiento 2.6.1 hace:

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

Para validar el branch maestro:

```
cd QGIS
git checkout master
```

Nota: En QGIS, mantenemos nuestro código más estable en el branch de lanzamiento actual. El Master contiene código para la serie de versiones llamadas 'inestables'. Periódicamente publicaremos una versión de master y luego continuaremos con la estabilización y la incorporación selectiva de nuevas funciones en master.

Vea el archivo INSTALL en el árbol de fuentes para obtener instrucciones especificas sobre la creación de versiones de desarrollo

---

2.4 Fuentes de documentación QGIS

Si estas interesado en verificar la documentacion fuente de QGIS:

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

También puede dar un vistazo al documento readme incluido en la documentación del repositorio para más información

2.5 Fuentes del sitio web de QGIS

Si esta interesado en verificar las fuentes en el sitio web de QGIS:

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

También se puede dar un vistazo al documento readme incluido con la el repositorio del sitio web para mayor información.

2.6 Documentación GIT

Vea los siguientes sitios para información de como volverse un maestro GIT.

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

2.7 Desarrollo en branches

2.7.1 Propósito

La complejidad del código fuente de QGIS se ha incrementado considerablemente durante los últimos años. Por lo tanto es difícil anticipar los efectos secundarios que traerá añadir una nueva. En el pasado, el proyecto QGIS tenia ciclos de lanzamiento muy largos porque era mucho trabajo reequilibrar la estabilidad del sistema de software después de que nuevos elementos se añadieron. Para superar estos problemas, QGIS cambio un modelo de desarrollo donde los nuevos elementos son codificados en branches de GIT primero y se fusionan al master (el branch principal) cuando están terminadas y estables. Esta sección describe el procedimiento para ramificar y fusionar el proyecto QGIS.

2.7.2 Procedimiento

- **Anuncio inicial en lista de correo:** Antes de comenzar, haga un anuncio en la lista de correo de desarrollo para ver si otro desarrollador esta trabajando en el mismo característica. También contacte al asesor técnico del comité directivo del proyecto (PSC). Si la nueva característica requiere algún cambio de la arquitectura de QGIS, se necesita una petición de comentarios (RFC)

Crear un branch: Cree un nuevo GIT branch para el desarrollo del nuevo elemento.

```
git checkout -b newfeature
```

Ahora puede comenzar con el desarrollo. Si planea hacer extensas tareas en ese branch, y desea compartir el trabajo con otros desarrolladores, y tener acceso de escritura al repositorio en sentido ascendente, puede enviar su repositorio al repositorio oficial de QGIS haciendo:

```
git push origin newfeature
```

Nota: Si en le branch ya existen sus cambios serán introducidos a él.

Rebase para masterizar regularmente: es recomendable para volver a establecer la base para incorporar los cambios en el master del branch en una rama de forma regular. Esto facilita la fusión de la rama nueva a la maestra más tarde. Después de reestablecer la base es necesario ‘‘git push -f‘‘ para el repositorio bifucado.

Nota: Nunca haga `git push -f` al repositorio original! solo utilice esto para su trabajo de branch.

```
git rebase master
```

2.7.3 Documentation on wiki

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

2.7.4 Pruebe antes de volver a fusionar al master

Cuando este terminada la nueva característica y este contento con la estabilidad, haga un anuncio en la lista de desarrollo. Antes de fusionar de nuevo, los cambios serán probados por los desarrolladores y usuarios.

2.8 Envío de parches y solicitudes de extracción

Hay algunas pautas que lo ayudaran a obtener sus parches y solicitudes de extracción en QGIS fácilmente, y nos ayudaran a mejorar los parches que se envían para utilizar fácilmente.

2.8.1 Pull Requests

En general es más fácil para los desarrolladores si envían solicitudes de extracción. Nosotros no describimos solicitudes de extracción aquí, sino que remitimos a Documentación de solicitud de extracción en GitHub.

Si realiza una solicitud de extracción, le pedimos que combine el master con su branch de PR con regularidad, de modo que su PR siempre se pueda combinar con la rama principal ascendente.

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

Por favor, consulte la sección a continuación sobre 'obtener su parche notado'. En general, cuando envía un PR, debe asumir la responsabilidad de seguirlo hasta el final - responder a las consultas publicadas por otros desarrolladores, buscar un 'campeón' para su función y darles un ligero recordatorio ocasionalmente si ve que su PR no siendo actuado Tenga en cuenta que el proyecto QGIS está impulsado por el esfuerzo voluntario y es posible que las personas no puedan asistir a su PR instantáneamente. Si siente que el PR no está recibiendo la atención que merece sus opciones para acelerar, debe ser (en orden de prioridad):

- Envié un mensaje a la lista de correo 'marketing' su PR y que maravilloso será incluirlo en la base del código.
- Envié un mensaje a la persona a la que se le asigno su PR en la cola de PR.
- Envíe un mensaje a Marco Hugentobler (quién gestiona la cola de PR).
- Envié un mensaje al comite directivo del proyecto pidiendo que le ayuden a incorporar su PR en el código base.

La mejor practica para crear una solicitud de extracción

- Siempre inicie un branch por característica del actual master.
- Si estas codificando un branch de característica, no ''fusionar" nada a ese branch, en lugar de rebase como se describe en el siguiente punto para mantener su historial limpio.
- Antes de crear una solicitud de extracción realice `git fetch origin`` y ``git rebase origin/master` (el origen dado es el control remoto para la conexión ascendente y no su propio control remoto, verifique su `.git/config` o haga '' git remote -v | grep github.com / qgis'').
- Puedes hacer una rebase git como en la última línea repetidamente sin hacer ningún daño (siempre y cuando el único propósito de tu branch sea fusionarse con el master).

- Atención: después de un rebase se necesita `git push -f` a su repositorio bifucado. **DESARROLLO BÁSICO: NO HACER ESTO EN EL REPOSITORIO PÚBLICO DE QGIS!**

Etiquetas especiales para notificar a los documentadores

Además de las etiquetas comunes puede añadir para clasificar su PR, existen otras especiales que puede usar para generar automáticamente informes de problemas en el repositorio de documentación-QGIS tan pronto como su solicitud de extracción se fusione.

- `[needs-docs]` para instruir a los escritores de documentos que agreguen documentos extra después de una corrección o añadir a una funcionalidad ya existente.
- `[feature]` en caso de una nueva funcionalidad. Llenar una buena descripción en su PR será un buen comienzo.

Por favor desarrolladores utilicen estas etiquetas (no distingue entre mayúsculas y minúsculas) para que los redactores de documentos tengan problemas para trabajar y tengan una visión general de las cosas por hacer. PERO por favor, tome el tiempo para añadir algo de texto: en la confirmación O en los documentos.

Para fusionar una solicitud de extracción

Opción A:

- clic sobre le botón de fusión (Crea una fusión no rápida)

Opcion B:

- Verificar la solicitud de extracción
- Prueba (también se requiere para la opción A, obviamente)
- verificar master, fusión git pr/1234
- Opcional: `git pull --rebase`: Crea un avance rápido, no se hace “merge commit”. Limpiar historial, pero será difícil revertir la fusión.
- `git push` (NUNCA JAMÁS utilizar la opción -f, aquí)

2.9 Nombramiento de archivos de parche

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.

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

2.10 Cree su parche en el directorio fuente QGIS de nivel superior

Esta es la forma más fácil de aplicar los parches ya que no necesitamos navegar a un parche específico. Además, cuando recibe parches, generalmente los evalúo usando merge, y tener el parche del directorio de nivel superior hace que esto sea mucho más fácil. A continuación se muestra un ejemplo de cómo puede incluir varios archivos modificados en su parche desde el directorio de nivel superior:

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

Esto asegurará que su rama maestra esté sincronizada con el repositorio en sentido ascendente, y luego generará un parche que contiene el delta entre su rama de características y lo que está en la rama maestra.

2.10.1 Haga que se note su parche

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 Technical Resources 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).

2.10.2 Debida diligencia

QGIS esta bajo licencia GPL. Debe hacer todos los esfuerzos posibles para garantizar que solo envíe parches que no estén sujetos a derechos de propiedad intelectual contradictorios. Además, no envíe un código que no esté satisfecho de haber puesto a disposición en virtud de la GPL.

2.11 Obtención de acceso de escritura GIT

El acceso de escritura al árbol fuente de QGIS es por invitación. Normalmente, cuando una persona envía varios parches (no hay un número fijo aquí) que demuestren la competencia básica y la comprensión de las convenciones de codificación C++ y QGIS, uno de los miembros del PSC u otros desarrolladores existentes pueden nominar a esa persona al PSC para otorgar acceso de escritura . El nominador debe dar un párrafo promocional básico de por qué creen que esa persona debe obtener acceso de escritura. En algunos casos otorgaremos acceso de escritura a desarrolladores que no sean C++, p. ej., para traductores y documentadores. En estos casos, la persona aún debe haber demostrado la capacidad de enviar parches y lo ideal es que haya enviado varios parches sustanciales que demuestren su comprensión de la modificación de la base de códigos sin romper las cosas, etc.

Nota: Desde que nos mudamos a GIT, es menos probable que otorguemos acceso de escritura a los nuevos desarrolladores, ya que es trivial compartir código dentro de github mediante la falsificación de QGIS y la emisión de solicitudes de extracción.

Siempre verifique que todo se compila antes de realizar cualquier solicitud de confirmación / extracción. Intente tener en cuenta las posibles roturas que sus confirmaciones pueden causar a las personas que compilan en otras plataformas y con las versiones más antiguas / más nuevas de las librerías.

Al realizar una confirmación, aparecerá su editor (como se define en la variable de entorno $ EDITOR) y deberá hacer un comentario en la parte superior del archivo (encima del área que dice 'no cambiar esto'). Ponga un comentario descriptivo y en lugar de hacer varias pequeñas confirmaciones si los cambios en una serie de archivos no están relacionados. Por el contrario, preferimos que agrupe los cambios relacionados en una única confirmación.

CHAPTER 3

Prueba de Unidad

- La infraestructura de prueba de QGIS - una visión general
  - Creating a unit test
- Adding your unit test to CMakeLists.txt
- The ADD_QGIS_TEST macro explained
- Building your unit test
- Ejecutar las pruebas

A noviembre de 2007, requerimos todas las nuevas características que van a sobreponerse a ser acompañados con una unidad de prueba. Al principio, nos hemos limitado a este requisito qgis_core, y vamos a ampliar este a otras partes de la base de código una vez que la gente está familiarizada con los procedimientos para las pruebas unitarias que se explican en las secciones que siguen.

3.1 La infraestructura de prueba de QGIS - una visión general

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 / cmakesetup. 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!

3.1.1 Creating a unit test

Crear una unidad de prueba es fácil - por lo general, esto se hace sólo por la creación de un único archivo .cpp (no se utiliza el archivo .h) y poner en práctica todos los métodos de prueba y públicos que devuelven void. Voy a usar una clase de prueba simple para QgsRasterLayer en toda la sección que sigue para ilustrar. Por convenio nombraremos nuestra prueba con el mismo nombre que la clase que están poniendo a prueba, pero con el prefijo 'Test'. Así que la implementación de prueba va en un archivo llamado qgsrasterlayer.cpp de prueba y la propia clase será prueba QgsRasterLayer. Primero añadimos nuestro banner estándar de copyright:

```
/***************************************************************************
 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 use 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. I am implementing two methods here which illustrates two 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).

```
//
// 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 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 redmine 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 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 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() << st

  //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 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 (descripción, modo)*
- *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 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 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.

3.2 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)
```

3.3 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.

```
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)
```

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:

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

Desde nuestra clase de prueba se debe ejecutar a través del compilador objeto meta Qt (moc) que necesitamos para proporcionar un par de líneas para que esto suceda también:

```
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 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 executable:

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

A continuación tenemos que especificar las dependencias de la bibliotecas. Por el momento, las clases han sido implementadas con un catch-all dependencia QT_LIBRARIES, pero se trabaja para sustituir eso con las librerías

Qt específicas que cada clase necesita solamente. Por supuesto también es necesario para enlazar a las bibliotecas qgis pertinentes según lo requiera la prueba de la unidad.

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

A continuación le digo a cmake para instalar las pruebas para el mismo lugar que los qgis binarios. Esto es algo que va a extraer en el futuro para que las pruebas se puedan ejecutar directamente desde el interior del árbol de origen.

```
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 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 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.

3.4 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.

3.5 Ejecutar las pruebas

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//mIconProjectionDisable
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.

CHAPTER 4

Iniciando y ejecutando con QtCreator y QGIS

- Instalando QtCreator
- Estableciendo su proyecto
- Setting up your build environment
- Estableciendo su ambiente de ejecución
- Ejecución y depuración

QtCreator es un IDE bastante nuevo de marcadores de la librería Qt. Con QtCreator puede construir cualquier proyecto C++, pero esta realmente optimizado para la gente que trabaja con Qt(4) base en aplicaciones (incluyendo aplicaciones móviles). Todo lo que describo a continuación asume que estás corriendo en Ubuntu 11.04 'Natty'.

4.1 Instalando QtCreator

Esta parte es fácil:

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

Después de instalar debería encontrarlo en su menú gnome.

4.2 Estableciendo su proyecto

Supongo que ya tiene un clon de QGIS local que contiene el código fuente, y han instalado todas las dependencias de compilación necesarias, etc. Hay instrucciones detalladas para *git access* and Instalación de dependencias.

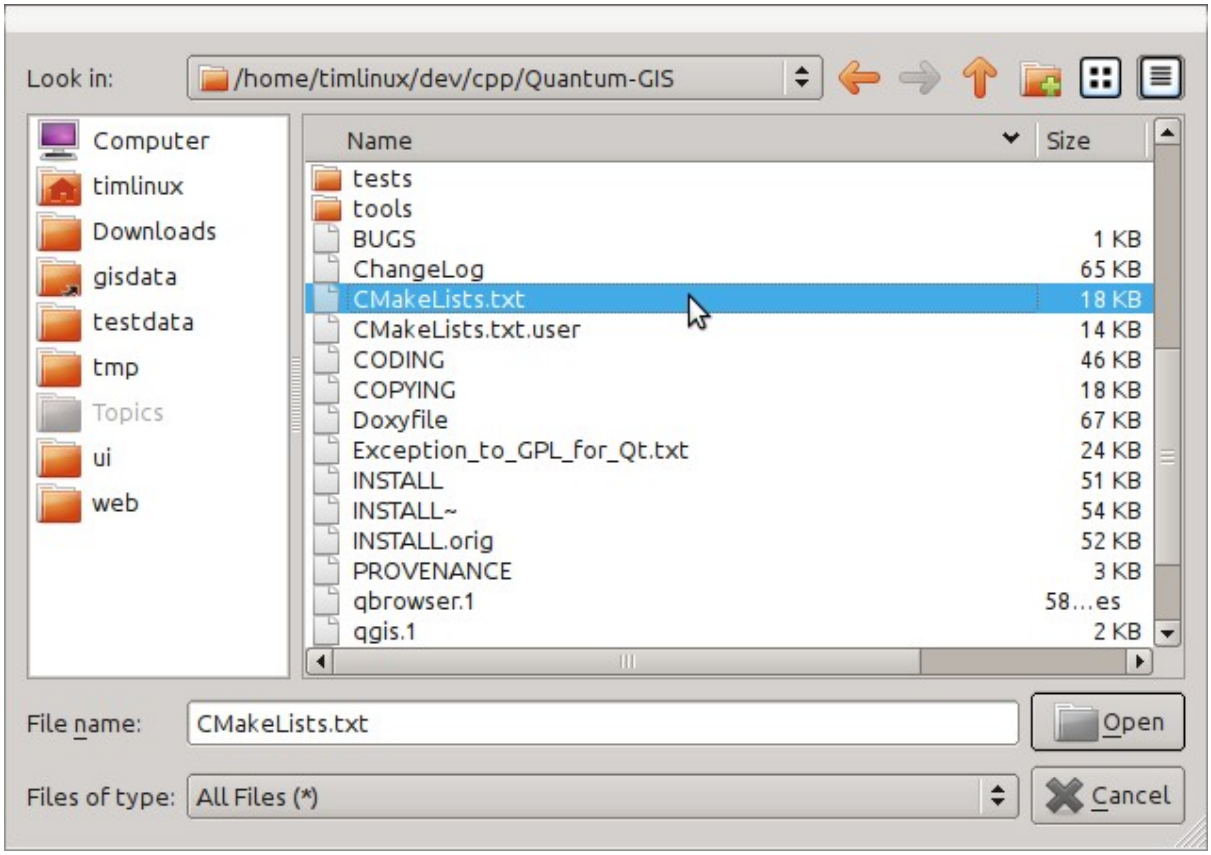
En mi sistema he comprobado el código en `$HOME/dev/cpp/QGIS` y el resto de los artículo esta escrito suponiendo eso, debe actualizar las rutas según corresponda para su sistema local.

Al lanzar QtCreator hace:

Archivo -> Abrir archivo o proyecto

Después utilice el diálogo de selección de archivos resultante para buscar y abrir este archivo:

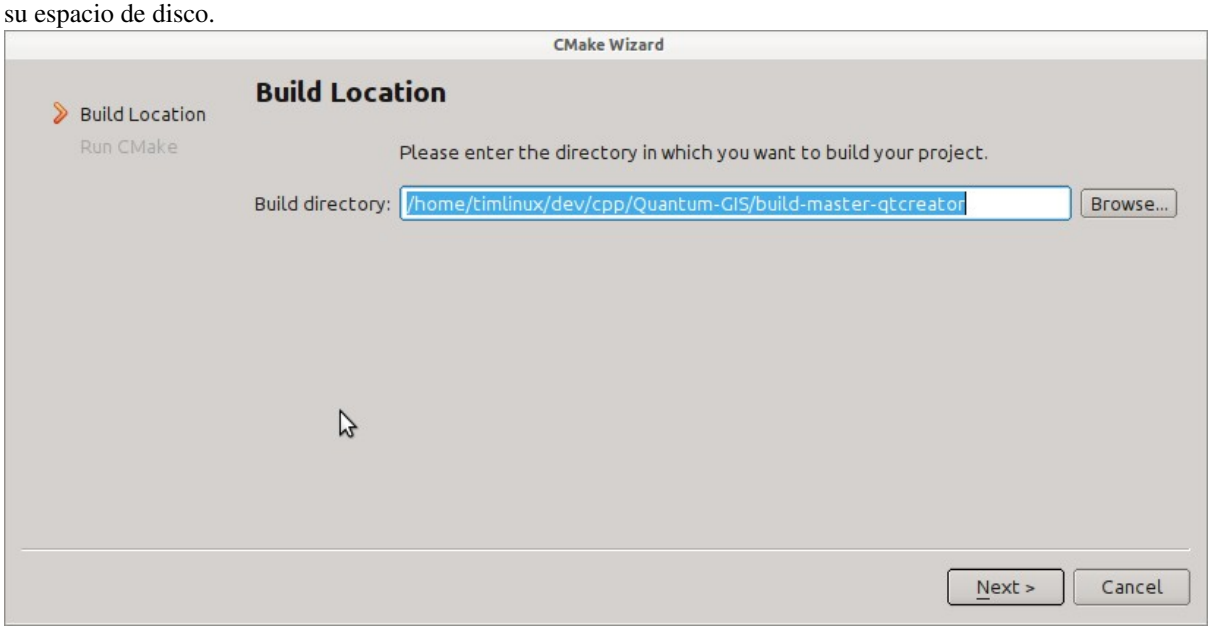
```
$HOME/dev/cpp/QGIS/CMakeLists.txt
```



A continuación, se le pedirá una ubicación de compilación. Cree un directorio de compilación específico para que QtCreator funcione en:

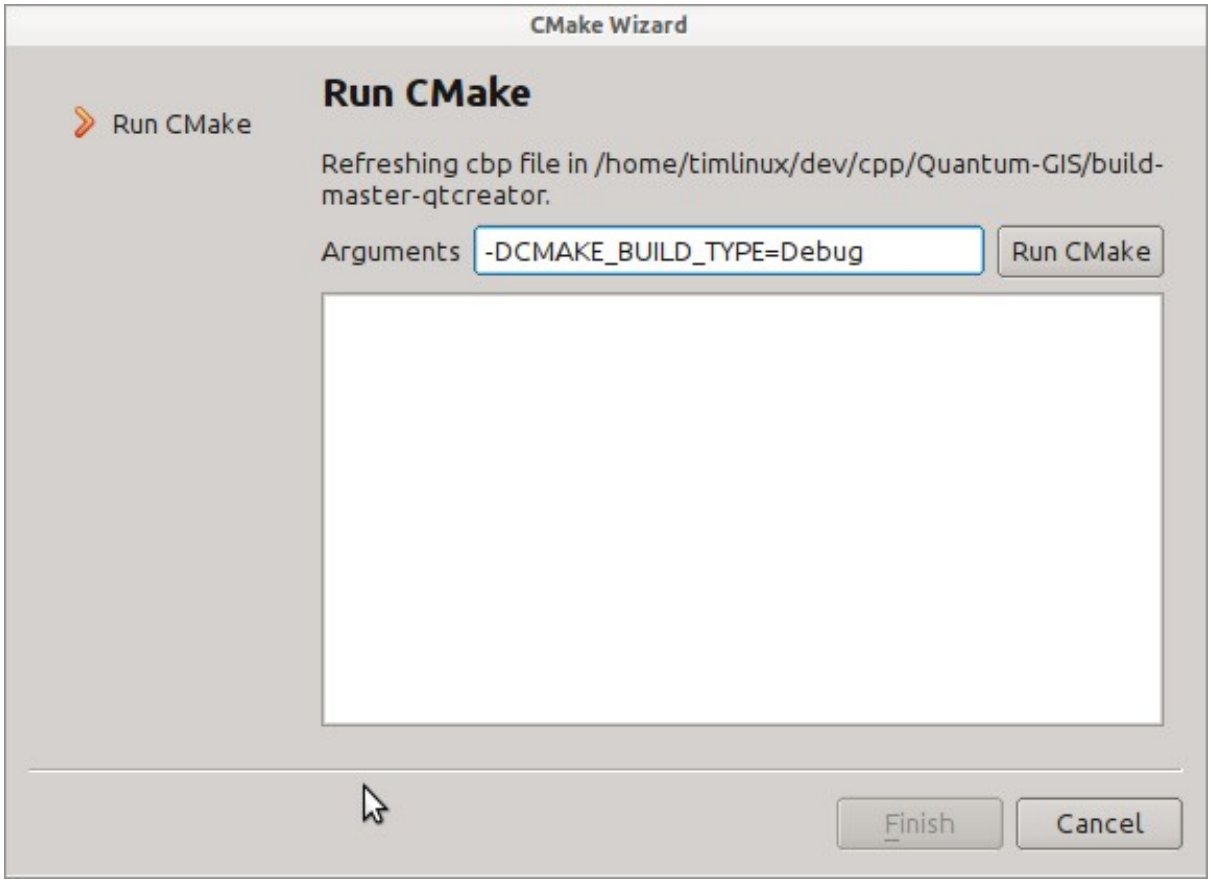
```
$HOME/dev/cpp/QGIS/build-master-qtcreator
```

Probablemente es una buena idea crear directorios de compilación separados para diferentes ramas si se lo permite su espacio de disco.



A continuación, se le preguntará si tiene alguna opción de compilación de CMake para pasar a CMake. Le diremos a CMake que queremos depurar la compilación agregando esta opción:

```
-DCMAKE_BUILD_TYPE=Debug
```

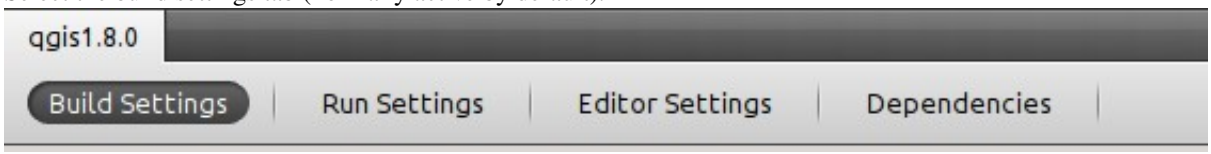


That's the basics of it. When you complete the Wizard, QtCreator will start scanning the source tree for autocompletion support and do some other housekeeping stuff in the background. We want to tweak a few things before we start to build though.

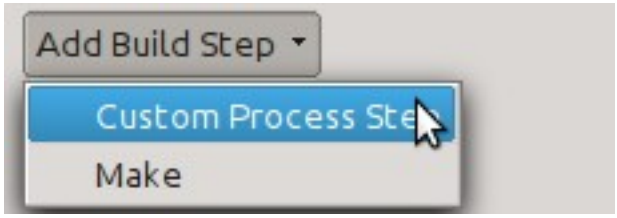
4.3 Setting up your build environment

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'.



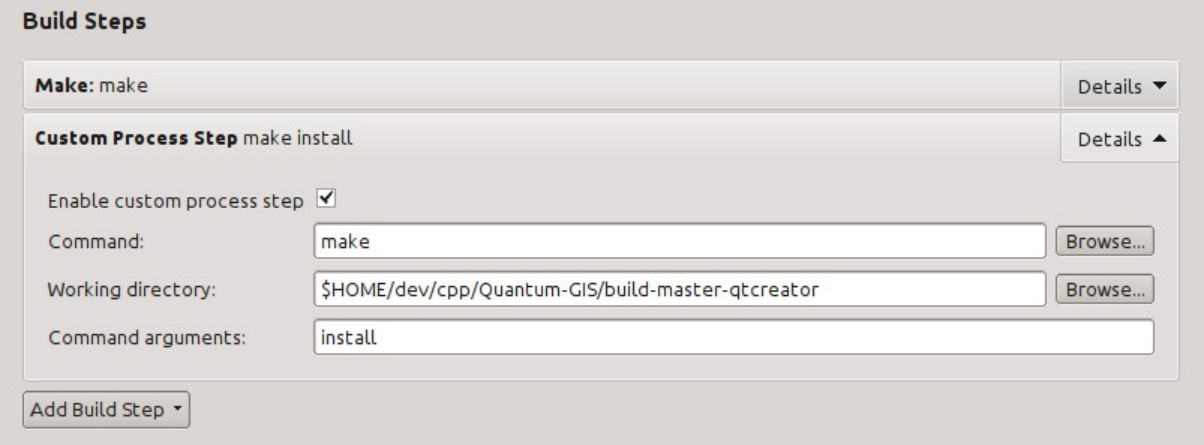
Ahora definimos los siguientes detalles:

Enable custom process step: [yes]

Comando: make

Directorio de trabajo: $HOME/dev/cpp/QGIS/build-master-qtcreator

Argumentos de comando: install



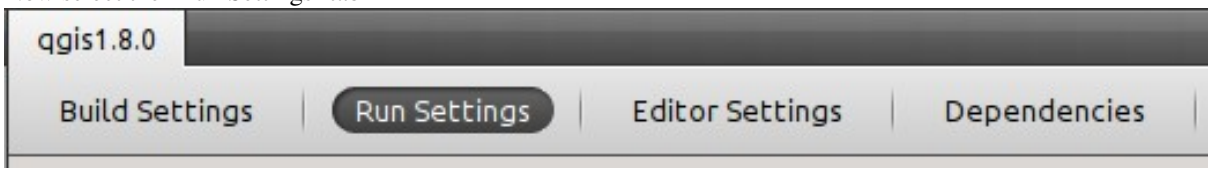
You are almost ready to build. Just one note: QtCreator will need write permissions on the install prefix. By default (which I am using here) QGIS is going to get installed to `/usr/local/`. For my purposes on my development machine, I just gave myself write permissions to the /usr/local directory.

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

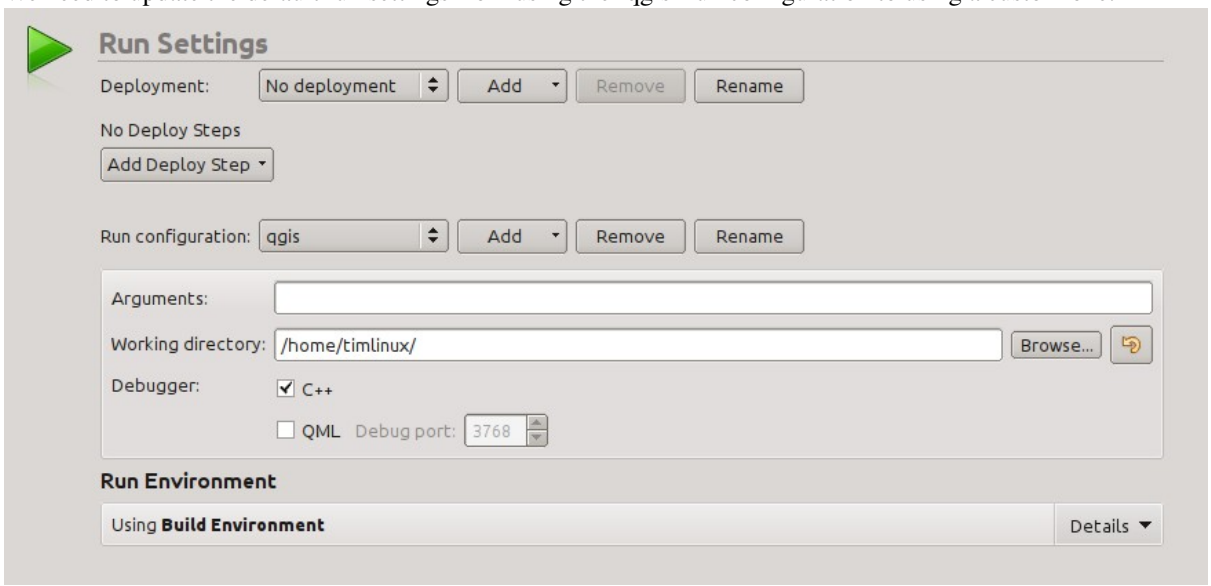
4.4 Estableciendo su ambiente de ejecución

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:

Ejecutable: /usr/local/bin/qgis

Argumentos :

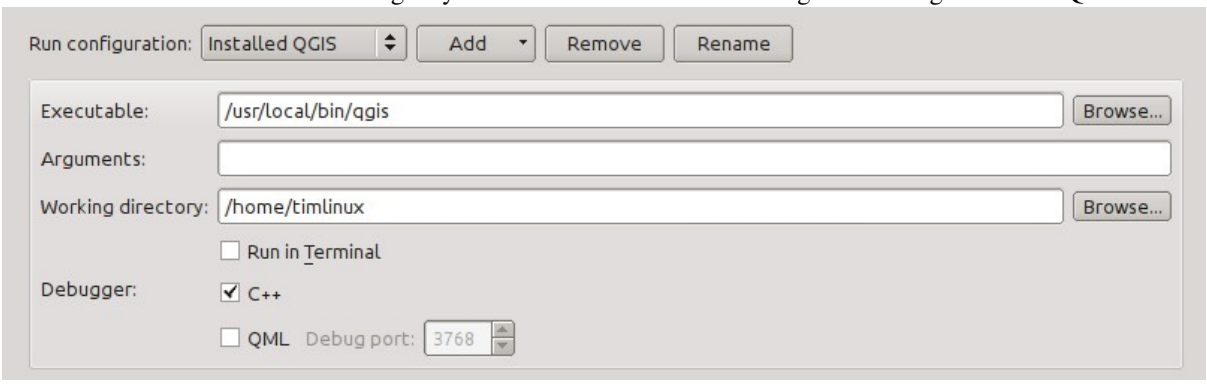
Working directory: $HOME

Ejecutar en la terminal: [no]

Debugger: C++ [yes]

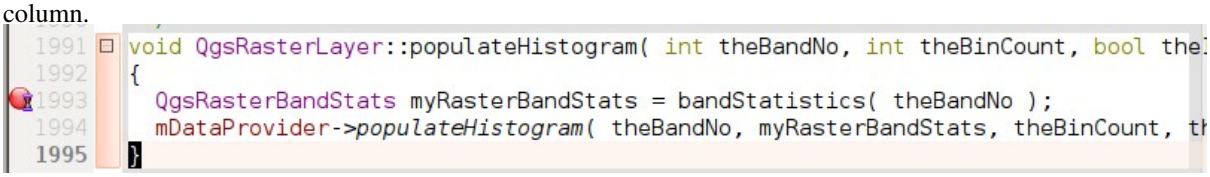
Qml [no]

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

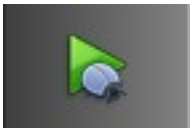


4.5 Ejecución y depuración

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

GIH (Guías de Interfaz Humano)

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. Agrupe los elementos relacionados utilizando cuadros de grupo: intente identificar elementos que puedan agruparse y luego use cuadros de grupo con una etiqueta para identificar el tema de ese grupo. Evite usar cuadros de grupo con solo un control / elemento dentro.
2. Capitalise first letter only in labels: Labels (and group box labels) should be written as a phrase with leading capital letter, and all remaining words written with lower case first letters
3. No termine las etiquetas de widgets o cuadros de grupo con dos puntos: Agregar dos puntos produce un ruido visual y no le da un significado adicional, por lo tanto, no los use. Una excepción a esta regla es cuando tiene dos etiquetas una al lado de la otra ejemplo: Label1 Plugin (Path :) 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. Siempre use un botón QButtonBox para 'Aceptar', 'Cancelar' etc: El uso de botones asegura que el orden de 'Aceptar' y 'Cancelar' etc, los botones son consistentes con el sistema operativo/ lugar / ambiente de escritorio que el usuario esta utilizando.
6. Las pestañas no deberían estar anidadas. Si utiliza pestañas, siga el estilo de las pestañas utilizadas en QgsVectorLayerProperties / QgsProjectProperties etc. p. j. las pestañas hasta arriba con iconos de 22x22.
7. Las pilas de control deben evitarse si es posible. Ellos causan problemas con los diseños y el cambio de tamaño inexplicable (para el usuario) de los cuadros de diálogos para acomodar controles que no son visibles.
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. Utilice iconografía consistente. Si necesita un icono o elementos de icono, contacte a Robert Szczepanek en la lista de correo para asistencia.
10. Coloque largas listas de widgets en cuadros de desplazamiento. Ningún cuadro de diálogo debe exceder los 580 píxeles de altura y 1000 píxeles de ancho.
11. Opciones avanzadas separadas de las básicas. Los usuarios novatos deben poder acceder rápidamente a los elementos necesarios para las actividades básicas sin tener que preocuparse por la complejidad de las funciones avanzadas. Las características avanzadas deben ubicarse debajo de una línea divisoria o ubicarse en una pestaña separada.
12. No agregue opciones por el simple hecho de tener muchas opciones. Esfuércese por mantener la interfaz de usuario minimalista y use valores predeterminados razonables.
13. If clicking a button will spawn a new dialog, an ellipsis (...) should be suffixed to the button text.

5.1 Autores

- Tim Sutton (autor y editor)
- Gary Sherman
- Marco Hugentobler
- Matthias Kuhn

CHAPTER 6

Pruebas de conformidad OGC

- Configuración de las pruebas de conformidad WMS 1.3 y WMS 1.1.1
- Proyecto de prueba
- Ejecución de la prueba WMS 1.3.0
- Ejecución de la prueba WMS 1.1.1

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.

6.1 Configuración de las pruebas de conformidad WMS 1.3 y WMS 1.1.1

```
sudo apt-get 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
```

Descargar e instalar las pruebas WMS 1.3.0

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

Descargar e instalar las pruebas WMS 1.1.1

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

6.2 Proyecto de prueba

Para las pruebas WMS, los datos se pueden descargar y cargar en un proyecto QGIS:

```
wget http://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 http://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.

6.3 Ejecución de la prueba WMS 1.3.0

```
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
```

6.4 Ejecución de la prueba WMS 1.1.1

```
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
```

Pruebas de conformidad OGC, 34