QGIS Developers Guide

Versión 3.4

QGIS Project

15 de marzo de 2020

Contents

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

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 caja camello.

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.

Los nombres de miembros de clases estáticas mutables debería comenzar con una letra minúscula `s`, pero los nombres de miembros de clases estáticas constantes deberían ser todo en mayúsculas:

- `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

Los argumentos de una función deberían tener nombres descriptivos. No use argumentos nombrados con una única letra (p.e.: use `setColor( const QColor& color )` en vez de `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

Devuelva objetos pequeños y que se puedan copiar de forma trivial como valores. Los objetos más grandes deberían devolverse como una referencia constante. Una excepción a esto son los objetos compartidos de forma implícita, que siempre se devuelven por valor. Devuelva los objetos de tipo `QObject` u objetos de subclases de este como punteros.

- `int maximumValue() const`
- `const LayerSet& layers() const`
- `QString title() const` (`QString` se comparte de forma implícita)
- `QList< QgsMapLayer* > layers() const` (`QList` se comparte de manera implícita)
- `QgsVectorLayer *layer() const;` (`QgsVectorLayer` hereda de `QObject`)
- `QgsAbstractGeometry *geometry() const;` (`QgsAbstractGeometry` es abstracta y probablemente necesitará ser convertida al tipo específico)

1.2 Documentación de la API

Se necesita escribir la documentación de la API para cada clase, método, enumerado y cualquier otro código que esté accesible a través de la API pública.

QGIS usa Doxygen para su documentación. Escriba comentarios descriptivos y con sentido para dar al lector información sobre lo que debe esperar, que sucede en los casos extremos y dar pistas acerca de otras interfaces que podría estar buscando, buenas prácticas y ejemplos de código.

1.2.1 Métodos

Las descripciones de los métodos deben estar escritas de forma descriptivas, usando la 3a persona. Los métodos necesitan una etiqueta `\since` para indicar cuando se han añadido. Debería añadir etiquetas `\since` adicionales para aquellos cambios importantes que se hayan introducido posteriormente.

```
/**
 * 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 Atributos de clase

Los atributos de clase deberían normalmente estar en la sección `private` y acceder a ellos a través de métodos getters y setters. Una excepción a esto son los contenedores de datos como los que se usan para informar de errores. En esos caso no se debe añadir el prefijo `m` al atributo.

```
/**
 * \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 Diseñador Qt

1.3.1 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`

1.3.2 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.

1.4 Archivos C++

1.4.1 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.

Por ejemplo: los ficheros fuente de la clase `QgsFeatureAttribute` son `qgsfeatureattribute.cpp` y `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.

1.4.2 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*.

1.5 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`

1.6 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`

1.7 Constantes & Macros Globales

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

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

1.8 Comentarios

Los comentarios a los métodos de clase deben usar un estilo indicativo en tercera persona en lugar del estilo imperativo:

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

1.9 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.

1.10 Editando

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

1.10.1 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`

1.10.2 Indentación

El código fuente debe sangrarse para facilitar su lectura. Existe un script `scripts/prepare-commit.sh` que revisa los ficheros que se han cambiado y les incluye la sangría correspondiente usando astyle. Este script debe ejecutarse antes de realizar una confirmación de cambios. También puede usarse el script `scripts/astyle.sh` para sangrar ficheros individuales.

Puesto que las nuevas versiones de astyle sangran de manera diferente a la versión que se usó para realizar el sangrado completo de todo el código fuente, el script usa una versión antigua de astyle que hemos incluido en nuestro repositorio (habilite `WITH_ASTYLE` en cmake para incluirla en la compilación)

1.10.3 Llaves

Llaves deberían iniciar la línea que sigue a la expresión:

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

1.11 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 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 Enlaces con SIP

Algunos de los ficheros SIP se generan de forma automática a través de un script específico.

1.12.1 Preprocesamiento de cabeceras

Toda la información necesaria para generar de manera correcta el fichero SIP debe estar presente en el fichero de cabecera de C++. Están disponibles algunas macros para dicha definición:

- Use `#ifdef SIP_RUN` para generar código sólo en los ficheros SIP o `#ifndef SIP_RUN` para generar sólo código C++. Las sentencias `#else` se procesan en ambos casos.
- Use `SIP_SKIP` para descartar una línea
- Se procesan las siguientes anotaciones:
  - `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/

- Las secciones private no se muestran, excepto si usa una sentencia #ifdef SIP_RUN en este bloque.
- SIP_PYDEFAULTVALUE(value) puede usarse para definir un valor alternativo por defecto para el método Python. Si el valor por defecto contiene una coma , , el valor debe rodearse por comillas simples '
- SIP_PYTYPE(type) puede usarse para definir un tipo alternativo para el argumento de un método Python. Si el tipo contiene una coma , , el tipo debe rodearse por comillas simples '

Se puede encontrar un fichero de demostración en tests/scripts/sipifyheader.h.

1.12.2 Generando el fichero SIP

El fichero SIP se puede generar usando un script específico. Por ejemplo:

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

Tan pronto como un fichero SIP se añade a un fichero fuente (python/core/core.sip, python/gui/gui.sip or python/analysis/analysis.sip), se considerará como generado de forma automática. Un fichero de test en Travis se encargará de asegurar que este fichero esté actualizado con su correspondiente cabecera.

Los ficheros más antiguos para los que la creación automática no está habilitada están listados en el fichero python/auto_sip.blacklist.

1.12.3 Mejorando el script sipify

Si se necesitan algunas mejoras en el script sipify, por favor añada los bits faltantes en el fichero de demostración tests/scripts/sipifyheader.h y cree la cabecera esperada tests/scripts/sipifyheader.expected.si. Esto también será automáticamente comprobado con Travis como un test unitario en el propio script.

1.13 Estilo de Codificación

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

1.13.1 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

1.13.2 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

1.13.3 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: El script `scripts/prepare-commit.sh` se encargará de esto.

1.13.4 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();
```

1.13.5 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();
}
```

1.13.6 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)

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

- escribiendo un artículo sobre la nueva funcionalidad en un blog, y añadirlo a planeta QGIS https://plugins.qgis.org/planet/
- agregando su nombre a:
  - https://github.com/qgis/QGIS/blob/release-3_4/doc/CONTRIBUTORS
  - https://github.com/qgis/QGIS/blob/release-3_4/doc/AUTHORS

CHAPTER 2

GIH (Guías de Interfaz Humano)

Con el objetivo de que todos los elementos de la interfaz gráfica aparezcan de forma consistente y que todos los usuarios usen los diálogos de forma instintiva, es importante que las siguientes pautas se tengan en cuenta a la hora de diseñar las interfaces de usuario (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. Escriba en mayúscula sólo la primera letra de las etiquetas, sugerencias, texto descriptivo y otros textos que no sean cabecera o títulos: estos deben escribirse como una frase empezando por letra mayúscula, y el resto de palabras con la primera letra en minúscula, salvo que sean sustantivos
3. Escriba en mayúsculas todas las palabras de los títulos (grupos de elementos, pestañas, columnas de listados, etc.), funciones (elementos de menús, elementos de lista en forma de árbol, etc.): escriba en mayúsculas todas las palabras, excepto preposiciones que tengan menos de 5 letras (por ejemplo, "with" pero no "Without"), conjunciones (por ejemplo, "and", "or", "but") y artículos ("a", "an", "the"). Sin embargo, siempre escriba en mayúscula la primera y última palabra.
4. No termine las etiquetas de elementos 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. Por ejemplo: Label1 Plugin (Path :) Label2 [/ path / to / plugins]
5. Separe las acciones que puedan conllevar algún peligro con las que no: si proporciona acciones para borrar ("delete", "remove", etc.) intente incluir un espacio suficiente entre las acciones peligrosas y las inocuas, de modo que los usuarios sean menos propensos a pulsar de forma inconsciente en la acción peligrosa.
6. Siempre use un QButtonBox para los botones "Aceptar", "Cancelar", etc: el uso de un cuadro de botones asegura que el orden de los botones "Aceptar", "Cancelar", etc, es consistente con el sistema operativo / idioma / ambiente de escritorio que el usuario está utilizando.
7. Las pestañas no deberían estar anidadas. Si utiliza pestañas, siga el estilo de las pestañas utilizadas en QgsVectorLayerProperties / QgsProjectProperties / etc. Por ejemplo: las pestañas en la parte superior con iconos de 22x22.
8. La acumulación de elementos de la interfaz deben evitarse todo lo posible. Causan problemas con los diseños y el cambio de tamaño inexplicable (para el usuario) de los cuadros de diálogos para acomodar elementos de la interfaz que no son visibles.
9. Intente evitar términos técnicos y use mejor el equivalente «profano». P.ej: use la palabra "Opacidad" en vez de "Canal Alfa" (un ejemplo forzado), "Texto" en vez de "Cadena", etc.

10. Utilice iconografía consistente. Si necesita un icono o elementos de icono, contacte con Robert Szczepanek en la lista de correo para solicitarle asistencia.
11. Coloque largas listas de elementos de interfaz en cuadros de desplazamiento. Ningún cuadro de diálogo debe exceder los 580 píxeles de altura y 1000 píxeles de ancho.
12. Separe las opciones avanzadas 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 funcionalidades avanzadas. Las características avanzadas deben situarse debajo de una línea divisoria o en una pestaña separada.
13. No agregue opciones por el simple hecho de tener muchas opciones. Esfuércese por mantener la interfaz de usuario minimalista y use valores predeterminados razonables.
14. Si pulsar un botón hace que se muestre un nuevo cuadro de diálogo, se debe incluir como sufijo unos puntos suspensivos (...) al texto del botón. Asegúrese de usar el carácter puntos suspensivos horizontales U+2026 en vez de tres puntos.

2.1 Autores

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

CHAPTER 3

Acceso a GIT

Esta sección describe como empezar a usar el repositorio GIT de QGIS. Antes de empezar, necesita tener instalado primero un cliente git en su sistema.

3.1 Instalación

3.1.1 Instalar git en GNU/Linux

Los usuarios de distribuciones basadas en Debian pueden hacer:

```
sudo apt install git
```

3.1.2 Instalar git en Windows

Los usuarios de Windows pueden usar msys git o usar el cliente git suministrado con cygwin.

3.1.3 Instalar git en OSX

El proyecto git tiene una compilación de git descargable. Asegúrese de obtener el paquete compatible con su procesador (probablemente x86_64, solo los primeros Mac con procesador Intel necesitan el paquete i386).

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

Nota PPC/fuente

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

Descargue el código fuente de https://git-scm.com/. Descomprímalo y en un terminal cámbiese al directorio donde esté el código fuente y entonces:

```
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=
```

3.2 Accediendo al repositorio

Para clonar la rama QGIS maestra:

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

3.3 Verificar una rama

Para obtener una rama concreta, por ejemplo, la rama de la versión 2.6.1 debe hacer:

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

Para obtener la rama principal:

```
cd QGIS
git checkout master
```

Nota: En QGIS mantenemos nuestro código más estable en la rama de la versión actual. La rama maestra contiene el código correspondiente a las versiones llamadas "inestables". Publicaremos de forma periódica una versión a partir de la rama maestra y luego continuaremos con la estabilización y la incorporación de forma selectiva de nuevas funcionalidades a la rama principal.

Lea el archivo INSTALL presente en el código fuente para obtener instrucciones especificas sobre la creación de versiones de desarrollo.

3.4 Fuentes de documentación de QGIS

Si está interesado en verificar las fuentes de documentación de QGIS:

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

También puede echar un vistazo al fichero "readme" incluido en la documentación del repositorio para obtener más información

3.5 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.

3.6 Documentación sobre GIT

Consulte los siguientes sitios para obtener información acerca de cómo convertirse en un maestro de GIT.

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

3.7 Desarrollo usando ramas

3.7.1 Objetivo

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 Procedimiento

- **Anuncio inicial en la lista de correo:** Antes de comenzar, haga un anuncio en la lista de correo de desarrollo para ver si otro desarrollador está ya trabajando en la misma funcionalidad. Contacte también con el asesor técnico del comité directivo del proyecto (PSC). Si la nueva funcionalidad requiere algún tipo de cambio en la arquitectura de QGIS, será necesaria una petición de comentarios (RFC)

Crear una rama: Cree una nueva rama en GIT para el desarrollo de la nueva funcionalidad.

```
git checkout -b newfeature
```

Ahora puede empezar a desarrollar. Si planea hacer tareas de larga duración en dicha rama, desea compartir el trabajo con otros desarrolladores, y tener acceso de escritura al repositorio padre, puede enviar su repositorio al repositorio oficial de QGIS haciendo lo siguiente:

```
git push origin newfeature
```

Nota: Si en la rama ya existe sus cambios serán introducidos en él.

Restablecer desde la rama principal de forma regular: Es recomendable reestablecer ("rebase") para incorporar los cambios realizados en la rama principal a la rama de forma regular. Esto facilita la fusión de la rama nueva a la maestra más tarde. Después de reestablecer la base es necesario ejecutar el comando `git push -f` en el repositorio bifurcado.

Nota: ¡Nunca haga `git push -f` al repositorio original! Utilice esto sólo durante el trabajo en la rama.

```
git rebase master
```

3.7.3 Realice las pruebas necesarias antes de volver a fusionar a la rama principal

Cuando esté terminada la nueva funcionalidad y esté 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.

3.8 Enviando parches y peticiones de incorporación de cambios ("Pull Requests")

Existen algunas pautas que lo ayudarán a conseguir que sus parches y peticiones de incorporación de cambios ("pull requests") formen parte de QGIS fácilmente, y nos ayudarán a facilitar el uso de los parches que se envían.

3.8.1 Peticiones de incorporación de cambios ("Pull Requests")

En general es más fácil para los desarrolladores si se envían peticiones de incorporación de cambios de GitHub. No describiremos en qué consisten las peticiones de incorporación de cambios aquí, sino que le recomendamos leer la Documentación de peticiones de incorporación de cambios en GitHub.

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.

Si es desarrollador y desea evaluar la cola de peticiones de incorporación de cambios, hay una herramienta muy amigable que le permite hacer esto desde línea de comandos

Por favor, consulte el apartado a continuación sobre "cómo alertar de su parche. En general, cuando envíe una petición de incorporación de cambios debe asumir la responsabilidad de seguirla hasta su finalización - responder a las preguntas publicadas por otros desarrolladores, buscar un "campeón" para su funcionalidad y recordarles de manera amable si detecta que su petición no está siendo revisada. Tenga en cuenta que el proyecto QGIS está impulsado por el esfuerzo voluntario y es posible que las personas no puedan revisar su petición de incorporación de cambios de forma instantánea. Si cree que la petición no está recibiendo la atención que merece sus opciones para acelerarla deberían ser (en orden de prioridad):

- Envíe un mensaje a la lista de correo "promocionando" su petición de incorporación de cambios y qué maravilloso sería incluirlo en el código base.
- Envíe un mensaje a la persona a la que se le asignó su petición de incorporación de cambios en la cola de peticiones de incorporación de cambios.
- Envíe un mensaje a Marco Hugentobler (es quién gestiona la cola de peticiones de incorporación de cambios).
- Envíe un mensaje al comité directivo del proyecto pidiendo que le ayuden a incorporar su petición de incorporación de cambios en el código base.

Mejores prácticas para la creación de una petición de incorporación de cambios

- Inicie siempre una rama para una nueva funcionalidad a partir de la rama principal actual.
- Si está programando en una rama una nueva funcionalidad, no ""fusione"" ("merge") nada en dicha rama; en lugar de ello use la opción de restablecer la base ("rebase") como se indica en el siguiente punto para mantener su historial limpio.
- Antes de crear una petición de incorporación de cambios ejecute `git fetch origin` y `git rebase origin/master` (dado que el origen es repositorio padre remoto y no su propio remoto, verifique su fichero `.git/config` o ejecute ‘‘ git remote -v | grep github.com / qgis‘‘).
- Puede hacer una reestablecer la base ("rebase") de git como en la última línea repetidamente sin hacer ningún daño (siempre y cuando el único propósito de su rama sea fusionarse con la rama principal).
- Atención: después de un rebase necesitará ejecutar `git push -f` en su repositorio bifurcado. **DESARROLLADORES DEL NÚCLEO: NO HAGAN ESTO EN EL REPOSITORIO PÚBLICO DE QGIS!**

Etiquetas especiales para notificar a los documentadores

Además de las etiquetas comunes que puede añadir a su petición de incorporación de cambios para clasificarla, existen otras etiquetas especiales que puede usar para generar de forma automática informes de incidencias en el repositorio de documentación de QGIS tan pronto como su petición de incorporación de cambios se fusione:

- `[needs-docs]` para indicar a los redactores de documentación que agreguen documentos extra después de una corrección o mejora de una funcionalidad ya existente.
- `[feature]` in case of new functionality. Filling a good description in your PR will be a good start.

Desarrolladores, utilicen por favor estas etiquetas (no distingue entre mayúsculas y minúsculas) para que los redactores de documentos tengan incidencias en las que trabajar y tengan una visión general de las cosas pendientes por hacer. PERO por favor dediquen también tiempo para añadir algo de texto: ya sea en el mensaje al guardar los cambios o en la documentación.

Para fusionar una petición de incorporación de cambios

Opción A:

- Pulse sobre el botón fusionar ("merge") (Crea una fusión que no es de avance rápido)

Opción B:

- Compruebe la petición de incorporación de cambios

- Prueba (también se necesita para la opción A, obviamente)
- verificar master, fusión git pr/1234
- Opcional: `git pull --rebase`: Crea un avance rápido, no se hace «merge commit». El historial estará más limpio, pero será más difícil revertir la fusión.
- `git push` (NUNCA JAMÁS utilizar la opción -f aquí)

3.9 Política de nombres para ficheros de parches

Si el parche es una solución para un error específico, nombre el archivo con el número de error, p. bug777fix.patch y adjúntelo al *informe de error original en GitHub <https://github.com/qgis/QGIS/issues> _*.

Si el error es una mejora o una nueva característica, generalmente es una buena idea crear un *ticket en GitHub <https://github.com/qgis/QGIS/issues> _* primero y luego adjunte su parche.

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

De esta forma es más fácil de aplicar los parches ya que no necesitamos navegar a un lugar específico del código fuente para aplicar el parche. Además, cuando recibo parches, generalmente los evalúo usando "merge", y estando el parche en el 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 original, y luego generará un parche que contiene las diferencias entre su rama con los cambios y lo que está en la rama maestra.

3.10.1 Llamando la atención sobre su parche

Los desarrolladores de QGIS son gente ocupada. Escaneamos los parches entrantes en los informes de errores, pero a veces echamos de menos cosas. No se ofenda ni se alarme. Intente identificar a un desarrollador para que lo ayude y contáctelos preguntándoles si pueden ver su parche. Si no recibe ninguna respuesta, puede escalar su consulta a uno de los miembros del Comité Directivo del Proyecto (los detalles de contacto también están disponibles en Recursos Técnicos).

3.10.2 Auditoría

QGIS se distribuye bajo licencia GPL. Debe hacer todos los esfuerzos posibles para garantizar que sólo envía parches que no estén sujetos a derechos de propiedad intelectual contradictorios. Además, no envíe código que no esté contento de haber puesto a disposición en virtud de la GPL.

3.11 Obteniendo acceso de escritura en 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) que demuestren la competencia básica y la comprensión de las convenciones de codificación C++ y de 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 proporcionar 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 de C++, p. ej., para traductores y documentadores. En estos casos, la persona aún así 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 del código fuente base 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 creando una bifurcación de QGIS y generando peticiones de incorporación de cambios.

Verifique siempre que todo compila antes de realizar cualquier solicitud de confirmación / petición de incorporación de cambios. Intente tener en cuenta las posibles roturas que sus confirmaciones pueden causar a las personas que compilan en otras plataformas y con 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 debería hacer un comentario en la parte superior del archivo (encima del área que dice "no cambiar esto"). Ponga un comentario descriptivo y es mejor hacer varias pequeñas confirmaciones si los cambios realizados a un conjunto de archivos no están relacionados. Por el contrario, preferimos que agrupe los cambios relacionados en una única confirmación.

CHAPTER 4

Iniciando y ejecutando QtCreator y QGIS

QtCreator es una IDE bastante nuevo de los creadores de la librería Qt. Con QtCreator puede construir cualquier proyecto C++, pero esta realmente optimizado para la gente que trabaja con aplicaciones basadas en Qt(4) (incluyendo aplicaciones móviles). Todo lo que describo a continuación asume que está ejecutando 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

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

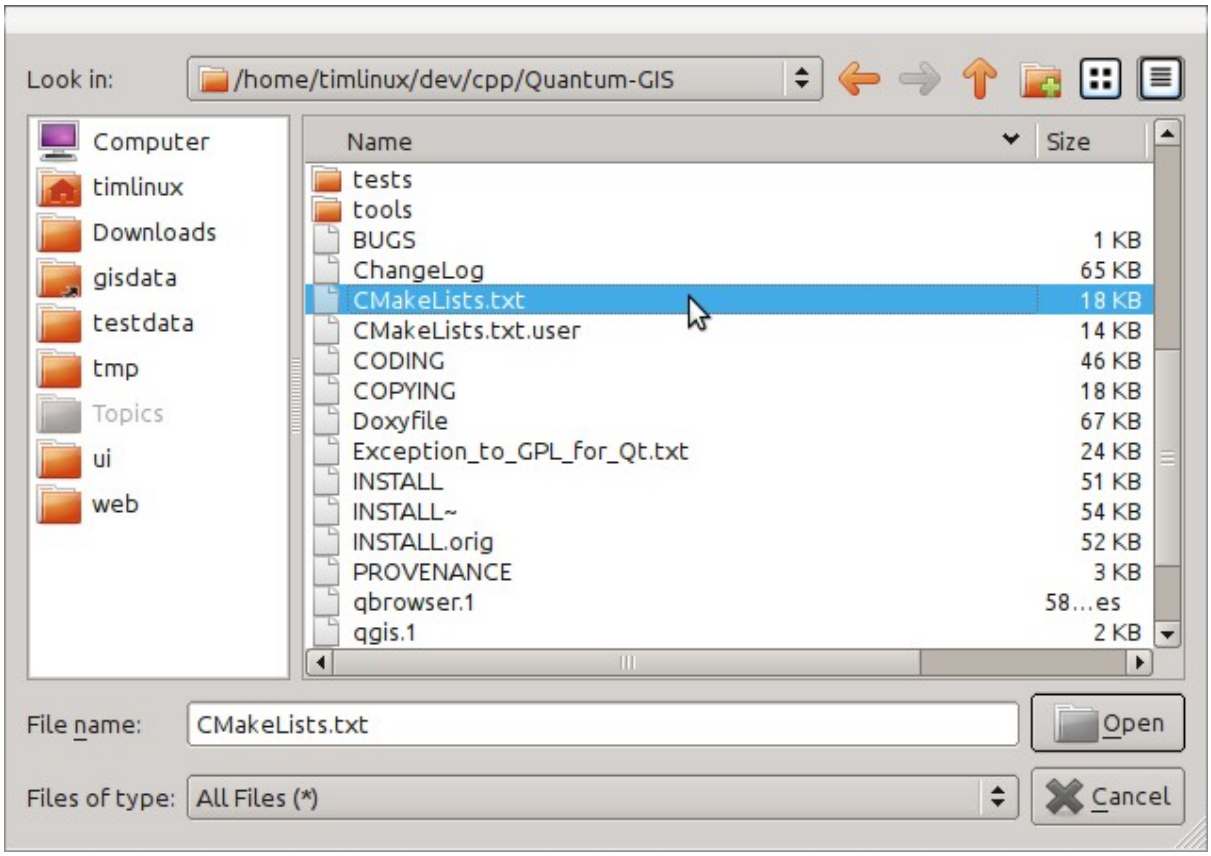
En mi sistema he descargado el código en `$HOME/dev/cpp/QGIS` y el resto del artículo está escrito suponiendo esto. Debería actualizar las rutas según corresponda para su sistema local.

Al iniciar QtCreator haga:

File -> Open File or Project

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

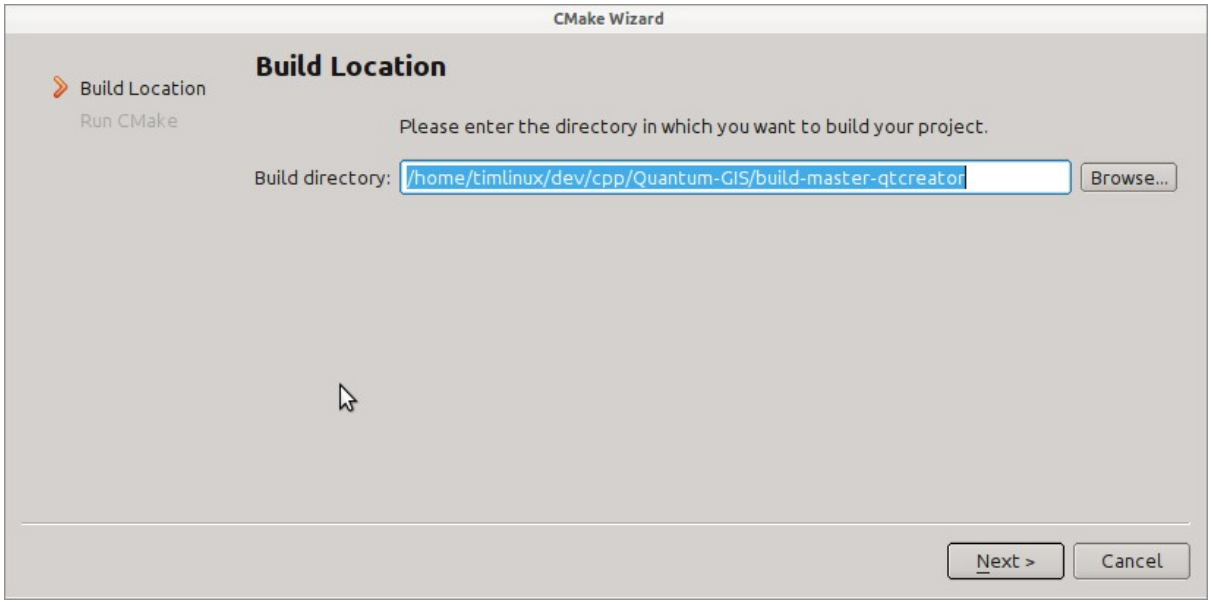
```
$HOME/dev/cpp/QGIS/CMakeLists.txt
```



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
```

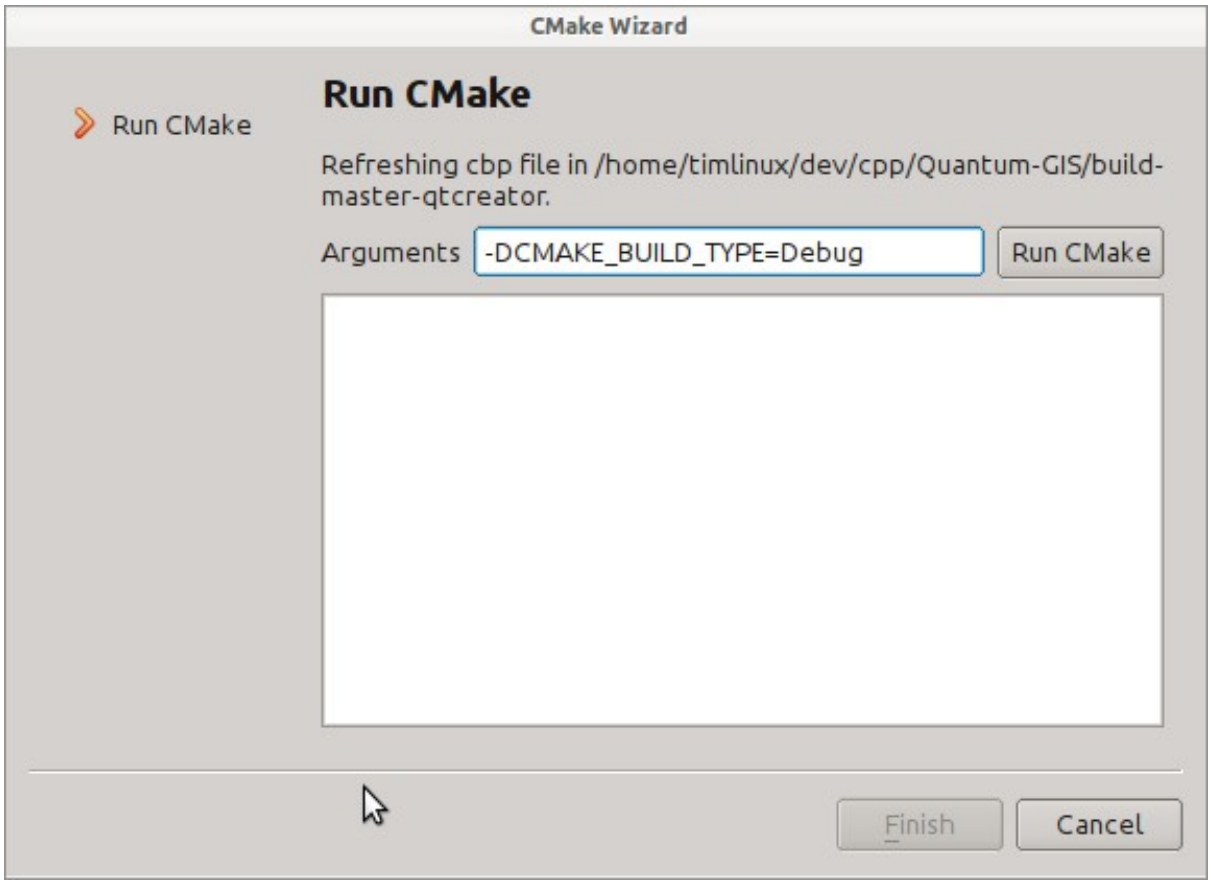
Probablemente es una buena idea crear directorios de compilación separados para diferentes ramas si se lo permite su espacio de disco.



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 añadiendo esta opción:

```
-DCMAKE_BUILD_TYPE=Debug
```

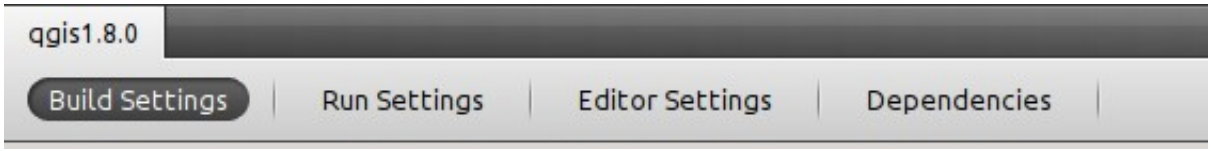


Y esto es lo básico. Cuando termine el asistente, QtCreator empezará a escanear el árbol de fuentes para el soporte de autocompletar y hacer algunas tareas propias de fondo. Sin embargo, queremos ajustar unas cuantas cosas antes de empezar a compilar.

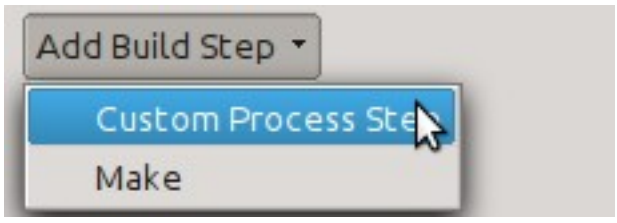
4.3 Configurando el entorno de compilación

Pulse en el icono “Projects” a la izquierda de la ventana de QtCreator.

Seleccione la pestaña de opciones de compilación (normalmente está activa por defecto).



Ahora queremos añadir un paso personalizado en el proceso. ¿Por qué? Porque QGIS sólo puede ejecutarse desde un directorio de instalación, no en su directorio de compilación, por lo que necesitamos asegurarnos de que está instalado donde vayamos a compilarlo. En "Build Steps", pulse en el botón "Add Build Step" y elija la opción "Custom Process Step".



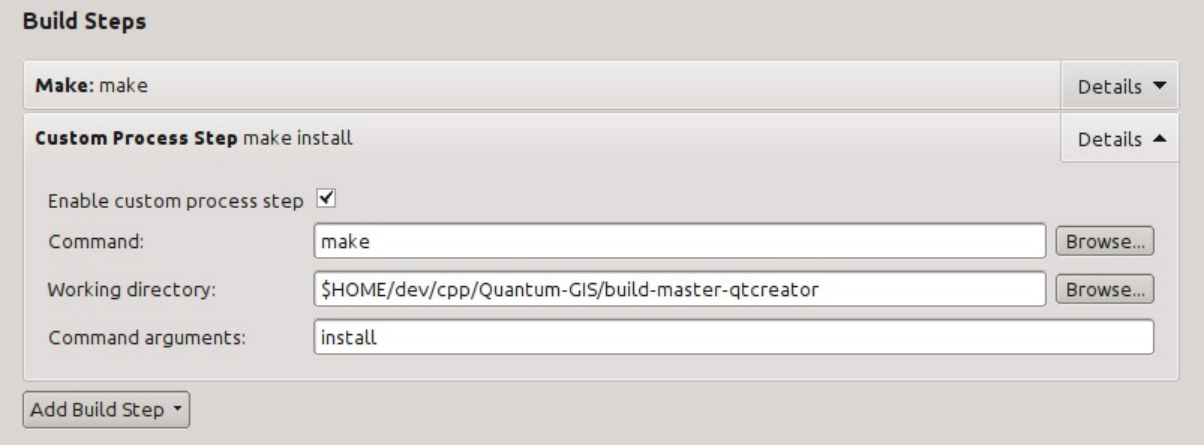
Ahora definimos los siguientes detalles:

Enable custom process step: [yes]

Command: make

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

Argumentos de comando: install



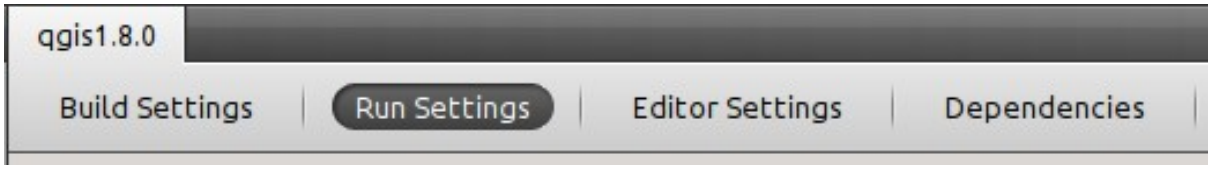
Ya está casi listo para empezar a compilar. Sólo una nota adicional: QtCreator necesita permisos de escritura en la ruta de instalación. Por defecto (que es lo que usaré) QGIS se va a instalar en `/usr/local/`. Para los objetivos de mi máquina de desarrollo, simplemente me he dado permisos a mi mismo de escritura en el directorio /usr/local/.

Para iniciar la compilación, pulse en el icono del martillo grande que está situado en la parte inferior izquierda de la ventana.

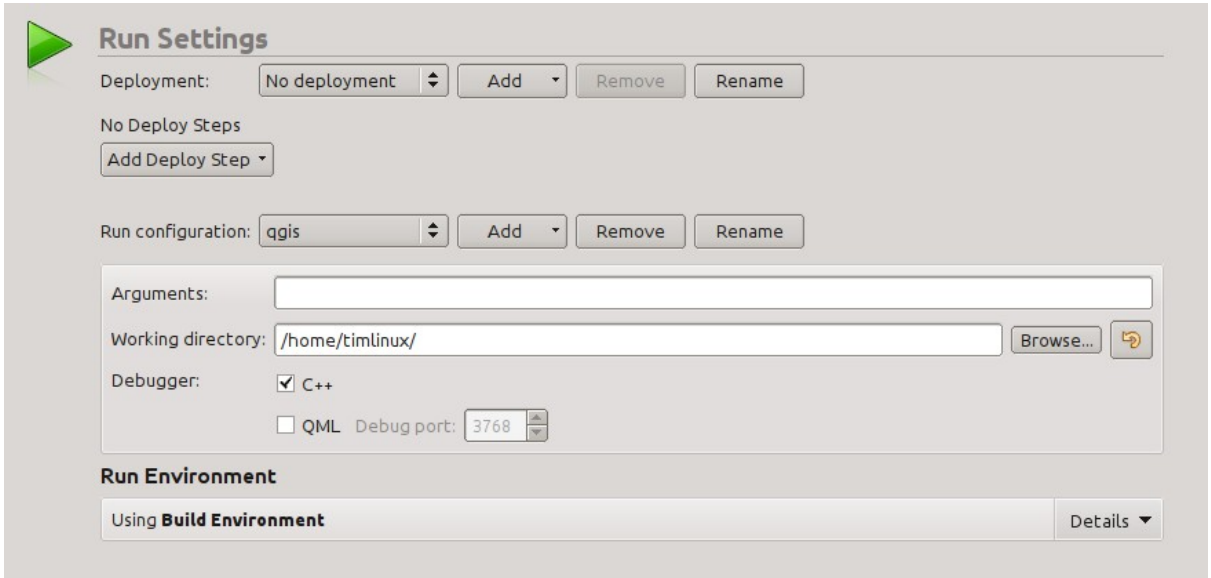
4.4 Estableciendo su ambiente de ejecución

Como ya mencioné anteriormente, no podemos ejecutar QGIS directamente desde el directorio de compilación, por lo que tenemos que crear un destino de ejecución personalizado para decirle a QtCreator que ejecute QGIS desde el directorio de instalación (en mi caso `/usr/local/`). Para hacer eso, volvamos a la pantalla de configuración de proyectos.

Seleccione ahora la pestaña “Run Settings”



Necesitamos actualizar las opciones de ejecución por defecto para cambiar de la configuración de ejecución “qgis” a una personalizada.



Para ello, pulse el botón “Add v” junto al desplegable “Run configuration” y seleccione “Custom Executable” de la parte superior de la lista.

A continuación introduzca en el área de propiedades los siguientes detalles:

Executable: /usr/local/bin/qgis

Arguments :

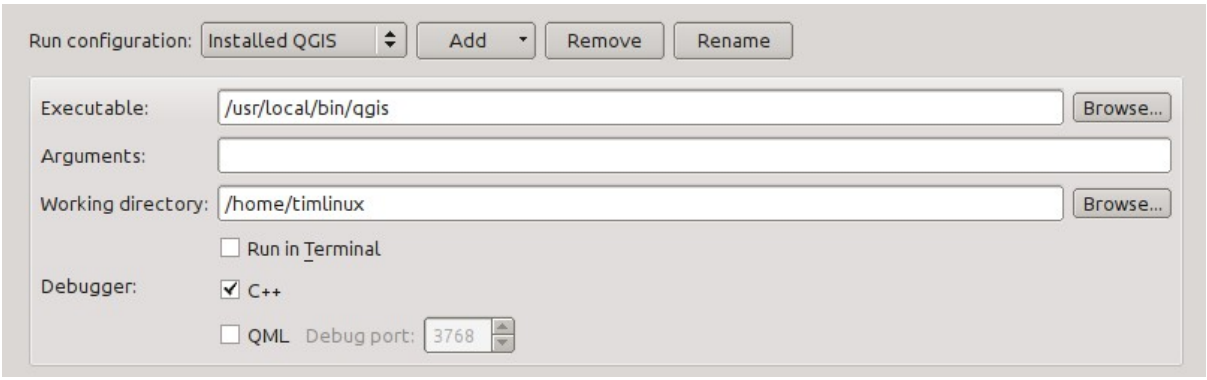
Working directory: $HOME

Run in terminal: [no]

Debugger: C++ [yes]

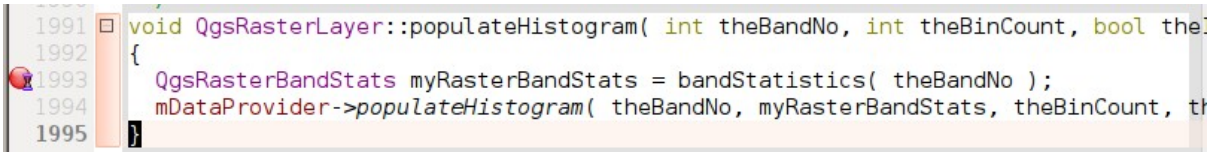
Qml [no]

A continuación pulse el botón “Rename” y proporcione a su ejecutable personalizado un nombre con significado (p.ej. “QGIS instalado”)

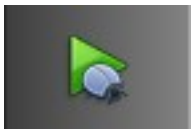


4.5 Ejecución y depuración

Ya está listo para ejecutar y depurar QGIS. Para establecer un punto de ruptura, abra simplemente un archivo fuente y pulse en la columna de la izquierda.



Ahora inicie QGIS en el depurador pulsando el icono con el bicho que está situado en la parte inferior izquierda de la ventana.



CHAPTER 5

Pruebas unitarias

A noviembre de 2007, requerimos que todas las nuevas características que vayan a entrar al master a ser acompañados con una unidad de prueba. Inicialmente, hemos limitado a este requisito qgis_core, y ampliaremos este requerimiento a otras partes del código base una vez que la gente está familiarizada con los procedimientos para las pruebas unitarias que se explican en las siguientes secciones que siguen.

5.1 La infraestructura de prueba de QGIS - una visión general

La prueba de unidad se lleva a cabo utilizando una combinación de QTestLib (la biblioteca de pruebas Qt) y CTest (una infraestructura para compilar y ejecutar pruebas como parte del proceso de construcción de CMake). Hagamos una vista general del proceso antes de ahondar en los detalles:

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. Creas una unidad de pruebas. Esto ocurre en `<QGIS Source Dir>/tests/src/core` en el caso de la librería núcleo. La prueba es básicamente un cliente que crea una instancia de una clase y llama a algunos métodos de esa clase. Comprobará los resultados de cada método para asegurar que coincide con los valores esperados. Si alguna de las llamadas falla, la prueba fallará.
3. Incluyes macros de QtTestLib en tu clase de pruebas. Este macro es procesado por el meta object compiler (moc) de Qt y convierte tu clase de prueba en una aplicación ejecutable.
4. Se agrega una sección a la CMakeLists.txt en el directorio de pruebas que construirá su prueba.
5. Te aseguras de que `ENABLE_TESTING` está activado en ccmake / cmakesetup. Esto asegura que tus pruebas son realmente compiladas cuando escribas make.
6. Se añaden opcionalmente datos de prueba a `<QGIS Source Dir>/tests/testdata` si su prueba utiliza la información (por ejemplo, necesita cargar un archivo shape). Estos datos de prueba deben ser lo más pequeñas posible y siempre que sea posible se debe utilizar las bases de datos existentes ya allí. Sus pruebas nunca deben modificar estos datos in situ, sino más bien debe hacer una copia temporal en algún lugar si es necesario.
7. Compilas tu código fuente e instalas. Para ello, utiliza el procedimiento normal `make && (sudo) make install`.
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.

Justo con ese panorama en mente, ahondaremos en un poco más de detalle. Ya hemos hecho la mayor parte de la configuración por ti en CMake y otros lugares en el árbol de código fuente por lo que todo lo que necesitas hacer es la parte fácil - ¡escribir las pruebas unitarias!

5.2 Creando una prueba de unidad

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`) e implementar todos tus métodos de prueba cómo métodos públicos que devuelven void. Vamos a usar una clase de prueba simple `QgsRasterLayer` en la siguiente sección para ilustrar. Por convenio nombraremos nuestra prueba con el mismo nombre que la clase que estamos poniendo a prueba, pero con el prefijo "Test". De manera que la implementación de prueba va en un archivo llamado `testqgsrasterlayer.cpp` de prueba y la propia clase será `TestQgsRasterLayer`. Primero añadimos nuestro titular 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.
 *
 **************************************************************************/
```

A continuación empezamos nuestros includes necesarios para las pruebas que pensamos correr. Hay un include especial que todos las pruebas deben incluir:

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

De este punto en adelante simplemente continua implementando tu clase como de costumbre, trayendo cualquier header que puedas necesitar:

```
//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>
```

Como estamos estamos combinando la declaración y la implementación de la clase en un solo archivo, la declaración de la clase viene a continuación. Comenzamos con nuestra documentación doxygen. Cada prueba debe estar adecuadamente documentada. usamos la directiva ingroup de doxygen de manera que todos los UnitTests aparezcan como un módulo en la documentación Doxygen generada. Después de esto viene una corta descripción de la prueba unitaria y la clase debe heredar de QObject e incluir el macro Q_OBJECT.

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

class TestQgsRasterLayer: public QObject
{
    Q_OBJECT
```

Todos nuestros métodos de prueba están implementados como «slots» privados. El framework QtTest llamará secuencialmente cada método «slot» privado en la clase de pruebas. Existen cuatro métodos «especiales» que si son implementados serán llamados al inicio de la prueba unitaria (`initTestCase`), y al final de la prueba unitaria (`cleanupTestCase`). Antes de que cada método de prueba sea llamado, el método `init()` será llamado y después de que cada método de prueba sea llamado, el método `cleanup()` será llamado. Estos métodos son útiles en cuanto a que te permiten asignar y limpiar recursos antes de correr cada prueba, y la prueba unitaria en su conjunto.

```
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();
```

A continuación vienen tus métodos de prueba, todos los cuales no deben tener parámetros y no deben devolver nada (void / vacío).

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})
```

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 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 Ejecutar las pruebas

La forma más sencilla de ejecutar las pruebas es como parte de su proceso de construcción normal:

```
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	Valor
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 Que te diviertas

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

Prueba de algoritmos de procesamiento

6.1 Pruebas de algoritmo

Nota: 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 proporciona varios algoritmos en el marco de procesamiento. Usted puede ampliar esta lista con algoritmos propios y, como cualquier característica nueva, es necesario agregar pruebas.

Para probar algoritmos, puede agregar entradas en `testdata/qgis_algorithm_tests.yaml` or `testdata/gdal_algorithm_tests.yaml` as appropriate.

Este fichero está estructurado con sintaxis yaml <http://www.yaml.org/start.html>‘_.

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 Cómo

Para agregar una nueva prueba, siga estos pasos:

1. Ejecutar algorithm que se desea comprobar en QGIS desde el processing toolbox. Si el resultado es una capa de linea escoja un GML, con su correspondiente XSD, como salida compatible con tipos de geometría mixta y buena legibilidad. Redirecione la salida a `python/plugins/processing/tests/testdata/expected`. Para las capas de entrada elegidas usar lo que ya hay en la carpeta `testdata`. Si se necesitan datos adicionales, se ponen en `testdata/custom`.
2. Cuando tenga que ejecutar el algoritmo vaya a *Processing* → *History* y busque el algoritmo que acaba de ejecutar.
3. Click con el botón derecho en el algoritmo y click en *Create Test*. Se abrirá una nueva ventana para poner nombre al algoritmo.»
4. Abra el fichero `python/plugins/processing/tests/testdata/algorithm_tests.yaml`, copie ahí el nombre del algoritmo.

La primera cadena de caracteres del comando va a la tecla `algoritmo`, las siguientes a `parametros` y la(s) última(s) a `resultado(s)`.

Lo anterior se traduce en

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

También es posible crear pruebas para los scripts de Procesamiento. Los scripts deberían situarse en el `scripts` en el subdirectorio de los datos de test `python/plugins/processing/tests/testdata/`. El nombre del fichero script debería coincidir con el nombre del script del algoritmo.»

6.1.2 Parámetros y resultados

Parámetros tipo trivial

Los parámetros y resultados se especifican como diccionarios y listas:

```
params:
  INTERVAL: 5
```

```
  INTERPOLATE: True
  NAME: A processing test
```

o

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

Parámetros tipo capa

A menudo se necesitará especificar las capas como parámetros. Para especificar una capa, deberá especificar:

- el tipo, p. ej. «vector» o «raster»
- un nombre, con un path relativo como `expected/polys_centroid.gml`

Así es como se muestra en realidad:

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

Parámetros tipo archivo

Si necesita un archivo externo para probar de algoritmo, necesita especificar el tipo de "archivo" y el path (relativo) al archivo con su "nombre"

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

Resultados

Los resultados se especifican de manera muy similar.

Archivos vector básicos

No podría ser más trivial

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

Agregue los archivos GML y XSD esperados en la carpeta.

Vector con tolerancia

A veces, diferentes plataformas crean resultados ligeramente diferentes que aún son aceptables. En este caso (pero solo entonces) se puede también usar propiedades adicionales para definir cómo se compara una capa.

Para trabajar con una cierta tolerancia para los valores de salida, se puede especificar una propiedad `de comparación` para una salida. La propiedad de comparación puede contener sub-propiedades para `campos`. Esto contiene información sobre con qué precisión se compara un determinado campo (`precisión`) o incluso puede ser `omitida`. Hay un nombre de campo especial `__all__`, el cual aplica una tolerancia a todos los campos. Hay otra propiedad `geometría` que también acepta una `precisión` que se aplicará a todos los vértice.

```
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
```

Archivos ráster

Los archivos raster se comparan con una función hash de suma de verificación. Esto se calcula cuando se crea un test desde la historia del procesamiento.

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

Archivos

Se puede comparar el contenido de un archivo de salida con un archivo de referencia de resultados esperados

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

O puede usar una o más expresiones regulares que serán *verificadas* <https://docs.python.org/3/library/re.html#re.search>‘_ con el archivo «contenido»

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

Directorios

Se puede comparar el contenido de un directorio de salida con un resultado esperado del directorio de referencia

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

6.1.3 Contexto Algoritmo

Hay algunas definiciones más que pueden modificar el contexto del algoritmo; estas se pueden especificar en el nivel superior de la prueba:

- *project* - cargará un proyecto QGIS específico antes de ejecutar el algoritmo. Si no se especifica, el algoritmo se ejecutará con un proyecto vacío.
- `project_crs` - sobrescribe el CRS del proyecto predeterminado - p. ej. `EPSG:27700`
- `ellipsoid` - sobreescribe el elipsoide proyecto predeterminado usado para mediciones, p. ej. `GRS80`

6.1.4 Ejecutando pruebas localmente

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

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

CHAPTER 7

Pruebas de conformidad OGC

El Open Geospatial Consortium (OGC) provee pruebas que pueden ser ejecutadas de manera gratuita para asegurarse de que un servidor cumple con ciertas especificaciones. Este capítulo provee un rápido tutorial para configurar las pruebas WMS en un sistema Ubuntu. En el sitio web del OGC se puede encontrar documentación detallada.

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

```
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
```

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
```

7.2 Proyecto de prueba

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

```
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 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
```

7.4 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
```