QGIS Developers Guide

Version 3.4

QGIS Project

mars 15, 2020

Contents

Bienvenue sur les pages dédiées au développement de QGIS. Vous trouverez ici les règles, les outils et les étapes qui vous permettront de facilement contribuer avec efficacité au code source de QGIS.

CHAPTER 1

Standards de développement QGIS

Ces standards doivent être suivis par tous les développeurs de QGIS.

1.1 Classes

1.1.1 Noms

Les classes dans QGIS commencent par Qgs et sont nommées en utilisant le camel case.

Exemples:

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

1.1.2 Membres

Les noms des membres de classe commencent par un m minuscule et sont composés de majuscules et de minuscules.

- `mMapCanvas`
- `mCurrentExtent`

Tous les membres d'une classe doivent être privés. Il est FORTEMENT déconseillé de déclarer des membres publics. Les membres protégés doivent être évités pour que le membre soit accessible depuis des sous-classes Python, car les membres protégés ne peuvent pas être utilisés à partir des liaisons Python.

Les noms des membres mutables des classes statiques doivent commencer par un `s` minuscule, mais les noms des membres constants des classes statiques doivent être en majuscules :

- `sRefCounter`
- `DEFAULT_QUEUE_SIZE`

1.1.3 Fonctions d'accesseurs

Les valeurs des membres de la classe doivent être obtenues via les fonctions d'accesseur. La fonction doit être nommée sans préfixe « get ». Les fonctions d'accesseur pour les deux membres privés ci-dessus seraient:

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

Assurez-vous que les accesseurs sont correctement marqués avec `const`. Le cas échéant, cela peut nécessiter que les variables membres du type de valeur en cache soient marquées avec `mutable`.

1.1.4 Fonctions

Les noms de fonction commencent par une minuscule et se composent de minuscules/majuscules. Le nom de fonction devrait évoquer sa fonctionnalité.

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

Par souci de cohérence avec l'API QGIS existante et avec l'API Qt, les abréviations doivent être évitées. Par exemple, `setDestinationSize` au lieu de `setDestSize`, `setMaximumValue` au lieu de `setMaxVal`.

Les acronymes devraient également être nommés en utilisant le camel case pour la cohérence. Par exemple, `setXml` au lieu de `setXML`.

1.1.5 Arguments de la fonction

Les noms des arguments de fonctions doivent être descriptifs. Ne pas utiliser de noms à une seule lettre (p. ex. `setColor( const QColor& color )` plutôt que `setColor( const QColor& c )`).

Portez une attention particulière à quand les arguments devraient être passés par référence. À moins que les objets argument soient petits et trivialement copiés (tels que les objets QPoint), ils doivent être passés par une référence const. Par souci de cohérence avec l'API Qt, même les objets partagés implicitement sont passés par une référence const (p. ex. `setTitle( const QString& title )` au lieu de `setTitle( QString title )`.

1.1.6 Valeurs de retour de la fonction

Renvoie les objets petits et trivialement copiés en tant que valeurs. Les objets plus grands doivent être renvoyés par une référence const. La seule exception à cela est implicitement les objets partagés, qui sont toujours renvoyés par valeur. Renvoie « QObject » ou les objets sous-classes en tant que pointeurs.

- `int maximumValue() const`
- `const LayerSet& layers() const`
- `QString title() const` (`QString` est implicitement partagée)
- `QList< QgsMapLayer* > layers() const` (QList est implicitement partagé)
- `QgsVectorLayer *layer() const;` (`QgsVectorLayer` hérite de `QObject`)
- `QgsAbstractGeometry *geometry() const;` (`QgsAbstractGeometry` est abastrait et nécessitera probablement d'être typé)

1.2 Documentation de l'API

Il est requis d'écrire la documentation API pour chaque classe, méthode, enumération et autres codes disponible dans l'API publique.

QGIS utilise Doxygen pour la documentation. Ecrivez des commentaires descriptifs et utiles qui donnent au lecteur des informations à propos de à quoi s'attendre, de ce qu'il se passe dans les cas limites et qui donne des indications à propos d'autres interfaces qu'il pourrait rechercher, les bonnes pratiques et des exemples de code.

1.2.1 Méthode

La description des méthodes doit être écrit sous forme descriptive en utilisant la 3ème personne. Les méthodes nécessitent le mot-clé *since*' qui définit quand elles ont été introduites.

```
/**
 * 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 Variables membres

Les variables membres devraient normalement être dans la section `private` et rendues accessibles via les getters et les setters. Une exception à cela existe pour les conteneurs de données comme pour la remontée d'erreurs. Dans ces cas, ne préfixez pas les membres par un `m`.

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

1.3 Qt Designer

1.3.1 Les classes générées

Les classes QGIS générées depuis des fichiers Qt Designer (.ui) doivent avoir comme suffixe -Base. Cela permet d'identifier la classe comme étant une classe générée.

Exemples:

- `QgsPluginManagerBase`
- `QgsUserOptionsBase`

1.3.2 Dialogues

Toutes les boîtes de dialogue doivent intégrer des info-bulles d'aide pour toutes les icônes de la barre d'outils ainsi que pour les autres widgets appropriés. Ces info-bulles apportent beaucoup à la découverte des fonctionnalités pour les utilisateurs débutants et confirmés.

Assurez-vous que l'ordre des onglets pour les widgets est bien mis à jour à chaque fois que la boîte de dialogue de la couche change.

1.4 Fichiers C++

1.4.1 Noms

L'intégration du C++ et des fichiers en-têtes doivent respectivement avoir une extension en .cpp et en .h. Les fichiers doivent être nommés intégralement en minuscule et, dans le cas des classes, doivent correspondre avec le nom des classes.

Exemple : Pour la classe `QgsFeatureAttribute`, les fichiers sources sont `qgsfeatureattribute.cpp` et `qgsfeatureattribute.h`

Note: Si le cas précédent n'est pas assez claire, pour qu'un nom de fichier corresponde au nom d'une classe, il est implicite que chaque classe soit déclarée et implémentée par son propre fichier. Cela permet aux nouveaux développeurs d'identifier plus rapidement le code lié à une classe donnée.

1.4.2 En-tête standard et licence

Chaque fichier source doit contenir une en-tête calquée sur l'exemple qui suit:

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

Note: Il existe un modèle pour Qt Creator dans le dépôt Git. Pour l'utiliser, copiez-le depuis `doc/qt_creator_license_template` vers un emplacement local, adaptez l'adresse de courrier électronique et, si requis, le nom du développeur et configurez Qt Creator pour utiliser ce modèle: `Outils` -> `Options` -> `C++` -> `File Naming`.

1.5 Noms de variable

Les noms des variables locales commencent par une minuscule et sont formés en utilisant des minuscules et majuscules. Ne pas utiliser de préfixes comme « my » ou « the ».

Exemples:

- `mapCanvas`
- `currentExtent`

1.6 Énumérations

Les énumérations doivent être nommés en CamelCase avec une première lettre en majuscule, ex:

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

N'utilisez pas de noms génériques qui peuvent entrer en conflit avec d'autres types. Par exemple, utilisez `UnkownUnit` plutôt que `Unknown`

1.7 Constantes globales et Macros

Les constantes globales et les macros doivent être écrites en majuscules avec des séparateurs en tirets bas, ex:

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

1.8 Commentaires

Les commentaires des méthodes de classes devraient utiliser un style indicatif à la troisième personne plus qu'un style impératif:

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

1.9 Signaux Qt et emplacements

Toutes les connections de signaux ou emplacements doivent être fait en utilisant les connections « new style » disponible en Qt5. De plus amples informations sur cette exigence sont disponibles dans « QEP #77 <https://github.com/qgis/QGIS-Enhancement-Proposals/issues/77>‘_.

Évité d'utiliser l'emplacement de connexion automatique de Qt (c'est-à-dire ceux nommés ‘‘ void on_mSpinBox_valueChanged‘‘). Les emplacement à connexion automatique sont fragiles et sujets à des ruptures sans avertissement si les boîtes de dialogue sont remaniées.

1.10 Edition

N'importe quel éditeur ou EDI peut être utilisé pour éditer le code de QGIS, sous réserve qu'il respecte les pré-requis suivants:

1.10.1 Tabulations

Paramétrez votre éditeur pour remplacer les tabulations par des espaces. L'espacement d'une tabulation doit être paramétrée pour occuper deux espaces.

Note: Sous Vim, vous pouvez utiliser `set expandtab ts=2`

1.10.2 Indentation

Le code source doit être indenté pour améliorer la lisibilité. Le fichier `scripts/prepare-commit.sh` vérifie les fichiers modifiés et les ré-indente en utilisant l'utilitaire astyle. Il devrait être joué avant de commiter. Vous pouvez également utiliser le script `scripts/astyle.sh` pour indenter des fichiers individuellement.

Les nouvelles version de astyle indentent différemment que la version utilisée pour ré-indenter l'intégralité des sources, nous avons inclus une veille version du script astyle dans notre dépôt (activez la variable `WITH_ASTYLE` dans le cmake pour l'inclure dans votre build).

1.10.3 Accolades

Les accolades doivent commencer sur la ligne suivant l'expression:

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

1.11 Compatibilité API

Il y a: api: *Documentation de l'API* <> pour C ++.

Nous essayons de conserver l'API stable et rétrocompatible. Les remises à niveau de l'API doivent être réalisées d'une manière similaire au code source de Qt, par ex.

```
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 Liaisons SIP

Certain fichiers SIP sont automatiquement générés en utilisant un script dédié.

1.12.1 Pré-traitement d'en-tête

Toutes les informations pour construire proprement le fichier SIP doivent être présentes dans le fichier d'en-tête C++. Quelques macros sont disponibles pour ces définitions:

- Utilisez ‘‘ #ifdef SIP_RUN‘‘ pour générer du code uniquement dans les fichiers SIP ou ‘‘ #ifndef SIP_RUN‘‘ pour le code C ++ uniquement. Les instructions ‘‘ # else‘‘ sont gérées dans les deux cas.
- Utilisez `SIP_SKIP` pour sauter une ligne
- Les annotations suivante sont traitées:
  - `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/

- Les sections ‘‘ privées “” ne sont pas affichées, sauf si vous utilisez une instruction ‘‘ #ifdef SIP_RUN‘‘ dans ce bloc.
- SIP_PYDEFAULTVALUE(valeur) peut être utilisé pour définir une valeur par défaut alternative à la méthode par python. Si la valeur par défaut contient une virgule « , », la valeur doit être entourée par des simples guillements » “ « .
- SIP_PYTYPE(type) peut-être utilisé pour définir une type alternatif d'un argument à la méthode par python. Si le type contient une virgule « , », le type doit être entouré de simples guillemets » “ « .

Un fichier d'exemple est disponible dans tests/scripts/sipifyheader.h.

1.12.2 Générer le fichier SIP

Le fichier SIP peut être généré en utilisant un script dédié. Par exemple :

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

Dès qu'un fichier SIP est ajouté à l'un des fichiers source (: fichier: *python / core / core.sip*,: fichier:‘ python / gui / gui.sip‘ ou: file: *python / analysis / analysis .sip*), il sera considéré comme généré automatiquement. Un test sur Travis permettra de s'assurer que ce fichier est à jour avec son en-tête correspondant.

Les fichiers plus anciens pour lesquels la création automatique n'est pas encore activée sont répertoriés dans : fichier : *python / auto_sip.blacklist*.

1.12.3 Améliorer le script sipify

Si des améliorations sont nécessaire pour le script sipify, merci d'ajouter l'élément manquant au fichier de demonstration tests/scripts/sipifyheader.h et créer l'en-tête attendu tests/scripts/sipifyheader.expected.si. Ceci sera aussi automatiquement testé sur Travis en tant qu'une unité de test du script.

1.13 Style de code

Voici quelques trucs et astuces de programmation qui, nous l'espérons, vous aideront à réduire les erreurs, le temps de développement et la maintenance.

1.13.1 Lorsque c'est possible, utiliser du code générique

Si vous copiez-collez du code, ou si vous écrivez la même chose plusieurs fois, pensez à consolider le code en une seule fonction.

Cela permettra:

- Autorisez les changements à s'effectuer à un seul endroit plutôt qu'en de multiples emplacements.
- de prévenir le code pourri
- de rendre plus difficile l'évolution différenciée de plusieurs copies au fil du temps, phénomène qui rend plus difficile la compréhension et la maintenance pour les autres développeurs

1.13.2 Placez les constantes en premier dans les expressions

Placez les constantes en premier dans les expressions

`0 == value` à la place de `value == 0`

Cela empêchera les développeurs d'utiliser accidentellement = au lieu de ==, ce qui peut introduire des bogues logiques et subtils. Le compilateur générera une erreur si vous utilisez accidentellement = à la place de == pour les comparaisons puisque les constantes ne peuvent pas se voir assigner des valeurs.

1.13.3 Le caractère espace peut être votre ami

Ajouter des espaces entre les opérateurs, les instructions et les fonctions rendent le code plus lisible.

Ce qui est plus facile à lire, ceci:

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

ou ceci:

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

Note: :file:*scripts/prepare-commit.sh* On s'occupera de ceci.

1.13.4 Placer les commandes sur des lignes séparées

Lors de la lecture du code il est facile de rater des commandes, si elles ne sont pas en début de ligne. Lors d'une lecture rapide, il est courant de sauter des lignes si elles ne semblent pas correspondre avec ce que l'on cherche dans les premiers caractères. Il est aussi commun de s'attendre à une commande après un conditionnel comme `if`.

Considérez ceci:

```
if (foo) bar();

baz(); bar();
```

Il est très facile de rater des extraits dans le flux de contrôle. Au lieu de cela, utiliser

```
if (foo)
  bar();

baz();
bar();
```

1.13.5 Indenter les modificateurs d'accès

Les modificateurs d'accès structurent une classe en sections publique, protégée et privée de l'API. Les modificateurs d'accès eux-mêmes groupent le code dans cette structure. Indentez les modificateurs d'accès et les déclarations.

```
class QgsStructure
{
  public:
    /**
     * Constructor
     */
```

```
    explicit QgsStructure();
}
```

1.13.6 Recommandation de lecture

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

You should also really read this article from Qt Quarterly on designing Qt style (APIs)

1.14 Crédits pour les contributions

Les contributeurs aux nouvelles fonctionnalités sont encouragés à faire connaître aux gens leur contribution via:

- l'ajout d'une note au fichier de changement lors de la première incorporation du code auquel ils ont contribué, du type:

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

- Écrivant un article à propos de cette nouvelle fonctionnalité sur un blog, et en l'ajoutant à l'agrégateur https://plugins.qgis.org/planet/
- Ajout de leur nom à:
  - https://github.com/qgis/QGIS/blob/release-3_4/doc/{}'CONTRIBUTEURS
  - https://github.com/qgis/QGIS/blob/release-3_4/doc/{}'AUTEURS

CHAPTER 2

Bonnes pratiques pour l'IHM (Interface homme-machine)

Il est important de suivre les bonnes pratiques d'agencement et de design d'interfaces graphiques qui suivent, dans l'objectif d'amener de la consistance dans l'affichage des éléments d'interface ainsi que pour permettre aux utilisateurs d'utiliser instinctivement les boîtes de dialogue.

1. Regrouper les éléments liés entre eux en utilisant des boîtes de groupe: Essayer d'identifier les éléments qui peuvent être regroupés ensemble et utiliser ensuite des boîtes de groupe avec une étiquette permettant d'identifier le sujet du groupe. Éviter d'utiliser des boîtes de groupe avec un seul élément/contrôle à l'intérieur.
2. Mettre en majuscule uniquement la première lettre dans les étiquettes, les infobulles, les descriptions et tout texte qui n'est ni un titre ni une entête: elles doivent être rédigées sous forme d'expression avec la première lettre en majuscule. Les autres mots doivent être écrits avec la première lettre en minuscule, sauf si ce sont des noms.
3. Mettre en majuscule la première lettre de tous les mots contenus dans un titre (boîte de groupe, onglet, entête de vues…), les fonctions (éléments de menus, boutons) et tous les éléments sélectionnables (éléments de liste déroulante, de tableaux…): tous les mots hormis les prépositions de moins de 5 lettres (par exemple "with" mais "Without"), les conjonctions (par exemple, "and" ou "but") et les articles (a, an, the). Cependant, il faudra toujours mettre en majuscule la première lettre des premier et dernier mots.
4. Ne pas mettre de caractère « deux-points » à la fin des étiquettes, des widgets ou des boîtes de groupe: Ajouter le caractère « deux-points » perturbe la vision et n'apporte aucune signification supplémentaire; ne pas les utiliser. Il peut être fait exception à cette règle lorsque vous avez deux étiquettes qui se suivent; par exemple: Label1 Plugin (Path:) Label2 [/path/to/plugins]
5. Éloigner les actions dangereuses des actions sans incidence : Si vous avez des actions de "suppression" ou "d'effacement", etc. essayez d'imposer un espace adéquat entre ces actions « dangereuses » et les actions sans conséquences de manière à ce que les utilisateurs ne cliquent pas par inadvertance sur l'action dangereuse.
6. Utiliser toujours un QButtonBox pour les boutons "Ok", "Annuler", etc: Utiliser une boîte à boutons permet de s'assurer que l'ordre des boutons "Ok", "Annuler", etc. sera cohérent pas rapport au système d'exploitation, la langue, l'environnement du bureau utilisé par l'utilisateur final.
7. Les onglets ne doivent pas être imbriqués. Si vous utilisez les onglets, suivez le style des onglets de QgsVectorLayerProperties / QgsProjectProperties, etc., c'est à dire, des onglets en haut avec des icônes de 22x22.
8. Les empilements de widget doivent être évités le plus possible. Ils causent des problèmes de mise en page et des re-dimensionnements inexplicables (pour l'utilisateur) pour afficher les widgets non visibles.

9. Éviter d'utiliser des termes techniques et utiliser plutôt des équivalents simples, ex: utiliser le mot "Transparence" plutôt que "Canal Alpha", "Texte" à la place de "Chaîne de caractères", etc.
10. Utilisez une iconographie cohérente. Si vous avez besoin d'icône ou d'éléments d'icônes, merci de contacter Robert Szczepanek sur la liste de diffusion pour de l'assistance.
11. Placer les longues listes de contrôles dans des boîtes à défilement (scroll). Aucune boîte de dialogue ne doit mesurer plus de 580 pixels de haut et 1000 pixels de large.
12. Séparer les options avancées des fonctions basiques. Les utilisateurs novices doivent être capables d'accéder facilement aux éléments indispensables aux activités basiques sans être inquiétés par la complexité des fonctionnalités avancées. Les fonctionnalités avancées devraient être placées en dessous d'une ligne de séparation ou placées dans un onglet séparé.
13. Ne pas ajouter d'option dans le seul objectif d'avoir de nombreuses options. Lutter pour conserver l'interface utilisateur minimaliste et utiliser des options par défaut adaptées.
14. Si un bouton doit ouvrir une nouvelle boîte de dialogue, des points de suspension (…) doivent être ajoutés en suffixe du texte du bouton. Assurez-vous de bien utiliser le caractère « U+2026 » du point de suspension et non trois points consécutifs.

2.1 Auteurs

- Tim Sutton (auteur et éditeur)
- Gary Sherman
- Marco Hugentobler
- Matthias Kuhn

CHAPTER 3

Accès GIT

Ce chapitre décrit comment se lancer dans l'utilisation du dépôt GIT de QGIS. Avant de commencer, assurez-vous de disposer d'un client git installé sur votre système d'exploitation.

3.1 Installation

3.1.1 Installation de git pour GNU/Linux

Les utilisateurs d'une distribution Debian ou dérivée peuvent faire:

```
sudo apt install git
```

3.1.2 Installation de git sous Windows

Les utilisateurs de MS Windows peuvent utiliser msys git ou utiliser le client git distribué avec cygwin.

3.1.3 Installation de git sous macOS

Le projet git dispose d'un binaire téléchargeable de git. Assurez-vous de prendre le paquet conforme à votre processeur (x86_64 la plupart du temps, seuls les premiers Macs Intel utiliseront le paquet i386).

Une fois téléchargé, ouvrez l'image disque et lancez l'installeur.

Note pour l'architecture PPC/Source

Le site de git ne met pas à disposition des binaires pour PPC. Si vous en avez besoin ou si vous voulez plus de contrôle sur l'installation, vous devrez compiler git vous-même.

Téléchargez les sources depuis https://git-scm.com/. Décompressez-les dans un répertoire. Ouvrez un terminal pointant sur ce répertoire puis :

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

Si vous n'avez pas besoin des extensions, de Perl, de Python ou de TclTk (GUI), vous pouvez les désactiver avant de lancer make avec:

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

3.2 Accéder au Répertoire

Clôner la branche "master" de QGIS :

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

3.3 Basculer vers une branche

Pour basculer vers une branche (checkout), par exemple la branche de la version 2.6.1, faites:

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

Pour basculer sur la branche maitre:

```
cd QGIS
git checkout master
```

Note: Dans QGIS, nous conservons le code le plus stable dans la branche de la version publiée. La branche master contient le code pour la série de version appelée “non stable”. Périodiquement nous créons une branche à publier depuis la branche master et non continuons la stabilisation ainsi que l’incorporation sélective de nouvelles fonctionnalités dans la branche master.

Consultez le fichier INSTALL dans l’arbre des sources pour plus d’instruction sur la compilation des versions de développement.

3.4 Sources de la documentation de QGIS

Si vous voulez vérifier les sources de la documentation QGIS:

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

Vous pouvez également jeter un oeil au fichier Lisez-moi qui est inclus dans le dépôt de la documentation pour plus d’information.

3.5 Sources du site web QGIS

Si vous voulez vérifier les sources du site web de QGIS:

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

Vous pouvez également jeter un oeil au fichier Lisez-moi qui est inclus dans le dépôt du site web pour plus d’information.

3.6 Documentation GIT

Consultez les sites suivants pour plus d’information sur Git.

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

3.7 Le développement par branches

3.7.1 Objectif

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 Procédure

- **Annonce initiale sur la liste de diffusion :** Avant de commencer, faites une annonce sur la liste de diffusion des développeurs pour voir si personne d'autre que vous ne travaille déjà sur la même fonctionnalité. Prenez également contact avec le conseiller technique du comité de direction du projet (PSC). Si la nouvelle fonctionnalité impose des changements d'architecture dans QGIS, un avis (RFC) est obligatoire.

Créer une branche : créer une nouvelle branche GIT pour le développement d'une nouvelle fonctionnalité.

```
git checkout -b newfeature
```

Vous pouvez maintenant commencer le développement. Si vous pensez travailler intensément sur cette branche et que vous voulez partager ce travail avec d'autres développeurs et avoir accès en écriture au dépôt amont, vous pouvez pousser votre dépôt dans le dépôt QGIS officiel par:

```
git push origin newfeature
```

Note: Si la branche existe déjà, les modifications seront ajoutées dans celle-ci.

Rebaser régulièrement vers la branche master: il est recommandé de réaliser un rebasage pour incorporer les changements de la branche master vers la branche courante, de manière régulière. Cela permet de faciliter la fusion ultérieure de la branche courante vers la branche master. Après un rebasage, vous devez lancer `git push -f` dans votre dépôt dupliqué.

Note: Ne faites jamais `git push -f` sur le dépôt d'origine! Ne l'utilisez que dans votre propre branche de production.

```
git rebase master
```

3.7.3 Tester avant de fusionner avec la branche master

Lorsque vous avez terminé avec la nouvelle fonctionnalité et êtes satisfait de sa stabilité, faites une annonce sur la liste des développeurs. Avant la fusion, les modifications seront testées par les développeurs et les utilisateurs.

3.8 Envoi de correctifs et de demandes

Il y a quelques règles qui vous aideront à obtenir vos correctifs et à extraire facilement les demandes dans QGIS, et à nous aider à traiter les correctifs envoyés plus facilement.

3.8.1 Envoi des demandes

Il est en général plus facile pour les développeurs que vous soumettiez des pull requests GitHub. Nous ne décrirons pas le mécanisme de pull requests ici mais vous pouvez vous référer à la documentation GitHub sur les pull requests.

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 vous êtes un développeur et que vous voulez évaluer la file de pull requests, il existe un outil simple qui vous permettra de le faire en ligne de commande.

Merci de consulter le chapitre ci-dessous sur comment "notifier votre patch". En général, lorsque vous soumettez une PR, vous devrez prendre la responsabilité de la suivre au long de son intégration, en répondant aux questions

posées par les autres développeurs, trouver un "champion" pour cette fonctionnalité et envoyer un courtois rappel si vous constatez que votre PR n'attire pas trop l'attention. Merci de garder à l'esprit que QGIS est un projet conduit par des volontaires et qu'il est probable que votre PR n'attire pas l'attention immédiatement. Si vous pensez que la PR ne reçoit pas l'attention qu'elle mérite, voici vos options pour accélérer son intégration (par ordre de priorité):

- Envoyez un message à la liste de diffusion à propos de votre PR pour nous dire combien il est important qu'elle puisse être intégrée au code principal.
- Envoyer un message à la personne à qui est attribuée la PR dans la liste.
- Envoyer un message à Marco Hugentobler (qui gère la file d'attente des PR)
- Envoyez un message au comité de direction du projet en leur demandant assistance pour incorporer votre PR au code principal.

Bonnes pratiques pour la création de pull request

- Commencez toujours une nouvelle branche pour une fonctionnalité à partir de la branche master actuelle.
- Si vous développez une branche de nouvelle fonctionnalité, ne fusionnez rien dans cette branche. A la place, effectuez un rebasement (rebase) comme décrit dans le prochain point, de manière à conserver un historique propre.
- Avant de créer une pull request, lancez `git fetch origin` et `git rebase origin/master` (origin étant ici le dépôt distant amont et non votre propre dépôt, vérifiez votre fichier `.git/config` ou faites: `git remote -v | grep github.com/qgis` pour identifier le nom utilisé dans votre configuration).
- Vous pouvez faire un *git rebase* comme dans la ligne précédente de manière répétée sans dommage (du moment que l'objectif de votre branche est d'être fusionnée dans la branche master).
- Attention, après une opération de rebasement, vous devrez faire un `git push -f` vers votre dépôt forké. **DÉVELOPPEURS PRINCIPAUX: NE FAITES PAS CELA DANS LE DEPÔT QGIS PUBLIC !**

Notifications destinées à la documentation

Outre les tags habituels pour classer votre PR, il existe des tags spéciaux permettant de générer automatiquement des tickets dans le dépôt de la documentation dès lors que votre PR est accepté.

- `[needs-docs]` permet aux rédacteurs d'identifier des correctifs ou des améliorations apportées à une fonctionnalité déjà existante.
- `[feature]` in case of new functionality. Filling a good description in your PR will be a good start.

Les développeurs sont donc priés de bien vouloir ajouter ces tags (insensibles à la casse) afin de faciliter la gestion des tickets pour la documentation mais aussi pour l'aperçu global des modifications liées à la version. Mais, veuillez s'il vous plaît prendre le temps d'ajouter quelques commentaires: soit dans le commit, soit dans la documentation elle-même.

Pour fusionner une pull request

Option A:

- cliquez sur le bouton merge (crée une fusion sans avance rapide)

Option B:

- Vérifiez une pull request
- Test (également requis pour l'option A évidemment)
- checkout master, git merge pr/1234

- En option: `git pull --rebase`: Crée une avance rapide, aucun commit de fusion n'est réalisé. Meilleur historique mais il est plus difficile de revenir en arrière.
- `git push` (NE JAMAIS utiliser l'option -f ici)

3.9 Nom des fichiers de patch

Si le patch est une correction pour un bug donné, merci de nommer le fichier avec le numéro de bug dedans, ex: bug777fix.patch et d'ajouter ce fichier dans le rapport de bug originel.

Si le bogue est une amélioration ou une nouvelle fonctionnalité, il est généralement bon de créer d'abord un ticket dans GitHub et d'y attacher le patch.

3.10 Créez votre patch à la racine du répertoire des sources QGIS

Cela permet d'appliquer le patch plus facilement étant donné que nous n'aurons pas besoin de naviguer dans un emplacement spécifique des sources. De plus, lorsque je reçois des patchs, je les inspecte en utilisant merge et avoir le patch à la racine du répertoire des sources est bien plus facile. Ci-dessous, voici un exemple pour inclure plusieurs changements de fichiers dans votre patch à partir de la racine des sources:

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

Cela permettra de vous assurer que la branche master est synchronisée avec la branche du dépôt amont et cela générera un patch contenant le delta entre votre branche de nouvelle fonctionnalité et ce qui se trouve dans la branche master.

3.10.1 Faire en sorte que votre patch soit remarqué

Les développeurs QGIS sont très occupés. Nous effectuons des recherches sur les patchs soumis dans les rapports de bug mais nous oublions parfois certaines choses. N'en prenez pas offense et ne vous alarmez pas. Essayez d'identifier les développeurs qui pourront vous aider et contactez-les pour leur demander de jeter un œil à votre patch. Si vous n'avez pas de réponse, vous pouvez remonter votre demande à l'un des membres du Comité de Direction du Projet (PSC, les détails de contacts sont également disponibles dans les Ressources Techniques).

3.10.2 Vérifications nécessaires

QGIS est sous licence GPL. Vous devez vous assurer que vous soumettez des patchs non encombrés de problème de propriété intellectuelle. Ne soumettez pas de code non disponible sous licence GPL.

3.11 Obtenir les droits d'écriture dans Git

L'accès en écriture à l'arbre des sources QGIS se fait par invitation. Généralement, lorsqu'une personne soumet plusieurs patchs conséquents (sans nombre fixe de participation) qui démontre de solides compétences et compréhension du C++ et des conventions de code QGIS, un des membres du PSC ou d'autres développeurs QGIS peuvent proposer au PSC de lui fournir les droits d'écriture. La personne qui recommande le nouveau venu doit rédiger un paragraphe promotionnel pour expliquer pourquoi il pense que la personne citée doit obtenir les droits d'écriture. Dans certains cas, nous donnerons accès à des développeurs non C++ comme des traducteurs et des personnes en charge de la documentation. Dans ce cas, la personne doit avoir fait la preuve de son habileté à

proposer des patchs et devrait avoir soumis plusieurs patchs démontrant sa compréhension de la modification de la base de code, de manière propre, sans rien casser, etc.

Note: Depuis le passage à Git, nous sommes moins enclins à fournir des droits en écriture aux nouveaux développeurs car il est maintenant trivial de partager du code sous GitHub en forkant QGIS et en proposant des pull requests.

Merci de vérifier que tout se compile correctement avant de créer des commits ou des pull requests. Essayez de rester attentif aux possibles problèmes que vos commits peuvent générer pour les développeurs compilant sur d'autres plateformes ou avec des versions plus ou moins récentes des différentes bibliothèques.

Lorsque vous faites un commit, votre éditeur de texte (défini dans la variable d'environnement $EDITOR) apparaîtra et vous devriez écrire un commentaire au début du fichier (au dessus de la partie qui indique "ne modifiez pas ceci"). Inscrivez un commentaire descriptif et faites plutôt plusieurs petits commits si vous effectuez des changements sur des fichiers qui ne sont pas liés entre eux. Inversement, nous préférerions que vous regroupiez les changements liés entre eux dans un seul commit.

CHAPTER 4

Découvrir et aller plus loin avec QtCreator et QGIS

QtCreator est un EDI développé par les concepteurs de la bibliothèque Qt. Avec QtCreator, vous pouvez construire n'importe quel projet C++, mais l'EDI est vraiment optimisé pour les personnes qui travaillent sur des applications basées sur Qt(4) (y compris les applications mobiles). Tout ce est décrit ci-dessous présume que vous travaillez avec Ubuntu 11.04 "Natty".

4.1 Installation de QtCreator

Cette partie est simple:

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

Une fois l'installation terminée, vous le trouverez dans votre menu GNOME.

4.2 Paramétrer votre projet

Je suppose que vous avez déjà un clone local de QGIS contenant le code source et que vous avez installé et compilé toutes les dépendances nécessaires, etc. Vous trouverez des instructions détaillées sous *accès git* et installation de dépendances.

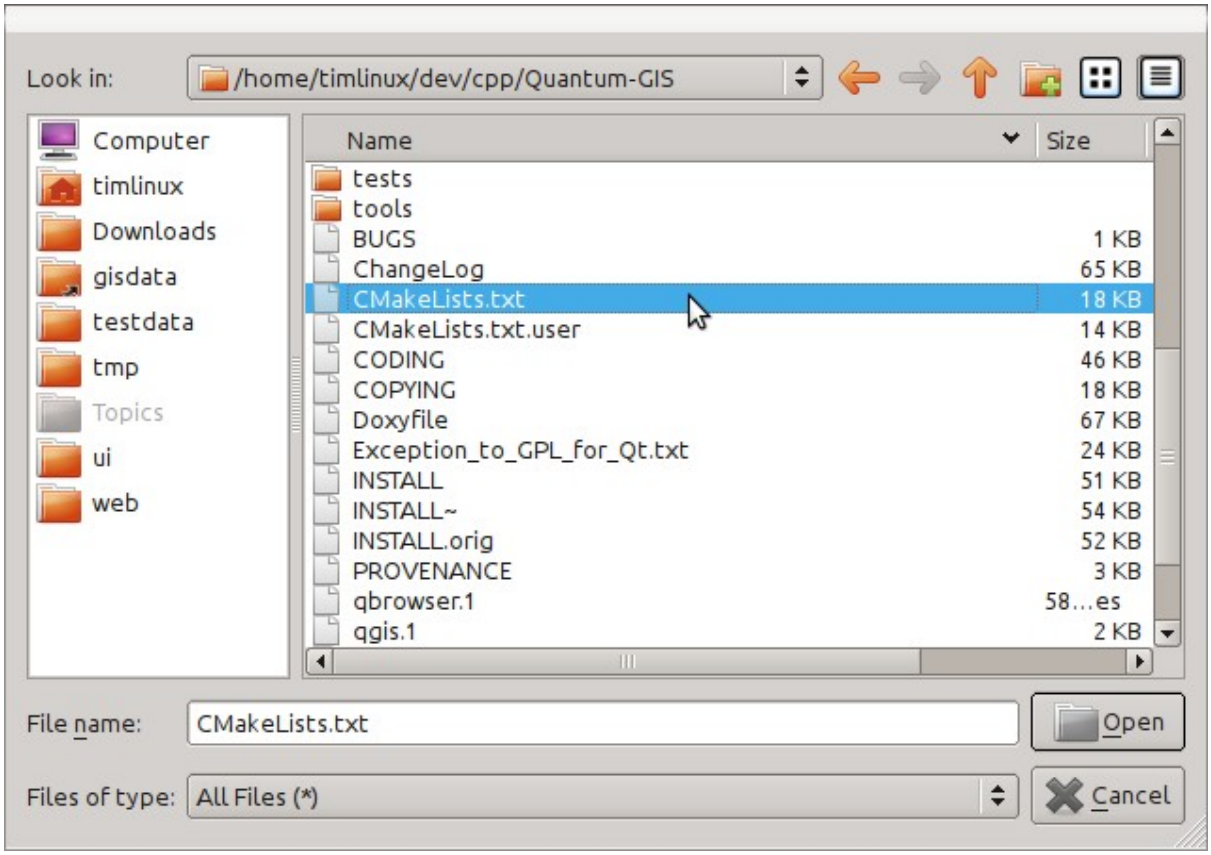
Sur mon système j'ai placé le code dans `$HOME/dev/cpp/QGIS` et le reste de cet article a été rédigé en présumant de cet emplacement pour ce répertoire. Vous pouvez mettre à jour ces chemins pour les adapter à votre système local.

Lors du lancement de QtCreator, faire:

Fichier -> Ouvrir un Fichier ou Projet

Utilisez ensuite la boîte de dialogue de sélection de fichier pour naviguer et ouvrir ce fichier:

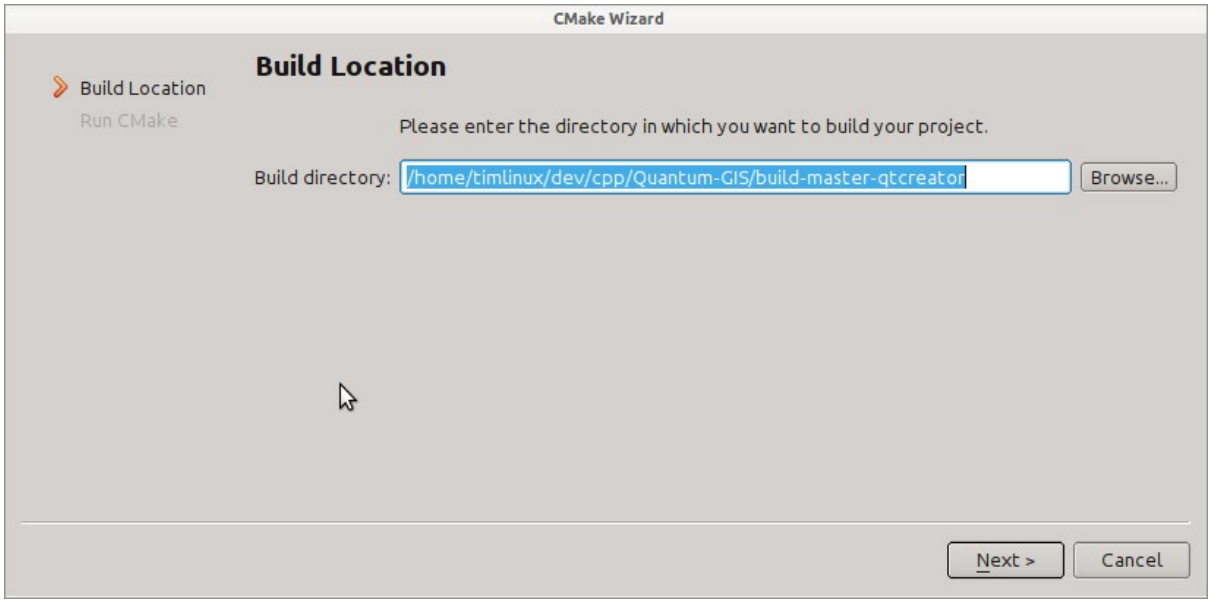
```
$HOME/dev/cpp/QGIS/CMakeLists.txt
```



Ensuite, on vous demandera l'emplacement d'un répertoire de compilation. Je créé un répertoire dédié au travail de QtCreator:

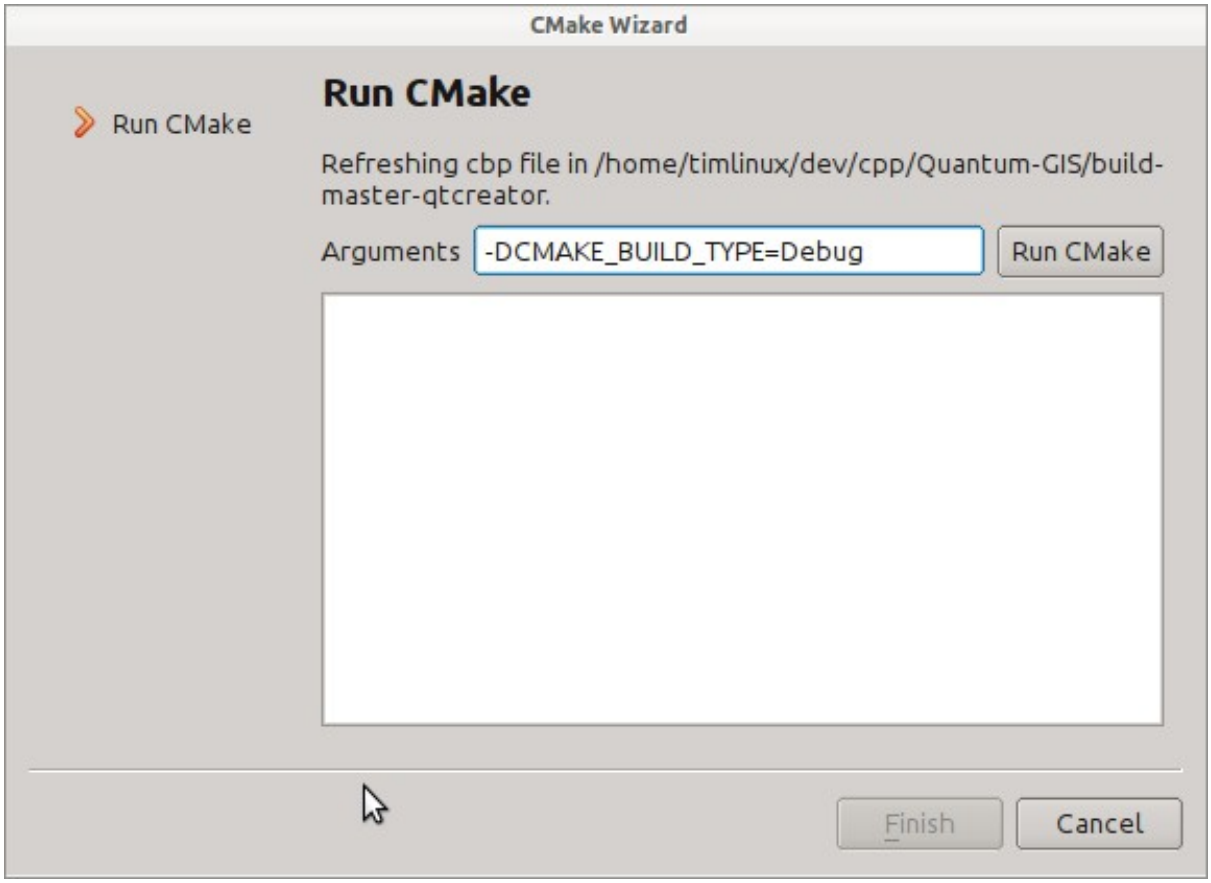
```
$HOME/dev/cpp/QGIS/build-master-qtcreator
```

Il est généralement bon de séparer les répertoires de compilation par branches différentes, si vous pouvez vous le permettre par rapport à l'occupation disque.



Ensuite, on vous demandera si vous avez une ou plusieurs options de compilation CMake à transmettre à CMake. Nous allons indiquer à CMake que nous voulons une compilation en mode déboguage en ajoutant l'option suivante:

```
-DCMAKE_BUILD_TYPE=Debug
```

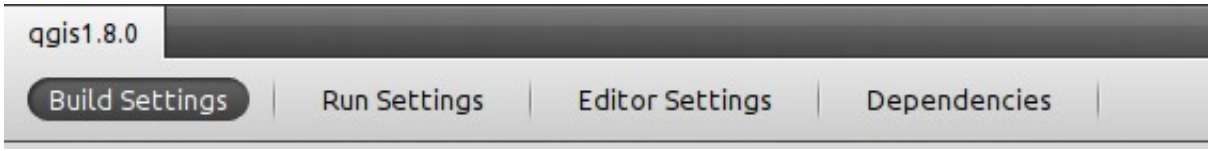


Voilà la base. Une fois que vous avez fermé l'assistant, QtCreator lancera une recherche dans l'arborescence du code source pour la gestion de l'autocomplétion et pour d'autres opérations, en tâche de fond. Avant de lancer la compilation, nous voulons encore paramétrer finement certains éléments.

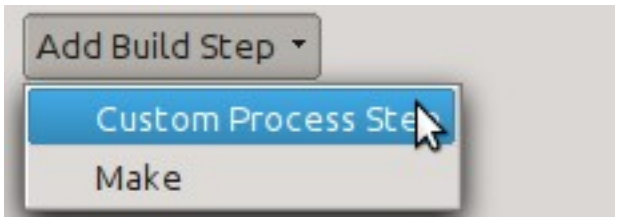
4.3 Paramétrez votre environnement de compilation

Cliquez sur l'icône "Projets" à la gauche de la fenêtre QtCreator.

Choisissez l'onglet de paramètres de compilation (normalement, actif par défaut).



Nous allons maintenant ajouter une étape supplémentaire. Pourquoi ? Comme QGIS ne peut actuellement être lancé qu'à partir d'un répertoire d'installation et non depuis le répertoire de construction, il est indispensable de s'assurer que QGIS est installé lorsqu'il est compilé. Sous "Étapes de compilation", cliquez sur le bouton "Ajouter l'étape compiler" et choisissez "Étape personnalisée".



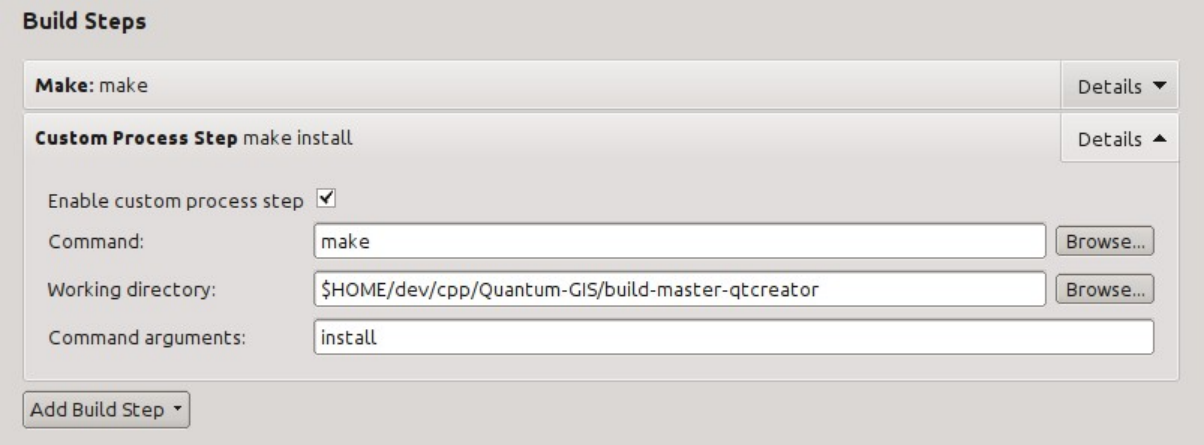
Maintenant, nous paramétrons les détails suivants:

Autoriser une étape personnalisée: [oui]

Commande: faire

Répertoire de travail: $HOME/dev/cpp/QGIS/build-master-qtcreator

Arguments de la commande: install



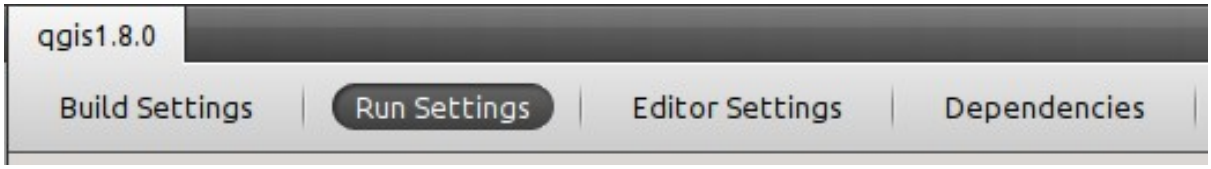
Vous êtes pratiquement prêts à compiler. Une dernière note: QtCreator a besoin des droits d'écriture sur le répertoire d'installation. Par défaut (ce que j'utilise ici), QGIS sera installé dans `/usr/local`. Pour mes besoins sur ma machine de développement, je me suis simplement donné des droits d'écriture dans le répertoire /usr/local.

Pour commencer la compilation, cliquez sur l'icône en forme de gros marteau dans le coin inférieur gauche de la fenêtre.

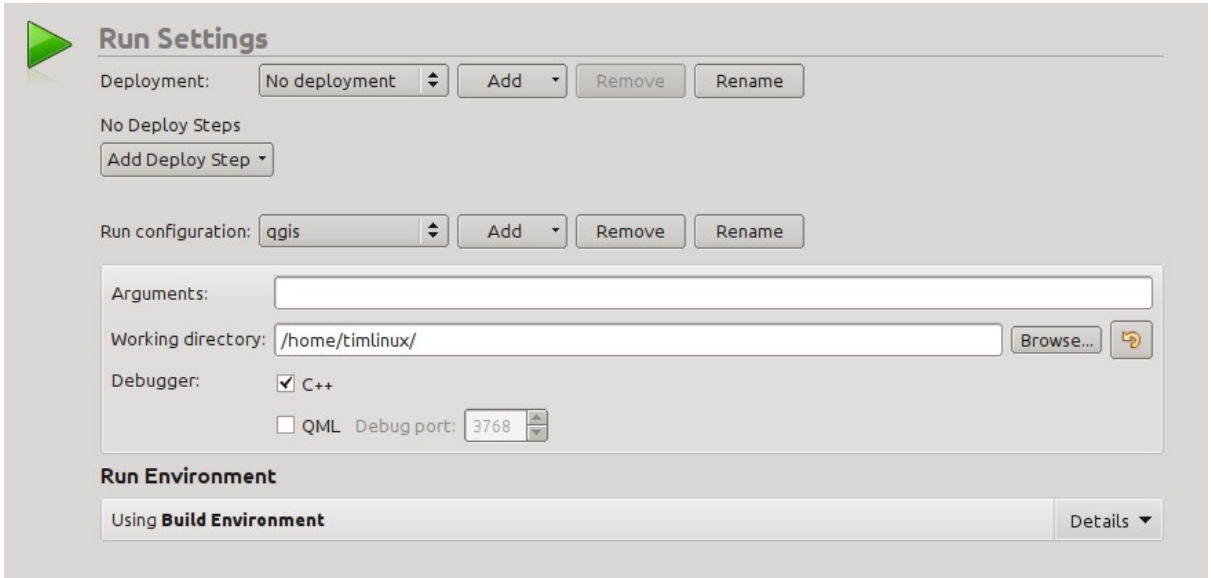
4.4 Paramétrez votre environnement de lancement

Comme mentionné ci-dessus, nous ne pouvons pas lancer QGIS directement depuis le répertoire de construction et nous devons donc créer une cible de lancement pour indiquer à QtCreator de lancer QGIS à partir du répertoire d'installation (dans mon cas `/usr/local/`). Pour cela, retournez dans l'écran de configuration des projets.

Sélectionnez maintenant l'onglet "Paramètres d'exécution".



Nous devons mettre à jour les paramètres d'exécution par défaut en remplaçant la configuration d'exécution "qgis" par une commande personnalisée.



Pour cela, cliquez sur le bouton "Ajouter v" à côté de la liste déroulante "Configuration d'exécution" et choisissez "Exécutable personnalisé" à partir du haut de la liste.

Maintenant, indiquez les éléments suivants dans les propriétés:

Exécutable: /usr/local/bin/qgis

Arguments:

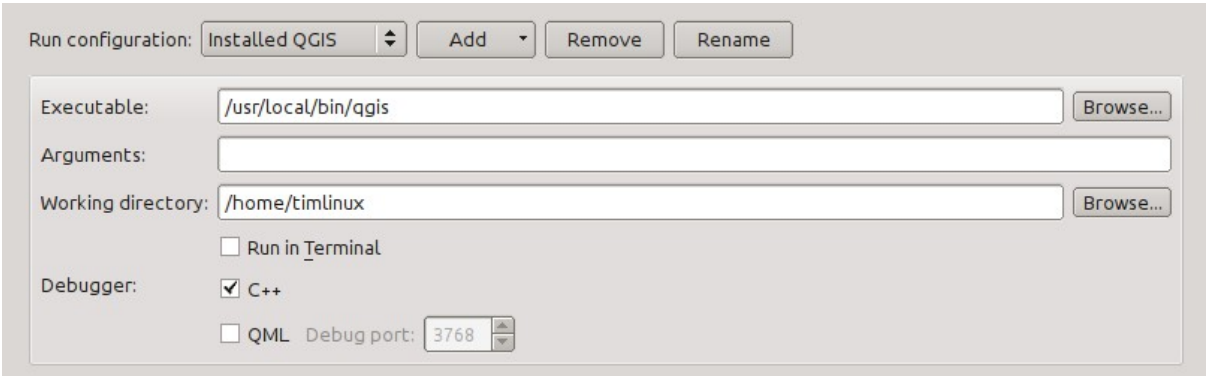
Répertoire de travail: $HOME

Lancer dans un terminal: [non]

Débogueur: C++ [oui]

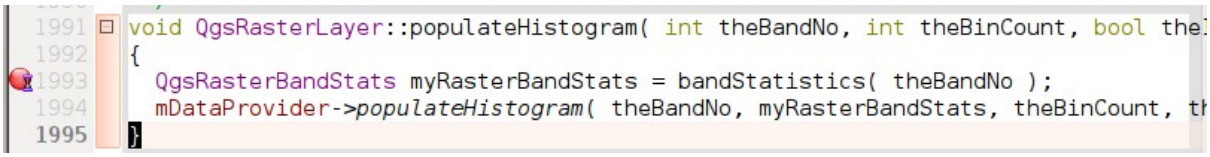
Qml [no]

Cliquez ensuite sur le bouton “Renommer” et donnez à votre exécutable personnalisé un nom significatif, ex: “QGIS installé”.

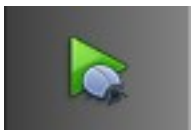


4.5 Exécution et déboguage

Maintenant, vous êtes prêts à lancer et à déboguer QGIS. Pour insérer un point d’arrêt, ouvrez simplement un fichier source et cliquez dans la colonne de gauche.



Maintenant, lancez QGIS dans le débogueur en cliquant sur l’icône avec un bogue dessus dans le coin en bas à gauche de la fenêtre.



CHAPTER 5

Tests unitaires

À partir de novembre 2007 nous exigeons que les nouvelles fonctionnalités de la branche master soient accompagnées d'un test unitaire. Nous avions initialement limité cette exigence à la partie qgis_core et nous allons étendre ce point aux autres parties du code une fois que les développeurs se seront familiarisés avec les procédures des tests unitaires, expliquées dans les sections qui suivent.

5.1 L'environnement de test de QGIS: un aperçu

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

1. There is some code you want to test, e.g. a class or function. Extreme programming advocates suggest that the code should not even be written yet when you start building your tests, and then as you implement your code you can immediately validate each new functional part you add with your test. In practice you will probably need to write tests for pre-existing code in QGIS since we are starting with a testing framework well after much application logic has already been implemented.

2. Créez un test unitaire. Tout se passe dans `/tests/src/core` dans le cas d'une bibliothèque du coeur du projet. Le test est essentiellement un client qui créé une instance de la classe et l'appelle avec des méthodes de classe. Cela permet de vérifier que le retour de chaque méthode renvoie bien la valeur attendue. Si un seul des appels échoue, le test sera également en échec.
3. Incluez les macros QTestLib dans votre classe de test. Cette macro est prise en compte par le compilateur de méta-objets Qt (moc) et elle transformera votre classe de test en application exécutable.
4. Ajoutez une section au fichier CMakeLists.txt dans le répertoire des tests qui construira votre test.
5. Assurez-vous d'avoir la variable `ENABLE_TESTING` activée dans ccmake / cmakesetup. Cela permettra de s'assurer que vos tests seront compilés lorsque vous lancerez make.
6. You optionally add test data to `<QGIS Source Dir>/tests/testdata` if your test is data driven (e.g. needs to load a shapefile). These test data should be as small as possible and wherever possible you should use the existing datasets already there. Your tests should never modify this data in situ, but rather make a temporary copy somewhere if needed.
7. Compilez vos sources et installez. Vous pouvez le faire avec le traditionnel `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.

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

5.2 Créer un test unitaire

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

```
/***************************************************************************
 testqgsvectorfilewriter.cpp
 ---------------------------------------
  Date : Friday, Jan 27, 2015
  Copyright: (C) 2015 by Tim Sutton
  Email: tim@kartoza.com
 ***************************************************************************
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 ***************************************************************************/
```

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

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

Ensuite, vous pouvez implémenter votre classe normalement en ajoutant les en-têtes dont vous pourrez avoir besoin:

```
//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>
```

Etant donné que nous combinons, dans un seul fichier, la déclaration et l'implémentation de la classe, nous ajoutons ensuite la déclaration de la classe. Nous ajoutons alors la documentation doxygen. Chaque test doit être correctement documenté. Nous utilisons la directive doxygen ingroup de manière à ce que tous les tests unitaires apparaissent dans un seul module dans la documentation générée par Doxygen. Vient ensuite une description résumée du test unitaire, la classe doit hériter de QObject et inclure la macro Q_OBJECT.

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

class TestQgsRasterLayer: public QObject
{
    Q_OBJECT
```

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

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

Then come your test methods, all of which should take no parameters and should return void. The methods will be called in order of declaration. We are implementing two methods here which illustrate two types of testing.

In the first case we want to generally test if the various parts of the class are working, We can use a functional testing approach. Once again, extreme programmers would advocate writing these tests before implementing the class. Then as you work your way through your class implementation you iteratively run your unit tests. More and more test functions should complete successfully as your class implementation work progresses, and when the whole unit test passes, your new class is done and is now complete with a repeatable way to validate it.

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

```
//
// Functional Testing
//

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

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

5.2.1 Implementing a regression test

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

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

   Better yet, before committing any changes in the future, running our tests will ensure our changes don't have unexpected side effects - like breaking existing functionality.

Il existe également une autre avancée offerte par les tests de non-régression: ils peuvent vous permettre de gagner du temps. Si vous avez déjà corrigé un bogue qui implique du changement de code et que vous avez lancé l'application et réalisé une série de tests manuels pour répliquer le problème, vous pourrez comprendre facilement que l'implémentation d'un test de non-régression avant la correction du bogue vous permettra d'automatiser cette correction de manière efficace.

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;
};
```

Ceci termine la déclaration de notre classe. L'implémentation est simplement incluse dans le même fichier plus bas. D'abord la fonction d'initialisation puis la fonction de nettoyage:

```
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;
}
```

La fonction d'initialisation ci-dessus illustre quelques points d'intérêt.

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 fournit également d'autres mécanismes d'intéressants pour les tests sur les données. Si vous désirez en savoir davantage sur le sujet, consultez la documentation Qt.

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*)

Certaines de ces macros sont utiles uniquement lorsque vous utilisez le cadriciel Qt pour les tests sur les données (consultez la documentation Qt pour plus de détails).

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

Normalement, vos tests fonctionnels devraient couvrir la totalité des fonctionnalités de vos classes publiques d'API, lorsque c'est possible. Maintenant que nos tests fonctionnels sont couverts, nous pouvons nous intéresser à notre exemple de test de non-régression.

Étant donné que le bogue #832 concerne un décompte de cellules incorrect, l'écriture de notre test est simplement une question d'utilisation de QVERIFY pour vérifier que le décompte des cellules correspond bien à la valeur attendue:

```
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 Comparer des images pour tests de rendu

Les images rendues sur des environnements différents peuvent produire de légères différences du aux implémentations spécifiques aux plateformes (ex : rendus de polices de caractères et algorithmes antialiasing différents), du aux polices disponibles sur le système et du à d'autres raisons obscures.

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.

Par exemple

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

Vous pouvez raccourcir le chemin du fichier vers l'image de référence en faisant passé à la place une partie du nom de test, ex :

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

(Ce raccourci fonctionne seulement si une seule image de référence correspondante est trouvée. Si plusieurs correspondances sont trouvées vous aurez besoin de fournir le chemin complet vers l'image de référence.)

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 Ajouter votre test unitaire à 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 La macro ADD_QGIS_TEST expliquée

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})
```

Etant donné que notre classe doit être lancée à travers le compilateur de méta-objet Qt (cmo), nous devons fournir quelques lignes en plus pour déclencher ce comportement:

```
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 Compiler votre test unitaire

Pour compiler notre test unitaire, vous devez vous assurer que `ENABLE_TESTS=true` est dans la configuration de CMake. Il existe deux moyens pour y parvenir:

1. Lancez `cmake ..` (ou `cmakesetup ..` sous MS-Windows) et positionnez interactivement l'option `ENABLE_TESTS` à `ON`.
2. Ajoutez une option à la ligne de commande de cmake; ex: `cmake -DENABLE_TESTS=true ..`

À part cela, compilez QGIS comme d'habitude et les tests devraient également se compiler.

5.6 Lancer vos tests

Le moyen le plus simple de lancer les tests est de les inclure directement dans le processus de compilation:

```
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
```

Si un test échoue, vous pouvez utiliser la commande ctest pour examiner plus en détails pourquoi il a échoué. Utilisez l'option `-R` pour indiquer une expression rationnelle pour désigner les tests que vous voulez lancer et `-V` pour activer la sortie verbeuse.

```
$ 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	Valeur
PYTHONPATH	[build]/output/python/:[build]/output/python/plugins:[source]/tests/src/python
QGIS_PREFIX_PATH	[build]/output
LD_LIBRARY_PATH	[build]/output/lib

Remplacez `[build]` par votre répertoire de compilation et `[source]` par le répertoire source.

5.6.2 Amusez-vous

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

CHAPTER 6

Processing Algorithms Testing

6.1 Algorithm tests

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

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

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

This file is structured with yaml syntax.

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

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

6.1.1 How To

To add a new test please follow these steps:

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

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

The above translates to

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

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

6.1.2 Paramètres et résultats

Trivial type parameters

Parameters and results are specified as lists or dictionaries:

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

ou

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

Layer type parameters

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

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

This is what it looks like in action:

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

File type parameters

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

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

Résultats

Results are specified very similarly.

Basic vector files

Ce ne pourrait être plus trivial!

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

Add the expected GML and XSD files in the folder.

Vector with tolerance

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

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

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

Fichiers raster

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

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

Fichiers

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

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

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

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

Répertoires

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

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

6.1.3 Algorithm Context

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

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

6.1.4 Exécuter localement des tests

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

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

CHAPTER 7

Tests de conformité OGC

L'Open Geospatial Consortium (OGC) fournit gratuitement des tests pour s'assurer de la conformité d'un serveur à certaines spécifications. Ce chapitre fournit un guide pas-à-pas rapide pour l'installation de tests WMS sur un système Ubuntu. Une documentation détaillée est disponible sur le site internet de l'OGC.

7.1 Installation des tests de conformité WMS 1.3 et 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
```

Télécharger et installer le test WMS 1.3.0

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

Télécharger et installer le test WMS 1.1.1

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

7.2 Projet test

Pour les tests WMS, les données doivent être téléchargées et chargées dans un projet 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 Lancer le test 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 Lancer le test 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
```