QGIS Developers Guide

Version 2.18

QGIS Project

08 April 2019

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 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

Une classe dans QGIS commence avec Qgs et est nommé en utilisant la casse camel.

Exemples:

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

1.1.2 Membres

Les noms des membres de classe commencent par un minuscule m 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 public. Les membres protégés doivent être évités pour que les membres soient accessibles depuis des sous-classes Python, alors que les membres protégés ne peuvent pas être utilisés à partir des liaisons Python.

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

- `sRefCounter`
- `DEFAULT_QUEUE_SIZE`

1.1.3 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. Il est recommandé de choisir un nom de fonction en rapport avec 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 enveloppés de casse de chameau pour la cohérence. Par exemple. ‘‘ setXml‘‘ au lieu de ‘‘ setXML‘‘.

1.1.5 Arguments de la fonction

Function arguments should use descriptive names. Do not use single letter argments (e.g. `setColor( const QColor& color )` instead of `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 (par exemple ‘‘ setTitle (const QString & title) ‘‘ au lieu de ‘‘ setTitle (titre QString) ‘‘.

1.1.6 Valeurs de retour de la fonction

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

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

1.2 Qt Designer

1.2.1 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`

1.2.2 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 gadgets 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 gadgets est bien mis à jour à chaque fois que la boîte de dialogue de la couche change.

1.3 Fichiers C++

1.3.1 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.

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

1.3.2 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`.

1.4 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`

1.5 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`

1.6 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;
```

1.7 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.

1.8 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:

1.8.1 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`

1.8.2 Indentation

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

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

1.8.3 Accolades

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

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

1.9 Compatibilité API

There is API documentation for 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 be deprecated, you are encouraged to
     *  connect to somethingHappenedBetter() rather.
     * @deprecated use somethingHappenedBetter()
     */
#ifndef Q_MOC_RUN
    Q_DECL_DEPRECATED
#endif
    bool somethingHappened();

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

1.10 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.

1.10.1 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

1.10.2 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.

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

1.10.4 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();
```

1.10.5 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();
}
```

1.10.6 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)

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

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

CHAPTER 2

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.

2.1 Installation

2.1.1 Installation de git pour GNU/Linux

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

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

2.1.2 Installation de git sous Windows

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

2.1.3 Installation de git sous macOS

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

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.

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

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

Si 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=
```

2.2 Accéder au Répertoire

Clôner la branche 'master' de QGIS :

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

2.3 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.

2.4 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.

2.5 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.

2.6 Documentation Git

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

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

2.7 Le développement par branches

2.7.1 Objectif

La complexité du code source de QGIS s'est considérablement accrue ces dernières années. Dès lors, il est difficile d'anticiper les effets de bord induits par l'ajout de fonctionnalités. Dans le passé, le projet QGIS avait de très long cycle de publication du fait du lourd travail à effectuer pour rétablir la stabilité du logiciel après l'ajout de nouvelles fonctionnalités. Pour dépasser ce problème, QGIS a basculé sur un modèle de développement où les nouvelles fonctionnalités sont d'abord programmées dans des branches GIT, puis fusionnées avec la branche principale ('master') lorsqu'elles sont finalisées et stables. Cette section décrit la procédure pour créer et fusionner des branches dans le projet QGIS.

2.7.2 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
```

2.7.3 Documentation on wiki

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

2.7.4 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.

2.8 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.

2.8.1 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.

Si vous avez créé une pull request, nous vous demandons de faire régulièrement une fusion de master vers la branche de votre PR de manière à ce que cette dernière puisse être toujours fusionnable directement avec la branche master.

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

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]` dans le cas où une nouvelle fonctionnalité est introduite. Fournir une bonne description de vos modifications est aussi vivement conseillé/apprécié.

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)

2.9 Nom des fichiers de patch

If the patch is a fix for a specific bug, please name the file with the bug number in it e.g. bug777fix.patch, and attach it to the original bug report in trac.

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

2.10 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.

2.10.1 Faire en sorte que votre patch soit remarqué

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

2.10.2 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.

2.11 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 3

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.

3.1 L'environnement de test de QGIS: un aperçu

Les tests unitaires sont utilisés en combinant QTestLib (la bibliothèque de test de Qt) et CTest (l'environnement de compilation et de tests faisant partie du processus de construction CMake). Voici un aperçu de ce processus avant de rentrer dans les détails:

- Prenons du code que vous voulez tester, par exemple, une classe ou une fonction. Les partisans de la programmation Extreme suggèrent que l'écriture du code ne commence pas tant que vous n'avez pas lancé la construction des tests. Ainsi, une fois que vous avez commencé à implémenter votre code, vous pouvez immédiatement valider chaque nouvelle partie fonctionnelle avec vos tests. Dans la pratique, vous devrez sans doute écrire des tests pour des parties déjà en place dans QGIS car nous avons lancé l'environnement de test bien après que la logique de l'application ait été déjà implémentée.
- 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.
- 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.
- Ajoutez une section au fichier CMakeLists.txt dans le répertoire des tests qui construira votre test.
- 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.
- Vous pouvez ajouter optionnellement des données de test dans le répertoire `/tests/testdata` si vos tests ont besoin de données externes (ex: besoin d'accéder à un fichier Shape). Ces données de test doivent être les plus compactes possibles et elles doivent, dans la mesure du possible, utiliser les jeux de données déjà présents. Vos tests ne doivent jamais modifier directement ces données mais plutôt faire une copie temporaire dans un répertoire en cas de besoin.

- Compilez vos sources et installez. Vous pouvez le faire avec le traditionnel `make && (sudo) make install`.
- Lancez les tests. Vous pouvez le faire simplement avec la commande `make test` après avoir fait un `make install` même si d'autres approches donnant plus de contrôle sur le déroulement des tests sont décrites plus loin.

Maintenant que nous venons de voir comment faire, je vais rentrer un peu plus dans les détails. J'ai déjà réalisé une grande partie de la configuration dans CMake et dans les autres parties des fichiers sources de manière à ce que vous puissiez vous concentrer sur la partie facile: écrire des tests unitaires !

3.1.1 Créer un test unitaire

Créer un test unitaire est facile. Généralement, vous pouvez en créer un en écrivant un simple fichier .cpp (aucun fichier d'en-tête .h n'est utilisé) et implémenter vos méthodes de test sous forme de méthodes publics ne renvoyant rien. Je vais utiliser un test simple pour la classe QgsRasterLayer tout au long de la section à suivre. Par convention, le test sera nommétel que la classe, avec le préfixe 'Test'. Notre implémentation de test ira donc dans un fichier nommé testqgsrasterlayer.cpp et la classe sera nommée TestQgsRasterLayer. Ajoutons d'abord notre bannière de droit d'auteur standardisée:

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

Ensuite, nous ajoutons les fichiers d'en-tête nécessaires à l'exécution du test. Il en existe un que tous les tests doivent avoir:

```
#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
```

Toutes nos méthodes de test sont implémentées sous la forme d'un emplacement privé. Le cadriciel QtTest appellera séquentiellement chaque méthode d'emplacement privé dans la classe de test. Il existe quatre méthodes 'spéciales' qui seront appelées au début du test unitaire (initTestCase) ou à la fin du test (cleanupTestCase). Avant le lancement de chaque méthode du test, la méthode init() est appelée. La méthode cleanup() est appelée à la fin de chaque méthode du tests. Ces méthodes sont intéressantes car elles vous permettent d'allouer et de nettoyer des ressources avant l'exécution de chaque test ainsi que pour l'intégralité du test unitaire.

```
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();
```

Viennent ensuite vos méthodes de tests qui ne doivent avoir aucun paramètre et renvoyer 'void'. Les méthodes sont appelées dans l'ordre de la déclaration. J'implémente ici deux méthodes qui illustrent deux types de tests. Dans le premier cas, je veux vérifier que les différentes parties de la classe fonctionnent et je peux utiliser une approche fonctionnelle. Une fois de plus, les partisans de l'extreme programming nous recommanderaient de créer ces tests avant d'implémenter la classe. Vous pouvez alors conduire vos tests unitaires au fur et à mesure de l'implémentation de votre classe. Vous devez écrire de plus en plus de fonctions de test à mesure que l'écriture de la classe s'améliore et une fois que le test unitaire est confirmé, votre nouvelle classe est terminée, tout en étant accompagnée d'un moyen de vérification répétable.

Généralement vos tests unitaires doivent couvrir uniquement la partie publique de l'API de votre classe et vous n'avez pas besoin d'écrire des tests pour les méthodes d'accès et les modificateurs. S'il devait arriver qu'une méthode d'accès ou de modification ne fonctionne pas comme prévu, vous devriez normalement implémenter un test de régression pour gérer ce cas (voir plus loin).

```
//
// Functional Testing
//

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

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

Ensuite, nous implémentons nos tests de non-régression. Les tests de non-régression doivent être écrits pour répliquer les conditions d'un bogue donné. Par exemple, j'ai reçu récemment un rapport de bogue par courrier électronique indiquant que le décompte des cellules de raster avait une erreur de 1, invalidant les statistiques des bandes de raster. J'ai ouvert un rapport de bogue (ticket #832) et j'ai alors créé un test de non-régression qui répliquait le bogue en utilisant un jeu de données assez compact (un raster de 10x10 cellules). J'ai ensuite lancé le test pour vérifier qu'il échouait bien (le décompte de cellule valait 99 au lieu de 100). J'ai ensuite corrigé le bogue et relancé le test unitaire qui, cette fois, est passé. J'ai déposé le test de non-régression ainsi que le correctif du bogue dans les sources. À partir de maintenant, si un futur développeur casse ce travail dans le code, nous pourrons directement identifier la régression dans le code. Lancer nos tests nous permettra de nous assurer que nos changements n'ont pas d'effets secondaires non prévus, comme casser des fonctionnalités existantes.

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.

Pour implémenter votre test de non-régression, il est préférable de suivre la convention de nommage du type regression pour les fonctions du test. S'il n'existe aucun ticket Redmin pour le bogue, vous devez en créer un d'abord. Cette approche permet de trouver plus facilement de l'information sur le test de non-régression qui échoue.

```
//
// 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 ...
```

En dernier lieu, vous pouvez déclarer en privé n'importe quelle propriété ainsi que les méthodes helper parmi les membres de la classe dans votre test unitaire. Dans notre cas, je vais déclarer un pointeur QgsRasterLayer * qui pourra être utilisé dans n'importe quelle méthode du test. La couche raster sera créée dans la fonction initTestCase() qui est lancée avant tous les autres tests; elle sera détruite par cleanupTestCase() qui sera lancée après les autres tests. En déclarant de manière privée les méthodes helper (qui peuvent être appelées par plusieurs fonctions de test), vous vous assurerez qu'elles ne seront pas automatiquement lancées par l'exécutable QTest créé lors de la compilation des tests.

```
  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() << st

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

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

```
}
```

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

1. J'avais besoin de paramétrer manuellement le chemin vers l'application QGIS de manière à ce que certaines ressources (comme le fichier srs.db) soient accessibles correctement.
2. De plus, il s'agit d'un test sur les données et nous avons donc besoin de fournir un moyen de localiser de manière générique le fichier `tenbytenraster.asc`. Il suffit d'utiliser la définition du compilateur `TEST_DATA_PATH`. Cette définition est créée dans le fichier de configuration `CMakeLists.txt` dans le répertoire `/tests/CMakeLists.txt` et elle est disponible pour tous les tests unitaires QGIS. Si vous avez besoin de données pour votre test, ajoutez les (via un commit) dans le répertoire `/tests/testdata`. Vous devriez uniquement ajouter des données de petite taille dans ce répertoire. Si votre test doit modifier des données, vous devriez copier ces dernière d'abord.

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.

Étudions ensuite notre test fonctionnel. Le test isValid() vérifie simplement que la couche raster a été correctement chargée dans initTestCase. QVERIFY est une macro Qt que vous pouvez utiliser pour évaluer une condition de test. Il existe quelques autres macros fournies par Qt que vous pouvez utiliser dans vos tests:

- *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 );
}
```

Une fois que les fonctions du test unitaire sont implémentées, il nous reste un élément final à ajouter à notre classe de test:

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

L'objectif de ces deux lignes est d'indiquer au cmo Qt qu'il s'agit d'un QtTest (qui générera une méthode main qui fera à son tour l'appel à chaque fonction du test). La dernière ligne correspond à l'inclusion pour les sources générées par le compilateur de meta-objet Qt. Vous devriez remplacer 'testqgsrasterlayer' par le nom de votre classe en caractères minuscules.

3.2 Ajouter votre test unitaire à CMakeLists.txt

Ajouter votre test unitaire au système de compilation consiste simplement à éditer le fichier CMakeLists.txt dans le répertoire de test, en clonant un des blocs de texte existants et en remplaçant l'ancien nom par le nom de votre classe. Par exemple:

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

3.3 La macro ADD_QGIS_TEST expliquée

Je vais brièvement expliquer ce qu'elles font mais si vous n'êtes pas intéressé, vous pouvez simplement reproduire les étapes expliquées dans les sections ci-dessus.

```
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)
```

Etudions plus en détails chaque ligne. Nous définissons d'abord la liste des sources de notre test. Étant donné que nous avons un seul fichier source (en suivant la méthodologie décrite au dessus ou la déclaration de la classe et la définition sont dans un seul fichier), il s'agit d'une simple formulation:

```
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})
```

Ensuite, nous indiquons à cmake qu'il doit produire un exécutable à partir de la classe de test. Souvenez-vous dans la section précédente, de la dernière ligne de l'implémentation de la classe où j'avais inclus les sorties vers le cmo directement dans la classe de test de manière à ce qu'il produise (entre-autres) une méthode main pour que la classe puisse être compilée comme un exécutable:

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

Ensuite, nous devons indiquer les dépendances vers des bibliothèques externes. Pour le moment, les classes ont été implémentées avec une dépendance générale QT_LIBRARIES mais je vais travailler ce point en remplaçant ce paramètre par les bibliothèques Qt spécifiques dont chaque classe à besoin. Bien entendu, vous devez réaliser une opération de lien (link) vers les bibliothèques QGIS concernées par votre test unitaire.

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

Ensuite, j'indique à cmake d'installer les tests au même endroit que les binaires de QGIS. Il s'agit d'un point que je pense supprimer dans le futur de manière à ce que les tests puissent être lancés directement depuis les sources.

```
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)
```

Enfin, ce qui est au dessus utilise `ADD_TEST` pour enregistrer le test avec cmake/ctest. C'est ici que le côté magique opère: nous enregistrons la classe avec ctest. Si vous vous rappelez de l'aperçu donné dans le début de ce chapitre, nous utilisons à la fois QtTest et CTest ensemble. Pour résumer, QtTest ajoute une méthode main à votre test unitaire et gère les appels aux méthodes du test au sein de la classe. Il fournit également des macros comme `QVERIFY` que vous pouvez utiliser comme test d'échec avec des conditions. La sortie d'un test unitaire QtTest est un exécutable que vous pouvez lancer depuis la ligne de commande. Néanmoins, lorsque vous disposez d'une suite de tests et que vous souhaitez lancer les exécutables les uns à la suite des autres ou que vous voulez intégrer le lancement des tests à la chaîne de compilation, CTest est alors utilisé.

3.4 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.

3.5 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
```

La commande make test invoquera CTest qui lancera chaque test enregistré en utilisant la directive CMake ADD_TEST décrite plus haut. Voici à quoi ressemble la sortie courante de make test:

```
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//mIconProjectionDisable
FAIL!: TestQgsApplication::checkTheme() '!myPixmap.isNull()' returned FALSE. ()
Loc: [/Users/tim/dev/cpp/qgis/tests/src/core/testqgsapplication.cpp(59)]
PASS : TestQgsApplication::cleanupTestCase()
Totals: 3 passed, 1 failed, 0 skipped
********* Finished testing of TestQgsApplication *********
-- Process completed
***Failed

## 0 tests passed, 1 tests failed out of 1

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

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

CHAPTER 4

Découvrir et aller plus loin avec QtCreator et QGIS

QtCreator est un IDE développé par les fondateurs de la bibliothèque Qt. Avec QtCreator, vous pouvez construire n'importe quel projet C++ mais l'IDE sera vraiment optimisé pour les personnes qui travaillent sur des applications basées sur Qt(4) (y compris les applications mobiles). Chaque point décrit ci-dessous assume 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 installé, vous devriez trouver ceci 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. Des instructions supplémentaires sont à *accès git* et installation de dépendances.

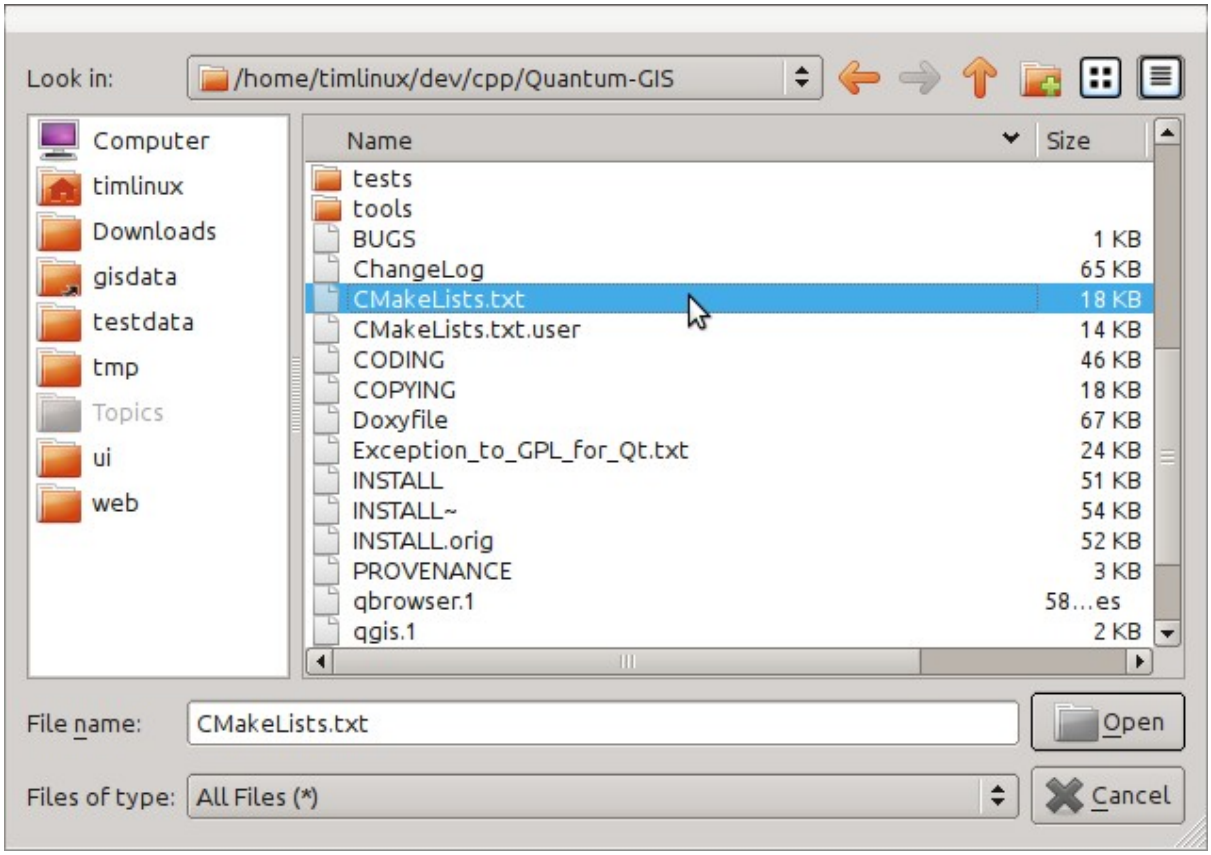
Sur mon système j'ai placé le code dans `$HOME/dev/cpp/QGIS` et le reste de cet article a été rédigé en présumant l'emplacement de ce répertoire. Vous pouvez modifier ces chemins pour les adapter à votre système local.

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:

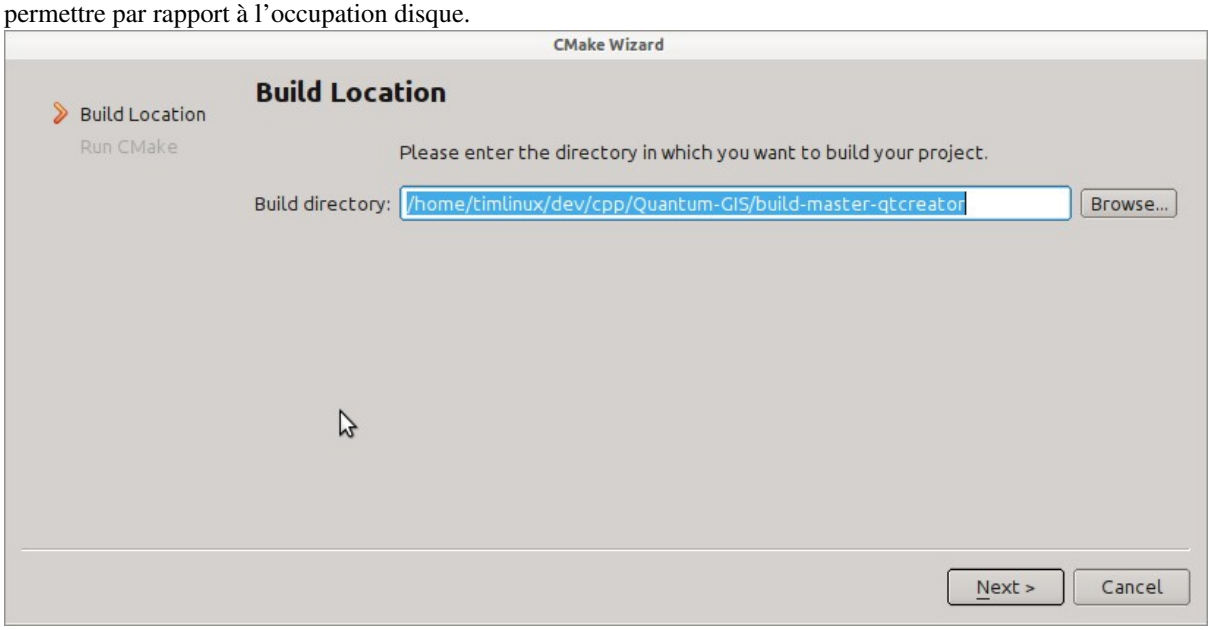
```
$HOME/dev/cpp/QGIS/CMakeLists.txt
```



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

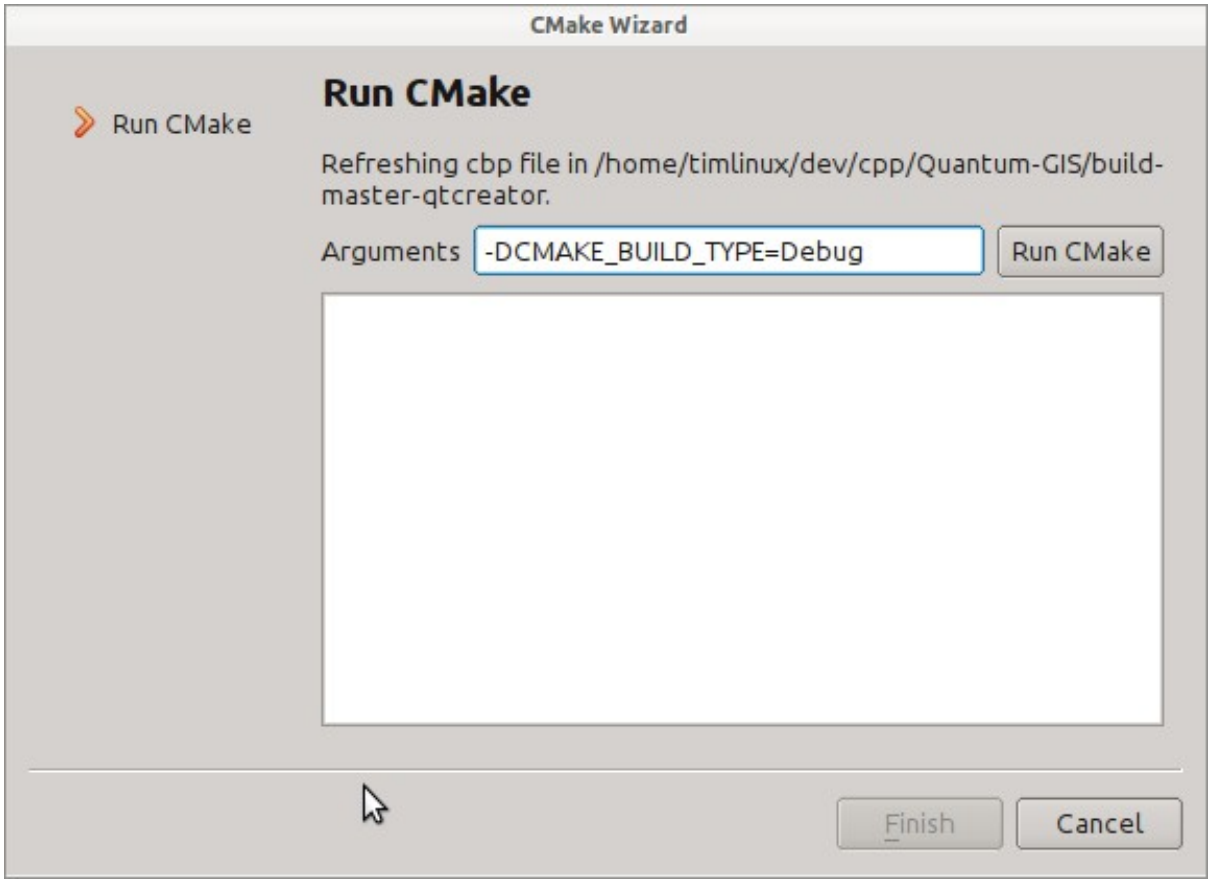
```
$HOME/dev/cpp/QGIS/build-master-qtcreator
```

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
```

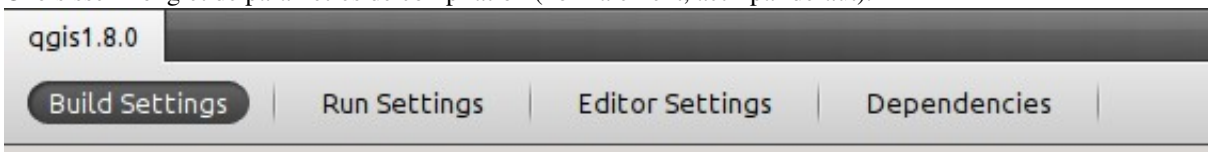


Voilà la base. Une fois que vous avez fermé l'assistant, QtCreator lancera une recherche dans l'arborescence du code source pour la gestion de l'autocomplétion et pour d'autres opérations, en tâche de fond. Avant de lancer la compilation, nous voulons encore paramétrer finement certains éléments.

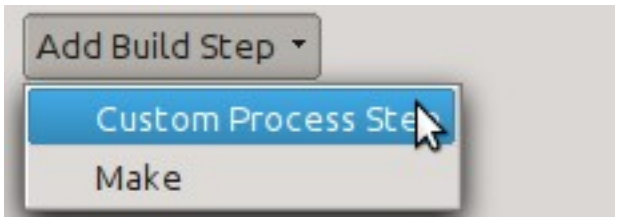
4.3 Paramétrer votre environnement de compilation

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 ? Parce que QGIS ne peut actuellement n'être lancé qu'à partir d'un répertoire d'installation et non depuis le répertoire de construction. Il est donc 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'.



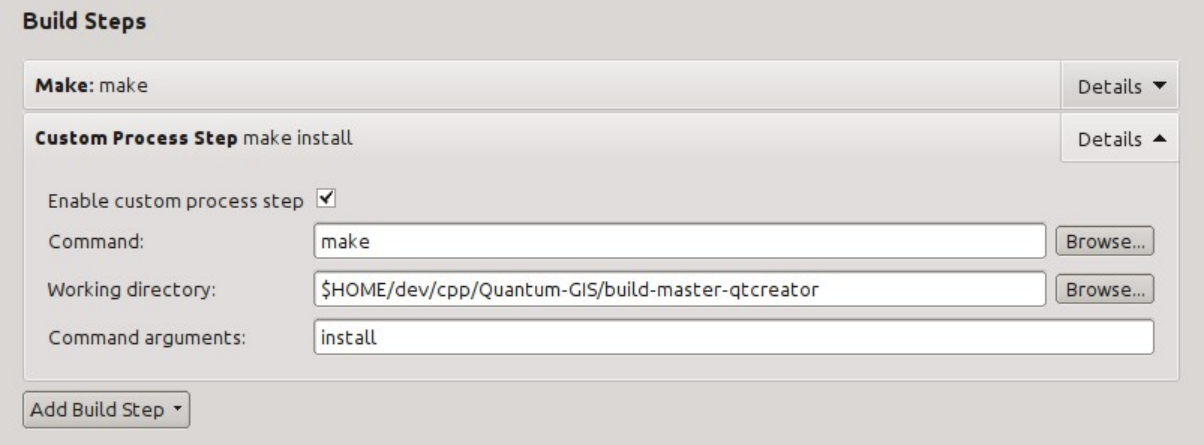
Maintenant, nous paramétrons les détails suivants:

Autoriser une étape personnalisée: [oui]

Commande: faire

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

Arguments de la commande: install



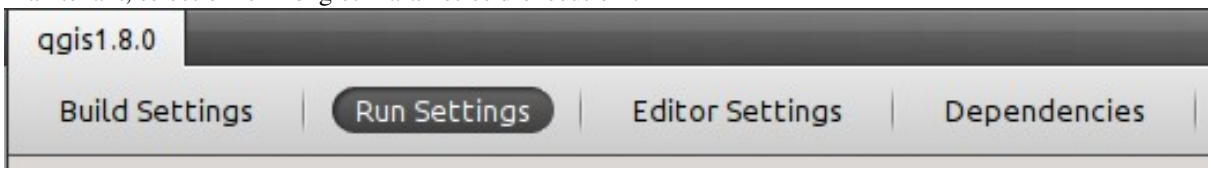
Vous êtes pratiquement prêts à compiler. Une dernière note: QtCreator a besoin des droits d'écriture sur le répertoire d'installation. Par défaut (ce que j'utilise ici), QGIS sera installé dans `/usr/local`. Pour mes besoins sur ma machine de développement, je me suis simplement donné des droits d'écriture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

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

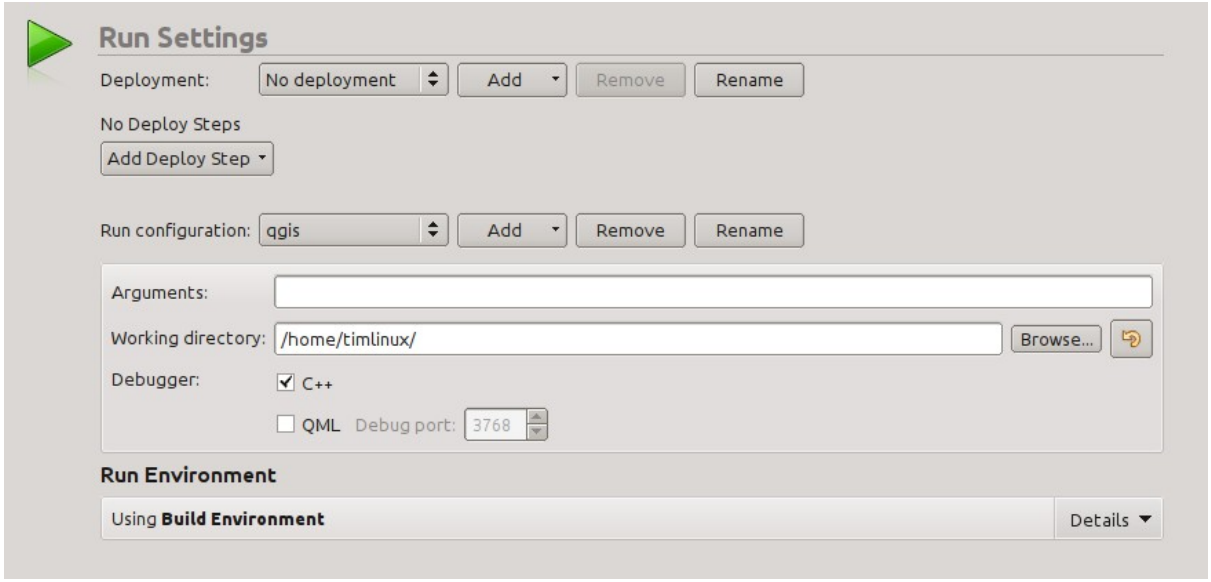
4.4 Paramétrer votre environnement de lancement

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 y parvenir, retournez dans l'écran de configuration des projets.

Maintenant, sélectionnez 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 y parvenir, 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:

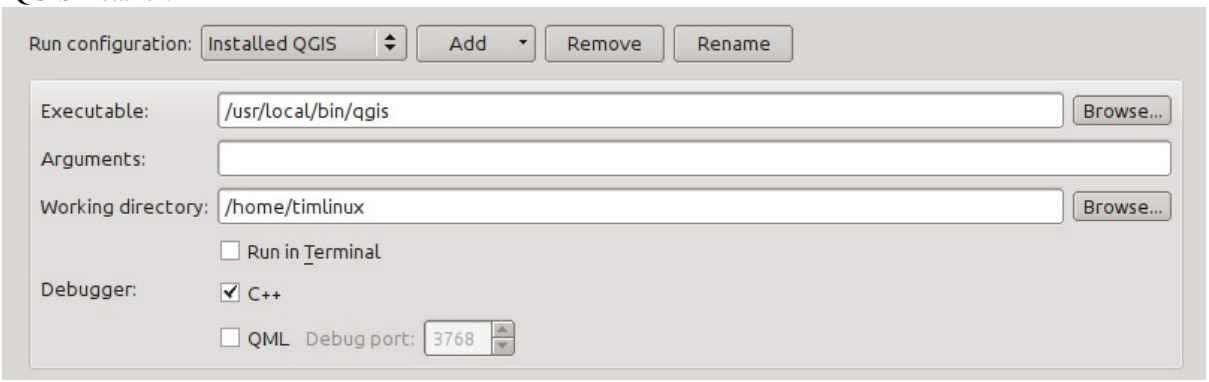
Répertoire de travail: $HOME

Lancer dans un terminal: [non]

Débogueur: C++ [oui]

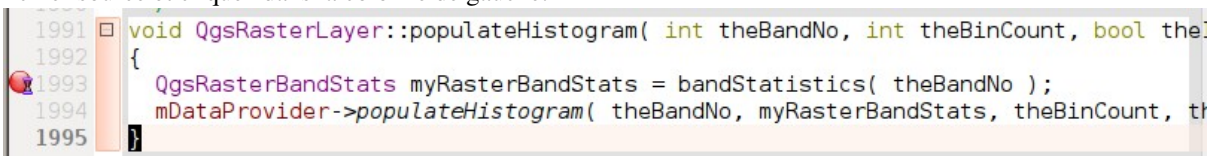
Qml [no]

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

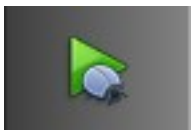


4.5 Exécution et déboguage

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

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

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

1. 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. Capitalise first letter only in labels: Labels (and group box labels) should be written as a phrase with leading capital letter, and all remaining words written with lower case first letters
3. 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]
4. Keep harmful actions away from harmless ones: If you have actions for 'delete', 'remove' etc, try to impose adequate space between the harmful action and innocuous actions so that the users is less likely to inadvertantly click on the harmful action.
5. 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.
6. 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.
7. 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.
8. Try to avoid technical terms and rather use a laymans equivalent e.g. use the word 'Transparency' rather than 'Alpha Channel' (contrived example), 'Text' instead of 'String' and so on.
9. Utilisez une iconographie cohérente. Si vous avez besoin d'icône ou d'éléments d'iones, merci de contacter Robert Szczepanek sur la liste de diffusion pour de l'assistance.
10. 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.
11. 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.
12. 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.
13. If clicking a button will spawn a new dialog, an ellipsis (...) should be suffixed to the button text.

5.1 Auteurs

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

CHAPTER 6

Tests de conformité OGC

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

6.1 Installation des tests de conformité WMS 1.3 et WMS 1.1.1

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

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
```

6.2 Projet test

Pour les tests WMS, les données doivent être téléchargées et chargées dans un projet Qgis :

```
wget http://cite.opengeospatial.org/teamengine/about/wms/1.3.0/site/data-wms-1.3.0.zip
unzip data-wms-1.3.0.zip
```

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

6.3 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
```

6.4 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
```