QGIS Developers Guide

Versiune 3.4

QGIS Project

mar. 15, 2020

Contents

Bine ați venit în Paginile Dezvoltatorului QGIS. Aici veți găsi regulile, instrumentele și pașii necesari pentru a contribui eficient, și cu ușurință, la codul QGIS.

CHAPTER 1

Standardele de codificare QGIS

Aceste standarde ar trebui să fie urmate de către toți dezvoltatorii QGIS.

1.1 Clase

1.1.1 Nume

Clasele din QGIS sunt prefixate cu Qgs, iar fiecare cuvânt începe cu majusculă.

Exemple:

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

1.1.2 Membri

Denumirile membrilor claselor sunt prefixate cu litera m și sunt formate utilizând majuscule și minuscule.

- `mMapCanvas`
- `mCurrentExtent`

Toți membrii claselor ar trebui să fie privați. Membrii publici sunt TOTAL nerecomandați. Membrii protejați ar trebui să fie evitați când membrul ar putea fi accesat din subclase Python, de vreme ce membrii protejați nu pot fi utilizați din legăturile Python.

Numele membrilor statici și mutabili ai claselor ar trebui să înceapă cu un `s` mic, în schimb numele membrilor statici constanți ar trebui să aibă toate literele mari:

- `sRefCounter`
- `DEFAULT_QUEUE_SIZE`

1.1.3 Funcțiile Accesor

Valorile membrilor unei clase vor fi obținute prin intermediul funcțiilor accesor. Numele acestora nu ar trebui să înceapă cu prefixul „get”. Funcțiile accesor pentru cei doi membri privați de mai sus ar putea fi:

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

Asigurați-vă că accesoriile sunt corect marcate cu `const`. Atunci când este cazul, este posibil ca variabilele membru, de tip cache de valoare, să fie marcate cu `mutable`.

1.1.4 Funcții

Numele funcțiilor încep cu literă mică și conțin majuscule și minuscule. Numele funcțiilor ar trebui să indice scopul acestora.

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

În concordanță cu API-urile QGIS și Qt, ar trebui evitate abrevierile. De exemplu: `setDestinationSize` în loc de `setDestSize`, `setMaximumValue` în loc de `setMaxVal`.

De asemenea, acronimele ar trebui să fie denumite folosind CamelCase. De exemplu: `setXml` în loc de `setXML`.

1.1.5 Argumente ale funcțiilor

Argumentele funcțiilor ar trebui să utilizeze nume descriptive. Nu utilizați argumente cu nume format dintr-o singură literă (ex. `setColor( const QColor& color )` în loc de `setColor( const QColor& c )`).

Acordați o atenție deosebită momentului când argumentele ar trebui să fie transmise prin referință. Cu excepția cazului în care obiectele argumentului nu au dimensiuni mari și sunt trivial de copiat (cum ar fi obiectele QPoint), ele ar trebui să fie transmise prin referințe constante. Pentru concordanța cu API-ul Qt, chiar și obiectele partajate în mod implicit sunt transmise prin referințe constante (ex.: `setTitle( const QString& title )` în loc de `setTitle( QString title )`.

1.1.6 Valorile Returnate de Funcție

Returnați ca valori obiectele mici și trivial copiate. Obiectele mai mari ar trebui returnate prin referințe constante. Singura excepție o reprezintă obiectele partajate în mod implicit, care sunt întotdeauna returnate prin valoare. Returnați ca pointeri `QObject` sau obiectele subclasate.

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

1.2 API Documentation

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

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

1.2.1 Metodele

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

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

1.2.2 Members Variables

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

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

1.3 Qt Designer

1.3.1 Clasele Generate

Clasele QGIS care sunt generate în fișierele produse de Qt Designer (UI) ar trebui să aibă sufixul Base. Acest lucru identifică o clasă de bază, generată.

Exemple:

- `QgsPluginManagerBase`
- `QgsUserOptionsBase`

1.3.2 Dialoguri

Toate dialogurile ar trebui să implementeze ajutorul pentru toate pictogramele barei de instrumente și alte controale grafice relevante. Baloanele cu indicii adaugă foarte mult la descoperirea caracteristicilor, atât pentru utilizatorii noi, cât și pentru cei experimentați.

Asigurați-vă că ordinea filelor pentru controalele grafice este actualizată ori de câte ori se modifică aspectul unui dialog.

1.4 Fișierele C++

1.4.1 Nume

Fișierele antet și de implementare C++ ar trebui să aibă extensiile .h, respectiv .cpp. Numele de fișier ar trebui să conțină doar litere mici și, în cazul claselor, să se potrivească numelui clasei.

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

Notă: În cazul în care nu este clară indicația de mai sus, cerința ca numele de fișier să se potrivească numelui de clasă, arată că, implicit, fiecare clasă trebuie să fie declarată și implementată în propriul fișier. Acest lucru facilitează nou-veniților identificarea codului care este specific anumitor clase.

1.4.2 Antetul Standard și Licența

Fiecare fișier sursă trebuie să conțină o secțiune antet, în conformitate cu exemplul următor:

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

Notă: Există un șablon pentru Qt Creator în git. Pentru a-l utiliza, copiați-l din `doc/qt_creator_license_template` într-un dosar local, ajustați adresa e-mail și numele - dacă este necesar - apoi configurați QtCreator pentru a-l folosi: *Instrumente* → *Opțiuni* → *C++* → *Denumirea Fișierului*.

1.5 Numele Variabilei

Numele variabilelor locale încep cu o literă mică, utilizând majuscule și minuscule. Nu folosiți prefixe precum `my` sau `the`.

Exemple:

- `mapCanvas`
- `currentExtent`

1.6 Tipurile Enumerate

Tipurile enumerate ar trebui să fie denumite folosind CamelCase, începând cu o majusculă, de ex.:

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

Nu utilizați tipuri de nume generice, care vor intra în conflict cu alte tipuri. De ex., folosiți mai degrabă `UnkownUnit` decât `Unknown`

1.7 Constantele locale și Comenzile Macro

Constantele locale și comenzile macro ar trebui să fie scrise cu majuscule, separate cu ajutorul caracterului de subliniere, de ex .:

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

1.8 Comentarii

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

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

1.9 Semnale și Sloturi Qt

Toate conexiunile la semnale/sloturi ar trebui să fie făcute folosindu-se legăturile de „stil nou" disponibile în Qt5. Informații suplimentare privind această cerință sunt disponibile în QEP #77.

Evitați folosirea sloturilor cu autoconectare QT (adică pe acelea denumite `void on_mSpinBox_valueChanged`). Sloturile cu autoconectare sunt fragile și predispuse la întrerupere, fără avertisment, dacă dialogurile sunt refactorizate.

1.10 Editarea

Orice editor de text/IDE poate fi folosit pentru a edita codul QGIS, oferind garanția că sunt îndeplinite următoarele cerințe.

1.10.1 Caracterele TAB

Setați în editorul dvs. emularea TAB-ului cu ajutotul spațiilor albe. Ar trebui folosite în acest scop 2 spații.

Notă: În vim, acest lucru se realizează prin comanda `set expandtab ts=2`

1.10.2 Indentarea

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

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

1.10.3 Acoladele

Acoladele ar trebui să fie poziționate pe linia următoare expresiei:

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

1.11 Compatibilitatea API-ului

There is API documentation for C++.

Încercăm să păstrăm API-ul stabil și compatibil cu versiunile anterioare. Curățarea API-ului ar trebui să fie făcută într-un mod similar cu al codului sursă Qt, de 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 SIP Bindings

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

1.12.1 Header pre-processing

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

- Use `#ifdef SIP_RUN` to generate code only in SIP files or `#ifndef SIP_RUN` for C++ code only. `#else` statements are handled in both cases.
- Use `SIP_SKIP` to discard a line
- The following annotations are handled:
  - `SIP_FACTORY`: `/Factory/`
  - `SIP_OUT`: `/Out/`
  - `SIP_INOUT`: `/In,Out/`
  - `SIP_TRANSFER`: `/Transfer/`
  - `SIP_PYNAME(name)`: `/PyName=name/`
  - `SIP_KEEPREFERENCE`: `/KeepReference/`

  - SIP_TRANSFERTHIS: /TransferThis/
  - SIP_TRANSFERBACK: /TransferBack/

- private sections are not displayed, except if you use a #ifdef SIP_RUN statement in this block.
- SIP_PYDEFAULTVALUE(value) can be used to define an alternative default value of the python method. If the default value contains a comma ,, the value should be surrounded by single quotes '
- SIP_PYTYPE(type) can be used to define an alternative type for an argument of the python method. If the type contains a comma ,, the type should be surrounded by single quotes '

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

1.12.2 Generating the SIP file

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

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

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

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

1.12.3 Improving sipify script

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

1.13 Stilul de Codificare

Aici sunt descrise câteva sugestii și sfaturi de programare care vor reduce, sperăm, erorile, timpul de întreținere și dezvoltare.

1.13.1 Generalizați Codul Atunci Când Este Posibil

Decât să duplicați un anumit cod, mai bine luați în considerare consolidarea acestuia într-o funcție unică.

Acest lucru vă va permite să:

- efectuați modificări într-o singură locație în loc de mai multe
- preveniți încurcarea codului
- împiedica apariția, de-a lungul timpului, a diferențelor între secțiunile identice de cod, făcându-le, astfel, mai greu de înțeles și de întreținut de către alții

1.13.2 Se preferă poziționarea Constantelor înaintea Predicatelor

Se preferă poziționarea constantelor la începutul predicatelor.

0 == value în loc de value == 0

Acest lucru va ajuta prevenirea folosirii accidentale de = în loc de ==, care poate introduce erori subtile de logică. Compilatorul va genera o eroare dacă folosiți accidental = în loc de == pentru comparații, deoarece nu se pot atribui valori constantelor.

1.13.3 Spațiile Albe Vă Pot Fi De Ajutor

Adăugarea de spații între operatori, declarații și funcții, facilitează utilizatorilor analiza codului.

Care cod este mai ușor de citit? Acesta:

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

sau acesta:

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

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

1.13.4 Puneți comenzile pe linii separate

La citirea codului, este ușor să omiteți comenzile dacă acestea nu se află la începutul liniei. La parcurgerea rapidă a codului, este normal să omiteți liniile atunci când acestea nu prezintă ceea ce căutați în primele câteva caractere. Este, de asemenea, normal să vă așteptați la o comandă după un condițional ca `if`.

Se ia în considerare următorul cod:

```
if (foo) bar();

baz(); bar();
```

Este foarte ușor să omiteți o parte din fluxul de control. Scrieți, în schimb

```
if (foo)
  bar();

baz();
bar();
```

1.13.5 Indentați modificatorii de acces

Modificatorii de acces structurează o clasă în secțiuni de API public, API protejat și API privat. Rolul lor este de a grupa codul pe această structură. Indentați modificatorii de acces și declarațiile.

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

1.13.6 Cărți recomandate

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

Ar trebui, de asemenea, să citiți acest articol de la Qt Quarterly despre proiectarea în stilul Qt (API)

1.14 Recunoașterea contribuțiilor

Contribuitorii la noile funcții sunt încurajați să-i informeze pe ceilalți despre contribuția lor prin:

- adăugarea unei note la jurnalul schimbărilor, pentru prima versiune în care a fost încorporat codul, de tipul:

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

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

CHAPTER 2

HIG (Ghidul Interfeței cu Utilizatorul)

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

1. Elementele aflate în relație cu un grup folosesc casetele de grup: Încercați identificarea elementelor care pot fi grupate împreună, apoi utilizați casetele de grup alături de o etichetă, pentru a identifica subiectul acelui grup. Evitați utilizarea casetelor de grup care conțin doar un singur control grafic / element.
2. Capitalise first letter only in labels, tool tips, descriptive text, and other non-heading or title text: These should be written as a phrase with leading capital letter, and all remaining words written with lower case first letters, unless they are nouns
3. Capitalize all words in Titles (group box, tab, list view columns, and so on), Functions (menu items, buttons), and other selectable items (combobox items, listbox items, tree list items, and so on): Capitalize all words, except prepositions that are shorter than five letters (for example, «with» but «Without»), conjunctions (for example, and, or, but), and articles (a, an, the). However, always capitalize the first and last word.
4. Nu încheiați etichetele componentelor de interfață sau a casetelor de grup cu două puncte: Adăugarea a două puncte produce un efect vizual neplăcut și nu conferă un sens suplimentar, astfel încât nu se vor folosi. O excepție de la această regulă apare atunci când două etichete apar una lângă alta, cum ar fi: Label1 Plugin (Path:) Label2 [/calea/spre/plugin-uri]
5. Keep harmful actions away from harmless ones: If you have actions for «delete», «remove» etc, try to impose adequate space between the harmful action and innocuous actions so that the users is less likely to inadvertently click on the harmful action.
6. Utilizați întotdeauna un QButtonBox pentru butoanele «OK», «Cancel» etc: Folosirea controlului grafic respectiv va garanta că ordinea butoanelor «OK» și «Cancel», etc, este în concordanță cu sistemul de operare / setările regionale / desktop-ul pe care îl folosește utilizatorul.
7. Evitați imbricarea filelor. În cazul în care utilizați filele, folosiți genul de file utilizate pentru QgsVectorLayerProperties / QgsProjectProperties etc, adică file poziționate în partea superioară, având pictograme de 22x22.
8. Stivuirea controalelor grafice ar trebui să fie evitată, dacă este posibil. Acest lucru produce confuzie și conduce la redimensionarea inexplicabilă (pentru utilizator) a dialogurilor, pentru a se potrivi controalelor grafice care nu sunt vizibile.
9. Try to avoid technical terms and rather use a laymans equivalent e.g. use the word «Opacity» rather than «Alpha Channel» (contrived example), «Text» instead of «String» and so on.

10. Utilizați o iconografie consistentă. În cazul în care aveți nevoie de o pictogramă sau de elemente de pictogramă, vă rugăm să-l contactați pe Robert Szczepanek, pe lista de discuții pentru asistență.
11. Poziționați listele lungi de controale grafice în casete cu derulare. Nici un dialog nu ar trebui să depășească 580 de pixeli în înălțime și 1000 de pixeli în lățime.
12. Separați opțiunile avansate față de cele de bază. Utilizatorii începători ar trebui să poată accesa rapid elementele necesare pentru activitățile de bază, fără a se preocupa de complexitatea caracteristicilor avansate. Funcțiunile avansate ar trebui să fie amplasate fie sub o linie de demarcație, fie într-o filă separată.
13. Nu adăugați opțiuni de dragul de a avea o mulțime de opțiuni. Încercați să mențineți o interfață cu utilizatorul minimalistă, și utilizați judicios setările implicite.
14. If clicking a button will spawn a new dialog, an ellipsis char (...) should be suffixed to the button text. Note, make sure to use the U+2026 Horizontal Ellipsis char instead of three periods.

2.1 Autori

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

CHAPTER 3

Accesul GIT

Această secțiune vă arată cum să începeți lucrul cu depozitul GIT al QGIS. Înainte de a face acest lucru, trebuie să aveți un client git instalat pe sistemul dumneavoastră.

3.1 Instalarea

3.1.1 Instalarea git pentru GNU/Linux

Utilizatorii distribuției Debian pot efectua:

```
sudo apt install git
```

3.1.2 Instalarea git pentru Windows

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

3.1.3 Instalarea git pentru OSX

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

O dată descărcat, deschideți imaginea discului și executați programul de instalare.

Notă PPC/sursă

Site-ul Git nu oferă compilații PPC. În cazul în care trebuie să construiți o compilație PPC, sau doriți doar ceva mai mult control asupra instalării, trebuie să efectuați singuri compilarea.

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

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

Dacă nu aveți nevoie de facilități suplimentare, Perl, Python sau TclTk (GUI), aveți posibilitatea să le dezactivați înainte de a rula comanda make:

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

3.2 Accesarea Depozitului

Pentru a clona QGIS master:

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

3.3 Selectați o ramură

Pentru a selecta o ramură, de exemplu, ramura 2.6.1:

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

Pentru a selecta ramura master:

```
cd QGIS
git checkout master
```

Notă: În QGIS păstrăm codul nostru cel mai stabil din ramura versiunii actuale. Ramura master conține codul pentru așa-numita serie de versiuni «instabile». Periodic vom ramifica o versiune pe care o vom stabiliza continuu și în care vom încorpora selectiv noi caracteristici.

Parcurgeți fișierul INSTALL din arborele sursă, pentru instrucțiunile specifice privind construirea versiunilor de dezvoltare.

3.4 Sursele documentației QGIS

În cazul în care sunteți interesat în verificarea surselor documentației QGIS:

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

Puteți arunca, de asemenea, o privire la fișierele readme incluse în depozitul cu documentație, pentru mai multe informații.

3.5 Sursele site-ului web QGIS

În cazul în care sunteți interesat în verificarea surselor site-ului web QGIS:

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

Puteți arunca, de asemenea, o privire la fișierele readme incluse în depozitul site-ului web, pentru mai multe informații.

3.6 Documentația GIT

Parcurgeți următoarele site-uri pentru informații despre cum putți deveni un maestru GIT.

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

3.7 Dezvoltarea în ramuri

3.7.1 Scopul

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 Procedura

- **Anunțul inițial pe listele de discuții:** Înainte de a începe, postați un anunț pe lista de discuții a dezvoltatorilor, pentru a vedea dacă un alt dezvoltator lucrează deja la aceeași caracteristică. De asemenea, contactați consilierul tehnic al comitetului (PSC) de coordonare a proiectului. În cazul în care noua caracteristică necesită o modificare a arhitecturii QGIS, este necesară o cerere de comentariu (RFC).

Crearea unei ramuri: Creați o nouă ramură GIT, pentru dezvoltarea noii funcțiuni.

```
git checkout -b newfeature
```

Acum puteți începe dezvoltarea. Dacă aveți de gând să lucrați intensiv în această ramură, și să lucrați alături de alți dezvoltatori, care să aibă drepturi de scriere în depozitul părinte, puteți urca depozitul dvs. în depozitul oficial al QGIS:

```
git push origin newfeature
```

Notă: În cazul în care ramura există deja, modificările dvs. vor fi urcate acolo.

Reîncorporați, în mod regulat, în ramura master: Este recomandabil să reintroduceți modificările în ramura master, în mod regulat. Acest lucru face mai ușoară fuziunea ulterioară a ramurii cu masterul. După încorporare trebuie să efectuați `git push -f` asupra depozitului dvs. derivat.

Notă: Niciodată nu efectuați `git push -f` în depozitul original! Folosiți această comandă numai pentru ramura în care lucrați.

```
git rebase master
```

3.7.3 Efectuarea de teste înaintea fuzionării cu ramura master

Când ați terminat cu noua caracteristică și vă mulțumește stabilitatea, faceți un anunț pe lista dezvoltatorilor. Înainte de fuziunea cu masterul, modificările vor fi testate de către dezvoltatori și utilizatori.

3.8 Transmiterea de corecții și a cererilor de actualizare

Iată câteva indicații care vă vor ajuta să obțineți cu ușurință corecții și solicitări de actualizare pentru proiectele QGIS, facilitându-ne gestionarea corecțiilor.

3.8.1 Cererile de actualizare

În general, este mai ușor pentru dezvoltatori dacă trimiteți cereri de actualizare a codului de pe GitHub. Pentru detalii asupra acestui aspect, faceți referire la documentația GitHub pull request.

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

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

Vă rugăm să consultați secțiunea de mai jos pentru a «obține atenție pentru corecția dvs.». În general, atunci când trimiteți o cerere PR ar trebui să vă asumați responsabilitatea de a o urmări până la finalizare - să răspundeți la întrebările postate de alți dezvoltatori, să identificați un «susținător» pentru funcționalitatea propusă de dvs., și să transmiteți un memento manierat, uneori, dacă vedeți că PR-ul nu este luat în considerare. Vă rugăm să țineți cont de faptul că proiectul QGIS este condus printr-un efort voluntar, fiind posibil ca oamenii să nu participe

instantaneu la PR. Dacă observați că un PR nu beneficiază de atenția pe care o merită, opțiunile de accelerare a aceastuia ar trebui să fie (în ordinea priorității):

- Trimiterea unui mesaj în lista de discuții, pentru a «populariza» PR-ul, indicând și cât de bine ar fi dacă ar fi inclus în baza de cod.
- Trimiterea unui mesaj persoanei din lista de PR-uri, căreia i-a fost asignată cererea dvs.
- Trimiterea unui mesaj lui Marco Hugentobler (care gestionează lista de PR-uri).
- Trimiterea unui mesaj comitetului director al proiectului, cerându-le să vă ajute la încorporarea PR-ului în codul de bază.

Cele mai bune practici pentru crearea unei cereri de actualizare

- Creați întotdeauna o ramură pentru o funcționalitate, pornind de la ramura master curentă.
- Atunci când codificați ramura pentru o caracteristică, nu „îmbinați" nimic cu acea ramură, ci mai degrabă folosiți comanda rebase, așa cum este descris în punctul următor, pentru a păstra curat istoricul.
- Înaintea creării unei cereri de actualizare a codului efectuați `git fetch origin` și `git rebase origin/master` (datorită faptului că originea reprezintă remote pentru părinte, efectuați `.git/config` sau `git remote -v | grep github.com/qgis`).
- Ați putea executa comanda Git rebase, în mod repetat, fără a provoca nici un prejudiciu (atât timp cât singurul scop al ramificării dvs. este de a obține fuzionarea cu ramura master).
- Atenție: După rebase trebuie să efectuați `git push -f`` în depozitul ramificat. **DEZVOLTATORI DE BAZĂ: NU FACEȚI ASTA ÎNTR-UN DEPOZIT QGIS PUBLIC**!

Etichete speciale pentru a notifica documentatorii

Pe lângă etichetele comune pe care le puteți adăuga pentru a clasifica PR-urile dvs., există unele speciale, pe care le puteți utiliza pentru a genera automat rapoarte de probleme în depozitul Documentației QGIS, de îndată ce codul dumneavoastră este combinat:

- `[needs-docs]` pentru a aminti creatorilor documentației să ceară adăugarea de informații suplimentare, după corectarea sau îmbunătățirea unei funcționalități existente deja.
- `[feature]` in case of new functionality. Filling a good description in your PR will be a good start.

Rugăm dezvoltatorii să folosească aceste etichete (nu se face diferenţiere între litere mari şi mici), astfel încât creatorii documentației să identifice problemele pe care le au și să aibă o privire de ansamblu asupra lucrurilor de făcut. DAR, faceți-vă, de asemenea, timp pentru a adăuga un text: fie în commit, fie chiar în documentație.

Pentru îmbinarea unei cereri de actualizare

Opţiunea A:

- faceți clic pe butonul de îmbinare (Se creează o îmbinare non-fast-forward)

Opţiunea B:

- Verificați cererea de actualizare
- Testare (De asemenea, necesar pentru opțiunea A, evident)
- checkout master, git merge pr/1234
- Opțional: `git pull --rebase`: Creează fast-forward, fără „merge commit". Istoricul este mai curat, dar este mai greu de anulat îmbinarea.
- `git push` (NICIODATĂ nu folosiți opțiunea -f aici)

3.9 Denumirea fişierului de corecţie

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

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

3.10 Crearea corecției la nivelul superior al directorului sursă QGIS

Procedând în acest mod, aplicarea corecțiilor este mai ușoară, deoarece nu trebuie să navigăm într-un loc specific din arborele sursă pentru aplicarea unei corecții. De asemenea, atunci când primim corecții, de obicei, le vom evalua folosind îmbinarea, iar aplicarea corecției la nivel directorului superior facilitează acest lucru. Mai jos este un exemplu despre cum se pot include în corecție, din directorul de nivel superior, mai multe fișiere modificate:

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

Acest lucru vă asigură că ramura master este sincronizată cu depozitul părinte, și apoi generează o corecție care conține diferențele dintre ramura noii funcționalități și ramura master.

3.10.1 Obținerea atenției necesare pentru corecția dvs.

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

3.10.2 Măsuri de Protecţie

QGIS este licențiat sub GPL. Ar trebui să depuneți toate eforturile pentru a vă asigura că trimiteți numai corecții care nu sunt grevate de drepturi de proprietate intelectuală, de natură conflictuală. De asemenea, nu trimiteți un cod care nu se încadrează în GPL.

3.11 Obținerea Accesului de Scriere în GIT

Accesul de scriere în arborele sursei QGIS are loc pe bază de invitație. În mod obișnuit, atunci când o persoană depune mai multe corecții substanțiale (nu există un număr fix), care să demonstreze competența și înțelegerea C++ de bază și a convențiilor de codificare QGIS, unul dintre membrii PSC, sau alți dezvoltatori, pot nominaliza persoana respectivă la acordarea accesului la scriere. Cel care face nominalizarea ar trebui să justifice printr-un paragraf promoțional, motivele pentru care consideră că o persoană ar trebui să aibă acces la scriere. În unele cazuri, vom acorda acces la scriere dezvoltatorilor non C++, de ex. pentru traducători și creatorii de documentație. Chiar și în aceste cazuri, persoanele trebuie să demonstreze o înțelegere a proceselor de modificare a bazei de cod fără producerea de erori, etc.

Notă: Din momentul orientării către GIT, suntem mai puțin susceptibili la acordarea accesului de scriere pentru noii dezvoltatori, deoarece este banală partajarea codului din cadrul github, prin bifurcarea QGIS, urmată de emiterea cererilor de actualizare.

Verificați întotdeauna că totul se compilează corect, înaintea efectuării cererilor de actualizare. Încercați să conștientizați posibilele defecțiuni pe care le-ar putea produce actualizările dvs. asupra celor care lucrează pe alte platforme, sau cu versiuni mai vechi/mai noi ale bibliotecilor.

Atunci când se face o actualizare, editorul dvs. va fi indicat (așa cum este definit în variabila de mediu $EDITOR), iar dvs. va trebui să efectuați un comentariu în partea de sus a fișierului (deasupra zonei care spune «nu efectuați modificări»). Introduceți un comentariu descriptiv și, mai degrabă, efectuați mai multe mici, în cazul în care modificările dintr-un număr mare de fișiere sunt independene. Pe de altă parte, preferăm să actualizați împreună modificările efectuate în fișiere înrudite.

CHAPTER 4

Pregătirea lucrului cu QtCreator și QGIS

QtCreator este un nou IDE produs de dezvoltatorii bibliotecii Qt. Folosind QtCreator puteți construi orice proiect C++, dar este cu adevărat optimizat pentru persoanele care lucrează cu aplicații bazate pw Qt(4) (inclusiv aplicații mobile). În acest articol am presupus că rulați Ubuntu 11.04 «Natty».

4.1 Instalarea aplicației QtCreator

Această parte este ușoară:

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

După instalare, ar trebui să-l găsiți în meniul gnome.

4.2 Configurarea proiectului dvs.

Presupunem că aveți deja o clonă de QGIS locală, care conține codul sursă, și că ați instalat toate dependențele necesare compilării etc. Există instrucțiuni detaliate pentru *accesul la git* și instalarea dependențelor.

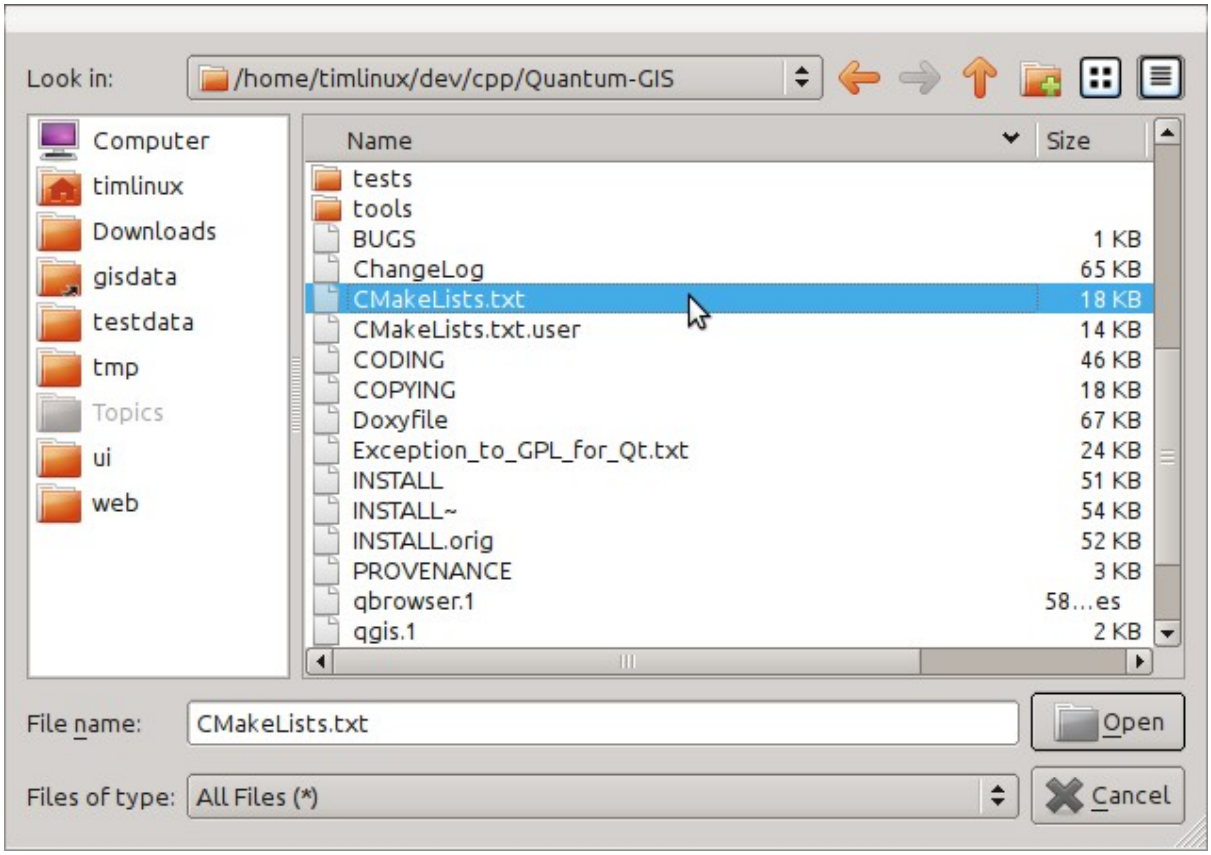
Pe sistemul de test, codul se află în `$HOME/dev/cpp/QGIS` iar restul articolului este scris presupunând că veți actualiza aceste căi, după caz, pentru sistemul dumneavoastră.

Pentru lansarea QtCreator:

Fișier -> Deschidere Fișier sau Proiect

Apoi, selectați în dialogul care se va deschide, următorul fișier:

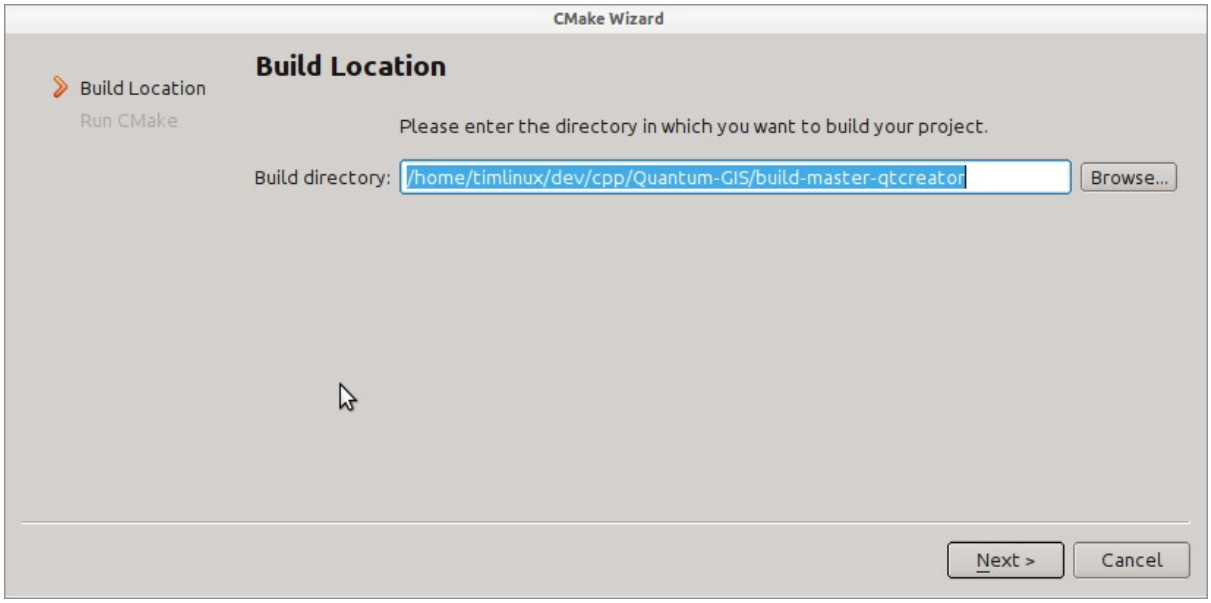
```
$HOME/dev/cpp/QGIS/CMakeLists.txt
```



În continuare vi se va cere o locație pentru compilare. Vom crea un director specific compilațiilor QtCreator:

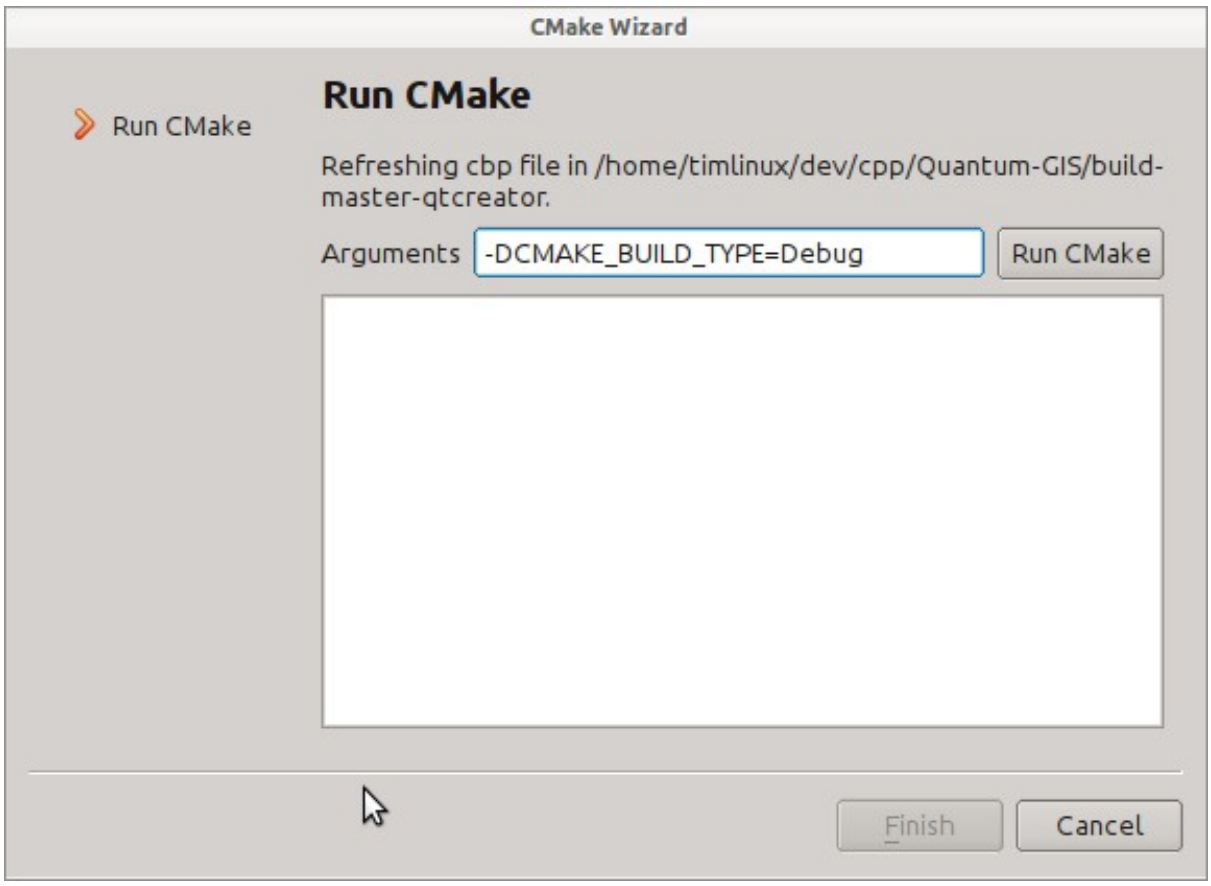
```
$HOME/dev/cpp/QGIS/build-master-qtcreator
```

Probabil că reprezintă o idee bună crearea unor directoare separate pentru diversele ramuri, dacă aveți suficient spațiu pe disc.



În continuare, vi se vor cere opțiunile de compilare pentru CMake. Îi vom indica lui CMake că dorim o compilare de depanare, prin adăugarea acestei opțiuni:

```
-DCMAKE_BUILD_TYPE=Debug
```

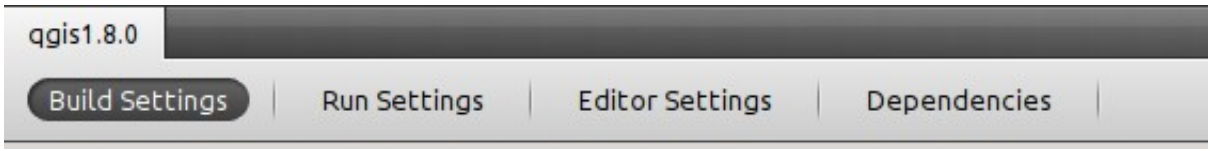


Astea sunt elementele de bază. La definitivare, QtCreator va începe scanarea arborelui sursă pentru autocompletare, și pentru alte operațiuni de întreținere în fundal. Dorim să mai stabilim câteva lucruri înainte de a începe compilarea.

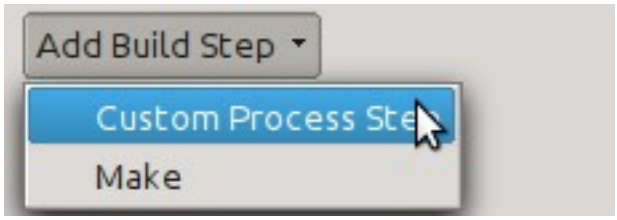
4.3 Instalarea mediului de compilare

Faceți clic pe pictograma «Proiecte», din partea stângă a ferestrei QtCreator.

Selectați fila setărilor de compilare (de obicei, activă în mod implicit).



Acum, dorim să adăugăm un pas de procesare personalizat. De ce? Pentru că aplicația QGIS poate rula numai dintr-un director de instalare, diferit de directorul de compilare, deci trebuie să ne asigurăm că este instalat după fiecare compilare. În conformitate cu «Pașii de Compilare», faceți clic pe butonul «Adăugare BuildStep» și alegeți «Pas de Procesare Personalizat».



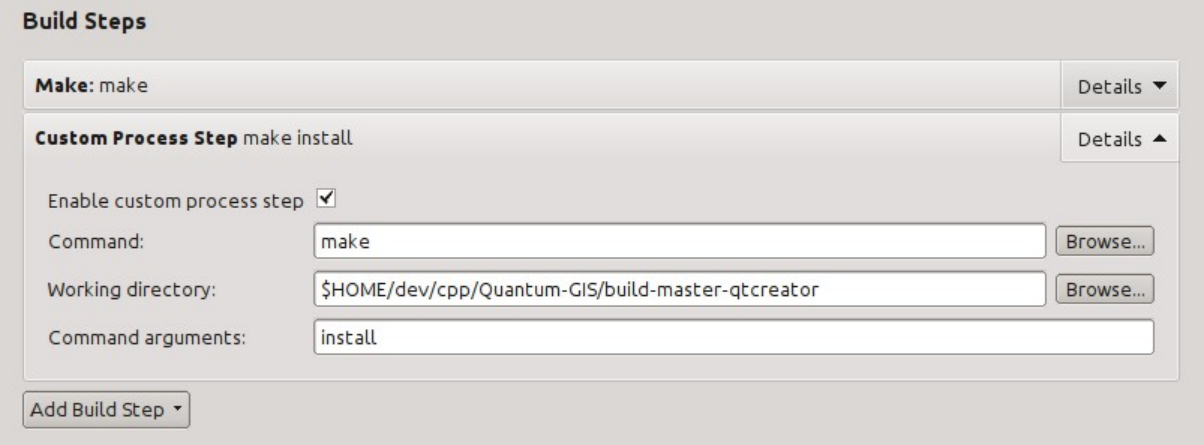
Acum, am stabilit următoarele detalii:

Activare pas de procesare personalizat: [da]

Comanda: make

Directorul de lucru: $HOME/dev/cpp/QGIS/build-master-qtcreator

Argumentele comenzii: install



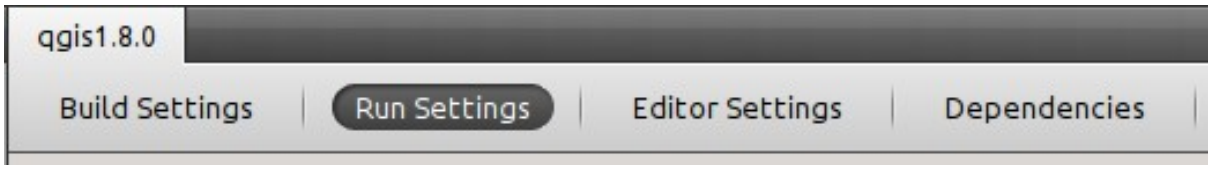
Sunteți aproape gata pentru compilare. Doar o singură notă: QtCreator va avea nevoie de permisiuni de scriere asupra prefixului de instalare. În mod implicit (așa cum am procedat aici) QGIS va fi instalat în `/usr/local/`. În scopuri de dezvoltare, există permisiuni de scriere pentru directorul /usr/local.

Pentru a porni compilarea, faceți clic pe acea pictogramă cu un ciocan mare, în partea din stânga-jos a ferestrei.

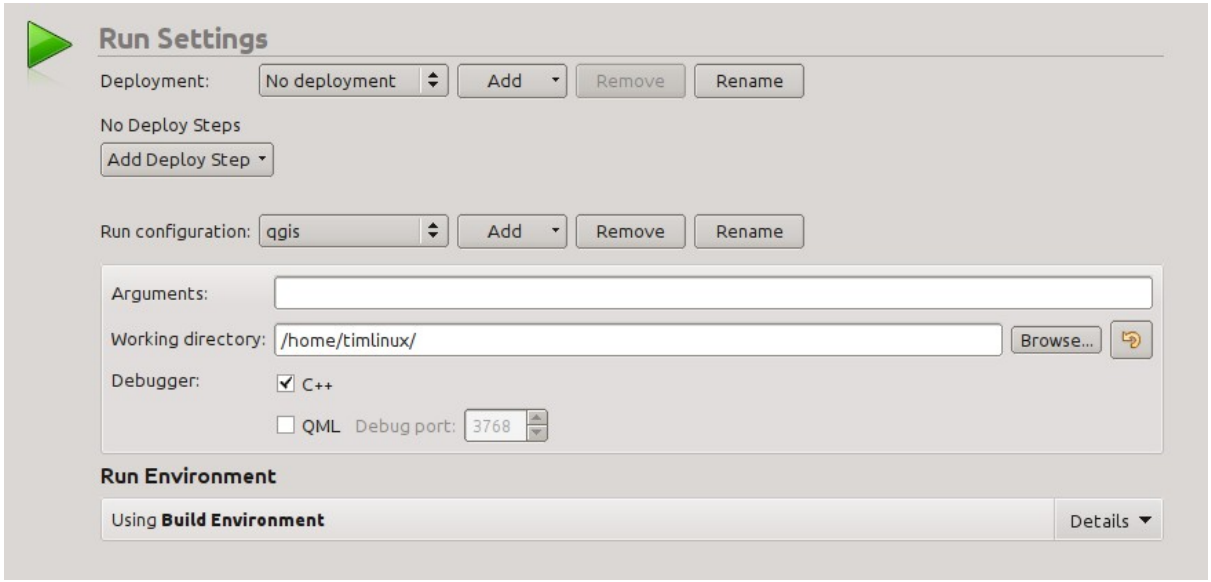
4.4 Instalarea mediului de dezvoltare

După cum s-a menționat mai sus, nu putem rula QGIS în mod direct, din directorul de compilare, așa că trebuie să oferim aplicației QtCreator posibilitatea de a rula QGIS din directorul de instalare (în cazul nostru `/usr/local/`). Pentru a face acest lucru, reveniți la ecranul de configurare a proiectelor.

Acum, selectați fila «Rulare Setări»



Trebuie să actualizăm setările implicite de rulare, personalizând configurația de rulare «qgis».



Pentru aceasta, faceți clic pe butonul «Add v» de lângă caseta de configurare Run, apoi alegeți «Executabil Personalizat» din partea de sus a listei.

Acum, în zona proprietăților setați următoarele detalii:

Executabil: /usr/local/bin/qgis

Argumente :

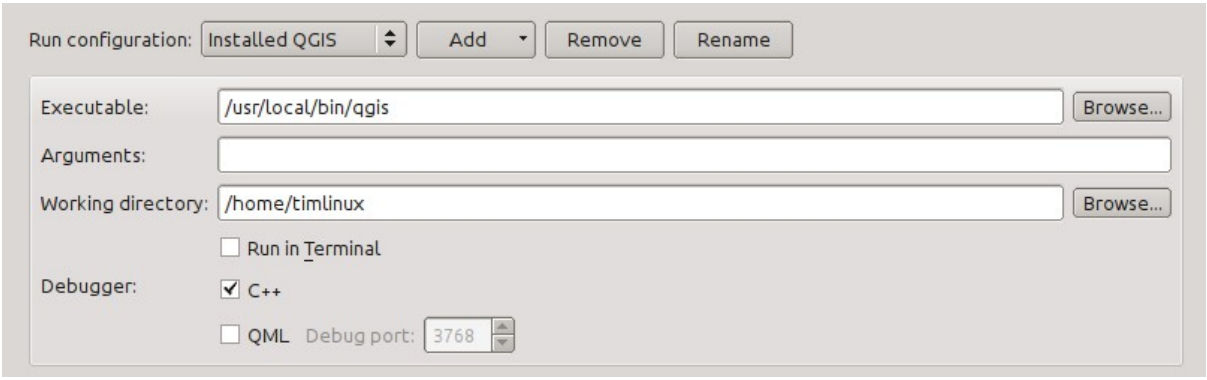
Directorul de lucru: $HOME

Rulare în terminal: [nu]

Depanator: C++ [da]

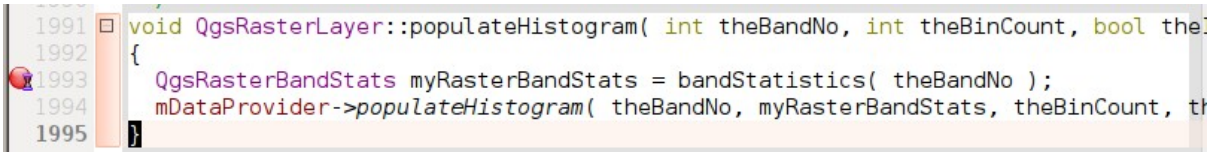
Qml [nu]

Apoi, faceți clic pe butonul «Redenumire» și dați-i executabilului personalizat un nume sugestiv, cum ar fi «QGIS Instalat»

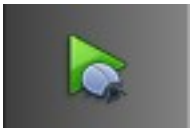


4.5 Rulare și depanare

Acum puteți rula și depana QGIS. Pentru a seta un punct de pauză, pur și simplu deschideți un fișier sursăm apoi faceți clic în coloana din stânga.



Lansați QGIS în modul de depanare, făcând clic pe pictograma corespunzătoare, în partea din stânga-jos a ferestrei.



CHAPTER 5

Testarea Unităților

Din noiembrie 2007 am cerut ca toate noile caracteristici care intră în versiunea master să fie însoțite de teste de unitate. Inițial ne-am limitat la qgis_core, apoi vom extinde această cerință pentru alte părți ale bazei de cod, o dată ce dezvoltatorii se vor familiariza cu procedurile pentru testele de unitate, detaliate în secțiunile următoare.

5.1 Cadrul de testare QGIS - o privire de ansamblu

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. Creați un test de unitate. Acest lucru se întâmplă în directorul `<QGIS Source Dir>/tests/src/core`, în cazul bibliotecilor de bază. Testul este de fapt un client care creează o instanță a unei clase,

apelând unele metode ale acestei clase. Acesta va verifica valorile returnate din fiecare metodă, pentru a se asigura că se potrivesc valorilor așteptate. În cazul în care unul dintre apeluri eșuează, testul de unitate va eșua.

3. Includeți macro-urile QtTestLib în clasa dvs. de test. Macrocomenzile sunt procesate de către compilatorul de obiecte meta Qt (MOC) și transformă clasa de testare într-o aplicație executabilă.
4. Adăugați o secțiune la fișierul CMakeLists.txt din directorul testelor dvs., care va construi testul.
5. Asigurați-vă că ați activat `ENABLE_TESTING` în ccmake / cmakesetup. Astfel, vă veți asigura că testele dumneavoastră se compilează atunci când tastați 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. Compilați sursele, apoi efectuați instalarea. Faceți acest lucru utilizând procedura normală `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 Crearea unui test de unitate

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>
```

Mai departe, continuați implementarea normală a clasei, incluzând antetele de care aveți nevoie:

```
//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>
```

Din moment ce combinăm atât declarația clasei cât și implementarea într-un singur fișier, urmează declarația clasei. Vom începe cu documentația doxygen. Fiecare caz de testare trebuie să fie documentat în mod corespunzător. Folosim directiva doxygen ingroup, astfel încât toate Unitățile de Testare apar ca module în documentația Doxygen generată. Urmează apoi o scurtă descriere a unității de testare, clasa trebuind să moștenească QObject și să includă macrocomanda 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.

Mai există un beneficiu al prezenței testelor de regresie - acestea pot aduce economie de timp. Dacă ați depanat vreodată o eroare care a implicat efectuarea de modificări în codul sursă, urmată de rularea aplicației și efectuarea unei serii de pași extrem de complicați pentru a reproduce problema, va fi imediat evident că simpla implementare a testului de regresie înainte de corectarea erorii permite automatizarea testării, ceea ce reprezintă o eliminare eficientă a defectelor.

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;
};
```

Astfel se termină declarația clasei noastre. Implementarea are loc pur și simplu, în partea de jos a fișierului. Mai întâi are loc inițializarea și curățarea funcțiilor:

```
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;
}
```

Funcția de inițializare de mai sus ilustrează câteva lucruri interesante.

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 oferă, de asemenea, alte câteva mecanisme interesante pentru testarea dirijată cu ajutorul datelor, așa că, dacă vă interesează să aflați mai multe despre acest subiect, consultați documentația 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*)

Unele dintre aceste macro-uri sunt utile numai atunci când se utilizează cadrul de lucru Qt, pentru testarea cu ajutorul datelor (a se vedea documentația Qt, pentru mai multe detalii).

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

În mod normal, testele funcționale ar putea acoperi toată gama de funcționalități a claselor dvs. API publice, acolo unde este fezabil. Testele funcționale fiind gata, ne putem uita la exemplul nostru de test de regresie.

Având în vedere că problema #832 raportează un număr eronat de celule, scrierea testului nostru constă pur și simplu în folosirea macrocomenzii QVERIFY, pentru a verifica dacă numărul de celule corespunde valorii așteptate:

```
void TestQgsRasterLayer::regression832()
{
  QVERIFY ( mpLayer->getRasterXDim() == 10 );
  QVERIFY ( mpLayer->getRasterYDim() == 10 );
  // regression check for ticket #832
  // note getRasterBandStats call is base 1
  QVERIFY ( mpLayer->getRasterBandStats(1).elementCountInt == 100 );
}
```

With all the unit test functions implemented, there's one final thing we need to add to our test class:

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

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

5.3 Comparing images for rendering tests

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

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

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

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

E.g.

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

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

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

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

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

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

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

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

To improve tests robustness here are few tips:

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

5.4 Adăugarea testului de unitate în 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 Adăugarea macocomenzii ADD_QGIS_TEST

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})
```

Din moment ce clasa noastră de test trebuie să fie executată prin intermediul compilatorului meta-obiect (MOC) din Qt, trebuie să indicăm acest lucru printr-o pereche de linii:

```
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 Compilarea testului de unitate

Pentru a compila testul de unitate rewbuie doar să vă asigurați că `ENABLE_TESTS = true` în configurația cmake. Există două moduri de a face acest lucru:

1. Rulați `ccmake ..` (sau `cmakesetup ..` sub windows) și setați, în mod interactiv, fanionul `ENABLE_TESTS` pe `ON`.
2. Adăugați o linie de comandă pentru fanion în cmake ex.: `cmake -DENABLE_TESTS=true ..`

La urmă, doar compilați normal QGIS, iar testele ar trebui să se compileze, la rândul lor.

5.6 Rularea testelor dumneavoastră

Cel mai simplu mod de a rula testele este ca parte a procesului normal de compilare:

```
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
```

Dacă un test eșuează, puteți utiliza comanda ctest pentru a-i examina mai îndeaproape cauza. Utilizați `opțiunea -R` pentru a specifica un regex pentru testele pe care doriți să le rulați, și un ‘‘ -V‘‘ pentru a obține o ieșire detaliată:

```
$ ctest -R appl -V

Start processing tests
Test project /Users/tim/dev/cpp/qgis/build
Constructing a list of tests
Done constructing a list of tests
Changing directory into /Users/tim/dev/cpp/qgis/build/tests/src/core
## 13 Testing qgis_applicationtest
Test command: /Users/tim/dev/cpp/qgis/build/tests/src/core/qgis_applicationtest
********* Start testing of TestQgsApplication *********
Config: Using QTest library 4.3.0, Qt 4.3.0
PASS : TestQgsApplication::initTestCase()
PrefixPATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/../
PluginPATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//lib/qgis
PkgData PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//share/qgis
User DB PATH: /Users/tim/.qgis/qgis.db
PASS : TestQgsApplication::getPaths()
PrefixPATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/../
PluginPATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//lib/qgis
PkgData PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//share/qgis
User DB PATH: /Users/tim/.qgis/qgis.db
QDEBUG : TestQgsApplication::checkTheme() Checking if a theme icon exists:
QDEBUG : TestQgsApplication::checkTheme()
/Users/tim/dev/cpp/qgis/build/tests/src/core/..//share/qgis/themes/default//
↪mIconProjectionDisabled.png
FAIL!: TestQgsApplication::checkTheme() '!myPixmap.isNull()' returned FALSE. ()
Loc: [/Users/tim/dev/cpp/qgis/tests/src/core/testqgsapplication.cpp(59)]
PASS : TestQgsApplication::cleanupTestCase()
Totals: 3 passed, 1 failed, 0 skipped
********* Finished testing of TestQgsApplication *********
-- Process completed
***Failed

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

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

5.6.1 Debugging unit tests

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

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

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

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

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

5.6.2 Have fun

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

CHAPTER 6

Testarea Algoritmilor de Procesare

6.1 Algorithm tests

Notă: 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 Procedura

Pentru a adăuga un test nou, urmați acești pași:

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

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

The above translates to

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

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

6.1.2 Parameters and results

Trivial type parameters

Parameters and results are specified as lists or dictionaries:

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

or

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

Layer type parameters

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

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

This is what it looks like in action:

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

File type parameters

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

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

Rezultate

Results are specified very similarly.

Basic vector files

It couldn't be more trivial

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

Add the expected GML and XSD files in the folder.

Vector with tolerance

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

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

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

Raster files

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

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

Fișiere

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

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

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

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

Directories

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

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

6.1.3 Algorithm Context

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

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

6.1.4 Running tests locally

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

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

CHAPTER 7

Testarea Conformității cu 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.

7.1 Configurarea testelor de conformitate cu WMS 1.3 and 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
```

Descărcați și instalați testul WMS 1.3.0

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

Descărcați și instalați testul WMS 1.1.1

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

7.2 Proiectul de testare

Pentru testele WMS, datele pot fi descărcate și încărcate într-un proiect 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 Rularea testului 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 Rularea testului 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
```