Developpez.com
Club des développeurs

# Utiliser cppUnit pour gérer ses tests unitaires

par Matthieu Brucher (http://matthieu-brucher.developpez.com/) (Blog)

Date de publication : 24/01/2007

Dernière mise à jour :

Comment tester son application ? Comment gérer ses tests unitaires ? cppUnit est un outil porté en C++ de la référence, JUnit. Voyons comment l'utiliser, quels sont ses avantages et ses inconvénients.

Présentation rapide de cppUnit
I - Structure de cppUnit
I-A - La classe Test
I-B - La classe TestCase
I-C - La classe TestSuite
I-D - La classe TestDecorator
I-E - La classe TestFixture
I-F - La classe TestRunner
II - Compilation de cppUnit
II-A - Compilation sous Linux ou avec les autotools - MingW et MSYS par exemple -
II-B - Compilation avec un environnement de développement (IDE)
II-B-1 - Compilation avec MSVC
II-B-2 - Compilation avec un autre IDE
III - Compilation de QtTestRunner
IV - Exemple d'utilisation de cppUnit
IV-A - Lancer l'interface graphique
IV-B - Des tests simples
IV-C - Des tests plus complexes
V - Plus de possibilités avec cppUnit
V-A - Avantages de cppUnit
V-B - Inconvénients de cppUnit
Conclusion
Téléchargements


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

## Présentation rapide de cppUnit

cppUnit est l'équivalent C++ de l'outil **JUnit** créé entre autres par Kent Beck. Plateforme de test unitaire, cppUnit sert à organiser ses tests unitaires. Un tel test permet de vérifier le bon fonctionnement de son code. Idéalement, le test devrait être écrit avant la fonction à tester. De plus, une application doit être testée sous toutes ses coûtures, chaque fonction, chaque module ayant un ou plusieurs tests associés. Pour plus de renseignements sur ces modes de développement, jetez un oeil sur ce livre : **Test-Driven Development : By Example**

cppUnit est donc une bibliothèque de tests pouvant se télécharger à l'adresse suivante sur **Sourceforge**. Dans ce petit tutoriel, nous essaierons de présenter rapidement mais correctement cette bibliothèque et l'une de ses interfaces graphiques, QtTestRunner. Outre vérifier les tests, cppUnit permet aussi de vérifier qu'une fonction ne produit aucune erreur imprévue. On a donc un contrôle des échecs d'une fonction - ce qu'on prévoit qui pourrait mal se passer - ainsi que de ses erreurs - ce qui se passe mal alors qu'on ne l'avait pas prévu -.


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

# I - Structure de cppUnit

Une rapide présentation des classes de cppUnit et de leur hiérarchie s'impose. Voici un diagramme réduit des classes.


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

*Diagramme réduit de cppUnit*

## I-A - La classe Test

Cette classe est la classe principale du framework. Virtuelle pure, elle définit entre autres la fonction **runTest**, fonction qui est naturellement appelée lors de l'exécution des tests. C'est elle qui sera surchargée par nos tests unitaires. Les autres fonctions membres servent à diverses tâches tel le comptage du nombre de tests, récupérer les résultats des tests à travers le pattern Visitor, ...

## I-B - La classe TestCase

Classe la plus élémentaire, elle est dérivée de **Test** au travers de **TestLeaf**. Elle peut être considérée comme une feuille de l'arbre des tests.

## I-C - La classe TestSuite

Cette classe implémente le pattern **Composite**, à savoir une arborescence dans les tests. C'est le moyen de hiérarchiser facilement et efficacement ses tests.

## I-D - La classe TestDecorator

Cette classe est virtuelle mais a 2 filles.

Dans certains cas, les tests envisagés nécessitent de mettre en place un cadre, charger des données, mais aussi de "nettoyer" après les tests. 2 fonctions supplémentaires sont à surcharger pour accomplir cette dernière tâche dans la classe **TestSetUp**.

Certains tests doivent être répétés plusieurs fois, ce que la classe **RepeatedTest** permet d'accomplir.

## I-E - La classe TestFixture

Parfois, on ne veut pas dériver une multitude de classes pour des tests différents. En effet, pour tester une bibliothèque matricielle par exemple, créer une classe par type d'opération peut être démesuré, tout comme mettre tous les tests dans une seule classe. La classe **TestFixture** permet de répondre à cette question.

## I-F - La classe TestRunner

Pour lancer les tests, une autre classe est utilisée. Chaque classe de test est enregistrée dans **TestRunner** à l'aide de la méthode **addTest**, et on lance la totalité des tests à l'aide de la méthode **run**.


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

# II - Compilation de cppUnit

Tout d'abord, téléchargez cppUnit ici pour obtenir **la dernière version stable** et décompressez l'archive.

## II-A - Compilation sous Linux ou avec les autotools - MingW et MSYS par exemple -

Un petit

```
./configure && make && make install
```

permet d'installer facilement la bibliothèque. Si vous voulez l'installer sans être root ou dans un dossier particulier, faites plutôt un

```
./configure --prefix=/dossier/local/dinstallation/ && make && make install
```

## II-B - Compilation avec un environnement de développement (IDE)

Avec un autre environnement de développement, plusieurs solutions sont envisageables. Si on décide de compiler avec Visual Studio, tout est déjà disponible. Dans le cas contraire, il faudra reconstruire un projet.

### II-B-1 - Compilation avec MSVC

Ouvrez le fichier **src/CppUnitLibraries.dsw**, les versions supérieures à MSVC6 demanderont de convertir la solution, ce que vous ferez. Compilez ensuite le tout, soit avec la configuration Debug, soit avec la Release. Avec la version 7.1, plusieurs sous-projets échouent lors de la compilation, mais ce n'est pas un problème, le projet important étant celui compilant le fichier cppunit_dll.dll ou cppunitd_dll.dll. PAr la suite, on admettra avoir compiler le projet en mode Debug.

### II-B-2 - Compilation avec un autre IDE

Dans votre IDE favori, créez un nouveau projet de bibliothèque dynamique. Indiquez que vous voulez ajouter le dossier include/ de cppUnit à votre chemin d'accès. Ajoutez la macro *CPPUNIT_DLL_BUILD* à la liste des commandes du préprocesseur si vous êtes sous Windows, incluez tous les fichiers .cpp du dossier **src/cppunit** et compilez-le.


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

## III - Compilation de QtTestRunner

Qt a l'avantage d'avoir son propre système de compilation. Grâce au **tutoriel d'Aurélien Regat-Barrel**, vous pouvez aussi utiliser cette méthode pour compiler le projet avec le compilateur Microsoft.

Pour compiler QtTestRunner, téléchargez le package **ici** et décompressez l'archive dans **src/qttestrunner** pour utiliser les fichiers les plus récents. Téléchargez de même notre version perso du fichier projet **ici** et remplacez la version qui se trouve dans **src/qttestrunner**. Ici, si vous êtes sous Linux, modifiez le chemin d'accès à la librairie cppUnit en conséquence à la ligne *LIBS+=*. Supprimez les fichiers commençant par **moc_**, les versions fournies ne sont pas adaptées ax dernières versions de Qt4, on demandera au script de compilation de les recréer.

Si vous utilisez Visual Studio, ouvrez un terminal à l'aide du raccourci dans Démarrer>Programmes>Microsoft Visual Studio XXX>Outils>Command Prompt et exécutez

```
qmake && nmake
```

si vous utilisez un GCC avec make, dans un terminal, faites un

```
qmake && nmake
```

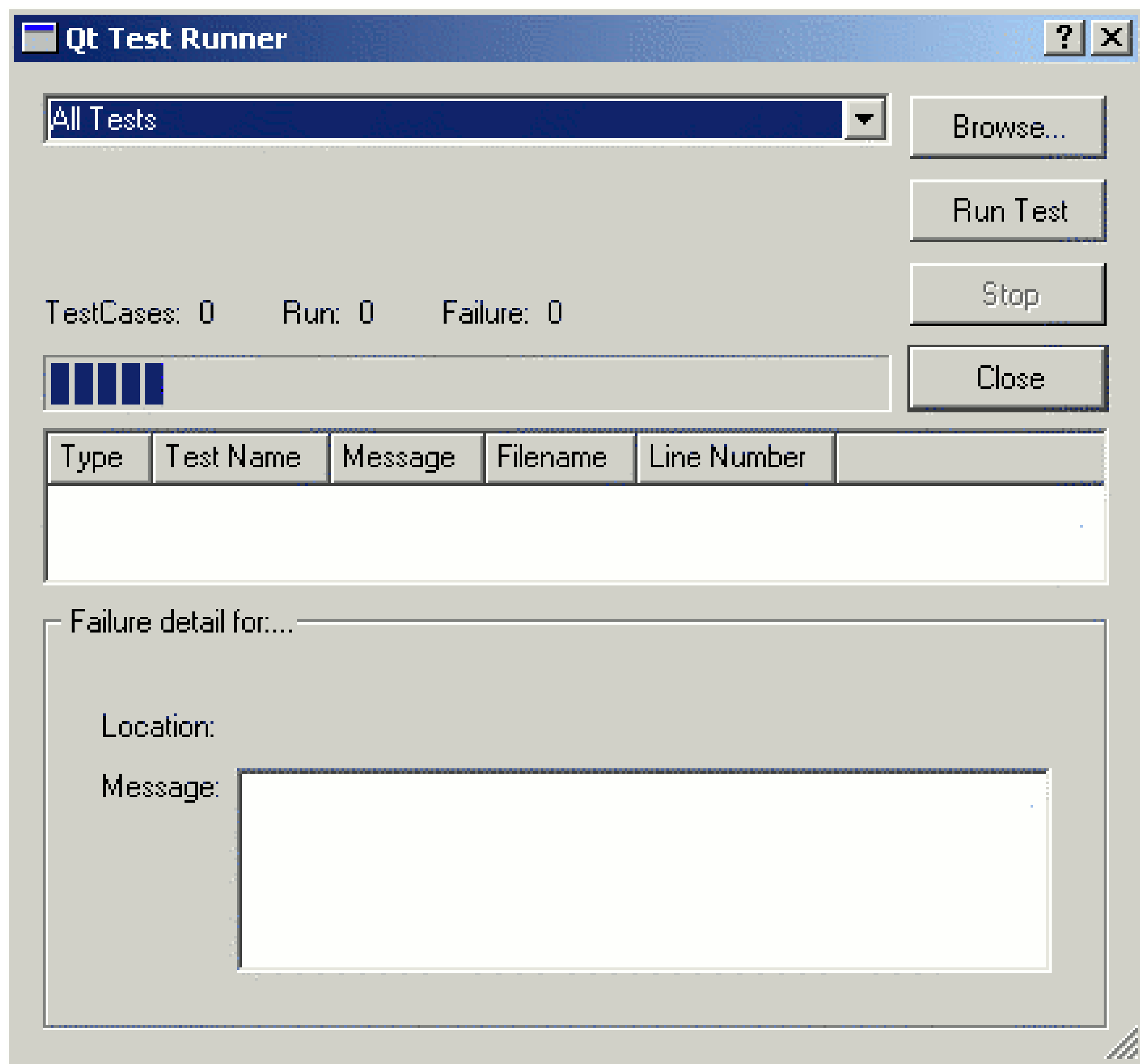
Maintenant, vous vous retrouvez avec une bibliothèque dynamique cppUnit et une bibliothèque dynamique pour QtTestRunner, le tout compilé en mode debug.


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

# IV - Exemple d'utilisation de cppUnit

## IV-A - Lancer l'interface graphique

Nous allons commencer par lancer l'interface graphique. Afin de simplifier la compilation, nous utiliserons les fichiers .pro à utiliser avec qmake - il faut ajouter plusieurs définitions du préprocesseur pour que l'application se lance avec une compilation standard sous Visual Studio -. L'interface doit donc ressembler à quelque chose de ce genre




*Interface Qt*

On va donc créer un fichier .pro. Tout d'abord la configuration

```
CONFIG  += debug qt warn_on
```

Donc on compile en mode debug, avec qt et avec les warnings.

```
SOURCES = main.cpp
```


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

On va simplement utiliser un seul fichier source, main.cpp

```
TARGET          = cppunit-example
DEPENDPATH      = .
OBJECTS_DIR     = objs
MOC_DIR         = mocs
```

La cible à créer est un exécutable s'appelant cppunit-example, on stocke les fichiers objets dans le sous-dossier **objs**, les fichiers résultant d'un moc dans le dossier **mocs** tandis que la liste des dossiers utilisés pour les dépendances est limité au dossier courant.

```
INCLUDEPATH     = c:/travail/cppunit-1.10.2/include
```

Ici, indiquez le dossier dans lequel se trouve les en-têtes de cppUnit, dans mon cas, c'est c:/travail/cppunit-1.10.2/include.

```
LIBS += c:\travail\cppunit-1.10.2\lib\cppunitd_dll.lib
 c:\travail\cppunit-1.10.2\lib\qttestrunner.lib
```

Ici, mettez vos libairies .lib, avec Visual Studio, il s'agit du chemin d'accès complet tandis que pour les GCC, on indiquera par -Lchemin le chemin d'accès et -llibrarie chaque librairie, donc -lcppunit et -lqttestrunner.

Maintenant, le fichier main.cpp

Les en-têtes

```
#include <QtGui/QApplication>
#include <cppunit/ui/qt/TestRunner.h>
```

D'abord, on va inclure les en-têtes indispensables, donc QApplication pour pouvoir créer une application avec Qt et TestRunner.h qui va définir l'interface graphique elle-même.

Le corps de la fonction main

```
int main(int argc, char* argv[])
{
 // Create the QtTestRunner:
 QApplication app(argc, argv);
 CppUnit::QtTestRunner runner;

 // Run the GUI:
 runner.run();

  return 0;
}
```

Le corps de la fonction main elle-même est simple, on crée la QApplication, l'interface graphique TestRunner puis on lance les tests.

## IV-B - Des tests simples

Comme précisé dans la première partie, les tests dérivent de la classe **Test**. En particulier, on va utiliser la classe **TestCase** et surcharger virtuellement sa fonction runTest().


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

A rajouter dans le fichier .pro

```
HEADERS = test.h
```

On va donc juste définir une petite classe dans le fichier test.h, classe qui sera entièrement inlinée pour simplifier.

Les en-têtes

```
#include <cppunit/extensions/HelperMacros.h>
```

On va ajouter un en-tête dans notre nouveau fichier test.h, celui qui définit des macros pour simplifier l'écriture des tests.

```
class PseudoTest : public CppUnit::TestCase
{
public:
  PseudoTest( std::string name ) : CppUnit::TestCase( name ) {}

  void runTest()
  {
    CPPUNIT_ASSERT( 1 == 1 );
    CPPUNIT_ASSERT( !(1 == 2) );
  }
};
```

Notre classe va donc s'appeler PseudoTest, surtout parce que c'est un exemple, pas une vraie classe. Le constructeur va passer à son père, TestCase, le paramètre name qui est le nom du test. La fonction de test elle-même, runTest, utilise la macro CPPUNIT_ASSERT. En paramètre, une simple condition à tester, comme pour la fonction assert classique.

Code à ajouter dans main.cpp après les en-têtes précédents

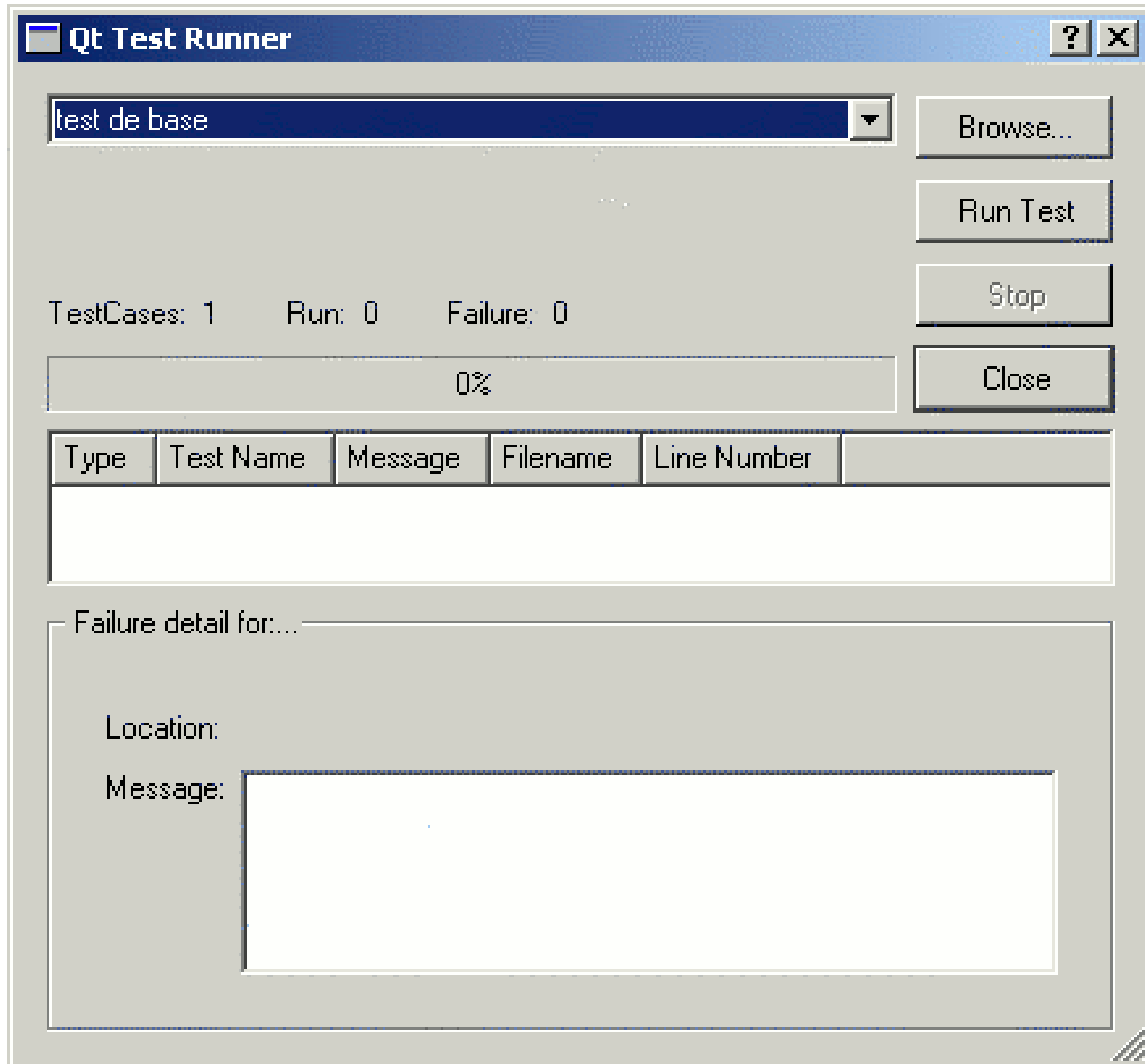
```
#include "test.h"
```

Code à ajouter dans main.cpp dans le corps de main

```
 runner.addTest(new PseudoTest("test de base"));
```

On indique simplement à l'instance de la classe TestRunner qu'on veut ajouter un test qui sera une instance de la classe PseudoTest qu'on a déclarer avant. Le résultat est maintenant clairement visible.


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.




*En ajoutant un test*

## IV-C - Des tests plus complexes

On va maintenant essayer d'aller un peu plus loin en utilisant une autre classe dérivant de Test, TestFixture. Cette classe propose simplement de gérer plusieurs fonctions de test. De plus, nous allons utiliser un singleton fourni par la bibliothèque pour enregistrer tous nos tests automatiquement au démarrage de l'application. Il ne sera donc plus utile de les ajouter avec la méthode addTest().

A remplacer dans le fichier .pro

```
SOURCES  = main.cpp test.cpp
```

On ajoute un fichier source à la compilation.

Le nouveau fichier test.cpp

```
#include "test.h"

CPPUNIT_TEST_SUITE_REGISTRATION( PseudoTest );
```



Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Le fichier est simple, outre l'inclusion du fichier d'en-tête, on a une nouvelle macro, CPPUNIT_TEST_SUITE_REGISTRATION, qui indique simplement qu'on enregistre au démarrage une suite de test au démarrage.

La nouvelle classe PseudoTest définie dans test.h

```
class PseudoTest : public CppUnit::TestFixture
{
public:
  CPPUNIT_TEST_SUITE( PseudoTest );
  CPPUNIT_TEST( testsReussis );
  CPPUNIT_TEST( testsRates );
  CPPUNIT_TEST_SUITE_END();

  /// Méthode dont les tests réussissent
  void testsReussis()
  {
    CPPUNIT_ASSERT( 1 == 1 );
    CPPUNIT_ASSERT( !(1 == 2) );
  }

  /// Méthode dont les tests ratent
  void testsRates()
  {
    CPPUNIT_ASSERT( !(1 == 1) );
    CPPUNIT_ASSERT( 1 == 2 );
  }
};
```

On décide donc de dériver PseudoTest de TestCase et de créer 2 fonctions, testResussis et testsRates, la deuxième fonction ne contenant que des tests qui ratent. Ensuite, 3 macros sont utilisées, CPPUNIT_TEST_SUITE, CPPUNIT_TEST, CPPUNIT_TEST_SUITE_END, qui vont permettre de commencer l'inclusion des différentes fonctions à la suite de test, inclure les fonctions que l'on veut et de finir la série d'inclusion.

Nouvelle en-tête pour main.cpp

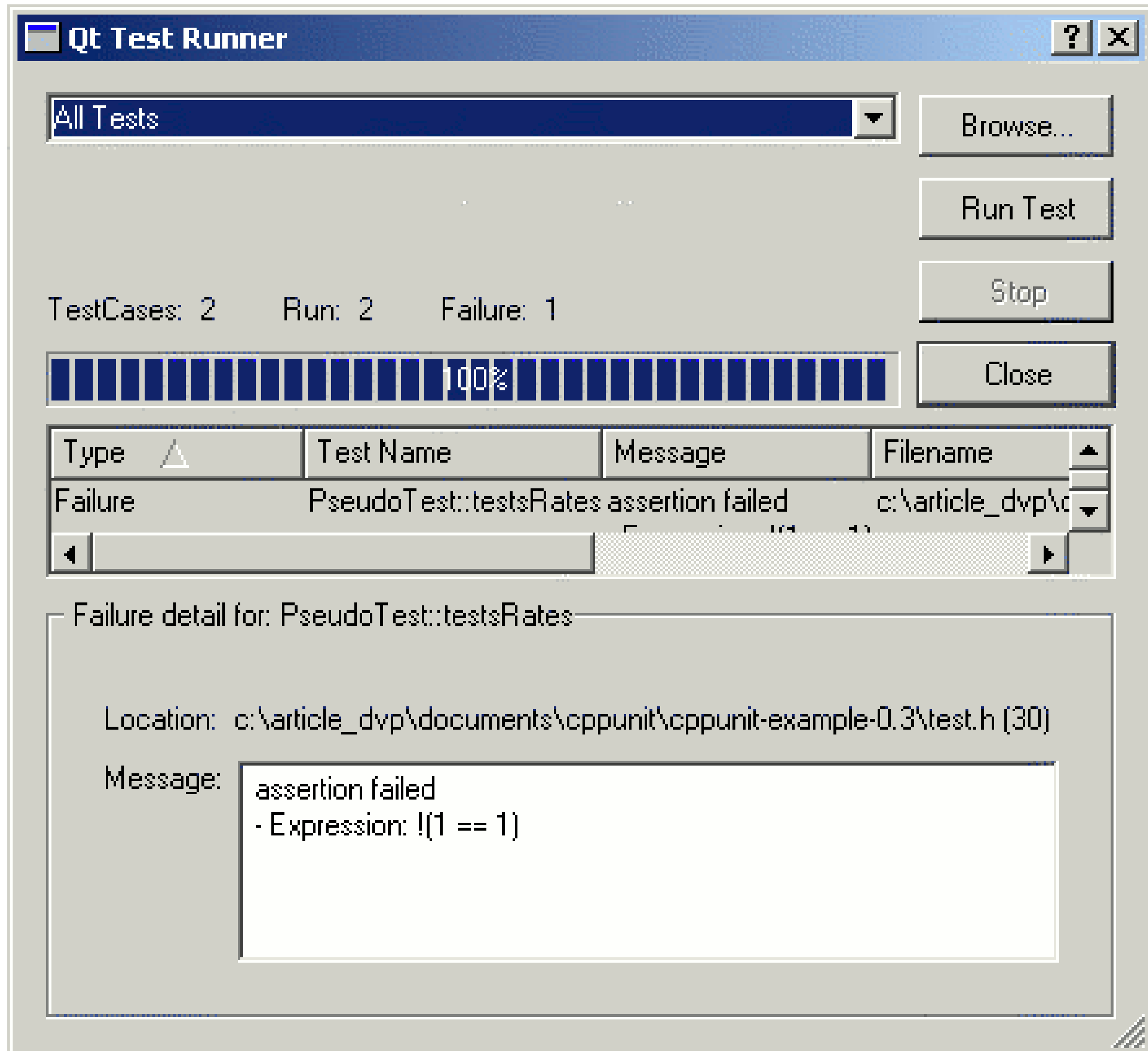
```
#include <cppunit/extensions/TestFactoryRegistry.h>
```

Modificatiosn dans le corps de main

```
        CppUnit::TestFactoryRegistry &registry = CppUnit::TestFactoryRegistry::getRegistry();
        runner.addTest( registry.makeTest() );
```

TestFactoryResgistry est le singleton qui contient les tests enregistrés par la macro CPPUNIT_TEST_SUITE_REGISTRATION. On peut ajouter d'autres tests tout de même à l'aide d'addTest. Le résultat est le suivant


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.




*Les tests exécutés en utilisant un singleton*

Les tests enregistrés au démarrage avec le singleton sont répertoriés avec leur arborescence - voir avec le bouton Browser - sous l'intitulé "All Tests".


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

# V - Plus de possibilités avec cppUnit

On a vu rapidement comment utiliser la bibliothèque cppUnit pour des tests simples, ainsi que certaines de ses capacités pour enregistrer les tests simplement, sans recompiler le fichier main.cpp et sans ajouter tous les en-têtes des tests dans ce fichier. Outre ces capacités, cppUnit propose d'autres choses, et utilisent certains patterns intéressants.

## V-A - Avantages de cppUnit

L'avantage indéniable, c'est qu'on a accès à tout, on peut tout faire. Des arborescences de test, des environnements, tester des plug-ins, ... Toutes ces capacités ont été testées et le design utilisé est propre, simple et efficace.

## V-B - Inconvénients de cppUnit

Les inconvénients, c'est plus difficile à trouver. On peut reprocher l'absence d'interface graphique "native", mais le plus gros point est que la mise en place et l'enregistrement des tests sont complexes, en même temps, on ne peut pas avoir le beurre et l'argent du beurre.

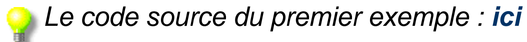

Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

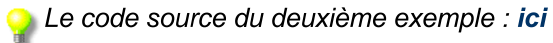
## Conclusion

cppUnit est un exemple d'utilisation des patterns, que ce soit le singleton, le composite, le décorateur, ... Disposer d'une bibliothèque de gestion des tests unitaires est primordial dans un projet de taille moyenne ou importante, permetant de s'assurer de la fiabilité du code écrit ou des modifications qu'on y apporterait pas la suite.


Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

## Téléchargements


*Le code source du premier exemple : **ici***


*Le code source du deuxième exemple : **ici***

*Le code source du troisième exemple : **ici***



Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Copyright © - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.