Tests unitaires et backtrace sous Windows et Linux

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

Date de publication : 29/01/2007

Tester son application de fond en comble nécessite l'utilisation d'une bibliothèque sur laquelle on peut se décharger. Plusieurs bibliothèques de tests existent, dont une partie orientée C++. La plus célèbre est sans doute cppUnit, dérivée de JUnit, en ligne de commande et utilisant intensivement de nombreux patterns. De l'autre côté, dans les plus légères et les plus simples, on peut trouver la bibliothèque de test de Qt, légère, simple, permettant juste de tester une classe.
Dans cet article, nous allons proposer une autre solution, à mi-chemin entre cppUnit et Qt Test. Elle est basée sur la bibliothèque Qt et peut avoir une interface graphique, ou pas. Outre les tests unitaires, il est aussi important de pouvoir récupérer des rapports d'erreurs complets de bêta testeurs ou tout simplement des utilisateurs finaux. Nous proposerons donc, dans une deuxième partie, une classe portable permettant de récupérer une trace plus ou moins complète du programme lors d'un bug.

Copyright © 2006 - 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 - La bibliothèque de tests unitaires

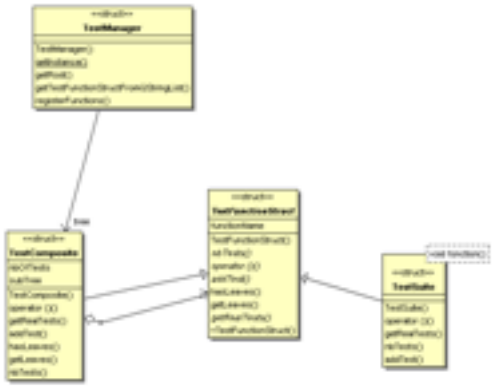

Diagramme de classes des tests unitaires

Afin de simplifier au maximum la gestion des tests, plusieurs facteurs ont été pris en compte :

- Structure arborescente des tests
  - Un test par feuille de l'arbre des tests
  - Possibilité de tester un sous-arbre
- Découplage de l'interface graphique et des tests eux-mêmes

En ce qui concerne le premier point, il suffit d'utiliser un pattern composite. La classe de base décrira n'importe quel noeud dans l'arborescence. Ensuite, une spécialisation en feuille permettra de lancer un test, tandis qu'une spécialisation en branche ou encore composite englobera plusieurs sous noeuds et sera responsable du test du sous-arbre.

Enfin, pour découpler efficament les tests d'une interface quelconque, on utilisera un pattern Registry qui sera responsable de l'enregistrement des données. Ce pattern, expliqué ici, utilisera la structure arborescente définie précédemment pour réaliser la bibliothèque complète.

I-A - Description des classes de la structure arborescente

I-A-1 - TestFunctionStruct

La classe de base de la structure arborescente est **TestFunctionStruct**. Elle définit, entre autres, le nom du test ainsi que plusieurs fonctions virtuelles surchargées par les classes filles.

- unsigned int nbTests() : retourne le nombre de tests dans le sous-arbre, utile pour l'interface graphique
- void operator()() : exécute le test
- void addTest(QStringList, TestFunctionStruct*) : ajoute un test défini par une liste de sous-chaînes
- bool hasLeaves() const {return false;} : indique si le test a des feuilles
- std::map<QString, TestFunctionStruct*>& getLeaves() : retourne la carte des noeuds fils - par défaut, cette fonction lève une exception -
- std::list<TestFunctionStruct*> getRealTests() : retourne la liste de tous les tests dans les sous-arbres

Copyright © 2006 - 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.

La fonction permettant de récupérer les tests dans un sous-arbre est traditionnellement implementée à l'aide du pattern Visiteur. Pour la simplicité de la bibliothèque, ce pattern n'a pas été utilisé, on retourne simplement une liste, celle-ci pourra être utilisée pour autre chose qu'effectuer les tests.

I-A-2 - TestSuite

Cette classe hérite publiquement de TestFunctionStruct, mais dépend d'un paramètre template. Cet argument permet de spécifier une fonction qui sera appelée et exécutée lors de l'appel à **operator()()**. Si vous préférez utiliser des foncteurs, c'est cette classe que vous devrez modifier/copier.

- Le constructeur : Outre l'initialisation de la classe parente avec le nom du test, la création d'une instance de TestSuite entraîne automatiquement son enregistrement dans le registre, le **TestManager**
- unsigned int nbTests() : retourne 1, car un test est défini ici.
- void operator()() : exécute la fonction
- std::list<TestFunctionStruct*> getRealTests() : retourne une liste à un seul élément

I-A-3 - TestComposite

Cette classe est la colle, le lien entre les différents tests.

- Le constructeur : Ce constructeur, très simple, initialise la classe mère ainsi que la variable interne nbTests à 0. En effet, au départ, il n'y a aucun test dans ce noeud.
- unsigned int nbTests() : retourne nbTests
- void operator()() : ne fait rien
- bool hasLeaves() const {return true;} : retourne true
- std::map<QString, TestFunctionStruct*>& getLeaves() : retourne la carte des noeuds fils qui stocke les sous-noeuds existants
- std::list<TestFunctionStruct*> getRealTests() : retourne une liste construite récursivement contenant tous les tests du sous-arbre
- void addTest(QStringList path, TestFunctionStruct* test) : cette fonction spécifique à **TestComposite** incrémente le nombre de tests dans l'arbre et ajoute récursivement le test dans le sous-arbre

Contrairement à l'exemple fourni du pattern Registry, il n'y a pas d'utilité à la recherche directe d'un élément dans le registre, donc cette fonction est inutile ; le test est directement l'élément enregistré dans l'arbre, aucun encapsuleur n'est utilisé, pour simplifier l'arborescence, et l'ajout de tests est fait de manière récursive, pour ajouter en modularité. Enfin, le nom complet du test est enregistré, une fois de plus contrairement à ce qui était proposé dans le pattern Registry, afin d'afficher facilement le nom complet du test.

I-B - Le registre

Implémenté sous la forme d'un singleton, cette classe est responsable de la gestion de la structure arborescente. On peut considérer que cette classe est un registre dans lequel on peut enregistrer et récupérer des informations hiérarchiques.

- Le constructeur : protégé, il crée le premier élément dans l'arbre des tests
- static TestManager& getInstance() : retourne le singleton
- TestComposite* getRoot() : retourne la racine de l'arbre des tests

Copyright © 2006 - 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.

- TestFunctionStruct* getTestFunctionStructFromQStringList(QStringList path) : retourne l'élément dans l'arbre décrit par le chemin path
- void registerFunctions(const QString& name, TestFunctionStruct* f) : est chargée d'enregistrer au bon endroit dans l'arbre le nouveau test. Agit récursivement à l'aide d'**addTests** disponible dans **TestComposite**

I-C - Proposer un test

Maintenant que les tests peuvent être enregistrés automatiquement, il est temps de créer une véritable fonction de test.

I-C-1 - Les tests d'assertion

Une assertion va tester une condition - par exemple une égalité ou une inégalité -. Elle utilisera les méthodes associées aux instances des classes comparées. Lorsque la comparaison échouera, on lancera une exception particulière indiquant le type d'erreur, la fonction où l'erreur est survenue, la ligne, ... En ce qui concerne ces derniers éléments, on utilisera une macro du préprocesseur, __FILE__, __LINE__ et __FUNCTION__, pour s'occuper du travail. Pour éviter d'avoir à écrire ces macros, on encapsule l'appel à la fonction de test dans une macro.

I-C-2 - La gestion de plusieurs tests

Lorsqu'on lance plusieurs tests, on peut obtenir une liste des tests à effectuer grâce à **getRealTests()**. Ensuite, on parcourt cette liste en lançant chaque test et en récupérant toutes les erreurs, celles attendues grâce à notre exception particulière, et les autres auxquelles on ne s'attendrait pas.

I-D - Lancer les tests

On propose de lancer les tests de plusieurs manières. La première, c'est de lancer tous les tests en ligne de commande, et d'afficher les résultats. L'autre solution est de proposer une interface graphique qui permet de sélectionner le sous-arbre à tester et reporter les erreurs et les échecs avec des indications complémentaires si besoin est.

I-E - Code d'exemple

Tout d'abord un exemple de test simple.

Exemple de tests

```
#include "test_register_function.h"
#include "test_functions.h"

/// Petite fonction de test
void petitTest()
{
   assertEqual(0U, 0U);
 assertDifferent(1U, 0U);
 assertNotEqual(1U, 0U);
 assertNotDifferent(0U, 0U);
 assertLess(2, 5);
 assertGreater(20U, 10U);
}

TestSuite<petitTest> TestSuitePerso("Petit:Test"); // On peut encapsuler cette déclaration dans une
 macro
```


Copyright © 2006 - 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.

Avec cela, il faudra un gestionnaire, donc soit un en ligne de commande, soit avec interface graphique.

I-E-1 - Version en ligne de commande

Pour l'utiliser, il suffit de compiler avec QCore(d)4.dll.

commandLine.cpp

```
#include <exception>
#include <fstream>

#include <cstdio>
#include <iostream>
#include <string>
#include "test_register_function.h"
#include "test_functions.h"

#ifdef Q_OS_WIN
#include "windows.h"

static CRITICAL_SECTION outputCriticalSection;
static HANDLE hConsole = INVALID_HANDLE_VALUE;
static WORD consoleAttributes = 0;

static const char *qWinColoredMsg(int prefix, int color, const char *msg)
{
    if (!hConsole)
        return msg;

    WORD attr = consoleAttributes & ~(FOREGROUND_GREEN | FOREGROUND_BLUE
              | FOREGROUND_RED | FOREGROUND_INTENSITY);
    if (prefix)
        attr |= FOREGROUND_INTENSITY;
    if (color == 32)
        attr |= FOREGROUND_GREEN;
    if (color == 31)
        attr |= FOREGROUND_RED | FOREGROUND_INTENSITY;
    if (color == 37)
        attr |= FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE;
    if (color == 33)
        attr |= FOREGROUND_RED | FOREGROUND_GREEN;
    SetConsoleTextAttribute(hConsole, attr);
    printf(msg);
    SetConsoleTextAttribute(hConsole, consoleAttributes);
    return "";
}

# define COLORED_MSG(prefix, color, msg) qWinColoredMsg(prefix, color, msg)
#else
# define COLORED_MSG(prefix, color, msg) "\033["#prefix";"#color"m" msg "\033[0m"
#endif

void iterateThroughTests(TestFunctionStruct* test)
{
  if(test->hasLeaves())
  {
    std::map<QString, TestFunctionStruct*>& leaves = test->getLeaves();
    for(std::map<QString, TestFunctionStruct*>::iterator it = leaves.begin(); it != leaves.end();
 ++it)
    {
      iterateThroughTests(it->second);
    }
  }
  else
  {
```


Copyright © 2006 - 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.

commandLine.cpp

```
      try
      {
        (*test)();
        std::cout << COLORED_MSG(0, 32, "PASS   ") <<  test->functionName.toStdString() << std::endl;
      }
      catch(const Tester::Thrown& failure)
      {
        std::cout << COLORED_MSG(0, 31, "FAIL!  ") <<  test->functionName.toStdString() << std::endl;
        std::cout << "* " << failure.explanation.toStdString() << std::endl;
      }
      catch(const std::exception& exception)
      {
        std::cout << COLORED_MSG(0, 31, "FAIL!  ") <<  test->functionName.toStdString() << std::endl
   << "   " << exception.what() << std::endl;
      }
      catch(...)
      {
        std::cout << COLORED_MSG(0, 37, "ERROR! ") <<  test->functionName.toStdString() << std::endl
   << "  Unknown error" << std::endl;
      }
    }
  }

  int main(int argc, char **argv)
  {

    TestManager& manager = TestManager::getInstance();
    iterateThroughTests(manager.getRoot());
    return EXIT_SUCCESS;
  }
```

I-E-2 - Version avec interface graphique

Ici, il faudra compiler avec **test_gui.h** et **test_gui.cpp**, le tout en liant avec QGui(d)4.dll ainsi que QCore(d)4.dll.

gui.cpp

```
  #define PACKAGE_STRING "Exemple"

  #include <exception>
  #include <fstream>

  #include <QtGui/QtGui>
  #include "testGui.h"

  int main(int argc, char **argv)
  {
   QApplication app(argc, argv);
   Tester::TestMainWindow mainWin(QString::fromAscii(PACKAGE_STRING));
   mainWin.show();
   return app.exec();
  }
```


Copyright © 2006 - 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 - Afficher l'état de la pile

Les méthodes pour afficher l'état de la pile sont différentes selon les systèmes d'exploitation.

Pour simplifier la gestion multiplatforme, on décide d'avoir une interface très simple. Il s'agira donc simplement d'un appel à une fonction indiquant que la trace de l'erreur sera sauvegardée dans un flux que l'on donnera en paramètre.

Il existe en C++ une fonction permettant de récupérer d'une exception non gérée, **set_exception_handler**. Malheureusement, elle ne permet pas de gérer les aborts, les erreurs de segmentation dans un code non C++, ...

II-A - Backtrace sous Linux

Sous Linux, il existe un mécanisme de signaux qui permettent d'appeler une fonction dès qu'il y a eu un problème. C'est ce mécanisme qui est utilisé pour cette partie.

La récupération de l'état de la pile sous Linux est relativement aisée, mais très limitée. En effet, seules les fonctions **backtrace** et **backtrace_symbol** sont apparemment disponibles. Ces deux fonctions récupèrent des chaînes de caractères préformatées avec le nom de la fonction, l'adresse, ... En revanche, pas de moyen immédiat pour récupérer les modules chargés, ou pour afficher chaque paramètre de chaque fonction appelée. Par défaut, vous pourrez constater que vous n'obtenez pas le nom des fonctions. En effet, pour cela, vous devez compiler avec **gcc** et l'option **-rdynamic**, sachant qu'à la fin vous devez obtenir des fichiers objets **ELF**, le format habituel sous Linux, mais pas sous Windows. Donc cette technique peut ne pas donner le nom des fonctions sous Windows avec MinGW.

Les prochaines étapes consisteront à améliorer cette partie de la récupération de la trace afin qu'elle soit plus complète.

II-B - Backtrace sous Windows

Sous Windows, le problème est plus simple et plus compliqué à la fois. Plus simple car il existe une bibliothèque dédiée à la récupération des informations - pile, paramètres, modules chargés, ... -, s'appelant **dbghelp.dll**, et il existe en plus un filtre permettant de gérer les exceptions non récupérées. Ce filtre permet d'afficher la fameuse boîte de dialogue sous Windows XP qui demande l'envoi ou non d'un rapport d'erreur.

En revanche, le point plus compliqué, c'est qu'il existe plusieurs versions de la même bibliothèque et que la version la plus récente pour Windows n'est pas forcément celle qui a le numéro de version le plus élevé, car l'équipe de Windows XP utilise des dates différentes... Ce qui fait qu'il vaut mieux installer le logiciel **WinDBG** qui contient entre autres la bibliothèque **dbghelp.dll** et a redistribuer pour avoir accès à l'API la plus récente.

De même, la récupération des infos est plus souple et donc plus complexe à mettre en place. On commencera par sauvegarder le nom des modules chargés avec leur taille dans une liste grâce à une fonction callback. Puis, on gèrera le cas délicat du parcours de la pile. Ce parcours est effectué par une fonction, **StackWalk64**, qui prend en paramètre des structures initialisées grâce à des informations données par le filtre des exceptions non récupérées, entre autres l'adresse de la pile, du compteur programme, ... Grâce à ce parcours récursif, on obtient chaque état de la pile pour la fonction à partir de la levée de l'exception.

Ensuite, il faut vérifier la validité de l'adresse, récupérer le nom de la fonction associée ainsi que le déplacement dans la fonction, le nom du fichier et la ligne, si possible. On pourrait aussi retourner les valeurs des paramètres des fonctions.

Copyright © 2006 - 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.

Naturellement, ces informations ne sont pas stockées dans le fichier - par exemple, le nom de la fonction, du fichier, la ligne, ... -, tout cela se trouve dans les fichiers **.pdb** qui devront être fournis avec l'exécutable.

II-C - Exemple d'utilisation

L'exemple d'utilisation est on ne peut plus simple.

```
#include <stdexcept>
#include <fstream>
#include "backtrace.h"

int main(int argc, char **argv)
{
  setExceptionHandler("test.txt");

  throw std::runtime_error("Test");
}
```

Et automatiquement un fichier *test.txt* est généré lors de la levée de l'exception. Pour moduler selon le pid, le dossier d'exécution, ... c'est à vous de voir.

Copyright © 2006 - 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

Grâce à ces 2 bibliothèques, vous ne pourrez plus fournir à vos clients des logiciels non testés avec beaucoup de bugs - il en reste toujours, mais d'autres bibliothèques fournissent des outils peut-être plus complets et adaptés à vos besoins -, et chaque client pourra vous donner une trace à peu près exhaustive de l'état du programme lors de son crash, pour vous aider à reproduire et à supprimer le bug.

Pour finir, n'oubliez pas, testez, testez et retestez vos applications.

Copyright © 2006 - 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

Voici la bibliothèque de tests unitaires ainsi que la classe backtrace disponible au téléchargement ici : **Télécharger**

On peut compiler rapidement la bibliothèque grâce à CMake en définissant **BUILD_TESTS_BINARY**, on directement avec votre outil favori.

Copyright © 2006 - 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 © 2006 - 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.