Stratégies de tests (unitaires)

G. Tremblay

Juin 2005, Novembre 2007

- *Test Your Software, or Your Users Will.*
- *Test Early. Test Often. Test Automatically.*
- *Coding Ain't Done 'Til all the Tests Run.*
- *Find Bugs Once.*

"The Pragmatic Programmer" [HT00]
A. Hunt & D. Thomas

1 Les différents types de tests (que faut-il tester?)

1.1 Les différents niveaux de tests [RK03]

- Tests unitaires :
  - Un test unitaire vérifie le bon fonctionnement *d'un module*, d'un *composant* indépendant.
  - Les tests unitaires forment la *fondation* sur laquelle repose l'ensemble des activités de tests : il est inutile de tester l'ensemble du système si *chacun* des modules n'a pas été testé à fond.
  - Note : les tests unitaires sont discutés plus en détail ultérieurement.
- Tests d'intégration :
  - Les tests d'intégration vérifient que les principaux *sous-systèmes* fonctionnent correctement, c'est-à-dire que les différents modules qui composent un sous-système donné sont correctement *intégrés* ensemble.
  - Ces tests peuvent être vus simplement comme une forme de tests unitaires, l'unité étant alors un *groupe (cohésif) de modules* plutôt qu'un unique module.
- Tests de système :
  - Les tests de système vérifient le fonctionnement du système dans son ensemble, en termes des fonctionnalités attendues du système.
  - Ces tests débutent lorsque le système dans son ensemble est prêt, ou lorsque des sous-ensembles importants de fonctionnalités sont prêts (approche itérative et incrémentale).
- Tests d'acceptation :
  - Les tests d'acceptation sont des tests finaux effectués lorsque le système est prêt à être déployé, donc juste avant qu'il soit livré et officiellement installé.
  - Objectif = vérifier que le logiciel est prêt à être utilisé par les véritables usagers, pour effectuer les tâches et fonctions pour lesquelles le système a été développé.

  - Deux types particuliers de tests d'acceptation
    * Tests *alpha* : tests effectués dans l'environnement de développement, par des développeurs et des clients, pour vérifier que le système est une réalisation *acceptable* des spécifications et exigences.
    * Tests *beta* : tests effectués dans un certain nombre de sites clients, qui vont utiliser le système et signaler les problèmes qu'ils rencontrent.

1.2 Autres types de tests [HT00]

- Tests de performance ou tests de stress : ces tests vérifient que le système fonctionne correctement dans un environnement réel fonctionnant à pleine charge.

  Peuvent nécessiter des outils spéciaux de tests (simulateurs de charge).

- Tests d'utilisabilité : ces tests peuvent être vus comme une forme de tests d'acceptation, où en plus de vérifier si les fonctionnalités désirées sont correctement livrées, on vérifie aussi si le logiciel est convivial, facile d'utilisation, etc.

1.3 Tests spécifiques à l'étape de construction de logiciels

Les tests suivants sont ceux qui s'appliquent plus spécifiquement à l'étape de construction des logiciels :

⋆ Tests unitaires

- Tests d'intégration

Tests unitaires

- Les cas de tests pour les tests unitaires d'un module sont définis relativement au *contrat* que doit satisfaire ce module :
  - Quelles sont les pré-conditions qui régissent l'utilisation de ce module?
  - Quelles sont les post-conditions qu'assurent l'exécution de ce module?
  - Quelles sont les exceptions signalées par ce module en présence de cas spéciaux ou anormaux?

- Les tests unitaires pour un module sont intimement liés à ce module. En fait, de nos jours, on considère que le *code source* pour un module consiste *autant* dans le code réalisant ce module que dans le code effectuant les *tests unitaires* de ce même module, donc :

  Code source d'un module = code exécutable + code pour les tests

  En d'autres mots, les cas de tests sont un livrable aussi important que le code lui-même (et il faut donc s'assurer de bien intégrer les deux).

- Autre avantage des tests unitaires intégrés au code : fournissent des exemples illustrant la façon d'utiliser le module.

2 Les principales activités liées aux tests (comment faut-il tester?)

2.1 La définition des tests

- La spécification des jeux d'essai [RK03] :
  - Données *typiques* et réelles : proviennent de sources réelles, de systèmes semblables, du problème réel qu'on veut automatiser.
  - Données synthétiques : données générées de façon artificielle (par exemple, obéissant à certaines contraintes statistiques), lorsqu'une grande quantité de données doit être traitée.
- La mise en oeuvre des tests à l'aide d'échafaudages de tests :
  - Analogie avec la construction d'édifices : lorsqu'on construit un édifice, il y a beaucoup de travail de mise en place et de *construction* qui doit être effectué pour pouvoir faire le véritable travail de construction lui-même, travail supplémentaire dont le résultat n'apparaît pas directement dans le produit final, par exemple : montage et démontage d'échafaudages, construction de cadres et moules pour couler le ciment, mise en place de structures temporaires, etc.
  - On retrouver deux types d'éléments lors de la construction d'échafaudages logiciels [RK03] :
    - Modules de remplacement (*stubs*, modules *bidons*) : composants qui sont développés pour simuler (temporairement) le comportement de modules qui ne sont pas disponibles — par exemple, parce qu'ils n'ont pas encore été développés ou parce que leur utilisation dans le cadre de tests est trop complexe ou trop lourd (par exemple, accès à un réseau).
      Deux types de modules de remplacement :
      1. Module simple de remplacement : retourne une valeur prédéfinie.
      2. Module complexe de remplacement : fait des traitements plus complexes, pouvant aller jusqu'à tenter de simuler le comportement du composant ou système remplacé.
    - Modules *pilotes* (*drivers*) : composants qui simulent ou remplacent des composants de haut niveau (*upstream*) et qui injectent des données dans le système.
      Deux types de modules pilotes :
      1. Pilote qui simule et contrôle les interactions externes avec l'application testée.
      2. Pilote qui fournit une interface personne–machine qui n'a pas encore été développée.

2.2 L'exécution des tests

Les différents cas de tests, une fois définis, doivent pouvoir être exécutés sur le logiciel qu'on veut tester.

- Exécution manuelle : acceptable seulement pour un logiciel simple, où on est certain que les tests n'auront pas besoin d'être exécutés souvent.
- Exécution à l'aide de programmes et de *scripts de tests*
  - Écriture de programmes dédiés à l'exécution des diverses séries de tests.
  - Utilisation de `make` et de `makefiles`.
  - Utilisation de *shell scripts* lorsque plusieurs de ces programmes sont requis.
- Utilisation d'outils d'exécution automatique (cadres de tests) : cf. plus bas.

2.3 L'importance de l'exécution automatique des tests

- Il est important de s'assurer, lorsqu'une ou plusieurs modifications ont été effectuées à un module ou au programme, que rien n'a été brisé, que ce qui fonctionnait correctement auparavant fonctionne toujours après que les modifications ont été faites. On parle alors de *tests de régression* :[1]

  Old tests never die, they just become regression tests

- Il faut pouvoir tester *souvent*, aussitôt que des changements significatifs ont été effectués à un module ou au programme :
  - pour assurer que tout fonctionne bien malgré les changements effectués (tests de régression) ;
  - pour mieux localiser la source du problème lorsque ça ne fonctionne pas : si on effectue les tests après avoir fait de nombreux ajouts ou modifications, alors il est plus difficile de savoir d'où vient le problème.

- Le plus tôt on trouve les problèmes et erreurs (les *bogues*), le plus facile il devient de les localiser et de les corriger ⇒ "*Code a little, test a little*".

- Le travail de programmation et construction d'un composant logiciel n'est pas terminé tant que tous les tests appropriés n'ont pas été *définis et exécutés* :
  - Les différents modules réussissent correctement les tests unitaires associés.
  - Les différents composants et sous-systèmes réussissent correctement les tests (d'intégration, unitaire de haut niveau) associés.
  - Le système dans son ensemble réussit correctement les tests d'acceptation.

- Si une erreur (un *bogue*) est détectée par un humain une première fois, il ne faut surtout pas que cette même erreur réapparaisse ultérieurement. Donc, si une erreur est détectée, il est crucial qu'un cas d'essai approprié soit ajouté aux cas de tests, et que ce cas de test soit ré-exécuté à chaque fois que des modifications sont apportées au système. [HT00]

3 Les cadres d'exécution automatique de tests

- Il existe de nombreux outils qui permettent d'automatiser l'exécution des tests unitaires, qui facilitent le développement de tels tests et leur association au code testé, et ce peu importe le langage de programmation utilisé. Ces outils sont appelés des *cadres de tests* (*tests framework*).

- L'outil de ce type le plus connu est `JUnit`, popularisé par les promoteurs de l'approche XP (*eXtreme Programming*).

- Des cadres équivalents existent pour d'autres langages :
  - `xUnit` = famille de cadres de tests avec variantes pour divers langages[2]
    (Ada, C, C++, Eiffel, Java, Perl, Python, etc.).
  - Divers cadres de tests pour le langage C existent, par exemple : `GUNit`, `Check`, `MinUnit`, `CUnit`, `CuTest`.

[1] Certains auteurs disent plutôt "*tests de non régression*".
[2] `http://www.xprogramming.com/software.htm`

- La caractéristique la plus importante, et ce pour tous ces cadres de tests :
  - On utilise des *assertions*, plutôt que de vérifier des résultats textuels :

    ```
    assertEquals( expectedResult, value )
    assertEquals( expectedResult, value, precision )
    assertTrue( booleanExpression )
    assertNotNull( reference )
    etc.
    ```

    ⇒ Aucun résultat n'est produit si le test ne détecte pas d'erreur.
- D'autres caractéristiques de tous ces cadres de tests :
  - Supportent l'exécution automatique des tests
    (support pour les tests de régression, pour l'intégration continue, etc.).
  - Permettent l'organisation structurée des jeux d'essais
    (cas de tests, suites de tests, classes de tests).

    Plus spécifiquement :
    * Un *cas de tests* porte sur une fonctionnalité limitée, une instance particulière d'un appel de méthode ou procédure.
    * Une *suite de tests* regroupe un certain nombre de cas de tests, qui représentent diverses instances liées à une même procédure ou groupe de procédures liées entre elles.
    * Une *classe de tests*, dans un langage objet, regroupe l'ensemble des suites de tests permettant de tester l'ensemble des fonctionnalités du module, c'est-à-dire de la classe.
  - Fournissent des mécanismes pour la construction d'échafaudages de tests (par exemple, `setUp`, `tearDown` en `JUnit`).
  - Fournissent des mécanismes permettant d'analyser les résultats de l'exécution des tests, ainsi que signaler clairement les problèmes.
  - Fournissent (parfois) des bibliothèques pour les tests d'objets complexes
    (par ex., *mock objects* pour objets `Output`, servlets, BDs, GUIs, etc.).

3.1 Le cadre de tests `cutest` pour le langage `C`

À compléter.

Cf. le site `http://www.xprogramming.com/software.htm` pour des informations sur ce cadre de tests et d'autres cadres.

4 Développement dirigé par les tests

Approche proposée initialement par les promoteurs de la *Programmation eXtrême* (XP = *eXtreme Programming*)

- *Test first*
  - Les cas de tests devraient être développés et écrits (codés) *avant* le code lui-même.
  - Du nouveau code ne devrait jamais être écrit s'il n'y a pas déjà un cas de test qui montre que ce code est nécessaire.

    Never write a line of functional code without a broken test case. (Beck, 2001)

- Approche TDD décrite en détail, et de façon indépendante de l'approche XP, dans "*Test-Driven Development — By Example*" [Bec03].
  - Objectif premier = *Clean code that works*
  - Les deux règles de base sont les suivantes :
    1. Écrire du nouveau code que s'il existe un test automatique qui ne fonctionne pas.
    2. Éliminer la redondance (*duplication*).
  - Les principales implications de l'approche TDD :
    * La conception d'un logiciel doit se faire de façon organique, avec le code exécutable qui fournit continuellement du *feedback*.
    * Les principaux tests doivent être écrits par les développeurs eux-mêmes, pour qu'ils soient disponibles rapidement.
    * L'environnement de développement doit pouvoir fournir une réponse rapide aux changements, petits et grands.
    * Le logiciel doit être conçu à l'aide d'un grand nombre de composants (modules) hautement cohésifs et faiblement couplés, de façon à faciliter le développement des tests.
  - Ordre d'exécution des tâches de programmation dans l'approche TDD :
    1. Rouge — Écrire un petit test qui ne fonctionne pas, et qui peut même ne pas compiler correctement au début.
    2. Vert — Faire fonctionner le test rapidement, quitte à écrire du code pas élégant.
    3. Remodelage (*refactoring*) — Éliminer la redondance, améliorer le code, en s'assurant que tous les tests déjà développés continuent de fonctionner correctement.

    Une idée maîtresse sous-jacente à l'approche TDD est celle *d'intégration continue* :
    * Lorsque du code est ajouté au système, on doit toujours s'assurer que le système dans son ensemble fonctionne correctement, c'est-à-dire, que *l'ensemble des tests fonctionnent correctement*.
    * À la fin de chaque journée pour des gros projets, et même plusieurs fois par jour pour des petits projets, on devrait pouvoir livrer une version *testée et fonctionnelle*, qui intègre les changements récents (qui ont évidemment été testés et qui fonctionnent correctement) (notion de *daily build* popularisée, entre autres, par Microsoft).

      L'intégration continue requiert les éléments suivants [FF04] :
      · Que l'ensemble du code source soit conservé dans un reposoir central et que n'importe qui puisse obtenir la version la plus récente (ou d'autres versions plus anciennes).
      · Que le processus de construction d'un *build* puisse se faire de façon automatique à partir du code source.
      · Que l'ensemble des tests puissent s'exécuter de façon automatique et rapide.
      · Que l'on soit assuré de pouvoir obtenir, en tout temps, la meilleure version exécutable.

A Un exemple de programme de tests pour un type abstrait

Dans ce qui suit, nous allons présenter un exemple de programme de test pour un module définissant un type abstrait pour des fractions (collection de valeurs). Plus précisément, nous allons illustrer *l'utilisation d'assertions* pour tester un module (tests unitaires). Nous n'utiliserons pas l'un des cadres de tests disponibles pour le langage C. Plus simplement, nous utiliserons les assertions disponibles dans le langage C, i.e., nous utiliserons la macro `assert`.

Fichier d'interface 1 Un module définissant un type abstrait (collection de valeurs) pour des fractions : `fractions.h`

```
#ifndef FRACTIONS_H
#define FRACTIONS_H

typedef struct {
  int numerateur;
  int denominateur;
} Fraction;

Fraction fraction( int num, int den );         // PRECONDITION: den != 0

Fraction plus   ( Fraction f1, Fraction f2 );
Fraction moins  ( Fraction f1, Fraction f2 );
Fraction fois   ( Fraction f1, Fraction f2 );
Fraction inverse( Fraction f );                // PRECONDITION: !estZero(f1)

int      estZero( Fraction f );
int      egales ( Fraction f1, Fraction f2 );

char*    chaine ( Fraction f );                // Equivalent a "toString".

#endif
```

Le fichier d'interface 1 présente la spécification d'interface pour ce module.[3] La mise en oeuvre est présentée en deux parties : Mise en oeuvre 1 et 2.

Dans cette mise en oeuvre, on remarque que des assertions ont été utilisées pour assurer que les pré-conditions des opérations `fraction` et `inverse` sont bien satisfaites. On remarque aussi que, dans chacune des opérations, du code a été ajouté pour le débogage, c'est-à-dire, pour imprimer une trace (utilisant `printf`) qui indique, au début de l'exécution d'une opération, le nom de l'opération appelée et les arguments transmis. En fait, ce code ne sera véritablement présent dans le programme exécutable final que si cela est explicitement requis lors de la compilation. Le fichier d'interface 2 donne la définition de la macro `DEBUG`. Lorsque le symbole `DEBUG_TRACE` n'est pas défini, alors ces traces ne s'exécuteront donc pas. La définition de ce symbole s'effectue, ou non, dans le fichier `makefile`, lequel est présenté dans le fichier `makefile` 1.

Dans le cadre de cet exemple simple, le fichier `makefile` ne comprend qu'un seul programme associé à l'exécution de tests (commande "`make tests`") : le programme `tester-fractions.c`. Le code pour ce programme est présenté dans le programme 1.

Le premier point important à souligner pour ce programme est que si tous les cas de tests s'exécutent sans problème, alors un feedback très minime est produit, à savoir un simple "`OK`". Lorsqu'on utilise des assertions pour définir des cas de tests, on ne produit des sorties et messages que s'il y a un problème, que s'il y a un cas de test qui ne fonctionne pas.

[3] Soulignons qu'il sagit d'une version un peu plus détaillée de l'exemple présenté dans le chapitre sur la conception.

```
#include "fractions.h"
#include "debuggage.h"

#include <assert.h>

Fraction fraction( int num, int den )
{
  DEBUG( printf( "fraction( %d, %d )\n", num, den ); );
  assert( den != 0 && "Dans fraction: le denominateur doit etre different de 0" );

  Fraction res;
  res.numerateur = num;
  res.denominateur = den;
  return( res );
}


Fraction plus( Fraction f1, Fraction f2 )
{
  DEBUG( printf( "plus( %s, %s )\n", chaine(f1), chaine(f2) ); );

  Fraction res;
  res.numerateur = f1.numerateur*f2.denominateur + f2.numerateur*f1.denominateur;
  res.denominateur = f1.denominateur * f2.denominateur;
  return( res );
}


Fraction moins( Fraction f1, Fraction f2 )
{
  DEBUG( printf( "moins( %s, %s )\n", chaine(f1), chaine(f2) ); );

  Fraction res;
  res.numerateur = f1.numerateur*f2.denominateur - f2.numerateur*f1.denominateur;
  res.denominateur = f1.denominateur * f2.denominateur;
  return( res );
}


Fraction fois( Fraction f1, Fraction f2 )
{
  DEBUG( printf( "fois( %s, %s )\n", chaine(f1), chaine(f2) ); );

  Fraction res;
  res.numerateur = f1.numerateur*f2.numerateur;
  res.denominateur = f1.denominateur * f2.denominateur;
  return( res );
}
```

Fichier de mise en oeuvre 1: La mise en oeuvre du module pour les fractions : `fractions.c` (première partie)

```
Fraction inverse( Fraction f )
{
  DEBUG( printf( "inverse( %s )\n", chaine(f) ); );
  assert( !estZero(f) && "Dans inverse: la fraction doit etre differente de 0" );

  Fraction res;
  res.numerateur = f.denominateur;
  res.denominateur = f.numerateur;
  return( res );
}


int estZero( Fraction f )
{
  DEBUG( printf( "estZero( %s )\n", chaine(f) ); )

  return( f.numerateur == 0 );
}


int egales( Fraction f1, Fraction f2 )
{
  DEBUG( printf( "egales( %s, %s )\n", chaine(f1), chaine(f2) ); );

  return( f1.numerateur * f2.denominateur == f2.numerateur * f1.denominateur );
}



#define MAX_CARS 1000

char* chaine ( Fraction f )
{
  DEBUG( printf( "chaine( %d/%d )\n", f.numerateur, f.denominateur ); );

  char tampon[MAX_CARS];

  sprintf( tampon, "%d/%d%c", f.numerateur, f.denominateur, '\0' );

  int ln = strlen( tampon );
  char *res = (char *) malloc(ln+1);
  strcpy( res, tampon );
  return( res );
}
```

Fichier de mise en oeuvre 2: La mise en oeuvre du module pour les fractions : `fractions.c` (deuxième partie)

Fichier d'interface 2 Fichier définissant la macro `DEBUG` pour l'impression de trace lors du débgogage

```
#ifndef DEBUGGAGE_H
#define DEBUGGAGE_H


#ifdef DEBUG_TRACE

/*
 * La macro suivante permet d'encapsuler du code ne devant etre genere que pour
 * l'epuration, i.e., que si DEBUG_TRACE est defini. Autrement
 * (#else: cf. plus bas), le code n'est pas inclus dans le programme.
 */
#define DEBUG(x) x

#else

/*
 * Si DEBUG_TRACE n'est pas defini, le code encapsule par les appels
 * a la macro DEBUG ne sera pas inclus dans le programme compile.
 */

#define DEBUG(x)

#endif

#endif
```

L'autre point important à signaler est que le programme consiste uniquement en des appels à des opérations (ici, des fonctions) et à des tests, via des assertions, pour s'assurer que les résultats retournés sont bien valides. L'utilisation d'un `makefile` signifie donc qu'à chaque fois que des modifications seront effectuées au module `fractions`, on pourra s'assurer que tout fonctionne correctement en tapant simplement "`make tests`", auquel cas on aura le résultat suivant si le programme passe toujours tous les tests :

```
% make tests
gcc -c  tester-fractions.c
gcc -c  fractions.c
gcc -o tester-fractions.out tester-fractions.o fractions.o
tester-fractions.out
OK
```

Dans le cas contraire, on aura un message qui aura l'allure suivante, l'exécution du programme se terminant aussitôt qu'un premier cas test invalide est exécuté :

```
% make tests
gcc -c  tester-fractions.c
gcc -c  fractions.c
gcc -o tester-fractions.out tester-fractions.o fractions.o
tester-fractions.out
tester-fractions.out: tester-fractions.c:21: main:
   Assertion 'egales( plus(fraction(1, 1), fraction(10, 10)), fraction(2, 1) )' failed.
make: *** [tests] Abandon
```

Il est important de mentionner que si un cadre de tests approprié avait été utilisé, tous les cas de tests seraient exécutés, et ce même si un cas de test incorrect était rencontré. Le

```
.SUFFIXES : .o .c

CC=gcc

#DEBUG_TRACE=-DDEBUG_TRACE
DEBUG_TRACE=

.c.o :
	$(CC) -c $(DEBUG_TRACE) $<

OBJ=tester-fractions.o fractions.o


tester-fractions.out : $(OBJ)
	$(CC) -o tester-fractions.out $(OBJ)

tests: tester-fractions.out
	tester-fractions.out


fractions.o : debuggage.h fractions.h fractions.c

clean :
	rm -f *.o core
	rm -f *.out
	rm -f *~
	rm -f *.h.gch
```

Fichier `makefile` 1: Fichier `makefile` pour la compilation (avec ou sans trace) et l'exécution automatique du programme de tests pour les fractions

```
#include "fractions.h"
#include <assert.h>

int main()
{
  // Tests estZero et egales.
  assert( estZero(fraction(0, 1)) );
  assert( estZero(fraction(0, 105)) );
  assert( egales(fraction(0, 1), fraction(0, 105)) );

  assert( !estZero(fraction(1, 1)) );
  assert( !estZero(fraction(105, 105)) );
  assert( egales(fraction(1, 1), fraction(105, 105)) );

  assert( !estZero(fraction(1, 1)) );
  assert( !estZero(fraction(10, 105)) );
  assert( !egales(fraction(1, 1), fraction(10, 105)) );


  // Tests plus.
  assert( egales( plus(fraction(1, 1),   fraction(10, 10)), fraction(2, 1) ) );
  assert( egales( plus(fraction(1, 100), fraction(1, 10)), fraction(11, 100) ) );

  // Tests moins.
  assert(  estZero( moins( fraction(10, 19), fraction(10, 19) ) ) );
  assert(  estZero( moins( fraction(1, 1), fraction(105, 105) ) ) );
  assert( !estZero( moins( fraction(1, 10), fraction(105, 105) ) ) );
  assert(  egales( moins(fraction(11, 14), fraction(15, 14)), fraction(-2, 7) ) );

  // Tests inverse.
  assert( egales( fraction(15, 10), inverse(fraction(10, 15)) ) );
  assert( egales( fraction(15, 10), inverse(inverse(fraction(15, 10))) ) );

  // Tests fois.
  assert(  egales( fois(fraction(11, 14), fraction(15, 14)), fraction(15*11, 14*14) ) );
  assert(  egales( fois(fraction(0, 14), fraction(15043, 14)), fraction(0, 2) ) );
  assert(  egales( fois(fraction(3, 19), inverse(fraction(3, 19))), fraction(1, 1) ) );
  assert(  egales( fois(fraction(1, 1), fois(fraction(2, 2),
                                        fois(fraction(3, 3), fraction(4, 4)))),
                   fraction(5, 5) ) );

  // Tests chaine.
  assert( strcmp( chaine(fraction(1, 2)), "1/2" ) == 0 );
  assert( strcmp( chaine(fraction(10, 20)), "10/20" ) == 0 );
  assert( strcmp( chaine(fraction(0, 2)), "0/2" ) == 0 );


  // Feedback pour indiquer que tout a ete OK.
  printf( "OK\n" );

  // On signale la fin du programme sans probleme.
  return( 0 );
}
```

Programme 1: Programme de tests pour le module `fractions` : `tester-fractions.c`

résultat indiquerait alors les divers cas de tests n'ayant pas fonctionné, ainsi qu'un sommaire de l'ensemble de l'exécution.

```
% make tests
mpd  -s matrices.mpd
mpd  -b matrices.mpd
mpd  -b tests-matrices.mpd
mpdl  -o a.out Matrices OpsAuxiliaires MPDUnit TestsMatrices
a.out
....E....
Termine en 5 ms

Il y a eu 4 assertions incorrectes:
 1) "Additionner"::"Test additionner"
     #1: assertTrue incorrect
 2) "Additionner"::"Test additionner"
     #2: assertTrue incorrect
 3) "Additionner"::"Test additionner"
     #3: assertTrue incorrect
 4) "Additionner"::"Test additionner"
     #4: assertTrue incorrect

ECHECS!!
5 suite(s), 9 test(s), 33 assertion(s), 1 cas echoue(s), 4 assertions incorrecte(s)
```

Figure 1: Exemple d'exécution d'une série de cas de tests par un cadre de tests (pour le langage MPD)

Par exemple, un cadre de tests (développé par l'auteur) pour le langage MPD produirait un résultat tel que celui présenté à la figure 1 (pour un programme testant un module de manipulation de matrices).

B Un exemple de programme de tests pour un type abstrait : Version avec `MiniCUnit`

Dans cette section, nous allons reprendre le programme de tests de la section précédente, mais cette fois en utilisant un *cadre de tests*. Plus précisément, nous allons utiliser le cadre de tests unitaires, spécifique au langage C, appelé `MiniCUnit` [TMSZ07]. Le code source, ainsi que des exemples, est disponible à l'adresse suivante : `http://www.info2.uqam.ca/~tremblay/INF3135/Liens/MiniCUnit.tar.gz`.

Le fichier d'interface 3 présente les extraits clés de ce cadre de tests — «`[...]`» indique une partie omise.

Un programme de tests utilisant le cadre de tests `MiniCUnit` a l'allure générale présentée dans le programme 2 : on fait un (ou plusieurs) appel(s) à `executerSuiteDeTests`, suivi d'un appel à `sommaireDeTests` pour obtenir et imprimer une chaîne qui représente le sommaire de l'exécution des diverses suites et divers cas de tests. Ce sommaire indique différentes informations quant au nombre de suites de tests exécutées, au nombre de cas de tests, d'assertions, etc.

Ensuite, le fichier définissant le programme de tests définit les différents cas de tests. Ces cas de tests peuvent utiliser l'une ou l'autre des routines suivantes pour exprimer des assertions :

- `void assertVrai( bool test, char* message )` : Vérifie qu'une expression booléenne est vraie.
- `void assertFaux( bool test, char* message )` : Vérifie qu'une expression booléenne est fausse.
- `void assertChainesEgales( char* ch1, char* ch2, char* message )` : Vérifie que deux chaînes de caractères sont identiques.
- `void assertEntiersEgaux( int v1, int v2, char* message )` : Vérifie que deux entiers sont égaux.

Dans tous les cas, rien n'est affiché si l'assertion est vérifiée. Dans le cas contraire, la chaîne indiquée comme dernier argument (`message`) est affichée sur `stdout`, avec dans certains cas des informations supplémentaires (dans les deux derniers cas, les valeurs des chaînes ou des entiers ayant été comparés sont affichées).

Finalement, toujours dans le programme de tests, la ou les suites de tests comme telles sont définies (`SUITE_DE_TESTS`), et ce en indiquant explicitement, avec `ajouterCasDeTest`, chacun des cas de tests qui doivent être associés à cette suite.[4]

Le programme 3 présente (des extraits d')une version équivalente du programme de tests 1, mais exprimeée cette fois à l'aide de `MiniCUnit` — le code complet est disponible avec le logiciel `MiniCUnit`.

Voici un exemple de résultat produit par l'exécution de ce programme de tests dans le cas où tous les tests s'exécutent sans erreur :

```
*** Sommaire: 1 suites executees; 6 tests executes, 0 tests echoues;
    24 assertions evaluees, 0 assertions echouees
```

[4] Signalons qu'il est possible de définir plusieurs suites de tests, auquel cas l'ensemble des informations pour ces diverses suites vont apparaître dans le sommaire.

Fichier d'interface 3 Extraits du fichier d'interface pour le cadre de tests `MiniCUnit` : `MiniCUnit.h`

```
/** Mini cadre de tests unitaires pour le langage C.

    Créé à l'hiver 2006 dans le cadre du projet FDP.  Initialement inspiré du
    cadre de tests MinUnit [...]: http://www.jera.com/techinfo/jtns/jtn002.html
    [...]
    @name MiniCUnit
    @author Guy Tremblay
*/

[...]

#ifndef MINI_CUNIT_H
#define MINI_CUNIT_H

[...]

/// Type pour le résultat d'un test
typedef void CasDeTest;

/// Type pour le résultat d'une suite de tests
typedef void SuiteDeTests;

[...]

/// Macro pour débuter la définition d'un nouveau cas de test.
#define CAS_DE_TEST( cas ) [...]

/// Macro pour terminer la définition d'un cas de test.
#define FIN_CAS_DE_TEST [...]

/// Macro pour débuter la définition d'une suite de tests.
#define SUITE_DE_TESTS( suite ) [...]

/// Macro pour terminer la définition d'une suite de tests.
#define FIN_SUITE_DE_TESTS [...]

[...]
void ajouterCasDeTest( CasDeTest (*test)() );

[...]
void executerSuiteDeTests( SuiteDeTests (*suite_de_tests)() );

[...]
void assertVrai( bool test, char* message );

[...]
void assertFaux( bool test, char* message );

[...]
void assertChainesEgales( char* ch1, char* ch2, char* message );

[...]
void assertEntiersEgaux( int v1, int v2, char* message );

[...]
char* sommaireDeTests();
#endif
```

```
#include "MiniCUnit.h"

static SuiteDeTests suite1();

int main( int argc, char *argv[] )
{
  executerSuiteDeTests( suite1 );

  printf( "%s", sommaireDeTest() );
  return 0;
}

// Definitions de divers cas de tests.
CAS_DE_TEST( test1 )
  ...
FIN_CAS_DE_TEST
    .
    .
    .
CAS_DE_TEST( testK )
  ...
FIN_CAS_DE_TEST


// Defintion d'une suite de tests.
SUITE_DE_TESTS( suite1 )
  ajouterCasDeTest( test1 );
  ...
  ajouterCasDeTest( testK );
FIN_SUITE_DE_TESTS
```

Programme 2: Allure générale d'un programme de tests utilisant `MiniCUnit`.

```
#include "MiniCUnit.h"
[...]
static SuiteDeTests suiteFractions();

int main()
{
  executerSuiteDeTests( suiteFractions );
  printf( "%s", sommaireDeTests() );
  return 0;
}

// Quelques procedures auxiliaires, pour simplifier le code des tests.
void assertEstZero( Fraction f1, char* msg )
{ assertVrai( estZero(f1), msg ); }

[...]

void assertFractionsEgales( Fraction f1, Fraction f2, char* msg )
{ assertVrai( egales(f1, f2), msg ); }

CAS_DE_TEST( tests_estZero_et_egales )
  char* MSG_EST_ZERO = "Fraction est zero";
  char* MSG_PAS_ZERO = "Fraction pas zero";
  char* MSG_PAS_EGALES = "Fractions pas egales";

  assertEstZero( fraction(0, 1), MSG_PAS_ZERO );
  assertEstZero( fraction(0, 105), MSG_PAS_ZERO );
  assertFractionsEgales( fraction(0, 1), fraction(0, 105), MSG_PAS_EGALES );
  [...]
FIN_CAS_DE_TEST

CAS_DE_TEST( tests_plus )
  char* MSG = "Resultat incorrect pour plus";

  assertFractionsEgales( plus(fraction(1, 1), fraction(10, 10)), fraction(2, 1), MSG );
  assertFractionsEgales( plus(fraction(1, 100), fraction(1, 10)), fraction(11, 100), MSG );
FIN_CAS_DE_TEST

CAS_DE_TEST( tests_moins )
  [...]
FIN_CAS_DE_TEST

CAS_DE_TEST( tests_inverse )
  [...]
FIN_CAS_DE_TEST

CAS_DE_TEST( tests_fois )
  [...]
FIN_CAS_DE_TEST

CAS_DE_TEST( tests_chaine )
  char* MSG = "Resultat incorrect pour chaine";

  assertChainesEgales( chaine(fraction(1, 2)),   "1/2",   MSG );
  [...]
FIN_CAS_DE_TEST

SUITE_DE_TESTS( suiteFractions )
     ajouterCasDeTest( tests_estZero_et_egales );
     ajouterCasDeTest( tests_plus );
     ajouterCasDeTest( tests_moins );
     ajouterCasDeTest( tests_inverse );
     ajouterCasDeTest( tests_fois );
     ajouterCasDeTest( tests_chaine );
FIN_SUITE_DE_TESTS
```

Programme 3: Programme de tests (extraits) pour le module `fractions` défini avec `MiniCUnit` : `tester-fractions.c`

Par contre, supposons que la fonction `moins` contienne une erreur de style *copier-coller*, c'est-à-dire qu'on a repris textuellement le code de `plus` en voulant ensuite remplacer le «+» par un «-», mais qu'on a... oublié de faire cette modification : (Voici alors le résultat qui serait produit par l'exécution du programme de tests :

```
 1) "suiteFractions"::"tests_moins"
        #12: 'Resultat incorrect pour moins'
 2) "suiteFractions"::"tests_moins"
        #13: 'Resultat incorrect pour moins'
 3) "suiteFractions"::"tests_moins"
        #15: 'Resultat incorrect pour moins'
*** Sommaire: 1 suites executees; 6 tests executes, 1 tests echoues;
    24 assertions evaluees, 3 assertions echouees
```

C Un exemple de programme de tests pour un programme de type filtre

Un filtre est un programme qui lit des données sur `stdin` et qui produit des résultats sur `stdout`. Dans ce qui suit, nous allons illustrer une façon de tester un tel programme.

Supposons donc que l'on désire développer et tester un programme qui réalise l'équivalent d'une version *simplifiée* de la commande `wc`. La version Linux de `wc` est décrite comme suit (`man wc`) :

> *The wc utility reads one or more input files and, by default, writes the number of newline characters, words and bytes contained in each input file to the standard output.*

De plus, si aucun fichier n'est fourni, ou si le nom "-" est mentionné, la lecture se fait depuis l'entrée standard.

Pour simplifier le problème, nous allons supposer que notre version de `wc` lit toujours depuis l'entrée standard. Son utilisation pourra donc se faire comme suit :

- Utilisation avec un fichier texte :

  ```
  % cat fich1
  quatre mots
  deux lignes
  % mon-wc.out < fich1
   2  4 24
  ```

- Utilisation directe au clavier, en utilisant le *device* `/dev/tty` comme nom de fichier, la fin de fichier étant indiquée par le caractère "^D" (invisible mais entré juste après le *Enter* qui suit "`lignes`") :

  ```
  % mon-wc.out < /dev/tty
  quatre mots
  deux lignes
        2       4      24
  ```

- Utilisation directe au clavier, avec un *here-file* :

  ```
  % mon-wc.out << FIN
  ? quatre mots
  ? deux lignes
  ? FIN
   2  4 24
  ```

```
.SUFFIXES : .o .c

CC=gcc

#DEBUG_TRACE=-DDEBUG_TRACE
DEBUG_TRACE=

.c.o :
    $(CC) -c $(DEBUG_TRACE) $<

OBJ=mon-wc.o


mon-wc.out : $(OBJ)
    $(CC) -o mon-wc.out $(OBJ)

tests: mon-wc.out
    executer-tests mon-wc.out Tests

mon-wc.o : debuggage.h

clean :
    rm -f *.o core
    rm -f *.out
    rm -f *~
    rm -f *.h.gch
```

Fichier `makefile` 2: Fichier `makefile` pour la compilation et l'exécution des tests du programme `mon-wc.c`

Nom du test	Contenu du fichier de données	Résultat attendu
`vide`		0 0 0
`multilignesVides`		2 0 2
`5blancs`		1 0 6
`1ligne`	`une ligne`	1 2 10
`2lignes`	`une ligne` `deux lignes`	2 4 22

`3lignes`	`a`
	`bb`
	`ccc`
		3 3 9

Tableau 1: Les différents cas de tests (données et résultats attendus) pour le programme `mon-wc`

Le fichier `mon-wc.c` contiendra notre programme, que nous ne présenterons pas. La compilation du programme s'effectuera tel que décrit dans le fichier `makefile` 2.

Dans un premier temps, nous allons définir un ensemble de fichiers qui représenteront nos cas de tests. Plus précisément, nous aurons besoin de deux fichiers par cas de tests :

- Un fichier `test-X.donnees` indiquant les données du cas de test.
- Un fichier `test-X.resultats` indiquant les résultats attendus pour ce cas de test.

Pour notre exemple, nous utiliserons les six cas de test présentés dans le tableau 1 — un nom de test `X` (1ère colonne, par exemple, `vide`) signifie que le fichier contenant les données (2ième colonne) sera `test-X.donnees` (par exemple, `test-vide.donnees`), alors que le fichier contenant les résultats attendus (3ième colonne) sera `test-X.resultats`.

Pour chacun des cas de test, disons le test `X`, nous allons effectuer les opérations suivantes, opérations que nous inclurons dans un script Unix (*C shell script*) :

- On exécute tout d'abord le programme `mon-wc.out` sur le fichier `test-X.donnees`, en conservant le résultat dans un fichier temporaire :

```
mon-wc.out <test-X.donnees >test-X.resObtenu
```

- On vérifie ensuite que les résultats produits sont bien ceux attendus. Pour ce faire, nous allons utiliser la commande `diff`, qui permet de comparer le contenu de deux fichiers, et ce ligne par ligne. Toutefois, comme les blancs qui apparaissent dans le fichier de résultat ne sont pas significatifs, nous allons donc utiliser les options `t`, `w` et `b` de `diff` pour les ignorer :

```
diff -twb $repertoireTests/$casDeTest.resultats $casDeTest.resObtenu | more
if ($status != 0) then
    # Des differences existent: on en prend note.
    set differences  = "$differences $casDeTest"
    set estDifferent = 1
endif
```

  Si des différences sont rencontrées, on en prend note en utilisant les variables `differences` et `estDifferent`.

- Après avoir exécuté chacun des cas de tests, on indique ensuite si des différences ont été rencontrées ou non :

```
if ($estDifferent == 0) then
  echo ""
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++"
  echo "--- OK (aucun fichier n'a produit de difference)."
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++"
else
  echo ""
  echo "----------------------------------------------------"
  echo "*** Les fichiers suivants ont produit des differences:"
  echo "  $differences"
  echo "----------------------------------------------------"
endif
```

Pour automatiser l'exécution des tests, nous allons donc définir un *shell script*. Nous allons supposer que tous les cas de tests sont définis dans un répertoire indépendant. Dans notre exemple, on utilisera le répertoire `Tests` :

```
% ls Tests
Tests:
test-1ligne.donnees             test-5blancs.donnees
test-1ligne.resultats           test-5blancs.resultats
test-2lignes.donnees            test-multiLignesVides.donnees
test-2lignes.resultats          test-multiLignesVides.resultats
test-3lignes.donnees            test-vide.donnees
test-3lignes.resultats          test-vide.resultats
```

Le script, pour qu'il puisse être réutilisé dans d'autres contextes, recevra deux arguments :

1. Le nom du programme à exécuter.
2. Le nom du répertoire contenant les divers cas de tests.

Le script 1 présente le *shell script*, de type *C-shell*. La figure 2 présente le résultat de l'exécution de ce script sur un programme où certains cas de tests ne fonctionnent pas alors que la figure 3 présente le résultat de l'exécution du script sur un programme qui fonctionne correctement et qui passe tous les tests.

```
#!/bin/csh

if ($#argv < 2) then
    echo "Script pour executer une serie de cas de tests"
    echo "usage:"
    echo "  Argument 1: Nom du programme (executable) a tester"
    echo "  Argument 2: Nom du repertoire contenant les fichiers de tests"
    echo ""
    echo "  Les donnees doivent etre dans des fichiers 'test*.donnees'"
    echo "  Les resultats attendus doivent etre dans des fichiers 'test*.resultats'"
    exit -1
endif

set programmeATester = $1
set repertoireTests  = $2

# Variables pour determiner si des differences existent et ce qu'elles sont.
set estDifferent = 0
set differences  = ""


# On execute le programme sur les divers cas de test dans le repertoire indique.
#
foreach f (`ls $repertoireTests/test*.donnees`)
    set casDeTestDonnees = $f:t
    set casDeTest = $casDeTestDonnees:r

    # On execute le programme sur les donnees du cas de test.
    echo "**** Execution de $programmeATester sur $repertoireTests/$casDeTest"
    ./$programmeATester <$repertoireTests/$casDeTest.donnees >$casDeTest.resObtenu

    # On compare les resultats obtenus.
    diff -twb $repertoireTests/$casDeTest.resultats $casDeTest.resObtenu | more
    if ($status != 0) then
        # Des differences existent: on en prend note.
        set differences  = "$differences $casDeTest"
        set estDifferent = 1
    endif
end


# On imprime les differences, si elles existent.
#
if ($estDifferent == 0) then
  echo ""
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++"
  echo "--- OK (aucun fichier n'a produit de difference)."
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++"
else
  echo ""
  echo "----------------------------------------------------"
  echo "*** Les fichiers suivants ont produit des differences:"
  echo "  $differences"
  echo "----------------------------------------------------"
endif

# On fait le menage.
rm -f *.resObtenu
```


Script 1: Le *shell script* `executer-tests` permettant d'exécuter les différents cas de test pour le programme `mon-wc.c`

```
% make tests
gcc -c  mon-wc.c
gcc -o mon-wc.out mon-wc.o
executer-tests mon-wc.out Tests
**** Execution de mon-wc.out sur Tests/test-1ligne
1c1
<  1  2 10
---
>  10  2 10
**** Execution de mon-wc.out sur Tests/test-2lignes
**** Execution de mon-wc.out sur Tests/test-3lignes
**** Execution de mon-wc.out sur Tests/test-5blancs
1c1
< 1 0 6
---
> 10 0 6
**** Execution de mon-wc.out sur Tests/test-multiLignesVides
**** Execution de mon-wc.out sur Tests/test-vide

-----------------------------------------------------
*** Les fichiers suivants ont produit des differences:
   Tests/test-1ligne Tests/test-5blancs
-----------------------------------------------------
```

Figure 2: Résultats de l'exécution du script `executer-tests` sur une version du programme `mon-wc` qui ne produit pas les bons résultats

```
% make tests
gcc -c  mon-wc.c
gcc -o mon-wc.out mon-wc.o
executer-tests mon-wc.out Tests
**** Execution de mon-wc.out sur Tests/test-1ligne
**** Execution de mon-wc.out sur Tests/test-2lignes
**** Execution de mon-wc.out sur Tests/test-3lignes
**** Execution de mon-wc.out sur Tests/test-5blancs
**** Execution de mon-wc.out sur Tests/test-multiLignesVides
**** Execution de mon-wc.out sur Tests/test-vide

+++++++++++++++++++++++++++++++++++++++++++++++++++++
--- OK (aucun fichier n'a produit de difference).
+++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 3: Résultats de l'exécution du script `executer-tests` sur une version du programme `mon-wc` qui produit les bons résultats

D Un exemple de tests pour un programme qu'on veut modifier et qui gère une base de données (textuelle)

Dans cet exemple, nous allons illustrer un problème typique de maintenance, à savoir, comment améliorer et restructurer un logiciel tout en s'assurant qu'il fonctionne toujours correctement.

Ainsi, dans cet exemple, un programme existe déjà pour gérer une banque de données personnelle de livres prêtés. Plus spécifiquement, les principales fonctionnalités sont les suivantes :

- Indiquer l'emprunt d'un livre.
- Indiquer le retour d'un livre.
- Identifier tous les livres empruntés par une personne.
- Identifier la personne qui a emprunté un livre... puis lui envoyer un courriel lui demandant de rapporter ce livre.
- Demander le rappel, par courriel, de tous les livres prêtés.

Dans son état actuel, ce logiciel semble fonctionner de façon minimalement correcte, puisqu'il est en utilisation depuis déjà plusieurs mois. Toutefois, le logiciel est mal conçu, mal structuré, pas documenté, sans aucun test pour vérifier son bon fonctionnement, etc. Bref, il souffre du syndrome du "*stinking code*". Or, on aimerait améliorer ce logiciel, lui ajouter de nouvelles fonctionnalités, le rendre plus fiable et plus propre, possiblement de façon à pouvoir le distribuer de façon publique.

De façon à s'assurer que, dans un premier temps, les travaux de restructuration du code qui seront effectués n'auront pas d'impact sur le bon fonctionnement du logiciel, nous allons donc développer un ensemble de tests avec exécution automatique, tests qui assureront la *non régression* du logiciel lorsqu'on le modifiera.

Comme il s'agit d'un programme qui gère une base de données (textuelle), et comme certaines des commandes peuvent avoir un effet sur cette base de données (par exemple, `emprunter` ou `rapporter`), il ne sera donc pas suffisant de définir des données et les résultats attendus pour ces données, comme dans le cas d'un programme de type filtre (cf. annexe C). Soit `X` un cas de test spécifique — dans ce qui suit, chaque cas de test sera simplement identifié par un numéro unique. Chaque cas de tests devra plutôt être défini par les informations suivantes, spécifiées dans quatre fichiers distincts :

1. `test-X.commandes` : fichier contenant une série de commandes qu'on veut exécuter et dont on veut vérifier le bon fonctionnement.
2. `test-X.avant` : fichier décrivant l'état de la BD avant l'exécution de la série de commandes.[5]
3. `test-X.resultats` : fichier décrivant les résultats produits sur `stdout` suite à l'exécution des commandes qu'on a voulu tester.
4. `test-X.apres` : fichier décrivant l'état de la BD après l'exécution des diverses commandes.

Pour chacun des cas de test, disons `X`, nous allons donc effectuer les opérations suivantes :

- On exécute tout d'abord les commandes contenues dans le fichier `test-X.commandes`, en conservant les résultats émis sur `stdout` dans un fichier temporaire — l'exécution des commandes s'effectue par un appel à l'interpréteur de commandes `csh`, avec comme argument le fichier des commandes :

```
csh $repertoireTests/$casDeTest.commandes >$casDeTest.resObtenu
```

[5] Dans ce qui suit, dans tous les cas, nous partirons d'une BD ne contenant aucun élément, donc d'un fichier vide.

- On vérifie ensuite que les résultats produits sont bien ceux attendus et, si de telles différences sont rencontrées, on en prend note. Pour faire la comparaison, nous allons utiliser la commande `diff`, qui permet de comparer le contenu de deux fichiers, et ce ligne par ligne. Toutefois, comme les blancs qui apparaissent dans le fichier de résultat ne sont pas significatifs, nous allons donc utiliser les options `t`, `w` et `b` de `diff` pour les ignorer :

```
diff -twb $repertoireTests/$casDeTest.resultats $casDeTest.resObtenu | more
if ($status != 0) then
    # Des differences existent: on en prend note.
    set differences  = "$differences $casDeTest"
    set estDifferent = 1
    echo "$casDeTest" >> $fichErreurs
    set casDeTestOk = 0
endif
```

- Ensuite, on doit aussi s'assurer que la base de données textuelle a été correctement modifiée. Pour rendre le script plus générique, le nom du fichier texte contenant cette base de données est reçue en argument, qu'on aura affecté préalablement à la variable `BD`. La vérification de l'état de la base de données se fait donc comme suit — on ne veut pas noter qu'un cas de test incorrect a été exécuté si ce cas de test avait déjà été pris en note, d'où l'utilisation de l'indicateur `casDeTestOk` :

```
diff -twb $repertoireTests/$casDeTest.apres $BD | more
if ( ($casDeTestOk != 0) && ($status != 0) ) then
    # Des differences existent: on en prend note.
    set differences  = "$differences $casDeTest"
    set estDifferent = 1
    echo "$casDeTest" >> $fichErreurs
endif
```

  Encore une fois, si des différences sont détectées, on en prend note

- Finalement, après avoir exécuté chacun des cas de tests, on indique si des différences ont été rencontrées ou non — une ligne finale sommaire indiquant le nombre total d'erreurs est aussi imprimée :

```
if ($estDifferent == 0) then
  echo ""
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++"
  echo "--- OK (aucun fichier n'a produit de difference)."
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++"
else
  echo ""
  echo "----------------------------------------------------"
  echo "*** Les fichiers suivants ont produit des differences:"
  echo "  $differences"
  echo "----------------------------------------------------"
endif

set nbErreurs = `wc -l <$fichErreurs`
echo "*** Sommaire: $nbErreurs cas de test errone(s) ***"
```

Le script 2 présente le script complet. La figure 4 présente le résultat produit par l'exécution du script via l'appel à la commande "`make tests`" dans le cas où tous les tests fonctionnent correctement (et dans le cas où la compilation des divers fichiers définissant le programme a été effectuée au préalable). Pour cet exemple, nous avons produit sept (7) cas de tests différents, dont les noms sont indiqués dans la figure 5

```
#!/bin/csh

if ($#argv < 2) then
    echo "Script pour executer une serie de cas de tests pour la bibliotheque"
    echo "usage:"
    echo "  Argument 1: Nom du repertoire contenant les fichiers de tests"
    echo "  Argument 2: Nom de la base de donnees (textuelle) utilisee par le programme"
    echo ""
    echo "Pour chaque cas de test X, on doit avoir les fichiers suivants:"
    echo "   test-X.commandes"
    echo "   test-X.avant"
    echo "   test-X.resultats"
    echo "   test-X.apres"
    exit -1
endif

set repertoireTests  = $1
set BD               = $2

set fichErreurs = fichAvecErreurs.txt
touch $fichErreurs

# Variables pour determiner si des differences existent et ce qu'elles sont.
set estDifferent = 0
set differences  = ""


# On execute le programme sur les divers cas de test dans le repertoire indique.
#
foreach f (`ls $repertoireTests/test*.commandes`)
    set casDeTestCommandes = $f:t
    set casDeTest = $casDeTestCommandes:r

    # On execute les commandes sur les donnees du cas de test.
    echo "**** Execution du cas $casDeTest"
    cp $repertoireTests/$casDeTest.avant $BD
    chmod +x $repertoireTests/$casDeTestCommandes
    csh $repertoireTests/$casDeTestCommandes >$casDeTest.resObtenu

    # On compare les resultats obtenus.
    set casDeTestOk = 1
    diff -twb $repertoireTests/$casDeTest.resultats $casDeTest.resObtenu | more
    if ($status != 0) then
        # Des differences existent: on en prend note.
        set differences  = "$differences $casDeTest"
        set estDifferent = 1
        echo "$casDeTest" >> $fichErreurs
        set casDeTestOk = 0
    endif
    diff -twb $repertoireTests/$casDeTest.apres $BD | more
    if ( ($casDeTestOk != 0) && ($status != 0) ) then
        # Des differences existent: on en prend note.
        set differences  = "$differences $casDeTest"
        set estDifferent = 1
        echo "$casDeTest" >> $fichErreurs
    endif
end


# On imprime les differences, si elles existent.
#
if ($estDifferent == 0) then
  echo ""
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
  echo "--- OK (aucun fichier n'a produit de difference)."
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
else
  echo ""
  echo "------------------------------------------------------"
  echo "*** Les fichiers suivants ont produit des differences:"
  echo "  $differences"
  echo "------------------------------------------------------"
endif

set nbErreurs = `wc -l <$fichErreurs`
echo "*** Sommaire: $nbErreurs cas de test errone(s) ***"

# On fait le menage.
rm -f *.resObtenu
rm -f $fichErreurs
```

Script 2: Le *shell script* `executer-tests` pour les différents cas de test du programme de gestion des prêts de livre

```
% make tests
gcc -o emprunter emprunter.c
chmod og+rx emprunter
gcc -o rappeler-livre rappeler-livre.c
chmod og+rx rappeler-livre
gcc -o demander-retours demander-retours.c
chmod og+rx demander-retours
gcc -o emprunts emprunts.c
chmod og+rx emprunts
gcc -o emprunteur emprunteur.c
chmod og+rx emprunteur
gcc -o rapporter rapporter.c
chmod og+rx rapporter
gcc -o rechercher_parametres2.cgi rechercher_parametres2.c
chmod og+rx rechercher_parametres2.cgi
gcc -o formulaire_emprunteurs.cgi proc_CGI.c formulaire_emprunteurs.c
chmod og+rx formulaire_emprunteurs.cgi
gcc -o formulaire_titres.cgi proc_CGI.c formulaire_titres.c
chmod og+rx formulaire_titres.cgi
executer-tests Tests livredb.txt
**** Execution du cas test-1
**** Execution du cas test-2
**** Execution du cas test-3
**** Execution du cas test-4
**** Execution du cas test-5
**** Execution du cas test-6
**** Execution du cas test-7

++++++++++++++++++++++++++++++++++++++++++++++++++
--- OK (aucun fichier n'a produit de difference).
++++++++++++++++++++++++++++++++++++++++++++++++++
*** Sommaire: 0 cas de test errone(s) ***
```

Figure 4: Résultats de l'exécution du script `executer-tests` pour le programme de gestion de prêts des livres et où le programme produit les bons résultats

```
% ls Tests
Tests:
CVS/              test-3.apres      test-5.commandes*
mk-test*          test-3.avant      test-5.resultats
test-1.apres      test-3.commandes* test-6.apres
test-1.avant      test-3.resultats  test-6.avant
test-1.commandes* test-4.apres      test-6.commandes*
test-1.resultats  test-4.avant      test-6.resultats
test-2.apres      test-4.commandes* test-7.apres
test-2.avant      test-4.resultats  test-7.avant
test-2.commandes* test-5.apres      test-7.commandes*
test-2.resultats  test-5.avant      test-7.resultats
```

Figure 5: Les sept (7) cas de tests et leurs (quatre) fichiers associés

```
% cat Tests/test-1.commandes
emprunter ^ Guy ^ guy@bidon.ca ^ Code complet ^ McConnell
emprunts Guy
emprunteur Code complet
% cat Tests/test-1.avant
% cat Tests/test-1.apres
Code complet | McConnell | Guy | guy@bidon.ca
cat Tests/test-1.resultats

Emprunter un livre:
-------------------

Livre ajoute a la liste des livres pretes :

    Titre      : "Code complet"
    Auteur     : McConnell
    Emprunteur : Guy
    Courriel   : guy@bidon.ca



Resultat de la recherche :

Titre      : Code complet
Emprunteur : Guy
Courriel   : guy@bidon.ca


Resultat de la recherche :

Titre      : "Code complet"
Emprunteur : Guy
Courriel   : guy@bidon.ca
```

Figure 6: Les différents fichiers pour le cas de test 1

Les différents fichiers pour le cas de test `1` (`test-1.*`) sont présentés à la figure 6.

Que se passe-t-il maintenant si on désire commencer à modifier le programme pour l'améliorer? Par exemple, lorsqu'on désire déterminer les emprunts effectués par quelqu'un et que cette personne n'a emprunté aucun livre, le message suivant est produit :

```
Resultat de la recherche :


Cette persone n'a emprunter aucun livre.
```

Or, ce message contient des fautes d'orthographe qu'on aimerait corriger.

```
% make tests
executer-tests Tests livredb.txt
**** Execution du cas test-1
**** Execution du cas test-2
**** Execution du cas test-3
**** Execution du cas test-4
73c73
< Cette persone n'a emprunter aucun livre.
---
> Cette personne n'a emprunte aucun livre.
**** Execution du cas test-5
**** Execution du cas test-6
**** Execution du cas test-7

-------------------------------------------------------
*** Les fichiers suivants ont produit des differences:
   test-4
-------------------------------------------------------
*** Sommaire: 1 cas de test errone(s) ***
```

Figure 7: Résultat de l'exécution des divers cas de test suite à la modification du fichier `emprunts.c` pour corriger des erreurs d'orthographe dans le message d'erreur

Supposons qu'on effectue la correction, dans le fichier `emprunts.c` et qu'on exécute ensuite les tests. Dans ce cas, on obtiendra les résultats indiqués à la figure 7. Or, ici, ce n'est pas le programme qui est erroné ... mais plutôt les résultats attendus d'un cas de tests qui ne sont plus corrects! Il faut donc *modifier les cas de tests* en conséquence — dans le cas présent, il faut modifier le fichier `test-4.resultats` pour tenir compte de la modification par rapport aux résultats attendus.

Des telles modifications conjointes du code et des cas de tests sont tout à fait normales et inévitables. Comme il a été indiqué plus tôt, "[...] les cas de tests sont un livrable aussi important que le code lui-même (et il faut donc s'assurer de bien intégrer les deux)". Or, les intégrer veut aussi dire les faire évoluer conjointement, lorsque nécessaire.

Références

[AMN02] D. Astels, G. Miller, and M. Novak. *A Practical Guide to eXtreme programming.* The Coad Series. Prentice-Hall PTR, 2002.

[Bec01] K. Beck. Aim, fire. *IEEE Software*, 18(6):87–89, 2001.

[Bec03] K. Beck. *Test-Driven Development — By Example.* Addison-Wesley, 2003.

[FF04] M. Fowler and M. Foemmel. Continuous integration. http://www.martinfowler.com/articles/continuousIntegration.html, June 2004.

[HT00] A. Hunt and D. Thomas. *The Pragmatic Programmer—From journeyman to master.* Addison-Wesley, 2000.

[KP99] B.W. Kernighan and R. Pike. *The Practice of Programming.* Addison-Wesley, 1999.

[McC93] S. McConnell. *Programmation professionnelle.* Microsoft Press, 1993.

[RK03] P.N. Robillard and P. Kruchten. *Software Engineering Process with the UPEDU.* Addison-Wesley, 2003.

[TMSZ07] G. Tremblay, B. Malenfant, A. Salah, and P. Zentilli. Introducing students to professional software construction: A "Software construction and maintenance" course and its maintenance corpus. In *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education*, pages 176–180. ACM, June 2007.