TP Tests Logiciels en C++

Sommaire

Tests Logiciels

Notion de test

Le test est une **recherche d'anomalie** (ou **défaut**, appelé souvent ***bug*** en informatique) dans le comportement d'un logiciel.


Si une batterie de tests ne montre pas de défaut cela n'implique pas que le logiciel n'a pas de défaut ...


Le test n'a pas pour objectif : de diagnostiquer la cause des erreurs, de corriger les fautes ou de prouver la correction !

A l'origine, il y a la **faute** (*mistake*) : c'est la cause d'une **erreur** (*error*). Un **défaut** (*bug*) est la manifestation d'une **erreur** dans un logiciel. Un **défaut** peut causer une **panne** (*failure*). Une **panne** est la fin de la capacité d'un système ou de l'un de ses composants d'effectuer la fonction requise, ou de l'effectuer à l'intérieur des limites spécifiés.


On évalue à environ 40% la part des tests dans le coût d'un logiciel (et plus pour des logiciels critiques).

Les tests logiciels en BTS couvrent : les tests unitaires, les test d'intégration et les tests de validation (recette).


La « recette » est la phase de validation de la conformité en rapport au cahier des charges fonctionnel.

Tests unitaires

Chaque module du logiciel est testé séparément par rapport à ses spécifications. **En programmation C++, on fera des tests unitaires au niveau des méthodes (fonctions)**, puis au niveau de la classe.


Les méthodes *Extreme programming* (XP) ou *Test Driven Development* (TDD) ont remis les tests unitaires (TU ou UT), appelés « tests du programmeur », au centre de l'activité de programmation.

Tests d'intégration

Les modules validés par les tests unitaires sont rassemblés dans un composant logiciel. Le test d'intégration vérifie que l'intégration des modules n'a pas altéré leur comportement.

Tests de validation

Le test vérifie que le logiciel réalisé correspond bien aux besoins exprimés par le client. La validation ou vérification d'un produit cherche donc à s'assurer qu'on a **construit le bon produit**.

Tests de recette

L'application doit fonctionner dans son environnement de production (tests d'intégration système), avec les autres applications présentes sur la plate-forme et avec le système d'exploitation. Les clients utilisateurs vérifient sur site que le système répond de manière parfaitement correcte.

Autres tests

En informatique, les tests logiciels sont nombreux : tests de boîte noire (ou test fonctionnel), tests de boîte blanche (test structurel), tests de conformité ou de non conformité, tests fonctionnels, tests de non régression (ou de régression), tests de robustesse (d'endurance, de fiabilité), tests de charge et de montée en charge, tests aux limites, tests de stress, ...

Ces tests sont décrits dans le document `fiche-t2-autres_tests.pdf`.

CppUnit

À l'origine, Kent Beck crée l'environnement de test `sUnit` pour le langage *Smalltalk* en octobre 1994. En 1997, Kent Beck rencontre Erich Gamma avec lequel il crée `JUnit` qui, suite à sa popularité, entraînera la création de nombreux *frameworks* de tests unitaires, cet ensemble se nomme `xUnit`.

Le terme générique « `xUnit` » désigne un outil permettant de réaliser des tests unitaires dans un langage donné (dont l'initiale remplace « `x` » le plus souvent) : `CppUnit` pour le `C++`, `CUnit` pour le `C`, `PHPUnit` pour `PHP`, `JUnit` pour `Java`, etc ...

La plupart des *frameworks* de la famille `xUnit` permettent la génération des classes de test unitaire. Cependant ces *frameworks* ne fournissent que le squelette des classes. Les tests devront donc être écrits par le développeur.

`CppUnit` est donc l'équivalent `C++` de l'outil `JUnit` créé entre autres par Kent Beck.

Liens :
- http ://sourceforge.net/projects/cppunit/
- http ://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html
- http ://matthieu-brucher.developpez.com/tutoriels/cpp/cppUnit/

Installation de `CppUnit` sur **Ubuntu** :

```
$ sudo apt-get install libcppunit-dev

$ cppunit-config --version
1.12.1

$ cppunit-config --cflags

$ cppunit-config --libs
-lcppunit -ldl
```


Mise en œuvre d'un test unitaire

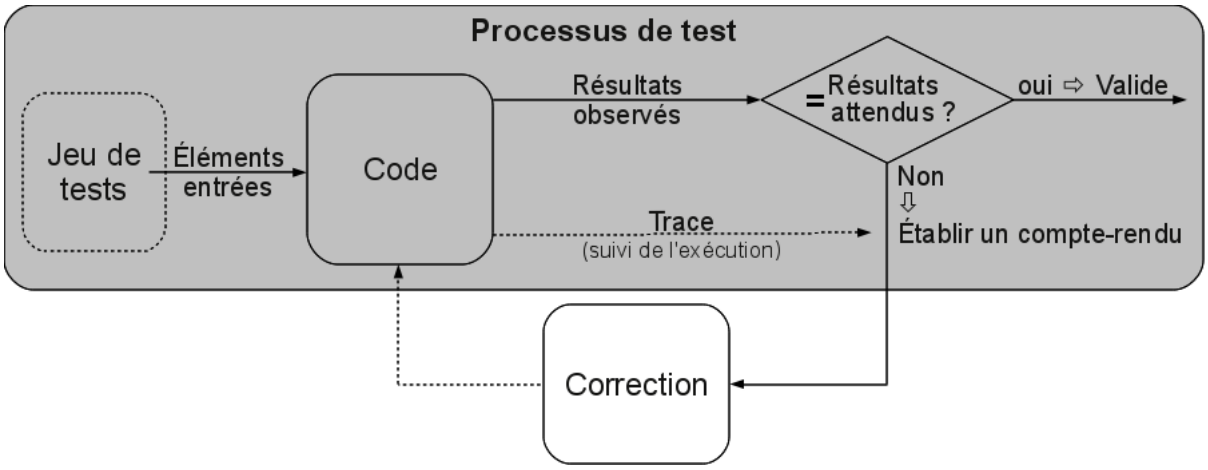
Principe

Le principe d'un test unitaire (**T.U.**) est simple : on va tester chaque fonction ou méthode individuellement (**unitairement**) et séparément par rapport à ses spécifications (c'est-à-dire ce qu'elle doit faire ou plus précisément ce que l'on attend d'elle).

Pour réaliser un test unitaire d'une fonction ou méthode, on procède de la manière suivante :

- on choisit un **jeu de tests** : c'est-à-dire les données d'entrée de la fonction (ses arguments)
- on définit le **résultat attendu** : c'est-à-dire le résultat qu'elle doit fournir
- on vérifie le **résultat obtenu** avec le résultat attendu

On validera une fonction lorsque tous les jeux de tests choisis ne détectent aucune anomalie (un résultat obtenu différent du résultat attendu).



Toute la difficulté est de bien choisir ses jeux de tests pour couvrir le maximum de cas afin de détecter un *bug*.

Préparation

Un test doit être **non intrusif**. Il faut tout d'abord penser à séparer le code source de l'application de la suite de tests.

Exemple classique d'organisation :

- `src` : répertoire contenant l'ensemble du code source de l'application
- `tests` : répertoire contenant l'ensemble du code source des tests unitaires de l'application

Le code source doit être intègre, c'est-à-dire non modifié pour réaliser le test. Par exemple pour un test fonctionnel on ne doit pas ajouter de saisie, de `printf` ou de `TRACE` qui modifie entre autre les temps d'exécution et donc la validité du module testé.

Les classes d'équivalence

Pour réduire le nombre de ces tests, on utilise la technique des **classes d'équivalence**. Cette technique consiste à identifier des classes d'équivalence dans le domaine des données d'entrées vis à vis d'une propriété d'une donnée de sortie. Tout test effectué avec une entrée quelconque appartenant à une classe d'équivalence déterminée entraîne un résultat soit correct (**classe valide**), soit incorrect (**classe invalide**).

Une fois les classes déterminées, il suffit de prendre au moins un représentant pour chacune de ces classes (ce sera un jeu de test).

Un plan de test se représente par un tableau contenant : une description du test, les valeurs en entrées de la fonction et le résultat attendu.

Exemple : la méthode `setCanalRouge()` permet d'affecter une valeur comprise entre 0 et 255 (codée sur 8 bits) au canal d'un appareil DMX

Module testé : setCanalRouge				Date : 25/12/2014
Testeur : Shadock				Version : 1.0
Classe	Description	Valeurs en entrée	Résultats attendus	Résultats observés
Valide n°1	première valeur de la plage	0	valeur de retour : 0 valeur du canal : 0	
Valide n°2	Une valeur quelconque dans la plage (au milieu)	128	valeur de retour : 0 valeur du canal : 128	
Valide n°3	Dernière valeur de la plage	255	valeur de retour : 0 valeur du canal : 255	
Invalide n°1	Valeur inférieure au minimum de la plage	-1	valeur de retour : -1 valeur du canal : inchangée	
Invalide n°2	Valeur supérieure au maximum de la palge	256	valeur de retour : -1 valeur du canal : inchangée	

Un rapport de test correspondra au tableau ci-dessus complété avec la possibilité d'ajouter des remarques et hypothèses en cas d'échec.

Action corrective

À partir du rapport de test, le développeur en charge de cette fonction propose une action corrective sur le code source. Cette activité ne fait pas partie de la procédure de tests.

Tests de non régression

Après chaque modification, correction ou adaptation du logiciel, il faudra vérifier que le comportement des fonctionnalités n'a pas été perturbé, même lorsqu'elle ne sont pas concernées directement par la modification. Il faudra donc **« rejouer » tous les tests après toutes modifications du code**.

Travail demandé

Objectifs

Les objectifs de ce TP sont de mettre en œuvre les procédures de tests unitaires en utilisant le *framework* `CppUnit`.

Dans le cadre d'un développement spécialisé dans la mesure industrielle, vous participez à la mise au point d'une classe `Mesure`.

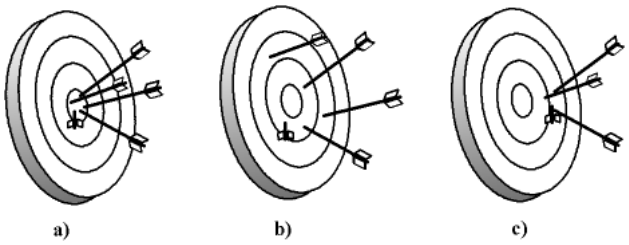
L'acquisition de mesures peut comporter des échantillons incohérents. Après traitement, on ne conservera que la médiane (et non la moyenne) de ces séries de mesures. La valeur médiane est la valeur qui se trouve au milieu d'un ensemble de nombres triés. Si cet ensemble contient un nombre pair de nombres, la médiane sera alors la moyenne des deux nombres du milieu.

Les mesures dans l'industrie

Dans le cas des mesures dans l'industrie, on considère trois sources d'erreur :
– la précision de la mesure ou l'incertitude ;
– la dispersion statistique ;
– l'erreur systématique.

L'erreur totale étant la somme des trois sources d'erreurs (cf. Erreur_de_mesure).

Si l'on fait la comparaison avec des flèches que l'on tire sur une cible :
– la précision de mesure désigne la taille de la pointe de la flèche ;
– la dispersion statistique désigne le fait que les flèches sont proches les unes des autres, ou bien au contraire éparpillées sur la cible ;
– l'erreur systématique indique si les flèches visaient bien le centre, ou bien un autre point de la cible.



Pour la dispersion statistique, on estime que si l'on mesure plusieurs fois le même phénomène avec un appareil suffisamment précis, on obtiendra chaque fois un résultat différent. Ceci est dû à des phénomènes perturbateurs ou, pour les mesures extrêmement précises, à la nature aléatoire du phénomène.

Parmi les phénomènes perturbateurs, on peut dénombrer :
– l'erreur d'échantillonnage : c'est lorsque l'on prélève un échantillon qui n'est pas représentatif de ce que l'on veut mesurer ; le résultat dépend alors de la manière dont on choisit l'échantillon ;
– l'erreur de préparation : l'échantillon s'altère pendant le transport, le stockage ou la manipulation (pollution, dégradation, transformation physique ou chimique) ;
– la stabilité de l'appareil : celui-ci peut être sensible aux variations de température, de tension d'alimentation électrique, aux vibrations, aux perturbations électromagnétiques des appareils environnants ou bien présenter un défaut de conception ou une usure (bruit de fond électronique, pièce instable ...).

Le calcul d'erreur, ou calcul d'incertitudes est un ensemble de techniques permettant d'estimer l'erreur faite sur un résultat numérique, à partir des incertitudes ou des erreurs faites sur les mesures qui ont

conduit à ce résultat. L'erreur de mesure détermine la sensibilité (capacité à sélectionner les bons « candidats ») et la sélectivité (capacité à éliminer les mauvais « candidats ») d'une méthode.

Moyenne vs Médiane

L'utilisation de la médiane à la place de la moyenne est fréquent pour les mesures dans l'industrie.

Exemple, soit deux listes de mesures provenant d'un capteur sur une période de 1mn30s :
– L1 : 35,53°C, 35,23°C, 35,10°C, 35,02°C, 34,45°C
– L2 : 35,53°C, 35,23°C, 35,10°C, 34,45°C, 12,22°C

Dans la série L2, la mesure incohérente (12,22°C) serait prise en compte dans la moyenne et fausserait donc le résultat obtenu (la moyenne sans cette valeur est de 35,07°C contre 30,50°C si on en tient compte) :

Liste	L1	L2
Moyenne	35,066°C	30,506°C
Médiane	35,10°C	35,10°C

Ici, l'utilisation de la médiane comme technique de sélectivité permet d'atténuer ce type de problème.

L'utilisation de la valeur médiane est donc préférable à la valeur moyenne. Cependant, son utilisation implique le tri des données au préalable.

La classe `Mesure` à tester

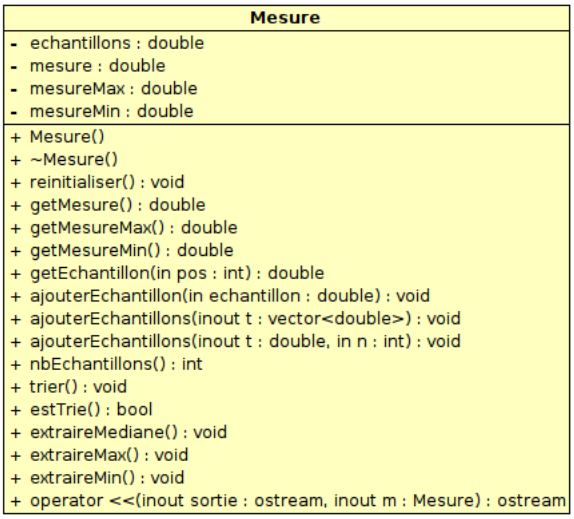
La classe `Mesure` possède les attributs suivants :

– `echantillons` : les échantillons obtenues par acquisition d'un capteur (un *vector* de `double`)
– `mesure` : la mesure sélectionnée parmi les échantillons (un `double`)
– `mesureMax` : la mesure maximale parmi les échantillons (un `double`)
– `mesureMin` : la mesure minimale parmi les échantillons (un `double`)

La classe `Mesure` offrira les services suivants :

– `reinitialiser()` : vide le *vector* `echantillons` pour une nouvelle mesure et réinitialise les valeurs de `mesure`, `mesureMax` et `mesureMin`
– `getMesure()` : retourne sous la forme d'un `double` la mesure sélectionnée parmi les échantillons en utilisant la médiane comme technique de sélectivité
– `getMesureMax()` : retourne sous la forme d'un `double` la mesure maximale parmi les échantillons
– `getMesureMin()` : retourne sous la forme d'un `double` la mesure minimale parmi les échantillons
– `getEchantillon(int pos)` : retourne la valeur de l'échantillon à la position `pos` dans le *vector* `echantillons`
– `ajouterEchantillon(double echantillon)` : permet ajouter un échantillon de type `double` dans le *vector* `echantillons`
– `ajouterEchantillons(vector<double> &t)` : permet ajouter un ensemble d'échantillons en passant un *vector* de `double`
– `ajouterEchantillons(double *t, int n)` : permet ajouter un ensemble d'échantillons en passant un tableau de `double` et sa taille
– `nbEchantillons()` : retourne le nombre d'échantillons stockés dans le *vector* `echantillons`
– `trier()` : trie le *vector* `echantillons`
– `estTrie()` : retourne `true` si le *vector* `echantillons` est trié sinon `false`

– `extraireMediane()` : trie et extrait la médiane parmi les échantillons
– `extraireMax()` : extrait la valeur maximale parmi les échantillons
– `extraireMin()` : extrait la valeur minimale parmi les échantillons



La déclaration de la classe `Mesure` :

```
#ifndef MESURE_H
#define MESURE_H

#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>

using namespace std;

class Mesure
{
   public:
      Mesure();
      ~Mesure();

      void  reinitialiser();

      double getMesure();
      double getMesureMax();
      double getMesureMin();

      double getEchantillon(int pos);
      void  ajouterEchantillon(double echantillon);
      void  ajouterEchantillons(vector<double> &t);
      void  ajouterEchantillons(double *t, int n);

      int   nbEchantillons() const ;
      void  trier();
      bool  estTrie() const;

      void  extraireMediane();
```

```
      void  extraireMax();
      void  extraireMin();

      friend ostream& operator<<(ostream &sortie, Mesure &m);

   private:
      vector<double> echantillons;
      double mesure;
      double mesureMax;
      double mesureMin;
};

#endif
```

Code 2 – mesure.h

La définition complète de la classe `Mesure` à tester est fournie en **Annexe 1**.

On écrit un simple programme utilisant la classe `Mesure` :

```
#include <iostream>
#include "mesure.h"

using namespace std;

int main()
{
    Mesure mesure;

    cout << mesure << endl;

    const int NB_ELT = 5;
    vector<double> echantillons(NB_ELT);

    // acquisition : on simule des échantillons
    echantillons[0] = 35.53;
    echantillons[1] = 35.23;
    echantillons[2] = 35.10;
    echantillons[3] = 35.02;
    echantillons[4] = 34.45;

    mesure.ajouterEchantillons(echantillons);

    cout << mesure << endl;

    cout << "mesure␣selectionnée␣:␣" << mesure.getMesure() << endl;

    return 0;
}
```

Code 3 – main.cpp

On obtient à l'exécution :

```
 $ ./src/mesure
nb echantillons :     0
echantillons :        aucun
```

```
mesure :              0
mesure min :          0
mesure max :          0

nb echantillons :     5
echantillons :        35.53 35.23 35.1 35.02 34.45
mesure :              35.1
mesure min :          0
mesure max :          0

mesure selectionnée : 35.1
$
```

La mesure obtenue est correcte ...

Est-ce suffisant pour déclarer que la classe Mesure ne comporte aucun *bug* ?

Il est nécessaire de mettre en œuvre une procédure de tests unitaires pour la classe `Mesure` en utilisant le *framework* `CppUnit`.

Séquence 1 : la classe de test `TestUnitaireMesure`

Cette étape est destinée à créer une classe de test `TestUnitaireMesure` avec le *framework* `CppUnit`.

Le principe est simple :

- on déclare la classe de test en héritant de la classe `CPPUNIT_NS::TestFixture`
- on crée une suite de tests unitaires en utilisant les macros forunies par `CppUnit` (`CPPUNIT_TEST_SUITE` et `CPPUNIT_TEST_SUITE_END`)
- on ajoute les méthodes de tests dans la suite de tests unitaires créée avec la macro `CPPUNIT_TEST`
- on déclare les méthodes de tests

La déclaration de la classe `TestUnitaireMesure` sera la suivante :

```
#ifndef _TESTUNITAIREMESURE_H
#define _TESTUNITAIREMESURE_H

#include <cppunit/extensions/HelperMacros.h>

class Mesure; // La classe à tester

class TestUnitaireMesure : public CPPUNIT_NS::TestFixture
{
   // On crée une suite de tests unitaires pour la classe
   CPPUNIT_TEST_SUITE( TestUnitaireMesure );
   CPPUNIT_TEST( testReinitialiser ) ;
   CPPUNIT_TEST( testAjouterEchantillons ) ;
   CPPUNIT_TEST( testTriTaillePaire ) ;
   CPPUNIT_TEST( testTriTailleImpaire ) ;
   CPPUNIT_TEST( testTriDejaTrie ) ;
   // TODO : etc ...
   CPPUNIT_TEST_SUITE_END();
```

```
   private:
      Mesure *mesure; // un pointeur sur une instance de la classe à tester

   public:
      TestUnitaireMesure();
      virtual ~TestUnitaireMesure();

      // Call before tests
      void setUp();

      // Call after tests
      void tearDown();

      // Liste des tests
      void testReinitialiser();
      void testAjouterEchantillons();
      void testTriTaillePaire();
      void testTriTailleImpaire();
      void testTriDejaTrie();
      // TODO : etc ...
};

#endif
```

Code 5 – TestUnitaireMesure.h

Il faut maintenant définir les méthodes de tests unitaires. Pour écrire un test unitaire, il faut respecter les principes énoncés précédemment :

1. choisir un **jeu de tests** : c'est-à-dire les données d'entrée
2. définir le **résultat attendu** : c'est-à-dire le résultat qu'elle doit fournir
3. exécuter la méthode à tester
4. vérifier le **résultat obtenu** avec le résultat attendu

Les tests sont vérifiés en s'appuyant sur des "assertions" fournies par `CppUnit`. La liste des "assertions" disponibles dans le *framework* `CppUnit` est fournie en **Annexe 3**.

Exemple pour la méthode `testTriTaillePaire()` :

```
void TestUnitaireMesure::testTriTaillePaire()
{
    // 1. Initialisation du jeu de test :
    const int NB_ELT = 6;
    vector<double> echantillons(NB_ELT);
    bool resultatAttendu; // expected
    bool resultatObtenu; // actual

    echantillons[0] = 35.23;
    echantillons[1] = 35.53;
    echantillons[2] = 35.02;
    echantillons[3] = 35.10;
    echantillons[4] = 34.45;
    echantillons[5] = 4.56;
```

```
    mesure->reinitialiser();
    mesure->ajouterEchantillons(echantillons);

    // 2. Définit le résultat attendu : true -> Le tableau doit être trié
    resultatAttendu = true; // expected

    // 3. Appel méthode testée (lancement) :
    mesure->trier();

    // 4. Vérification des résultats attendus (version détaillée) :
    resultatObtenu = mesure->estTrie();

    // Avec message :
    //CPPUNIT_ASSERT_MESSAGE( "classe valide : le vector doit être trié (taille paire)",
        resultatAttendu == resultatObtenu);
    // ou : CPPUNIT_ASSERT_EQUAL_MESSAGE(message, expected, actual);
    CPPUNIT_ASSERT_EQUAL_MESSAGE( "classe valide : le vector doit être trié (taille paire)",
         resultatAttendu, resultatObtenu);

    // Sans message :
    //CPPUNIT_ASSERT( resultatObtenu == resultatAttendu );
    // ou : CPPUNIT_ASSERT_EQUAL(expected, actual);
    CPPUNIT_ASSERT_EQUAL( resultatAttendu, resultatObtenu );

    // Vérifions que les échantillons sont maintenant triés
    CPPUNIT_ASSERT_EQUAL( echantillons[5], mesure->getEchantillon(0) );
    CPPUNIT_ASSERT_EQUAL( echantillons[4], mesure->getEchantillon(1) );
    CPPUNIT_ASSERT_EQUAL( echantillons[2], mesure->getEchantillon(2) );
    CPPUNIT_ASSERT_EQUAL( echantillons[3], mesure->getEchantillon(3) );
    CPPUNIT_ASSERT_EQUAL( echantillons[0], mesure->getEchantillon(4) );
    CPPUNIT_ASSERT_EQUAL( echantillons[1], mesure->getEchantillon(5) );
}
```

La définition complète de la classe `TestUnitaireMesure` est fournie en **Annexe 4**.

Ensuite, le *framework* `CppUnit` fournit un programme de test générique (le code source est fourni en **Annexe 2**).

On fabrique et on lance le programme de tests :

```
$ make

$ ./tests/testsUnitairesMesure
TestUnitaireMesure::testReinitialiser : OK
TestUnitaireMesure::testAjouterEchantillons : OK
TestUnitaireMesure::testTriTaillePaire : assertion
TestUnitaireMesure::testTriTailleImpaire : assertion
TestUnitaireMesure::testTriDejaTrie : OK
TestUnitaireMesure.cpp:124:Assertion
Test name: TestUnitaireMesure::testTriTaillePaire
equality assertion failed
- Expected: 1
- Actual : 0
- classe valide : le vector doit être trié (taille paire)
```

```
TestUnitaireMesure.cpp:160:Assertion
Test name: TestUnitaireMesure::testTriTailleImpaire
equality assertion failed
- Expected: 1
- Actual : 0

Failures !!!
Run: 5  Failure total: 2 Failures: 2  Errors: 0
```

Bien qu'on pensait avoir une classe `Mesure` correcte, on détecte des erreurs dans le code puisqu'on ne passe pas l'ensemble des tests réalisés !

Bilan : 3 tests sur 5 sont passés ! Seules les méthodes `testReinitialiser()`, `testAjouterEchantillons()` et `testTriDejaTrie()` ont été validées par leur test unitaire.

Par contre, la méthode `Mesure::trier()` comporte un défaut qui a été détecté grâce aux tests `testTriTaillePaire()` et `testTriTailleImpaire()`.

Question 1. Corriger le défaut détecté dans la méthode `Mesure::trier()` et relancer le programme de test jusqu'à la disparition de tout défaut pour valider unitairement cette méthode.

Séquence 2 : tests unitaires

Question 2. Le jeu de test pour valider la méthode `TestUnitaireMesure::testMesureMax()` est incomplet. Compléter la méthode `testMesureMax()` pour qu'elle couvre toutes les classes valides. Valider alors la méthode `Mesure::getMesureMax()` en utilisant le programme de tests jusqu'à ce que la méthode passe le test unitaire.

Question 3. Coder le test unitaire réalisé par la méthode `testMesure()` pour valider unitairement la méthode `Mesure::getMesure()`.

Séquence 3 : développement piloté par les tests (TDD)

Le *Test Driven Development* (TDD) ou en français développement piloté par les tests est une technique de développement de logiciel qui préconise d'écrire les tests unitaires **avant** d'écrire le code source d'un logiciel.

Le cycle préconisé par TDD comporte cinq étapes :

1. écrire un premier test
2. vérifier qu'il échoue (car le code qu'il teste n'existe pas), afin de vérifier que le test est valide
3. écrire juste le code suffisant pour passer le test
4. vérifier que le test passe
5. *puis refactoriser le code, c'est-à-dire l'améliorer tout en gardant les mêmes fonctionnalités*

La refactorisation de code (*refactoring*) sera traitée à la séquence 4.

Question 4. Coder le test unitaire réalisé par la méthode `testMesureMin()`.

Question 5. Coder et valider la méthode `Mesure::getMesureMin()` en utilisant le programme de tests jusqu'à ce que la méthode passe le test unitaire.

À la fin de cette séquence, l'ensemble des méthodes de la classe `Mesure` doivent être validées et doivent "passer" les tests unitaires.

Séquence 4 : tests de non régression avec refactorisation du code

Les **tests de non régression** permettent, après chaque modification, correction ou adaptation du logiciel, de vérifier que le comportement des fonctionnalités n'a pas été perturbé, même lorsqu'elle ne sont pas concernées directement par la modification.

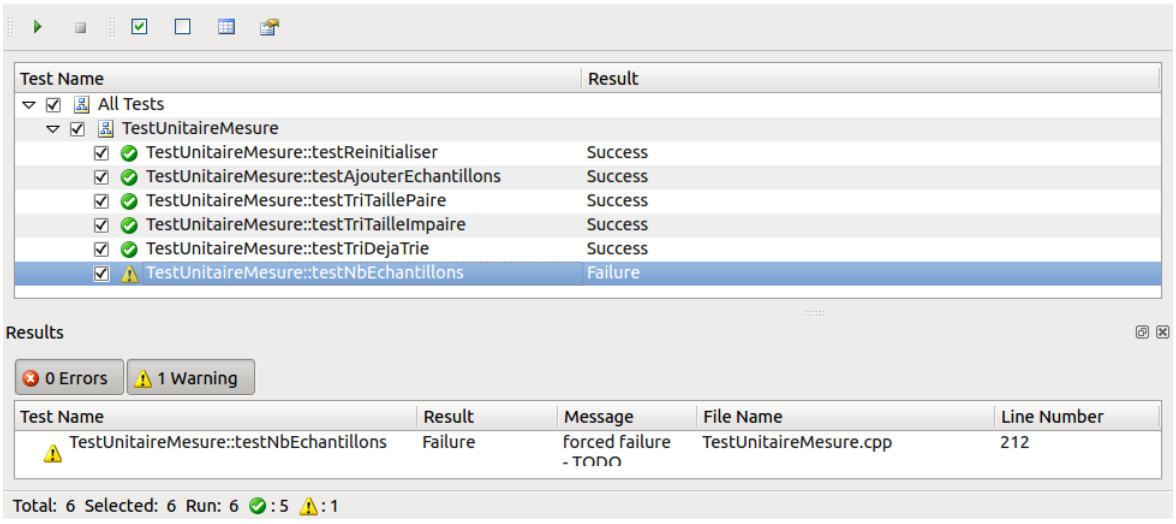
La refactorisation de code (*refactoring*) consiste à retravailler le code source sans ajouter de fonctionnalités au logiciel ni corriger de bogues, mais en améliorant sa lisibilité pour simplifier sa maintenance, ou le rendre plus générique. On parle aussi de remaniement.

En considérant ici que le nombre d'échantillons à trier est très faible, on vous demande d'implémenter un **tri par insertion** car il est considéré comme le tri le plus efficace sur des entrées de petite taille (cf. Tri_par_insertion).

Question 6. Refactoriser le code des méthodes `Mesure::trier()` puis `Mesure::extraireMax()` et `Mesure::extraireMin()` et réaliser les tests de non régression en relançant le programme de tests.

Séquence 5 : une GUI pour les tests

On va mettre en œuvre `qxcppunit`, une interface GUI (*Graphical User Interface*) sous Qt pour `CppUnit`.



Installation : `$ sudo apt-get install libqxcppunit-dev`

Le *framework* `QxCppUnit` fournit un programme de test générique pour Qt4 (le code source est fourni en **Annexe 5**).

Modifier le fichier de projet Qt `.pro` :

```
TEMPLATE = app
TARGET = testMesure
DEPENDPATH += .
INCLUDEPATH += . ../src
HEADERS += TestUnitaireMesure.h ../src/mesure.h
SOURCES += TestUnitaireMesure.cpp main.cpp ../src/mesure.cpp
LIBS += -lcppunit -ldl -lqxcppunitd
```

Question 7. Ajouter un test unitaire pour la méthode `nbEchantillons()` et lancer le programme de test sous Qt.

Annexe 1 : La définition de la classe Mesure à tester

```
#include "mesure.h"

Mesure::Mesure()
{
   reinitialiser();
}

Mesure::~Mesure()
{
}

void Mesure::reinitialiser()
{
   echantillons.clear();
   mesure = 0.;
   mesureMax = 0.;
   mesureMin = 0.;
}

double Mesure::getMesure()
{
   extraireMediane();
   return mesure;
}

double Mesure::getMesureMax()
{
   extraireMax();
   return mesureMax;
}

double Mesure::getMesureMin()
{
   // TODO : séquence 3
   return 0.;
}

double Mesure::getEchantillon(int pos)
{
   if(pos >= 0 && pos < nbEchantillons())
      return echantillons.at(pos);

   return 0.;
}

void Mesure::ajouterEchantillon(double echantillon)
{
   echantillons.push_back(echantillon);
}

void Mesure::ajouterEchantillons(vector<double> &t)
```



```
{
   for(vector<double>::iterator it = t.begin(); it != t.end(); ++it)
   {
       ajouterEchantillon(*it);
   }
}

void Mesure::ajouterEchantillons(double *t, int n)
{
   for(int i=0;i<n;i++)
   {
       ajouterEchantillon(t[i]);
   }
}

int Mesure::nbEchantillons() const
{
   return echantillons.size();
}

void Mesure::trier()
{
   if(!estTrie())
      sort(echantillons.begin(), echantillons.end());
}

bool Mesure::estTrie() const
{
   for(int i=1;i<nbEchantillons();i++)
      if(echantillons[i-1] > echantillons[i])
      {
         return false;
      }
   return true;
}

void Mesure::extraireMediane()
{
   int n = nbEchantillons();

   if(n == 0)
       return;

   trier();

   if(n%2)
   {
      /* n est impair */
      mesure = echantillons[((n+1)/2)-1];
   }
   else
   {
      /* n est pair */
```



```
      mesure = (echantillons[(n/2)-1] + echantillons[((n/2)+1)-1])/2;
   }
}

void Mesure::extraireMax()
{
   if(nbEchantillons() == 0)
      return;
   mesureMax = *std::max_element(echantillons.begin(), echantillons.end());
}

void Mesure::extraireMin()
{
   // TODO : séquence 3
}

ostream& operator<<(ostream &sortie, Mesure &m)
{
   sortie << "nb␣echantillons␣:\t" << m.nbEchantillons() << "\n";
   sortie << "echantillons␣:\t\t";
   if(m.nbEchantillons() == 0)
      sortie << "aucun";
   for (int i=0; i<m.nbEchantillons(); i++)
   {
      sortie << m.echantillons[i] << "␣";
   }
   sortie << "\n";

   sortie << "mesure␣:\t\t" << m.getMesure() << "\n";
   sortie << "mesure␣min␣:\t\t" << m.getMesureMin() << "\n";
   sortie << "mesure␣max␣:\t\t" << m.getMesureMax() << "\n";

   return sortie ;
}
```

Code 9 – mesure.cpp

Annexe 2 : le programme de tests principal pour CppUnit

Le code source “générique” d’un programme de tests CppUnit :

```
#include <cppunit/BriefTestProgressListener.h>
#include <cppunit/CompilerOutputter.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/TestResult.h>
#include <cppunit/TestResultCollector.h>
#include <cppunit/TestRunner.h>

int main( int argc, char* argv[] )
{
  // Create the event manager and test controller
  CPPUNIT_NS::TestResult controller;

  // Add a listener that colllects test result
  CPPUNIT_NS::TestResultCollector result;
  controller.addListener( &result );

  // Add a listener that print dots as test run.
  CPPUNIT_NS::BriefTestProgressListener progress;
  controller.addListener( &progress );

  // Add the top suite to the test runner
  CPPUNIT_NS::TestRunner runner;
  runner.addTest( CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest() );
  runner.run( controller );

  // Print test in a compiler compatible format.
  CPPUNIT_NS::CompilerOutputter outputter( &result, CPPUNIT_NS::stdCOut() );
  outputter.write();

  // On retourne le code d’erreur 1 si un test a échoué
  return result.wasSuccessful() ? 0 : 1;
}
```

Code 10 – main.cpp

Annexe 3 : la liste des assertions de CppUnit

```
Assertion with a user specified message.
message       Message reported in diagnostic if condition evaluates to false.
condition     If this condition evaluates to false then the test failed.
CPPUNIT_ASSERT_MESSAGE(message,condition)

Asserts that two values are equals.
CPPUNIT_ASSERT_EQUAL(expected,actual)
CPPUNIT_ASSERT_DOUBLES_EQUAL(expected,actual,delta)

Asserts that two values are equals, provides additional message on failure.
CPPUNIT_ASSERT_EQUAL_MESSAGE(message,expected,actual)
CPPUNIT_ASSERT_DOUBLES_EQUAL_MESSAGE(message,expected,actual,delta)

Asserts that an assertion fail.
CPPUNIT_ASSERT_ASSERTION_FAIL( assertion )
Example of usage:
       CPPUNIT_ASSERT_ASSERTION_FAIL( CPPUNIT_ASSERT( 1 == 2 ) );

Asserts that an assertion fail, with a user-supplied message in
CPPUNIT_ASSERT_ASSERTION_FAIL_MESSAGE( message, assertion )
Example of usage:
       CPPUNIT_ASSERT_ASSERTION_FAIL_MESSAGE( "1␣==␣2", CPPUNIT_ASSERT( 1 == 2 ) );

Asserts that an assertion pass.
CPPUNIT_ASSERT_ASSERTION_PASS( assertion )
Example of usage:
       CPPUNIT_ASSERT_ASSERTION_PASS( CPPUNIT_ASSERT( 1 == 1 ) );

Asserts that an assertion pass, with a user-supplied message in
CPPUNIT_ASSERT_ASSERTION_PASS_MESSAGE( message, assertion )
Example of usage:
       CPPUNIT_ASSERT_ASSERTION_PASS_MESSAGE( "1␣!=␣1", CPPUNIT_ASSERT( 1 == 1 ) );

Fails with the specified message (Always fails and abort current test with the given message
   ).
message       Message reported in diagnostic.
CPPUNIT_FAIL( message )
```


Annexe 4 : La définition de la classe TestUnitaireMesure

```
#include <cppunit/config/SourcePrefix.h>

#include "TestUnitaireMesure.h"
#include "mesure.h" // Classe à tester

// Enregistrement des différents cas de tests
CPPUNIT_TEST_SUITE_REGISTRATION( TestUnitaireMesure );

////////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////////

TestUnitaireMesure::TestUnitaireMesure()
{

}

TestUnitaireMesure::~TestUnitaireMesure()
{

}

void TestUnitaireMesure::setUp()
{
    // Initialisation pour les tests
    mesure = new Mesure;
}

void TestUnitaireMesure::tearDown()
{
    // fin du test
    delete mesure;
}

void TestUnitaireMesure::testReinitialiser()
{
    // Test 1

    // Appel méthode testée (lancement)
    mesure->reinitialiser();

    // Vérification des résultats attendus
    CPPUNIT_ASSERT_EQUAL( 0., mesure->getMesure() ); // valide
    CPPUNIT_ASSERT_EQUAL( 0., mesure->getMesureMin() ); // valide
    CPPUNIT_ASSERT_EQUAL( 0., mesure->getMesureMax() ); // valide
    CPPUNIT_ASSERT_EQUAL( 0, mesure->nbEchantillons() ); // valide

    // Test 2
    const int NB_ELT = 4;
    vector<double> echantillons(NB_ELT);
```

```
    echantillons[0] = 35.02;
    echantillons[1] = 35.10;
    echantillons[2] = 35.23;
    echantillons[3] = 35.53;

    mesure->ajouterEchantillons(echantillons);

    // Appel méthode testée (lancement)
    mesure->reinitialiser();

    // Vérification des résultats attendus
    CPPUNIT_ASSERT_EQUAL( 0., mesure->getMesure() ); // valide
    CPPUNIT_ASSERT_EQUAL( 0., mesure->getMesureMin() ); // valide
    CPPUNIT_ASSERT_EQUAL( 0., mesure->getMesureMax() ); // valide
    CPPUNIT_ASSERT_EQUAL( 0, mesure->nbEchantillons() ); // valide
}

void TestUnitaireMesure::testAjouterEchantillons()
{
    // Initialisation du test
    const int NB_ELT = 4;
    vector<double> echantillons(NB_ELT);

    echantillons[0] = 35.02;
    echantillons[1] = 35.10;
    echantillons[2] = 35.23;
    echantillons[3] = 35.53;

    mesure->reinitialiser();

    // Appel méthode testée (lancement)
    mesure->ajouterEchantillons(echantillons);

    // Vérification des résultats attendus
    CPPUNIT_ASSERT_EQUAL( 0., mesure->getEchantillon(-1) ); // invalide
    CPPUNIT_ASSERT_EQUAL( echantillons[0], mesure->getEchantillon(0) ); // valide
    CPPUNIT_ASSERT_EQUAL( echantillons[1], mesure->getEchantillon(1) ); // valide
    CPPUNIT_ASSERT_EQUAL( echantillons[2], mesure->getEchantillon(2) ); // valide
    CPPUNIT_ASSERT_EQUAL( echantillons[3], mesure->getEchantillon(3) ); // valide
    CPPUNIT_ASSERT_EQUAL( 0., mesure->getEchantillon(4) ); // invalide
}

void TestUnitaireMesure::testTriTaillePaire()
{
    // 1. Initialisation du jeu de test :
    const int NB_ELT = 6;
    vector<double> echantillons(NB_ELT);
    bool resultatAttendu; // expected
    bool resultatObtenu; // actual

    echantillons[0] = 35.23;
    echantillons[1] = 35.53;
    echantillons[2] = 35.02;
```

```
    echantillons[3] = 35.10;
    echantillons[4] = 34.45;
    echantillons[5] = 4.56;

    mesure->reinitialiser();
    mesure->ajouterEchantillons(echantillons);

    // 2. Définit le résultat attendu : true -> Le tableau doit être trié
    resultatAttendu = true; // expected

    // 3. Appel méthode testée (lancement) :
    mesure->trier();

    // 4. Vérification des résultats attendus (version détaillée) :
    resultatObtenu = mesure->estTrie();

    // Avec message :
    //CPPUNIT_ASSERT_MESSAGE( "classe valide : le vector doit être trié (taille paire)",
        resultatAttendu == resultatObtenu);
    // ou : CPPUNIT_ASSERT_EQUAL_MESSAGE(message, expected, actual);
    CPPUNIT_ASSERT_EQUAL_MESSAGE( "classe␣valide␣:␣le␣vector␣doit␣être␣trié␣(taille␣paire)",
         resultatAttendu, resultatObtenu);

    // Sans message :
    //CPPUNIT_ASSERT( resultatObtenu == resultatAttendu );
    // ou : CPPUNIT_ASSERT_EQUAL(expected, actual);
    CPPUNIT_ASSERT_EQUAL( resultatAttendu, resultatObtenu );

    // Vérifions que les échantillons sont maintenant triés
    CPPUNIT_ASSERT_EQUAL( echantillons[5], mesure->getEchantillon(0) );
    CPPUNIT_ASSERT_EQUAL( echantillons[4], mesure->getEchantillon(1) );
    CPPUNIT_ASSERT_EQUAL( echantillons[2], mesure->getEchantillon(2) );
    CPPUNIT_ASSERT_EQUAL( echantillons[3], mesure->getEchantillon(3) );
    CPPUNIT_ASSERT_EQUAL( echantillons[0], mesure->getEchantillon(4) );
    CPPUNIT_ASSERT_EQUAL( echantillons[1], mesure->getEchantillon(5) );
}

void TestUnitaireMesure::testTriTailleImpaire()
{
    // Initialisation du test
    const int NB_ELT = 5;
    vector<double> echantillons(NB_ELT);

    echantillons[0] = 35.23;
    echantillons[1] = 35.53;
    echantillons[2] = 35.02;
    echantillons[3] = 35.10;
    echantillons[4] = 34.45;

    mesure->reinitialiser();
    mesure->ajouterEchantillons(echantillons);

    // Appel méthode testée (lancement)
```

```
    mesure->trier();

    // Vérification des résultats attendus
    // Le tableau doit être trié
    CPPUNIT_ASSERT_EQUAL( true, mesure->estTrie() );

    // Vérifions que les échantillons sont maintenant triés
    CPPUNIT_ASSERT_EQUAL( echantillons[4], mesure->getEchantillon(0) );
    CPPUNIT_ASSERT_EQUAL( echantillons[2], mesure->getEchantillon(1) );
    CPPUNIT_ASSERT_EQUAL( echantillons[3], mesure->getEchantillon(2) );
    CPPUNIT_ASSERT_EQUAL( echantillons[0], mesure->getEchantillon(3) );
}

void TestUnitaireMesure::testTriDejaTrie()
{
    // Initialisation du test
    const int NB_ELT = 4;
    vector<double> echantillons(NB_ELT);

    echantillons[0] = 35.02;
    echantillons[1] = 35.10;
    echantillons[2] = 35.23;
    echantillons[3] = 35.53;

    mesure->reinitialiser();
    mesure->ajouterEchantillons(echantillons);

    // Appel méthode testée (lancement)
    mesure->trier();

    // Vérification des résultats attendus
    // Le tableau doit être trié
    CPPUNIT_ASSERT_EQUAL( true, mesure->estTrie() );
}

// TODO : etc ...
```

Code 12 – TestUnitaireMesure.cpp

Annexe 5 : le programme de tests principal pour QxCppUnit

Le code source “générique” d’un programme de tests QxCppUnit :

```
#include <QApplication>
#include <qxcppunit/testrunner.h>
#include <cppunit/extensions/TestFactoryRegistry.h>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);

   QxCppUnit::TestRunner runner;

   runner.addTest(CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest());
```

```
   runner.run();

   return 0;
}
```

Code 13 – main.cpp

