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 est quand même exempt 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 oeuvre d'un test unitaire

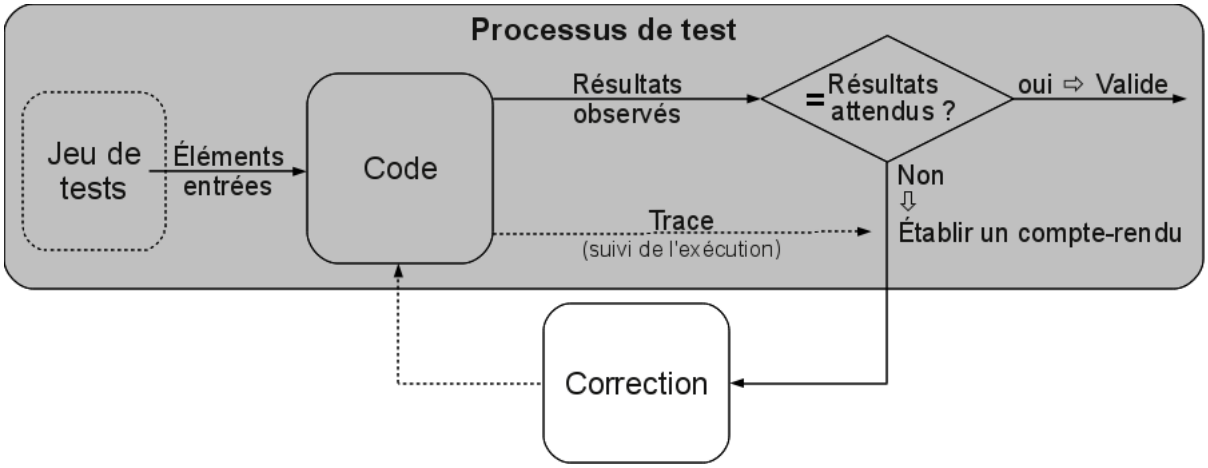
Principe

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 :

- `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.

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

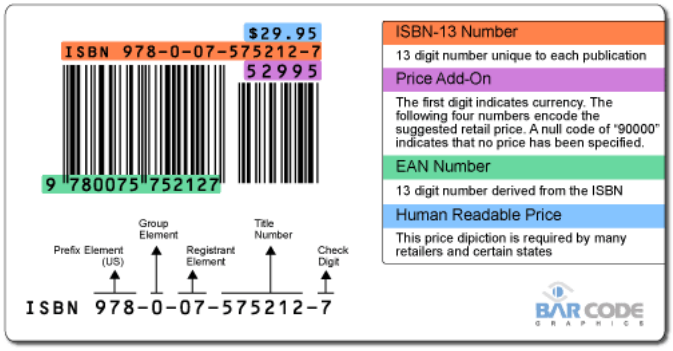
Objectifs

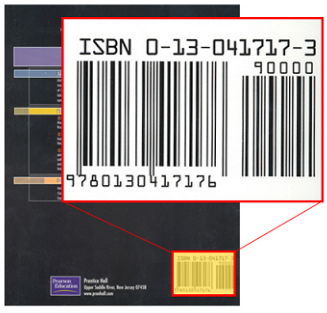
Les objectifs de ce TP sont de mettre en oeuvre les procédures de tests unitaires en utilisant le *framework* `CppUnit`.

La classe `ISBN` à tester

Dans le cadre d'une application de gestion de livres, on a besoin de **valider une classe `ISBN`**.

L'ISBN (*International Standard Book Number*) est un numéro international qui permet d'identifier, de manière unique, chaque édition de chaque livre publié, que son support soit numérique ou sur papier. Il est destiné à simplifier la gestion informatique pour tous les intervenants de la chaîne du livre (imprimeur, éditeur, libraire, bibliothèque, etc.).





Avant 2007, le numéro **ISBN-10** se composait de quatre segments, trois segments de longueur variable et un segment de longueur fixe, la longueur totale de l'ISBN comprenait **10 chiffres**. Depuis le 1er janvier 2007, la longueur a été étendue à 13 chiffres en ajoutant un groupe initial de 3 chiffres.

Pour faciliter leur gestion informatique, chaque livre porte un code à barres à la norme **EAN 13** et un code **ISBN-13** dont il est dérivé.

Ce code comporte **13 chiffres** et est aujourd'hui **obligatoire** (depuis janvier 2007), et doit être utilisé pour tous les nouveaux codes ISBN à la place du code à 10 chiffres. En effet, l'ancienne numérotation est arrivée à saturation et ne permettrait plus d'attribuer simplement des groupes de codes spécifiques aux différents éditeurs.

La conversion d'un ancien code ISBN-10 en code ISBN-13 compatible avec la norme EAN est automatique (et obligatoire pour toutes les transactions électroniques à compter de janvier 2007).

La classe `ISBN` possèdera donc un attribut `isbn` de type `string` pour conserver le code. Le type `string` est le meilleur choix car :

- les codes ISBN sont structurés par segment et il sera donc plus simple de traiter une chaîne de caractère qu'un nombre
- les codes ISBN séparent les différents segments par un tiret : `"2-266-11156-6"` correspond au code `2266111566`

La classe `ISBN` permettra de gérer des codes ISBN-10 et des codes ISBN-13.

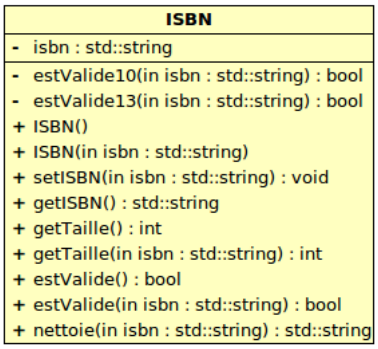


FIGURE *1 – La classe à tester*

Les codes ISBN comportent une **clé de contrôle** calculée à partir des 9 chiffres précédents pour un code ISBN-10 et des 12 chiffres précédents pour un code ISBN-13.

La classe `ISBN` offrira les services suivants :

- une méthode `nettoie()` qui reçoit un code isbn de type `string` et qui retourne le code isbn sans les tirets (pour un code ISBN-10, `"2-266-11156-6"` → `"2266111566"`)
- les méthodes `getTaille()` qui retournent le nombre de chiffres contenus dans un code isbn "nettoyé" de type `string` (pour un code ISBN-10, `"2-266-11156-6"` → `10`)
- les méthodes `estValide()` qui retournent vrai (`true`) pour un code isbn valide (en taille, en chiffres et en clé de contrôle) sinon elles retournent faux (`false`) : pour un code ISBN-10, `"2-266-11156-6"` → `true`

Si le code ISBN n'est pas valide (en taille, en chiffres ou en clé de contrôle), il ne doit pas être stocké dans l'attribut `isbn`. Dans ce cas, une chaîne vide sera affectée à l'attribut. Ce comportement devra être implémenté pour le constructeur et le *setter* `setISBN()`. Si il est valide, il sera conservé dans sa forme initiale non "nettoyée" (`"2-266-11156-6"` par exemple).

La déclaration de la classe `ISBN` :

```
#ifndef ISBN_H
#define ISBN_H

#include <iostream>

// Longueur des codes
#define ISBN_10 10
#define ISBN_13 13

class ISBN
{
   private:
      std::string isbn;

      // Vérifie un code ISBN-10
      bool estValide10(const std::string &isbn) const;
      // Vérifie un code ISBN-13
      bool estValide13(const std::string &isbn) const;

   public:
      ISBN();
      ISBN(const std::string &isbn);

      // Setter
      void setISBN(const std::string &isbn);
      std::string getISBN() const;

      // Services
      int getTaille() const;
      int getTaille(const std::string &isbn) const;
      bool estValide() const;
      bool estValide(const std::string &isbn) const;
      std::string nettoie(const std::string &isbn) const;
};

#endif
```

Code 2 – isbn.h

Les quatre segments d'un ancien code ISBN à 10 chiffres sont : A - B - C - **D**
- A identifie un groupe de codes par pays, zone géographique ou zone linguistique
- B identifie l'éditeur de la publication
- C correspond au numéro d'ordre de l'ouvrage chez l'éditeur qui l'attribue normalement séquentiellement
- **D est un code clé de vérification** sur un caractère calculé à partir des chiffres précédents. Outre les chiffres de 0 à 9, la clé de contrôle peut prendre la valeur '`X`', qui représente le nombre 10. Ce code est calculé de la façon suivante :
  - on attribue une pondération à chaque position (de 1 à 9 en allant en sens croissant) et on fait la somme des produits ainsi obtenus ;
  - on conserve le reste de la division euclidienne de ce nombre par 11 ;
  - si le reste de la division euclidienne est 10, la clé de contrôle n'est pas 10 mais la lettre `X`. Ceci permet donc d'avoir les codes 0, 1, 2, ..., 8, 9, `X`.

Remarques :
- 11 étant un nombre premier, une erreur portant sur un chiffre entraînera automatiquement une incohérence du code de contrôle.
- La vérification du code de contrôle peut se faire en effectuant le même calcul sur le code ISBN complet, en appliquant la pondération 10 au dixième chiffre de la clé de contrôle (si ce chiffre clé est `X`, on lui attribue la valeur 10) : la somme pondérée doit alors être un multiple de 11.

Lire : http ://fr.wikipedia.org/wiki/International_Standard_Book_Number

```
#include "isbn.h"

/**
   Vérifie un code ISBN-10
*/
bool ISBN::estValide10(const std::string &isbn) const
{
   int ponderation = 10;
   int somme = 0;
   int codeDeControle = 0;

   std::string _isbn = nettoie(isbn); // enleve les tirets

   /* seulement ISBN 10 */
   if(_isbn.size() != ISBN_10)
   {
      std::cerr << "Le code ISBN n'a pas la bonne taille (ISBN 10 accepté)" << std::endl;
      return false;
   }

   /* pour les 9 premiers chiffres du code ISBN-10 */
   for(int i = 0; i < (ISBN_10 - 1); i++)
   {
      /* seulement des chiffres */
      if(!isdigit(_isbn[i]))
         return false;
      int code = _isbn[i] - '0'; // convertit en chiffre
      somme = somme + (code * ponderation);
      ponderation--;
   }

   codeDeControle = (11 - (somme % 11)) % 11;

   // la clé de contrôle peut prendre la valeur X !
   if(codeDeControle == 10 && _isbn[(ISBN_10 - 1)] == 'X')
      return true;

   // vérifie la clé de contrôle
   if(codeDeControle == _isbn[(ISBN_10 - 1)])
      return true;

   return false;
}
```

```
/**
   Nettoie un code ISBN en enlevant les tirets
*/
std::string ISBN::nettoie(const std::string &isbn) const
{
   std::string _isbn = "";

   for (unsigned int i = 0; i < isbn.size(); i++)
   {
      if(isbn[i] != '-')
      {
         _isbn += isbn[i];
      }
   }

   return _isbn;
}

...
```

La méthode `estValide10()` comporte au moins une erreur !

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

Séquence 1 : la classe de test `TestUnitaireISBN`

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

La déclaration de la classe `TestUnitaireISBN` est la suivante :

```
#ifndef _TESTUNITAIREISBN_H
#define _TESTUNITAIREISBN_H

#include <cppunit/extensions/HelperMacros.h>

class ISBN; // La classe à tester

class TestUnitaireISBN : public CPPUNIT_NS::TestFixture
{
  // On crée une suite de tests unitaires pour la classe
  CPPUNIT_TEST_SUITE( TestUnitaireISBN );
  CPPUNIT_TEST( testEstValideISBN10 );
  CPPUNIT_TEST( testEstValideISBN13 );
  CPPUNIT_TEST( testSetterISBN10 );
  CPPUNIT_TEST( testSetterISBN13 );
  CPPUNIT_TEST( testNettoieISBN );
  CPPUNIT_TEST_SUITE_END();

private:
   ISBN *isbn; // un pointeur sur une instance de la classe à tester

public:
   TestUnitaireISBN();
   virtual ~TestUnitaireISBN();

   // Call before tests
   void setUp();
   // Call after tests
   void tearDown();

   // Liste des tests
   void testEstValideISBN10();
   void testEstValideISBN13();
   void testSetterISBN10();
   void testSetterISBN13();
   void testNettoieISBN();
};

#endif
```

Code 4 – TestUnitaireISBN.h

La définition de la classe `TestUnitaireISBN` :

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

#include "TestUnitaireISBN.h"
#include "isbn.h" // Classe à tester

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

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

TestUnitaireISBN::TestUnitaireISBN()
{

}

TestUnitaireISBN::~TestUnitaireISBN()
{

}

void TestUnitaireISBN::setUp()
{
   // Initialisation pour les tests
   isbn = new ISBN; // instancie un objet ISBN à tester
}

void TestUnitaireISBN::tearDown()
{
   delete isbn; // libère la mémoire allouée à l'objet ISBN
}

void TestUnitaireISBN::testEstValideISBN10()
{
   // Initialisation du test

   // Vérification des résultats attendus
   // Classes valides :
   // Ici on peut prendre un échantillon pour chaque clé de contrôle possible
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("9-604-25059-0") );
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("2-100-03781-1") );
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("2-123-45680-2") );
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("1-932-69818-3") );
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("2-702-13411-4") );
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("0-684-84328-5") );
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("2-266-11156-6") );
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("1-402-89462-7") );
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("2-212-08774-8") );
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("2-765-40912-9") );
   // Et sans les tirets :
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("2266111566") );
```



```
   // Classes invalides :
   CPPUNIT_ASSERT_EQUAL( false, isbn->estValide("2-266-11156-5") ); // mauvaise code de
      controle
   CPPUNIT_ASSERT_EQUAL( false, isbn->estValide("206-11156-6") ); // mauvaise longueur
   CPPUNIT_ASSERT_EQUAL( false, isbn->estValide("2-XXX-11156-6") ); // mauvais codes
   CPPUNIT_ASSERT_EQUAL( false, isbn->estValide("") ); // code vide

   // Technique détaillée :
   bool resultatObtenu;
   bool resultatAttendu = true;

   // Appel méthode testée (lancement)
   resultatObtenu = isbn->estValide("2-266-11156-6");

   // Vérification des résultats attendus
   CPPUNIT_ASSERT_MESSAGE( "classe␣valide␣:␣code␣ISBN-10␣valide", resultatObtenu ==
      resultatAttendu);
   // ou : CPPUNIT_ASSERT_EQUAL_MESSAGE(message, expected, actual);
   CPPUNIT_ASSERT_EQUAL( resultatAttendu, isbn->estValide("2-266-11156-6") );
   // ou :
   //CPPUNIT_ASSERT( isbn->estValide("2-266-11156-6") == resultatAttendu );
   // etc ...
}

void TestUnitaireISBN::testEstValideISBN13()
{
   // Initialisation du test

   // Vérification des résultats attendus
   // Classe valide :
   CPPUNIT_ASSERT_EQUAL( true, isbn->estValide("978-2-86889-006-1") );
   // etc ...

   // Classes invalides :
   CPPUNIT_ASSERT_EQUAL( false, isbn->estValide("978-2-86889-006-5") ); // mauvaise code de
      controle
   CPPUNIT_ASSERT_EQUAL( false, isbn->estValide("78-2-86889-006-1") ); // mauvaise longueur
   CPPUNIT_ASSERT_EQUAL( false, isbn->estValide("978-2-86889-AAA-1") ); // mauvais codes
   CPPUNIT_ASSERT_EQUAL( false, isbn->estValide("908-2-86889-006-1") ); // mauvais
      identifiants attribués aux livres dans la codification EAN
   CPPUNIT_ASSERT_EQUAL( false, isbn->estValide("") ); // code vide
}

void TestUnitaireISBN::testSetterISBN10()
{
   // Classe valide :
   isbn->setISBN("2-266-11156-6");
   CPPUNIT_ASSERT( isbn->getISBN() == "2-266-11156-6" );
   // etc ...

   // Classes invalides :
   isbn->setISBN("2-266-11156-5"); // mauvaise code de controle
```

```
   CPPUNIT_ASSERT( isbn->getISBN() == "" );
   isbn->setISBN("206-11156-6"); // mauvaise longueur
   CPPUNIT_ASSERT( isbn->getISBN() == "" );
   isbn->setISBN("2-XXX-11156-6"); // mauvais codes
   CPPUNIT_ASSERT( isbn->getISBN() == "" );
   isbn->setISBN(""); // code vide
   CPPUNIT_ASSERT( isbn->getISBN() == "" );
}

void TestUnitaireISBN::testSetterISBN13()
{
   // TODO
   CPPUNIT_FAIL( "TODO" ); // on fait échouer le test volontairement en attendant de l'
      écrire
}

void TestUnitaireISBN::testNettoieISBN()
{
   std::string codeISBN;
   // Classe valide :
   codeISBN = isbn->nettoie("2-266-11156-6");
   CPPUNIT_ASSERT( codeISBN == "2266111566" );
   codeISBN = isbn->nettoie("978-2-86889-006-1");
   CPPUNIT_ASSERT( codeISBN == "9782868890061" );
}

// TODO : d'autres tests etc ...
```

Code 5 – TestUnitaireISBN.cpp

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

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/testISBN
TestUnitaireISBN::testEstValideISBN10 : assertion
TestUnitaireISBN::testEstValideISBN13 : assertion
TestUnitaireISBN::testSetterISBN10 : assertion
TestUnitaireISBN::testSetterISBN13 : assertion
TestUnitaireISBN::testNettoieISBN : OK
TestUnitaireISBN.cpp:40:Assertion
Test name: TestUnitaireISBN::testEstValideISBN10
equality assertion failed
- Expected: 1
- Actual : 0

TestUnitaireISBN.cpp:70:Assertion
Test name: TestUnitaireISBN::testEstValideISBN13
equality assertion failed
```

```
- Expected: 1
- Actual : 0

TestUnitaireISBN.cpp:84:Assertion
Test name: TestUnitaireISBN::testSetterISBN10
assertion failed
- Expression: isbn->getISBN() == "2-266-11156-6"

TestUnitaireISBN.cpp:100:Assertion
Test name: TestUnitaireISBN::testSetterISBN13
forced failure
- TODO

Failures !!!
Run: 5  Failure total: 4 Failures: 4  Errors: 0
```

Bilan : 1 test sur 5 est passé ! Seule la méthode `nettoie()` a été validée par son test unitaire. Le test unitaire de validation d'un code ISBN-13 a échoué normalement car la méthode `estValideISBN13()` n'a pas encore été implémentée (cf. séquence 2).

Par contre, la méthode `ISBN::estValideISBN10()` comporte un défaut. Ce défaut provoque aussi une panne dans les méthodes `get/set` de l'attribut `isbn`.

Question 1. Corriger le défaut détecté dans la méthode `ISBN::estValideISBN10()` et relancer le programme de test jusqu'à la disparition de tout défaut pour valider unitairement cette méthode.

Question 2. Le jeu de test pour valider la méthode `ISBN::estValideISBN10()` est incomplet car il manque une classe valide à tester. À partir du code source `ISBN::estValideISBN10()`, compléter la méthode `testEstValideISBN10()` pour qu'elle couvre tous les classes valides.

Séquence 2 : 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 3.

Question 3. Coder le test unitaire réalisé par la méthode `testSetterISBN13()`.

Une code ISBN-13 comporte 13 chiffres et est aujourd'hui obligatoire (depuis janvier 2007), et doit être utilisé pour tous les nouveaux codes ISBN à la place du code à 10 chiffres. En effet, l'ancienne numérotation est arrivée à saturation et ne permettrait plus d'attribuer simplement des groupes de codes spécifiques aux différents éditeurs.

La conversion d'un ancien code ISBN-10 en code ISBN-13 compatible avec la norme EAN est automatique (et obligatoire pour toutes les transactions électroniques à compter de janvier 2007). Ce code comporte 13 chiffres, et est calculé à partir du numéro ISBN-10 de la façon suivante :

- les trois premiers chiffres valent « 978 » (978 est le premier des identifiants attribués aux livres dans la codification EAN) ou « 979 » ;
- les neuf chiffres suivants sont les neuf premiers chiffres de l'ISBN-10 (code de la zone géographique, code de l'éditeur, numérotation interne à l'éditeur) ;
- le dernier chiffre (c13) est une clé de contrôle calculée en fonction des 12 premiers chiffres en calculant le reste de la division par 10 (le modulo 10) de la différence entre 10 et le modulo 10 de la somme de chaque chiffre, chaque chiffre étant pondéré selon un indice de position égal à 1 pour les positions impaires et 3 pour les positions paires, soit suivant la formule :
  c13 = modulo(10 - modulo(c1 + 3 × c2 + c3 + 3 × c4 + ... + c11 + 3 × c12, 10), 10),
  - le chiffre clé obtenu ne peut varier que de 0 à 9 (il n'y a plus de chiffre `X`),
  - la vérification de la clé de contrôle peut se faire en vérifiant que la somme pondérée (calculée sur les 13 chiffres) est bien un multiple de 10.

Lire : http ://fr.wikipedia.org/wiki/International_Standard_Book_Number

Question 4. Coder et valider la méthode `ISBN::estValideISBN13()` 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 `ISBN` doivent être validées et "passer" les tests unitaires.

Séquence 3 : tests de non régression avec refactorisation du code

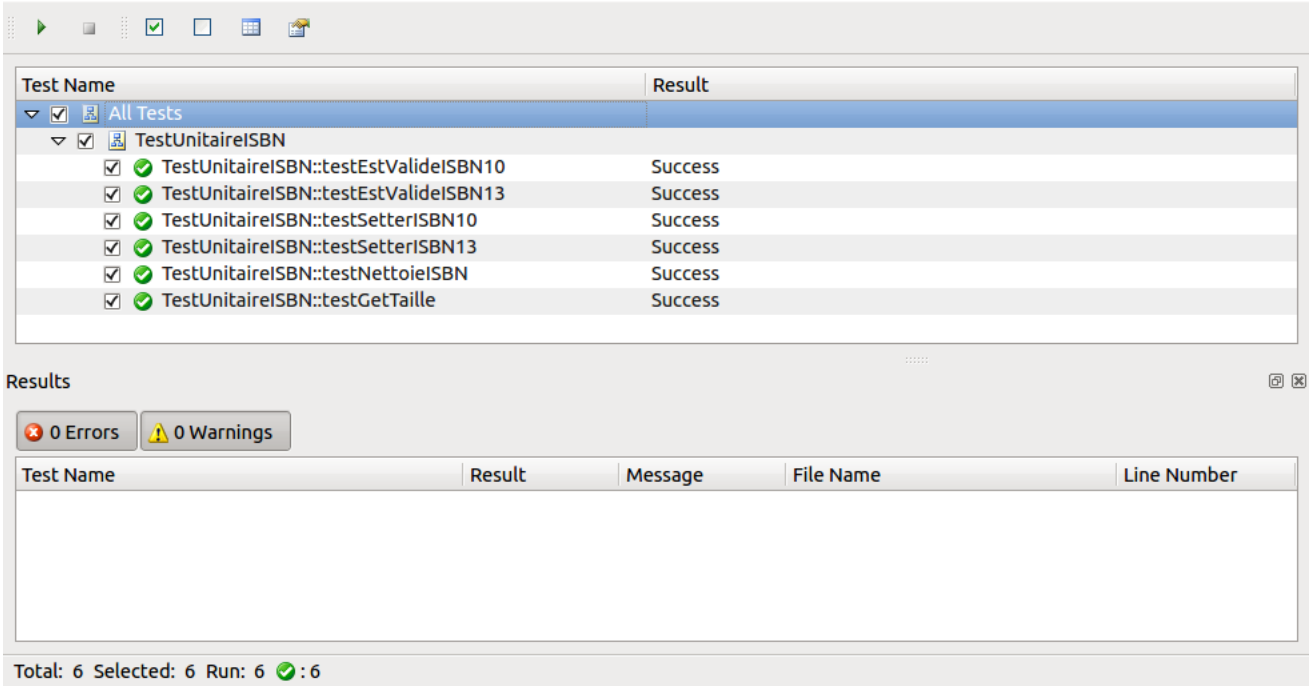
Les **tests de non régression** permettent, après chaque modification, correction ou adaptation du logiciel, de vérifier que le comportement des fonctionnalités n'a pas été perturbé, même lorsqu'elle ne sont pas concernées directement par la modification.

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.

Question 5. Le code du constructeur `ISBN::ISBN()` et de la méthode `ISBN::setISBN()` peut être amélioré. Refactoriser le code de ces méthodes et réaliser les tests de non régression en relançant le programme de tests.

Séquence 4 : une GUI pour les tests

On va mettre en oeuvre `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 4).

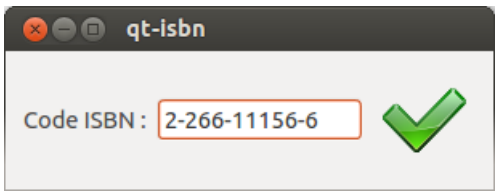
Modifier le fichier de projet Qt `.pro` :

```
TEMPLATE = app
TARGET = testISBN
DEPENDPATH += .
INCLUDEPATH += . ../src

HEADERS += TestUnitaireISBN.h ../src/isbn.h
SOURCES += TestUnitaireISBN.cpp main.cpp ../src/isbn.cpp

LIBS += -lcppunit -ldl -lqxcppunitd
```

Question 6. Ajouter un test unitaire pour la méthode `getTaille()` et lancer le programme de test sous Qt.

Séquence 5 : une application sous Qt

Une fois la **classe ISBN validée**, on peut l'exploiter dans un programme d'exemple sous Qt :



Un code ISBN-10 valide

Un code ISBN-10 invalide

On va créer un nouveau *widget* qui intégre la saisie et la validation automatique d'un code ISBN :

```
#ifndef MYWIDGET_H
#define MYWIDGET_H

#include <QtGui>
class ISBN;

class MyWidget : public QWidget {
    Q_OBJECT
    private:
        QLineEdit *saisieISBN;
        QLabel *etatISBN;
        ISBN *isbn;
        void setRouge();
        void setVert();

    public:
        MyWidget( QWidget *parent = 0 );
        virtual ~MyWidget();

    public slots:
        void valider( const QString & code );
};

#endif
```

Code 8 – mywidget.h

La définition de la classe `MyWidget` à compléter :

```
#include "mywidget.h"
#include "isbn.h"

MyWidget::MyWidget( QWidget *parent ) : QWidget( parent )
{
   // Création des widgets
   saisieISBN = new QLineEdit( this );
   etatISBN = new QLabel( this );

   QLabel *label1 = new QLabel( "Code␣ISBN␣:", this );

   // Création des layouts
   QHBoxLayout *mainLayout = new QHBoxLayout;
```

```
   // Mise en place des widgets et des layouts
   mainLayout->addWidget(label1);
   mainLayout->addWidget(saisieISBN);
   mainLayout->addWidget(etatISBN);
   setLayout(mainLayout);

   // Initialisation
   isbn = new ISBN();
   if(isbn->estValide())
      setVert();
   else
      setRouge();

   // Connexion des signaux/slots
   connect( saisieISBN, SIGNAL(textChanged ( const QString & )), this, SLOT(valider( const
      QString & )) );
}

MyWidget::~MyWidget()
{
    delete isbn;
}

void MyWidget::valider( const QString & code )
{
    // TODO
}

void MyWidget::setRouge()
{
    QImage *image;
    QPixmap pixmap;

    image = new QImage();
    if(!(image->load("rouge.png")))
    {
       qWarning("Fichier␣introuvable␣!");
    }

    pixmap = QPixmap::fromImage(*image);
    etatISBN->setPixmap(pixmap);

    delete image;
}

void MyWidget::setVert()
{
    QImage *image;
    QPixmap pixmap;

    image = new QImage();
    if(!(image->load("vert.png")))
```

```
    {
       qWarning("Fichier␣introuvable␣!");
    }

    pixmap = QPixmap::fromImage(*image);
    etatISBN->setPixmap(pixmap);
    delete image;
}
```

Code 9 – mywidget.cpp

Question 7. Compléter la définition de la classe `MyWidget` (le *slot* `valider()`) puis fabriquer un exécutable.

Annexe 1 : La définition de la classe ISBN à tester

```
#include "isbn.h"

/* Lire : http://fr.wikipedia.org/wiki/International_Standard_Book_Number */

/**
   Vérifie un code ISBN-10
*/
bool ISBN::estValide10(const std::string &isbn) const
{
   int ponderation = 10;
   int somme = 0;
   int codeDeControle = 0;

   std::string _isbn = nettoie(isbn);

   /* seulement ISBN 10 */
   if(_isbn.size() != ISBN_10)
   {
      std::cerr << "Le code ISBN n'a pas la bonne taille (ISBN 10 accepté)" << std::endl;
      return false;
   }

   for(int i = 0; i < (ISBN_10 - 1); i++)
   {
      /* seulement des chiffres */
      if(!isdigit(_isbn[i]))
         return false;
      int code = _isbn[i] - '0'; // convertit en chiffre (char en int)
      somme = somme + (code * ponderation);
      ponderation--;
   }

   codeDeControle = (11 - (somme % 11)) % 11;

   if(codeDeControle == 10 && _isbn[(ISBN_10 - 1)] == 'X')
      return true;

   if(codeDeControle == _isbn[(ISBN_10 - 1)])
      return true;

   return false;
}

/**
   Vérifie un code ISBN-13
*/

bool ISBN::estValide13(const std::string &isbn) const
{
   // TODO
```



```
   return false;
}

/**
   Nettoie un code ISBN en enlevant les tirets
*/
std::string ISBN::nettoie(const std::string &isbn) const
{
   std::string _isbn = "";

   for (unsigned int i = 0; i < isbn.size(); i++)
   {
      if(isbn[i] != '-')
      {
         _isbn += isbn[i];
      }
   }

   return _isbn;
}

bool ISBN::estValide() const
{
   if(getTaille() == ISBN_10)
      return estValide10(this->isbn);
   if(getTaille() == ISBN_13)
      return estValide13(this->isbn);
   return false;
}

bool ISBN::estValide(const std::string &isbn) const
{
   if(getTaille(isbn) == ISBN_10)
      return estValide10(isbn);
   if(getTaille(isbn) == ISBN_13)
      return estValide13(isbn);
   return false;
}

ISBN::ISBN()
{
   this->isbn = "";
}

ISBN::ISBN(const std::string &isbn)
{
   if(nettoie(isbn).size() == ISBN_10 && estValide10(nettoie(isbn)))
   {
      this->isbn = isbn;
      return;
   }
   if(nettoie(isbn).size() == ISBN_13 && estValide13(nettoie(isbn)))
   {
```



```
      this->isbn = isbn;
      return;
   }
   this->isbn = "";
}

void ISBN::setISBN(const std::string &isbn)
{
   if(nettoie(isbn).size() == ISBN_10 && estValide10(nettoie(isbn)))
   {
      this->isbn = isbn;
      return;
   }
   if(nettoie(isbn).size() == ISBN_13 && estValide13(nettoie(isbn)))
   {
      this->isbn = isbn;
      return;
   }
   this->isbn = "";
}

std::string ISBN::getISBN() const
{
   return isbn;
}

int ISBN::getTaille() const
{
   return nettoie(isbn).size();
}

int ISBN::getTaille(const std::string &isbn) const
{
   return nettoie(isbn).size();
}
```

Code 10 – isbn.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 11 – main.cpp

Annexe 3 : 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 : 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

