
Module R3.04

Chapitre 11

Petite Introduction aux Tests Unitaires
Google Test

11.1. Tests (1)

- Il est très important de rédiger des **tests pour confirmer la validité** du code que l'on produit.
- Il s'agit d'un **exercice difficile** qui requiert autant d'attention que la phase de spécification.
- Écrire des tests **prend du temps** (pour tester une certaine quantité de code, il faut produire une quantité équivalente de tests).
- Le bénéfice apporté n'est pas immédiat. Mais **à long terme, la productivité s'en trouve améliorée**.
- Une application soigneusement testée est plus **stable et plus fiable**.
- Au cours de la durée de vie d'un projet, les opérations réalisées sont susceptibles de changer, sans pour autant que les API ne soient modifiées : **les tests unitaires sont là pour garantir le bon fonctionnement de votre travail à tout instant** (non régression)
- **Les tests constituent également une forme de spécification** et de documentation d'une application.
- **Maintenir et déboguer** une application pour laquelle une batterie de tests existe sera beaucoup plus facile pour un développeur qui reprend le code d'un autre.

11.1. Tests (2) – Vieux sondage sur 460 entreprises

- 13%: Il n'y a pas de tests unitaires
- 46%: Les tests unitaires sont « informels »
- 11% : Les cas de tests unitaires sont documentés
- 16% : Les cas de tests unitaires et leurs exécutions sont documentés
- 14% : Utilisation du **Test Driven Development**

Explication :

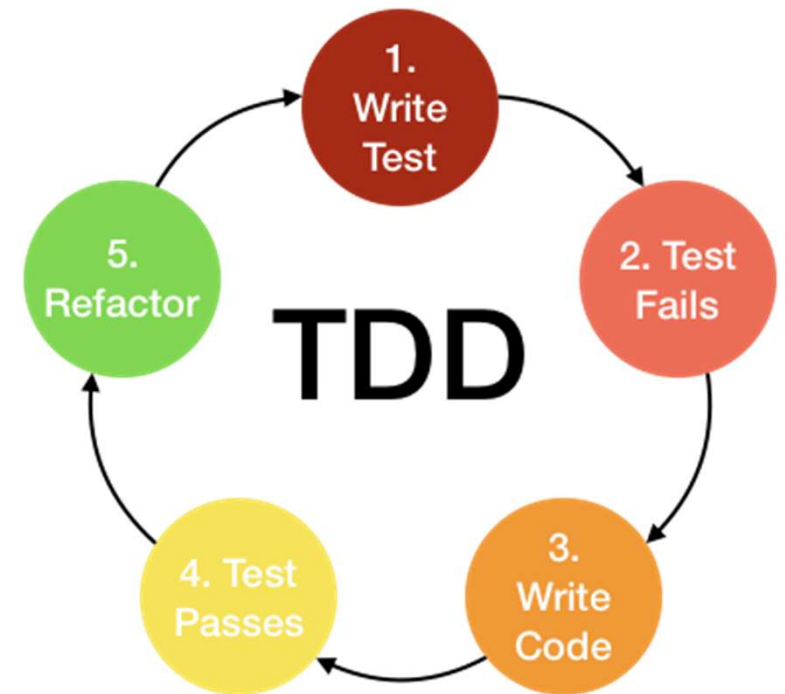
- Culture du test peu répandue chez les (vieux) développeurs
- Mise en place de tests coûteuse (temps de développement doublé)
- ... Mais la situation générale s'améliore !

11.2. TDD : Test Driven Development (1)

- Méthode qui préconise d'écrire les tests **AVANT** d'écrire le code qui sera testé
- **TDD** aide à spécifier un traitement en se focalisant sur le résultat qui doit être produit et non pas sur la façon de le produire...
- L'approche **TDD** est préconisée dans toutes les méthodes de développement dites "**agiles**" (**RAD, DSDM, UP, RUP, XP** ...) qui ont pour objectif de réduire le cycle de vie du logiciel :
 - Développement rapide d'une 1ère version minimale
 - Intégration de fonctionnalités supplémentaires, par un processus itératif, en fonction des besoins exprimés au fur et à mesure par le client
- Dans le cadre d'un développement itératif, il est important de pouvoir faire tourner facilement les tests à chaque itération afin de vérifier que le développement de la dernière version n'a pas fait "régresser" l'application !

11.2. TDD (2) : cycle de développement

1. Ecrire un cas de test
2. Vérifier qu'il échoue (car le code qu'il teste n'existe pas encore)
3. Ecrire juste le code suffisant pour passer le test
4. Vérifier que le test passe
5. Puis refactoriser le code : l'améliorer en vérifiant qu'il n'y a pas de régressions



11.3. Outils pour tester

- Un **test unitaire** sert à tester des fonctions isolées (vérifier qu'une fonction, méthode, classe « fait bien son travail »).
- Un **test fonctionnel** sert à tester un « scénario » d'utilisation du logiciel
- Pour chaque type de test, il existe des outils permettant de faciliter l'écriture des tests et l'exploitation de leurs résultats
- Il existe aussi des outils permettant, entre autres, de faire tourner automatiquement et régulièrement des tests afin de vérifier qu'une application en cours de développement ne régresse pas (serveur d'intégration continue, exemple : **Jenkins**, **GitLab CD/CI**, ...)
- Dans le domaine des tests unitaires, un *framework* Java, **Junit**, est devenu très populaire :
 - Il a été porté dans d'autres langages (architecture **xUnit**) : **CppUnit**, **PHPUnit**, ...
 - Il met en œuvre plusieurs Design Patterns (composite, visitor, decorator)
 - Il a fortement inspiré un framework récent, bien intégré à CLion : **Google Test**

<http://google.github.io/googletest/>

11.4. Gtest – Principes

- Vous écrivez des **assertions** qui sont des déclarations qui vérifient si une condition est vraie. Le résultat d'une assertion peut être : succès, échec non fatal ou échec fatal.
- En cas d'échec fatal, la fonction s'arrête, sinon le programme continue normalement
- Un **test** (on parle aussi de **cas de test**) comporte une ou plusieurs assertions afin de vérifier le comportement d'un morceau de code (par exemple une méthode).
Un cas de test a échoué s'il a "planté" ou si une de ses assertions a échoué. Sinon, le cas de test est un succès.
- Une **suite de tests** regroupe plusieurs **tests** pour refléter la structure du code testé. Quand plusieurs **tests** d'une même **suite de tests** partagent des objets et des méthodes, on peut les regrouper dans une classe de test appelée **fixture**
- Un **Programme de Test** peut comporter plusieurs **suites de tests**
- Idéalement, les tests unitaires d'un composant (classe, méthode) doivent **couvrir 100% du code** (notion de **code coverage**)

11.5. Démarche – bonnes pratiques

- Les tests doivent être indépendants et doivent pouvoir être répétés aussi souvent que nécessaires
 - Chaque test doit tourner de façon isolée sur un objet (utilisation d'objets « mock »)
 - Quand un test échoue, on doit pouvoir ne relancer que lui pour déboguer rapidement
- Les tests doivent être bien organisés pour refléter la structure du code testé
 - Par exemple : une suite de tests par classe, un cas de test par méthode
 - Les cas de test d'une même suite peuvent partager des données et des procédures (fixtures)
- Les tests doivent être portables et réutilisables
 - Ils doivent fonctionner sous différents OS, avec différents compilateurs
- Quand des tests échouent :
 - Ils ne doivent pas interrompre les tests suivants
 - Ils doivent fournir le plus d'information possible pour aider au débogage
- Le framework de test doit faciliter la vie du testeur
- Les tests doivent être rapides pour pouvoir être rejoués souvent (pendant les cycles TDD)

11.6. Assertions (1)

- Les assertions sont des macros
 - Lorsque vous écrivez une assertion, son code va être remplacé, par le pré-processeur, par un code plus complexe (vous gagnez du temps !)
- La plupart des assertions de Google Test sont disponibles sous deux formes, **EXPECT_...** et **ASSERT_...**, par exemple :
 - **EXPECT_TRUE**(une_condition) : teste si la condition est vraie et poursuit le cas de test courant même si elle ne l'est pas (non fatal failure)
 - **ASSERT_TRUE**(une_condition) : teste si la condition est vraie et interrompt le cas de test courant si elle ne l'est pas (fatal failure)
- Toutes les assertions peuvent (**doivent** !) être suivies d'un opérateur **<<** pour produire un message d'information en cas d'échec
 - `EXPECT_TRUE(uneCondition) << "uneCondition n'est pas vraie mais devrait l'être !"`
 - Ce message doit aider à comprendre ce qui a échoué
 - **C'est une forme de documentation des tests et donc du code testé**
- **Google Test** propose beaucoup plus de formes d'assertion que le framework original CppUnit dont il s'inspire :
<http://google.github.io/googletest/reference/assertions.html>

11.6. Assertions (2)

- Assertions Booléennes
 - EXPECT_TRUE(condition) / ASSERT_TRUE(condition)
vérifie que condition est vraie
 - EXPECT_FALSE(condition) / ASSERT_FALSE(condition)
vérifie que condition est fausse
- Assertions Comparaisons Binaires
 - EXPECT_EQ(val1, val2) / ASSERT_EQ(val1, val2)
vérifie que val1 == val2
 - EXPECT_NE(val1, val2) / ASSERT_NE(val1, val2)
vérifie que val1 != val2
 - EXPECT_LT(val1, val2) / ASSERT_LT(val1, val2)
vérifie que val1 < val2
 - EXPECT_LE(val1, val2) / ASSERT_LE(val1, val2)
vérifie que val1 <= val2
 - EXPECT_GT(val1, val2) / ASSERT_GT(val1, val2)
vérifie que val1 > val2
 - EXPECT_GE(val1, val2) / ASSERT_GE(val1, val2)
vérifie que val1 >= val2

11.6. Assertions (3)

- Assertions Chaînes de Caractères du C (char *)
 - EXPECT_STREQ(str1, str2) / ASSERT_STREQ(str1, str2)
vérifie que str1 et str2, chaînes du langage C, ont le même contenu
 - EXPECT_STRNE(str1, str2) / ASSERT_STRNE(str1, str2)
vérifie que str1 et str2, chaînes du langage C, n'ont pas le même contenu
 - EXPECT_STRCASEEQ(str1, str2) / ASSERT_STRCASEEQ(str1, str2)
vérifie que str1 et str2, chaînes du langage C, ont le même contenu indépendamment de la casse
 - EXPECT_STRNE(str1, str2) / ASSERT_STRNE(str1, str2)
vérifie que str1 et str2, chaînes du langage C, n'ont pas le même contenu, indépendamment de la casse
- Assertions Comparaison Réels (float, double)
 - EXPECT_FLOAT_EQ(val1, val2) / ASSERT_FLOAT_EQ(val1, val2)
vérifie que val1 et val2, de type float, ont la même valeur à 4 ULP près
ULP : Unit of Least Precision : différence entre 2 flottants différents mais aussi proches l'un de l'autre que possible
 - EXPECT_DOUBLE_EQ(val1, val2) / ASSERT_DOUBLE_EQ(val1, val2)
vérifie que val1 et val2, de type double, ont la même valeur à 4 ULP près
 - EXPECT_NEAR(val1, val2, err) / ASSERT_NEAR(val1, val2, err)
vérifie que la différence entre val1 et val2 n'excède pas err

11.6. Assertions (4)

- Assertions Exceptions
 - EXPECT_THROW(inst, type_except)
ASSERT_THROW(inst, type_except)
vérifie que l'instruction `inst` lève une exception de type `type_except`
 - EXPECT_ANY_THROW(inst)
ASSERT_ANY_THROW(inst)
vérifie que l'instruction `inst` lève une exception (n'importe laquelle !)
 - EXPECT_NO_THROW(inst)
ASSERT_NO_THROW(inst)
vérifie que l'instruction `inst` ne lève aucune exception

11.7. Gtest – exemple (1)

On veut tester la classe **MontantDevise** qui sert à représenter une somme d'argent dans une certaine devise, et à comparer de tels objets :

```
class MontantDevise {
public:
    MontantDevise(double montant, std::string devise)
        : m_montant(montant), m_devise(devise) {}
    double          getMontant() const { return m_montant; }
    const std::string & getDevise() const { return m_devise; }
    bool operator == (const MontantDevise & somme) const {
        // Objets égaux si montants égaux et devises égales
        return this->m_montant == somme.m_montant &&
            this->m_devise == somme.m_devise;
    }
    bool operator != (const MontantDevise & somme) const {
        // Objets différents si pas égaux
        return (*this == somme);
    }
    virtual ~MontantDevise() {}
private:
    double          m_montant;
    std::string m_devise;
};
```

11.7. Gtest – exemple (2)

- Pour tester cette classe, on écrit un **programme de test** avec une seule **suite de tests** `MontantDeviseTest`, composée de 3 **tests** :
 - **Constructeur** : 2 **assertions** pour vérifier que le constructeur fonctionne bien
 - **Egalite** : 4 **assertions** pour vérifier que l'opérateur `==` fonctionne bien
 - **Difference** : 4 **assertions** pour vérifier que l'opérateur `!=` fonctionne bien
- Chaque **test** est implémenté dans une fonction qui a la syntaxe :
`TEST(nom_suite_test, nom_test)`
- **TEST** est une macro du préprocesseur qui va « fabriquer » un code plus complexe à partir de votre code (et vous épargner beaucoup de travail)
- Chaque **TEST** va comporter des assertions qui sont elles aussi des macros qui vous facilitent le travail de rédaction des tests. 2 formes d'assertions :
 - `EXPECT_...(...)` : vérifie une assertion et poursuit le test même en cas d'échec
 - `ASSERT_...(...)` : vérifie une assertion et arrête le test en cas d'échec
- Chaque assertion **doit** s'agrémenter d'un message qui explique ce qu'elle vérifie (`<< "message"` à la suite d'une assertion)

11.7. Gtest – exemple (3)

```
#include "gtest/gtest.h"
#include "MontantDevise.h"
// Définition de la suite de test "MontantDeviseTest"
// La suite est composée de 3 tests : Constructeur, Egalite, Différence
TEST(MontantDeviseTest, Constructeur) {
    // Déclaration des données nécessaires au test
    MontantDevise somme100EUR(100.0, "EUR");
    // Assertions pour vérifier que le constructeur a bien fait son travail
    EXPECT_FLOAT_EQ(somme100EUR.getMontant(), 100.0) << "Montant bien construit";
    EXPECT_EQ(somme100EUR.getDevise(), "EUR") << "Devise bien construite";
}
TEST(MontantDeviseTest, Egalite) {
    // Déclaration des données nécessaires au test
    MontantDevise somme100EUR(100, "EUR");
    MontantDevise somme100USD(100, "USD");
    MontantDevise somme500EUR(500, "EUR");
    MontantDevise somme500USD(500, "USD");
    // Assertions pour vérifier que l'opérateur == fonctionne
    EXPECT_TRUE(somme100EUR == somme100EUR) << "== renvoie vrai si meme montant et meme devise";
    EXPECT_FALSE(somme100EUR == somme100USD) << "== renvoie faux si meme montant mais pas meme devise";
    EXPECT_FALSE(somme100EUR == somme500EUR) << "== renvoie faux si pas meme montant mais meme devise";
    EXPECT_FALSE(somme100EUR == somme500USD) << "== renvoie faux si pas meme montant et pas meme devise";
}
TEST(MontantDeviseTest, Difference) {
    // Déclaration des données nécessaires au test
    MontantDevise somme100EUR(100, "EUR");
    MontantDevise somme100USD(100, "USD");
    MontantDevise somme500EUR(500, "EUR");
    MontantDevise somme500USD(500, "EUR");
    // Assertions pour vérifier que l'opérateur != fonctionne
    EXPECT_FALSE(somme100EUR != somme100EUR) << "== renvoie faux si meme montant et meme devise";
    EXPECT_TRUE(somme100EUR != somme100USD) << "== renvoie vrai si meme montant mais pas meme devise";
    EXPECT_TRUE(somme100EUR != somme500EUR) << "== renvoie vrai si pas meme montant mais meme devise";
    EXPECT_TRUE(somme100EUR != somme500USD) << "== renvoie vrai si pas meme montant et pas meme devise";
}
```

Gtest propose toute une collection d'assertions : voir documentation
<http://google.github.io/googletest/reference/assertions.html>

11.7. Gtest – exemple (4)

- Ces 3 tests manipulent des données qui peuvent être regroupées au sein d'une classe **fixture** :
 - qu'on appellera du nom de la suite de tests, **MontantDeviseTest**
 - qui devra hériter de la classe **testing::Test**
- On peut alors ré-écrire le programme de test en écrivant les tests à l'aide de la macro **TEST_F** :
TEST_F(nom_classe_fixture, nom_test)
- Un objet de la classe **fixture** sera automatiquement instancié et transmis au test qui pourra accéder aux membres de la classe, en général tous déclarés **protected**.

11.7. Gtest – exemple (5)

```
class MontantDeviseTest : public testing::Test { // Classe « Fixture » pour le test de MontantDevise
protected:
    MontantDeviseTest()
        : somme100EUR(100.0, "EUR"), somme100USD(100.0, "USD"),
          somme500EUR(500.0, "EUR"), somme500USD(500.0, "USD") {}
    void SetUp() override {} // Exécuté au début pour initialiser les données du test
    void TearDown() override {} // Exécuté à la fin pour nettoyer (delete) les données du test
    // Déclaration d'attributs qui seront les données utilisées pendant les tests
    MontantDevise somme100EUR; MontantDevise somme100USD;
    MontantDevise somme500EUR; MontantDevise somme500USD;
};

TEST_F(MontantDeviseTest, Constructeur) {
    // Assertions pour vérifier que le constructeur a bien fait son travail
    EXPECT_FLOAT_EQ(somme100EUR.getMontant(), 100.0) << "Montant bien construit";
    EXPECT_EQ(somme100EUR.getDevise(), "EUR") << "Devise bien construite";
}

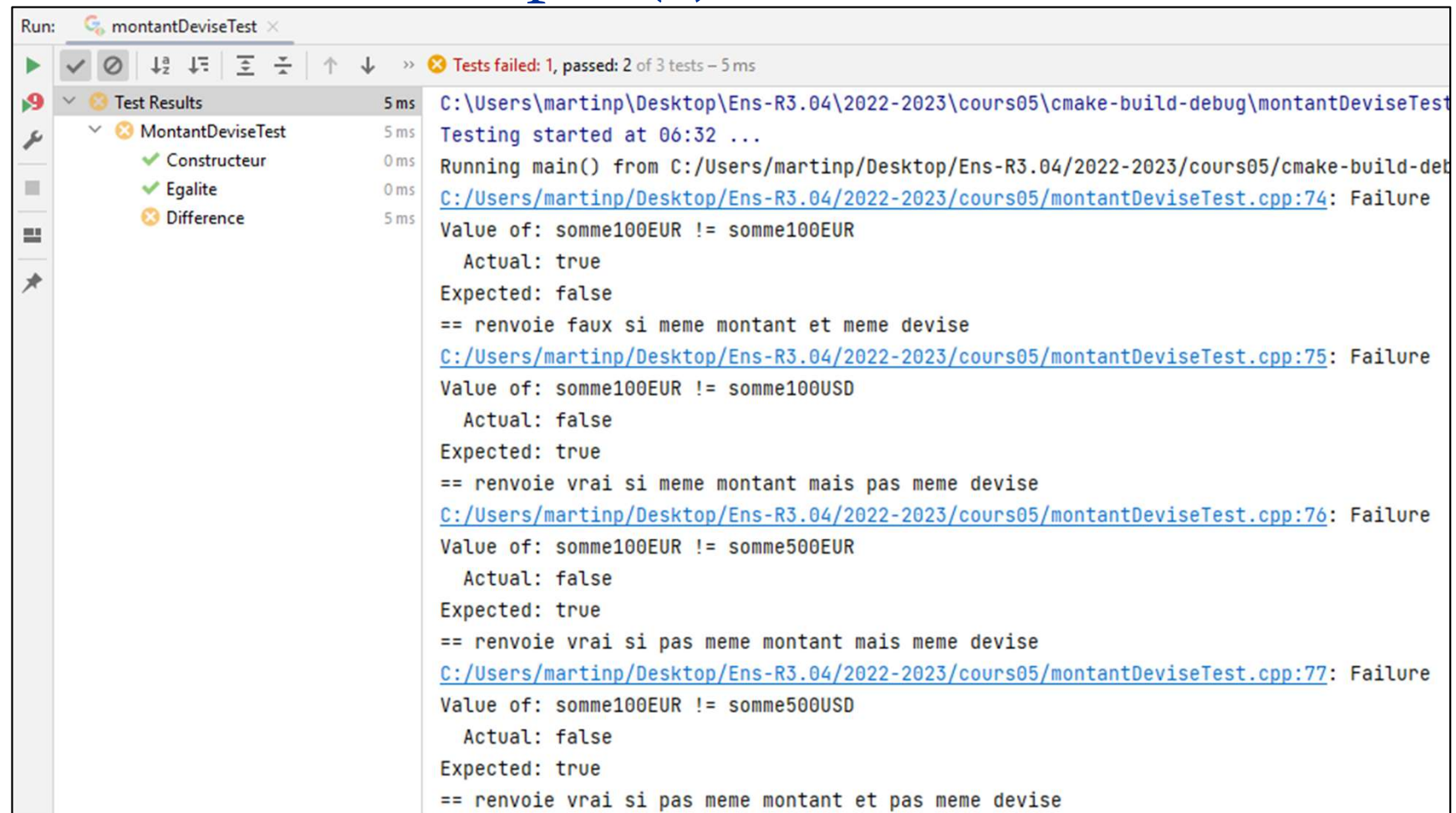
TEST_F(MontantDeviseTest, Egalite) {
    // Assertions pour vérifier que l'opérateur == fonctionne
    EXPECT_TRUE(somme100EUR == somme100EUR) << "== renvoie vrai si meme montant et meme devise";
    EXPECT_FALSE(somme100EUR == somme100USD) << "== renvoie faux si meme montant mais pas meme devise";
    EXPECT_FALSE(somme100EUR == somme500EUR) << "== renvoie faux si pas meme montant mais meme devise";
    EXPECT_FALSE(somme100EUR == somme500USD) << "== renvoie faux si pas meme montant et pas meme devise";
}

TEST_F(MontantDeviseTest, Difference) {
    // Assertions pour vérifier que l'opérateur != fonctionne
    EXPECT_FALSE(somme100EUR != somme100EUR) << "== renvoie faux si meme montant et meme devise";
    EXPECT_TRUE(somme100EUR != somme100USD) << "== renvoie vrai si meme montant mais pas meme devise";
    EXPECT_TRUE(somme100EUR != somme500EUR) << "== renvoie vrai si pas meme montant mais meme devise";
    EXPECT_TRUE(somme100EUR != somme500USD) << "== renvoie vrai si pas meme montant et pas meme devise";
}
```

11.7. Gtest – exemple (6)

- Les fichiers contenant les tests, ainsi que les classes testées, doivent être compilés et liés avec la bibliothèque Google Test : `gtest_main`
- Cette bibliothèque contient (entre autres) un programme principal qui va enchaîner les suites de tests et comptabiliser les résultats.
- Ces résultats peuvent être visualisés et exploités de différentes façons.
- Clion offre une interface spécifique, qui permet d'exécuter tous les tests ou de n'en exécuter qu'un seul à la fois, et de visualiser de façon hiérarchique les résultats obtenus.

11.7. Gtest – exemple (7)



```
Run: montantDeviseTest x
Tests failed: 1, passed: 2 of 3 tests – 5 ms

Test Results 5 ms
  MontantDeviseTest 5 ms
    Constructeur 0 ms
    Egalite 0 ms
    Difference 5 ms

C:\Users\martinp\Desktop\Ens-R3.04\2022-2023\cours05\cmake-build-debug\montantDeviseTest
Testing started at 06:32 ...
Running main() from C:/Users/martinp/Desktop/Ens-R3.04/2022-2023/cours05/cmake-build-det
C:/Users/martinp/Desktop/Ens-R3.04/2022-2023/cours05/montantDeviseTest.cpp:74: Failure
Value of: somme100EUR != somme100EUR
Actual: true
Expected: false
== renvoie faux si meme montant et meme devise
C:/Users/martinp/Desktop/Ens-R3.04/2022-2023/cours05/montantDeviseTest.cpp:75: Failure
Value of: somme100EUR != somme100USD
Actual: false
Expected: true
== renvoie vrai si meme montant mais pas meme devise
C:/Users/martinp/Desktop/Ens-R3.04/2022-2023/cours05/montantDeviseTest.cpp:76: Failure
Value of: somme100EUR != somme500EUR
Actual: false
Expected: true
== renvoie vrai si pas meme montant mais meme devise
C:/Users/martinp/Desktop/Ens-R3.04/2022-2023/cours05/montantDeviseTest.cpp:77: Failure
Value of: somme100EUR != somme500USD
Actual: false
Expected: true
== renvoie vrai si pas meme montant et pas meme devise
```

- **Gtest** et **CLion** montrent clairement le nombre de tests réussis / échoués et où se situent les échecs.
- Oops... il y a une erreur dans l'opérateur **!=** de la classe **MontantDevise**
- Il est facile de trouver l'erreur : un « not » (!) manquait dans l'opérateur !=

11.8. Gtest – conclusion

- Les tests unitaires permettent de vérifier rapidement, de manière automatisée, le bon fonctionnement du code tout au long de sa mise au point, de son évolution et de sa maintenance
- CLion (et autres IDE) fournissent des outils pour intégrer (plus ou moins facilement) des frameworks de tests unitaires
- Les assertions, commentées « avec message », fournissent une forme de spécification et donc une documentation pour le code à développer
- Le travail pour mettre en place les tests est long :
 - Il augmente le coût initial de développement (court terme)
 - Mais il est rentable sur l'ensemble du cycle de vie (long terme)

11.9. Autre Exemple Complet (1)

- Tester unitairement le template ObjetContraint :

```
template <class T>
class ObjetContraint {
public:
    ObjetContraint(T valeur, T min, T max);
    // Construit un nombre contraint en vérifiant que min <= val <= max
    // sinon lève une exception domain_error
    void setVal(T val);
    // Modifie la valeur de l'entrée contraint si m_min <= val <= m_max
    // sinon lève une exception domain_error
    void saisir(std::istream & entree = std::cin);
    // Lit une valeur sur le flux entree et l'affecte à m_val si m_min <= valeur <= m_max
    // sinon lève une exception domain_error
    void afficher(std::ostream & sortie = std::cout) const;
    // Ecrit m_val sur le flux sortie
    T getMin() const; // résultat = m_min
    T getMax() const; // résultat = m_max
    T getVal() const; // résultat = m_val
    operator T() const; // résultat = m_val

    virtual ~ObjetContraint() {}
private:
    T m_min;
    T m_max;
    T m_val;
};
```

11.9. Autre Exemple (2) – Test Constructeur

```
#include "gtest/gtest.h"
#include "ObjetContraint.h"
#include <stdexcept>
using namespace std;
TEST(ObjetContraintTest, Constructeur) {
    const int MIN = 0; const int MAX = 100; const int VAL = 10;
    // On vérifie qu'une exception est levée si la valeur n'est pas comprise entre min et max
    EXPECT_THROW(ObjetContraint<int>(MIN - 1, MIN, MAX), domain_error)
        << "Construction avec val < min < max : exception";
    EXPECT_THROW(ObjetContraint<int>(MAX + 1, MIN, MAX), domain_error)
        << "Construction avec min < max < val : exception";
    EXPECT_THROW(ObjetContraint<int>(MAX, MAX, MIN), domain_error)
        << "Construction avec val = max < min : exception";
    // On vérifie qu'aucune exception n'est levée si la valeur est comprise entre min et max
    EXPECT_NO_THROW(ObjetContraint<int>(VAL, MIN, MAX))
        << "Construction avec min < val < max : pas d'exception";
    EXPECT_NO_THROW(ObjetContraint<int>(MIN, MIN, MAX))
        << "Construction avec min = val < max : pas d'exception";
    EXPECT_NO_THROW(ObjetContraint<int>(MAX, MIN, MAX))
        << "Construction avec min < val = max : pas d'exception";
    EXPECT_NO_THROW(ObjetContraint<int>(MIN, MIN, MIN))
        << "Construction avec min = val = max : pas d'exception";
    // On vérifie que l'objet est bien construit
    // Les assertions suivantes font l'hypothèse que les getters sont justes
    ObjetContraint<int> n(VAL, MIN, MAX);
    EXPECT_EQ(n.getVal(), VAL) << "Après Construction : la valeur est juste";
    EXPECT_EQ(n.getMin(), MIN) << "Après Construction : le minimum est juste";
    EXPECT_EQ(n.getMax(), MAX) << "Après Construction : le maximum est juste";
}
```

11.8. Autre Exemple (3) – Test SetVal

- Test « boîte noire » : on ne sait pas que setVal est utilisé dans le constructeur. Il faut donc le (re)tester complètement.

```
TEST(ObjetContraintTest, SetVal) {
    const int MIN = 0;
    const int MAX = 100;
    const int VAL = 10;
    ObjetContraint<int> n(VAL, MIN, MAX);
    // On vérifie qu'une exception est levée si val n'est pas compris entre min et max
    EXPECT_THROW(n.setVal(MIN - 1), domain_error)
                << "Change valeur avec val < min : exception";
    EXPECT_THROW(n.setVal(MAX + 1), domain_error)
                << "Change valeur avec val > max : exception";
    // On vérifie qu'aucune exception n'est levée
    // et que la valeur est bien changée si val entre min et max
    ASSERT_NO_THROW(n.setVal(MAX)) << "Change valeur avec val = max : pas d'exception";
    EXPECT_EQ(n.getVal(), MAX) << "Après setVal, La valeur vaut MAX";
    ASSERT_NO_THROW(n.setVal(MIN)) << "Change valeur avec val = min : pas d'exception";
    EXPECT_EQ(n.getVal(), MIN) << "Après setVal, La valeur vaut MIN";
    ASSERT_NO_THROW(n.setVal((MIN + MAX) / 2))
                << "Change valeur avec min <= val <= max : pas d'exception";
    EXPECT_EQ(n.getVal(), (MIN + MAX) / 2) << "Après setVal, la valeur vaut (MIN+MAX)/2";
}
```

11.9. Autre Exemple (4) – Test Afficher

- Pour tester des méthodes d'Entrées/Sorties, il faut utiliser des fichiers (qui doivent être supprimés à la fin du test)

```
#include <fstream>
using namespace std;

TEST(ObjetContraintTest, Afficher) {
    const int MIN = 0;
    const int MAX = 100;
    const int VAL = 10;
    const char *nomFichier = "TestAfficher.txt";
    // On crée un fichier texte vide en lecture/écriture
    fstream fSortie(nomFichier, ios_base::in | ios_base::out | ios_base::trunc);
    // On écrit un entier contraint dans fsortie
    ObjetContraint<int> n(VAL, MIN, MAX);
    n.afficher(fSortie);
    // On lit le contenu du fichier et on vérifie qu'il ne contient que la valeur
    fSortie.seekg(ios_base::beg); // On se place au début du fichier
    int valeur;
    fSortie >> valeur;
    EXPECT_EQ(VAL, valeur) << "La valeur écrite est juste";
    EXPECT_TRUE(fSortie.peek() == EOF) << "On n'a écrit que la valeur";
    // on fait le ménage : fermeture & suppression du fichier créé
    fSortie.close();
    remove(nomFichier);
}
```


11.9. Autre Exemple (5) – Test Saisir

- Idem pour la méthode saisir :

```
TEST(ObjetContraintTest, Saisir) {
    const int MIN = 0;
    const int MAX = 100;
    const int VAL = 10;
    const char *nomFichier = "TestSaisir.txt";
    // on crée un fichier texte vide en lecture/ecriture
    fstream fEntree(nomFichier, ios_base::in | ios_base::out | ios_base::trunc);
    // on écrit dans fEntree la valeur VAL, la valeur MIN-1 et la valeur MAX+1
    fEntree << VAL << endl << MIN - 1 << endl << MAX + 1 << endl;
    // on construit un nombre contraint entre 0 et 100
    ObjetContraint<int> n(MIN, MIN, MAX);
    // on lit le contenu de fEntree et on vérifie qu'il ne contient que la valeur
    fEntree.seekg(ios_base::beg); // on se place au début du fichier
    EXPECT_NO_THROW(n.saisir(fEntree))
        << "Lecture d'une valeur avec min <= val <= max : pas d'exception";
    EXPECT_EQ(n.getVal(), VAL) << "La valeur lue est bien la bonne";
    EXPECT_THROW(n.saisir(fEntree), domain_error)
        << "Lecture d'une valeur < min : exception";
    EXPECT_THROW(n.saisir(fEntree), domain_error)
        << "Lecture d'une valeur > max : exception";
    EXPECT_TRUE(fEntree.peek() == EOF) << "Toutes les valeurs ont bien été lues";
    // on fait le ménage : fermeture & suppression du fichier créé
    fEntree.close();
    remove(nomFichier);
}
```