

Test Driven Development GoogleTest / GoogleMock

Par Nicolas BENOIT

Août 2016

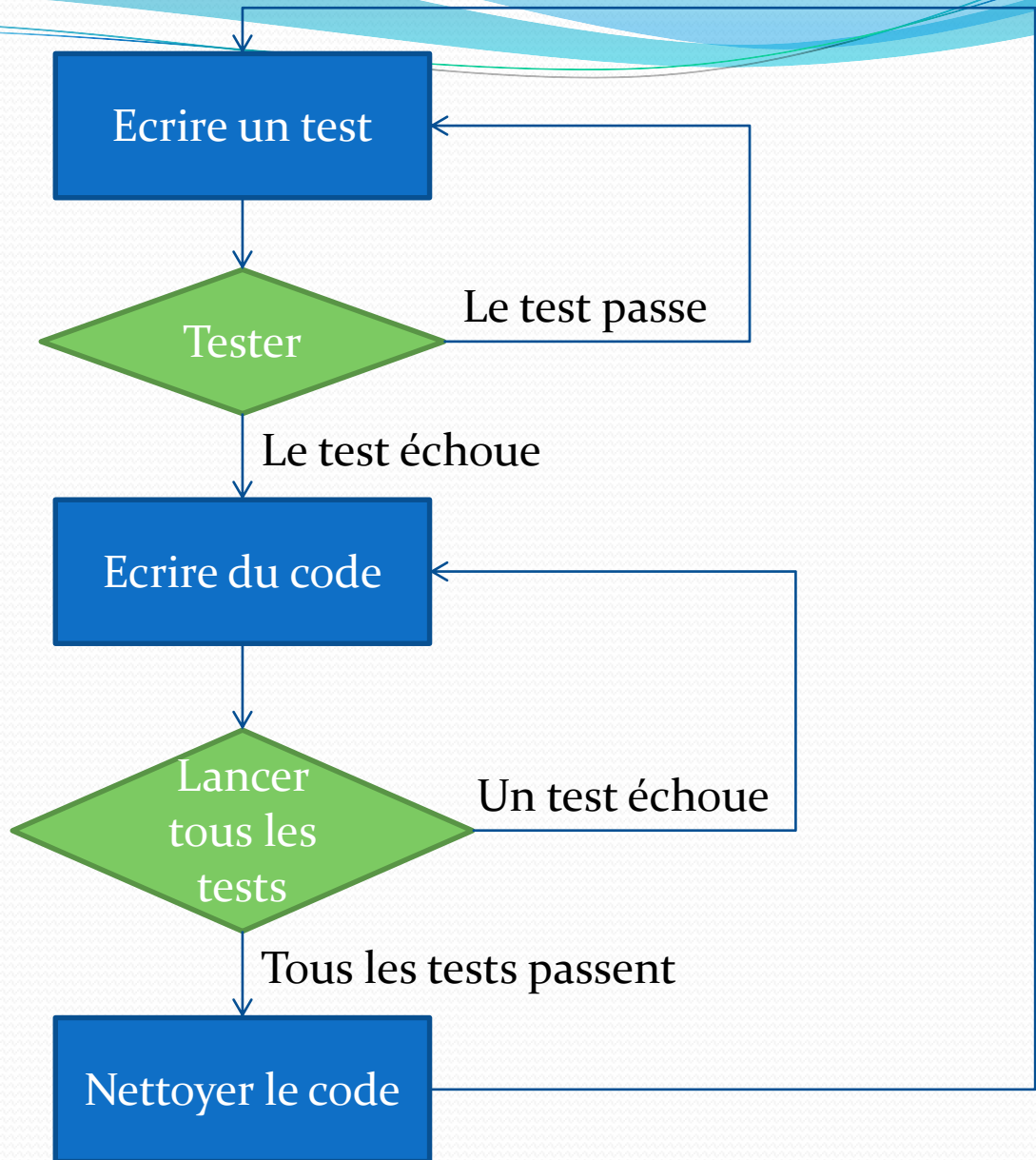
Test Driven Development

Test Driven Development

- Méthode de développement basée sur les tests unitaires
- Tests unitaires
 - A chaque fonction est associée au moins un test
 - A chaque mise à jour du programme tous les tests sont lancés
 - Préviend toute régression

Test Driven Development

- On veut ajouter une nouvelle fonctionnalité à l'application
- Schéma classique de développement :
 - On code une fonction, une classe, une méthode
 - On la teste
 - On l'implémente dans l'application
- Développement Piloté par Test
 - On implémente le test de la fonctionnalité
 - On développe la fonction/classe qui remplit le test



Ecrire des tests

- Exemple : on veut écrire une méthode « addition » dans une classe MathPerso
- On va écrire avant une classe TestMathPerso chargée de tester le bon fonctionnement de la méthode
- Penser aux cas spéciaux, aux différents types de variables...
- Si un bug est découvert par la suite : ajouter un test

On commence par écrire les tests

```
class TestMathPerso
{
public:
    //On a une méthode centrale chargée
    //De lancer tous les tests
    //Elle renvoie true si tous les tests passent
    static bool lancerTests()
    {
        return testAddition1() && testAddition2() && testAddition3();
    }
    //Chaque test renvoie un booléen
    //True si passé, false sinon
    static bool testAddition1()
    {
        int a = 42, b = 18;
        if(MathPerso::additionner(a,b) == 60)
        {
            cout << "1";
            return true;
        }
        else
        {
            cout << "0";
            return false;
        }
    }

    static bool testAddition2()
    {
        double a = 42.6, b = 18.3, c = MathPerso::additionner(a,b);

        int cInt = (int)(c*10);

        if(cInt == 609)
        {
            cout << "1";
            return true;
        }
        else
        {
            cout << "0";
            return false;
        }
    }

    static bool testAddition3()
    {
        int a = -7, b = -3;
        if(MathPerso::additionner(a,b) == -10)
```

Puis on écrit la méthode et on lance les tests

```
class MathPerso
{
    //On veut créer une classe de fonctions mathématiques

    public:

    //La méthode addition
    template<typename T>
    static T additionner(T a, T b)
    {
        return a+b;
    }

};
```

```
int main()
{
    cout << TestMathPerso::lancerTests();

    return 0;
}
```



En réel: Frameworks de Test

- La pratique n'est pas si loin de ce cas « bricolé »
- Mais un framework apporte son lot de fonctions pensées pour les tests qui facilitent la vie

GoogleTest

Google Test

- Google Test permet de tester le retour de fonctions
- On définit des blocs de tests dans lesquels on définit un ou plusieurs tests
- On lance enfin tous les tests dans le main et GTest nous fait un retour dans la console

Blocs de Tests

- La fonction TEST permet de définir un test unitaire
- `TEST (groupeTest, testUnitaire){//liste d'assertions}`
- `testUnitaire` est le nom de votre test unitaire
 - Contient plusieurs assertions
 - En général un `testUnitaire` teste une fonction de plusieurs manières différentes
- `groupeTest` permet d'organiser les résultats en regroupant les tests unitaires par lot
- Il ne peut y avoir deux TEST avec les deux mêmes noms

Définir des tests

- Dans un test unitaire, devant chaque assertion
 - EXPECT_... : en cas d'échec de l'assertion, l'exécution du bloc continue
 - ASSERT_... : en cas d'échec de l'assertion l'exécution du s'arrête
- EXPECT/ASSERT_EQ(a,b) teste l'égalité entre a et b
- EXPECT/ASSERT_STREQ(str1,str2) entre deux strings
 - Fonctionne avec des char*, utiliser string.c_str()

Le cas des nombres décimaux

- A cause de la manière dont ils sont stockés dans la mémoire, les nombre décimaux sont rarement des valeurs exactes
- Au lieu de `EXPECT_EQ` :
 - `EXPECT_FLOAT_EQ` et `EXPECT_DOUBLE_EQ`
 - De même pour les `ASSERT`
- Pour encore plus de précision :
 - `EXPECT_NEAR` (`expected`, `actual`, `absolute_range`)
 - Ex : `EXPECT_NEAR(25.4, sqrt(645.16), 0.0000001)`

Définition Tests Exemple

```
#include "gtest/gtest.h"

TEST (SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}

TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EQ (-1, square-root (-22.0));
}
```

Lancer les tests

- Les tests se lancent dans le `main()`
- La fonction `::testing::InitGoogleTest()` initialise le framework
- La fonction `RUN_ALL_TESTS()` lance les tests
 - Cette fonction ne doit être appelée qu'une fois sous peine de créer des conflits

Lancement Tests Exemple

```
#include "gtest/gtest.h"

TEST(SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}

TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EQ (-1, square-root (-22.0));
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Exemple Résultats

- Les résultats des tests s'affichent dans la console

```
Running main() from user_main.cpp
[=====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 2 tests from SquareRootTest
[ RUN      ] SquareRootTest.PositiveNos
..\user_sqrt.cpp(6862): error: Value of: sqrt (2533.310224)
Actual: 50.332
Expected: 50.3321
[  FAILED  ] SquareRootTest.PositiveNos (9 ms)
[ RUN      ] SquareRootTest.ZeroAndNegativeNos
[          OK ] SquareRootTest.ZeroAndNegativeNos (0 ms)
[-----] 2 tests from SquareRootTest (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test case ran. (10 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SquareRootTest.PositiveNos

1 FAILED TEST
```

En utilisant une classe

- Il est possible de déclarer des variables dans un test unitaire

```
#include "Animal.h"
#include <gtest/gtest.h>

using namespace std;

TEST(testAnimal, Instanciation) {

    Animal max("Max", "okapi", 7);

    EXPECT_STREQ("Max", max.getNom().c_str());
    EXPECT_STREQ("okapi", max.getEspece().c_str());
    EXPECT_EQ(7, max.getAge());
}
```

- Mais si l'on veut qu'une variable serve pour plusieurs tests, il faut définir une classe Fixture

Les classes Fixture

- Permettent :
 - D'initialiser avant les tests
 - De stocker des données pendant plusieurs tests
- S'écrit comme une classe classique
 - Avec deux méthodes en plus
- Le groupe de tests portera obligatoirement le nom de la classe Fixture

Définition d'une classe Fixture

```
//Hérite de ::testing::Test
class AnimalFixture : public ::testing::Test {

protected :
    //Placer les attributs comme d'habitude
    Animal* m_animal;

public:
    //Constructeur de la classe fixture
    //Permet l'initialisation
    AnimalFixture( ) {
        m_animal = new Animal("Max", "okapi", 7);
    }

    //Ce code sera exécuté juste avant les tests
    void SetUp( ) {
    }

    //Ce code sera exécuté juste après les tests
    void TearDown( ) {
    }

    //Destructeur de la classe fixture
    ~AnimalFixture( ) {
        delete(m_animal);
    }

};
```

Tests utilisant la classe Fixture

TestsAnimal.cpp

```
#include <iostream>
#include "AnimalFixture.cpp"
#include <gtest/gtest.h>

using namespace std;

//Le bloc de tests porte le nom de la classe Fixture
TEST_F(AnimalFixture, Instanciation) {

    EXPECT_STREQ("Max", m_animal->getNom().c_str());
    EXPECT_STREQ("okapi", m_animal->getEspece().c_str());
    EXPECT_EQ(7, m_animal->getAge());
}

TEST_F(AnimalFixture, Execution) {

    string s = "";
    s+="Bonjour ! Je suis Max\n";
    s+="Je suis un okapi\n";
    s+="Et j'ai 7 ans.";

    EXPECT_STREQ(s.c_str(), m_animal->saluer().c_str());
}
```

main.cpp

```
int main(int argc, char** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();

    return 0;
}
```

```
===== Running 2 tests from 1 test case.
----- Global test environment set-up.
----- 2 tests from AnimalFixture
[ RUN      ] AnimalFixture.Instanciation
[ OK       ] AnimalFixture.Instanciation (0 ms)
[ RUN      ] AnimalFixture.Execution
[ OK       ] AnimalFixture.Execution (0 ms)
----- 2 tests from AnimalFixture (0 ms total)

----- Global test environment tear-down
===== 2 tests from 1 test case ran. (11 ms total)
[ PASSED  ] 2 tests.
```

Pour savoir tous les tests possibles...

- La doc Google Test !
- https://github.com/google/googletest/blob/master/googletest/docs/V1_5_Documentation.md

GoogleMock

Les limites des Tests Unitaires

- Les tests unitaires peuvent se révéler difficiles à mettre en place suivant les objets avec lesquels ils interagissent
- Typiquement avec des objets ayant :
 - Des facteurs aléatoires (utilisation de random)
 - Des facteurs dépendants du monde extérieur (capteurs, communication réseau)
 - Besoin de beaucoup de ressources (bases de données)

L'intérêt de Mocker

- Mocker un objet signifie le simuler
- Au lieu que le véritable objet fasse ses traitements et délivre un résultat
 - Le mock de l'objet va fournir des résultats prévus à l'avance
- Gain de ressource et plus grande facilité à écrire les tests

Exemple : une Alerte Tempête

- Nous voulons tester une classe `AlerteTempete` qui donne l'alerte si la météo est trop mauvaise
- Cette classe a comme attribut une interface vers une sonde météo
- L'interface récupère les données brutes de la sonde et les transmet en valeurs simples à `AlerteTempete`
- Nous voudrions mocker cette interface

ce que l'on veut MOCKER (simuler)

InterfaceSondeMeteo.h

```
class InterfaceSondeMeteo {  
  
    //Accès à une sonde météo  
  
public :  
  
    virtual ~InterfaceSondeMeteo(){}  
  
    virtual double getVitesseVent (std::string unite);  
    virtual double getPression (std::string unite);  
  
};  
  
double InterfaceSondeMeteo::getVitesseVent (std::string unite)  
{  
    double vitesseVent = 0;  
    //Appels à une vraie sonde  
    return vitesseVent;  
}  
  
double InterfaceSondeMeteo::getPression (std::string unite)  
{  
    double pressionAthmospherique = 0;  
    //Appels à une vraie sonde  
    return pressionAthmospherique;  
}
```

AlerteTempete.h

C O D E Q U E L O N V E U X T E S T

```
#include "InterfaceSondeMeteo.h"

class AlerteTempete {

    private :

        InterfaceSondeMeteo* m_sonde;

    public :

        AlerteTempete(InterfaceSondeMeteo* sonde):
            m_sonde(sonde)
        {}

        bool donnerAlerte()
        {
            double pression = m_sonde->getPression("hPa");
            double vent = m_sonde->getVitesseVent("kmh");

            if(pression<980 && vent>75)
                return true;
            else
                return false;
        }

};
```

Ecrire une Mock Class

- Avant toute chose, les méthodes que l'on voudra « mocker » doivent être virtuelles
 - Il suffit de leur ajouter le mot clef virtual
- Créer une classe MockNomClasse héritant de NomClasse dans .h
 - Pas besoin de .cpp

Ecrire une Mock Class

- Pour chaque méthode à « mocker » :
 - Son nom est `MOCK_METHODn`
 - ou `MOCK_CONST_METHODn`
 - Avec `n` le nombre d'arguments de la méthode
- Ces mock méthodes prennent comme arguments
 - Le 1^{er} : le nom de la méthode à « mocker »
 - En 2^e : `typeRetour(arguments)`
- Ex : `void fonction(int a, string b)`
 - -> `MOCK_METHOD2(fonction, void(int a, string b))`

MockInterfaceSondeMeteo.h

```
#include <iostream>
#include "gmock/gmock.h"
#include "InterfaceSondeMeteo.h"

class MockInterfaceSondeMeteo : public InterfaceSondeMeteo {

public :
    MOCK_METHOD1(getVitesseVent, double(std::string unite));
    MOCK_METHOD1(getPression, double(std::string unite));

};
```


Utiliser la classe

- Maintenant que nous avons défini notre classe, on retourne dans le schéma classique Google Test
- Nouveau test : `EXPECT_CALL(nomInstance, methode(args))`
 - `.Times(x)` //Vérifie qu'elle a été appelée au moins x fois
 - `.WillOnce(Return(y))` //Retournera y une fois
- Si l'on veut tester un nombre exact d'appels
 - `.Times(Exactly(x))`

ListeTests.cpp

```
#include "AlerteTempete.h"
#include "MockInterfaceSondeMeteo.h"
#include "gtest/gtest.h"
#include "gmock/gmock.h"
using ::testing::Return;

TEST(AlerteTempeteTest, GlobalTest) {

    MockInterfaceSondeMeteo mockSonde;
    AlerteTempete alerteur(&mockSonde);


    EXPECT_CALL(mockSonde, getPression("hPa"))
        .Times(1)
        .WillOnce(Return(1200));
    EXPECT_CALL(mockSonde, getVitesseVent("kmh"))
        .Times(1)
        .WillOnce(Return(100));

    EXPECT_FALSE(alerteur.donnerAlerte());

    EXPECT_CALL(mockSonde, getPression("hPa"))
        .Times(1)
        .WillOnce(Return(950));
    EXPECT_CALL(mockSonde, getVitesseVent("kmh"))
        .Times(1)
        .WillOnce(Return(180));

    EXPECT_TRUE(alerteur.donnerAlerte());
}
```

Autre exemple EXPECT_CALL

- On veut par exemple :
 - Tester que notre méthode sera appelée trois fois
 - Que ces trois fois elle retourne 1
 - Et qu'à chaque appel suivant elle retourne -1
- `EXPECT_CALL(nomInstance, méthode(args))`
 - `.Times(3)` 
 - `.WillRepeatedly(Return(1))`
- `EXPECT_CALL(nomInstance, méthode(args))`
 - `.WillRepeatedly(Return(-1))`