



AI03

Projet Tests logiciel Google Test

**Arman Soltani
Guillaume Nibert**

14/01/2020

1. Préambule

Suite à la réalisation de l'état de l'art sur les outils de tests et validation de logiciels nous avons choisi de nous concentrer sur le logiciel Google Test (GTest). Le travail réalisé en amont servira de base à notre projet. Nous nous appuyerons sur diverses sources de connaissances pour ce projet tel que le cours AI03 de l'UTC ainsi que de la documentation de GTest¹.

Le but de ce rapport est de servir d'introduction à GTest. Dans le chapitre ["Cas d'étude"](#) nous présenterons un projet plus réaliste développé avec GTest.

L'utilisation de GTest requiert un certain nombre de prérequis. Aussi nous considérons que notre lecteur est à l'aise avec les points suivants :

- Développement en C++ moderne (≥ 11) ;
- Utilisation de CMake² afin de compiler le C++ ;
- Utilisation d'un EDI moderne (ex: CLion³).

¹ Documentation de GTest :

<https://github.com/google/googletest/blob/master/googletest/docs/primer.md>

² Documentation de CMake : <https://cmake.org/>

³ Page officiel de CLion : <https://www.jetbrains.com/clion/>

2. Table des matières

Préambule	1
Table des matières	2
Introduction	3
Rappel - architecture xUnit	3
Justification du choix du logiciel	5
Présentation de Google Test	6
Description	6
Popularité	6
Recommandation/philosophie de Google concernant les tests	7
Récapitulatif	8
Installation de Google Test	9
Prise en main de Google Test	12
Exemple introductif	12
Types d'assertions	16
Tests fixtures	17
Génération d'un rapport au format XML	21
Cas d'étude	23
Initialisation de la classe Barrage	23
Initialisation des matrices	26
Simulation / calcul des valeurs de Bellman	27
Optimisation / recherche du meilleur chemin	29
Conclusion	31

1. Introduction

L'outil ayant retenu notre attention est Google Test (GTest). C'est un outil extrêmement léger, fonctionnant en ligne de commande et performant (possibilité de paralléliser les tests). Bien qu'il nécessite un niveau minimum de compétences en informatique il reste très intuitif et possède un grand nombre d'assertions prédéfinies notamment en implémentant l'architecture xUnit mais aussi en implémentant des fixtures.

Rappel - architecture xUnit

L'architecture xUnit⁴ définit un ensemble de composants de base que différents programmes de test peuvent ensuite implémenter dans le langage de leur choix. L'avantage d'utiliser un programme suivant l'architecture xUnit est qu'il sera très rapide à prendre en main puisque le vocabulaire utilisé entre les programmes implémentant xUnit est le même.

Typiquement un test est décrit suivant le format :

- **setup** : prépare le contexte du test en modifiant l'état interne du programme à tester.
- **instructions** : les instructions à effectuer dans le cadre du test (incluant les assertions sur les résultats et l'état interne du code).
- **teardown** : remet l'état interne du programme à tester dans un état neutre afin de ne pas perturber les tests qui suivent.

Test fixture

L'objectif des fixtures est de fixer un environnement pour l'exécution de tests. Cela se fait au moyen d'un morceau de code appelé fixture. L'idée est d'éviter de dupliquer du code inutilement en partageant des objets et des fonctions pour tous les tests (**Test Cases**) d'une **Test Suite**.

Le processus complet d'une fixture pour un test unitaire est le suivant :

1. **Initialisation** (set up) : définition d'un environnement de test complètement reproductible ;
2. **Exercice** : le module à tester est exécuté ;
3. **Vérification** : comparaison des résultats obtenus avec un vecteur de résultat défini. Le résultat du test est SUCCÈS ou ÉCHEC ;
4. **Désactivation** : désinstallation des fixtures pour laisser le système dans son état initial." - Source : Wikipédia⁵

Remarque : c'est basé sur l'architecture xUnit.

⁴ Description originale de l'architecture xUnit :

<https://web.archive.org/web/20150315073817/http://www.xprogramming.com/testfram.htm>

⁵ Définition des fixtures : https://fr.wikipedia.org/wiki/Test_fixture

La deuxième partie de ce projet a donc pour but d'approfondir et de prendre en main ce logiciel de tests. Nous avons fait le choix d'utiliser GTest dans l'environnement de développement CLion, car il est très bien intégré et que cet EDI est puissant et très populaire dans le monde entier, aussi bien par des communautés indépendantes que de grandes entreprises.

2. Justification du choix du logiciel

Lors de la réalisation de notre état de l'art nous avons classifié différents types de logiciels de tests présents sur le marché selon la nature de leur tests (fonctionnels / structurels) et leur type de licence (propriétaire / open source). Cela nous a permis de nous rendre compte de la variété des logiciels de tests. Ils ne réalisent pas tous les même tests, cela dépend essentiellement du besoin et du domaine d'application (un test dans le domaine de l'aviation ne sera pas nécessairement le même qu'un test dans le domaine de l'automobile).

À la suite de cet état de l'art, le logiciel Google Test a le plus retenu notre attention car il est open source, léger, multiplateforme, performant et utilisé par de grandes entreprises pour des projets d'envergures et permettra d'approfondir nos connaissances sur l'architecture xUnit, un type d'architecture générique que l'on peut appliquer sur beaucoup de langages de programmation (C, C++ avec Google Test, Java avec JUnit, Python avec PyUnit, PHP avec PHPUnit et bien d'autres...).

3. Présentation de Google Test

Description

Google Test⁶ est activement développé par l'équipe "Testing Technology"⁷ de Google et est utilisé pour tester les codes C++ internes de Google. C'est une librairie de test open source basée sur l'architecture xUnit utilisée pour tester les codes sources écrits en C++. La librairie se veut extrêmement portable et multi-plateforme (Linux, macOS et Microsoft Windows) et est intégrable dans la plupart des IDE modernes tel que JetBrains CLion et Microsoft Visual Studio.

Voici la liste exhaustive des fonctionnalités du logiciel : une infrastructure de tests xUnit, découverte de test, un grand jeu d'assertions dont certaines peuvent être définies par l'utilisateur, des "death tests", détection de potentielles défaillances fatales ou non fatales, des tests avec des valeurs ou types paramétrés, de nombreuses options d'exécution des tests ainsi que la génération d'un rapport de test au format XML.

GTest est prévu pour tester des programmes écrits en C++ mais par compatibilité il gère aussi le langage C.

Popularité

Le projet est open source sous license BSD 3-Clause⁸ et est maintenu sur GitHub (environ 3200 commits et 260 contributeurs).

En plus d'être utilisé comme logiciel de test au sein de Google, GTest est aussi utilisé dans d'autres projets open source d'envergures tels que OpenCV, le compilateur LLVM ou encore les projets Chromium (à l'origine du navigateur web Google Chrome et du système d'exploitation Chrome OS).

Enfin les grandes entreprises recommandent aussi le logiciel comme IBM qui propose un guide d'utilisation avec des applications pédagogiques⁹, Microsoft qui fournit un tutoriel pour utiliser Google Test dans Visual Studio¹⁰ ou JetBrains dans CLion¹¹.

⁶ Dépôt GitHub de Google Test : <https://github.com/google/googletest>

⁷ Blog de l'équipe "Testing Technology" de Google : <https://testing.googleblog.com/>

⁸ BSD 3-Clause : <https://opensource.org/licenses/BSD-3-Clause>

⁹ IBM - A quick introduction to the Google C++ Testing Framework : <https://developer.ibm.com/articles/au-googletestingframework/>

¹⁰ Microsoft - Guide pratique pour utiliser Google Test dans Visual Studio : <https://docs.microsoft.com/fr-fr/visualstudio/test/how-to-use-google-test-for-cpp>

¹¹ JetBrains CLion - Google Test support : <https://www.jetbrains.com/help/clion/creating-google-test-run-debug-configuration-for-test.html>

Recommandation/philosophie de Google concernant les tests

L'équipe derrière le logiciel a écrit quelques points sur leur vision de ce qu'est un bon test et de la façon à laquelle GTest les suit. En voici un résumé rapide :

- Les tests devraient être indépendants et répétables. GTest est capable d'isoler l'exécution de tests indépendants.
- Les tests devraient être bien organisés et refléter la structure du code testé. GTest est capable de grouper les tests en suite de tests (routines).
- Les tests devraient être portables et réutilisables.
- Quand les tests échouent, ils devraient fournir le plus d'informations possible concernant le problème. GTest ne s'arrête pas dès le premier échec d'un test, il s'arrête à celui qui ne fonctionne pas mais continue les tests suivants. Le détail sera affiché dans le rapport.
- Le framework de test devrait libérer les concepteurs de tests des tâches ingrates et les laisser se concentrer sur le contenu du test. GTest garde automatiquement la trace de tous les tests définis. Il ne demande pas à l'utilisateur de les énumérer afin de les exécuter.
- Les tests devraient être rapides. Avec GTest, il est possible de partager les ressources du processeur pour exécuter des tests indépendants en même temps.

Google a initialement défini ses propres termes à ne pas confondre avec ceux de l'ISTQB :

Terme Google	Terme équivalent ISTQB	Définition
Test	Test Case ¹²	À partir d'un jeu de données / de valeurs d'entrées prédéfinies, vérifier la cohérence / validité des résultats.
Test Case	Test Suite ¹³	Il s'agit d'une collection de Test Case (au sens de l'ISTQB) qui permet de tester un comportement spécifique du programme à tester.

Google a commencé à se conformer à l'ISTQB¹⁴ en remplaçant progressivement ses propres termes par ceux de l'ISTQB. Afin de lever toute ambiguïté, nous utiliserons dans notre rapport les termes anglais provenant de l'ISTQB.

¹² Test Case : <https://glossary.istqb.org/en/term/test-case-1>

¹³ Test Suite : <https://glossary.istqb.org/en/term/test-suite-2>

¹⁴ <https://github.com/google/googletest/blob/master/googletest/docs/primer.md#beware-of-the-nomenclature>

Récapitulatif

Voici un tableau récapitulant les informations principales descriptives de GTest

Google Test	
Développeur	Google (équipe "Testing Technology")
Dépôt	https://github.com/google/googletest
Licence	BSD 3-clauses
Systèmes d'exploitation	Linux, macOS, Microsoft Windows
Langages de programmation supportés	C / C++
Fonctionnalités	<ul style="list-style-type: none"> - Infrastructure de tests xUnit avec possibilité de faire des tests fixtures; - Grand jeu d'assertions prédéfinis; - Détection de potentielles défaillances fatales ou non fatales; - Paramétrage de tests; - Génération de rapports XML.
Intégration	Microsoft Visual Studio, JetBrains CLion...
Popularité	Utilisation dans les projets Chromium, LLVM ou encore OpenCV. Création de guides par de grandes entreprises (IBM, Microsoft, JetBrains...).

4. Installation de Google Test

GTest est pensé pour être portable. Ainsi aucune installation particulière n'est nécessaire afin d'utiliser GTest dans un projet C++. Nous allons dans cette partie présenter la façon recommandée par l'équipe "Testing Technology" de Google d'utiliser GTest.

Dans un premier temps il faut inclure les sources de GoogleTest dans le projet. Cela peut être fait en téléchargeant le dépôt git à la main ou en utilisant les submodules¹⁵ git afin d'ajouter une dépendance au dépôt git googletest.

La commande git pour ajouter le dépôt googletest au projet est la suivante:

```
git submodule add -b v1.10.x https://github.com/google/googletest.git
```

Les personnes voulant cloner le dépôt pourront utiliser la commande suivante afin de récupérer automatiquement le submodule défini précédemment :

```
git clone --recurse-submodules -j8 https://github.com/user/projet.git
```

Voici une organisation possible des fichiers du projet C++:

```
Root_Projet/
├── googletest/
├── Tests/
│   ├── CMakeLists.txt
│   └── test.cpp
├── CMakeLists.txt
└── Src
    ├── code.cpp
    └── code.hpp
```

Dans l'organisation ci-dessus le dossier *googletest* contient les codes sources de GTest, le dossier *Src* contient les codes de notre projet et le dossier *Tests* contient les tests que nous avons créés pour ce projet.

¹⁵ Documentation de la commande git submodule
<https://git-scm.com/book/en/v2/Git-Tools-Submodules>

Pour compiler facilement le projet il est recommandé d'utiliser CMake. Voici une façon d'écrire le CMakeLists.txt principal :

```
cmake_minimum_required(VERSION 3.15)
set(This AI03_tuto)

project(${This} CXX)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_POSITION_INDEPENDENT_CODE ON)

enable_testing()

add_subdirectory(googletest)
set(Headers Src/code.hpp)
set(Sources Src/code.cpp)

add_library(${This} STATIC ${Sources} ${Headers})
add_subdirectory(Tests)
```

CMakeFile.txt

Nous ne reviendrons pas sur l'utilisation de CMake. Notons cependant l'utilisation de la commande `enable_testing()` permettant d'informer CMake qu'il y aura des tests à lancer. Le "linkage" des codes sources de GTest est fait via la commande `add_subdirectory(googletest)` ce dossier contient son propre CMakeLists.txt et CMake saura donc comment compiler GTest sans plus d'intervention de notre part. Il est important de noter que nous devons compiler notre programme comme une librairie et non comme un exécutable, en effet c'est GTest qui implémente la fonction `main()` qui va charger notre programme en tant que librairie.

Le CMakeList.txt du dossier *Tests* est le suivant :

```
cmake_minimum_required(VERSION 3.15)
set(This gtest_AI03_Tests)

set(Sources test.cpp)

add_executable(${This} ${Sources})
target_link_libraries(${This} PUBLIC gtest_main AI03_tuto)

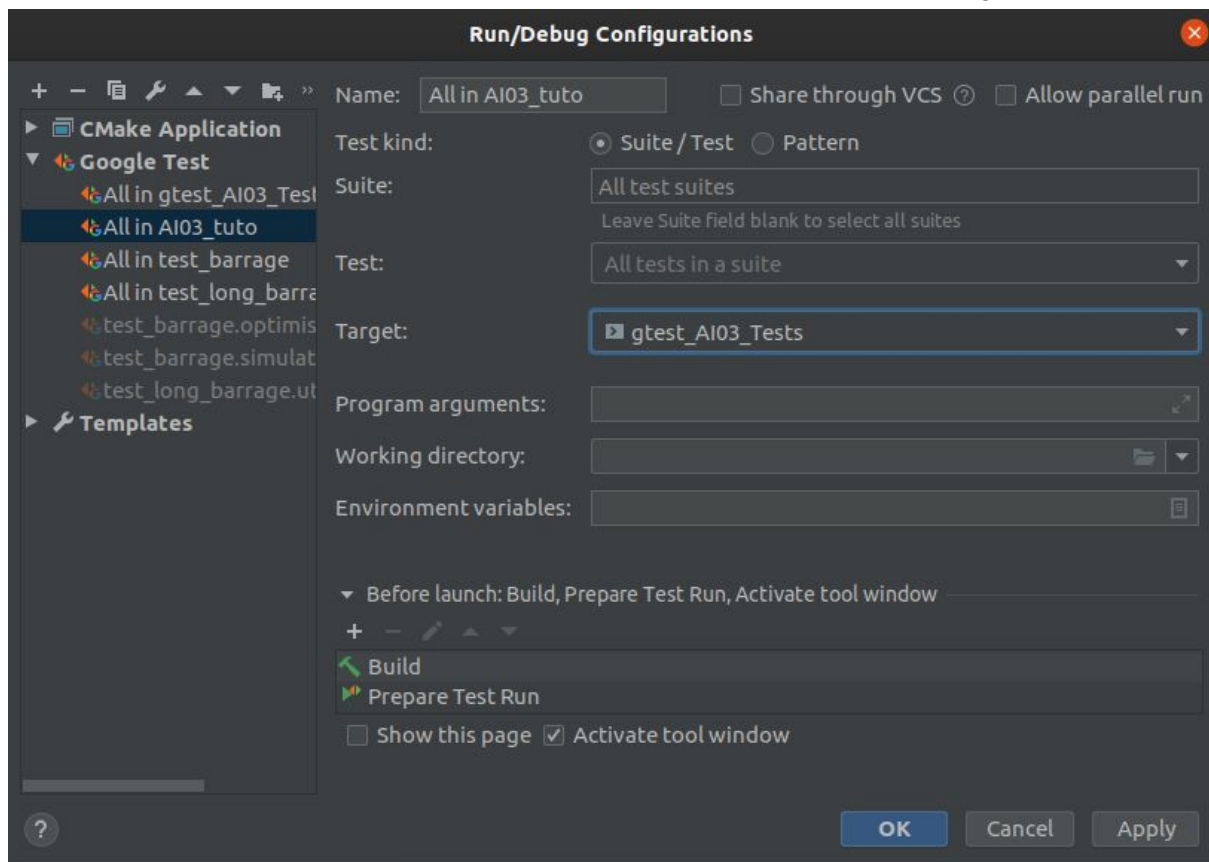
add_test( NAME ${This} COMMAND ${This})
```

Tests/CMakeFile.txt

Ici nous compilons les tests en tant qu'exécutable (car ce sont les tests qui vont charger notre programme), puis nous "linkons" les bibliothèques *gtest_main* (la fonction `main()` définie par GTest) ainsi que la bibliothèque contenant notre programme. Enfin la commande `add_test` permet à CMake de savoir que cet exécutable correspond au lancement de tests.

La mise en place de GTest est terminée. CMake va générer deux règles de compilation *AI03_tuto* permettant de compiler notre code et *gtest_AI03_tuto* permettant de compiler un exécutable lançant les tests unitaires que nous avons défini dans *Tests/test.cpp*.

Dans CLion il est ensuite possible de définir une *configuration de lancement* afin de bénéficier d'un post-traitement des sorties de GTest¹⁶ (pour plus d'information sur la visualisation de résultats de GTest voir le [chapitre 5](#) "Prise en main de Google Test").



Configuration de lancement des tests GTest dans CLion

¹⁶ Utilisation de CLion et GTest : <https://www.youtube.com/watch?v=8Up5eNZ0FLw>

5. Prise en main de Google Test

Après installation du logiciel, il est possible de réaliser des tests. Ces tests ne viennent pas polluer le programme principal, en effet ils sont écrits dans un dossier dédié aux tests (exemple : un dossier contenant tous les tests). Dans cette partie, les fonctionnalités de Google Test seront présentées sur des cas simples et concrets.

Voici notre environnement de travail appelé **GoogleTest_Conceptsbasiques** :

GoogleTest_Conceptsbasiques/	Dossier du projet
├── googletest/	Les codes sources de GTest
├── Tests/	Dossier des tests
│ ├── CMakeLists.txt	
│ └── ConceptsBasiquesTests.cpp	Source contenant le code des tests
├── CMakeLists.txt	
├── ConceptsBasiques.cpp	Source contenant le code à tester
└── ConceptsBasiques.hpp	Fichier d'en-tête du code à tester

Environnement de travail - Concepts basiques

Les fichiers en gras sont ceux que nous allons explorer afin d'expliquer d'un point de vue pratique les fonctionnalités de Google Test.

Dans **ConceptsBasiques.cpp** se trouvera les fonctions du programme à tester, il est associé à **ConceptsBasiques.hpp** son fichier d'en-tête contenant les prototype des fonctions et les classes du programme à tester. Le fichier **ConceptsBasiquesTests.cpp** contient les tests ainsi que les suites de tests spécifiques à chaque fonction/partie du programme.

Exemple introductif

Voici le contenu des fichiers sources :

```
#include "ConceptsBasiques.hpp"
#include <iostream>
using namespace std;

float surface(int r){ // Fonction qui renvoie la surface d'un disque de rayon r.
    float surface;
    if (r < 0) {
        surface = -1; // Choix arbitraire.
    }
    else {
        const float Pi = 3.14;
        surface = Pi*r; // Erreur volontaire pour afficher un test qui échoue.
    }
    if (surface == -1) {
        cout << "Erreur, le rayon est négatif." << endl;
    }
}
```

```

    }
    else {
        cout << "La surface vaut " << surface << "m²." << endl;
    }
    return(surface);
}

```

ConceptsBasiques.cpp

Dans cet exemple, la fonction `surface(int r)` est une fonction permettant de calculer la surface d'un disque de rayon `r` où `r` est un entier. Nous avons volontairement falsifié la formule d'une surface d'un disque ($\pi * r$ au lieu de $\pi * r * r$) afin de montrer des tests qui échouent.

```

#ifndef CONCEPTSBASIQUES_HPP
#define CONCEPTSBASIQUES_HPP

float surface(int r);

#endif // CONCEPTSBASIQUES_HPP

```

ConceptsBasiques.hpp

Ce fichier contient pour le moment le prototype de la fonction `surface(int r)`.

```

#include <gtest/gtest.h>
#include "../ConceptsBasiques.hpp"

TEST(SurfaceDisque, Zero) {
    EXPECT_EQ(0, surface(0));
}

TEST(SurfaceDisque, Positif) {
    EXPECT_FLOAT_EQ(3.14, surface(1));
    EXPECT_FLOAT_EQ(12.56, surface(2));
    EXPECT_FLOAT_EQ(28.26, surface(3));
    EXPECT_EQ(31400, surface(100));
    EXPECT_EQ(3140000, surface(1000));
}

TEST(SurfaceDisque, Negatif) {
    EXPECT_EQ(-1, surface(-1));
    EXPECT_EQ(-1, surface(-2));
    EXPECT_EQ(-1, surface(-100));
    EXPECT_EQ(-1, surface(-10000));
}

```

ConceptsBasiquesTests.cpp contient les tests que nous allons faire sur **ConceptsBasiques.cpp**. Il est nécessaire d'inclure **gtest.h** et le fichier d'en-tête **ConceptsBasiques.hpp** pour pouvoir tester notre logiciel. Ici nous cherchons à tester la fonction **surface(int r)**.

Basiquement la syntaxe pour réaliser un **Test Case** est la suivante :

```
TEST(nom_de_la_suite_de_test, nom_du_test) {  
    Assertions // Instructions qui vérifient si une condition est vraie.  
}
```

Syntaxe d'un Test Case sur Google Test

Prenons le premier test **TEST(SurfaceDisque, Zero)** : ici l'assertion **EXPECT_EQ(0, surface(0))** vérifie que la surface d'un disque de rayon **0** vaut bien **0**. Ici le premier paramètre de l'assertion **EXPECT_EQ** est calculé manuellement par l'utilisateur (application de la formule de l'aire d'un disque de rayon **r**), le deuxième paramètre est le résultat calculé informatiquement par exécution des instructions présentes dans la fonction **surface(int r)**.

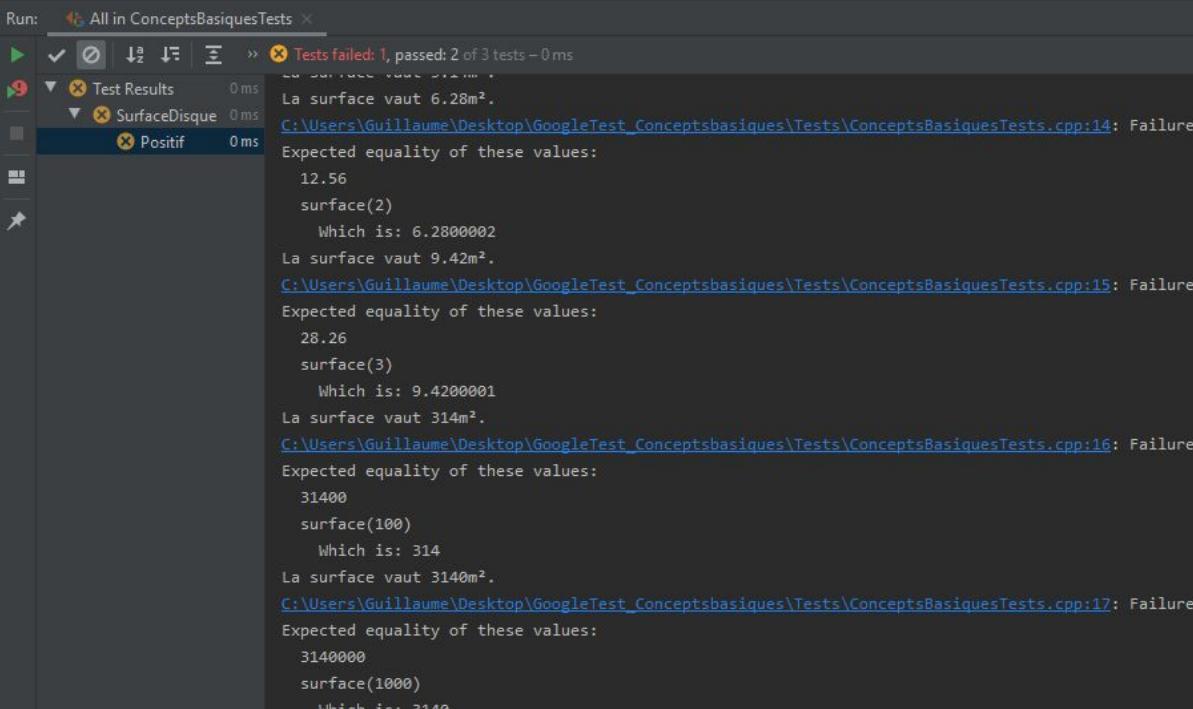
Le résultat d'une assertion est soit un **succès** soit un **échec**.

Le deuxième test **TEST(SurfaceDisque, Positif)** va tester la fonction pour des valeurs où **r** est un entier naturel strictement positif.

Et le dernier test **TEST(SurfaceDisque, Negatif)** pour des entiers relatifs strictement négatifs.

L'ensemble de ces trois tests forme une **Test Suite** : le premier paramètre de chaque test est "**SurfaceDisque**". Ceci permet de grouper les tests pour une fonction spécifique par exemple lorsqu'il s'agit de tester de grands programmes.

Pour revenir à l'exemple, dont la formule a été falsifiée, lorsque nous exécutons la **Test Suite** avec CLion voici le résultat que nous obtenons :



```

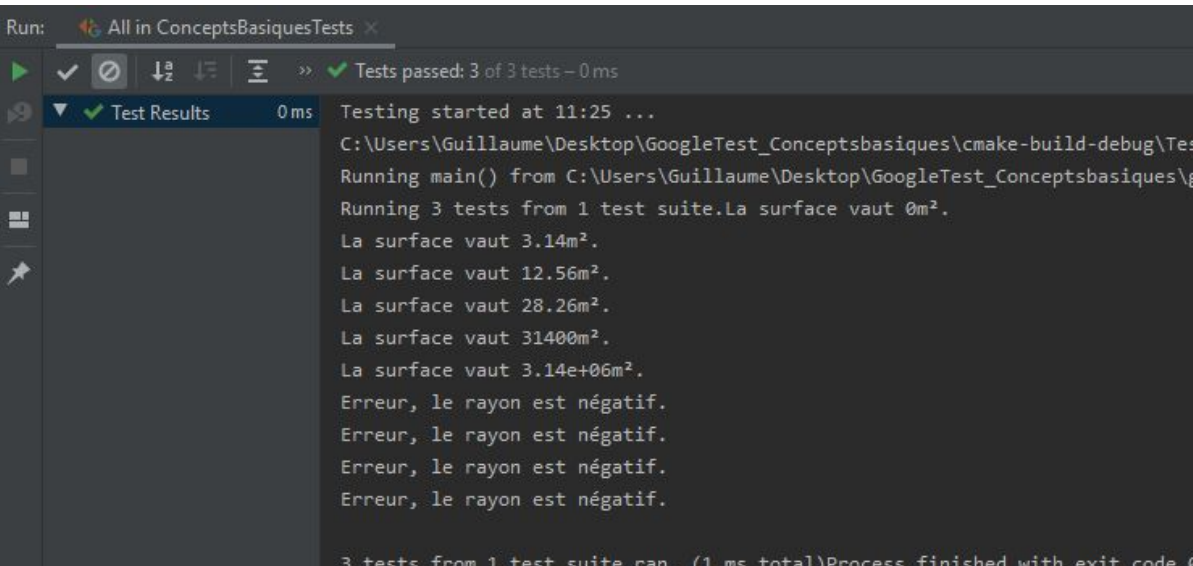
Run: All in ConceptsBasiqesTests x
Tests failed: 1, passed: 2 of 3 tests - 0 ms
Test Results 0 ms
SurfaceDisque 0 ms
Positif 0 ms
C:\Users\Guillaume\Desktop\GoogleTest_ConceptsBasiqes\Tests\ConceptsBasiqesTests.cpp:14: Failure
Expected equality of these values:
  12.56
  surface(2)
    Which is: 6.2800002
La surface vaut 9.42m².
C:\Users\Guillaume\Desktop\GoogleTest_ConceptsBasiqes\Tests\ConceptsBasiqesTests.cpp:15: Failure
Expected equality of these values:
  28.26
  surface(3)
    Which is: 9.4200001
La surface vaut 314m².
C:\Users\Guillaume\Desktop\GoogleTest_ConceptsBasiqes\Tests\ConceptsBasiqesTests.cpp:16: Failure
Expected equality of these values:
  31400
  surface(100)
    Which is: 314
La surface vaut 3140m².
C:\Users\Guillaume\Desktop\GoogleTest_ConceptsBasiqes\Tests\ConceptsBasiqesTests.cpp:17: Failure
Expected equality of these values:
  3140000
  surface(1000)
    Which is: 3140

```

Test Case Positif échouant

Nous avons le résultat du-des test-s (ici, c'est le **Test Case Positif** qui n'est pas passé) qui a-ont échoué-s avec une description de l'erreur de ce qui était attendu et des lignes auxquelles les assertions n'ont pas été validées.

En corrigeant la fonction surface avec la formule valide ($\pi \cdot r^2$), puis en réexécutant la **Test Suite** (l'ensemble des **Test Cases** : Zero, Positif et Négatif) il n'y a que des succès :



```

Run: All in ConceptsBasiqesTests x
Tests passed: 3 of 3 tests - 0 ms
Test Results 0 ms
Testing started at 11:25 ...
C:\Users\Guillaume\Desktop\GoogleTest_ConceptsBasiqes\cmake-build-debug\Test
Running main() from C:\Users\Guillaume\Desktop\GoogleTest_ConceptsBasiqes\g
Running 3 tests from 1 test suite.La surface vaut 0m².
La surface vaut 3.14m².
La surface vaut 12.56m².
La surface vaut 28.26m².
La surface vaut 31400m².
La surface vaut 3.14e+06m².
Erreur, le rayon est négatif.
Erreur, le rayon est négatif.
Erreur, le rayon est négatif.
Erreur, le rayon est négatif.
3 tests from 1 test suite ran. (1 ms total)Process finished with exit code 0

```

Tous les tests passent après correction de la formule de l'air d'un disque de rayon r

Types d'assertions

Dans l'exemple précédent nous n'avons utilisé que des assertions non fatales. Il existe de nombreuses assertions utilisables différenciées selon deux catégories : les assertions non fatales commençant par **EXPECT_** : lorsque ces assertions sont exécutées la prochaine assertion sera nécessairement exécutée quelque soit le résultat (**succès** ou **échec**) et les assertions fatales commençant par **ASSERT_**. Lorsque ces dernières sont exécutées, si le résultat d'une assertion fatale est un échec, alors toutes les assertions suivantes (y compris les tests suivants) ne seront jamais exécutées.

Voici un tableau non exhaustif des différents types d'assertions les plus utilisés sur Google Test :

	Non fatale	Fatale	Description
Assertions basiques	EXPECT_TRUE(condition);	ASSERT_TRUE(condition);	Vérifie que condition est true
	EXPECT_FALSE(condition);	ASSERT_FALSE(condition);	Vérifie que condition est false
Comparaison binaire	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
Comparaison de chaînes de caractères	EXPECT_STREQ(ch1, ch2);	ASSERT_STREQ(ch1, ch2);	Vérifie que les chaînes ch1 et ch2 sont identiques.
	EXPECT_STRNE(ch1, ch2);	ASSERT_STRNE(ch1, ch2);	Vérifie que les chaînes ch1 et ch2 ne sont pas identiques.
	EXPECT_STRCASEEQ(ch1, ch2);	ASSERT_STRCASEEQ(ch1, ch2);	Vérifie que les chaînes ch1 et ch2 sont identiques en ignorant la casse.
	EXPECT_STRCASENE(ch1, ch2);	ASSERT_STRCASENE(ch1, ch2);	Vérifie que les chaînes ch1 et ch2 ne sont pas identiques en ignorant la casse.

Tableau non exhaustif des assertions les plus utilisées sur Google Test

Tests fixtures

Les tests fixtures ont un but précis : éviter de dupliquer du code inutilement en partageant le code d'initialisation et de destruction entre les tests (**Test Cases**) d'une **Test Suite**. Le fonctionnement est basé sur l'architecture xUnit avec ses trois phases décrites dans [l'introduction](#).

Dans cet exemple, nous avons une classe **Fraction** qui possède deux attributs : **numérateur** et **denominateur** et des méthodes : un constructeur, un destructeur, des accesseurs (**getNumerateur()** et **getDenominateur()**), **setFraction()** qui est appelé par le constructeur pour fabriquer la fraction et **simplification()** qui, comme son nom l'indique simplifie au maximum une fraction dès qu'elle est instanciée.

```
#ifndef CONCEPTSBASIQUES_HPP
#define CONCEPTSBASIQUES_HPP

#include <iostream>
using namespace std;

class Fraction {
private:
    int numerateur;
    int denominateur;
    void simplification(); // Méthode simplifiant les fractions
public:
    Fraction(int n=0, int d=1) {
        // Appel à setFraction qui contrôle la validité des valeurs renseignées
        setFraction(n, d);
        cout << "Constructeur " << numerateur << "/" << denominateur << " : " << this << endl;
    }
    ~Fraction() {
        cout << "Destructeur " << numerateur << "/" << denominateur << " : " << this << endl;
    }

    int getNumerateur() const { return this->numerateur; }
    int getDenominateur() const { return this->denominateur; }
    void setFraction(int n, int d);

    Fraction const somme(Fraction const & f) const; // Calcule la somme de deux fractions
    // Syntaxe : f1.somme(f2)
};

#endif // CONCEPTSBASIQUES_HPP
```

ConceptsBasiques.hpp

Ici se trouvent les définitions des méthodes qui ne sont pas inline :

```
#include "ConceptsBasiques.hpp"
#include <iostream>
using namespace std;
```

```

void Fraction::setFraction(int n, int d) {
    numerateur = n;
    denominateur = d;
    if(d==0) {
        cerr << "erreur : denominateur = 0" << endl;
        denominateur = 1; // Choix arbitraire
    }
    simplification();
}

void Fraction::simplification(){
    // si le numerateur est 0, le denominateur prend la valeur 1
    if (numerateur==0) { denominateur=1; return; }
    /* un denominateur ne devrait pas être 0;
    si c'est le cas, on sort de la méthode */
    if (denominateur==0) return;
    /* utilisation de l'algorithme d'Euclide pour trouver le Plus Grand Commun
    Denominateur (PGCD) entre le numerateur et le denominateur */
    int a=numerateur, b=denominateur;
    // on ne travaille qu'avec des valeurs positives...
    if (a<0) a=-a; if (b<0) b=-b;
    while(a!=b){ if (a>b) a=a-b; else b=b-a; }
    // on divise le numerateur et le denominateur par le PGCD=a
    numerateur/=a; denominateur/=a;
    // si le denominateur est négatif, on fait passer le signe - au denominateur
    if (denominateur<0) { denominateur=-denominateur; numerateur=-numerateur; }
}

const Fraction Fraction::somme(const Fraction & f) const {
    return Fraction(numerateur*f.denominateur+denominateur*f.numerateur,
denominateur*f.denominateur);
+96}

```

ConceptsBasiques.cpp

Ici se trouve notre test fixture, les explications sont développées après.

```

#include <gtest/gtest.h>
#include "../ConceptsBasiques.hpp"

class FractionTest : public testing::Test {
protected:

    void SetUp() override {
        f1 = new Fraction(1, 3);
        f2 = new Fraction(2, 3);
        f3 = new Fraction(1, 0);
    }

```

```

        f4 = new Fraction(2, 4);
    }
    void TearDown() override {
        delete f1;
        delete f2;
        delete f3;
        delete f4;
    }

    Fraction * f1;
    Fraction * f2;
    Fraction * f3;
    Fraction * f4;
};

TEST_F(FractionTest, Somme) {
    Fraction SumF1F2 = f1->somme(*f2);

    EXPECT_EQ(SumF1F2.getNumerator(), 1);
    EXPECT_EQ(SumF1F2.getDenominateur(), 1);
}

TEST_F(FractionTest, DivisionZero) {
    ASSERT_NE(f1->getDenominateur(), 0); // ASSERT_NE car la potentielle
    ASSERT_NE(f3->getDenominateur(), 0); // défaillance causée par une division
    [...] // par zéro peut avoir des répercussions très graves. Le choix d'un
    // ASSERT_ au lieu d'un EXPECT_ est jugé par le testeur et dépend du besoin.
}

TEST_F(FractionTest, Simplification) {
    EXPECT_EQ(f1->getNumerator(), 1);
    EXPECT_EQ(f1->getDenominateur(), 3);
    [...]
    EXPECT_EQ(f4->getNumerator(), 1); // initialement la fraction était 2/4,
    EXPECT_EQ(f4->getDenominateur(), 2); // ce qui fait 1/2 après simplification.
}

```

Tests/ConceptsBasiques.cpp

La syntaxe pour réaliser un test fixture est la suivante :

```

class TestFixture : public testing::Test { // cette classe dérive de testing::Test
    protected: // pour que les membres soient accessibles par d'éventuelles
                // sous-classes.
    void SetUp() override {
        // Appelé avant l'exécution de chaque test, c'est ici qu'on initialise le

```

```

    // variables utilisées par chaque test.
}
void TearDown() override {
    // Appelé après l'exécution de chaque test, c'est ici qu'on effectue le
    // nettoyage (ex : delete un pointeur si initialisé dans SetUp()).
}

// Déclaration des variables que l'on souhaite utiliser dans nos tests de
// TestSuite.
};

```

Préparation d'un test fixture

Pour revenir à notre exemple, nous avons créé une classe **FractionTest** dans laquelle on initialise des fractions (**f1**, **f2**, **f3** et **f4** pointeurs sur **Fraction**) dans le **SetUp()** qui seront bien évidemment libérés dans le **TearDown()** après l'exécution des tests.

Quant aux tests, ils s'écrivent de la même façon que les tests normaux hormis le nom (**TEST_F**) car les tests se font dans le cadre d'un test fixture :

```

TEST_F(nom_du_test_fixture, nom_du_test) {
    Assertions // Instructions qui vérifient si une condition est vraie.
}

```

Syntaxe d'un Test Case pour un test fixture sur Google Test

Dans notre exemple, nous testons la somme de deux fractions, la division par zéro ainsi que la méthode de simplification d'une fraction. Nous considérons que la division par zéro est un point important pour la sécurité, c'est pourquoi nous avons mis une assertion fatale. L'ensemble des tests partagent le même code (les méthodes **SetUp()** et **TearDown()** seront appelées à chaque test. Attention : ce n'est pas un partage des instances d'objets **f1**, **f2**, **f3** et **f4**. Chaque test va exécuter le **SetUp()** et donc instancier ces quatre objets. C'est bien un partage de code, mais pas un partage des instances. C'est un atout d'un point de vue de la sécurité puisque les quatre instances utilisées pour le test de division par zéro ne sont pas les mêmes que pour le test de simplification. Ce serait un problème si un test modifiait une instance utilisée pour un autre test. On peut donc affirmer que chaque **Test Case** est totalement indépendant d'un autre **Test Case** au sein d'un **test fixture**. Et c'est aussi un atout pour le testeur : cette "factorisation" de code lui fournit un gain de temps considérable pour de grands programmes à tester.

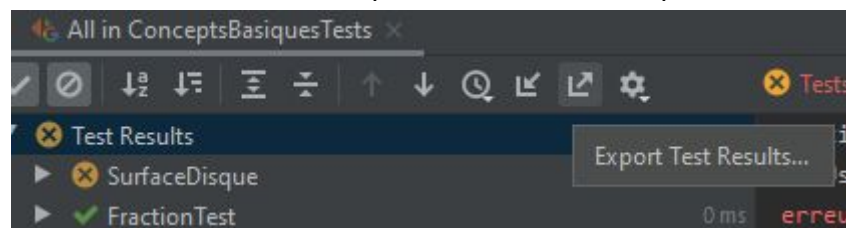
À titre indicatif, tous les tests de cet exemple ont réussi !

Tous ces concepts permettent d'utiliser la quasi totalité des fonctionnalités de GTest. Si ces concepts basiques ne répondent à un besoin extrêmement spécifique, il est toutefois possible de fabriquer tout un environnement à la main. Google explique dans sa documentation comment réaliser ceci en écrivant sa propre fonction `main()` soi-même sur Google Test Primer¹⁷. Cependant, dans la majorité des cas il n'est pas nécessaire d'avoir recours à cette méthode.

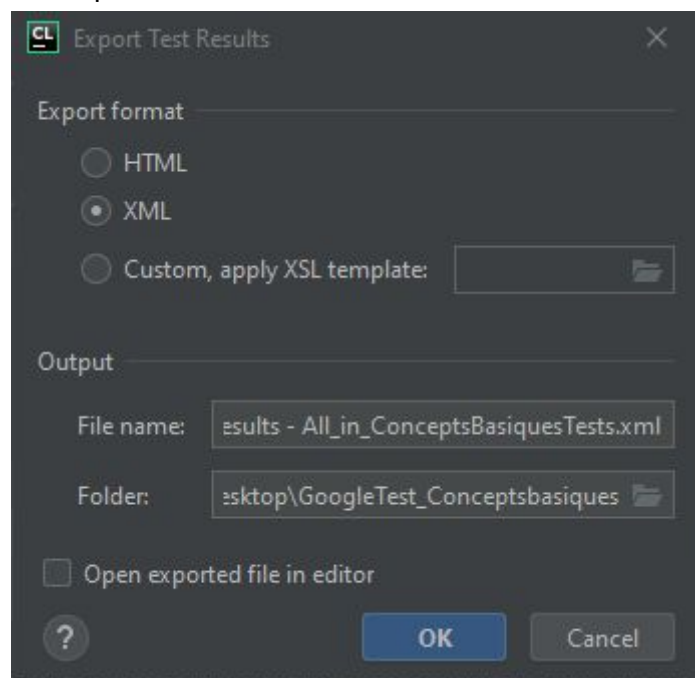
Génération d'un rapport au format XML

Afin de sauvegarder les résultats des tests, Google Test implémente une fonctionnalité permettant l'export des résultats au format XML. Cette fonctionnalité s'intègre parfaitement avec l'EDI CLion :

- a. Après exécution du test, il suffit de cliquer sur le bouton d'export :



- b. Choix du format et emplacement de destination :



¹⁷ Google Test Primer - Writing the main() Function
<https://github.com/google/googletest/blob/master/googletest/docs/primer.md#writing-the-main-function>

c. Affichage du rapport XML dans l'éditeur de notre choix :

```
<?xml version="1.0" encoding="UTF-8"?><testrun duration="2" footerText="Generated by CLion on 11/01/2020 13:37" name="
  <count name="total" value="6"/>
  <count name="failed" value="1"/>
  <count name="passed" value="5"/>
  <config nameIsGenerated="true" PASS_PARENT_ENVS_2="true" PROJECT_NAME="ConceptsBasiques" TARGET_NAME="ConceptsB
  <suite duration="2" locationUrl="gtest://suite:SurfaceDisque" name="SurfaceDisque" status="failed">
    <test duration="2" locationUrl="gtest://suite:SurfaceDisque/test:Zero" name="Zero" status="passed">
      <output type="stdout">La surface vaut 0m².
    </output>
  </test>
  <test duration="0" locationUrl="gtest://suite:SurfaceDisque/test:Positif" name="Positif" status="failed">
    <output type="stdout">La surface vaut 3.14m².
  </test>
  <test duration="0" locationUrl="gtest://suite:SurfaceDisque/test:Negatif" name="Negatif" status="failed">
    <output type="stdout">La surface vaut 6.28m².
  </test>
</suite>
</testrun>
```

C:\Users\Guillaume\Desktop\GoogleTest_Conceptsbasiques\Tests\ConceptsBasiquesTests.cpp:14: Failure
Expected equality of these values:
12.56
surface(2)

6. Cas d'étude

Dans ce chapitre nous allons présenter un cas d'utilisation réaliste de GTest afin de développer un programme d'optimisation de gestion d'un barrage vis-à-vis d'un signal de prix se basant sur l'algorithme de Bellman-Ford. GTest sera utilisé pour écrire des tests unitaires dans une optique de Test Driven Development¹⁸ (TDD).

L'ensemble des codes présentés dans ce chapitre (avec quelques tests et commentaires supplémentaires) est disponible sur le repo git : https://gitlab.utc.fr/asoltani/ai03_gtest

Nous vous conseillons d'ouvrir le projet dans un EDI moderne afin de suivre ce chapitre.

Notre projet est organisé comme suivant:

```
AI03_tuto/
├── googletest/
├── tests/
│   ├── CMakeLists.txt
│   ├── test_barrage.cpp
│   └── test_long_barrage.cpp
├── CMakeLists.txt
├── Barrage
│   ├── Barrage.cpp
│   └── Barrage.hpp
```

Pour plus d'information sur comment compiler le projet vous pouvez vous référer au chapitre ["Installation de GTest"](#).

Initialisation de la classe Barrage

Comme nous adoptons une approche de TDD nous allons écrire nos tests avant d'implémenter le code correspondant. Dans un premier temps, écrivons les tests correspondant à la création d'un nouvel objet *Barrage*.

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

#include "../Barrage/Barrage.hpp"

TEST(test_barrage, setup_barrage) {
    Barrage b(2, 3, 1, 1, 0);
    vector<float> prix{1, 2, 3};
    vector<int> ruissellement{1, 2, 3, 4};

    ASSERT_THROW(b.setRuissellement(ruissellement).setPrixMarcheElec(prix),
invalid_argument);
    ASSERT_THROW(b.setPrixMarcheElec(prix).setRuissellement(ruissellement),
invalid_argument);
}
```

¹⁸ Définition du TDD : https://en.wikipedia.org/wiki/Test-driven_development


```

    ruissellement.pop_back();

    Barrage b1(2, 3, 1, 1, 0);
    b1.setPrixMarcheElec(prix).setRuissellement(ruissellement);

    ASSERT_THROW(Barrage b2(0, 0, -1, 0, 0), invalid_argument);
    ASSERT_THROW(Barrage b2(0, 0, 0, 0, 1), invalid_argument);
    ASSERT_THROW(Barrage b2(1, 0, 0, 0, 0), invalid_argument);
}

```

tests/test_barrage.cpp

Nous utilisons la macros `ASSERT_THROW` défini dans GTest afin de s'assurer que, si nos arguments sont invalides, notre code retourne bien une erreur de type `invalid_argument`. Nous pouvons ensuite écrire le code de la classe `Barrage` afin de faire en sorte que ce test passe.

```

#ifndef AI03_TUTO_BARRAGE_HPP
#define AI03_TUTO_BARRAGE_HPP

#include ...

using namespace std;
using namespace Eigen;

class Barrage {
    uint volume_eau; // m**3
    uint volume_eau_max; // m**3
    float rendement; // MWh/m**3
    uint debit_max; // m**3/h
    uint debit_min; // m**3/h

    // ruissellement représente le volume d'eau entrant dans le barrage
    vector<int>* ruissellement = nullptr; // m**3
    // simplification: prix de marché == prix de vente
    vector<float>* prix_marche_elec = nullptr; // €/MWh

public:
    Barrage(uint volumeEau, uint volumeEauMax, float rendement, uint debitMax, uint
debitMin);
    ~Barrage();

    Barrage& setRuissellement(vector<int>& r);
    Barrage& setPrixMarcheElec(vector<float>& p);
};

#endif //AI03_TUTO_BARRAGE_HPP

```

Barrage/Barrage.hpp

```

#include "Barrage.hpp"

Barrage::Barrage(uint volumeEau, uint volumeEauMax, float rendement, uint debitMax, uint

```

```

debitMin) :
    volume_eau(volumeEau), volume_eau_max(volumeEauMax), rendement(rendement),
    debit_max(debitMax), debit_min(debitMin)
{
    if (rendement < 0)
        throw invalid_argument("Le rendement ne peut pas être négatif.");
    if (debit_max < debit_min)
        throw invalid_argument("Le débit max doit être supérieur au débit min.");
    if (volume_eau > volume_eau_max)
        throw invalid_argument("Le volume d'eau doit être inférieur au volume d'eau
max.");
}

Barrage::~Barrage() {
    delete ruissellement;
    delete prix_marche_elec;
}

Barrage& Barrage::setRuissellement(vector<int>& r) {
    if (prix_marche_elec && r.size() != prix_marche_elec->size())
        throw invalid_argument("La dimension temps de correspond pas");

    ruissellement = new vector<int>;
    for (auto& v : r)
        ruissellement->push_back(v);
    return *this;
}

Barrage &Barrage::setPrixMarcheElec(vector<float> &p) {
    if (ruissellement && p.size() != ruissellement->size())
        throw invalid_argument("La dimension temps de correspond pas");

    prix_marche_elec = new vector<float>;
    for (auto& v : p)
        prix_marche_elec->push_back(v);
    return *this;
}

```

Barrage/Barrage.cpp

Comme nous développons en TDD nous n'implémentons que le strict nécessaire pour que nos tests passent.

Initialisation des matrices

Afin d'utiliser l'algorithme de Ford-Bellman il nous faut une structure dans laquelle enregistrer la valeur associée à chaque état possible de notre barrage ainsi que l'état précédant cet état. Pour cela nous allons définir deux matrices de taille *volume-d'eau-max-du-barrage* x *nombre-de-pas-de-temps*, ces matrices seront initialisées avec la valeur -1. Ci-dessous le test correspondant à cette fonctionnalité (nous utilisons la librairie Eigen¹⁹ afin de manipuler des matrices).

```
TEST(test_barrage, initialisation_matrices) {
    int v_max = 3;
    Barrage b(2, v_max, 1, 1, 0);
    vector<float> prix{1, 2, 3};

    ASSERT_THROW(b.getGrilleBellmanValeur(), runtime_error);
    ASSERT_THROW(b.getGrilleBellmanParent(), runtime_error);

    b.setPrixMarcheElec(prix);
    auto expected_grille_valeur = Barrage::GrilleBellmanValue::Constant(v_max+1,
    prix.size()+1, -1.0f);
    auto& grille_valeur = b.getGrilleBellmanValeur();
    EXPECT_EQ(expected_grille_valeur, grille_valeur) << "grille de bellman des
valeurs:\n" << grille_valeur << '\n';

    Barrage b1(2, v_max, 1, 1, 0);
    b1.setPrixMarcheElec(prix);
    auto expected_grille_parent = Barrage::GrilleBellmanParent::Constant(v_max+1,
    prix.size()+1, -1);
    auto& grille_parent = b.getGrilleBellmanParent();
    EXPECT_EQ(expected_grille_parent, grille_parent) << "grille de bellman des
parents:\n" << grille_parent << '\n';
}
```

tests/test_barrage.cpp

Ici nous utilisons l'opérateur << (ex : EXPECT_EQ(expected_grille_valeur, grille_valeur) << "grille de bellman des valeurs:\n" << grille_valeur << '\n';) afin d'afficher un message lorsque nos assertions échouent car le message généré par défaut par GTest ne permet pas de voir les valeurs contenues dans les matrices Eigen comparées.

Voici les ajouts à la classe Barrage permettant de passer ces tests :

```
class Barrage {
public:
    typedef Matrix<float, Dynamic, Dynamic> GrilleBellmanValue;
    typedef Matrix<int, Dynamic, Dynamic> GrilleBellmanParent;

private:
    // Grilles utilisées dans Bellman
    GrilleBellmanValue * grille_bellman_valeur = nullptr;
    GrilleBellmanParent * grille_bellman_parent = nullptr;
}
```

¹⁹ Documentation de la librairie Eigen : <http://eigen.tuxfamily.org/>

```
void initialiserGrilleBellmanValeurs();
void initialiserGrilleBellmanParents();
};
```

Barrage/Barrage.hpp

```
Barrage::GrilleBellmanParent& Barrage::getGrilleBellmanParent() {
    if (grille_bellman_parent)
        return *grille_bellman_parent;
    initialiserGrilleBellmanParents();
    return *grille_bellman_parent;
}

Barrage::GrilleBellmanValue& Barrage::getGrilleBellmanValeur() {
    if (grille_bellman_valeur)
        return *grille_bellman_valeur;
    initialiserGrilleBellmanValeurs();
    return *grille_bellman_valeur;
}
```

Barrage/Barrage.cpp

Simulation / calcul des valeurs de Bellman

Nous pouvons maintenant penser à la fonctionnalité de simulation (remplissage de la grille des valeurs de Bellman et de la grille des parents). Afin de tester ceci nous définissons un exemple simple pour lequel nous avons fait tourner l'algorithme de Ford-Bellman (avec valorisation de fin de stock) à la main. Nous allons donc vérifier que notre implémentation C++ de cet algorithme nous donne les mêmes résultats :

```
TEST(test_barrage, simulation1) {
    int v_max = 3;
    Barrage b(2, v_max, 2, 1, 0);
    vector<float> prix = {2, 3, 1};
    vector<int> ruissellement = {1, 2, 0};
    b.setPrixMarcheElec(prix).setRuissellement(ruissellement);

    b.simuler();
    auto& grille_valeur = b.getGrilleBellmanValeur();
    auto& grille_parent = b.getGrilleBellmanParent();

    Barrage::GrilleBellmanValue expected_grille_valeur(v_max+1, prix.size()+1);
    expected_grille_valeur <<
        -1.0f,  -1.0f,  -1.0f,  0.0f,
        -1.0f,  -1.0f,  -1.0f,  4.0f,
        0.0f,   4.0f,   -1.0f,  20.0f,
        -1.0f,  0.0f,   10.0f,  22.0f;

    Barrage::GrilleBellmanParent expected_grille_parent(v_max+1, prix.size()+1);
    expected_grille_parent <<
        -1,  -1,  -1,  -1,
        -1,  -1,  -1,  -1,
        -1,  2,  -1,  3,
        -1,  2,   2,  3;

    ASSERT_EQ(expected_grille_valeur.cols(), grille_valeur.cols());
```

```

    ASSERT_EQ(expected_grille_valeur.rows(), grille_valeur.rows());
    EXPECT_EQ(expected_grille_valeur, grille_valeur) << "grille de bellman des
valeurs:\n" << grille_valeur << '\n';

    ASSERT_EQ(expected_grille_parent.cols(), grille_parent.cols());
    ASSERT_EQ(expected_grille_parent.rows(), grille_parent.rows());
    EXPECT_EQ(expected_grille_parent, grille_parent) << "grille de bellman des
parents:\n" << grille_parent << '\n';
}

```

tests/test_barrage.cpp

Ci-dessous le code de la fonction simuler.

```

void Barrage::simuler() {
    auto &valeurs = getGrilleBellmanValeur();
    auto &parents = getGrilleBellmanParent();

    valeurs(volume_eau, 0) = 0.0f;

    for (size_t pdt = 0; pdt < valeurs.cols(); pdt++) {
        for (int volume = 0; volume < valeurs.rows(); volume++) {
            if (valeurs(volume, pdt) != -1) {
                for (int turbinee = debit_min; turbinee <= debit_max; turbinee++) {
                    int nouveau_volume = min(volume - turbinee + (*ruissellement)[pdt],
                                             static_cast<int>(valeurs.rows()) - 1);

                    // Impossible d'atteindre un volume d'eau négatif
                    if (nouveau_volume >= 0) {
                        float nouvelle_valeur = valeurs(volume, pdt)
                                                + (*prix_marche_elec)[pdt] * rendement
                                                * static_cast<float>(turbinee);

                        if (pdt + 1 < valeurs.cols() &&
                            nouvelle_valeur > valeurs(nouveau_volume, pdt + 1)) {
                            valeurs(nouveau_volume, pdt + 1) = nouvelle_valeur;
                            parents(nouveau_volume, pdt + 1) = volume;
                        }
                    }
                }
            }
        }
    }

    // valorisation du stock de fin
    int benefice_moyen = static_cast<int>(accumulate(prix_marche_elec->begin(),
prix_marche_elec->end(), 0.0) / prix_marche_elec->size() * rendement);
    for (int volume_final = 0; volume_final < static_cast<int>(valeurs.rows());
        volume_final++) {
        valeurs(volume_final, valeurs.cols() - 1) = max(0.0f, valeurs(volume_final,
valeurs.cols() - 1)) + static_cast<float>(benefice_moyen * volume_final);
    }
}

```

Optimisation / recherche du meilleur chemin

Enfin, la dernière fonctionnalité dont nous avons besoin est la fonctionnalité d'optimisation (recherche du chemin maximisant les bénéfices dans la grille de Bellman). Encore une fois nous avons dans un premier temps fait tourner l'algorithme à la main afin de pouvoir définir ce test. La politique de gestion ainsi calculée est représentée par la suite des volumes d'eau que doit contenir le barrage à chaque pas de temps.

```
TEST(test_barrage, optimisation1) {
    Barrage b(2, 3, 2, 1, 0);
    vector<float> prix = {2, 3, 1};
    vector<int> ruissellement = {1, 2, 0};
    b.setPrixMarcheElec(prix).setRuissellement(ruissellement);

    ASSERT_THROW(b.optimiser(), runtime_error);

    b.simuler();

    vector<int> expected_policy{2, 2, 3, 3};
    EXPECT_EQ(expected_policy, b.optimiser()) << "grille de bellman des valeurs:\n" <<
    b.getGrilleBellmanValeur() << '\n';
}
```

tests/test_barrage.cpp

Le code de la fonction optimiser est le suivant :

```
vector<int> Barrage::optimiser() {
    auto& valeurs = getGrilleBellmanValeur();
    auto& parents = getGrilleBellmanParent();

    // Recherche du meilleur future possible
    int v_max = -1;
    float b_max = -1.0f;
    for (int v=static_cast<int>(valeurs.rows())-1; v>=0 ; v--) {
        if (valeurs(v, valeurs.cols()-1) > b_max) {
            v_max = v;
            b_max = valeurs(v, valeurs.cols()-1);
        }
    }
    if (v_max == -1)
        throw runtime_error("optimisation impossible");

    // Recherche de la meilleure politique de gestion du barrage pour atteindre ce future
    int volume = v_max;
    vector<int> meilleure_politique(valeurs.cols());
    for (int pdt=static_cast<int>(meilleure_politique.size()-1); pdt > 0; pdt--)
    {
        meilleure_politique[pdt] = volume;
        volume = parents(volume, pdt);
    }
}
```

```
meilleure_politique[0] = volume_eau;  
  
return meilleure_politique;  
}
```

Barrage/Barrage.cpp

Nous avons vu avec cet exemple comment intégrer GTest dans le développement d'un projet réaliste. L'avantage de cette méthode de développement où nous définissons les tests unitaires de chaque fonctionnalité avant de les coder est que nous pouvons approcher une couverture de code de 100% assez facilement. Aussi, le refactoring du code existant peut se faire sans crainte de modifier le comportement du programme car ce comportement est défini et testé dans les tests unitaires.

GTest nous permet de définir rapidement et facilement les tests unitaires dont nous avons besoin et est donc un très bon outil pour implémenter le TDD en C++.

7. Conclusion

GTest a comme tout logiciel des avantages et des faiblesses capitalisées dans ce tableau :

Points forts	Points faibles
<ul style="list-style-type: none"> • Détection automatique des tests • Multiplateforme • Support un grand nombre d'assertions différentes fatales et non fatales. • Tests indépendants et parallélisables, grandes performances. • Les résultats des tests sont exportables en XML. • Équipe de développement très active • Intégration dans les EDI modernes 	<ul style="list-style-type: none"> • Aucune interface utilisateur autre que le terminal. • Nécessite un niveau minimum requis en C/C++ et quelques connaissances en POO²⁰.

Avantages/Inconvénients de GTest

En conclusion, GTest est un outil simple mais puissant. Sa prise en main sera facile pour un développeur habitué au C++ mais demandera un certain nombre de prérequis qu'un développeur avec peu d'expérience n'a pas forcément.

L'utilisation la plus courante de GTest est celle présenté dans le chapitre "[Cas d'étude](#)". Elle consiste à utiliser GTest pour définir des tests unitaires dans une optique de Test Driven Development. La portabilité et flexibilité de GTest le rend parfaitement adapté à cette tâche.

²⁰ Programmation orientée objet