## Unit testing

Als etablierte Methode der Softwareentwicklung bezeichnet *unit testing* die konzeptionelle Aufteilung von selbst entwickelten Programmen in *units* und das unabhängige Testen dieser units durch eigens dafür geschrieben Tests.

Bei der Aufteilung in units sollte auf Unabhängigkeit und Testbarkeit geachtet werden. Es soll möglich sein durch einige, wiederrum als Code formulierbare, Beispiele jeweils i.W. die gesamte Komplexität des Verhaltens jeder dieser units abzudecken. Insb. muss es möglich sein units separat voneinander zu testen sodass fehlgeschlagene Tests klar mit einer konkreten unit im Code assoziiert sind und die Fehlersuche daher erleichtern.

Es ist wichtig die unit tests oft auszuführen, z.B. bei jedem build, um stets einen Eindruck (oder sogar eine quantifizierbare Metrik) von der Korrektheit des Programms zu haben.

Oft werden für das etablieren von unit tests in einem Projekt sog. *unit testing frameworks* verwendet. Es handelt sich hierbei um Programmbibliotheken die die Formulierung und das Ausführen von unit tests erleichtern.

## GoogleTest

GoogleTest ist ein quelloffenes unit testing framework, dass in der Meson WrapDB zur Verfügung steht.

Kerninhalt von GoogleTest ist eine Sammlung von Präprozessor-Makros, die zum Ausdrucken von Erwartung innerhalb eines Tests und zur Deklaration der Tests selbst dienen:

**`TEST`** Nimmt zwei Parameter; einen Namen für eine *Test-Suite*, d.h. Sammlungen von Tests für die gleiche unit, und einen Namen für den konkreten Test, der deklariert werden soll. Beide Parameter sind C++-identifier, insb. keine strings o.Ä..
Das Makro löst auf zu Text, der an Stelle einer Funktionsdeklaration treten kann.

**`TEST_F`** Nimmt zwei Parameter; einen Namen einer vorher definierten Klasse, die von `testing::Test` abgeleitet ist und einen Namen für den konkreten Test, der deklariert werden soll.
Das Makro löst ebenfalls zu Text auf, der an Stelle einer Funktionsdeklaration treten kann.

Als Unterschied zu `TEST` führt die Verwendung von `TEST_F` statt zu der Definition einer Funktion zur Definition einer Methode einer wiederrum vom ersten Parameter abgeleiteten Klasse. Für Ausführen des Tests erstellt GoogleTest ein Objekt dieser Klasse mit dem Standardkonstruktor und ruft die Methode auf diesem Objekt auf. Innerhalb des Körpers des Tests hat man so direkten Zugriff auf die Attribute und Methoden die in der Klasse vereinbart wurden, deren Name als erster Parameter an `TEST_F` übergeben wurde, sofern diese in abgeleiteten Klassen sichtbar sind (also mindestens `protected`).

GoogleTest bezeichnet das erstellte Objekt bzw. die Klasse als *fixture*.

**Assertions** GoogleTest liefert zahlreiche Makros zum Ausdrucken von Erwartungen zur Verwendung innerhalb von Tests. Die unten aufgeführten Makros prüfen jeweils eine Eigenschaft ihrer Parameter und generieren bei Ausführung des Tests eine Fehlermeldung, falls die Eigenschaft nicht erfüllt ist. Es kann

notwendig sein die Parameter der Makros jeweils in (ansonsten redundante) runde Klammern zu fassen, da der Präprozessor in den Argumenten enthaltene Kommas und schließende runde Klammern ansonsten evtl. fehl interpretiert.

| | |
|---|---|
| `EXPECT_EQ(`$v_1$`, `$v_2$`)` `EXPECT_NE(`$v_1$`, `$v_2$`)` `EXPECT_LT(`$v_1$`, `$v_2$`)` `EXPECT_LE(`$v_1$`, `$v_2$`)` `EXPECT_GT(`$v_1$`, `$v_2$`)` `EXPECT_GE(`$v_1$`, `$v_2$`)` | Prüft, dass $v_1$ `==` $v_2$ bzw. $v_1$ `!=` $v_2$ bzw. $v_1$ `<` $v_2$ bzw. $v_1$ `<=` $v_2$ bzw. $v_1$ `>` $v_2$ bzw. $v_1$ `>=` $v_2$ |
| `EXPECT_FLOAT_EQ(`$v_1$`, `$v_2$`)` `EXPECT_DOUBLE_EQ(`$v_1$`, `$v_2$`)` | Prüft, dass die gegebenen Werte als `float` bzw. `double` gleich sind bis auf eine vordefinierte Genauigkeit (4 ULPs[1]) |
| `EXPECT_NEAR(`$v_1$`, `$v_2$`, `$\varepsilon$`)` | Prüft, dass $\lvert v_1 - v_2 \rvert \leq \varepsilon$ |
| `EXPECT_NO_THROW(`$e$`)` | Prüft, dass bei der Auswertung des Ausdrucks $e$ keine exception geworfen wird |

*Beispiel.* Innerhalb eines kleinen Software-Projekts implementieren wir eine Funktion `add`, die zwei Werte vom Typ `int` addiert und versehen diese in einer separaten Übersetzungseinheit mit unit tests.

Wir bauen das Software-Projekt mit Meson und definieren in der Meson-Spezifikation einen `test`, sodass wir diesen später bequem mit `meson test -C build` ausführen können.

In der Variable `gtest_main_dep` hinterlegt die Meson-Spezifikation des subproject `gtest` geeignete Quelldateien, sodass ein von GoogleTest mitgeliefertes Hauptprogramm Teil wird der Übersetzungsenheit für die ausführbare Binärdatei `gtest_add`. Das Hauptprogramm durchsucht die Übersetzungseinheit nach allen Vorkommnissen der Makros `TEST` und `TEST_F` und führt die damit spezifizierten Tests aus.

gtest_add/add.h

```
#pragma once

int add(int a, int b);
```

gtest_add/add.cpp

```
#include "add.h"

int add(int a, int b) {
  return a + b;
}
```

gtest_add/add_unittest.cpp

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

namespace {
  TEST(Add, Negative) {
    EXPECT_EQ(add(2, -2), 0);
  }
```

[1] https://web.archive.org/web/20240204145454/https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

```
  TEST(Add, Four) {
    EXPECT_EQ(add(2, 2), 4);
  }
}
```

gtest_add/main.cpp

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

using namespace std;

int main() {
  int a, b;
  while (cin >> a >> b)
    cout << add(a, b) << endl;
}
```

gtest_add/meson.build

```
project('add', 'cpp')

add_lib = library('add_lib', 'add.cpp')
executable('add', 'main.cpp', link_with : add_lib)

gtest = subproject('gtest')
test('gtest',
  executable('gtest_add', 'add_unittest.cpp', 'add.cpp', dependencies :
  ↪  gtest.get_variable('gtest_main_dep'))
)
```

*Beispiel.* Wir implementieren unit tests für die Klasse `vector` aus der STL unter Verwendung einer eigens erstellten Klasse als test fixture.

gtest_vector/vector_unittest.cpp

```
#include <vector>
#include <algorithm>
#include <functional>
#include <gtest/gtest.h>

namespace {
  using namespace std;

  class VectorFixture : public testing::Test {
    protected:
      vector<int> vs;

      VectorFixture(): vs({1, 4, 3, 2, 5}) {}
  };

  TEST_F(VectorFixture, DoubleReverse) {
    vector<int> vs_orig(vs);
```

```
    reverse(vs.begin(), vs.end());
    reverse(vs.begin(), vs.end());
    EXPECT_EQ(vs_orig, vs);
  }
  TEST_F(VectorFixture, SortReverse) {
    sort(vs.begin(), vs.end(), greater<int>());

    vector<int> vs_comm(vs);
    sort(vs_comm.begin(), vs_comm.end(), less<int>());
    reverse(vs_comm.begin(), vs_comm.end());

    EXPECT_EQ(vs, vs_comm);
  }
}
```