TDD, gtest i nie tylko

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
AGH University of Science and Technology

Dominik Dziuba 10-01-2019

Testy jednostokowe

Co to jest test jednostkowy?
To metoda testowania oprogramowania, w której pojedyncze metody, funkcje czy obiekty są indywidualnie sprawdzane pod kątem odpowiedniego działania.
Pozwala na wczesne wykrywanie problemów w fazie projektowania.

TDD

Jest to technika projektowania oprogramowania, w której często testy są pisane, zanim implementacja ma miejsce. Następnie napisane testy są uruchamiane, by sprawdzić błędy w istniejącym kodzie po czym pisany jest kod właściwy, tak aby testy się powiodły. Kod jest „sprzątany” po czynnościach, które miały powodować zdawanie testów, które są uruchamiane ponownie. Ten proces jest powtarzany aż do uzyskania końcowego produktu.

gtest

1. Testy powinny być niezależne i powtarzalne.
2. Testy powinny być dobrze zorganizowane i odzwierciedlać strukturę kodu.
3. Testy powinny być przenośne i możliwe do ponownego użytku.
4. Kiedy testy się nie udają, powinny dawać jak najwięcej informacji.
5. Framework powinien pozwalać twórcy testów na skupienie się w całości na zawartości.
6. Testy powinny być szybkie.

Parafrazowane z: https://github.com/google/googletest/blob/master/googletest/docs/primer.md

Dostępne asercje

Fatalna asercja	Niefatalna asercja	Sprawdza
ASSERT_EQ(val1, val2);	EXPECT_EQ(val1, val2);	val1 == val2
ASSERT_NE(val1, val2);	EXPECT_NE(val1, val2);	val1 != val2
ASSERT_LT(val1, val2);	EXPECT_LT(val1, val2);	val1 < val2
ASSERT_LE(val1, val2);	EXPECT_LE(val1, val2);	val1 <= val2
ASSERT_GT(val1, val2);	EXPECT_GT(val1, val2);	val1 > val2
ASSERT_GE(val1, val2);	EXPECT_GE(val1, val2);	val1 >= val2

Pierwszy test

```
TEST(nazwa_serii_testow, nazwa_konkretnego_testu)
{
      /* kod testu - asercje, etc... */
}

TEST(nazwa_serii_testow, nazwa_innego_konkretnego_testu)
{
      /* kod testu - asercje, etc... */
}
```

przyklad1.cpp

Test fixtures

```
class nazwa_typu_naszej_struktury : public ::testing::Test {
protected:
        void SetUp() override {
                /*
                ta funkcja jest wołana na początku każdego testu i działa de
                facto jako konstruktor,
                nic nie stoi na przeszkodzie by kod ten umieścić w
                konstruktorze
                */
        }

        void TearDown() override {
                /*
                podobnie jak SetUp ale odpowiednikiem jest destruktor
                */
        }
};

TEST_F(nazwa_typu_naszej_struktury, nazwa_testu) {
        /*
        kod testu - asercje, etc...
        */
}
```

przyklad2.cpp

Więcej asercji

Nie trzeba polegać na makrach dostępnych w bibliotece gtest.
Można samemu definiować stan testu używając:

```
SUCCEED();

FAIL();
ADD_FAILURE();
ADD_FAILURE_AT("ścieżka_pliku", numer_linii);
```

przyklad3.cpp

Predykaty asercji

Dla lepszych wiadomości o błędach można użyć poniższych makr.
W przypadku gdy predykat zwraca wartość bool można użyć:

Fatalna asercja	**Niefatalna asercja**	**Sprawdza**
ASSERT_PRED1(pred1, val1);	EXPECT_PRED1(pred1, val1);	pred1(val1) jest prawdą
ASSERT_PRED2(pred2, val1, val2);	EXPECT_PRED2(pred2, val1, val2);	pred2(val1, val2) jest prawdą
...	...	...

przyklad4.cpp

Predykaty asercji

Dla jeszcze bardziej dostosowanych wiadomości o błędach można użyć predykatu zwracającego obiekt klasy AssertionResult. gtest udostępnia wygodne funkcje dla używania tej klasy.

```
namespace testing {
// adekwatnie do nazw funkcje zwracają odpowiedni
//obiekt AssertionResult
AssertionResult AssertionSuccess();
// można, podobnie jak do FAIL() strumieniować naszą
// własną wiadomość
AssertionResult AssertionFailure();
}
```

przyklad5.cpp

Predykaty asercji z formatem

Jeżeli zajdzie taka potrzeba, można samemu zdefiniować format wypisywanej wiadomości, zależnie od własnych potrzeb. Funkcja formatująca ma wtedy sygnaturę

```
::testing::AssertionResult predykatZFormatem(const char* wyr1,
                                              const char* wyr2,
                                              ...
                                              const char* wyrn,
                                              T1 war1,
                                              T2 war2,
                                              ...
                                              Tn warn);
```

przyklad6.cpp

Asercje liczb zmiennoprzecinkowych

Fatalna asercja	**Niefatalna asercja**	**Sprawdza**
ASSERT_FLOAT_EQ(val1, val2);	EXPECT_FLOAT_EQ(val1, val2);	dwie wartości typu float są prawie róne
ASSERT_DOUBLE_EQ(val 1, val2);	EXPECT_DOUBLE_EQ(val 1, val2);	dwie wartości typu double są prawie równe
ASSERT_NEAR(val1, val2, abs_error);	EXPECT_NEAR(val1, val2, abs_error);	dwie wartości nie różnią się o zadaną wartość

Asercje korzystające z gMock'owych porównywaczy

gtest w nowszych wersjach jest dostarczany razem z google-mock, które posiada wiele przydatnych porównywaczy(matcher). Mogą się okazać bardzo przydatne przy walidowaniu string'ów itp.

przyklad7.cpp

Printer'y

gtest wie jak wydrukować proste typy, kontenery STL, natywne tablice oraz typy z zaimplementowanym operatorem <<. Jeżeli ten nie jest zdefiniowany gtest drukuje obiekt bajt po bajcie licząc na to, że programista będzie mógł z tej informacji coś uzyskać. W sytuacji, gdy nie można zmienić/dodać tego operatora można napisać funkcję PrintTo o takiej sygnaturze

```
void PrintTo(const nasz_typ& n_t, std::ostream* o);
```

przyklad8.cpp

Testy śmierci

Fatalne asercje	Niefatalne asercje	Sprawdza
ASSERT_DEATH(statement, regex);	EXPECT_DEATH(statement, regex);	czy statement kończy się z podanym kodem
ASSERT_DEATH_IF_SUPPORTED(statement, regex);	EXPECT_DEATH_IF_SUPPORTED(statement, regex);	jeżeli testy śmierci są wspierane robi to co zwykły test śmierci
ASSERT_EXIT(statement, predicate, regex);	EXPECT_EXIT(statement, predicate, regex);	czy statement kończy się z kodem zgadzającym się z predykatem

Przydatna flaga: --gtest_death_test_style="threadsafe"

przyklad9.cpp

Scoped trace

Czasami ciężko jest znaleźć wywołanie, które powoduje błąd testu. Scoped trace pozwala na łatwiejsze znajdowanie asercji, które się nie powiodą. W takiej sytuacji można użyć makra:

```
SCOPED_TRACE(wiadomosc);
```

przyklad10.cpp

Asercje i jak działają

Trzeba pamiętać, że fatalne asercje opuszczają jedynie obecną funkcję. Może dojść do sytuacji, że pożądanym działaniem będzie zakończenie testu w sytuacji wystąpienia błędu, gdyż następujące operacje mogą okazać się niepożądane w skutkach.

przyklad11.cpp
przyklad12.cpp

Testy parametryzowane wartościami

Żeby uniknąć pisania wielu takich samych testów, różniących się wartościami można skorzystać z testów parametryzowanych wartościami, co potencjalnie może znacznie ułatwić pracę programiście.
gtest oferuje generatory wartości:

Generator	**Zachowanie**
Range(begin, end [, step])	Zwraca wartości od begin, przez begin + step, do end – step.
Values(v1, v2, ..., vN)	Zwraca wartości w podanej kolejności.
ValuesIn(container) and ValuesIn(begin,end)	Zwraca wartości z kontenerów STL jak i z tablic oraz iteratorów.
Bool()	Zwraca sekwencję {false, true}.
Combine(g1, g2, ..., gN)	Zwraca kombinacje wartości(iloczynu kartezjańskiego) w postaci std::tuple

Ciekawe opcje gtest

Program wygenerowany przy pomocy gtest może przyjować różne flagi, które mogą robić różne rzeczy:

- ./program --gtest_filter=*filtr*
- ./program --gtest_repeat=*liczba*
- ./program --gtest_break_on_failure
- ./program --gtest_shuffle (--gtest_random_seed=*SEED*)
- ./program --gtest_output="xml:ścieżka_do_pliku_wynikowego"

Trochę o gMock

Zdjęcie zapożyczone z: https://www.express.co.uk/

przyklad14.cpp

Akcje

Jeżeli funkcja nie ma wyspecyfikowanej akcji a ma typ jest wbudowany:

- void – to po prostu robi return;
- bool – zwraca false
- każda inna – zwraca 0
- od c++11 – jeżeli typ ma domyślny konstruktor to ta akcja jest wykonywana

Ale jeśli żadne z powyższych nie zachodzi to można wyspecyfikować akcję używając i narzucić ich ilość:

- .WillOnce()
- .WillRepeatedly()

przyklad15.cpp

Specyfikowanie wielu oczekiwań

Można zadeklarować wiele oczekiwań, występujących po sobie. Należy pamiętać, że sprawdzane są one w odwrotnej kolejności do zapisu(z dołu do góry) oraz że każde wywołanie, które satysfakcjonuje napotkane wymaganie, ale które już było usatysfakcjonowane, generuje błąd testu.

przyklad16.cpp

A co jeśli interesuje nas kolejność zapisu?

Załóżmy, że mamy potrzebę narzucenia sprawdzania kolejności oczekiwań. Domyślnie te oczekiwania nie są sprawdzane w kolejności. W takim przypadku można użyć klasy InSequence, która w danym bloku kodu działa jako flaga.

```
::testing::InSequence i;
```

przyklad17.cpp

Jak pozbyć się wypełniania oczekiwań?

Wystarczy użyć .RetiresOnSaturation().
Daje to ciekawe efekty w połączeniu z wymuszaniem kolejności sprawdzania oczekiwań.

```
EXPECT_CALL(obiekt, wywolanie_funkcji)
                    .specyfikatory_ilosci()
                    .RetiresOnSaturation();
```

przyklad18.cpp

A jak mock’ować nie-virtualne metody i zwykłe funkcje

Prawie tak samo jak w poprzednich przykładach jeśli chodzi o metody. Trzeba jednak pamiętać, że w tym przypadku klasy mock’owe są niepowiązane z mock’owaną klasą.
Natomiast funkcje, to tak, żeby działało. Należy utworzyć prosty interfejs dla niej, a potem wszystko dokładnie tak samo jak przy klasach.

przyklad19.cpp

BONUS: benchmark

Ma podobną konstrukcję do gtest, jest to mała biblioteka, ale ma całkiem spore możliwości.

```
void nazwa_funkcji_testujacej(benchmark::State s)
{
      for(auto& _ : s)
            {
                  // magia
            }
}

BENCHMARK(nazwa_funkcji_testujacej); // rejestracja
BM_MAIN();
```

przyklad20.cpp

Źródła

- https://github.com/google/googletest
- https://github.com/google/benchmark