Zaawansowane CPP/Wykład 4: Testowanie

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wstęp

Programowanie rozumiane jako pisanie kodu jest tylko częścią procesu tworzenia oprogramowania. Analiza i opis tego procesu jest przedmiotem inżynierii oprogramowania i znacznie wykracza poza ramy tego wykładu. Niemniej chciałbym pokrótce w tym wykładzie poruszyć jedno zagadnienie związane bezpośrednio z programowaniem - testowanie. Testowanie jest nieodłączną częścią programowania i powinno być obowiązkiem każdego programisty. Jak będę się starał Państwa przekonać, testowanie to może być coś więcej niż "tylko" sprawdzenie poprawności kodu.

Testowanie

Testowanie wydaje się oczywistą koniecznością w przypadku każdego programu komputerowego (choć co rok zdarza mi się spotkać studentów przekonanych o swojej nieomylności :)). Mniej oczywiste jest stwierdzenie kto, gdzie, kiedy i co ma testować. To w ogólności bardzo złożony problem, ale tu chciałbym się ograniczyć do tzw. testów jednostkowych. Wyrażenie "test jednostkowy" należy interpretować jako test jednej jednostki. Przez pojedynczą jednostkę będziemy rozumieli: funkcję, metodę lub klasę. Zadaniem takiego testu jest sprawdzenie czy dana jednostka działa poprawnie.

Dlaczego w ogóle pisać takie testy? Czy nie wystarczy przetestowanie całego programu? Testować cały program też oczywiście trzeba. Służą do tego liczne testy odbioru, integracyjne itp., wykonywane poprzez dedykowane zespoły. Ale im większą część kodu testujemy, tym trudniejsze są testy i tym trudniej będzie znaleźć przyczynę wykrytej nieprawidłowości działania programu. Testy jednostkowe wykrywają błędy (a przynajmniej ich część) "u źródła", często w bardzo prostym kodzie, a więc ich poprawianie może być dużo szybsze.

Jak się zastanowić, to testowanie każdej wykonanej jednostki przed użyciem jej w dalszym kodzie powinno być oczywistą koniecznością. No, ale równie oczywiste jest, że należy uprawiać sporty, nie palić papierosów, nie jeździć po pijanemu, itp. Statystyki dobitnie jednak pokazują, że natura człowiecza grzeszną jest i łatwo ulegamy słabościom, w tym wypadku pokusie nietestowania programów, a powodem jest jeden z siedmiu grzechów głównych czyli lenistwo. Testy nie piszą, ani nie wykonuja się same, w rzeczywistości wymagają całkiem sporego nakładu pracy (czasami większego niż pisanie testowanego kodu!). Część programistów traktuje ten wysiłek jako "czas stracony", tzn. nie wykorzystany na pisanie kodu, za który im płacą.

To wszystko jest prawdą, ale w rzeczywistości nie możemy uniknąć tej straty czasu. Jest to tylko kwestia wyboru gdzie i ile tego czasu stracimy: czy na pisanie testów w trakcie kodowania jednostek i poprawianie prostych błędów, czy na szukanie błędów w dużo większych fragmentach programu? Doświadczenie wykazuje,

że czas potrzebny na znalezienie i poprawienie błędu jest w tym drugim przypadku dużo większy, w krańcowych przypadkach poprawienie programu może być wręcz niemożliwe. Testy jednostkowe skalują się liniowo z rozmiarem pisanego kodu, a nie ekspotencjalnie jak np. testy integracyjne. Oczywiście testy jednostkowe nie rozwiążą wszystkich problemów, jeśli mamy zły projekt całości to nic nam nie pomogą idealnie działające jednostki, ale przynajmniej będzie nam łatwiej się o tym przekonać.

Jak już napisałem testy służą do sprawdzania poprawności działania danej jednostki: **ciągłego** sprawdzanie działania tej jednostki. Ponieważ kod się nieustannie zmienia, powinniśmy wykonywać testy cały czas aby sprawdzić czy te zmiany wynikające, np. z poprawienia wykrytych błędów, nie wprowadziły nowych usterek. Aby to było możliwe testy musza być łatwe do wykonywania czyli zautomatyzowane. Nie mogą polegać na tym, że puszczamy program, a następnie przeglądamy wydruk. Musimy "nacisnąć guzik" a w odpowiedzi zapali nam się szereg "światełek": zielone wskażą testy zaliczone, a czerwone testy niezaliczone. Taki automatyczny proces testujący może być zintegrowany z innymi narzędzimi programistycznymi, takimi jak np. `make`.

Refaktoryzacja

Wysiłek włożony w napisanie takich powtarzalnych testów nie jest zaniedbywalny, ale korzyści są duże. Możliwość przetestowania kodu w dowolnym momencie to więcej niż tylko świadomość, że nasz obecny kod przeszedł testy. Taka możliwość, w połączeniu z narzędziami zarządzającymi kodem źródłowym, pozwala na bezpieczne dokonywanie zmian według schematu: zmieniamy, testujemy, jeśli testy się nie powiodą szukamy błedów, jeśli ich nie znajdujemy to cofamy zmiany. Takie podejście jest szczególnie pomocne, a właściwie niezbędne, podczas programowania przyrostowego i refaktoryzacji.

Programowanie przyrostowe to technika zalecana dla większości projektów, która polega na programowaniu "malymi krokami", czyli na kolejnym dodawaniu nowych funkcji do istniejącego działajacego kodu. Testy umożliwiają sprawdzenie czy dodany kod nie wprowadził błędów do starej części. Oczywiście każdy przyrost wymaga stworzenia nowego zestawu testów.

Refaktoryzacja to zmiana kodu bez zmiany jego funkcjonalności w celu poprawy jego jakości.

Projektowanie sterowane testami

Pisanie testu sprawdza również nasze zrozumienie tego, co dana jednostka ma robić. W tym sensie testy stanowią sformalizowany zapis wymagań. To zaleta, którą trudno przecenić. Jeżeli nie wiemy jak przestestować daną jednostkę, to najprawdopodobniej nie powinniśmy się wcale brać za jej kodowanie. Dlatego niektóre metodologie (np. extreme programming) zalecają projektowanie sterowane testami, czyli zaczęcie pisania programu od pisania testów do niego. Przebieg pracy przy takim podejściu wygląda następująco:

1. Piszemy test
2. Kompilujemy test
3. Test się nie kompiluje
4. Piszemy tyle kodu aby test sie skompilował
5. Test się kompiluje
6. Wykonujemy test
7. Test najprawdpodobniej nie wykonuje się poprawnie
8. Poprawiamy/dopisujemy kod tak aby test się wykonał
9. Test wykonuje się poprawnie

Osobiście wydaje mi się, że zalety tego podejścia są ogromne. Nawet jeśli brakuje czasu i ochoty na pisanie testów, to należy po prostu zastanowić się nad sposobem przetestowania naszego kodu zanim zaczniemy go pisać. To znakomicie zmusza do ścisłego określenia tego, co właściwie nasz program ma robić.

Testy

Po tej całej propagandzie - czas na testowanie w praktyce. Przetestujemy przykłady wprowadzone w poprzednich rozdziałach, zaczynając od funkcji `max`:

```
template<typename T> T max(T a,T b) {return (a>b)?a:b;}
```

Być może część z Państwa oburzy się: jak to? testować tę jedną linijkę? Gdyby chodziło o tę jedną linijkę to rzeczywiście wystarczy na nią popatrzeć (i mieć wiarę w kompilator), ale przypominam, że dodaliśmy do niej dwie przeciążone wersje i dwie specjalizacje. A to oznacza, że ta linijka nie działa poprawnie w każdym przypadku. Napiszemy więc testy, które to wyłapią. Jak już pisałem będzie to jednoczesne zdefiniowanie tego jak funkcja `max` ma działać. Po chwili zastanowienia ustalamy więc, że nasza funkcja `max` zwraca:

- większą z dwu przekazanych wartości (większą w sensie porównania operatorem >)
- jeśli argumentami są wskaźniki to `max` zwraca wskaźnik na większą wartość
- jeśli argumentami są wskaźniki na `char` to `max` traktuje je jako napisy i zwraca wskaźnik do napisu większego zgodnie z uporządkowaniem leksykalnym.

Testować funkcję `max` będziemy poprzez porównanie wartości zwracanej do wartości poprawnej, którą sami wskażemy:

```
assert(max(1,2)==2);
assert(max(2,1)==2);
```

Należy też sprawdzić przypadek symetryczny

```
assert(max(1,1)==1);
assert(max(2,2)==2);
```

Testowanie szablonu jest trudne, bo w zasadzie musimy rozważyć przekazanie argumentów dowolnego typu. Co więcej, ten typ wcale nie musi posiadać operatora porównania ==. Aby sprawdzić działanie `max` na jakimś typie niewbudowanym posłużymy się własną klasą:

```
class Int {
private:
 int _val;
public:
 Int(int i):_val(i) {};
 int val()const {return _val;};
 friend bool operator>(const Int &a,const Int &b)  {
 return a._val>b._val;
 }
};
```

Kod testowy może teraz wyglądać następująco:

```
Int i(5);
Int j(6);
assert(max(i,j).val() == 6);
assert(max(j,i).val() == 6);
assert(max(j,j).val() == 6);
assert(max(i,i).val() == 5);
```

Następnie testujemy wersję wskaźnikową:

```
assert(max(&i,&j)->val() == 6);
assert(max(&j,&i)->val() == 6);
assert(max(&j,&j)->val() == 6);
assert(max(&i,&i)->val() == 5);
```

i na koniec wersję dla napisów (`const char *`):

```
assert(0==strcmp(max("abcd","acde"),"acde"));
assert(0==strcmp(max("acde","abcd"),"acde"));
assert(0==strcmp(max("abcd","abcd"),"abcd"));
assert(0==strcmp(max("acde","acde"),"acde"));
```

oraz (`char *`):

```
char s1[]="abcde";
char s2[]="ac";
assert(0==strcmp(max(s1,s2),s2));
assert(0==strcmp(max(s2,s1),s2));
assert(0==strcmp(max(s1,s1),s1));
assert(0==strcmp(max(s2,s2),s2));
```

Napisy zostały tak dobrane, żeby miały pierwszy znak identyczny, ponieważ procedura ogólna dla wskaźników porównuje właśnie pierwsze znaki. To pokazuje jak ważny jest wybór danych testowych. Testy jednostkowe są testami "białej skrzynki", tzn. piszący testy ma wgląd do implementacji testowanego kodu (powinien to być ten sam programista), można więc przygotować dane testowe tak, aby pokrywały możliwie wiele ścieżek wykonywania programu, warunków brzegowych itp.

Oczywiście bezbłędne przejście powyższych testów nie gwarantuje nam jeszcze poprawności funkcji `max` (tego zresztą nie zagwarantuje nam żaden test), ale bardzo zwiększa prawdopodobieństwo tego, że tak jest.

Pozostaje nam do wytestowanie procedura szukająca maksimum w tablicy. To zadanie zostawiam jako ćwiczenie dla czytelników.

Powyższy przykład może budzić pewne obawy. Kod funkcji `max` liczy około 20 linii, a kod testujący 40. No cóż, testowanie kosztuje, tylko czy możemy sobie pozwolić na zrezygnowanie z niego? Rozważmy powyższy przykład, te dwadzieścia linii kodu definiującego szablon `max` jest dużo bardziej skomplikowane niż te czterdzieści linii kodu testujacego, tak że jeśli porównamy czas pisania, to nie wypada to już tak źle. Kod szablonu `max` zawiera przeciążenia i specjalizację, czyli wykorzystany jest algorytm rozstrzygania przeciążenia. Osobiście nie ufam swojej znajomości tego algorytmu na tyle, aby nie sprawdzić kodu. Jeżeli i tak wykonujemy jakieś testy to warto zainwestować trochę więcej pracy i przygotować porządny zestaw testujący. Może też się zdarzyć, że będziemy chcieli dodać kolejne przeciążenia, które mogą wpłynąć w niezamierzony sposób na przeciążenia już istniejące. Gotowy programik testujący pozwoli nam to szybko wykryć.

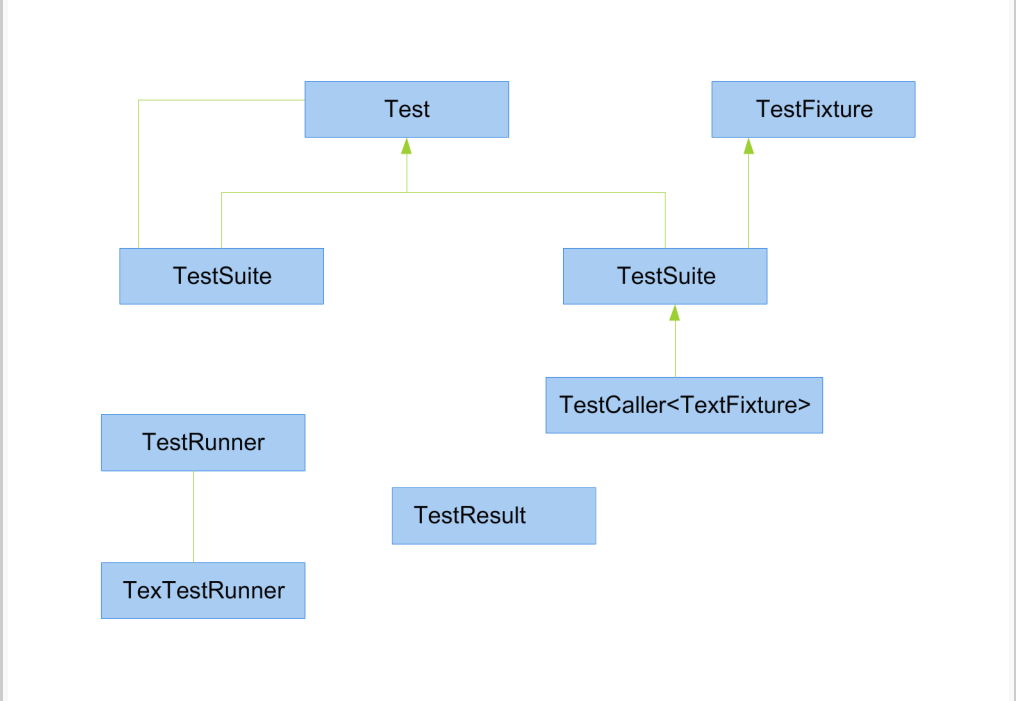
Pozostaje jeszcze sprawa poprawności samego programu testującego, w końcu to też jest kod i może zawierać błędy. Czy więc musimy pisać kod testujący program testujący, i tak dalej w nieskończoność? Na szczęście nie, błędy w kodzie testującym mogą mieć dwa efekty. Pierwszy to wykazanie błędu, który nie jest błędem, w tej sytuacji taki błąd wykrywa się sam. Musimy tylko pamiętać podczas szukania źródeł wykazanych przez program testujacy usterek, że mogą one pochodzić z kodu testującego. Drugi rodzaj błędu to nie wykrycie błędu w programie. Występowanie tego rodzaju błedów testujemy poprzez wprowadzanie do programu zamierzonych błędów i sprawdzając czy nasze testy je wykryją.

CppUnit

Rysunek 4.1 przedstawia częściową hierachię klas w szkielecie `CppUnit`.

Jest to bardzo mała część, uwzględniająca tylko te klasy, które zostaną użyte w naszym przykładzie. Proszę przede wszystkim zwrócić uwagę na hierachię klasy `Test`.

Jest to klasyczny wzorzec `Kompozyt` (zob. E. Gamma, R. Helm, R. Johnson, J. Vlissides *"Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku"*), umożliwiający dowolne składanie i zagnieżdżanie testów. Jest to możliwe dzieki temu, że klasa `TestSuite` może zawierać inne testy, w tym też inne `TestSuite`.



Rysunek 4.1. Częściowa hierarchia klas w szkielecie `CppUnit`.

Każda klas z tej hierarchii zawiera funkcję `run (TestResult *result)`, która wywołuje funkcję `runTest()`, którą musimy przeładować definiując nasz własny test. Klasa `TestResult`, a raczej jej obiekty, służą do zbierania wyników testów.

Nasz przykład z `max` możemy zapisać następująco:

```
#include <cppunit/TestResult.h>
#include <cppunit/TestCase.h>

class Max_test : public CppUnit::TestCase {
public:

  void runTest() {
    assert(max(1,2)==1); sztucznie wprowadzony błąd!
    assert(max(2,1)==2);

    assert(max(1,1)==1);
    assert(max(2,2)==2);

    .
    .
    .
};
```

(Źródło: max_test_case.cpp)

Powyższy test możemy wywołać następująco:

```
main() {
CppUnit::TestResult result;
Max_test max;

max.run( &result );
}
```

Jak na razie większym kosztem uzyskaliśmy ten sam efekt co programik z poprzedniego podrozdziału :). Czas więc wykorzystać jakieś własności szkieletu.

Zaczniemy od wykorzystania, zamiast standardowego makra `assert`, makr szkieletu `CppUnit`:

```
#include <cppunit/TestResult.h>
#include <cppunit/TestCase.h>
#include <cppunit/TestAssert.h>

class Max_test : public CppUnit::TestCase {
public:

  void runTest() {
    CPPUNIT_ASSERT_EQUAL(1,max(1,2)); sztucznie wproadzony błąd!
    CPPUNIT_ASSERT_EQUAL(2,max(2,1));

    CPPUNIT_ASSERT_EQUAL(1,max(1,1));
    CPPUNIT_ASSERT_EQUAL(2,max(2,2));
    .
    .
    .
};
```

(Źródło: max_test_runner.cpp)

Jednak wykonanie tego samego kodu `main` co poprzednio nie da żadnych wyników! To dlatego, że makra `CPPUNIT_ASSERT_...` nie powodują przerwania programu tylko zapisanie wyników do obiektu typu `TestResult`. Najłatwiej odczytać te wyniki nie uruchamiając testów samemu tylko za pośrednictwem obiektu `TextUi::TestRunner`.

```
#include <cppunit/ui/text/TestRunner.h>

main() {
Max_test *max = new Max_test;
CppUnit::TextUi::TestRunner runner;

runner.addTest( max );
max musi byc wskaźnikiem utworzynym za pomocą new bo runner
przejmuje go na własność i sam zwalnia przydzieloną dla niego pamięć
runner.run();
}
```

(Źródło: max_test_runner.cpp)

Teraz uruchomienie programu spowoduje wydrukowanie na `std::cout` raportu z wykonania testu. Ciągle jednak jest to niewspółmierne do włożonego wysiłku. Postaramy się więc teraz zacząć strukturyzować nasze testy. Możemy oczywiście stworzyć kilka różnych klas testujących z inną funkcją `runTest()` każda, ale zamiast tego wykorzystamy klasę `TestFixture`. Rzut oka na rysunek 4.1 pokazuje, że nie jest to `Test`, ale przerobimy ją na test za pomocą `TestCaller`.

Zaczynamy od zdefiniowanie szeregu testów w jednej klasie:

```
#include <cppunit/TestFixture.h>
#include <cppunit/TestAssert.h>

class Max_test : public CppUnit::TestFixture {
    Int i,j;
public:
 Max_test():i(5),j(6) {};

 void int_test() {
    CPPUNIT_ASSERT_EQUAL(1,max(1,2)); sztucznie wprowadzony błąd!
    ...
    CPPUNIT_ASSERT_EQUAL(1,max(2,2)); sztucznie wprowadzony błąd!
  }
  void class_test() {
    CPPUNIT_ASSERT_EQUAL( 6,max(i,j).val() );
```

```
    ...
  }
  void class_ptr_test() {
    CPPUNIT_ASSERT_EQUAL( 6,max(&i,&j)->val() );
    ...
  }
  void const_char_test() {
    CPPUNIT_ASSERT_EQUAL(strcmp(max("abcd","acde"),"acde"),0);
    ...
  }
  void char_test() {
    char s1[]="abcde";
    char s2[]="ac";
    CPPUNIT_ASSERT_EQUAL(strcmp(max(s1,s2),s2),0);
    ...
    CPPUNIT_ASSERT_EQUAL(strcmp(max(s2,s2),s2),1);
    sztucznie wprowadzony błąd!
}
```

(Źródło: max_test_caller.cpp)

Podzieliliśmy nasz test na kilka mniejszych, z których każdy testuje zachowanie jednej klasy argumentów funkcji `max`. Zmienne używane wspólnie zadeklarowalismy jako składowe klasy i inicjalizujemy w konstruktorze. Potem pokażemy jak można wywoływać dodatkowy kod inicjalizujący przed wykonaniem każdej metody, ale na razie ten mechanizm nie jest nam potrzebny.

No, ale jak wywołać te testy? Klasa `TestFixture` nie jest testem i nie posiada funkcji `runTest`. Aby móc jej użyć musimy skorzystać z szablonu `TestCaller`. Szablon `TestCaller` przyjmuje jako swój parametr klasę `TextFixture`, w naszym przypadku `Max_test`. W konstruktorze obiektów tej klasy podajemy nazwę testu i adres metody klasy `Max_test`, która ten test implementuje. Tak skonstruowany obiekt jest już testem i możemy go przekazać do obiektu `runner`:

```
main() {

CppUnit::TextUi::TestRunner runner;

 runner.addTest( new CppUnit::TestCaller<Max_test>(
                       "int_test",
                       &Max_test::int_test ) );
... podobnie dla reszty metod klasy Maxtest

runner.run();
}
```

(Źródło: max_test_caller.cpp)

Teraz wykonanie programu spowoduje wywołanie 5 testów. Co ważne - niepowodzenie ktorejś z asercji w jednym teście przerywa wykonywanie tego testu, ale nie ma wpływu na wykonywanie się innych testów.

Zamiast dodawać testy do "wykonywacza" pojedynczo, możemy je najpierw pogrupować za pomocą obiektów `TestSuite`:

```
main() {
CppUnit::TextUi::TestRunner runner;
 CppUnit::TestSuite *obj_suite = new CppUnit::TestSuite;
 CppUnit::TestSuite *ptr_suite = new CppUnit::TestSuite;

 obj_suite->addTest( new CppUnit::TestCaller<Max_test>(
                       "int_test",
                       &Max_test::int_test ) );
 obj_suite->addTest( new CppUnit::TestCaller<Max_test>(
                       "class_test",
                       &Max_test::class_test ) );
  ptr_suite->addTest( new CppUnit::TestCaller<Max_test>(
                       "class_ptr_test",
```

```
                        &Max_test::class_ptr_test ) );
  ptr_suite->addTest( new CppUnit::TestCaller<Max_test>(
                        "const_char_test",
                        &Max_test::const_char_test ) );
  ptr_suite->addTest( new CppUnit::TestCaller<Max_test>(
                        "char_test",
                        &Max_test::char_test ) );
  runner.addTest(obj_suite);
  runner.addTest(ptr_suite);
  runner.run();
}
```

Obiekty `TestSuite` są testami i możemy je dalej grupować:

```
 CppUnit::TestSuite *max_suite = new CppUnit::TestSuite;

 max_suite->addTest(obj_suite);
 max_suite->addTest(ptr_suite);
 runner.addTest(max_suite);

runner.run();
```

Widać, że szkielet `CppUnit` daje nam sporo możliwości, ale ciągle jest to niewspółmierne do włożonego wysiłku. Powodem tego jest prostota użytego przykładu, który nie wymaga takich narzędzi. Pakiet `CppUnit` posiada jednak o wiele więcej możliwości, między innymi:

- W klasie `TextFixture` można przeładować funkcje `void setUp()` i `void tearDown()`, które będą wywoływane odpowiednio przed i po wykonaniu każdego testu. Mogą być użyte do konstrukcji środowiska testowego i jego rozmontowania. Nie używałem tej możliwości, bo nie była ona potrzebna w tak prostym przykładzie.
- Bardzo często chcemy wykonać razem wszystkie testy zdefiniowane w jednej klasie. `CppUnit` dostarcza makr ułatwiających grupowanie wszystkich metod danej klasy w jeden `TestSuite`.
- Szkielet `CppUnit` oferuje poza `CPPUNIT_ASSERT_EQUAL` szereg innych makr ulatwiających pisanie testów.
- Stosunkowo łatwo można zmienić format wyświetlania wyników testów np. dodanie

```
    runner.setOutputter( new CppUnit::XmlOutputter(
                           &runner.result(),
                           std::cout ) );
```

spowoduje wypisanie wyników testu w formacie XML.