



Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Laboratorium informatyki

Ćwiczenie nr 14. Testy oprogramowania

Zagadnienia do opracowania:

- cel i systematyka testów oprogramowania
- testy jednostkowe
- Test-Driven Development (TDD)
- Google Test framework

Spis treści

1	Cel ćwiczenia	2
2	Wprowadzenie	2
2.1	Testowanie oprogramowania	2
2.2	Test-Driven Development	4
2.3	Google Test framework	5
2.3.1	Instalacja i konsolidacja	5
2.3.2	Implementacja testów jednostkowych	8
2.3.3	Testowanie wywołań zwrotnych	17
3	Program ćwiczenia	27
4	Dodatek	28
4.1	System budowania CMake	28

1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z ideą i systematyką testów oprogramowania, ze szczególnym uwzględnieniem opanowania umiejętności implementacji prostych testów jednostkowych.

2. Wprowadzenie

2.1. Testowanie oprogramowania

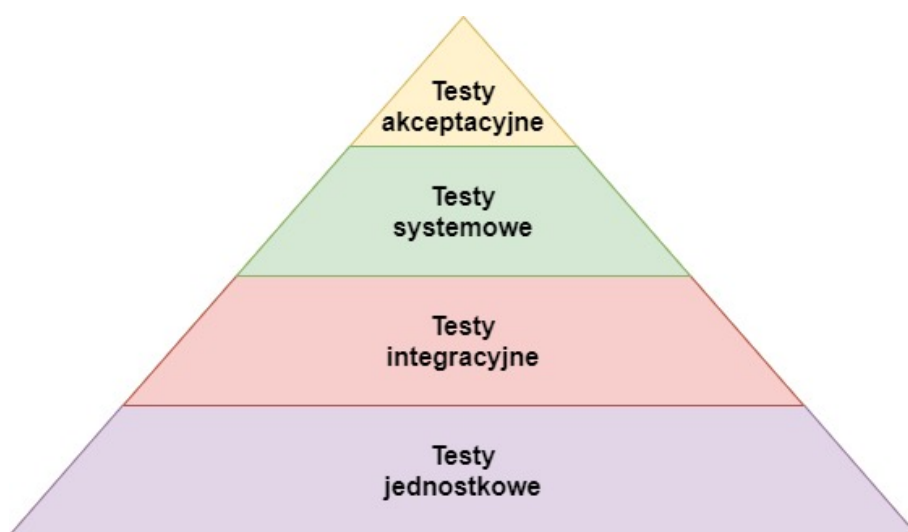
Testowanie jest integralną częścią procesu wytwarzania oprogramowania. **Testy oprogramowania** umożliwiają sprawdzenie czy zaimplementowana aplikacja jest zgodna z założeniami projektowymi (**weryfikacja**, statyczne testowanie kodu) oraz czy działa we właściwy sposób (**walidacja**, dynamiczne testowanie kodu). Możliwe jest wykrycie błędów oprogramowania nawet na wczesnym etapie jego realizacji. Dzięki temu koszty naprawy oprogramowania są niższe niż w przypadku wykrycia błędu już po wydaniu aplikacji. Należy mieć świadomość, że **testowanie** nie gwarantuje wykrycia wszystkich błędów oprogramowania, ale może przyczynić się do znaczącego zmniejszenia ich liczby. Ze względu na sposób wykonywania testów wyróżnia się **testy manualne** oraz **testy automatyczne**.

Można wyróżnić kilka poziomów testów oprogramowania (rys. 2.1) [3]:

- **testy jednostkowe** (*ang. unit tests*) – stanowią najniższy poziom **testów oprogramowania**. Ich założeniem jest testowanie odseparowanego komponentu systemu (funkcji, modułu) przez programistów;
- **testy integracyjne** (*ang. integration tests*) – zakładają testowanie grupy współpracujących ze sobą modułów;
- **testy systemowe** (*ang. system tests*) – to testy pełnego, zintegrowanego systemu informatycznego z wykorzystaniem odpowiedniego środo-

wiska testowego, przeprowadzane przez przeszkolonych testerów, w celu porównania jego działania z założeniami projektowymi;

- **testy akceptacyjne** (*ang. acceptance tests*) – stanowią najwyższy poziom **testów oprogramowania**. Polegają na testowaniu działającego systemu przez użytkowników końcowych w środowisku zapewnionym przez programistów (*testy alfa*) lub użytkownika-klienta (*testy beta*).



Rys. 2.1. Poziomy testów oprogramowania

Testy oprogramowania (w szczególności **testy jednostkowe** i **integracyjne**) mogą stanowić tzw. **testy regresywne** (*ang. regression tests*). Są to testy walidujące działanie oprogramowania po wprowadzeniu zmian (w kodzie lub środowisku uruchomieniowym aplikacji). Dzięki temu możliwe jest stwierdzenie czy poszczególne zmiany w oprogramowaniu spowodowały powstanie nowych błędów (regresję), przez co działanie systemu nie jest zgodne z określonymi wymaganiami.

Testy oprogramowania można również podzielić na dwie grupy ze względu na testowaną warstwę systemu:

- **testy czarnoskrzynkowe** – nie uwzględniają wewnętrznej struktury (implementacji) modułu; stanowią testy interfejsu;

-
- **testy białoskrzynkowe** – uwzględniają wewnętrzną strukturę modułu lub systemu, co rozumie się przez implementację (kod), architekturę oraz przepływy sterowania w aplikacji. Stopień realizacji **testowania białoskrzynkowego** mierzy się za pomocą tzw. **stopnia pokrycia strukturalnego**, czyli liczby instrukcji wykonanych w ramach testów do liczby wszystkich instrukcji zawartych w kodzie modułu lub systemu [4].

2.2. Test–Driven Development

Zasadniczo wyróżnia się dwa odmienne podejścia do implementacji **testów oprogramowania**. W pierwszym podejściu testy pisane są w kolejnym etapie po implementacji modułu lub systemu. Jest to rozwiązanie stosowane najczęściej w przypadku konieczności szybkiego wytworzenia oprogramowania, gdy dokładne, całościowe testowanie schodzi na drugi plan (przed dostarczeniem zmian w kodzie programista przeprowadza testy manualne). Drugie podejście zakłada implementację testów jeszcze przed realizacją kodu aplikacji. Metodyka ta znana jest pod nazwą **Test–Driven Development, TDD** (*programowanie sterowane testami*), a jej autorstwo przypisywane jest K. Beckowi. **TDD** może zostać opisane w kilku (wielokrotnie powtarzanych) krokach:

1. **Implementacja testu** – programista pisze test automatyczny (jednostkowy, integracyjny), który ma na celu sprawdzenie nowej funkcjonalności. W tym celu **nie jest potrzebna znajomość konkretnej implementacji** modułu. Nowy test powinien bazować na **interfejsie programistycznym** oraz dokumentacji (założeniach) projektowych
2. **Uruchomienie testów** – nowo dodany test nie powinien się udać, co świadczy o jego zdolności do wykrycia błędów aplikacji (na tym etapie może się nawet nie kompilować)
3. **Implementacja funkcjonalności** – wytworzenie roboczej wersji oprogramowania, która powinna spełniać założenia sprawdzane w teście

-
4. **Uruchomienie testów** – wszystkie testy powinny się udać (pozytywna walidacja kodu)
 5. **Refaktoryzacja kodu** – dostosowanie implementacji do standardów projektowych oraz optymalizacja rozwiązania

2.3. Google Test framework

Na rynku dostępnych jest wiele platform programistycznych ułatwiających tworzenie testów oprogramowania na różnych poziomach. Jedną z najpopularniejszych jest **Google Test**, czyli **framework testowy** języka *C++*, opracowany przez firmę *Google* do implementacji **testów jednostkowych/integracyjnych**. **Google Test** jest darmową, sprawdzoną, wykorzystywaną komercyjnie i prostą w użyciu platformą. Dokumentacja **frameworku testowego** dostępna jest pod adresem <https://github.com/google/googletest>. **Google Test** wymaga kompilatora wspierającego co najmniej standard *C++11*.

2.3.1. Instalacja i konsolidacja

Aby skompilować **Google Test framework** należy wcześniej zainstalować **system budowania CMake**. Oprogramowanie można pobrać ze strony <https://cmake.org/download/>. Najwygodniej jest pobrać plik instalatora dla konkretnego systemu operacyjnego (*Binary distributions*). W przypadku instalacji na systemie **Windows** należy pamiętać o dodaniu **CMake** do zmiennej systemowej **PATH** (opcja instalatora: *Add CMake to the system PATH*; może być konieczne ponowne uruchomienie komputera). Po zakończonej instalacji wykonanie polecenia **cmake --version** w konsoli systemowej powinno skutkować wypisaniem wersji zainstalowanego oprogramowania (np. *cmake version 3.6.2*).

Spakowany projekt **Google Test** dostępny jest pod adresem <https://github.com/google/googletest> (zakładka *Code*). [Uwaga: Pracując w systemie *Windows* nie należy docelowo wypakowywać projektu *Google Test* na

Pulpit!] Wewnątrz katalogu zawierającego plik **CMakeLists.txt** (*googletest-main*) należy utworzyć nowy folder (np. *build*), w którym znajdzie się skompilowany **framework testowy**. Za pomocą konsoli systemowej, **z poziomu nowo utworzonego folderu**, należy wywołać polecenie:

cmake ..

Polecenie **cmake** powoduje przetworzenie przez system budowania pliku **CMakeLists.txt** znajdującego się we wskazanym katalogu. Podwójna kropka (..) określa folder nadrzędny względem aktualnego (tu: *googletest-main*). [Uwaga: Bezwzględna ścieżka do katalogu ze skompilowanym frameworkiem testowym (tu: *build*) nie może zawierać spacji!] W wyniku tej operacji powstaje plik **Makefile**. Po wywołaniu polecenia:

make

zbudowane zostaje archiwum **libgtest.a** (w katalogu *lib*). Aby otrzymać bibliotekę dynamiczną należy wywołać polecenie:

cmake -D BUILD_SHARED_LIBS=ON ..

Jeżeli wywołanie **cmake ..** z poziomu katalogu *build* kończy się błędem:

CMake Error at CMakeLists.txt:10 (project):

The CMAKE_C_COMPILER: cl is not a full path and was not found in the PATH

oznacza to, że konfiguracja **cmake** zawiera nieprawidłowy generator, przez co system budowania nie może zlokalizować kompilatorów **gcc** i **g++**:

- **The C compiler identification is unknown**
- **The CXX compiler identification is unknown**

Aktualnie wybrany generator jest widoczny w wygenerowanym (wewnątrz katalogu *build*) pliku **CMakeCache.txt**:

//Name of generator.

CMAKE_GENERATOR:INTERNAL=Unix Makefiles

Jeżeli ustawiony generator jest inny niż *Unix Makefiles*, należy go zmienić, aby *cmake* poprawnie zlokalizował kompilatory pakietu *MinGW*. Nie należy robić tego bezpośrednio w pliku **CMakeCache.txt** (jest to plik generowany automatycznie)! W tym celu należy:

1. usunąć wszystkie pliki i katalogi wygenerowane przez *cmake* z katalogu *build* (**CMakeCache.txt** i **CMakeFiles**);
2. w konsoli systemowej z poziomu katalogu *build* wywołać polecenie:

cmake -G "Unix Makefiles" ..

Omówione błędy związane z niewłaściwym generatorem nie powinny teraz wystąpić i powinien zostać wygenerowany plik **Makefile**;

3. w konsoli systemowej z poziomu katalogu *build* wywołać polecenie:

make

Konsolidację archiwum *libgtest.a* można przeprowadzić następująco:

g++ main.cpp libgtest.a -o app

Uwaga: Kompilator g++ musi wspierać co najmniej standard C++11 (od wersji 4.8.1)

Podczas budowania aplikacji testowej można napotkać problemy z lokalizacją pliku *libgtest.a* i/lub pliku nagłówkowego *gtest.h*. Aby tego uniknąć można zbudować aplikację testową posługując się następującymi flagami:

```
g++  
-I[względna_ścieżka_do_katalogu_include_frameworku_googletest]  
-L[względna_ścieżka_do_katalogu_zawierającego_plik_biblioteki]  
[pliki_źródłowe_projektu]  
-lgtest
```

Przykładowe wywołanie może wyglądać następująco:

```
g++ -Igoogletest-main\googletest-main\googletest\include  
-Lgoogletest-main\googletest-main\build\lib main.cpp -lgtest
```

Należy zachować podaną kolejność wywołań flag **-I**, **-L**, **-l** oraz plików źródłowych. Flaga **-I** informuje preprocesor gdzie znajdują się zewnętrzne pliki nagłówkowe (tu: *gtest/gtest.h*). Flaga **-L** informuje konsolidator gdzie znajduje się zbudowana biblioteka (tu: *libgtest.a*). Flaga **-l** podaje konsolidatorowi nazwę załączanej biblioteki (bez przedrostka *lib* i rozszerzenia *.a*).

2.3.2. Implementacja testów jednostkowych

Testy jednostkowe nie stanowią integralnej części z testowaną aplikacją. Ich uruchomienie wymaga utworzenia samodzielnej aplikacji testowej (osobna funkcja *main()*), która niezależnie wywoła walidowane funkcje w określonym kontekście. Typową funkcję *main()*, której zadaniem jest uruchomienie wszystkich zaimplementowanych **testów jednostkowych**, przedstawiono na listingu 1. Funkcja *InitGoogleTest()* z przestrzeni nazw **testing** inicjalizuje framework testowy. Dzięki temu możliwe jest sterowanie uruchamianymi **testami** za pomocą argumentów wywołania programu. Przykładowe flagi sterujące zestawiono w tabeli 1. Makro **RUN_ALL_TESTS** uruchamia wszystkie skompilowane testy i zwraca wartość **0**, jeśli wszystkie testy wykonały się prawidłowo, albo **1** w przeciwnym wypadku. Nagłówek *gtest.h* można odnaleźć pod ścieżką `\googletest-main\googletest\include\gtest\`.

```

1 #include "gtest/gtest.h"
2
3 int main(int argc, char **argv) {
4     testing::InitGoogleTest(&argc, argv);
5     return RUN_ALL_TESTS();
6 }

```

Listing 1. Funkcja *main()* aplikacji testowej

Tabela 1. Przykładowe flagi sterujące frameworkiem *Google Test* [1]

Flaga	Opis
--gtest_list_tests	wypisz wszystkie testy
--gtest_filter	uruchom wybrane testy (nazwa w formacie TestSuiteName.TestName)
--gtest_fail_fast	zakończ działanie po pierwszym nieudanym teście
--gtest_repeat	powtórz testy określoną liczbę razy
--gtest_shuffle	uruchom testy w losowej kolejności

Na rys. 2.2. przedstawiono komunikat, jaki otrzyma się po uruchomieniu aplikacji testowej. Framework informuje ile testów zostało uruchomione, jak długo trwało ich wykonanie oraz ile z nich zakończyło się powodzeniem.

```

[=====] Running 0 tests from 0 test suites.
[=====] 0 tests from 0 test suites ran. (2 ms total)
[ PASSED ] 0 tests.

```

Rys. 2.2. Uruchomienie aplikacji testowej z listingu 1.

Pojedynczy *test* w ujęciu frameworku *Google Test* stanowi osobną funkcję (niezwracającą żadnej wartości), zadeklarowaną z wykorzystaniem makra *TEST*, jak przedstawiono na listingu 2. Makro *TEST* przyjmuje dwa

argumenty: pierwszy stanowi nazwę *pakietu testów* (*ang. test suite*), natomiast drugi nazwę konkretnego *testu* (*ang. test case*). **Każdy test powinien sprawdzać pojedynczy przypadek testowy**, np. testując funkcję *divide()*, przeprowadzającą dzielenie dwóch liczb, należy utworzyć dwa niezależne testy – jeden sprawdzający działanie funkcji w przypadku dzielenia przez liczbę niezerową; drugi testujący dzielenie przez zero. **Pakiety** służą do grupowania testów odnoszących się do tej samej funkcjonalności. Wygodnie jest umieszczać testy należące do tego samego pakietu w obrębie oddzielnej jednostki kompilacji (pliku źródłowego).

```
1 TEST(TestSuiteName , TestName) {  
2     // Cialo funkcji (testu)  
3 }
```

Listing 2. Struktura testu w ujęciu frameworku *Google Test*

Dobrym nawykiem jest zawieranie w nazwie testu oczekiwanego zachowania. Wówczas nietrudno jest domyślić się intencji programisty, która przyświecała mu przy pisaniu określonego przypadku testowego, bez konieczności szczegółowej analizy ciała (kodu) testu. Przykład dla wywołania funkcji *divide()* z dzielnikiem równym zero przedstawiono na listingu 3. Oczekiwany rezultatem jest zakończenie działania programu z odpowiednim kodem błędu.

```
1 TEST(DivisionTest ,  
2     ShouldQuitWithErrorWhenDividedByZero) {  
3     // Cialo funkcji (testu)  
4 }
```

Listing 3. Dobra praktyka nazewnictwa testów jednostkowych

Podstawowym elementem testów są **asercje** (*ang. assertions*). Stanowią one makra sprawdzające czy zadany warunek jest prawdziwy. **Google Test** wyróżnia dwa rodzaje asercji:

-
- ***ASSERT_**** – w przypadku niepowodzenia przerywa dalsze wykonywanie testu. Powinna być stosowana, gdy wykonywanie kolejnych instrukcji mogłoby doprowadzić np. do niezdefiniowanego zachowania (szczególnie istotne w przypadku wskaźników). Należy mieć na uwadze, że nieumiejętne posługiwanie się makrem ***ASSERT_**** może prowadzić do wycieków zasobów, wynikających z przerywania testu przed przeprowadzeniem operacji sprzątania zasobów (zwalnianie pamięci, zamykanie plików, itp.);
 - ***EXPECT_**** – w przypadku niepowodzenia kontynuuje dany test. Informacja o błędzie zostanie umieszczona w zbiorowym podsumowaniu wykonanych testów (przeważnie preferowane zachowanie).

Każda asercja ***ASSERT_**** posiada swój odpowiednik w postaci ***EXPECT_****. W tabeli 2. zestawiono wybrane asercje frameworku ***Google Test*** wraz z objaśnieniami użycia.

Wykorzystanie asercji w testach jednostkowych zostanie omówione na przykładzie funkcji ***power()***, wyznaczającej zadaną potęgę liczby. Deklaracja funkcji ***power()*** została umieszczona w nagłówku ***power.h***, natomiast jej definicja (przedstawiona na listingu 4.) w pliku źródłowym ***power.cpp***. Przedstawiona implementacja celowo zawiera błąd, który mają za zadanie wykryć ***testy jednostkowe***.

Tabela 2. Wybrane asercje frameworku *Google Test* [2][1]

Asercja	Sprawdzenie
ASSERT_TRUE(condition)	$condition == true$
ASSERT_FALSE(condition)	$condition == false$
ASSERT_EQ(val1, val2)	$val1 == val2$
ASSERT_NE(val1, val2)	$val1 != val2$
ASSERT_LT(val1, val2)	$val1 < val2$
ASSERT_LE(val1, val2)	$val1 \leq val2$
ASSERT_GT(val1, val2)	$val1 > val2$
ASSERT_GE(val1, val2)	$val1 \geq val2$
ASSERT_FLOAT_EQ(val1, val2)	$val1 == val2$, dla zmiennych typu <i>float</i> (dopuszczalny błąd w granicach 4 ULP ¹)
ASSERT_DOUBLE_EQ(val1, val2)	$val1 == val2$, dla zmiennych typu <i>double</i> (dopuszczalny błąd w granicach 4 ULP)
ASSERT_NEAR(val1, val2, error)	$val1 == val2$, dla liczb zmiennoprzecinkowych (dopuszczalny błąd bezwzględny <i>error</i>)
ASSERT_STREQ(str1, str2)	łańcuchy <i>str1</i> i <i>str2</i> mają tę samą zawartość
ASSERT_STRNE(str1, str2)	łańcuchy <i>str1</i> i <i>str2</i> mają różną zawartość
ASSERT_STRCASEEQ(str1, str2)	łańcuchy <i>str1</i> i <i>str2</i> mają tę samą zawartość (asercja niewrażliwa na wielkość znaków)
ASSERT_STRCASENE(str1, str2)	łańcuchy <i>str1</i> i <i>str2</i> mają różną zawartość (asercja niewrażliwa na wielkość znaków)
ASSERT_EXIT(statement, predicate, str)	wyrażenie <i>statement</i> zakończy działanie aplikacji z kodem błędu spełniającym warunek <i>predicate</i> , wpisując łańcuch <i>str</i> na <i>standardowe wyjście dla błędów</i>

¹Jednostek na ostatnim miejscu (*ang. units in the last place*)

```

1 #include "power.h"
2
3 int power(int base, unsigned int exponent) {
4     // Bledna wartosc poczatkowa
5     int result = 0;
6     while (exponent--)
7         result *= base;
8     return result;
9 }

```

Listing 4. Błędna implementacja funkcji *power()*

Testy funkcji *power()* (listing 5.) zostały umieszczone w pliku źródłowym *powerTests.cpp*. Do ich implementacji wykorzystano asercję *EXPECT_EQ*. [Uwaga: przypadki testowe zerowego i niezerowego wykładnika celowo nie zostały rozdzielone na dwa osobne testy w celu zaprezentowania różnicy między makrami *EXPECT_EQ* a *ASSERT_EQ*]

```

1 #include "gtest/gtest.h"
2 #include "power.h"
3
4 TEST(PowerTest, DifferentExponents) {
5     // Zerowy wykladnik
6     EXPECT_EQ(power(2, 0), 1);
7     // Niezerowy wykladnik
8     EXPECT_EQ(power(5, 3), 125);
9 }

```

Listing 5. Przykład zastosowania asercji w testach jednostkowych

Kompilacja aplikacji testowej została przeprowadzona z użyciem polecenia:

```
g++ main.cpp power.cpp powerTests.cpp libgtest.a -o testApp
```

Po uruchomieniu aplikacji zostaje wyświetlony komunikat, jak na rys. 2.3. Zawiera on informację, że uruchomiony test nie został wykonany pomyślnie (*FAILED*), ze szczegółowym uwzględnieniem wyników poszczególnych asercji, np. w siódmej linii pliku *powerTests.cpp* oczekiwano, że rezultatem wywołania funkcji *power(2, 0)* będzie wynik **1**, natomiast otrzymanym wynikiem jest **0**.

```
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from PowerTest
[ RUN    ] PowerTest.DifferentExponents
powerTests.cpp:7: Failure
Expected equality of these values:
  power(2, 0)
    which is: 0
    1
powerTests.cpp:9: Failure
Expected equality of these values:
  power(5, 3)
    which is: 0
    125
[  FAILED  ] PowerTest.DifferentExponents (8 ms)
[-----] 1 test from PowerTest (8 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (14 ms total)
[ PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] PowerTest.DifferentExponents

1 FAILED TEST
```

Rys. 2.3. Podsumowanie wykonania testów jednostkowych błędnej implementacji funkcji *power()* z użyciem asercji *EXPECT_EQ*

Zmiana asercji *EXPECT_EQ* na *ASSERT_EQ* skutkuje przerwaniem wykonywania testu na linii siódmej pliku *powerTests.cpp* – pierwszy wykryty błąd (rys. 2.4). Nie zostanie przeprowadzone sprawdzenie funkcji potęgującej dla niezerowego wykładnika.

```

[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from PowerTest
[ RUN     ] PowerTest.DifferentExponents
powerTests.cpp:5: Failure
Expected equality of these values:
  power(2, 0)
    which is: 0
    1
[  FAILED ] PowerTest.DifferentExponents (4 ms)
[-----] 1 test from PowerTest (11 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (19 ms total)
[ PASSED ] 0 tests.
[  FAILED ] 1 test, listed below:
[  FAILED ] PowerTest.DifferentExponents

1 FAILED TEST

```

Rys. 2.4. Podsumowanie wykonania testów jednostkowych błędnej implementacji funkcji *power()* z użyciem asercji *ASSERT_EQ*

Wykorzystując przeciążony operator przesunięcia bitowego w lewo (*operator«*) możliwe jest przekazywanie do asercji własnych komunikatów, które mają zostać wypisane na ekranie w przypadku wykrycia błędu. Przykład takiego rozwiązania przedstawiono na listingu 6., natomiast podsumowanie wykonanych testów na rys. 2.5.

```

1 #include "gtest/gtest.h"
2 #include "power.h"
3
4 TEST(PowerTest, DifferentExponents) {
5     ASSERT_EQ(power(2, 0), 1) << "Failed to calculate
6         power with zero exponent";
7     ASSERT_EQ(power(5, 3), 125) << "Failed to
8         calculate power with nonzero exponent";
9 }

```

Listing 6. Rejestrowanie własnych komunikatów błędów w asercjach


```

[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from PowerTest
[ RUN     ] PowerTest.DifferentExponents
powerTests.cpp:5: Failure
Expected equality of these values:
  power(2, 0)
    Which is: 0
  1
Failed to calculate power with zero exponent
[  FAILED ] PowerTest.DifferentExponents (7 ms)
[-----] 1 test from PowerTest (9 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (18 ms total)
[ PASSED ] 0 tests.
[  FAILED ] 1 test, listed below:
[  FAILED ] PowerTest.DifferentExponents

1 FAILED TEST

```

Rys. 2.5. Podsumowanie wykonania testów jednostkowych błędnej implementacji funkcji *power()* z uwzględnieniem własnych komunikatów błędów

Podsumowanie wykonania testów jednostkowych po skorygowaniu implementacji funkcji *power()*, jak na listingu 7., zostało przedstawione na rys. 2.6.

```

1 #include "power.h"
2
3 int power(int base, unsigned int exponent) {
4     int result = 1;
5     while (exponent--)
6         result *= base;
7     return result;
8 }

```

Listing 7. Poprawna implementacja funkcji *power()*

```

[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from PowerTest
[ RUN     ] PowerTest.DifferentExponents
[ OK      ] PowerTest.DifferentExponents (0 ms)
[-----] 1 test from PowerTest (2 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (7 ms total)
[ PASSED ] 1 test.

```

Rys. 2.6. Podsumowanie wykonania testów jednostkowych poprawnej implementacji funkcji *power()*

2.3.3. Testowanie wywołań zwrotnych

Na listingu 8. przedstawiono zawartość pliku nagłówkowego *dispatcher.h*, zawierającego deklaracje funkcji *registerCallback()*, *deregisterCallback()* oraz *dispatch()*. Zadaniem funkcji *dispatch()* jest wykonanie operacji arytmetycznej na dwóch zmiennych typu *double*, zmapowanej na typ wyliczeniowy *Operation*. Operacja, aby została przeprowadzona, musi zostać wcześniej zarejestrowana za pomocą funkcji *registerCallback()* w postaci wskaźnika funkcyjnego *Callback*. Przekazany wskaźnik funkcyjny można wyrejestrować za pomocą funkcji *deregisterCallback()*.

```

1 #pragma once
2
3 // Deskryptory operacji arytmetycznych
4 enum class Operation : unsigned int {
5     ADD = 0,
6     SUBTRACT,
7     MULTIPLY,
8     DIVIDE
9 };
10
11 // Wskaznik funkcyjny operacji arytmetycznej

```

```

12 using Callback = double (*)(double, double);
13
14 // Rejestracja wywołania zwrotnego
15 void registerCallback(Operation, const Callback);
16 // Wyrejestrowanie wywołania zwrotnego
17 void deregisterCallback(Operation);
18 // Wywołanie zarejestrowanej operacji arytmetycznej
19 double dispatch(Operation, double, double);

```

Listing 8. Zawartość pliku nagłówkowego *dispatcher.h*

Implementację funkcji *registerCallback()* i *dispatch()* przedstawiono na listingu 9. (plik *dispatcher.cpp*). Komponent wykorzystuje statyczną tablicę struktur *DispatcherEntry*, mapującą deskryptory operacji arytmetycznych (*Operation*) na odpowiadające im wskaźniki funkcyjne (*Callback*), które mają zostać wywołane przez funkcję *dispatch()*. Początkowo tablica *DISPATCHER_TABLE* inicjalizowana jest z wykorzystaniem domyślnej funkcji obsługi operacji arytmetycznych *nullOperationHandler*, której wywołanie nie powoduje żadnych *skutków ubocznych* (ang. *side effects*). Rejestracja operacji arytmetycznej jest realizowana przez umieszczenie wypełnionej struktury *DispatcherEntry* pod odpowiednim indeksem tablicy *DISPATCHER_TABLE*. Warto zwrócić uwagę, że wartości liczbowe enumeratorów odpowiadają kolejnym indeksom tablicy, co wykorzystane zostało w implementacji funkcji *registerCallback()* i *deregisterCallback()*. Przekazanie do funkcji *dispatch()* enumeratora leżącego poza zdefiniowanym zakresem dla *Operation* (listing 8.) skutkuje *niezdefiniowanym zachowaniem*.

```

1 #include "dispatcher.h"
2
3 // Struktura mapujaca operacje arytmetyczne na
  wywołania zwrotne

```

```
4 struct DispatcherEntry {
5     Operation operation;
6     Callback callback;
7 };
8
9 // Rownowazne z uzyciem specyfikatora static
10 namespace {
11
12 // Domyslna funkcja obslugi operacji arytmetycznych
13 // (bez efektow ubocznych)
14 double nullOperationHandler(double, double) { return
15     0.0; }
16
17 // Tablica struktur DispatcherEntry zainicjalizowana
18 // domyslnymi funkcjami obslugi operacji
19 // arytmetycznych
20 DispatcherEntry DISPATCHER_TABLE[] = {
21     {Operation::ADD, nullOperationHandler},
22     {Operation::SUBTRACT, nullOperationHandler},
23     {Operation::MULTIPLY, nullOperationHandler},
24     {Operation::DIVIDE, nullOperationHandler}
25 };
26
27 } // namespace
28
29 void registerCallback(Operation operation, const
30     Callback callback) {
31     if (callback)
32         // Wartosci liczbowe enumeratorow odpowiadaja
33         // kolejnym indeksom DISPATCHER_TABLE
```

```

28     DISPATCHER_TABLE[static_cast<unsigned int>(
    operation)] = {operation, callback};
29 }
30
31 void deregisterCallback(Operation operation) {
32     // Ponowne przypisanie domyslniej funkcji obslugi
    do deskryptora
33     DISPATCHER_TABLE[static_cast<unsigned int>(
    operation)] = {operation, nullOperationHandler};
34 }
35
36 double dispatch(Operation operation, double x,
    double y) {
37     // Petla po wszystkich elementach DISPATCHER_TABLE
38     for (unsigned int i = 0u; i < sizeof(
    DISPATCHER_TABLE) / sizeof(DispatcherEntry); ++i)
    {
39         if (DISPATCHER_TABLE[i].operation == operation)
    {
40             // Wywołanie zwrotne
41             return DISPATCHER_TABLE[i].callback(x, y);
42         }
43     }
44     return 0.0;
45 }

```

Listing 9. Zawartość pliku źródłowego *dispatcher.cpp*

Testy jednostkowe komponentu *dispatcher*, znajdujące się w pliku źródłowym *dispatcherTests.cpp*, przedstawiono na listingu 10. Plik zawiera cztery funkcje (*add()*, *subtract()*, *multiply()*, *divide()*), rejestrowane za pomocą funkcji *registerCallback()*. Funkcja *divide()*, w przypadku dzielenia przez zero, wypisuje na *standardowe wyjście dla błędów* informację o niedozwolonej operacji i kończy działanie programu z kodem błędu *EXIT_FAILURE*. Testy zostały pogrupowane w obrębie dwóch *pakietów*: *DispatcherTest* oraz *DispatcherDeathTest*. *Death test* jest nazwą zalecaną przez *Google Test framework* dla testów sprawdzających błędne zakończenie działania aplikacji (*exit(EXIT_FAILURE)*, obsługa sygnałów systemowych, itp.) [1]. Całość umieszczona jest wewnątrz *anonimowej przestrzeni nazw* w celu ograniczenia widoczności użytych identyfikatorów funkcji w obrębie jednostki kompilacji testów.

W przypadku testu *DispatcherDeathTest.DivideByZero* wykorzystano asercję *EXPECT_EXIT* w celu sprawdzenia wywołania funkcji *divide()* dla dzielenia przez zero. Porównanie zwracanego kodu błędu zostało zrealizowane z wykorzystaniem funkcji *ExitedWithCode()* z przestrzeni nazw *testing frameworku Google Test*.

```
1 #include <iostream>
2
3 #include "dispatcher.h"
4 #include "gtest/gtest.h"
5
6 // Uzyte identyfikatory funkcji sa unikalne w
   obrebie jednostki kompilacji
7 namespace {
8
9 double add(double x, double y) {
10     return x + y;
11 }
```

```
12
13 double subtract(double x, double y) {
14     return x - y;
15 }
16
17 double multiply(double x, double y) {
18     return x * y;
19 }
20
21 double divide(double x, double y) {
22     if (y == 0) {
23         std::cerr << "Division by zero";
24         exit(EXIT_FAILURE);
25     }
26     return x / y;
27 }
28
29 // Wyrejestrowanie niezarejestrowanej operacji
30 TEST(DispatcherTest, DeregisterUnregisteredCallback)
31 {
32     deregisterCallback(Operation::ADD);
33 }
34
35 // Wykonanie niezarejestrowanej operacji na
36 // niezmodyfikowanej tablicy
37 TEST(DispatcherTest,
38     CallUnregisteredCallbackUnmodifiedTable) {
39     // Oczekiwane wywołanie nullOperationHandler()
40     EXPECT_DOUBLE_EQ(dispatch(Operation::ADD, 1.2,
41         -0.7), 0.0);
42 }
```

```
39
40 // Rejestracja i uruchomienie wywołania zwrotnego
41 TEST(DispatcherTest, CallRegisteredCallback) {
42     registerCallback(Operation::ADD, add);
43     EXPECT_DOUBLE_EQ(dispatch(Operation::ADD, 2.0,
44                               3.5), 5.5);
44     deregisterCallback(Operation::ADD);
45     // Oczekiwane wywołanie nullOperationHandler()
46     EXPECT_DOUBLE_EQ(dispatch(Operation::ADD, 2.0,
47                               3.5), 0.0);
47 }
48
49 // Rejestracja operacji i wywołanie operacji
   niezarejestrowanej
50 TEST(DispatcherTest,
51       CallUnregisteredCallbackModifiedTable) {
52     registerCallback(Operation::ADD, add);
53     // Oczekiwane wywołanie nullOperationHandler()
54     EXPECT_DOUBLE_EQ(dispatch(Operation::SUBTRACT,
55                               0.5, 6.7), 0.0);
56     deregisterCallback(Operation::ADD);
57 }
58
59 // Rejestracja wszystkich operacji
60 TEST(DispatcherTest, AllCallbacksRegistered) {
61     registerCallback(Operation::ADD, add);
62     registerCallback(Operation::SUBTRACT, subtract);
63     registerCallback(Operation::MULTIPLY, multiply);
64     registerCallback(Operation::DIVIDE, divide);
65     EXPECT_DOUBLE_EQ(dispatch(Operation::ADD, 3.4,
66                               4.6), 8.0);
```



```

64 EXPECT_DOUBLE_EQ(dispatch(Operation::SUBTRACT,
    6.5, -2.1), 8.6);
65 EXPECT_DOUBLE_EQ(dispatch(Operation::MULTIPLY,
    3.0, 1.1), 3.3);
66 EXPECT_DOUBLE_EQ(dispatch(Operation::DIVIDE, 16.0,
    -4.0), -4.0);
67 deregisterCallback(Operation::ADD);
68 deregisterCallback(Operation::SUBTRACT);
69 deregisterCallback(Operation::MULTIPLY);
70 deregisterCallback(Operation::DIVIDE);
71 }
72
73 // Rejestracja nullptr
74 TEST(DispatcherTest, RegisterNullptr) {
75     registerCallback(Operation::ADD, nullptr);
76     // Oczekiwane wywołanie nullOperationHandler()
77     EXPECT_DOUBLE_EQ(dispatch(Operation::ADD, 11.3,
    0.42), 0.0);
78     deregisterCallback(Operation::ADD);
79 }
80
81 // Dzielenie przez zero
82 TEST(DispatcherDeathTest, DivideByZero) {
83     registerCallback(Operation::DIVIDE, divide);
84     EXPECT_EXIT(dispatch(Operation::DIVIDE, 2.5, 0.0),
85         ::testing::ExitedWithCode(EXIT_FAILURE), "
    Division by zero");
86 }
87
88 } // namespace

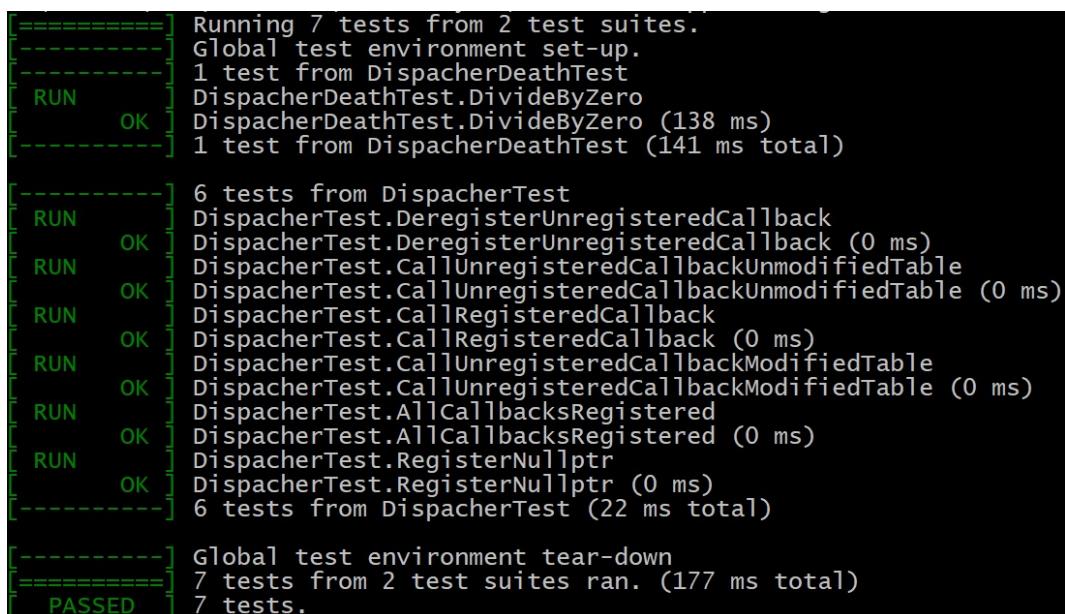
```

Listing 10. Zawartość pliku źródłowego *dispatcherTests.cpp*

Kompilację aplikacji testowej przeprowadzono z użyciem polecenia:

```
g++ main.cpp dispatcher.cpp dispatcherTests.cpp libgtest.a -o
testApp
```

Podsumowanie wykonania testów komponentu *dispatcher* przedstawiono na rys. 2.7.



```
===== Running 7 tests from 2 test suites.
----- Global test environment set-up.
----- 1 test from DispatcherDeathTest
RUN      DispatcherDeathTest.DivideByZero
OK       DispatcherDeathTest.DivideByZero (138 ms)
----- 1 test from DispatcherDeathTest (141 ms total)

----- 6 tests from DispatcherTest
RUN      DispatcherTest.DeregisterUnregisteredCallback
OK       DispatcherTest.DeregisterUnregisteredCallback (0 ms)
RUN      DispatcherTest.CallUnregisteredCallbackUnmodifiedTable
OK       DispatcherTest.CallUnregisteredCallbackUnmodifiedTable (0 ms)
RUN      DispatcherTest.CallRegisteredCallback
OK       DispatcherTest.CallRegisteredCallback (0 ms)
RUN      DispatcherTest.CallUnregisteredCallbackModifiedTable
OK       DispatcherTest.CallUnregisteredCallbackModifiedTable (0 ms)
RUN      DispatcherTest.AllCallbacksRegistered
OK       DispatcherTest.AllCallbacksRegistered (0 ms)
RUN      DispatcherTest.RegisterNullptr
OK       DispatcherTest.RegisterNullptr (0 ms)
----- 6 tests from DispatcherTest (22 ms total)

----- Global test environment tear-down
===== 7 tests from 2 test suites ran. (177 ms total)
PASSED  7 tests.
```

Rys. 2.7. Podsumowanie wykonania testów jednostkowych komponentu *dispatcher*

Pisane **testy jednostkowe** powinny być (w miarę możliwości) od siebie niezależne, tzn. kolejność ich wykonania nie powinna mieć znaczenia. W celu uruchomienia testów w losowej kolejności można posłużyć się flagą sterującą **--gtest_shuffle** (tabela 1). Flaga przekazywana jest jako argument wywołania programu:

```
testApp.exe --gtest_shuffle
```

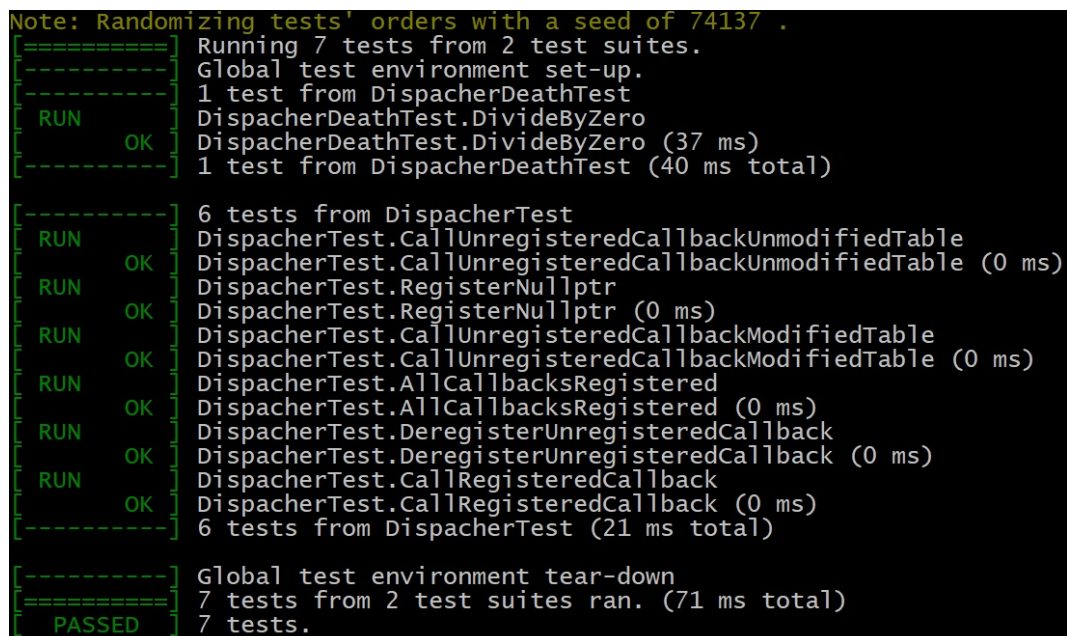
Przykładowe podsumowanie tak uruchomionej aplikacji testowej przedstawiono na rys. 2.8. Inną popularną flagą jest **--gtest_repeat**, która znajduje zastosowanie szczególnie w testach funkcji uwzględniających element

losowości. Wielokrotne powtórzenie takiego testu zwiększa prawdopodobieństwo wykrycia błędu. Składania uruchomienia aplikacji testowej przy założeniu tysiąckrotnego powtórzenia każdego testu jest następująca:

```
testApp.exe --gtest_repeat=1000
```

Flagi sterujące *frameworkiem Google Test* można łączyć, np.

```
testApp.exe --gtest_shuffle --gtest_repeat=250
```



```
Note: Randomizing tests' orders with a seed of 74137 .
[=====] Running 7 tests from 2 test suites.
[-----] Global test environment set-up.
[-----] 1 test from DispatcherDeathTest
[ RUN     ] DispatcherDeathTest.DivideByZero
[ OK      ] DispatcherDeathTest.DivideByZero (37 ms)
[-----] 1 test from DispatcherDeathTest (40 ms total)

[-----] 6 tests from DispatcherTest
[ RUN     ] DispatcherTest.CallUnregisteredCallbackUnmodifiedTable
[ OK      ] DispatcherTest.CallUnregisteredCallbackUnmodifiedTable (0 ms)
[ RUN     ] DispatcherTest.RegisterNullptr
[ OK      ] DispatcherTest.RegisterNullptr (0 ms)
[ RUN     ] DispatcherTest.CallUnregisteredCallbackModifiedTable
[ OK      ] DispatcherTest.CallUnregisteredCallbackModifiedTable (0 ms)
[ RUN     ] DispatcherTest.AllCallbacksRegistered
[ OK      ] DispatcherTest.AllCallbacksRegistered (0 ms)
[ RUN     ] DispatcherTest.DeregisterUnregisteredCallback
[ OK      ] DispatcherTest.DeregisterUnregisteredCallback (0 ms)
[ RUN     ] DispatcherTest.CallRegisteredCallback
[ OK      ] DispatcherTest.CallRegisteredCallback (0 ms)
[-----] 6 tests from DispatcherTest (21 ms total)

[-----] Global test environment tear-down
[=====] 7 tests from 2 test suites ran. (71 ms total)
[ PASSED ] 7 tests.
```

Rys. 2.8. Losowa kolejność wykonania testów jednostkowych komponentu *dispatcher*

3. Program ćwiczenia

Zadanie 1. Zaimplementuj funkcję *unsigned int fibonacci(unsigned int)*, obliczającą n -ty wyraz ciągu Fibonacciego (w sposób iteracyjny bądź rekurencyjny), a następnie napisz pakiet testów, którymi sprawdzisz poprawność jej działania.

Zadanie 2. Napisz pakiet testów, którymi sprawdzisz poprawność działania funkcji *removeByIndex()*, usuwającej węzeł listy jednokierunkowej o zadanym indeksie (Zadanie 2. Ćw. 10). W tym celu rozpatrz następujące przypadki testowe:

- pusta lista – usuwanie węzła o indeksie 0;
- pusta lista – usuwanie węzła o indeksie większym niż 0;
- lista jednoelementowa - usuwanie węzła o indeksie 0;
- lista jednoelementowa - usuwanie węzła o indeksie większym niż 0;
- lista n -elementowa – usuwanie węzła o indeksie 0;
- lista n -elementowa – usuwanie węzła o indeksie 1;
- lista n -elementowa – usuwanie węzła o indeksie $n + 1$.

[Uwaga: funkcja *removeByIndex()* powinna być zabezpieczona przed usuwaniem węzłów o indeksie spoza listy.]

Zadanie 3. Napisz pakiet testów, którymi sprawdzisz poprawność działania funkcji *dispatch()*, mapującej instrukcje tekstowe na odpowiadające im funkcje porównujące wartości węzłów listy jednokierunkowej (Zadanie 3. Ćw. 11).

4. Dodatek

4.1. System budowania CMake

CMake (skrót od ang. *Cross-platform Make*) to narzędzie służące do automatyzacji procesu budowania aplikacji napisanych w języku *C* lub *C++*. Jest on niezależny od platformy sprzętowej, systemu operacyjnego i kompilatora. Wspiera również kompilację skrośną (ang. *cross-compilation*). **CMake** bazuje na języku skryptowym, który umożliwia konfigurację procesu kompilacji i konsolidacji kodu źródłowego. Kod skryptów umieszczany jest w plikach o nazwie **CMakeLists.txt**.

Na listingu 11. przedstawiono prosty skrypt systemu budowania **CMake** zawierający trzy instrukcje. Jego wykonanie umożliwia zbudowanie aplikacji napisanej w języku *C++* składającej się z jednej jednostki translacji o nazwie *main.cpp*. Pierwsze polecenie, tj. **cmake_minimum_required()** określa minimalną wersję systemu budowania **CMake**, jaka jest wymagana do wykonania skryptu (tu: wersja 3.22). Instrukcja **project()** ustawia nazwę projektu (pod zmienną *PROJECT_NAME*). Instrukcja **add_executable()** określa jakie pliki będą współtworzyć plik wykonywalny aplikacji (tu: plik *main.cpp* jest jedynym źródłem tworzącym plik wykonywalny *my_app*). Skrypty systemu **CMake** są niewrażliwe na wielkość liter. Aby wykonać skrypt zawarty w pliku **CMakeLists.txt** należy za pomocą konsoli systemowej wywołać polecenie:

cmake [ścieżka_do_pliku_CMakeLists.txt]

```
1 cmake_minimum_required(VERSION 3.22)
2 project(my_app)
3
4 add_executable(my_app main.cpp)
```

Listing 11. Podstawowy skrypt *CMake*

Jeżeli polecenie wywoływane jest z poziomu katalogu zawierającego plik **CMakeLists.txt**, to można posłużyć się uproszczonym zapisem:

cmake .

Pojedyncza kropka (.) określa bieżący katalog w systemie plików, podczas gdy podwójna kropka (..) określa katalog nadrzędny (*ang. parent directory*). W wyniku wywołania polecenia **cmake** zostanie wygenerowany plik **Makefile**, zawierający zestaw instrukcji narzędzia **GNU Make**, służącego do budowania pliku wykonywalnego aplikacji. Wywołanie polecenia **make** z poziomu katalogu zawierającego plik **Makefile** skutkuje utworzeniem pliku wykonywalnego aplikacji (tu: *my-app.exe*). Polecenie **make** nie wymaga przekazywania dodatkowych argumentów. Poza plikiem **Makefile** polecenie **cmake** generuje kilka dodatkowych plików (*CMakeCache.txt*, *cmake-intall.cmake*) oraz katalog *CMakeFiles*, co może zaciemniać utrzymywaną strukturę katalogów projektu. Powszechnym rozwiązaniem jest wstępne utworzenie pustego katalogu pomocniczego **build** jako folderu przeznaczonego na przechowywanie plików wygenerowanych w procesie przetwarzania skryptu **CMake**. Wówczas wywołanie polecenia:

cmake ..

z poziomu katalogu **build** skutkuje umieszczeniem wygenerowanych plików pomocniczych wewnątrz katalogu **build**.

Częstym wymaganiem jest ustawienie określonej wersji standardu języka *C++*. W tym celu można się posłużyć poleceniem **set()**, jak na listingu 12. Instrukcja **set(CMAKE_CXX_STANDARD 20)** przypisuje zmiennej *CXX_STANDARD* wartość 20, co przekłada się na użycie flagi **-std=c++20** (lub równoważnej) w procesie kompilacji kodu źródłowego. Z kolei instrukcja **set(CMAKE_CXX_STANDARD_REQUIRED TRUE)** określa, że proces budowania pliku wykonywalnego aplikacji ma zostać przerwany, jeżeli zainstalowany kompilator nie wspiera ustalonego standardu języka *C++*. W przeciwnym razie standard języka *C++* zostałby obniżony do najwyższej

wersji wspieranej przez wykorzystywany kompilator.

```
1 cmake_minimum_required(VERSION 3.22)
2 project(my_app)
3
4 set(CMAKE_CXX_STANDARD 20)
5 set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
6
7 add_executable(my_app main.cpp)
```

Listing 12. Ustawianie standardu języka *C++* w skrypcie *CMake*

W analogiczny sposób, korzystając z instrukcji *set()*, można zwiększyć poziom szczegółowości logowania podczas budowania pliku wykonywalnego aplikacji (listing 13).

```
1 cmake_minimum_required(VERSION 3.22)
2 project(my_app)
3
4 set(CMAKE_VERBOSE_MAKEFILE ON)
5
6 set(CMAKE_CXX_STANDARD 20)
7 set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
8
9 add_executable(my_app main.cpp)
```

Listing 13. Zwiększenie poziomu szczegółowości logowania narzędzia *Make*

Skrypty systemu *CMake* pozwalają również na definiowanie własnych zmiennych pomocniczych. Służy do tego opisane wcześniej polecenie *set()*. Przykład przedstawiono na listigu 14. Zmienna pomocnicza *SOURCE_FILES* przechowuje nazwy wszystkich plików źródłowych wykorzystywanych w procesie budowania pliku wykonywalnego aplikacji (tu: jeden plik – *main.cpp*).

Wyłuskanie wartości zmiennej realizowane jest za pomocą operatora `${}`. Linie poprzedzone symbolem `#` traktowane są jako komentarze. Wszystkie instrukcje ***set()*** muszą zostać umieszczone powyżej instrukcji ***add_executable()***.

```
1 cmake_minimum_required(VERSION 3.22)
2 project(my_app)
3
4 set(CMAKE_VERBOSE_MAKEFILE ON)
5
6 set(CMAKE_CXX_STANDARD 20)
7 set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
8
9 #Definicja zmiennej pomocniczej SOURCE_FILES
10 set(SOURCE_FILES main.cpp)
11
12 add_executable(my_app ${SOURCE_FILES})
```

Listing 14. Definiowanie zmiennych pomocniczych w skrypcie *CMake*

Skrypt z listingu 14. można w łatwy sposób rozbudować, aby uwzględnić dodatkowe pliki (nagłówkowe i źródłowe) wchodzące w skład projektu. Zostało to przedstawione na listingu 15. Oprócz pliku *main.cpp* plik wykonywalny aplikacji współtworzą teraz pliki nagłówkowe *definitions.h* i *parser.h* oraz plik źródłowy *parser.cpp*. Aby uniknąć dublowania kodu, nazwa pliku wykonywalnego została przekazana do instrukcji ***add_executable()*** za pośrednictwem zmiennej *PROJECT_NAME* (ustawianej za pomocą polecenia ***project()***).

```
1 cmake_minimum_required(VERSION 3.22)
2 project(my_app)
3
4 set(CMAKE_VERBOSE_MAKEFILE ON)
5
6 set(CMAKE_CXX_STANDARD 20)
7 set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
8
9 #Definicja zmiennej pomocniczej HEADER_FILES
10 set(HEADER_FILES
11     definitions.h
12     parser.h)
13
14 #Definicja zmiennej pomocniczej SOURCE_FILES
15 set(SOURCE_FILES
16     main.cpp
17     parser.cpp)
18
19 add_executable(${PROJECT_NAME} ${SOURCE_FILES} ${
    HEADER_FILES})
```

Listing 15. Dodawanie kolejnych plików projektu w skrypcie *CMake*

Jeżeli kod źródłowy projektu ma zostać skonsolidowany do postaci biblioteki programistycznej należy posłużyć się poleceniem ***add_library()*** zamiast polecenia ***add_executable()***. Instrukcja ***add_library()*** przyjmuje dodatkowy parametr określający czy kod ma zostać przetworzony do postaci biblioteki statycznej:

```
add_library(${PROJECT_NAME} STATIC ${SOURCE_FILES}
           ${HEADER_FILES})
```

czy dynamicznej:

```
add_library(${PROJECT_NAME} SHARED ${SOURCE_FILES}
            ${HEADER_FILES}).
```

System **CMake** umożliwia również wskazanie ścieżki, pod którą ma zostać umieszczony wygenerowany plik (wykonywalny bądź biblioteki). W tym celu należy za pomocą instrukcji **set()** ustawić wartości odpowiednich zmiennych:

- **CMAKE_ARCHIVE_OUTPUT_DIRECTORY** – w przypadku budowania pliku biblioteki statycznej;
- **CMAKE_LIBRARY_OUTPUT_DIRECTORY** – w przypadku budowania pliku biblioteki dynamicznej;
- **CMAKE_RUNTIME_OUTPUT_DIRECTORY** – w przypadku budowania pliku wykonywalnego aplikacji.

Przykład przedstawiono na listingu 16. Plik wykonywalny *my_app.exe* zostanie umieszczony w katalogu *deploy*. Jeżeli katalog nie istnieje, to wywołanie polecenia **make** skutkuje jego utworzeniem. Zmienna **PROJECT_BINARY_DIR** domyślnie przechowuje ścieżkę do folderu, w którym generowane są pliki **CMake** (np. do folderu *build*).

```
1 cmake_minimum_required(VERSION 3.22)
2 project(my_app)
3
4 set(CMAKE_VERBOSE_MAKEFILE ON)
5
6 set(CMAKE_CXX_STANDARD 20)
7 set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
8
9 #Definicja zmiennej pomocniczej HEADER_FILES
10 set(HEADER_FILES
11     definitions.h
12     parser.h)
```

```

13
14 #Definicja zmiennej pomocniczej SOURCE_FILES
15 set(SOURCE_FILES
16     main.cpp
17     parser.cpp)
18
19 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${
20     PROJECT_BINARY_DIR}/deploy)
21
22 add_executable(${PROJECT_NAME} ${SOURCE_FILES} ${
23     HEADER_FILES})

```

Listing 16. Określanie położenia budowanego pliku wykonywalnego w skrypcie *CMake*

Aby skonsolidować bibliotekę programistyczną podczas budowania pliku wykonywalnego aplikacji należy wywołać polecenie ***target_link_libraries()*** po instrukcji ***add_executable()***. Przykład przedstawiono na listingu 17. Aplikacja *my_app* konsoliduje bibliotekę *my_lib*. Pierwszy argument wywołania instrukcji ***target_link_libraries()*** stanowi nazwa budowanego pliku wykonywalnego. Drugi argument to pełna ścieżka do konsolidowanej biblioteki programistycznej. Polecenie ***find_library()*** umożliwia automatyczne wyszukanie określonej biblioteki programistycznej (tu: *my_lib*) i zapisanie ścieżki do pliku pod wskazaną zmienną pomocniczą (tu: *LIB_FILE*). Flaga ***REQUIRED*** określa, że proces generowania pliku *Makefile* ma zostać przerwany w przypadku nieodnalezienia pliku biblioteki. Jeżeli biblioteka leży w nie-standardowej lokalizacji na dysku komputera, koniecznym może okazać się jawne wskazanie katalogów do przeszukania przez system *CMake*. Służy do tego opcja ***HINTS***:

```

find_library([nazwa_zmiennej_pomocniczej] [nazwa_biblioteki]
             HINTS [bezwzględna_ścieżka_do_katalogu_biblioteki]
             REQUIRED)

```

Jeżeli załączane pliki nagłówkowe położone są w innym katalogu niż plik **CMakeList.txt** (tu: w katalogu *include*), to warto posłużyć się poleceniem **target_link_directories()** w celu poinformowania kompilatora o niestandardowej lokalizacji plików nagłówkowych. W przeciwnym razie koniecznym będzie uwzględnianie ścieżki do pliku nagłówkowego w dyrektywach preprocesora `#include`. Flaga **PUBLIC** określa, że wskazany katalog zawiera publiczny interfejs aplikacji lub biblioteki.

```
1 cmake_minimum_required(VERSION 3.22)
2 project(my_app)
3
4 set(CMAKE_VERBOSE_MAKEFILE ON)
5
6 set(CMAKE_CXX_STANDARD 20)
7 set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
8
9 #Definicja zmiennej pomocniczej HEADER_FILES
10 set(HEADER_FILES
11     include/definitions.h
12     include/parser.h)
13
14 #Definicja zmiennej pomocniczej SOURCE_FILES
15 set(SOURCE_FILES
16     main.cpp
17     parser.cpp)
18
19 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${
20     PROJECT_BINARY_DIR}/deploy)
21 find_library(LIB_FILE my_lib REQUIRED)
22
```

```
23 add_executable(${PROJECT_NAME} ${SOURCE_FILES} ${  
    HEADER_FILES})  
24 target_link_libraries(${PROJECT_NAME} ${LIB_FILE})  
25 target_link_directories(${PROJECT_NAME} PUBLIC  
    include)
```

Listing 17. Konsolidacja biblioteki programistycznej w skrypcie *CMake*

Aby zapewnić przenośność kodu między odrębnymi platformami sprzętowymi i systemami operacyjnymi należy unikać jawnego podawania bezwzględnych ścieżek do plików bądź katalogów w skryptach **CMake**. Wygodnym rozwiązaniem jest odwoływanie się do zmiennych zawierających ścieżki do katalogów powiązanych z systemem budowania **CMake** i tworzenie przy ich pomocy ścieżek względnych. Jedną z takich zmiennych jest wspomniana wcześniej **PROJECT_BINARY_DIR**, odnosząca się do folderu z plikami wygenerowanymi przez system **CMake** (por. listing 16.). Inną przydatną zmienną jest **CMAKE_CURRENT_SOURCE_DIR**, zawierająca ścieżkę do aktualnie wykonywanego pliku *CMakeLists.txt*.

Literatura

- [1] *Advanced googletest Topics*. URL: <https://github.com/google/googletest/blob/master/googletest/docs/advanced.md>.
- [2] *Googletest Primer*. URL: <https://github.com/google/googletest/blob/master/googletest/docs/primer.md>.
- [3] *Levels of Testing*. URL: <https://artoftesting.com/levels-of-software-testing>.
- [4] *Testowanie białoskrzynkowe*. URL: <http://getistqb.com/docs/sylabus-poziomu-podstawowego-istqb-2018-wersja-1-01/2-testowanie-w-cyklu-zycia-oprogramowania/2-3-typy-testow/2-3-3-testowanie-bialoskrzynkowe/>.