JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 11 (2019) 131-136
Wysłane: 2019-02-14
Przyjęte: 2019-02-22

Metody prowadzenia testów jednostkowych w standardzie C++14 z wykorzystaniem biblioteki GMOCK

Kamil Strózik*
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. W artykule przedstawiono jeden z problemów występujących podczas korzystania ze standardu C++14 wraz z biblioteką Google Mock. Na podstawie wprowadzonego problemu omówiono możliwe rozwiązania, a także zaprezentowano poprawiony schemat dla podejścia TDD.

Słowa kluczowe: testy jednostkowe; C++14; Google Mock; TDD;

*Autor do korespondencji.

Adres/adresy e-mail: struzikk@gmail.com

Methods for conducting unit tests in the C++14 standard using the GMOCK library

Kamil Strózik*
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The article presents one of the problems encountered when using the C ++ 14 standard along with the Google Mock library. Based on the introduced problem, possible solutions were discussed, as well as an improved scheme for the TDD approach was presented.

Keywords: unit tests; C++14; Google Mock; TDD;

*Corresponding author.

E-mail address/addresses: struzikk@gmail.com

1. Wstęp

Wraz z kolejnymi standardami języka C++ pojawił się problem zgodności nowych funkcjonalności z biblioteką Google Mock. Proste podejście „red-green-refactor" z metodyki Test-Driven Development w przypadkach braku zgodności przestało mieć zastosowanie. W związku z tym należy ponownie zastanowić się nad krokami użycia TDD poprzez istniejące problemy.

2. Idee testowania kodu

2.1. Znaczenie testowania

Testowanie przestaje być dodatkową, opcjonalną częścią projektu, zaczyna być uznawane za coś obowiązkowego i oczywistego. Dodatkowo coraz więcej firm, projektów planuje skrupulatnie fazy testowania.

Testowanie stało się obowiązkiem z powodu omyłkowości ludzi. Dobrze znanym przykładem jest uniknięcie o włos III wojny światowej. 26 września 1983 roku system Związku Radzieckiego podał komunikat o rozpoznaniu lecących amerykańskich rakiet w stronę podmoskiewskiej bazy wojskowej. Zgodnie z ówczesnymi procedurami, taki komunikat miał rozpocząć kontratak Związku Radzieckiego doprowadzając do kolejnej wojny światowej. Na szczęście komunikat został podważony i system nie przyczynił się do śmierci ludzi [1].

Kolejnym przykładem jest zaciemnienie północno-wschodnie z 2003 roku (Northeast blackout of 2003). Dostęp do prądu straciło wtedy 55 milionów osób ze Stanów Zjednoczonych oraz Kanady. Przywracanie dostępu do prądu trwało przez kolejne dwa tygodnie [2].

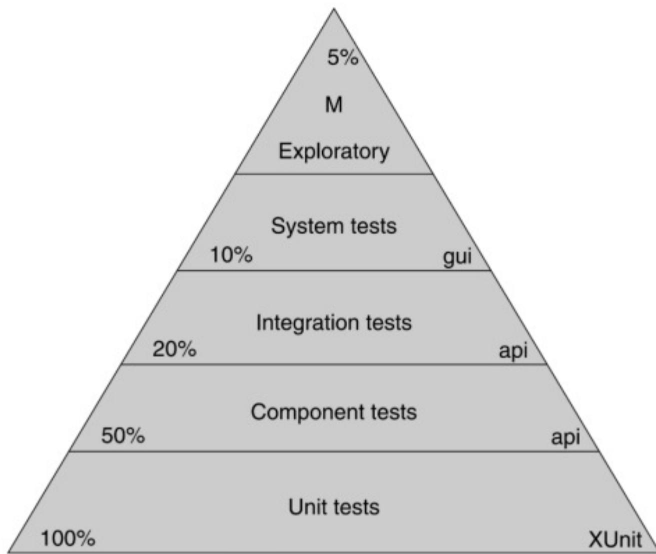
Wszystkie te błędy mogły zostać wykryte przez testowanie oprogramowania. W celu znajdowania błędów został opracowany model V, który dla każdego etapu projektu informatycznego wprowadza jego testowanie. Czy to poprzez przejrzenie wymagań, analizę struktury projektu, czy późniejsze testy powstałego kodu oraz aplikacji jako całości.

Obecnie testy są integralną częścią produktu, a kod testów jest tak samo przechowywany i wersjonowany jak każdy inny kod źródłowy [3]. Pozwala to zauważyć, że jakość kodu testów jest tak samo ważna, jak jakość kodu testowanego, jeśli nie wyższa [4].

2.2. Znaczenie testów jednostkowych

Bazując na modelu V Robert Martin [5] określa jak dokładnie ma być przetestowane oprogramowanie konkretnym rodzajem testów. O ile same testy systemowe powinny się

skupić na przetestowaniu około 10% kodu programu, to testy jednostkowe mają pokrywać całość kodu, zatem celem powinno być 100% (rys. 1.).



Rys. 1. Piramida testów automatycznych [3]

W celu łatwiejszego osiągnięcia wysokiego pokrycia testami jednostkowymi powstało pojęcie TDD (Test-Driven Development), które jest dyscypliną pomagającą rozwijać jakość projektu, a nie tylko losowym podejściem do weryfikacji elementów systemu [6]. TDD można przedstawić również jako sposób zarządzania strachem podczas programowania [7]. Zdarza się również, że w obrębie metodyki TDD wyróżnia się UTDD (Unit Test-Driven Development), które ma potwierdzać poprawne działanie systemu oraz ATDD (Acceptance Test-Driven Development), które ma potwierdzić działanie logiki biznesowej. [8]

Całą listę argumentów dlaczego testy jednostkowe wraz z TDD są tak istotne przedstawia Jeff Langr, m. in. zwiększoną wykrywalność błędów z 40% do 90% w fazie implementacji przy jednoczesnym wzroście czasu jej trwania w projekcie z 15% do 35%, czy zależność odwrotnie proporcjonalną między ilością testów jednostkowych, a złożonością kodu [6]. Z kolei programiści z wielu firm przedstawiają wykorzystanie napisanych testów jednostkowych jako testów regresyjnych, które w krótkim czasie są w stanie potwierdzić działanie całego produktu [9]. Natomiast koreańscy naukowcy przedstawiają możliwość wykorzystania istniejących testów jednostkowych do wygenerowania testów integracyjnych, jako bazę scenariuszy testowych [10].

Skoro pisanie testów jednostkowych jest tak istotne, to jaka jest ich definicja? Test jednostkowy jest to zautomatyzowany fragment kodu wywołujący kod testowanej jednostki, który następnie sprawdza pewne założenia o pojedynczym wyniku końcowym tej jednostki. Prawie zawsze do pisania testów jednostkowych jest wykorzystywana zewnętrzna biblioteka, co pozwala na łatwe i szybkie uruchamianie. Test powinien być wiarygodny, czytelny oraz łatwy w utrzymaniu. Dodatkowo powinien zwracać te same wyniki dopóki nie zmieni się testowany kod. [11]

Każdy element powyższej definicji jest ważny jeśli chodzi o metodę prowadzenia testów. Pozwala ona na łatwiejsze pisanie kodu, implementację nowych funkcjonalności, a uogólniając ulepsza fazę implementacji oraz cały etap powstawania oprogramowania i jego utrzymanie.

2.3. Metoda tworzenia testów jednostkowych

W poprzednim podrozdziale została już przedstawiona definicja i zalety techniki TDD. Niniejszy podrozdział omówi kolejne kroki jej stosowania.

TDD można przedstawić w stosunkowo krótkim schemacie krokowym:

1) Napisać test dla nowego wymagania funkcjonalnego.
2) Uruchomić wszystkie testy i sprawdzić, że warunki z nowo napisanego testu nie są spełniane.
3) Zaimplementować jedynie taką ilość kodu produkcyjnego, który pozwoli na spełnienie warunków z nowego testu.
4) Uruchomić wszystkie testy, w przypadku nie działania któregoś z testów, powrócić do kroku 3.
5) Uporządkować kod pod kątem zmniejszenia złożoności, poprawy czytelności, czy łatwości konserwacji.
6) Uruchomić wszystkie testy, jeśli wykryją one błędy, to należy je poprawić.
7) Jeśli są nowe wymagania funkcjonalne, to powrócić do kroku 1. w przeciwnym wypadku można zakończyć proces.

Odnosząc się do znanej mantry „red-green-refactor” [7] z metodyki TDD, można do niej przypisać odpowiednie kroki schematu:

- „red” – kroki 1 i 2, gdy nowy test potwierdza brak potrzebnej funkcjonalności,
- „green” – kroki 3 i 4, gdy test daje „zielone światło” szybkiej zmianie kodu produkcyjnego implementującego funkcjonalność
- „refactor” – kroki 5 i 6, gdy zostaje uprzątnięty kod produkcyjny oraz testowy.

Tak prosty schemat przyczynił się do upowszechnienia się techniki TDD, która oprócz licznych zalet wymienionych wcześniej, posiada również ograniczenia, przez które nie zawsze może być stosowana.

3. Różnice pomiędzy standardami C++03, a C++14

3.1. Wprowadzenie

„Zaskakujące jak C++11 wydaje się być nowym językiem. Elementy pasują lepiej do siebie, tworząc wyższy poziom programowania, które jest bardziej naturalne niż dotychczas i tak wydajne jak nigdy dotąd.” Bjarne Stroustrup [12]

Powyższy cytat pokazuje jak następcę standardu C++03 postrzega jego twórca. Wiele elementów dodano, sporo

poprawiono. Jednak by można było mówić o kompletnych zmianach należy wziąć pod uwagę standard C++14. Poprawia on wady wprowadzonych nowości, niedociągnięcia przedstawionych udogodnień.

3.2. Inteligentne wskaźniki

Język C++ od dawna był oskarżany o to, że programy napisane z jego wykorzystaniem niejednokrotnie posiadają wycieki pamięci. Programiści wskazywali na trudności w zarządzaniu pamięcią, konieczność ręcznego zwalniania zaalokowanej pamięci. A same wskaźniki były równie ważne jak niebezpieczne [13]. W związku z tymi zarzutami już w C++03 występował auto_ptr równolegle z surowymi wskaźnikami. Niestety stwarzał on wiele problemów, przez które nie był powszechnie używany. Standard C++14 posiada alternatywę dla auto_ptr. Począwszy od C++11, biblioteka języka udostępnia dwa rodzaje inteligentnych wskaźników: unique_ptr, shared_ptr [13].

Unique_ptr przejął całą funkcjonalność wskaźników auto_ptr, dodatkowo implementując brakującą semantykę przenoszenia [14]. W standardzie C++14 auto_ptr jest oznaczony jako przestarzały o czym informuje kompilator wyświetlając ostrzeżenia w czasie kompilacji, natomiast w standardzie C++17 został on całkowicie usunięty [15]. Sam unique_ptr zyskał prostszy i czytelniejszy interfejs, co w połączeniu z szybkością działania i niezawodnością wpłynęło na jego popularność. Stosuje się go wszędzie tam, gdzie istotne jest wyłączne prawo do posiadania danego obszaru pamięci oraz zarządzanie nim. Dlatego wielokrotnie wykorzystuje się unique_ptr do pozyskania pewnych zasobów, wykonania na nich operacji, a następnie na zwolnieniu tych pozyskanych zasobów [13]. Samo zwolnienie zasobów odbywa się w sposób automatyczny, przez co ten typ wskaźnika jest używany jako zmienna klasy, dzięki czemu nie ma już potrzeby implementacji destruktora w klasie.

Wśród inteligentnych wskaźników większą popularność zdobył shared_ptr, który w sposób automatyczny zarządza współdzieleniem własności zasobu. Dzięki temu shared_ptr daje możliwość utworzenia wielu wskaźników odnoszących się to tego samego obiektu, a w chwili gdy przestaje istnieć ostatni wskaźnik do obiektu jest on usuwany, a jego zasoby zwalniane [13]. Dla bardziej złożonych zastosowań w standardzie języka znajdują się również klasy pomocnicze, takie jak weak_ptr, bad, weak_ptr, czy enabled_shared_from_this.

4. Google Mock

By móc pisać testy zgodnie z podejściem TDD należy korzystać z atrap obiektów, które w języku angielskim określa się je jako mock. Atrapa to fałszywy obiekt zewnętrznej zależności w systemie, który pozwala sterować przebiegiem testu, a także sprawdzać wywoływanie funkcji udawanego interfejsu [16].

Czy biblioteki Google Test oraz Google Mock zachęcają do korzystania z TDD w czystej formie? Odpowiedź jest niejednoznaczna. Ponieważ rozpoczynając pracę nad projektem warto zacząć od prostego testu.

Przykład 1. Prosty test działania biblioteki GTest

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

TEST(VideoConverterTest, SimpleTest)
{
ASSERT_TRUE(true);
}
```

Przykład 1. pokazuje taki prosty test, który pozwala rozpocząć cykl Red-Green-Refactoring. Podczas pierwszej próby kompilacji tego testu zostają wyświetlone błędy o braku pliku nagłówkowego gtest.h. Po wskazaniu ścieżki do bibliotek Google udaje się zbudować program oraz zaimplementowany test kończy się sukcesem, co potwierdza prawidłowe skonfigurowanie projektu. Faza refaktoryzacji, uprzątnięcia kodu oznacza w tym momencie implementację właściwych testów do wymagań.

Wymaganiem będzie implementacja klasy VideoConverter, która wykorzystując interfejs IManager, którego funkcja na postawie MemoryStruct wylicza ilość wolnej pamięci.

Samo wymaganie może wydać się trudne i skomplikowane, ale dzięki implementacji testu upraszcza się.

Przykład 2. Implementacja prostego testu dla wymagania

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

TEST(VideoConverterTest,
computeFreeSpace_MemoryAsParam_ReturnedValue)
{
  IManagerMock managerMock;
  VideoConverter testObj(managerMock);

  MemoryStruct* pMemory = new MemoryStruct();
  const unsigned EXPECTED_SIZE{5u};

  EXPECT_CALL(managerMock,
          computeFreeSpace(pMemory))
                 .WillOnce(Return(EXPECTED_SIZE));

  ASSERT_DOUBLE_EQ(EXPECTED_SIZE,
              testObj.getFreeMemorySize(pMemory));

  delete pMemory;
}
```

Test przedstawiony w przykładzie 2. nie kompiluje się. Uzyskane błędy mówią o braku definicji dla pojęć: VideoConverter, IManagerMock, MemoryStruct. Analizując przebieg testu można przewidzieć implementację. Klasa VideoConverter w konstruktorze przyjmie referencję do interfejsu IManager, konstruktor powinien zachować referencję jako zmienną obiektu, dodatkowo klasa powinna mieć funkcję getFreeMemorySize, która przyjmuje obiekt typu MemoryStruct, a zwraca liczbę bez znaku. Interfejs powinien mieć funkcję o nazwie computeFreeSpace o takim samym typie parametru i zwracanym jak testowana funkcja.

Przykład 3. Implementacja prostego testu wraz zależnościami dla wymagania

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

using namespace testing;

struct MemoryStruct {};

class IManager {
 public:
  virtual ~IManager(){};
  virtual unsigned computeFreeSpace(MemoryStruct*
memory) = 0;
};

class IManagerMock : public IManager {
 public:
  MOCK_METHOD1(computeFreeSpace,
unsigned(MemoryStruct*));
};

class VideoConverter {
 public:
  VideoConverter(IManager& manager) :
m_manager(manager) {}
  unsigned getFreeMemorySize(MemoryStruct* memory) {
    return m_manager.computeFreeSpace(memory);
  }

 private:
  IManager& m_manager;
};

TEST(VideoConverterTest,
     computeFreeSpace_MemoryAsParam_ReturnedValue) {
  IManagerMock managerMock;
  VideoConverter testObj(managerMock);
  MemoryStruct* pMemory = new MemoryStruct();
  const unsigned EXPECTED_SIZE{5u};

  EXPECT_CALL(managerMock,
computeFreeSpace(pMemory))
      .WillOnce(Return(EXPECTED_SIZE));

  ASSERT_DOUBLE_EQ(EXPECTED_SIZE,
testObj.getFreeMemorySize(pMemory));

  delete pMemory;
}
```

Kompilacja i uruchomienie testu z przykładu 3. została zakończona sukcesem, został napisany działający kod. Wciąż jednak potrzebna jest faza sprzątania kodu. Kod produkcyjny tj. VideoConverter, IManager, MemoryStruct powinny się znaleźć w osobnych plikach, w katalogach aplikacji, a testy i atrapy klas w analogicznej, lecz osobnej strukturze. Jeśli IManagerMock będzie wykorzystywany przez kilka klas testowych, to także powinien znaleźć się w oddzielnym pliku.

Przeniesienie klas do osobnych plików powinno zakończyć się próbą kompilacji i uruchomieniem testów. Napisany wcześniej test powinien potwierdzić, że nie była zmieniana logika biznesowa, sam przebieg programu się nie zmienił. Kod produkcyjny został uprzątnięty, warto również tworzyć testy jednostkowe najprościej jak to tylko możliwe.

Przykład 3. prezentuje sposób tworzenia i wykorzystania bibliotek oferowanych przez Google. Atrapę tworzy się na podstawie klasy bazowej, dla funkcji wirtualnych. Ważne, że klasa posiada wirtualny destruktor, który zapewni właściwe niszczenie hierarchii tworzonych obiektów. Samo utworzenie sprowadza się do wykorzystania makra MOCK_METHOD1 [17]. Przyjmuje ono jako pierwszy parametr nazwę funkcji, następnie typ zwracany, a w nawiasie tyle typów parametrów ile przedstawia samo makro, w tym przypadku jest to jeden parametr. W ten sposób na etapie kompilacji jest tworzona klasa atrapy, która przesłania metody z prawdziwego interfejsu. Najczęściej do tworzenia atrap wykorzystywany jest skrypt gmock_gen, który na podstawie podanej ścieżki do pliku nagłówkowego klasy oraz jej nazwy, generuje do niej atrapę [18].

Wykorzystanie utworzonego obiektu atrapy jest równie proste. Dla weryfikacji wywołania funkcji na obiekcie atrapy używa się makra EXPECT_CALL, które przyjmuje konkretny obiekt atrapy, funkcje i jej argumenty, dodatkowo można zdefiniować szereg akcji, które mają się wydarzyć w czasie wywołania funkcji. Przykład 3. demonstruje funkcje atrapy, która ma zwrócić oczekiwaną wartość.

5. Rozwiązanie problemu niekopiowalnych obiektów

Język C++14 posiada semantykę przenoszenia, posiada inteligentne wskaźniki. Konstrukcją, która łączy te dwa zagadnienia jest unique_ptr, który implementuje politykę przeniesienia, jednak nie ma zaimplementowanej semantyki kopiowania. W części zastosowań jest to zaletą, gdy istotna jest pewność, że aktualny wskaźnik jako jedyny odnosi się do przechowywanej pamięci.

Przykład z poprzedniego rozdziału sugeruje możliwość wykorzystania inteligentnego wskaźnika w miejsce surowego, dzięki czemu program będzie odporniejszy na wycieki pamięci. Aktualnym zadaniem będzie jedynie taka zmiana, bez ingerencji w logikę funkcji. Takie zadanie wydaje się być mechanicznym, skryptowym zadaniem, które powinno zamienić wystąpienia MemoryStruct* na std::unique_ptr, warto również do tworzenia użyć funkcji std::make_shared. Kod wygląda poprawnie, wszystkie odwołania do surowych wskaźników zostały zamienione. Wydaje się, że kompilacja i utworzony test powinny przejść bezbłędnie.

Przykład 4. Błąd zwracany przy próbie użycia unique_ptr

```
/home/kamil/01_UniquePtrAsParam/tests/VideoConverterTes
t.cpp:39:31:
error: use of deleted function ‘std::unique_ptr<_Tp,
_Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with
_Tp = MemoryStruct; _Dp = std::default_delete]’
computeFreeSpace(pMemory)).WillOnce(Return(EXPECTED_S
IZE));
```

Przykład 4. pokazuje, że makro EXPECT_CALL próbuje wywołać konstruktor kopiujący, który jest usunięty. Ta próba wywołania powoduje, że kompilacja obiektu Matcher nie może zakończyć się pomyślnie, przez co cały proces kompilacji ulega przerwaniu.

W przypadku tego błędu można jego rozwiązania albo pozostawić ten fragment kodu nieprzetestowanym, co nierzadko się zdarza. Ignorowanie błędu, unikanie go jest złym rozwiązaniem, mogącym w przyszłości zaszkodzić całemu projektowi. Jak zatem rozwiązać ten problem?

Problem sprawia weryfikacja wywołania, jest ona jednak kluczowa z punktu widzenia testu, nie można z niej zrezygnować. W takim razie powraca problem braku możliwości kopiowania unique_ptr, pojawia się możliwość zamiany na shared_ptr. Jednak z można z niej skorzystać jedynie wtedy, gdy nie jest istotna wydajność, ponieważ w niektórych operacjach shared_ptr potrafi być dwukrotnie wolniejszy niż unique_ptr. Ponadto shared_ptr nie zapewnia kontroli, czy jest jedynym właścicielem przechowywanego obiektu.

Gdy unique_ptr jest jedynym rozwiązaniem to trzeba się zastanowić co tak naprawdę jest ważne w aktualnym teście, co należałoby przetestować. W tym przypadku powinno zostać sprawdzone wywołanie funkcji interfejsu, że zostanie jej przekazany ten sam obiekt, który trafił do testowanej funkcji. Jednak czy istotna jest sama klasa unique_ptr, czy to co na nią wskazuje? Jeśli istnieje potrzeba sprawdzenia jedynie obiektu, na który wskazuje ten wskaźnik problem wydaje się o wiele prostszy.

Rozwiązaniem, które przedstawia przykład 5. jest wykorzystanie mechanizmu dziedziczenia, dzięki któremu można przesłonić funkcję bazową własną implementacją. W ciele weryfikowanej funkcji zostało wywołana funkcja z makra GMock'a. Makro natomiast jako parametr przyjmuje wskaźnik.

Przykład 5. Poprawione makro dla sprawdzania metody computeFreeSpace

```
MOCK_METHOD1(computeFreeSpaceProxy,
unsigned(MemoryStruct*));

unsigned computeFreeSpace(
  std::unique_ptr<MemoryStruct> memory) override {
    return computeFreeSpaceProxy(memory.release());
}
```

Dzięki przekazywaniu do makra jedynie wskaźnika, istnieje obecnie możliwość weryfikacji wywołania w niemal identyczny sposób, jak podczas starej implementacji tego przykładu z użyciem surowych wskaźników, co przedstawia przykład 6.

Przykład 6. Sprawdzenie wywołania funkcji computeFreeSpace przy użyciu makra

```
EXPECT_CALL(*m_IManagerMock,
    computeFreeSpaceProxy(pMemory.get()))
        .WillOnce(Return(EXPECTED_SIZE));
```

Kompilacja i uruchomienie testu przebiegło pomyślnie. Żądana zmiana w kodzie źródłowym zgodnie z przewidywaniami wymagała jedynie zamianę typów. Więcej problemów przysporzył kod testów. Wygenerowana przez skrypt gmock_gen atrapa interfejsu IManager zawiera jedynie makro, które uniemożliwia jego użycie. W tym wypadku zostaje zaburzona idea TDD, w której buduje się test, a w dalszych krokach implementuje się kod potrzebny do pozytywnego wykonania testu. Oczywiście pierwszy powinien być test, by wiedzieć co powinno zostać zaimplementowane. Jednak w kolejnym kroku podczas korzystania ze standardu C++14 wraz z biblioteką GMock należy poprawić wygenerowaną atrapę oraz test.

6. Poprawiona metoda tworzenia testów jednostkowych

Przedstawiony schemat krokowy dla metodyki TDD nie znalazł zastosowania w przedstawionych przykładach. Mantra „red-green-refactor" chociaż ogólnie poprawna to jednak nie była w pełni przestrzegana. Błędem w schemacie krokowym jest założenie braku problemów przy implementacji testu na podstawie wymagań, czy implementacji kodu produkcyjnego na podstawie testu. Jest to jednak zrozumiałe, ponieważ trudno stworzyć ogólny schemat uwzględniający wszystkie problemu.

Na podstawie przedstawionych przykładów opracowano poprawiony schemat krokowy metodyki TDD dla nowego standardu C++ wykorzystywanego wraz z biblioteką GMock.

Poniżej przedstawiono poprawiony schemat krokowy:

1) Napisać test dla nowego wymagania funkcjonalnego.
2) Uruchomić wszystkie testy i sprawdzić wynik uruchomienia,
3) Zaimplementować na podstawie testu jedynie taką ilość kodu produkcyjnego, który pozwoli na spełnienie warunków z nowego testu, jeśli wystąpił problem z testem niezwiązany z funkcjonalnością to należy poprawić test stosując omówione rozwiązania (metoda proxy, saveArg).
4) Uruchomić wszystkie testy, w przypadku nie działania któregoś z testów, powrócić do kroku 3.
5) Uporządkować kod pod kątem zmniejszenia złożoności, poprawy czytelności, czy łatwości konserwacji.
6) Uruchomić wszystkie testy, jeśli wykryją one błędy, to należy je poprawić.
7) Jeśli są nowe wymagania funkcjonalne, to powrócić do kroku 1. w przeciwnym wypadku można zakończyć proces.

Największa zmiana schematu nastąpiła w kroku 3., gdzie został dodany warunek „jeśli wystąpił problem z testem niezwiązany z funkcjonalnością". Warunek utrudnia pisanie testów początkującym programistom, którzy będą musieli nauczyć się rozpoznawać znane problemy testów, które nie będą wynikały z implementacji funkcjonalności. Wraz z nowo dodawanymi funkcjonalnościami do kolejnych standardów języka C++, baza znanych problemów może się powiększać. Należy pamiętać, że nawet dobrze dopracowane biblioteki posiadają swoje ograniczenia, a w wielu produktach występuje problem aktualizacji funkcjonalności do nowych standardów.

Bazę znanych problemów można by zamknąć w postaci skryptu, który poprawiałby wynik działania istniejącego skryptu gmock_gen. Wtedy w przypadku wykrycia, że parametrem funkcji jest unique_ptr lub referencja do rvalue skrypt automatycznie tworzyłby metodę proxy.

Natomiast gdy wykryłby wyrażenie lambda, które należy testować korzystając z konstrukcji SaveArg w kodzie testowym, to skrypt mógłby poinformować o tym w komentarzu przed daną metodą.

Obiektem lub metodą proxy można nazwać model, w którym istnieje pośrednik, który kontroluje dostęp [19]. Zatem skrypt tworzyłby funkcję przesłaniającą funkcję z klasy bazowej. Przesłonięta funkcja wywoływała funkcję z makra MOCK_METHOD, która przyjmowałaby obsługiwany przez bibliotekę Google Mock typ. Dla typu std::unique_ptr byłby to typ TYPE*, natomiast w przypadku TYPE&& to TYPE&.

7. Wnioski

Język C++ wraz z bibliotekami Google Test oraz Mock w standardzie C++03 pozwalał na korzystanie z dobrodziejstw podejścia TDD do tworzenia oprogramowania. Pojawienie się nowszych standardów postawiło wyzwanie przed GMock'iem oraz utrudniło pracę programistom podczas prób pisania testów, które niejednokrotnie się nie kompilowały, co mogło doprowadzić do rezygnacji przez programistów z wykorzystywania funkcjonalności oferowanych przez nowe standardy.

Istniejące problemy kompatybilności standardu C++14 z biblioteką GMock można rozwiązać poprzez dostosowanie procesu TDD, gdzie należy zacząć od implementacji testu, następnie przejść do napisania fragmentów kodu jedynie w takim stopniu, by test mógł zostać spełniony. Jednak gdy wystąpią błędy należy zwrócić uwagę czy to nie jest jeden z omówionych problemów. Zastosowanie innego sposobu weryfikacji wywoływanej funkcji, przedstawionej w poprzednich rozdziałach, pozwala na wykorzystanie zaplanowanej struktury kodu produkcyjnego, a w przypadku testów na powrócenie do standardowego podejścia TDD.

W przypadku schematycznych problemów jak potrzeba tworzenia funkcji Proxy dla np. unique_ptr lub dla referencji do rvalue rozwiązaniem może być napisanie skryptu takiego jak gmock_gen, który jednak rozpoznawałby parametry i w większym stopniu dopasowywałby do nich wygląd ostatecznie wygenerowanej atrapy.

Literatura

[1] Stanisław Pietrow, https://pl.wikipedia.org/wiki/Stanis%C5%82aw_Pietrow . [10.01.2019]

[2] Northeast blackout of 2003, https://en.wikipedia.org/wiki/Northeast_blackout_of_2003 [11.01.2019]

[3] Ganesan D., Lindvall M.: An analysis of unit tests of a flight software product line. Science of Computer Programming, Volume 78, 2013.

[4] Garousi Yusifoğlu V., Amannejad Y., Betin Can A.: Software test-code engineering: A systematic mapping. Information and Software Technology, Volume 58, 2015.

[5] Martin R. C.: The Clean Coder: a code of conduct for professional programmers. Pearson Education, 2011.

[6] Langr J.: Modern C++ Programming with Test-Driven Development. The Pragmatic Programmers, 2013.

[7] Beck K., Test Driven Development: By Example. Addison-Wesley Professional, 2003.

[8] Latore R.: Effects of Developer Experience on Learning and Applying Unit Test-Driven Development. IEEE Transactions on Software Engineering, Volume 40, 2014.

[9] Runeson P.: A Survey of Unit Testing Practices. IEEE Software, volume 23, 2006.

[10] Shin Y., Choi Y., Lee W. J.: Integration testing through reusing representative unit test cases for high-confidence medical software. Computers in Biology and Medicine, Volume 43, 2013.

[11] Osherove R.: The art of Unit Testing Second Edition. Manning Publications Co., 2014.

[12] C++11 - the new ISO C++ standard, http://www.stroustrup.com/C++11FAQ.html, [11.01.2019]

[13] Josuttis N.: C++. Biblioteka standardowa. Podręcznik programisty. Wydanie II. Helion, 2014.

[14] Langr J.: Modern C++ Programming with Test-Driven Development. The Pragmatic Programmers, 2013.

[15] ISO/IEC JTC1 SC22 WG21 N4660. Programming Languages - C++.

[16] Langr J.: Modern C++ Programming with Test-Driven Development. The Pragmatic Programmers, 2013.

[17] Dokumentacja dla biblioteki Google Mock, https://github.com/google/googletest/blob/master/googlemock/docs/CookBook.md [19.01.2019]

[18] Dokumentacja dla skryptu gmock_gen.py, https://github.com/google/googletest/tree/master/googlemock/scripts/generator [13.01.2019]

[19] Freeman E, Freeman E.: Head First Design Patterns. O'Reilly, 2004.