TESTOWANIE #3

WSTRZYKIWANIE ZALEŻNOŚCI

CODERS SCHOOL

ŁUKASZ ZIOBROŃ

AGENDA

1. Wstrzykiwanie zależności
2. Atrapy (test doubles)
3. Biblioteka GMock

WSTRZYKIWANIE ZALEŻNOŚCI

DEPENDENCY INJECTION (DI)

PROBLEM #1 - POŁĄCZENIA SIECIOWE

Jak przetestować funkcję, która wysyła dane przez sieć?

```
size_t send(int socket, const void *buffer, size_t length, int flags);
```

- Co jeśli gniazdko sieciowe (socket) nie będzie otwarte?
- Co jeśli flagi są źle ustawione?
- Co jeśli połączenie nie zostało nawiązane?
- Co jeśli połączenie zostanie zerwane przed wysłaniem danych?
- Co jeśli połączenie zostanie zerwane w trakcie wysyłania danych?
- Co jeśli nasze bufory sieciowe są zapchane i nie dostaniemy informacji o liczbie wysłanych bajtów?
- Co jeśli opóźnienia na łączu (lagi) są bardzo duże?
- Co jeśli testy wykonujemy współbieżnie i spróbujemy nawiązać na raz 1000 połączeń?

Część odpowiedzi znajdziesz w podręczniku funkcji send - `man 2 send`.

Programowanie sieciowe i rozproszone wymaga przetestowania wielu scenariuszy nietypowych z punktu widzenia zwykłego programowania.

PROBLEM #2 - BAZA DANYCH

Jak przetestować funkcję, która wysyła zapytania do bazy danych?

```
Response sendQuery(Query q);
```

- Co się stanie, jeśli serwer bazy danych nie będzie aktywny?
  - Nie będzie połączenia z serwerem bazy danych. Test nie przejdzie. Trzeba pamiętać, aby serwer był zawsze aktywny przed uruchomieniem testów.
- Co się stanie, jeśli serwer bazy danych jest na innej maszynie?
  - Możemy nie mieć połączenia z serwerem bazy danych. Test nie przejdzie. Trzeba pamiętać, aby mieć zawsze łączność z serwerem.
- Co się stanie, jeśli serwer bazy danych jest obciążony?
  - Odpowiedź zostanie zwrócona po długim czasie. Testy będą wykonywać się bardzo długo lub nie będą przechodzić z powodu zbyt długiego czasu oczekiwania.
- Co się stanie, jeśli baza przechowuje terabajty danych?
  - Wyszukiwanie danych może trwać zbyt długo. Testy mogą nie przechodzić.

PROBLEM 3 - FUNKCJA Z TIMEOUTEM

Jak przetestować funkcję, która czeka na wykonanie zdalnej procedury przez określony czas, po czym zgłasza błąd?

```
template <typename T>
Response executeRemotely(IpAddr address, T function, Timeout timeout);
```

- Czy test jednostkowy wykonujący się 2 minuty jest w porządku?
  - Nie jest. UT-ki powinny dawać programiście praktycznie natychmiastową odpowiedź. Inne rodzaje testów (systemowe, integracyjne) mogą trwać odpowiednio dłużej.
- Co się stanie jeśli timeout ustawimy na 0?
  - Funkcja natychmiastowo się zakończy czy będzie czekać w nieskończoność i zawieszą nam się testy?
- Czy zmiana timeoutu na 1ms na potrzeby testu będzie w porządku?
  - A co jeśli maszyna będzie akurat obciążona i test potrwa 2ms? Czy test powinien zostać zgłoszony jako FAILED?

ODCINANIE I ZMIANA ZALEŻNOŚCI

Aby przeprowadzić skuteczne testy dla wspomnianych problemów należy odciąć wszelkie zależności od czasu, baz danych czy operacji sieciowych.

Tego typu operacje powinny być pozamykane w oddzielne klasy, które podczas testów zastąpimy czymś, co nie będzie sprawiać problemów. Przykładowo możemy użyć specjalnych wersji wyżej wspomnianych funkcji.

```
size_t send(int socket, const void *buffer, size_t length, int flags) {
    return 10;
}

Response sendQuery(Request r) {
    return Response{r};
}

template <typename T>
Response executeRemotely(IpAddr address, T function, Timeout timeout) {
    if (timeout == 0) {
        throw RemoteExecutionTimeout{};
    } else {
        return Response{};
    }
}
```

WSTRZYKIWANIE ZALEŻNOŚCI

Technika, w której zamiast faktycznych obiektów wstawiamy do testów nasze własne implementacje w celu pozbycia się problemów lub uproszczenia testów nazywa się **wstrzykiwaniem zależności (ang. dependency injection)**.

Istnieje kilka różnych jej typów, ale najpopularniejszą jest wykorzystanie dynamicznego polimorfizmu w celu podmiany zależności. Posługując się interfejsami można podstawiać dowolne inne obiekty, które implementują dany interfejs (w C++ dziedziczą po tym interfejsie).

TESTUJEMY KLASĘ PLAYER

```
class Player {
    std::vector<std::shared_ptr<Weapon>> weapons_;
    Weapon& currentWeapon_;
    // ...
public:
    Player(const std::shared_ptr<Weapon> & defaultWeapon)
        : weapons_.emplace_back(defaultWeapon)
        , currentWeapon_(*defaultWeapon)
    {}
    void shoot() { currentWeapon_.shoot(); }
    void reload() { currentWeapon_.reload(); }
    // ...
};

class Weapon {
public:
    virtual void shoot() = 0;
    virtual void reload() = 0;
    virtual ~Weapon() = default;
};
```

UŻYCIE KLASY PLAYER W KODZIE

```
auto ak47 = std::make_shared<Machinegun>();
Player grzesiek{ak47};
grzesiek.shoot();
// shoot 40 times
grzesiek.reload();
// no possibility to check if mp5 has shoot, no such interface
```

UŻYCIE W TESTACH

Testujemy klasę `Player`, wycinamy zależność - `Machinegun`. Klasa `Machinegun` nie ma możliwości sprawdzenia czy strzeliliśmy, bo jej interfejs nie udostępnia sprawdzania pojemności magazynka.

```
class TestWeapon : public Weapon {
public:
    void shoot() override { hasShoot = true; }
    void reload() override { hasShoot = false; }
    bool hasShoot = false;
};
```

Klasa `TestWeapon` posiada publiczne pole mówiące o tym, czy broń wystrzeliła. Dziedziczy ona po interfejsie `Weapon`, więc możemy jej użyć zamiast prawdziwej broni w testach.

```
TEST(PlayerTest, shootingAndReloadingWeapon) {
    auto testWeapon = std::make_shared<TestWeapon>;
    Player testPayer{testWeapon};
    EXPECT_FALSE(testWeapon.hasShoot);
    testPlayer.shoot();
    EXPECT_TRUE(testWeapon.hasShoot);
    testPlayer.reload();
    EXPECT_FALSE(testWeapon.hasShoot);
}
```

Jeśli klasa `Player` poprawnie używa klasy `Weapon` to test powinien przejść.

WSTRZYKIWANIE ZALEŻNOŚCI Z UŻYCIEM INTERFEJSÓW

Wstrzykiwanie zależności pozwala uprościć kod testów. Nie musimy np. wywołać metody `shoot()` 40 razy aby rozładować magazynek. Gdyby klasa `Machinegun` potrzebowała w konstruktorze obiektu klasy `Magazine`, a klasa `Magazine` 40 obiektów klasy `Bullet`, to w testach również trzeba byłoby je utworzyć.

Pisanie własnej klasy `TestWeapon` to również pewien narzut, ale frameworki do wstrzykiwania zależności pozwalają nam to zrobić nawet w jednej linijce kodu.

CO WYCINAĆ?

W **testach jednostkowych** wszystko poza testowaną klasą.

W **testach modułowych** wszystko poza testowanym modułem (kilka, kilkanaście klas realizujących jedną większą funkcjonalność).

W **testach systemowych** zewnętrzne systemy - bazy danych, sieć, zależności czasowe. Testujemy działanie całego systemu.

W **testach integracyjnych** nic. Testujemy współdziałanie systemu z prawdziwymi zależnościami.

INNE TECHNIKI

- split backend - wstrzykiwanie zależności przez linkowanie z inną implementacją. Należy w tym celu odpowiednio skonfigurować system budowania (Makefile, CMakeLists.txt), aby podczas budowania testów kompilował i linkował z podmienionymi implementacjami.
- split header - wstrzykiwanie zależności przez includowanie innych plików nagłówkowych do testów.
- wstrzykiwanie przez settery - mało eleganckie, w szczególności gdy musimy napisać setter specjalnie do testów.

ATRAPY

(ANG. TEST DOUBLES)

RODZAJE ATRAP

- dummy
- fake
- stub
- spy
- mock

DUMMY

- najprostsza atrapa
- nie robi nic, puste funkcje
- ma za zadanie tylko spełnić wymagania sygnatur funkcji i może nie być w ogóle używany

ORYGINALNE FUNKCJE

```
double Car::accelerate(int) { /* complicated implementation */ }
void sayHello(std::string name) { std::cout << "Hello " << name << '\n'; }
```

IMPLEMENTACJA DUMMY

```
double DummyCar::accelerate(int) { return 0.0; }
void dummyHello(std::string name) {}
```

STUB

- trochę bardziej zaawansowana atrapa
- może mieć logikę lub minimalną implementację
- zwraca zdefiniowane przez nas wartości

ORYGINALNE FUNKCJE

```
double Car::accelerate(int) { /* complicated implementation */ }
void sayHello(std::string name) { std::cout << "Hello " << name << '\n'; }
```

IMPLEMENTACJA STUBA

```
double StubCar::accelerate(int value) {
    return value < 0 ? -10.0 : 10.0;
}
void stubHello(std::string name) {
    if (name == "anonymous") {
        throw std::logic_error("anonymous not allowed");
    }
}
```

FAKE

- modeluje bardziej złożone interakcje niż stub
- w praktyce w C++ nawet się go nie wyróżnia jako oddzielny twór
- często fake == stub

Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).

MOCK

- weryfikuje zachowanie testowanego obiektu
- sprawdza czy wywołał on na mocku oczekiwane funkcje w oczekiwany sposób

ORYGINALNA FUNKCJA

```
double Car::accelerate(int) { /* complicated implementation */ }
```

IMPLEMENTACJA MOCKA

```
class MockCar : public Car {
    MOCK_METHOD(double, accelerate, (int), (override));
};
```

Mocks are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.

MOCKOWANIE FUNKCJI SPOZA KLASY

- Zazwyczaj używa się stubów lub dummy
- Można użyć GMock, ale komplikuje to kod

Zobacz GoogleMock CookBook

SPY

- mock, który dodatkowo liczy ilość wywołań funkcji
- w GMock spy == mock

Spies are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.

W SKRÓCIE

- Dummy
  - Jego celem jest “zaspokojenie sygnatury”.
  - Nie są wykonywane na nim żadne interakcje.
- Stub
  - Minimalna implementacja do interakcji między obiektami.
  - Metody void są puste.
  - Wartości zwracane przez składowe są domyślne dla danego typu lub zdefiniowane (“hard-coded”).
- Fake
  - Zawiera implementację logiki, która imituje kod produkcyjny, ale w możliwie najprostszy sposób.
- Mock
  - Weryfikuje zachowanie poprzez rejestrowanie czy dana interakcja została wykonana.
- Spy
  - Weryfikuje zachowanie poprzez rejestrowanie ilości wykonanych interakcji.

Źródło: Kurs TDD cz. 19

W PRAKTYCE

W C++ spotkamy się z takim uproszczeniem:

- dummy ~= stub
  - atrapa bez logiki
- stub == fake
  - atrapa z logiką
- mock == spy
  - atrapa z rejestrowaniem akcji

Obiekty typu dummy i stub implementujemy sami. Są one tak proste, że nie potrzeba do nich żadnych frameworków.

Do obiektów typu mock najczęściej wykorzystuje się bibliotekę GMock.

Zobacz przykłady atrap w repo Pizzas

GMOCK

GMOCK - DOKUMENTACJA

- GMock for Dummies
- GMock cheatsheet
- GMock cookbook

TWORZENIE MOCKÓW

1. Pisanie z palca
2. Użycie skryptu generującego

```
class Foo {
  ...
  virtual ~Foo();
  virtual int GetSize() const = 0;
  virtual string Describe(const char* name) = 0;
  virtual string Describe(int type) = 0;
  virtual bool Process(Bar elem, int count) = 0;
};
```

```
#include "gmock/gmock.h"

class MockFoo : public Foo {
  ...
  MOCK_METHOD(int, GetSize, (), (const, override));
  MOCK_METHOD(string, Describe, (const char* name), (override));
  MOCK_METHOD(string, Describe, (int type), (override));
  MOCK_METHOD(bool, Process, (Bar elem, int count), (override));
};
```

USTAWIANIE OCZEKIWAŃ

```
MockFoo foo;
// ...
EXPECT_CALL(foo, Describe(5))
      .Times(3)
      .WillRepeatedly(Return("Category 5"));
```

Oczekujemy, że na `foo` zostanie 3-krotnie zawołana funkcja `Describe` z parametrem `5` i ustawiamy, że każdorazowo zwróci ona `std::string` o treści "Category 5".

```
EXPECT_CALL(foo, Process(_, 10))
      .WillOnce(Return(true))
      .WillOnce(Return(false));
```

Oczekujemy, że na `foo` zostanie zawołana funkcja `Process` z obojętnie jakim pierwszym parametrem i 10 jako drugi parametr. Za pierwszym razem zwróci true, za drugim false.

Więcej możliwości w GoogleMock CheatSheet

DEKORATORY MOCKÓW

NICE MOCK

```
NiceMock<MockFoo> nice_foo;      // The type is a subclass of MockFoo.
```

Ignoruje dodatkowe funkcje, które zostały wywołane oprócz funkcji oczekiwanych.

NAGGY MOCK (DOMYŚLNY)

```
MockFoo foo;
NaggyMock<MockFoo> naggy_foo;    // The type is a subclass of MockFoo.
```

Wyświetla ostrzeżenia o dodatkowych funkcjach, które zostały wywołane oprócz tych oczekiwanych.

STRICT MOCK

```
StrictMock<MockFoo> strict_foo;  // The type is a subclass of MockFoo.
```

Wywołania dodatkowych funkcji poza oczekiwanymi są traktowane jak błędy, przez które test nie przechodzi.

ZADANIE

REPO PIZZAS

0. Popraw błędy w programie, aby testy przechodziły
1. Popraw klasę PizzaMock, aby była napisana z użyciem nowszej wersji GMocka.
2. Usuń zależność od czasu w testach za pomocą dummy lub stuba.

1. USUŃ ZALEŻNOŚĆ OD CZASU W TESTACH

W KTÓRYM MIEJSCU UZALEŻNIAMY SIĘ OD CZASU?

`Pizzeria.cpp:52` `std::this_thread::sleep_for(pizza->getBakingTime());`

W JAKI SPOSÓB MOŻNA WYCIĄĆ TĘ ZALEŻNOŚĆ?

Należy utworzyć interfejs, po którym będą dziedziczyć 2 implementacje.

- Pierwsza z nich będzie faktycznie wywoływać obecną funkcję `std::this_thread::sleep_for()` i będzie używana w zwykłej binarce.
- Druga z nich nie będzie nic robiła i będzie używana w testach.

W JAKI SPOSÓB WSTRZYKIWAĆ TE ZALEŻNOŚCI?

- W konstruktorze klasy `Pizzeria` należy dodać dodatkowy argument - wskaźnik/referencję do interfejsu.
- Klasa `Pizzeria` powinna posiadać nowe pole - również wskaźnik/referencję do interfejsu.
- W miejscu dawnego wywołania funkcji `std::this_thread::sleep_for()` należy wywołać odpowiednią funkcję z interfejsu.

PRACA DOMOWA

POST-WORK

REPO PIZZAS

1. (10 XP) Napisz klasę TimeMock, którą zastąpisz dummy/stuba napisanego podczas zajęć. Używaj go poprzez StrictMock. Dodaj właściwe `EXPECT_CALL` w kodzie testów.
2. (15 XP) Dopisz nowy przypadek testowy, w którym powielasz działanie funkcji main.cpp, ale zamawiasz 3 pizze: `StubPizza` oraz 2 różne `MockPizza` (różne nazwy, ceny i czas pieczenia; jedna jako StrictMock, druga jako NiceMock). Ustaw właściwe `EXPECT_CALL`.
3. (5 XP) Utwórz własny plik .github/workflows/module3.yml, który spowoduje, że GitHub automatycznie uruchomi testy.

+3 XP za każdy z 3 punktów dostarczony do 20.09 23:59.

Możecie pracować w parach 🙂

PRE-WORK

- Znajdź i przeczytaj czym są:
  - RAII
  - wycieki pamięci i jak je wykrywać
  - wyjątki i jak ich używać w C++

CODERS SCHOOL