Zaawansowane testowanie jednostkowe – od podstaw do wzorca Test Routine

Szkolenie

BIO – mgr inż. Maksymilian Piechota

- Programista Systemów Webowych
- Lider Zespołu Technicznego
- Pasjonat Testowania

Maksymilian Piechota LinkedIn i Substack

Dla kogo

- **Programistów**
- **Inżynierów QA**
- **Architektów**
- **Studentów IT**

Cele

- **Poznanie narzędzia do budowania niezawodnych testów jednostkowych**

Plan

- Wstęp, teoria
- Ćwiczenie 1
- Omówienie rozwiązań i problemów
- Wprowadzenie wzorca Test Routine
- Ćwiczenie 2
- Podsumowanie i Q&A

Testy jednostkowe - definicja

- **Testy jednostkowe (ang. unit tests) to automatyczne testy sprawdzające poprawność działania najmniejszej, logicznie odrębnej części programu — zwykle pojedynczej funkcji, metody lub klasy.**

Kluczowe cechy testów jednostkowych:

- **Izolacja – testowana jednostka jest odseparowana od innych części systemu (np. przez użycie mocków, stubów lub fake’ów).**
- **Szybkość – testy uruchamiają się bardzo szybko, dzięki czemu można je wykonywać po każdej zmianie w kodzie.**
- **Powtarzalność – wynik testu jest zawsze taki sam przy tych samych danych wejściowych.**

Cel testów jednostkowych:

- **Ułatwiają wczesne wykrywanie błędów.**
- **Dają szybką informację zwrotną o wpływie zmian w kodzie.**
- **Pomagają utrzymać wysoką jakość i stabilność kodu.**
- **Umożliwiają refaktoryzację bez strachu przed wprowadzeniem regresji.**

Test Driven Development

- **programowanie sterowane testami, to metodyka tworzenia oprogramowania, w której najpierw pisze się testy jednostkowe, a dopiero potem kod produkcyjny, który te testy ma przechodzić.**

Cykl TDD

- **Red – napisz test, który nie przechodzi, bo kod jeszcze nie istnieje.**
- **Green – napisz minimalny kod, który pozwoli testowi przejść.**
- **Refactor – ulepsz kod, zachowując zielony stan testów.**

Cele TDD

- **Zapewnić wysoką jakość i prostotę kodu.**
- **Umożliwić bezpieczną refaktoryzację dzięki sieci testów.**
- **Wymusić dokładne zrozumienie wymagań przed implementacją.**
- **Skrócić czas uzyskania informacji zwrotnej o błędach.**

Efekt uboczny: Po zakończeniu procesu mamy zarówno działający kod, jak i zestaw automatycznych testów jednostkowych, które można uruchamiać przy każdej zmianie w projekcie.

Ćwiczenie 1

- **Napisz prosty mechanizm rejestracji użytkownika i pokryj funkcjonalność testem jednostkowym**
- **https://github.com/makspiechota/advanced-unit-testing-test-routine/blob/main/training/README.md**

Problemy z testowaniem

- Testowanie zbyt wielu rzeczy naraz
- Brak izolacji (testy zależne od innych komponentów)
- Testy oparte na implementacji, a nie na zachowaniu
- Nieczytelność testów
- Testy są zbyt wolne
- Flaky tests (niestabilne testy)
- Brak pokrycia przypadków brzegowych

Problemy z mockowaniem

- **Implementacja mocków “ad hoc” w test casach nie daje gwarancji, że mock ten poprawnie symuluje zachowanie zależności**
- **Dodatkowo jeśli zachowanie zależności się zmieni, zależne testy, które zamockowały to zachowanie u siebie, nie dowiedzą się o tym**

Czym jest Test Routine

- **Test Routine to sposób pisania testów, w którym definiujesz zestaw testów opisujących zachowanie jakiegoś komponentu (np. repozytorium użytkowników, modułu płatności, itp.) jako funkcję testową.**
- **Ta funkcja przyjmuje instancję komponentu i uruchamia zawsze ten sam zestaw testów — niezależnie od tego, jak komponent jest zaimplementowany (np. w pamięci, na bazie danych, przez sieć itd.).**
- **Dzięki temu raz definiujesz kontrakt zachowania, a potem możesz wielokrotnie sprawdzać, czy różne wersje Twojego komponentu działają tak samo.**

Test Routine: krok po kroku

- **Zacznij od zachowania (kontraktu) i napisz Test Routine**
  - Utwórz funkcję testową opisującą kontrakt (zachowania).
  - To Twoje „źródło prawdy” o tym, jak system ma się zachowywać.
  - Funkcja przyjmuje obiekt (jeszcze nieistniejący interfejs!) i uruchamia testy:

```
export function testUserRepositoryRoutine(repo: IRepo) {
  describe('UserRepository behavior', () => {
    it('should create user with valid data', async () => { /* ... */ });
    it('should reject duplicate email', async () => { /* ... */ });
  });
}
```

Test Routine: krok po kroku

- **Zdefiniuj interfejs na podstawie testu**
- **Zorkiestruj test jednostkowy**

```
describe('UserRepositoryStub', () => {
  const repo = new UserRepositoryStub();
  testUserRepositoryRoutine(repo);
});
```

- **Zaimplementuj najprostszy możliwy stub in-memory**
- **Uruchamiaj test w trakcie implementacji**

Test Routine: krok po kroku

- **Zorkiestruj test integracyjny**

```
describe('UserRepository', () => {
  const repo = new UserRepository();
  testUserRepositoryRoutine(repo);
});
```

- **Zaimplementuj kod prawdziwej zależności (baza danych, serwis HTTP)**
- **Uruchamiaj test w trakcie implementacji**

Test Routine: krok po kroku

- **Refaktoruj i utrzymuj kontrakt jako źródło prawdy**
  - Gdy zmieniają się wymagania, aktualizujesz tylko test routine.
  - Wszystkie implementacje automatycznie muszą spełniać nowy kontrakt.

Rekomendowana struktura - Port&Adapter

```
src/
├── application/ports/{port-name}/
│   ├── {port-name}.interface.ts         # Port interface
│   ├── __tests__/
│   │   └── {port-name}.test-routine.ts   # Shared test routine
│   └── stub/
│       ├── {port-name}.stub.ts          # In-memory implementation
│       └── __tests__/
│           └── {port-name}.unit-test.ts  # UNIT test orchestrator
│
└── infrastructure/adapters/{adapter-name}/
    ├── {adapter-name}.adapter.ts        # Real implementation
    └── __tests__/
        └── {adapter-name}.integration-test.ts  # INTEGRATION test orchestrator
```

Ćwiczenie 2

- **Przepisz swoje testy zgodnie ze wzorcem Test Routine**
  - Zaimplementuj Port + Test Routine dla każdej zewnętrznej zależności
    - Zorkiestruj Test Routine dla zewnętrznej zależności (test integracyjny)
    - Zorkiestruj Test Routine dla stuba in-memory
  - Napisz Test Routine dla metody rejestracji użytkowników
    - Zorkiestruj Test Routine używając zaimplementowanych in-memory stubów (unit test)
    - Opcjonalnie zorkiestruj Test Routine używając prawdziwych adapterów (integration test)

Problemy z testowaniem – rozwiązanie Test Routine

- **Testowanie zbyt wielu rzeczy naraz**
  - Test Routine skupia się na jednym konkretnym komponencie i jego zachowaniu.
- **Brak izolacji (testy zależne od innych komponentów)**
  - Routine pozwala uruchamiać te same testy zarówno na prostych, w pełni izolowanych wersjach komponentu (np. działających w pamięci), jak i na ich rzeczywistych odpowiednikach
- **Testy oparte na implementacji, a nie na zachowaniu**
  - W Routine definiujesz testy w formie kontraktu zachowania — testujesz co komponent ma robić, a nie jak.
- **Nieczytelność testów**
  - Kolejny warsztat :)

Problemy z testowaniem – rozwiązanie Test Routine

- **Testy są zbyt wolne**
  - Dzięki podziałowi na różne uruchomienia (np. wersja w pamięci vs. wersja z bazą danych), możesz testować szybko logikę, a wolniejsze testy uruchamiać tylko wtedy, gdy to potrzebne.
- **Flaky tests (niestabilne testy)**
  - Test Routine pozwala najpierw sprawdzić logikę w środowisku w pełni deterministycznym (bez sieci, bez bazy, bez czasu).
- **Brak pokrycia przypadków brzegowych**
  - Ponieważ Routine jest jednym, centralnym zestawem testów, każdy przypadek brzegowy opisany w kontrakcie obowiązuje wszystkie wersje komponentu.
  - + kolejny warsztat ;)

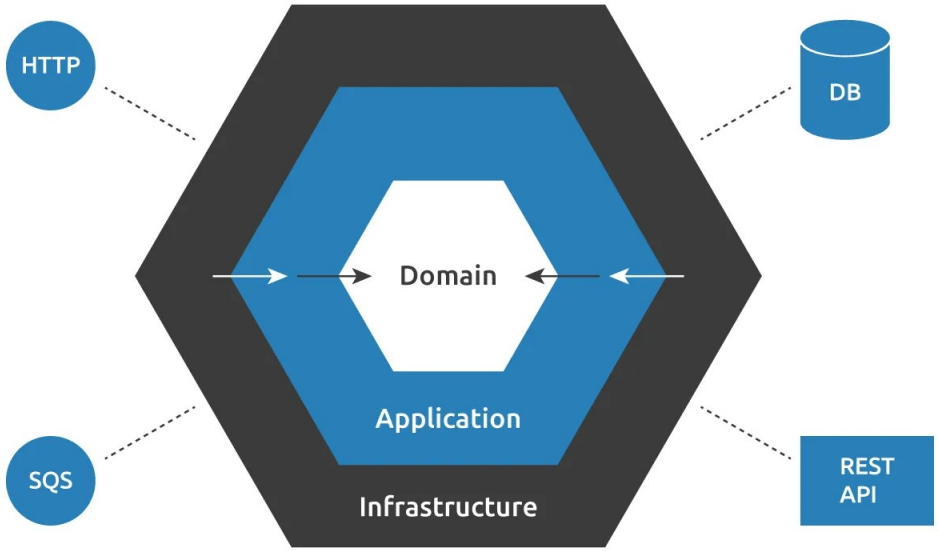
Problemy z mockowaniem

- Stub, poprzez wykorzystanie tego samego Test Routine, gwarantuje idealne* odwzorowanie zachowania realnego adapteru
- Zmiana kontraktu lub zachowania komponentu, poprzez sprzężenie przez Test Routine wymusza zaktualizowanie zachowania stuba
- Testy komponentów zależne, wykorzystujące tego stuba, z automatu otrzymują zaktualizowane zachowanie i muszą się dostosować

Automatycznie wykorzystane wzorce

- **Port&Adapter (Architektura Hexagonalna)**
- **Dependency Injection**

Architektura Hexagonalna



Maksymilian Piechota LinkedIn i Substack

Zaawansowane testowanie jednostkowe – od podstaw do wzorca Test Routine

Q&A