# Testowanie - Przypomnienie i rozszerzenie wiedzy

## Spis treści



## Wstęp

> Jak zabezpieczyć swój własny ~~t%$@k~~ interes podczas pisania kodu
>
> — *Karol Rogowski*

W tym warsztacie przyjrzymy się dokładniej tematyce testowania aplikacji. Wprowadzenie do testowania aplikacji było już poruszone wcześniej. Wtedy nie poruszyliśmy natomiast kwestii związanych z testowaniem klas ze wstrzykniętymi zależnościami. Wspomnieliśmy tam natomiast o tej tematyce. W tej części pójdziemy o krok dalej i wyjaśnimy dodatkowe zagadnienia dotyczące testów jednostkowych w kontekście Springa. Przyjrzymy się przydatnym rozwiązaniom i koncepcjom, które istnieją w świecie Javy jako jej istotny element. Spojrzymy na testy z punktu widzenia aplikacji Springowych, a przykładach będziemy się skupiać na frameworku **JUnit** oraz wprowadzimy bibliotekę **Mockito**.

## Przypomnienie o testach

### Czym jest testowanie? Po co się testuje?

Ogólnie testowanie jest to proces, który sprawdza, czy oprogramowanie spełnia założone wymagania, dodatkowo wykrywając w nim błędy.

W praktyce testy to oprogramowanie, które wywołuje jakiś fragment programu i sprawdza, czy zachowuje się on w określony sposób. Testy istnieją po to, aby móc automatycznie zapewnić nas, że napisany przez nas program zachowuje się zgodnie z oczekiwaniami. Testy w procesie wytwarzania oprogramowania powinny być wykonywane wcześnie, aby jak najszybciej zasygnalizować, że coś przestało działać. Ważnym wskaźnikiem testów jest poziom pokrycia kodu testami (**code coverage**), którego wysoki poziom zapewnia nas o poprawnym działaniu testowanego systemu. Opisana tematyka była już poruszana wcześniej.

# Test Driven Development, TDD

**TDD** jest podejściem do programowania, w którym piszemy kod funkcjonalności biznesowych na podstawie wcześniej zdefiniowanych przypadków testowych. W praktyce sprowadza się to do tego, że zanim napiszemy kod, piszemy testy. Początkowo testy takie są wszystkie na czerwono, ale stopniowo jak będziemy definiować funkcjonalność, testy zaczynają świecić się na zielono, aż osiągniemy stan, gdzie wszystkie testy przechodzą.


*Obraz 1. Cykl Test Driven Development*

W tym podejściu development zaczyna się od wymyślenia jakie funkcjonalności mają być dostarczone z perspektywy podziału na metody testowe. Inaczej mówiąc, pisząc testy, możemy zdefiniować wymagania, jakie ma spełniać nasz kod. Następnie, implementujemy takie wymagania, faktycznie ten kod pisząc.

Jeżeli rozdzielimy pisanie programu na kroki, pisalibyśmy program w taki sposób:

- Dodaj test,
- Uruchom wszystkie testy i zobacz czy któryś nie przechodzi,
- Dopisz kod, aby test przechodził,
- Uruchom testy i jeżeli wszystkie przechodzą, możesz refactorować swój kod,
- Powtórz.

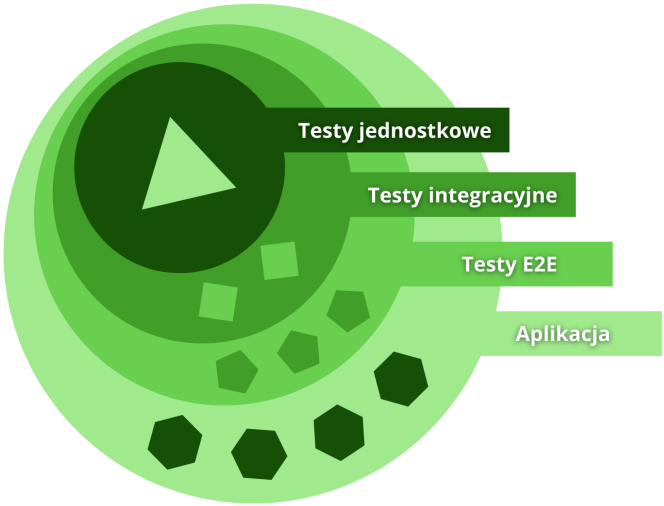
# Rodzaje testów

- Testy **jednostkowe** (*unit tests*) - pisane przez programistę, są szybkie i najtańsze w utrzymaniu, dotyczą małych, odizolowanych funkcjonalności. Pisząc te testy szukamy błędów implementacyjnych. Zapewniają one bezpieczeństwo w przypadku wprowadzania kolejnych zmian w aplikacji,
- Testy **integracyjne** (*integration tests*) - testują zachowanie komponentów, na styku których następuje integracja. Komponenty są testowane w grupie odpowiedzialnej za jakąś funkcjonalność. Takie testy trwają odpowiednio dłużej. Sprawdzają poprawne działanie komunikacji pomiędzy komponentami. Są też nazywane testami komponentów lub testami podsystemów,
    - Przykładem testu integracyjnego może być kod komunikujący się z bazą danych. W takim teście

nie potrzeba w pełni działającej bazy, a wystarczy jedynie jej atrapa. Zamiast angażować jakiś ciężki i zaawansowany mechanizm można skorzystać z czegoś lekkiego, co będzie taką bazę imitować. Takie podejście ma swoje zalety i wady. Będziemy o tym rozmawiać później.

    - Ten link najlepiej wyjaśnia różnicę pomiędzy testami jednostkowymi a integracyjnymi ☺,

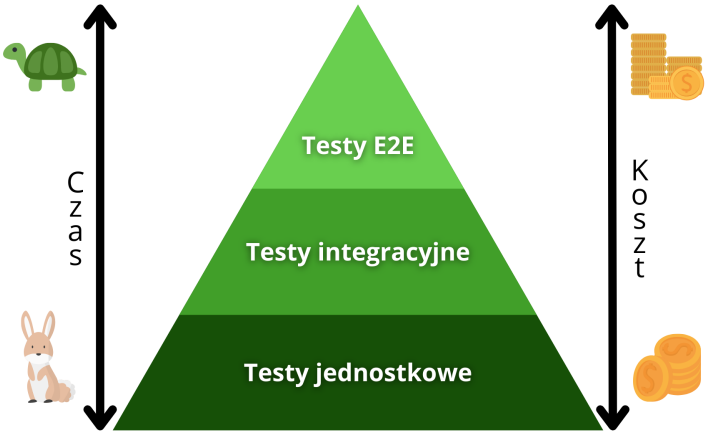
- Testy **systemowe** (*end to end, e2e*) - testy przeprowadzane na działającej aplikacji. Ich celem jest zasymulowanie prawdziwego zachowania użytkownika w systemie. Są trudniejsze, bardziej czasochłonne do zautomatyzowania i bardzo kosztowne w utrzymaniu ich prawidłowego działania ze względu na ciągły rozwój systemów. Z tego powodu często są przeprowadzanie manualnie, czyli, tester klika w systemie, sprawdzając poszczególne procesy biznesowe.




*Obraz 2. Rodzaje testów*

# Kosztowność testów

Rodzaje testów można zobrazować w postaci piramidy.

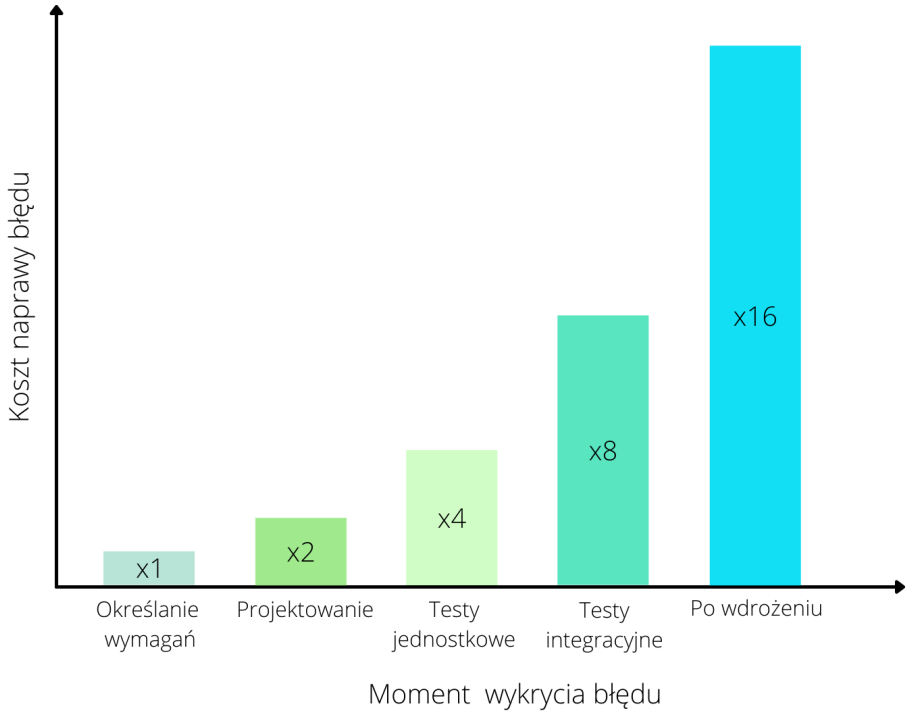



*Obraz 3. Piramida testów*

Widać na niej podstawowe wady i zalety poszczególnych testów. Im niższy poziom piramidy tym testów powinno być więcej oraz testy powinny sprawdzać mniejsze obszary systemu, ponieważ są tańsze, krócej się wykonują i szybciej zwracają wyniki. Z drugiej strony, im poziom jest wyższy, tym testów jest mniej, za to sprawdzają większe części systemu. Są znacznie bardziej kosztowne i trudniejsze do przeprowadzenia, ponieważ angażują więcej zasobów, a dodatkowo trwają dłużej. Teoretycznie wszystko można by było przetestować za pomocą testów systemowych, ale zawsze jest to kompromis

między czasem, kosztem i skutecznością.

Międzynarodowa rada certyfikacji do testowania oprogramowania, czyli **ISTQB (International Software Testing Qualifications Board)** definiuje testy w inny, bardziej złożony sposób. Na nasze potrzeby wystarczy ten standardowy podział. Warto jednak wiedzieć, że wraz z postępem systemów informatycznych, obszar dziedziny testów też przeszedł znaczny rozwój.




*Obraz 4. Wzrost kosztów naprawy błędu*

*Foundations of Software Testing ISTQB Certification,*
*Dorothy Graham, Erik van Veenendaal, Rex Black*

Dlaczego warto poświęcić czas na testy? Ponieważ koszt naprawy błędu rośnie wraz z momentem jego wykrycia. Jeżeli błąd zostanie wykryty jeszcze przed implementacją, w momencie definiowania wymagań, taki koszt jest najniższy. Jeżeli ten sam błąd znajdziemy już po wdrożeniu aplikacji, wtedy nie tylko trzeba tłumaczyć się przed klientem, ale też musimy wrócić do kodu, znaleźć przyczynę i naprawić błąd, a następne przejść od początku cały proces testów i wdrażania aplikacji. Testy jednostkowe to pierwszy krok, w którym można testy sformalizować w postaci zautomatyzowanej. To też najlepszy moment do wykrycia jak największej ilość błędów. Bo jak Ty je znajdziesz, to nikt się o nich nie dowie ☺

# Kolejne definicje

W kwestii definicji z jakimi możemy się spotkać w praktyce. **SUT** (ang. *System Under Test*) - określenie dotyczące tego, co jest testowane. Dla testów jednostkowych może to być klasa (**CUT**, ang. *Class Under Test*), obiekt (**OUT**, ang. *Object Under Test*), czy metoda (**MUT**, ang. *Method Under Test*). Dla testów klienckich może to być cała aplikacja (**AUT**, ang. *Application Under Test*) albo jakaś jej część. Zaznaczam jednak, że możemy się z tymi stwierdzeniami spotkać albo i nie ☺.