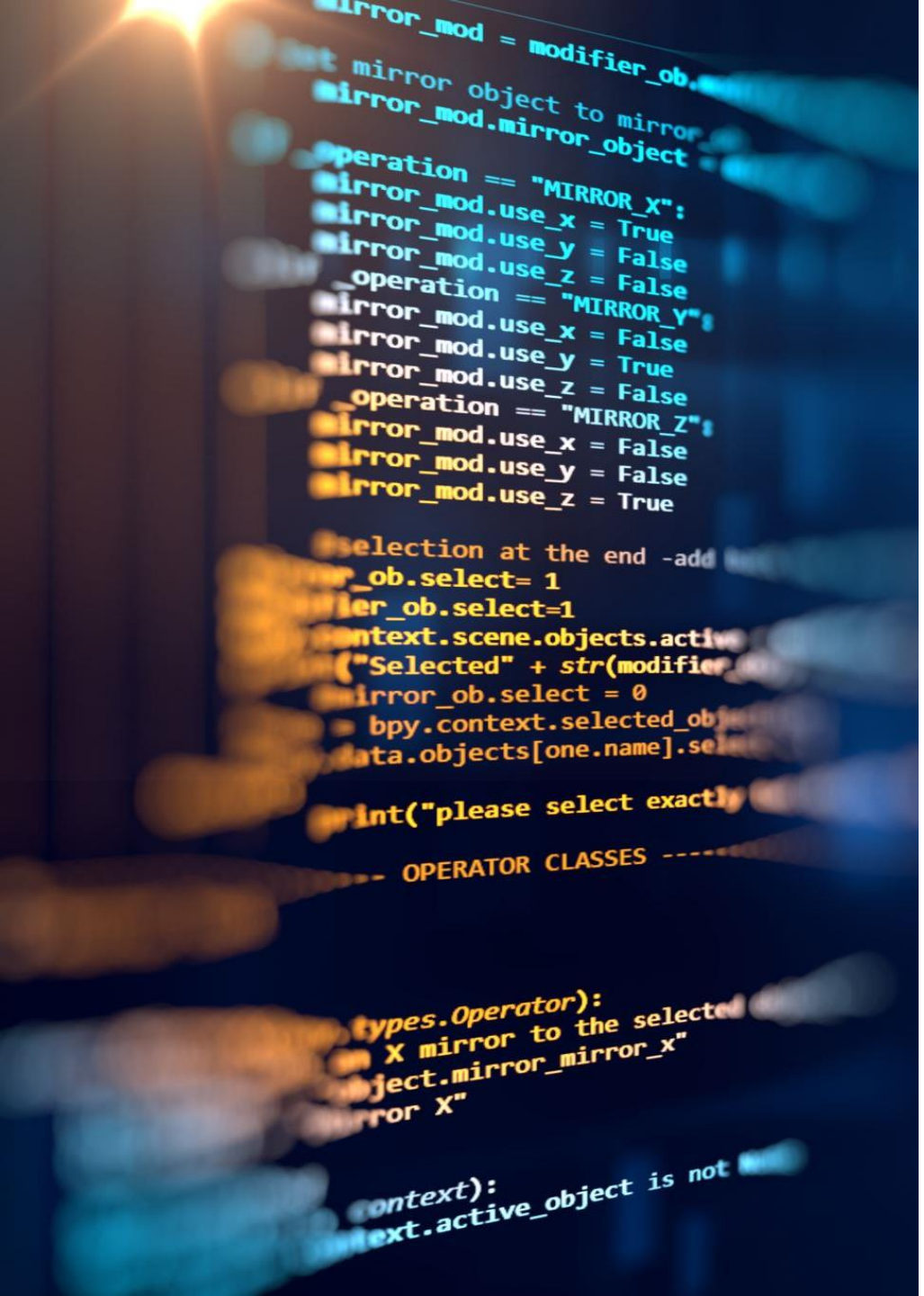
NEYGADB – testowanie i jakość kodu



Dlaczego jakość kodu ma znaczenie?

- **Utrzymanie oprogramowania to większość pracy** – statystyki pokazują, że nawet 70–80% kosztów projektu to utrzymanie, nie początkowy rozwój.
- **Czytelny i testowalny kod = szybsze naprawianie błędów i dodawanie nowych funkcji**.
- **Zły kod spowalnia zespół** – tzw. „dług technologiczny” sprawia, że nowe zmiany są trudne do wprowadzenia.
- **Bezpieczeństwo i stabilność** – błędy nie tylko przeszkadzają, ale mogą mieć poważne konsekwencje (np. wyciek danych, straty finansowe).

Zasady programowania dotyczące jakości kodu

KISS

DRY

SOLID

YAGNI

SoC

LoD

KISS (Keep It Simple, Stupid)

1

Zasada: Prosty kod jest lepszy niż skomplikowany – nawet jeśli wydaje się mniej „sprytny”.

2

Dlaczego to ważne? Mniej kodu to mniej błędów, łatwiejsze testowanie, refaktoryzacja i przyswajanie przez innych programistów.

3

Przykład: Nie twórz ogólnego frameworka do rozwiązania jednego problemu – napisz prostą funkcję.

DRY (Don’t Repeat Yourself)

1

Zasada: Unikaj powielania tego samego kodu lub logiki w wielu miejscach.

2

Dlaczego to ważne? Jeśli trzeba coś zmienić, wystarczy to zrobić w jednym miejscu.

3

Przykład: Jeśli masz kilka miejsc w kodzie, które obliczają podatek tak samo – wydziel to do jednej funkcji.

SOLID (5 zasad projektowania obiektowego)

Inicjał	Skrót	Koncepcja
S	SRP	Single responsibility principle – Klasa powinna mieć tylko jedną odpowiedzialność (nigdy nie powinien istnieć więcej niż jeden powód do modyfikacji klasy).
O	OCP	Open/closed principle – Klasy (encje) powinny być otwarte na rozszerzenia i zamknięte na modyfikacje.
L	LSP	Liskov substitution principle – Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów.
I	ISP	Interface segregation principle – Wiele dedykowanych interfejsów jest lepsze niż jeden ogólny.
D	DIP	Dependency inversion principle – Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych – zależności między nimi powinny wynikać z abstrakcji.

YAGNI (You Aren't Gonna Need It)

NIE IMPLEMENTUJ CZEGOŚ „NA ZAPAS" – TYLKO TO, CZEGO FAKTYCZNIE POTRZEBUJESZ TERAZ.

POZWALA OGRANICZYĆ NADMIAROWOŚĆ I SKUPIĆ SIĘ NA REALNYCH WYMAGANIACH.

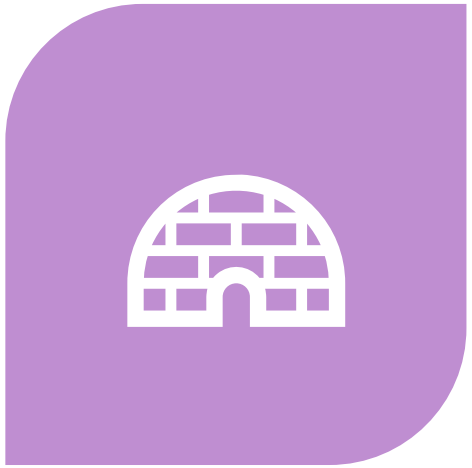
SoC (Separation of Concerns)

ROZDZIELAJ ODPOWIEDZIALNOŚCI W SYSTEMIE.

UI NIE POWINIEN „ZNAĆ" BAZY DANYCH, LOGIKA BIZNESOWA NIE POWINNA OBSŁUGIWAĆ BŁĘDÓW HTTP ITD.

Law of Demeter (LoD)

„MÓW DO SWOICH NAJBLIŻSZYCH ZNAJOMYCH, A NIE DO ZNAJOMYCH ZNAJOMYCH".



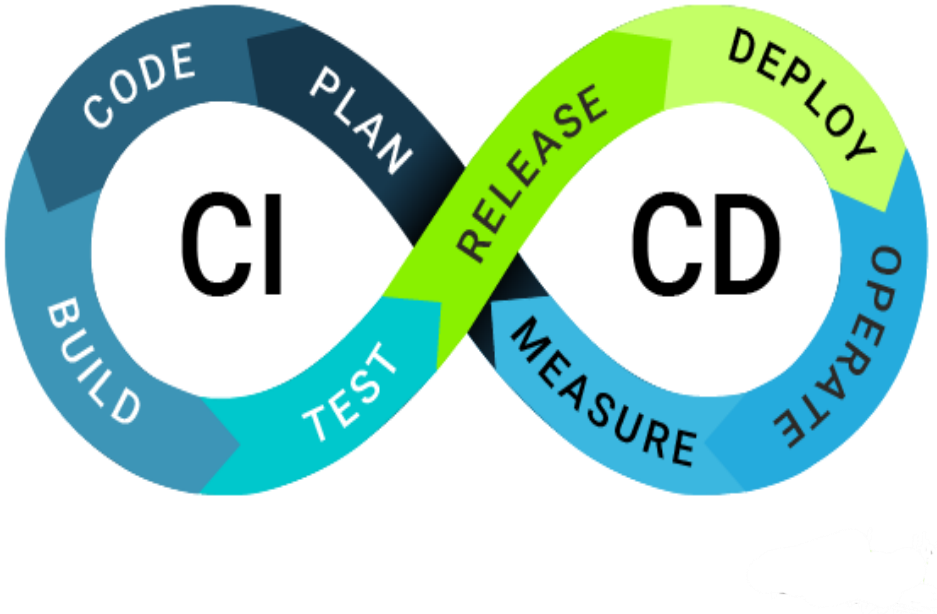
UNIKAJ WYWOŁAŃ TYPU: USER.GETADDRESS().GETCITY().GETZIPCODE() – TO ŁAMIE ENKAPSULACJĘ I ZWIĘKSZA ZALEŻNOŚCI.

Techniki zapewniania jakości kodu

1. Code review – przegląd kodu
2. Lintery i formatowanie (np. ESLint, Prettier, Checkstyle)
3. Testy automatyczne
4. Refaktoryzacja
5. Dobre praktyki projektowania (KISS, SOLID, DRY, YAGNI)
6. Komentarze i dokumentacja
7. CI/CD

CI – Continuous Integration

1. CEL: SZYBKO WYKRYWAĆ BŁĘDY, ZANIM TRAFIĄ NA PRODUKCJĘ.



Jej głównym celem jest szybkie wykrywanie błędów i konfliktów między fragmentami kodu tworzonymi przez różnych członków zespołu. Za każdym razem, gdy programista wprowadza zmiany i wysyła je do systemu kontroli wersji (np. robi git push do GitHuba), uruchamiany jest tzw. pipeline – czyli zautomatyzowany zestaw kroków. W ramach takiego procesu projekt jest budowany (np. kompilowany lub instalowany), uruchamiane są testy jednostkowe i integracyjne, a także analiza statyczna kodu (np. przez lintery). Dzięki temu już po kilku minutach wiadomo, czy nowy kod działa poprawnie i czy nie zepsuł niczego innego.

CD –Continuous Delivery/Deployment.

CEL: ZMIANY TRAFIAJĄ NA PRODUKCJĘ (LUB ŚRODOWISKO TESTOWE) AUTOMATYCZNIE I BEZPIECZNIE.

Polega na tym, że cały proces – od integracji po przygotowanie paczki do wdrożenia – jest zautomatyzowany, ale ostateczne zatwierdzenie publikacji nowej wersji pozostaje w rękach człowieka. Continuous deployment idzie o krok dalej: tutaj cały proces – łącznie z wdrożeniem zmian na serwer produkcyjny – odbywa się automatycznie, bez udziału człowieka, pod warunkiem że wcześniejsze kroki przeszły pomyślnie.

Wzorce projektowe

- Uniwersalne, sprawdzone w praktyce rozwiązania często pojawiających się, powtarzalnych problemów projektowych. Pokazują powiązania i zależności pomiędzy klasami oraz obiektami i ułatwiają tworzenie, modyfikację oraz utrzymanie kodu źródłowego. Są opisem rozwiązania, a nie jego implementacją. Wzorce projektowe stosowane są głównie w projektach wykorzystujących programowanie obiektowe. Można go traktować jako schemat lub zestaw zasad, które pomagają tworzyć kod bardziej elastyczny, zrozumiały i łatwy do utrzymania.

Rodzaje wzorców projektowych

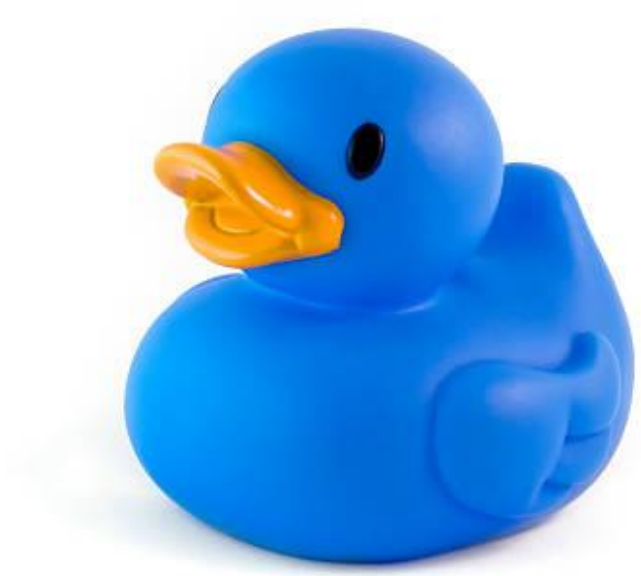
Kreacyjne – pomagają w efektywnym tworzeniu obiektów, np. Singleton (zapewnia istnienie tylko jednej instancji klasy).

Strukturalne – organizują zależności między obiektami, np. Adapter (umożliwia współpracę niekompatybilnych klas).

Behawioralne – zarządzają interakcjami między obiektami, np. Observer (pozwala obiektom reagować na zmiany w innych obiektach).

Więcej przykładów wzorców projektowych wraz z opisami:
https://refactoring.guru/design-patterns

Rodzaje testów – jak sprawdzać, czy nasz kod działa?



- Testy jednostkowe (Unit tests)
- Testy integracyjne
- Testy end-to-end (E2E)
- Testy regresji, dymne, akceptacyjne

Unit Test

```python
def add(a, b):
    return a + b

def test_add():
    assert add(2, 3) == 5
```

Co to jest?

Testy sprawdzające **pojedyncze funkcje, klasy lub metody** w izolacji – bez zależności od bazy danych, sieci, czy innych modułów.

Testujemy **jednostkę logiki**, bez otoczenia.

Są **szybkie, łatwe do napisania** i **łatwe do debugowania**.

Po co?

Wykrywają błędy wcześnie.

Idealne do TDD (Test-Driven Development).

Pomagają przy refaktoryzacji – szybko widzisz, czy coś zepsułeś.

Testy integracyjne

Testy sprawdzające, czy **różne komponenty współpracują ze sobą poprawnie** – np. czy kontroler poprawnie komunikuje się z bazą danych lub API.

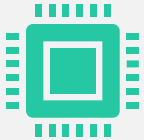
Upewniają się, że **interakcje między modułami działają**.

Często wykrywają błędy, których testy jednostkowe nie są w stanie wychwycić.

```java
@Test
public void saveUser_shouldStoreUserInDatabase() {
    userService.save(new User("Jan"));
    assertTrue(database.contains("Jan"));
}
```

Przykład testu integracyjnego

Testy end-to-end (E2E)



- Testują cały system od początku do końca — tak, jak robi to **prawdziwy użytkownik**.

- **Przykład**: Użytkownik loguje się przez interfejs przeglądarkowy. Test z użyciem narzędzia typu Cypress, Selenium, Playwright.

- Sprawdzają **realne scenariusze użytkownika**. Wykrywają błędy, które przeszły przez testy jednostkowe i integracyjne. Wymagają więcej zasobów i czasu, ale dają największe pokrycie.

Przykład testu integracyjnego

```
// Cypress
cy.visit('/login')
cy.get('input[name="email"]').type('test@example.com')
cy.get('input[name="password"]').type('123456')
cy.get('button[type="submit"]').click()
cy.contains('Welcome back!')
```

Testy regresji, dymne, akceptacyjne

1. Testy regresji (Regression Tests) – Sprawdzają, czy **zmiana kodu nie zepsuła** istniejącej funkcjonalności. Często uruchamiane automatycznie po każdym deployu/merge'u.

2. Testy dymne (Smoke Tests) – Szybkie sprawdzenie, czy „system się uruchamia" i kluczowe funkcje działają. Jak test „czy aplikacja się w ogóle odpala?".

3. Testy akceptacyjne (Acceptance Tests) – Weryfikują, czy system **spełnia wymagania biznesowe**. Często definiowane wspólnie z klientem lub Product Ownerem. Przykład: "Użytkownik może zapisać się na newsletter i dostaje maila potwierdzającego."

Filozofie tworzenia oprogramowania, w których testy nie są dodatkiem *po wszystkim*, tylko częścią procesu od samego początku.

- TDD
- BDD
- ATDD
- Property-Based Testing
- Exploratory Testing

TDD – Test-Driven Development –

metodyka programowania, w której najpierw piszemy testy, a dopiero potem kod.

Cykl TDD – „Red → Green → Refactor”:

Red – piszesz test, który *musi się nie powieść*, bo nie ma jeszcze implementacji.

Green – piszesz minimalną ilość kodu, aby test przeszedł.

Refactor – poprawiasz kod (czyścisz, upraszczasz), ale test nadal musi przechodzić.

TDD – Test-Driven Development

Zalety:

- Wymusza prostą i testowalną architekturę
- Szybko wykrywa błędyPozwala na bezpieczną refaktoryzację
- Dokumentuje wymagania

Wady:

- Na początku wolniejsze
- Trzeba umieć dobrze projektować testy
- Może prowadzić do zbyt dużej liczby mało istotnych testów

BDD – Behavior-Driven Development

- BDD to rozwinięcie TDD, w którym skupiamy się nie na „jak działa kod", ale **jak ma się zachowywać z perspektywy użytkownika lub biznesu**. Zamiast „testować funkcję X", piszemy scenariusze typu: „Jeśli użytkownik jest niezalogowany i wejdzie na /profile, to zostanie przekierowany na /login"

BDD skupia się na następujących aspektach:

- Od czego zacząć w procesie
- Co testować, a czego nie testować
- Ile można przetestować za jednym razem
- Jak nazwać testy
- Jak zrozumieć przyczynę ewentualnego niepowodzenia testu

Inne podejścia

- https://en.wikipedia.org/wiki/Acceptance_test-driven_development
- https://en.wikipedia.org/wiki/Property_testing
- https://pl.wikipedia.org/wiki/Testowanie_eksploracyjne

Jak testować w Pythonie w pytest?

1. Pip install pytest

2. Kod testów, np:

```
2 usages
def is_even(n):
    return n % 2 == 0


def test_is_even_with_even_number():
    assert is_even(2) == True


def test_is_even_with_odd_number():
    assert is_even(3) == False
```

3. W terminalu wpisać pytest plik.py

Jak testować w Pythonie w pytest?

Kilka testów w jednej funkcji:

```python
import pytest


1 usage
def is_even(n):
    return n % 2 == 0


@pytest.mark.parametrize("n,expected", [
    (2, True),
    (3, False),
    (4, True),
    (13, False),
    (1, False),
    (0, True),
    (-7, False)
])
def test_is_even(n, expected):
    assert is_even(n) == expected
```

Jak testować w Pythonie w pytest?

Testowanie czy rzucany jest wyjątek:

```python
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero!")
    return a / b



def test_divide_by_zero():
    with pytest.raises(ValueError, match="Cannot divide by zero!"):
        divide( a: 5,  b: 0)
```

Mockowanie Bazy Danych

Mockowanie bazy danych to technika używana w testach, która polega na **symulowaniu zachowania bazy danych bez faktycznego łączenia się z nią**.

- Zamiast korzystać z prawdziwej bazy (która może być wolna, trudna do przygotowania lub zawierać nieprzewidywalne dane), tworzymy **"podróbkę"** (mock), która **naśladuje jej metody i zwraca ustalone wyniki**.

Mockowanie Bazy Danych

🧩 Po co to robić?

- ✅ **Szybkość** – testy działają błyskawicznie, bez połączeń do bazy.
- ✅ **Izolacja** – testujemy tylko logikę aplikacji, a nie bazę.
- ✅ **Powtarzalność** – nie musisz przygotowywać danych za każdym razem.
- ✅ **Bezpieczeństwo** – nie ryzykujesz przypadkowego usunięcia danych produkcyjnych.

```python
from unittest.mock import Mock

def user_exists(db, email):
    return db.find_user_by_email(email) is not None

def test_user_exists():
    mock_db = Mock()
    mock_db.find_user_by_email.return_value = {"email": "a@a.com"}
    assert user_exists(mock_db, "a@a.com")
```

Mockowanie Bazy Danych

- mock_db udaje prawdziwą bazę
- find_user_by_email zwraca to, co mu każemy
- Nie potrzebujemy żadnej bazy ani danych testowych

Dziękujemy za uwagę

Źródła:

https://www.baytechconsulting.com/blog/projecting-costs-in-software-maintenance

https://pl.wikipedia.org/wiki/SOLID

https://pl.wikipedia.org/wiki/Wzorzec_projektowy_(informatyka)

https://refactoring.guru/design-patterns

https://pl.wikipedia.org/wiki/Behavior-driven_development

ChatGPT