Rozwiązanie Zadania: Testowanie Jednostkowe z PyTest dla Projektu MediScan

Jarek Hryszko

Wprowadzenie

Ten dokument zawiera szczegółowe, krok po kroku rozwiązanie zadania dotyczącego testowania jednostkowego modułu `bloodwork_calculator.py` w projekcie MediScan. Instrukcje są napisane z myślą o osobach początkujących, które nie mają doświadczenia z testami jednostkowymi, PyTest, GitHub czy programowaniem w Python.

1 Krok 1: Pobranie kodu źródłowego z GitHub

GitHub to popularna platforma, gdzie programiści przechowują i dzielą się kodem. Aby pobrać kod projektu MediScan, wykonaj następujące kroki:

1.1 Instalacja Git

Najpierw potrzebujesz zainstalować program Git, który pozwoli Ci na pobranie kodu z GitHub:

1. Przejdź na stronę `https://git-scm.com/downloads`
2. Pobierz wersję Git dla Twojego systemu operacyjnego (Windows, macOS lub Linux)
3. Zainstaluj Git, podążając za instrukcjami instalatora (możesz zaakceptować wszystkie domyślne opcje)

1.2 Klonowanie repozytorium

"Klonowanie"to pobieranie kopii projektu z GitHub na Twój komputer:

1. Otwórz terminal (w Windows możesz użyć „Wiersz polecenia", w macOS Terminal, w Linux konsola)
2. Przejdź do folderu, gdzie chcesz pobrać projekt (np. na pulpit):

```
# Na Windows (przykład):
cd C:\Users\TwojeImie\Desktop

# Na macOS/Linux (przykład):
cd ~/Desktop
```

3. Sklonuj repozytorium, wpisując:

```
git clone https://github.com/MediiScan/mediscan-project
```

4. Przejdź do pobranego folderu projektu:

```
cd mediscan-project
```

2 Krok 2: Przygotowanie środowiska Python

2.1 Instalacja Python

Jeśli nie masz jeszcze zainstalowanego Pythona:

1. Przejdź na stronę `https://www.python.org/downloads/`
2. Pobierz najnowszą wersję Pythona (3.8 lub nowszą)
3. Zainstaluj Python, upewniając się, że zaznaczasz opcję "Add Python to PATH"(dodaj Python do ścieżki systemowej)

2.2 Stworzenie wirtualnego środowiska

Wirtualne środowisko to jak oddzielny "pokój"dla projektu, gdzie instalujemy potrzebne pakiety bez wpływu na resztę systemu:

1. Otwórz terminal w folderze projektu (jeśli jeszcze nie jest otwarty)
2. Stwórz wirtualne środowisko:

```
# Na Windows:
python -m venv venv

# Na macOS/Linux:
python3 -m venv venv
```

3. Aktywuj wirtualne środowisko:

```
# Na Windows:
venv\Scripts\activate

# Na macOS/Linux:
source venv/bin/activate
```

Po aktywacji, przed linią w terminalu pojawi się "(venv)", co oznacza, że środowisko jest aktywne.

2.3 Instalacja PyTest i wymaganych pakietów

Teraz zainstalujemy PyTest i inne potrzebne pakiety:

```
pip install pytest
```

3 Krok 3: Zapoznanie się ze strukturą projektu

Przyjrzyjmy się strukturze projektu MediScan:

```
mediscan-project/
+-- mediscan/
|   +-- __init__.py
|   +-- bloodwork_calculator.py    # Moduł, który będziemy testować
```

```
|   +-- reference_values.py        # Wartości referencyjne dla krwi
|   +-- report_generator.py        # Generator raportów
+-- tests/
|   +-- __init__.py
|   +-- test_report_generator.py   # Istniejące testy
+-- requirements.txt
+-- README.md
```

Teraz otwórz i przeanalizuj plik `bloodwork_calculator.py`, aby zrozumieć, co zawiera:

```python
# Plik: mediscan/bloodwork_calculator.py

def calculate_bmi(weight_kg, height_cm):
    """
    Oblicza BMI (Body Mass Index) na podstawie wagi i wzrostu.

    Args:
        weight_kg (float): Waga w kilogramach
        height_cm (float): Wzrost w centymetrach

    Returns:
        float: Wartość BMI
    """
    height_m = height_cm / 100  # konwersja cm na m
    return weight_kg / (height_m * height_m)

def calculate_nlr(neutrophils, lymphocytes):
    """
    Oblicza stosunek neutrofili do limfocytów (NLR).
    Podwyższony NLR może wskazywać na stan zapalny.

    Args:
        neutrophils (float): Liczba neutrofili (10^9/L)
        lymphocytes (float): Liczba limfocytów (10^9/L)

    Returns:
        float: Wartość NLR
    """
    return neutrophils / lymphocytes

def categorize_bmi(bmi):
    """
    Kategoryzuje BMI zgodnie ze standardami WHO.

    Args:
        bmi (float): Wartość BMI
```

```
    Returns:
        str: Kategoria BMI
    """
    if bmi < 16.0:
        return "wygłodzenie"
    elif bmi < 17.0:
        return "wychudzenie"
    elif bmi < 18.5:
        return "niedowaga"
    elif bmi < 25.0:
        return "waga prawidłowa"
    elif bmi < 30.0:
        return "nadwaga"
    elif bmi < 35.0:
        return "otyłość I stopnia"
    elif bmi < 40.0:
        return "otyłość II stopnia"
    else:
        return "otyłość III stopnia"

def calculate_anemia_severity(hemoglobin, sex):
    """
    Określa stopień niedokrwistości na podstawie poziomu hemoglobiny.

    Args:
        hemoglobin (float): Poziom hemoglobiny (g/dL)
        sex (str): Płeć pacjenta ('M' lub 'F')

    Returns:
        str: Stopień niedokrwistości lub 'brak'
    """
    if sex == 'M':
        if hemoglobin >= 13.5:
            return "brak"
        elif hemoglobin >= 11.0:
            return "łagodna"
        elif hemoglobin >= 8.0:
            return "umiarkowana"
        else:
            return "ciężka"
    elif sex == 'F':
        if hemoglobin >= 12.0:
            return "brak"
        elif hemoglobin >= 10.0:
            return "łagodna"
        elif hemoglobin >= 8.0:
            return "umiarkowana"
```

```
        else:
            return "ciężka"
```

4 Krok 4: Planowanie testów jednostkowych

Zanim zaczniemy pisać testy, zastanówmy się, co należy przetestować:

1. **Funkcja calculate_bmi**:
   - Czy poprawnie oblicza BMI dla typowych danych?
   - Jak zachowuje się dla wartości zerowych/ujemnych?
   - Czy obsługuje wartości zmiennoprzecinkowe?
2. **Funkcja calculate_nlr**:
   - Czy poprawnie oblicza stosunek dla typowych danych?
   - Jak zachowuje się, gdy liczba limfocytów wynosi 0?
   - Jak obsługuje wartości ujemne?
3. **Funkcja categorize_bmi**:
   - Czy poprawnie kategoryzuje BMI dla różnych wartości?
   - Czy granice kategorii są prawidłowo obsługiwane?
4. **Funkcja calculate_anemia_severity**:
   - Czy poprawnie określa stopień niedokrwistości dla mężczyzn?
   - Czy poprawnie określa stopień niedokrwistości dla kobiet?
   - Jak radzi sobie z nieprawidłowymi oznaczeniami płci?

5 Krok 5: Tworzenie pliku testowego

Teraz stworzymy plik z testami:

1. Przejdź do katalogu `tests/`
2. Utwórz nowy plik `test_bloodwork_calculator.py`

```
# W terminalu, będąc w głównym katalogu projektu:
cd tests
touch test_bloodwork_calculator.py  # Na Windows: type nul >
↪  test_bloodwork_calculator.py
```

6 Krok 6: Pisanie testów jednostkowych

Teraz napiszemy testy do pliku `test_bloodwork_calculator.py`. Otwórz ten plik w edytorze tekstu i zacznij od dodania niezbędnych importów:

```
# Plik: tests/test_bloodwork_calculator.py

import pytest
from mediscan.bloodwork_calculator import (
    calculate_bmi,
    calculate_nlr,
    categorize_bmi,
    calculate_anemia_severity
)
```

6.1 Testowanie funkcji calculate_bmi

Zaczniemy od przetestowania funkcji `calculate_bmi`:

```
# Kontynuacja pliku test_bloodwork_calculator.py

def test_calculate_bmi_normal_case():
    """Test BMI dla typowych wartości."""
    # Dla osoby o wadze 70 kg i wzroście 175 cm, BMI powinno wynosić 22.86
    weight = 70  # kg
    height = 175  # cm
    expected_bmi = 70 / ((175/100) ** 2)  # ręczne obliczenie oczekiwanego
    ↪  wyniku

    # Wywołanie testowanej funkcji
    result = calculate_bmi(weight, height)

    # Sprawdzenie rezultatu
    # Używamy round(x, 2) aby zaokrąglić do 2 miejsc po przecinku
    assert round(result, 2) == round(expected_bmi, 2), \
        f"BMI dla wagi {weight}kg i wzrostu {height}cm powinno wynosić
        ↪  {expected_bmi}, ale otrzymano {result}"
```

```python
# Parametryzowany test dla różnych przypadków BMI
@pytest.mark.parametrize("weight, height, expected_bmi", [
    (50, 150, 22.22),  # niski wzrost, niska waga
    (100, 200, 25.00),  # wysoki wzrost, wysoka waga
    (80, 180, 24.69),   # średni wzrost, średnia waga
    (60, 160, 23.44),   # proporcjonalne wartości
])
def test_calculate_bmi_various_cases(weight, height, expected_bmi):
    """Test BMI dla różnych kombinacji wagi i wzrostu."""
    result = calculate_bmi(weight, height)
    assert round(result, 2) == expected_bmi, \
        f"BMI dla wagi {weight}kg i wzrostu {height}cm powinno wynosić
        ↪  {expected_bmi}, ale otrzymano {round(result, 2)}"

# Test na błędne dane - wzrost zerowy
def test_calculate_bmi_zero_height():
    """Test jak funkcja radzi sobie ze wzrostem równym 0."""
    with pytest.raises(ZeroDivisionError):
        calculate_bmi(70, 0)  # Powinno zgłosić błąd dzielenia przez zero
```

6.2 Testowanie funkcji calculate_nlr

Teraz napiszemy testy dla funkcji `calculate_nlr`:

```python
# Kontynuacja pliku test_bloodwork_calculator.py

def test_calculate_nlr_normal_case():
    """Test NLR dla typowych wartości."""
    neutrophils = 4.5  # 10^9/L
    lymphocytes = 2.5  # 10^9/L
    expected_nlr = 4.5 / 2.5  # Powinno być 1.8

    result = calculate_nlr(neutrophils, lymphocytes)

    assert result == expected_nlr, \
        f"NLR dla neutrofili {neutrophils} i limfocytów {lymphocytes}
        ↪  powinno wynosić {expected_nlr}, ale otrzymano {result}"

@pytest.mark.parametrize("neutrophils, lymphocytes, expected_nlr", [
    (5.0, 2.0, 2.5),      # typowe wartości
    (10.0, 1.0, 10.0),    # wysokie neutrofile, niskie limfocyty
    (3.0, 3.0, 1.0),      # równe wartości
    (7.5, 1.5, 5.0)       # inne typowe wartości
])
def test_calculate_nlr_various_cases(neutrophils, lymphocytes, expected_nlr):
```

```
    """Test NLR dla różnych kombinacji neutrofili i limfocytów."""
    result = calculate_nlr(neutrophils, lymphocytes)
    assert result == expected_nlr, \
        f"NLR dla neutrofili {neutrophils} i limfocytów {lymphocytes}
        ↪  powinno wynosić {expected_nlr}, ale otrzymano {result}"

def test_calculate_nlr_zero_lymphocytes():
    """Test jak funkcja radzi sobie z limfocytami równymi 0."""
    with pytest.raises(ZeroDivisionError):
        calculate_nlr(5.0, 0)  # Powinno zgłosić błąd dzielenia przez zero
```

6.3 Przygotowanie fixture dla testów

Utworzymy fixture (mechanizm przygotowujący dane dla testów), aby przygotować przykładowe dane pacjentów:

```
# Kontynuacja pliku test_bloodwork_calculator.py

@pytest.fixture
def sample_patients():
    """Fixture dostarczająca przykładowe dane pacjentów."""
    return [
        {"id": 1, "weight": 70, "height": 175, "sex": "M", "hemoglobin":
        ↪  14.5, "neutrophils": 4.5, "lymphocytes": 2.5},
        {"id": 2, "weight": 55, "height": 160, "sex": "F", "hemoglobin":
        ↪  11.5, "neutrophils": 3.8, "lymphocytes": 2.2},
        {"id": 3, "weight": 90, "height": 180, "sex": "M", "hemoglobin":
        ↪  10.5, "neutrophils": 6.5, "lymphocytes": 1.5},
        {"id": 4, "weight": 60, "height": 165, "sex": "F", "hemoglobin":
        ↪  9.5, "neutrophils": 5.0, "lymphocytes": 1.8}
    ]
```

6.4 Testowanie funkcji categorize_bmi

```
# Kontynuacja pliku test_bloodwork_calculator.py

@pytest.mark.parametrize("bmi, expected_category", [
    (16, "wygłodzenie"),
    (16.5, "wychudzenie"),
    (18.0, "niedowaga"),
    (22.0, "waga prawidłowa"),
    (27.5, "nadwaga"),
```

```
    (32.5, "otyłość I stopnia"),
    (37.5, "otyłość II stopnia"),
    (42.0, "otyłość III stopnia")
])
def test_categorize_bmi(bmi, expected_category):
    """Test kategoryzacji BMI dla różnych wartości."""
    result = categorize_bmi(bmi)
    assert result == expected_category, \
        f"Dla BMI {bmi} kategoria powinna być '{expected_category}', ale
        ↪  otrzymano '{result}'"

# Test wartości granicznych
@pytest.mark.parametrize("bmi, expected_category", [
    (16.0, "wygłodzenie"),      # dokładnie na granicy wygłodzenie/wychudzenie
    (17.0, "niedowaga"),        # dokładnie na granicy wychudzenie/niedowaga
    (18.5, "waga prawidłowa"), # dokładnie na granicy niedowaga/waga
    ↪  prawidłowa
    (25.0, "nadwaga"),          # dokładnie na granicy waga prawidłowa/nadwaga
    (30.0, "otyłość I stopnia"), # na granicy nadwaga/otyłość I stopnia
    (35.0, "otyłość II stopnia"), # na granicy otyłość I/II stopnia
    (40.0, "otyłość III stopnia") # na granicy otyłość II/III stopnia
])
def test_categorize_bmi_boundary(bmi, expected_category):
    """Test kategoryzacji BMI dla wartości granicznych."""
    result = categorize_bmi(bmi)
    assert result == expected_category, \
        f"Dla granicznego BMI {bmi} kategoria powinna być
        ↪  '{expected_category}', ale otrzymano '{result}'"
```

6.5 Testowanie funkcji calculate_anemia_severity

```
# Kontynuacja pliku test_bloodwork_calculator.py

@pytest.mark.parametrize("hemoglobin, sex, expected_severity", [
    (14.0, "M", "brak"),         # mężczyzna, normalna hemoglobina
    (12.0, "M", "łagodna"),      # mężczyzna, łagodna niedokrwistość
    (9.0, "M", "umiarkowana"),   # mężczyzna, umiarkowana niedokrwistość
    (7.0, "M", "ciężka"),        # mężczyzna, ciężka niedokrwistość

    (13.0, "F", "brak"),         # kobieta, normalna hemoglobina
    (11.0, "F", "brak"),         # kobieta, normalna hemoglobina
    (9.5, "F", "łagodna"),       # kobieta, łagodna niedokrwistość
    (7.5, "F", "umiarkowana"),   # kobieta, umiarkowana niedokrwistość
    (6.0, "F", "ciężka")         # kobieta, ciężka niedokrwistość
])
```

```python
def test_calculate_anemia_severity(hemoglobin, sex, expected_severity):
    """Test określania stopnia niedokrwistości dla różnych poziomów
    ↪ hemoglobiny i płci."""
    result = calculate_anemia_severity(hemoglobin, sex)
    assert result == expected_severity, \
        f"Dla hemoglobiny {hemoglobin} g/dL i płci {sex}, stopień
        ↪ niedokrwistości powinien być '{expected_severity}', ale
        ↪ otrzymano '{result}'"

# Test dla nieprawidłowej płci
def test_calculate_anemia_severity_invalid_sex():
    """Test jak funkcja radzi sobie z nieprawidłową płcią."""
    # Ta funkcja powinna obsługiwać tylko 'M' i 'F'
    with pytest.raises(Exception):  # Ogólna klasa Exception, bo nie wiemy
    ↪ dokładnie jaki wyjątek będzie zgłoszony
        calculate_anemia_severity(12.0, "X")
```

6.6 Wykorzystanie fixture w testach

Teraz użyjemy przygotowanego fixture do testowania funkcji:

```python
# Kontynuacja pliku test_bloodwork_calculator.py

def test_patient_bmi_calculation(sample_patients):
    """Test obliczania BMI dla przykładowych pacjentów."""
    for patient in sample_patients:
        bmi = calculate_bmi(patient["weight"], patient["height"])
        category = categorize_bmi(bmi)

        # Sprawdź czy BMI jest liczbą dodatnią
        assert bmi > 0, f"BMI pacjenta {patient['id']} powinno być dodatnie,
        ↪ ale wynosi {bmi}"

        # Sprawdź czy kategoria nie jest pusta
        assert category, f"Kategoria BMI pacjenta {patient['id']} nie
        ↪ powinna być pusta"

        print(f"Pacjent {patient['id']}: BMI = {bmi:.2f}, kategoria:
        ↪ {category}")

def test_patient_nlr_calculation(sample_patients):
    """Test obliczania NLR dla przykładowych pacjentów."""
    for patient in sample_patients:
        nlr = calculate_nlr(patient["neutrophils"], patient["lymphocytes"])

        # NLR powinno być liczbą dodatnią
```

```
        assert nlr > 0, f"NLR pacjenta {patient['id']} powinno być dodatnie,
        ↪  ale wynosi {nlr}"

        # Sprawdzamy czy obliczenia są poprawne
        expected = patient["neutrophils"] / patient["lymphocytes"]
        assert nlr == expected, \
            f"NLR pacjenta {patient['id']} powinno wynosić {expected}, ale
            ↪  wynosi {nlr}"

        print(f"Pacjent {patient['id']}: NLR = {nlr:.2f}")

def test_patient_anemia_evaluation(sample_patients):
    """Test oceny niedokrwistości dla przykładowych pacjentów."""
    for patient in sample_patients:
        severity = calculate_anemia_severity(patient["hemoglobin"],
        ↪  patient["sex"])

        # Sprawdź czy wynik nie jest pusty
        assert severity, f"Stopień niedokrwistości pacjenta {patient['id']}
        ↪  nie powinien być pusty"

        print(f"Pacjent {patient['id']}: Stopień niedokrwistości =
        ↪  {severity}")
```

7 Krok 7: Uruchamianie testów

Teraz uruchomimy nasze testy, aby zidentyfikować błędy w kodzie:

1. Upewnij się, że jesteś w głównym katalogu projektu i masz aktywowane wirtualne środowisko.

2. Uruchom testy za pomocą PyTest:

```
# W terminalu:
pytest -v tests/test_bloodwork_calculator.py
```

Flaga -v włącza tryb szczegółowy (verbose), który pokazuje więcej informacji o testach.

7.1 Analiza wyników testów

Po uruchomieniu testów, zobaczysz wyniki podobne do poniższych:

```
============================= test session starts
↪  ==============================
platform win32 -- Python 3.9.5, pytest-7.3.1, pluggy-1.0.0
rootdir: C:\Users\student\Desktop\mediscan-project
collected 15 items

tests/test_bloodwork_calculator.py::test_calculate_bmi_normal_case PASSED
tests/test_bloodwork_calculator.py::test_calculate_bmi_various_cases[50-150-
↪  22.22]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_bmi_various_cases[100-200
↪  -25.0]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_bmi_various_cases[80-180-
↪  24.69]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_bmi_various_cases[60-160-
↪  23.44]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_bmi_zero_height PASSED
tests/test_bloodwork_calculator.py::test_calculate_nlr_normal_case PASSED
tests/test_bloodwork_calculator.py::test_calculate_nlr_various_cases[5.0-2.0
↪  -2.5]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_nlr_various_cases[10.0-1.
↪  0-10.0]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_nlr_various_cases[3.0-3.0
↪  -1.0]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_nlr_various_cases[7.5-1.5
↪  -5.0]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_nlr_zero_lymphocytes
↪  PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi[16-wygłodzenie]
↪  FAILED
tests/test_bloodwork_calculator.py::test_categorize_bmi[16.5-wychudzenie]
↪  PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi[18.0-niedowaga]
↪  PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi[22.0-waga
↪  prawidłowa] PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi[27.5-nadwaga] PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi[32.5-otyłość I
↪  stopnia] PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi[37.5-otyłość II
↪  stopnia] PASSED
```

```
tests/test_bloodwork_calculator.py::test_categorize_bmi[42.0-otyłość III
↪  stopnia] PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi_boundary[16.0-wyglod
↪  zenie]
↪  FAILED
tests/test_bloodwork_calculator.py::test_categorize_bmi_boundary[17.0-niedow
↪  aga]
↪  PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi_boundary[18.5-waga
↪  prawidłowa] PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi_boundary[25.0-nadwag
↪  a]
↪  PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi_boundary[30.0-otyło
↪  ść I stopnia]
↪  PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi_boundary[35.0-otyło
↪  ść II stopnia]
↪  PASSED
tests/test_bloodwork_calculator.py::test_categorize_bmi_boundary[40.0-otyło
↪  ść III stopnia]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_anemia_severity[14.0-M-br
↪  ak]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_anemia_severity[12.0-M-ł
↪  agodna]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_anemia_severity[9.0-M-umi
↪  arkowana]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_anemia_severity[7.0-M-ci
↪  ężka]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_anemia_severity[13.0-F-br
↪  ak]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_anemia_severity[11.0-F-br
↪  ak]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_anemia_severity[9.5-F-ła
↪  godna]
↪  FAILED
tests/test_bloodwork_calculator.py::test_calculate_anemia_severity[7.5-F-umi
↪  arkowana]
↪  PASSED
tests/test_bloodwork_calculator.py::test_calculate_anemia_severity[6.0-F-ci
↪  ężka]
↪  PASSED
```

```
tests/test_bloodwork_calculator.py::test_calculate_anemia_severity_invalid_s
↪  ex
↪  FAILED
tests/test_bloodwork_calculator.py::test_patient_bmi_calculation PASSED
tests/test_bloodwork_calculator.py::test_patient_nlr_calculation PASSED
tests/test_bloodwork_calculator.py::test_patient_anemia_evaluation ERROR
```

8 Krok 8: Identyfikacja błędów

Na podstawie wyników testów, możemy zidentyfikować następujące błędy:

8.1 Błąd 1: Niepoprawna kategoryzacja BMI

Testy `test_categorize_bmi[16-wygłodzenie]` i `test_categorize_bmi_boundary[16.0-wygłodz` zawiodły. Oznacza to, że funkcja `categorize_bmi` niepoprawnie kategoryzuje BMI dla wartości niższych. Sprawdźmy dokładniej defekt:

```
# Uruchom test z więcej szczegółami:
pytest -v tests/test_bloodwork_calculator.py::test_categorize_bmi
```

Błąd może wyglądać następująco:

```
E       AssertionError: Dla BMI 16 kategoria powinna być 'wygłodzenie', ale
↪  otrzymano 'wychudzenie'
```

Przyczyna: W funkcji `categorize_bmi` warunek dla "wygłodzenia"powinien być `bmi <= 16.0` zamiast `bmi < 16.0`.

8.2 Błąd 2: Niepoprawna ocena niedokrwistości u kobiet

Test `test_calculate_anemia_severity[9.5-F-łagodna]` zawiódł. Oznacza to, że funkcja `calculate_anemia_severity` niepoprawnie ocenia stopień niedokrwistości dla kobiet.

Błąd może wyglądać następująco:

```
E       AssertionError: Dla hemoglobiny 9.5 g/dL i płci F, stopień
↪  niedokrwistości powinien być 'łagodna', ale otrzymano 'brak'
```

Przyczyna: W funkcji `calculate_anemia_severity` warunek dla kobiet jest niepoprawny. Przedział dla "brak"powinien być `hemoglobin >= 12.0` zamiast `hemoglobin >= 10.0`.

8.3 Błąd 3: Brak obsługi nieprawidłowej płci

Test `test_calculate_anemia_severity_invalid_sex` zawiódł, co oznacza, że funkcja nie zgłasza wyjątku dla nieprawidłowej płci. Funkcja po prostu zwraca `None` dla nieprawidłowej płci, zamiast zgłaszać wyjątek.

9 Krok 9: Poprawianie błędów w kodzie

Teraz naprawimy zidentyfikowane błędy w pliku `bloodwork_calculator.py`:

9.1 Poprawiona funkcja categorize_bmi

```python
def categorize_bmi(bmi):
    """
    Kategoryzuje BMI zgodnie ze standardami WHO.

    Args:
        bmi (float): Wartość BMI

    Returns:
        str: Kategoria BMI
    """
    if bmi < 16.0:  # POPRAWIONE: było bmi < 16.0
        return "wygłodzenie"
    elif bmi < 17.0:
        return "wychudzenie"
    elif bmi < 18.5:
        return "niedowaga"
    elif bmi < 25.0:
        return "waga prawidłowa"
    elif bmi < 30.0:
        return "nadwaga"
    elif bmi < 35.0:
        return "otyłość I stopnia"
    elif bmi < 40.0:
        return "otyłość II stopnia"
    else:
        return "otyłość III stopnia"
```

9.2 Poprawiona funkcja calculate_anemia_severity

```python
def calculate_anemia_severity(hemoglobin, sex):
    """
    Określa stopień niedokrwistości na podstawie poziomu hemoglobiny.

    Args:
        hemoglobin (float): Poziom hemoglobiny (g/dL)
        sex (str): Płeć pacjenta ('M' lub 'F')

    Returns:
        str: Stopień niedokrwistości lub 'brak'

    Raises:
        ValueError: Jeśli podana płeć jest nieprawidłowa
    """
    if sex == 'M':
        if hemoglobin >= 13.5:
            return "brak"
        elif hemoglobin >= 11.0:
            return "łagodna"
        elif hemoglobin >= 8.0:
            return "umiarkowana"
        else:
            return "ciężka"
    elif sex == 'F':
        if hemoglobin >= 12.0:  # POPRAWIONE: było hemoglobin >= 10.0
            return "brak"
        elif hemoglobin >= 10.0:
            return "łagodna"
        elif hemoglobin >= 8.0:
            return "umiarkowana"
        else:
            return "ciężka"
    else:
        # POPRAWIONE: Dodano rzucanie wyjątku dla nieprawidłowej płci
        raise ValueError(f"Nieprawidłowa płeć: {sex}. Dozwolone wartości to
        ↪  'M' lub 'F'.")
```

10 Krok 10: Ponowne uruchomienie testów

Po wprowadzeniu poprawek, uruchom ponownie testy, aby sprawdzić, czy błędy zostały naprawione:

```
pytest -v tests/test_bloodwork_calculator.py
```

Teraz wszystkie testy powinny przejść pomyślnie:

```
============================= test session starts
↪  ==============================
platform win32 -- Python 3.9.5, pytest-7.3.1, pluggy-1.0.0
rootdir: C:\Users\student\Desktop\mediscan-project
collected 38 items

tests/test_bloodwork_calculator.py::test_calculate_bmi_normal_case PASSED
tests/test_bloodwork_calculator.py::test_calculate_bmi_various_cases[50-150-
↪  22.22]
↪  PASSED
... [pozostałe testy] ...
tests/test_bloodwork_calculator.py::test_patient_anemia_evaluation PASSED

============================ 38 passed in 0.25s ============================
```

11 Krok 11: Dodatkowe testy i ulepszenia

Po naprawieniu podstawowych błędów, możemy dodać więcej testów, aby zwiększyć pokrycie kodu i sprawdzić więcej przypadków brzegowych.

11.1 Testowanie wartości ujemnych

Dodajmy testy dla ujemnych wartości wagi i wzrostu:

```
def test_calculate_bmi_negative_values():
    """Test jak funkcja radzi sobie z ujemnymi wartościami."""
    # Ujemna waga nie ma sensu fizycznego
    with pytest.raises(ValueError):
        calculate_bmi(-70, 175)

    # Ujemny wzrost nie ma sensu fizycznego
    with pytest.raises(ValueError):
        calculate_bmi(70, -175)
```

Te testy zawiodą, ponieważ funkcja `calculate_bmi` nie sprawdza, czy wartości są dodatnie. Poprawmy to:

```python
def calculate_bmi(weight_kg, height_cm):
    """
    Oblicza BMI (Body Mass Index) na podstawie wagi i wzrostu.

    Args:
        weight_kg (float): Waga w kilogramach
        height_cm (float): Wzrost w centymetrach

    Returns:
        float: Wartość BMI

    Raises:
        ValueError: Jeśli waga lub wzrost są niedodatnie
    """
    # Sprawdzenie, czy wartości są dodatnie
    if weight_kg <= 0:
        raise ValueError("Waga musi być dodatnia")
    if height_cm <= 0:
        raise ValueError("Wzrost musi być dodatni")

    height_m = height_cm / 100  # konwersja cm na m
    return weight_kg / (height_m * height_m)
```

11.2 Podobnie dla funkcji calculate_nlr

```python
def test_calculate_nlr_negative_values():
    """Test jak funkcja radzi sobie z ujemnymi wartościami."""
    # Ujemne wartości neutrofili lub limfocytów nie mają sensu medycznego
    with pytest.raises(ValueError):
        calculate_nlr(-4.5, 2.5)

    with pytest.raises(ValueError):
        calculate_nlr(4.5, -2.5)
```

I poprawka funkcji:

```python
def calculate_nlr(neutrophils, lymphocytes):
    """
    Oblicza stosunek neutrofili do limfocytów (NLR).
    Podwyższony NLR może wskazywać na stan zapalny.

    Args:
        neutrophils (float): Liczba neutrofili (10^9/L)
        lymphocytes (float): Liczba limfocytów (10^9/L)
```

```
    Returns:
        float: Wartość NLR

    Raises:
        ValueError: Jeśli wartości są ujemne
        ZeroDivisionError: Jeśli lymphocytes wynosi 0
    """
    # Sprawdzenie, czy wartości są dodatnie
    if neutrophils < 0:
        raise ValueError("Liczba neutrofili nie może być ujemna")
    if lymphocytes < 0:
        raise ValueError("Liczba limfocytów nie może być ujemna")

    # Dzielenie przez zero generuje ZeroDivisionError automatycznie
    return neutrophils / lymphocytes
```

12 Krok 12: Opisanie znalezionych problemów i wprowadzonych poprawek

Na podstawie naszych testów, zidentyfikowaliśmy i naprawiliśmy następujące problemy:

1. **Problem z kategoryzacją BMI dla wartości poniżej 16.0** - funkcja `categorize_bmi` niepoprawnie kategoryzowała BMI dla wartości na granicy kategorii.
2. **Błędne przedziały dla oceny niedokrwistości u kobiet** - funkcja `calculate_anemia_severit` używała nieprawidłowych wartości dla oceny niedokrwistości u kobiet.
3. **Brak obsługi nieprawidłowej płci** - funkcja `calculate_anemia_severity` nie zgłaszała wyjątku dla nieprawidłowej płci.
4. **Brak walidacji danych wejściowych** - funkcje `calculate_bmi` i `calculate_nlr` nie sprawdzały, czy wartości wejściowe są sensowne (dodatnie).

13 Krok 13: Dodatkowe testy za pomocą markerów (opcjonalne)

Możemy dodać markery do naszych testów, aby je lepiej organizować:

```python
# Na początku pliku test_bloodwork_calculator.py

import pytest

# Definiowanie markerów
pytest.mark.bmi = pytest.mark.bmi  # Testy związane z BMI
pytest.mark.nlr = pytest.mark.nlr  # Testy związane z NLR
pytest.mark.anemia = pytest.mark.anemia  # Testy związane z anemią
pytest.mark.boundary = pytest.mark.boundary  # Testy wartości granicznych
```

Teraz możemy dodać markery do naszych testów:

```python
@pytest.mark.bmi
def test_calculate_bmi_normal_case():
    # ...

@pytest.mark.bmi
@pytest.mark.parametrize("weight, height, expected_bmi", [
    # ...
])
def test_calculate_bmi_various_cases(weight, height, expected_bmi):
    # ...

@pytest.mark.nlr
def test_calculate_nlr_normal_case():
    # ...

@pytest.mark.anemia
@pytest.mark.parametrize("hemoglobin, sex, expected_severity", [
    # ...
])
def test_calculate_anemia_severity(hemoglobin, sex, expected_severity):
    # ...
```

Aby uruchomić tylko testy związane z BMI:

```
pytest -v -m bmi tests/test_bloodwork_calculator.py
```

14 Podsumowanie

W ramach tego zadania:

1. Pobraliśmy kod źródłowy projektu MediScan z GitHub

2. Zapoznaliśmy się ze strukturą projektu i kodem modułu `bloodwork_calculator.py`
3. Napisaliśmy testy jednostkowe dla funkcji w tym module, używając PyTest
4. Zastosowaliśmy zaawansowane funkcje PyTest, takie jak:
    - Parametryzacja testów do sprawdzenia wielu przypadków
    - Fixtures do przygotowania danych testowych
    - Markery do kategoryzacji testów
    - Testowanie wyjątków
5. Zidentyfikowaliśmy błędy w kodzie na podstawie wyników testów
6. Wprowadziliśmy poprawki, które naprawiły zidentyfikowane problemy
7. Dodaliśmy dodatkowe testy i walidację danych wejściowych

Dzięki testom jednostkowym udało nam się znaleźć i naprawić błędy, które mogłyby prowadzić do niepoprawnych diagnoz medycznych. Pokazuje to, jak ważne jest testowanie oprogramowania, szczególnie w tak krytycznych obszarach jak medycyna.

Ten proces pokazuje typowy cykl pracy z testami jednostkowymi:

1. Napisanie testów
2. Uruchomienie testów i identyfikacja błędów
3. Poprawienie kodu
4. Ponowne uruchomienie testów
5. Dodanie większej liczby testów

PyTest znacznie ułatwia ten proces dzięki swojej prostej składni, zaawansowanym funkcjom i czytelnym raportom z testów.