XD PROJEKT – Dynamiczna Analiza Oprogramowania

Dariusz Kołodziejczyk, Sebastian Bek, Mikołaj Maliszewski

11 stycznia 2026

1 Wprowadzenie

Poniższe testy zostały przeprowadzone w środowisku Visual Studio 2022 na komputerze o specyfikacji

- **Karta graficzna:** NVIDIA GeForce GTX 1060
- **Procesor:** Intel Core i7-8750H
- **Pamięć RAM:** 16 GB DDR4
- **Dysk:** SSD NVMe
- **System operacyjny:** Windows 10 Home

2 Testy jednostkowe

2.1 Izolacja środowiska testowego cache

```
@pytest.fixture(autouse=True)
def isolated_cache(tmp_path, monkeypatch):
fake_cache_dir = tmp_path / "cache"
fake_cache_dir.mkdir()

monkeypatch.setattr("CDPdata.CACHE_DIR", str(
   fake_cache_dir))
monkeypatch.setattr(
"CDPdata.CACHE_FILE",
str(fake_cache_dir / "trends_cache.json")
)

yield
```

W celu zapewnienia pełnej izolacji testów jednostkowych oraz uniknięcia wpływu testów na rzeczywiste dane aplikacji, zastosowano fixturę testową frameworka `pytest`. Jej zadaniem jest przygotowanie odrębnego, tymczasowego katalogu cache dla każdego uruchamianego testu.

Fixtura automatycznie podmienia ścieżki do katalogu oraz pliku cache wykorzystywane przez aplikację, kierując je do tymczasowego katalogu tworzonego na potrzeby testu. Dzięki temu każdy test działa w pełni niezależnie, a jego wykonanie nie wpływa na inne testy ani na środowisko uruchomieniowe aplikacji.

2.2 Test zapisu i odczytu danych cache

```
def test_save_and_load_cache():
data = {"test": {"7d": {"json": "{}", "period": "
   7d"}}}
save_cache(data)

loaded = load_cache()
assert loaded == data
```

Cel testu Celem testu jest weryfikacja poprawności działania mechanizmu cache w podstawowym scenariuszu użycia, obejmującym zapis danych do cache oraz ich późniejszy odczyt bez utraty lub modyfikacji informacji.

Przebieg testu W ramach testu do mechanizmu cache zapisywany jest przykładowy zestaw danych w postaci struktury JSON. Następnie dane te są odczytywane z cache przy użyciu tego samego klucza identyfikującego wpis.

Efekt oczekiwany Oczekuje się, że dane odczytane z cache będą identyczne z danymi zapisanymi, co potwierdza poprawność działania mechanizmu zapisu i odczytu cache.

2.3 Test obsługi uszkodzonego pliku cache

```
def test_invalid_json_moves_file(tmp_path,
   monkeypatch):
import importlib
mod = importlib.import_module("CDPdata")

moved = []
def fake_replace(src, dst):
moved.append((src, dst))

monkeypatch.setattr(mod.os, "replace",
   fake_replace)

with open(mod.CACHE_FILE, "w", encoding="utf-8")
   as f:
f.write("{zlezlezle}")

cache = mod.load_cache()
assert cache == {}
assert len(moved) == 1
```

Cel testu Celem testu jest sprawdzenie odporności mechanizmu cache na sytuację, w której plik cache zawiera niepoprawne dane w formacie JSON.

Przebieg testu Test symuluje obecność uszkodzonego pliku cache poprzez zapisanie niepoprawnej struktury JSON. Następnie podejmowana jest próba odczytu danych z cache przez aplikację.

Efekt oczekiwany Oczekiwanym rezultatem jest przeniesienie uszkodzonego pliku cache oraz zwrócenie pustej struktury danych, co zapobiega przerwaniu działania aplikacji i umożliwia jej dalsze funkcjonowanie.

2.4 Test unieważniania wpisu cache

```
1                   def test_invalidate_removes_entry():
2                   data = {"bitcoin": {"7d": {"json": "{}", "period"
                        : "7d"}}}
3                   save_cache(data)
4
5                   invalidate_trends_period("bitcoin", "7d")
6
7                   loaded = load_cache()
8                   assert loaded == {}
```

Cel testu Celem testu jest weryfikacja poprawności działania mechanizmu unieważniania pojedynczego wpisu w cache.

Przebieg testu W cache zapisywany jest wpis identyfikowany unikalnym kluczem. Następnie wywoływana jest operacja unieważnienia tego wpisu, po czym następuje próba jego ponownego odczytu.

Efekt oczekiwany Oczekuje się, że po unieważnieniu wpis nie będzie dostępny w cache, co potwierdza skuteczność mechanizmu usuwania danych.

2.5 Test poprawnej konwersji danych do obiektu DataFrame

```
def test_df_from_entry_valid():
df = pd.DataFrame({"bitcoin": [1, 2, 3]},
index=pd.date_range("2024-01-01", periods=3))
entry = {"json": df.to_json(orient="split")}

out = df_from_entry(entry)

assert isinstance(out, pd.DataFrame)
assert list(out.columns) == ["bitcoin"]
assert len(out) == 3
```

Cel testu Celem testu jest weryfikacja poprawności działania funkcji odpowiedzialnej za konwersję danych zapisanych w formacie JSON do obiektu `DataFrame` biblioteki `pandas`.

Przebieg testu W teście tworzony jest przykładowy obiekt `DataFrame` zawierający dane dla wybranego zasobu, który następnie zapisywany jest do formatu JSON. Tak przygotowane dane przekazywane są do funkcji konwertującej.

Efekt oczekiwany Oczekuje się, że funkcja zwróci poprawny obiekt typu `DataFrame` z zachowaną strukturą kolumn oraz liczbą rekordów odpowiadającą danym wejściowym.

2.6 Test obsługi niepoprawnych danych wejściowych

```
def test_df_from_entry_invalid_json():
entry = {"json": "{kielbasa␣XD"}
        assert df_from_entry(entry) is None
```

Cel testu Celem testu jest sprawdzenie zachowania funkcji konwertującej w sytuacji, gdy przekazane dane wejściowe nie są poprawnym zapisem w formacie JSON.

Przebieg testu Do funkcji przekazywana jest struktura danych zawierająca niepoprawny łańcuch znaków zamiast prawidłowego zapisu JSON.

Efekt oczekiwany Oczekiwanym rezultatem testu jest zwrócenie wartości `None`, co sygnalizuje brak możliwości wykonania poprawnej konwersji danych.

2.7 Test obsługi pustych danych wejściowych

```
def test_df_from_entry_empty():
assert df_from_entry({}) is None
```

Cel testu Celem testu jest weryfikacja zachowania funkcji konwertującej w przypadku, gdy przekazana struktura danych nie zawiera wymaganych informacji.

Przebieg testu Do funkcji przekazywana jest pusta struktura danych, pozbawiona klucza zawierającego dane w formacie JSON.

Efekt oczekiwany Oczekuje się, że funkcja zwróci wartość `None`, co potwierdza poprawną obsługę brakujących danych wejściowych.

3 Testy integracyjne

3.1 Test pobrania danych trendów i zapis do cache

```
def test_gettrends_fetches_and_saves(monkeypatch)
    :
df = pd.DataFrame({"bitcoin": [10, 20]}, index=pd
    .date_range("2024-01-01", periods=2))
monkeypatch.setattr("CDPdata.TrendReq", lambda hl
    , tz: FakePytrends(df))

out = getTrendsData("bitcoin", "7d")
assert isinstance(out, pd.DataFrame)
assert len(out) == 2

cache = load_cache()
assert "bitcoin" in cache
assert "7d" in cache["bitcoin"]
```

Cel testu Celem testu jest weryfikacja poprawności współpracy funkcji pobierającej dane trendów z mechanizmem cache. Test sprawdza, czy dane pobrane z API są poprawnie zapisywane w lokalnym cache.

Przebieg testu Test wykorzystuje atrapę klasy `FakePytrends`, która zwraca przykładowy `DataFrame`. Funkcja `getTrendsData` jest wywoływana dla określonego zasobu i przedziału czasowego. Po wykonaniu operacji sprawdzany jest stan cache.

Efekt oczekiwany Oczekuje się, że funkcja zwróci obiekt `DataFrame` odpowiadający danym z API oraz że dane zostaną zapisane w cache pod właściwym kluczem.

3.2 Test użycia danych już zapisanych w cache

```
def test_gettrends_uses_cache(monkeypatch):
df = pd.DataFrame({"bitcoin": [10]}, index=pd.
    date_range("2024-01-01", periods=1))
monkeypatch.setattr("CDPdata.TrendReq", lambda hl
    , tz: FakePytrends(df))
_ = getTrendsData("bitcoin", "7d")

def fail_pytrends(*a, **kw):
raise AssertionError("Nie powinno byc zapytania
    do pytrends!")

monkeypatch.setattr("CDPdata.TrendReq",
    fail_pytrends)
out = getTrendsData("bitcoin", "7d")
assert len(out) == 1
```

Cel testu Celem testu jest weryfikacja, czy funkcja pobierająca dane trendów korzysta z istniejącego cache zamiast wykonywać niepotrzebne zapytania do API.

Przebieg testu Do cache zapisywany jest przykładowy zestaw danych. Następnie funkcja `getTrendsData` jest wywoływana ponownie dla tego samego zasobu i okresu, przy czym symulacja API została zastąpiona funkcją generującą błąd w przypadku wywołania.

Efekt oczekiwany Oczekuje się, że funkcja zwróci dane z cache bez wywoływania API, co potwierdza poprawne wykorzystanie lokalnego mechanizmu pamięci podręcznej.

3.3 Test fallbacku do cache przy błędzie API

```
def test_gettrends_fallback_on_error(monkeypatch)
    :
df = pd.DataFrame({"bitcoin": [10]}, index=pd.
    date_range("2024-01-01", periods=1))
monkeypatch.setattr("CDPdata.TrendReq", lambda hl
    , tz: FakePytrends(df))
_ = getTrendsData("bitcoin", "7d")

def exploding_pytrends(*a, **kw):
raise Exception("API ERROR")

monkeypatch.setattr("CDPdata.TrendReq",
    exploding_pytrends)
out = getTrendsData("bitcoin", "7d")
assert len(out) == 1
```

Cel testu Celem testu jest sprawdzenie, czy funkcja pobierająca dane trendów poprawnie korzysta z danych zapisanych w cache w przypadku wystąpienia błędu po stronie API.

Przebieg testu Do cache zapisywane są przykładowe dane. Następnie wywołanie funkcji `getTrendsData` symuluje awarię API poprzez atrapę generującą wyjątek. Funkcja powinna w takim przypadku wykorzystać dane z cache.

Efekt oczekiwany Oczekuje się, że funkcja zwróci dane zapisane w cache, pomimo błędu API, co potwierdza poprawność mechanizmu fallback.

4 Testy funkcjonalne (GUI)

4.1 Test wyświetlania wykresu na ekranie trendów

```
def test_trends_screen_shows_plot(monkeypatch):
df = pd.DataFrame(
{"CD Projekt": [10, 20, 30]},
index=pd.date_range("2024-01-01", periods=3)
)
monkeypatch.setattr("CDPdata.getTrendsData",
    lambda keyword, period: df)

root = ctk.CTk()
screen = Screen2(root)

screen.showTrendsPlot("7d")

widgets = screen.plot_frame.winfo_children()
assert len(widgets) > 0
```

Cel testu Celem testu jest weryfikacja, czy po wywołaniu funkcji odpowiedzialnej za wyświetlanie wykresu w interfejsie użytkownika, dane są poprawnie renderowane na ekranie trendów.

Przebieg testu Do funkcji GUI przekazywane są przykładowe dane trendów poprzez atrapę funkcji `getTrendsData`. Następnie wywoływana jest akcja użytkownika, powodująca wyświetlenie wykresu. Test sprawdza, czy w ramce odpowiedzialnej za wykres (`plot_frame`) pojawiły się widgety reprezentujące wykres.

Efekt oczekiwany Oczekuje się, że w ramce `plot_frame` pojawią się przynajmniej jeden lub więcej widgetów, co potwierdza poprawne wyświetlenie wykresu.

4.2 Test wymuszenia ponownego ładowania danych (refresh)

```
                def test_refresh_forces_reload(monkeypatch):
                calls = {"count": 0}

                def fake_get(keyword, period):
                calls["count"] += 1
                return pd.DataFrame(
                {"CD Projekt": [calls["count"]]},
                index=pd.date_range("2024-01-01", periods=1)
                )

                monkeypatch.setattr("CDPdata.getTrendsData",
                   fake_get)

                root = ctk.CTk()
                screen = Screen2(root)

                calls["count"] = 0

                screen.showTrendsPlot("7d")
                first = screen.plot_frame.winfo_children()

                screen.refreshTrends()
                second = screen.plot_frame.winfo_children()

                assert calls["count"] == 2
                assert first != second
```

Cel testu Celem testu jest weryfikacja, czy funkcja odświeżania danych (`refreshTrends`) poprawnie wymusza ponowne pobranie danych trendów i aktualizację widoku wykresu.

Przebieg testu Przy pomocy atrap funkcji `getTrendsData` monitorowana jest liczba wywołań API. Najpierw wyświetlany jest wykres, następnie wywoływana jest funkcja odświeżenia danych. Test porównuje widgety w ramce `plot_frame` przed i po odświeżeniu oraz sprawdza, czy liczba wywołań funkcji pobierającej dane wzrosła o jeden.

Efekt oczekiwany Oczekuje się, że liczba wywołań funkcji pobierającej dane wzrośnie do dwóch, a widgety w ramce `plot_frame` ulegną zmianie, co potwierdza odświeżenie danych i aktualizację wykresu.

4.3 Test wyświetlania komunikatu o błędzie

```
def test_trends_error_message(monkeypatch):
monkeypatch.setattr(
"CDPdata.getTrendsData",
lambda keyword, period: None
)

root = ctk.CTk()
screen = Screen2(root)

screen.showTrendsPlot("7d")

labels = [
w for w in screen.plot_frame.winfo_children()
if "Nie udalo sie" in getattr(w, "cget", lambda *
    _: "")("text")
]

assert len(labels) == 1
```

Cel testu Celem testu jest sprawdzenie, czy w przypadku braku danych funkcja GUI poprawnie wyświetla komunikat błędu dla użytkownika.

Przebieg testu Przy pomocy atrakcji funkcji `getTrendsData` symulowany jest scenariusz, w którym brak danych (zwracane jest `None`). Następnie wywoływana jest funkcja wyświetlająca wykres, a test przeszukuje ramkę `plot_frame` pod kątem etykiet zawierających komunikat o błędzie.

Efekt oczekiwany Oczekuje się, że w ramce `plot_frame` pojawi się etykieta z komunikatem „Nie udało się pobrać danych”, potwierdzając poprawną obsługę błędu w GUI.

4.4 Test wyświetlania wykresu dla standardowego okresu

```
def test_showPlot_shows_plot(monkeypatch):
root = ctk.CTk()

fake_df = pd.DataFrame(
{"Close": [10, 12, 11]},
index=pd.date_range("2024-01-01", periods=3)
)

calls = {"get": 0, "plot": 0}

monkeypatch.setattr(
"CDPdata.getCdpData",
lambda period: calls.__setitem__("get", calls["get"] + 1) or fake_df
)

monkeypatch.setattr(
"CDPplot.createCdpPlot",
lambda *args, **kwargs: calls.__setitem__("plot", calls["plot"] + 1)
)

monkeypatch.setattr(
"CDPdata.getCurrentPrice",
lambda: 123.45
)

monkeypatch.setattr(
"CDPdata.getMinMaxPrice",
lambda df: (10, 12)
)

screen = Screen1(root)
screen.showPlot("7d")

assert calls["get"] == 1
assert calls["plot"] == 1
assert screen.price_label_value.cget("text") == "123.45 PLN"
assert screen.min_label_value.cget("text") == "10 PLN"
assert screen.max_label_value.cget("text") == "12 PLN"
```

Cel testu Celem testu jest weryfikacja, czy metoda `showPlot` poprawnie pobiera dane, generuje wykres i aktualizuje etykiety GUI dla standardowego okresu (np. 7 dni).

Przebieg testu Do testu użyto fikcyjnych danych w postaci DataFrame. Funkcje pobierające dane i tworzące wykresy zostały podmienione za pomocą `monkeypatch`, aby śledzić wywołania i kontrolować wyniki. Następnie wywołano `showPlot("7d")`.

Efekt oczekiwany Oczekuje się, że:

- Funkcje pobierające dane i tworzące wykres zostaną wywołane dokładnie raz,
- Etykiety GUI z ceną bieżącą, minimalną i maksymalną będą zawierały poprawne wartości.

4.5 Test wyświetlania wykresu dla niestandardowego zakresu dat

```
def test_custom_date_valid(monkeypatch):
root = ctk.CTk()

fake_df = pd.DataFrame(
{"Close": [100, 110]},
index=pd.date_range("2024-01-01", periods=2)
)

calls = {"data": 0, "plot": 0}

monkeypatch.setattr(
"CDPdata.getCustomCdpData",
lambda start, end: calls.__setitem__("data",
   calls["data"] + 1) or fake_df
)

monkeypatch.setattr(
"CDPplot.createCustomDataCdpPlot",
lambda *args, **kwargs: calls.__setitem__("plot",
    calls["plot"] + 1)
)

monkeypatch.setattr(
"CDPdata.getMinMaxPrice",
lambda df: (100, 110)
)

screen = Screen1(root)

screen.start_date_entry.set_date(fake_df.index.
   min().date())
screen.end_date_entry.set_date(fake_df.index.max
   ().date())

screen.showCustomDatePlot()

assert calls["data"] == 1
```

```
34                  assert calls["plot"] == 1
35                  assert screen.min_label_value.cget("text") == "
                        100 PLN"
36                  assert screen.max_label_value.cget("text") == "
                        110 PLN"
```

Cel testu Celem testu jest weryfikacja, że metoda `showCustomDatePlot` poprawnie działa dla poprawnego, niestandardowego zakresu dat.

Przebieg testu Do testu użyto fikcyjnych danych. Funkcje pobierające dane i tworzące wykres zostały podmienione. Daty start i end zostały ustawione na odpowiadające zakresowi danych. Wywołano `showCustomDatePlot()`.

Efekt oczekiwany Oczekuje się, że:

- Dane i wykres zostaną pobrane i wygenerowane dokładnie raz,
- Etykiety GUI z minimalną i maksymalną ceną będą zawierały poprawne wartości.

4.6 Test niestandardowego zakresu dat - niepoprawny zakres

```
1                   def test_custom_date_invalid_range(monkeypatch):
2                   root = ctk.CTk()
3                   screen = Screen1(root)
4
5                   errors = []
6
7                   monkeypatch.setattr(
8                   screen,
9                   "showError",
10                  lambda title, message, icon="warning": errors.
                        append((title, message))
11                  )
12
13                  screen.start_date_entry.set_date(date(2024, 5,
                        10))
14                  screen.end_date_entry.set_date(date(2024, 5, 1))
15
16                  screen.showCustomDatePlot()
17
18                  assert len(errors) == 1
19                  assert "Bledny zakres dat" in errors[0][0]
```

Cel testu Celem testu jest sprawdzenie, czy metoda `showCustomDatePlot` poprawnie reaguje na niepoprawny zakres dat (data startowa późniejsza niż końcowa).

Przebieg testu Podmieniono metodę `showError`, aby rejestrować komunikaty błędu. Ustawiono datę startową późniejszą niż końcową i wywołano `showCustomDatePlot()`.

Efekt oczekiwany Oczekuje się, że zostanie wywołana funkcja `showError` z komunikatem informującym o błędnym zakresie dat.

4.7 Test niestandardowego zakresu dat - data w przyszłości

```
def test_custom_date_future_date(monkeypatch):
root = ctk.CTk()
screen = Screen1(root)

errors = []

monkeypatch.setattr(
screen,
"showError",
lambda title, message, icon="warning": errors.
    append((title, message))
)

tomorrow = date.today() + timedelta(days=1)

screen.start_date_entry.set_date(date.today())
screen.end_date_entry.set_date(tomorrow)

screen.showCustomDatePlot()

assert len(errors) == 1
assert "przyszlosci" in errors[0][1]

root.destroy()
```

Cel testu Celem testu jest weryfikacja, czy metoda `showCustomDatePlot` poprawnie reaguje, gdy wybrana jest data w przyszłości.

Przebieg testu Podmieniono metodę `showError` w celu rejestrowania komunikatów. Ustawiono datę końcową na przyszłą i wywołano `showCustomDatePlot()`.

Efekt oczekiwany Oczekuje się, że pojawi się komunikat błędu informujący o niemożliwości wyboru daty w przyszłości.

5 Testy wydajnościowe

5.1 Test czasu odczytu danych z cache

```
KEYWORD = "CD Projekt"
PERIOD = "7d"

def test_cache_read_is_fast(tmp_path, monkeypatch):
import time

# duzy DataFrame
df = pd.DataFrame(
{"CD Projekt": range(365)},
index=pd.date_range("2023-01-01", periods=365)
)

save_cache({
        KEYWORD: {
                PERIOD: {
                        "period": PERIOD,
                        "json": df.to_json(orient="split")
                }
        }
})

# pomiar czasu
start = time.perf_counter()
result = getTrendsData(KEYWORD, PERIOD)
elapsed = time.perf_counter() - start

assert result is not None
assert len(result) == 365

# twardy limit (lokalnie powinno byc < 50 ms)
assert elapsed < 0.05
```

Cel testu Celem testu jest weryfikacja wydajności mechanizmu cache przy odczycie dużego obiektu danych. Test sprawdza, czy pobranie danych zapisanych lokalnie jest szybkie i nie powoduje opóźnień w działaniu aplikacji.

Przebieg testu Do cache zapisywany jest przykładowy, duży obiekt typu `DataFrame` zawierający 365 rekordów. Następnie wywoływana jest funkcja `getTrendsData` dla tego samego zasobu i przedziału czasowego. Czas wykonania operacji jest mierzony przy użyciu `time.perf_counter`.

Efekt oczekiwany Oczekuje się, że funkcja zwróci pełny `DataFrame` o 365 rekordach, a czas odczytu nie przekroczy 50 ms. Test potwierdza, że mechanizm cache zapewnia szybki

dostęp do lokalnie zapisanych danych.

6 Debugging

Aplikacja została przetestowana z wykorzystaniem debuggera w środowisku `Microsoft Visual Studio`, co pozwoliło na identyfikację i eliminację błędów logicznych oraz nieprawidłowego przetwarzania danych.

6.1 Obsługa danych JSON

Aplikacja wczytuje dane zapisane w formacie JSON do obiektu `DataFrame` przy użyciu funkcji `pd.read_json()`.

Podczas testowania aplikacji z wykorzystaniem debuggera zauważono pojawienie się ostrzeżenia `FutureWarning` przy wczytywaniu danych .

```
FutureWarning: Passing literal json to 'read_json' is deprecated and will be removed in a future version. \\
df = pd.read_json(entry["json"], orient="split")
```

Problem został wykryty w następujący sposób:

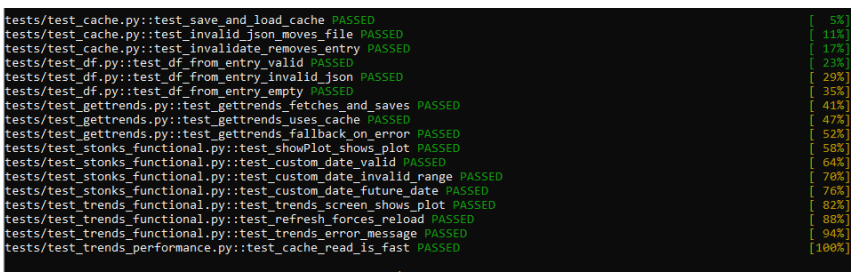
- Aplikacja została uruchomiona w trybie debugowania w środowisku `[nazwa IDE]`.
- Ustawiono breakpoint w miejscu, w którym następuje wczytywanie danych JSON.
- Podczas krokowego wykonywania programu (`step over`) obserwowano wartość zmiennej `entry["json"]`.
- Zauważono, że przekazywany jest dosłowny string JSON do funkcji `pd.read_json()`, co powoduje ostrzeżenie.

Rozwiązaniem problemu było opakowanie stringa w obiekt `StringIO`, co zapewnia poprawne wczytanie danych i kompatybilność z przyszłymi wersjami Pandas:

```
from io import StringIO
import pandas as pd

df = pd.read_json(StringIO(entry["json"]), orient="split")
```

7 Screen poprawności testów



8 Pokrycie linii

Moduł	Stmts	Miss	Cover
CDPdata	160	77	52%
CDPplot	131	100	24%

Tabela 1: Podsumowanie pokrycia kodu