Testowanie i Jakość Oprogramowania

Akademia Tarnowska

Katedra Informatyki

Tomasz Gądek

Wykład 1: Wprowadzenie

Cele

- Testowanie oprogramowania (unittest, TDD).
- Jakość oprogramowania (czysty kod, wzorce projektowe).
- Zrozumieć SOLID.
- Zrozumieć FIRST.

Zakres tematyczny zajęć

1. Zagadnienia wstępne (4 wykłady).
2. Jakość oprogramowania (4 wykłady).
3. Podsumowanie wiadomości, Q&A (1 wykład).
4. Testowanie oprogramowania (4 wykłady).
5. Podsumowanie wiadomości, Q&A (1 wykład).
6. Kolokwium (1 wykład).

Zagadnienia wstępne (4 wykłady)

- Wprowadzenie.
- Python.
- Flask.
- SCRUM.

Jakość oprogramowania (4 wykłady)

- Elementy programowania funkcyjnego.
- Czysty kod / OOP / dobre praktyki.
- SOLID, DRY, KISS, PRAWO DEMETER, CODE SMELL.
- Wzorce projektowe.

Testowanie oprogramowania (4 wykłady)

- Wprowadzenie do testowania, piramida testów.
- Testy jednostkowe i integracyjne.
- Atrapy.
- TDD.

Podsumowanie wiadomości

- Q&A na wykładzie.

Kolokwia

- Wykład (kolokwium teoretyczne).
- Laboratorium (kolokwium praktyczne).

Struktura kursu

- Wykłady (7 spotkań + kolokwium).
- Laboratoria (14 spotkań + kolokwium).

Zasady zaliczenia kursu

- Obecność na zajęciach.
- Zaangażowanie na laboratorium.
- Aktywność na wykładzie (Q&A).
- Kolokwium.

Materiały dydaktyczne

- Wykłady.
- Konspekty.
- Dokumentacja.
- Literatura.

Literatura

- *Nauka programowania opartego na testach. Jak pisać przejrzysty kod w kilku językach programowania* - **Saleem Siddiqui**.
- *Czysty kod* - **Robert C. Martin**.

Kontakt

- Strona: tgadek.bitbucket.io.
- Konsultacje: Piątek 14:45-15:45, C100d.
- E-mail: t_gadek@atar.edu.pl.

Narzędzia

- Git, GitHub / Bitbucket.
- Python, Flask.
- unittest, Framework-less Unit Tests.
- HTML, CSS, JS.
- PyCharm / IntelliJ IDEA.

Instalacja Python

Pobierz: python.org.

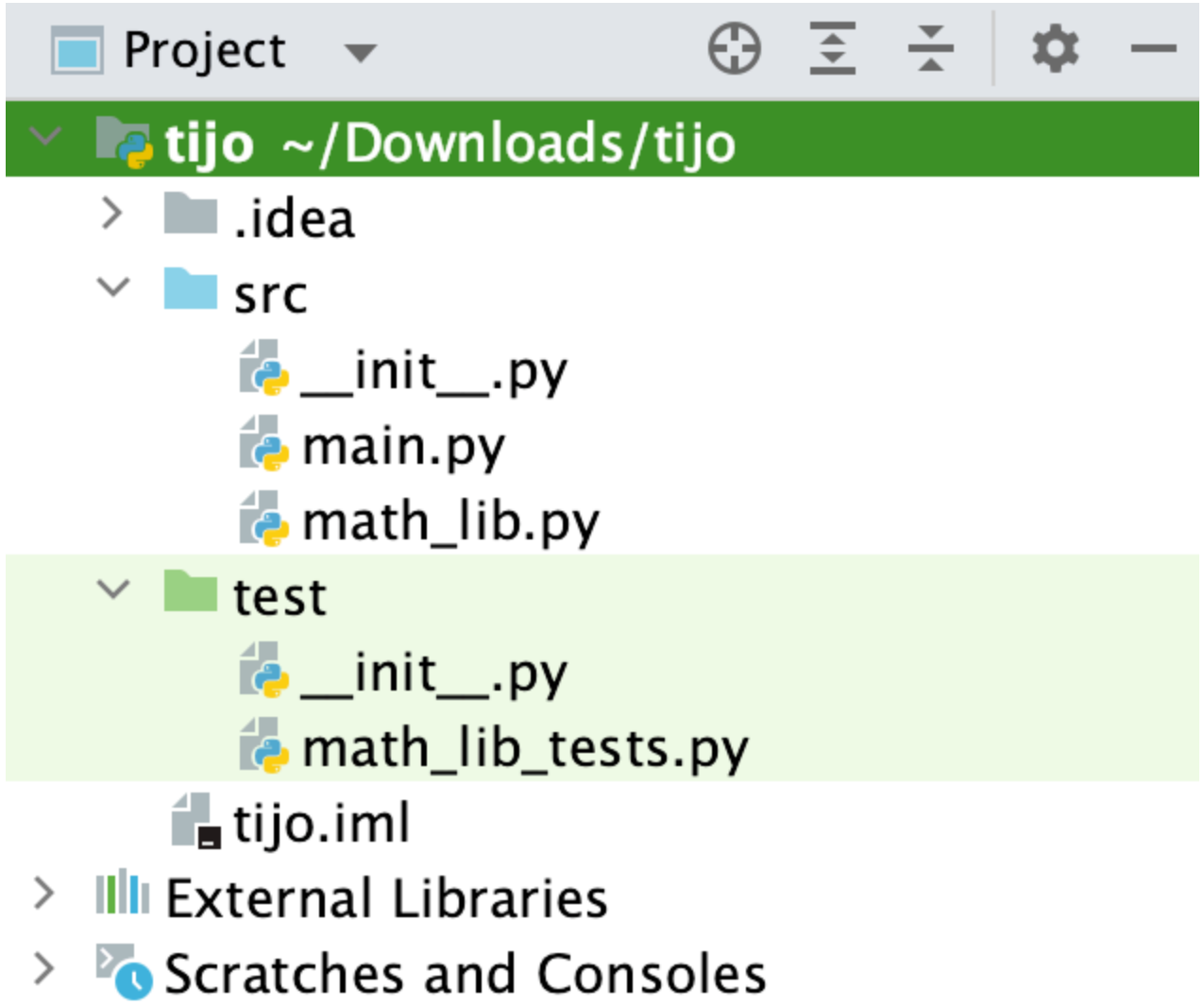
Weryfikacja wersji:

```
python --version
```

Instalacja PyCharm

Pobierz: jetbrains.com/pycharm i zainstaluj.

Struktura projektu (moduły i pakiety)



Python - Historia

- **1991** – Guido van Rossum tworzy język Python.
- **2000** – Premiera Python 2.
- **2008** – Premiera Python 3.
- **Obecnie** – Python dominuje w AI, analizie danych i automatyzacji.
- **Przyszłość** – Dynamiczny rozwój, nowe biblioteki i zastosowania.

Python - Co to za język?

- Wysokopoziomowy.
- Dynamicznie typowany.
- Interpretowany.

Zmienne (C)

```c
int x = 10;
printf("%d", x);
```

Zmienne (Python)

```
x = 10
print(x)
```

Funkcje (C)

```c
int add(int a, int b) {
    return a + b;
}
```

Funkcje (Python)

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

Instrukcje warunkowe (C)

```c
if (x > 0) {
    printf("Liczba dodatnia");
} else if(x == 0) {
    printf("Zero")
} else {
    printf("Liczba ujemna");
}
```

Instrukcje warunkowe

```
if x > 0:
    print("Liczba dodatnia")
elif x == 0:
    print("Zero")
else:
    print("Liczba ujemna")
```

Pętle (C)

```c
for(int i = 0; i < 5; i++) {
    printf("%d", i);
}
```

Pętle (Python)

```python
for i in range(5):
    print(f"{i}")
```

Struktura (C)

```c
struct Person {
    char name[50];
    int age;
};
```

Klasa (Python)

```
class Person:
    def __init__(self):
        self.name = ""
        self.age = 0
```

Asercja (C)

```
assert(x > 0);
```

Asercja (Python)

```
assert x > 0
```

Struktura testu / Framework-less Unit Tests

Framework-less:

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

def test_add():
    assert add(2, 3) == 5, "2 + 3 = 5?"

if __name__ == "__main__":
    test_add()
```

Struktura testu / unittest

Biblioteka **unittest**:

```
import unittest

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

class TestMath(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)

if __name__ == "__main__":
    unittest.main()
```

GWT - definicja

Given-When-Then to struktura stosowana w testach akceptacyjnych i BDD (Behavior Driven Development). Pomaga w precyzyjnym opisaniu warunków testu, akcji oraz oczekiwanych rezultatów.

GWT - przykład

- **Given**: Użytkownik wprowadza poprawne dane logowania.
- **When**: Naciśnie przycisk **Zaloguj**.
- **Then**: Zostanie przekierowany do strony głównej.

AAA - definicja

Arrange-Act-Assert jest bardziej klasycznym podejściem stosowanym w testach jednostkowych, gdy zależy nam na jasnym podziale na etap przygotowania, wykonania i weryfikacji wyników. Jest świetnym wyborem do testowania algorytmów, metod i funkcji, które nie są zbyt związane z zachowaniem użytkownika czy systemu.

AAA - przykład

- **Arrange**: przygotuj zmienne `a = 2, b = 3`.
- **Act**: wykonaj `result = add(a, b)`.
- **Assert**: sprawdź, czy `result == 5`.

AAA - przykład

```
def add(x, y):
    return x + y

def test_add():
    # Arrange
    a = 2
    b = 3

    # Act
    result = add(a, b)

    # Assert
    assert result == 5, "2 + 3 = 5?"

test_add()
```

Własny moduł

Moduł **my_module.py**:

```
def greet():
    return "Hello!"

if __name__ == "__main__":
    print("my_module.py")
```

Import modułu **my_module.py** do modułu **main.py**:

```
import my_module

print(my_module.greet())
```

Koniec