
unit testing makes you happy

Ewa Bielska
Pyladies Poznań

22 kwietnia 2015

unit testing makes you happy

Pliki

- <https://github.com/bielski/pyladies>
 - <https://www.dropbox.com/s/is3v0vjijt1hs7j/unitTests.zip?dl=0>
-

Kim jest Pylady/Pylord?

1. Nazywam się...
 2. Pracuję w...
 3. Interesuję się...
-

Plan działania

1. Testy jednostkowe
2. Moduł *unittest*
3. 
4. Testowanie mutacyjne
5. 
6. Arrange, act, assert
7. Mockowanie
8. 

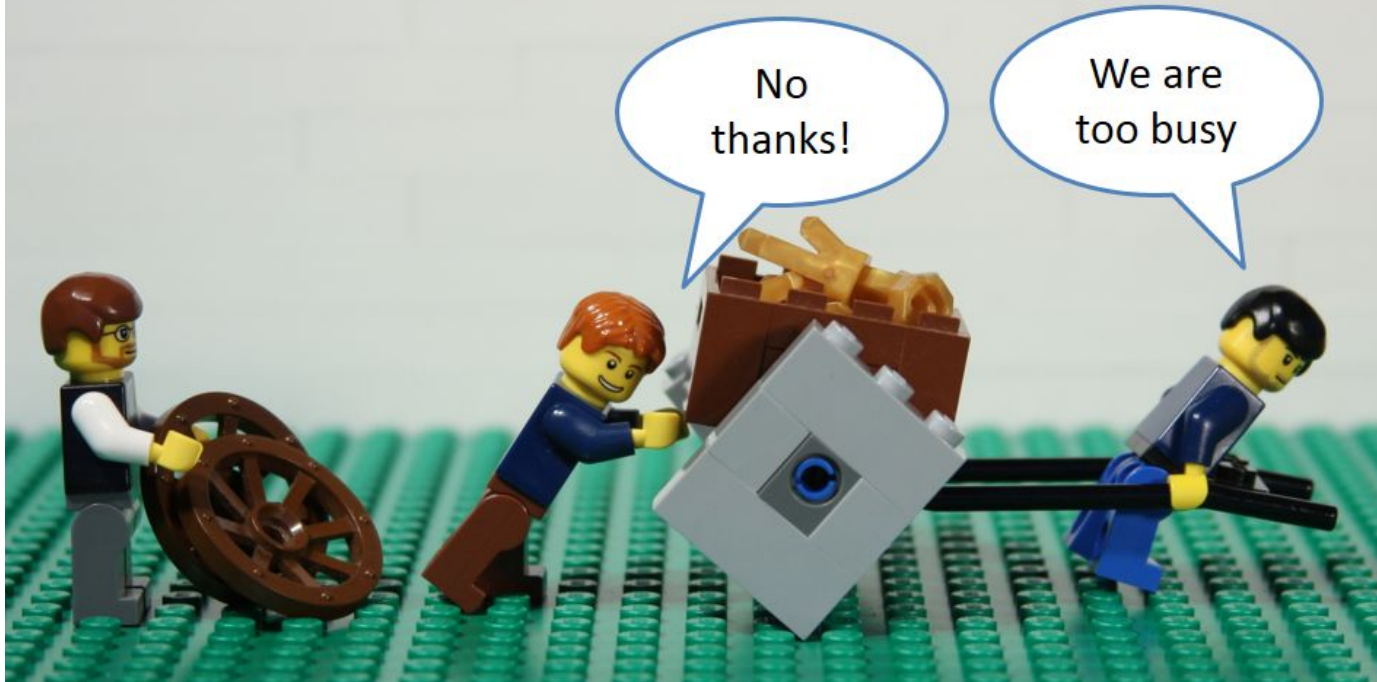
Cel dzisiejszego spotkania

- napisanie pierwszego testu jednostkowego
 - przeprowadzenie prostego mutowania
 - użycie mockowania w teście jednostkowym
-

unit testing makes you happy

- łatwiejsze wprowadzanie zmian i naprawa błędów
 - sprawniejsza refaktoryzacja kodu
 - dokumentacja projektu
 - znalazłeś błąd > napisz test
-

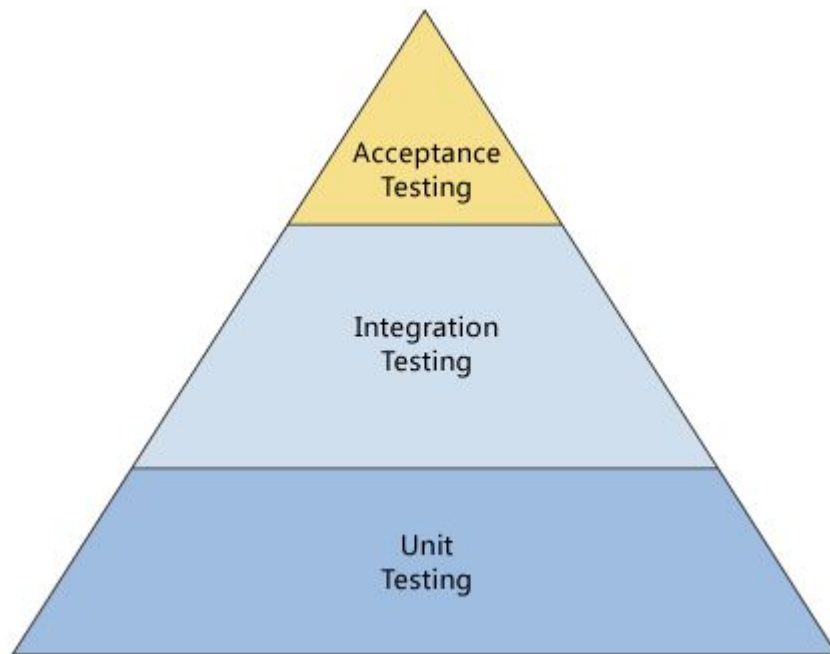
Are you too busy to improve?



Testy jednostkowe

- testują możliwie małą jednostkę kodu
 - sprawdzają czy zachowanie jest zgodne z oczekiwanym (**asercje**)
 - powinny być niezależne
 - powinny wykonywać się szybko
-

Testy jednostkowe



TDD

“TDD is **not** about testing. It is about development and design. The resulting unit tests are an extremely useful by-product.”

TDD

red > green > refactor

Plan działania

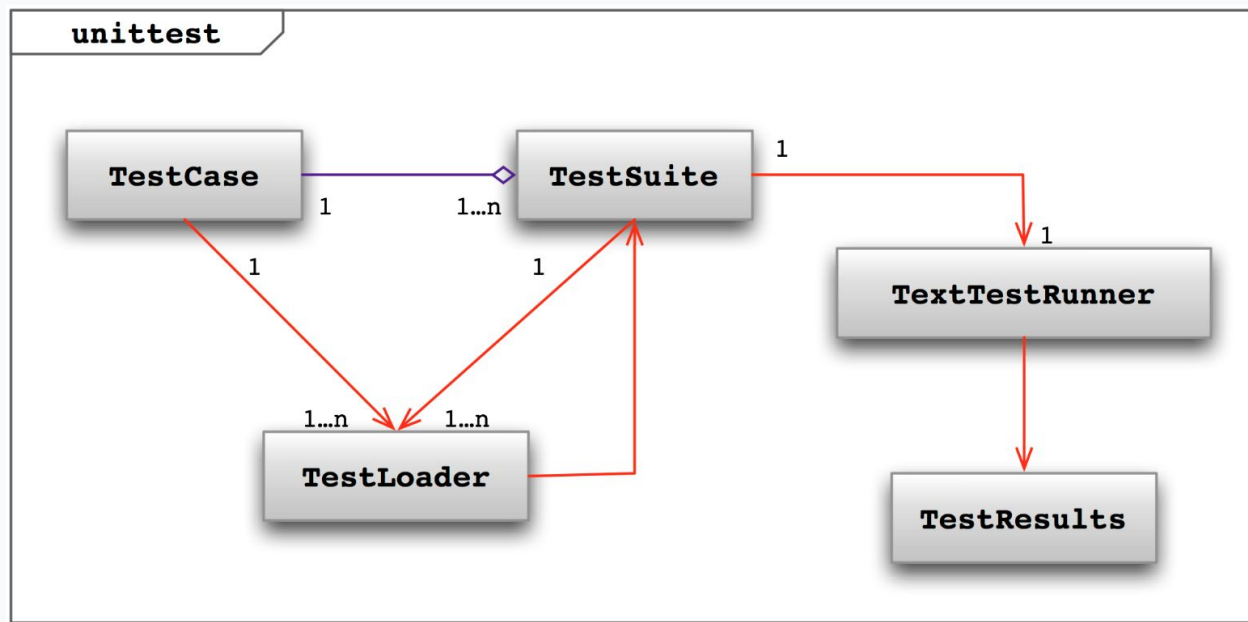
1. Testy jednostkowe
2. Moduł *unittest*
3. 
4. Testowanie mutacyjne
5. 
6. Arrange, act, assert
7. Mockowanie
8. 



moduł *unittest*

- dostępny od wersji Python 2.1
 - wcześniejsza nazwa PyUnit
 - korzysta z JUnit, opartego na bibliotece do testowania w Smalltalk
 - wspiera automatyzację testów jednostkowych
-

moduł *unittest*





moduł *unittest*



1. Wykorzystanie biblioteki **unittest** (Python 2.X)

Cel: pierwszy test jednostkowy

- import biblioteki
- stworzenie klasy dziedziczącej z klasy `unittest.TestCase`
- napisanie testów dla metod klasy *String* - *upper*, *isUpper*, *split*
- użycie asercji - *assertEqual*, *assertTrue*, *assertRaises*
- dodanie funkcji *unittest.main()*
- uruchomienie testu (wewnątrz IDE/z linii poleceń)

```
$python3 projectPath/string_tests.py
```

```
$python3 -m unittest discover -s projectPath/ -p *_tests.py
```

Plan działania

1. Testy jednostkowe
2. Moduł *unittest*
3. 
4. Testowanie mutacyjne
5. 
6. Arrange, act, assert
7. Mockowanie
8. 



testowanie mutacyjne

- celowe wprowadzanie błędów (mutacji) w programie i sprawdzenie czy test je wychwyci

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



```
def add(x):  
    return x - x
```

<http://pitest.org/quickstart/mutators/>

testowanie mutacyjne

Po co testować testy?

- sprawdzenie efektywności testów
 - wykrycie potencjalnych błędów
 - lepszy wgląd w jakość kodu niż mierzenie pokrycia testami
-

testowanie mutacyjne

- mutant
- mutacja
- zabicie mutantu
- przetrwanie mutantu
- wynik mutacji





testowanie mutacyjne



1. Wykorzystanie biblioteki **mutPy** (Python 3.X)

Cel: zabić mutantą!

- instalacja

```
$python3 setup.py install
```


- stworzenie pliku *calculator.py* z funkcją *multiply(x, y)*

- stworzenie pliku *calculator_test.py* z testem do funkcji *multiply(x,y)*

- mutowanie

```
$mut.py --target calculator --unit-test calculator_test -m
```

Plan działania

1. Testy jednostkowe
2. Moduł *unittest*
3. 
4. Testowanie mutacyjne
5. 
6. Arrange, act, assert
7. Mockowanie
8. 





Arrange, Act, Assert

- wzorzec do formatowania struktury testu jednostkowego
 1. **Arrange** all necessary preconditions and inputs.
 2. **Act** on the object or method under test.
 3. **Assert** that the expected results have occurred.
-

One assert per method

- test sprawdza tylko jedną rzecz
 - nieudany test jednoznacznie wskazuje na powód błędu
 - test jest niezależny
 - test jest bardziej czytelny
-

Mockowanie



Mockowanie

“In short, mocking is creating objects that simulate the behavior of real objects.”

moduł *unittest.mock*



- umożliwia zastępowanie części testowanego systemu obiektami mocków
 - pozwala na sprawdzenie założeń poprzez użycie **asercji**
 - **Mock/MagicMock** odtwarzają wszystkie atrybuty i metody mockowanej klasy
 - mockowane atrybuty i metody można dowolnie konfigurować
 - dekorator **patch()** umożliwia “łatanie” atrybutów na poziomie modułu lub klasy
 - domyślnie patch() zwraca klasę MagicMock
-

Mockowanie



1. Wykorzystanie biblioteki **unittest.mock** (Python 3.3)

Cel: pierwsze mocki za płoty

```
$python3
```

```
>>> from unittest.mock import *
```

```
>>> Calculator.multiply = Mock(return_value=3)
```

```
>>> Calculator.multiply()
```

```
>>> Calculator.multiply(2, 111111)
```

```
>>> Calculator.multiply.assert_called_with(2,111111)
```

```
>>> Calculator.multiply = Mock(side_effect=KeyError('foo'))
```

```
>>> Calculator.multiply()
```

Mockowanie



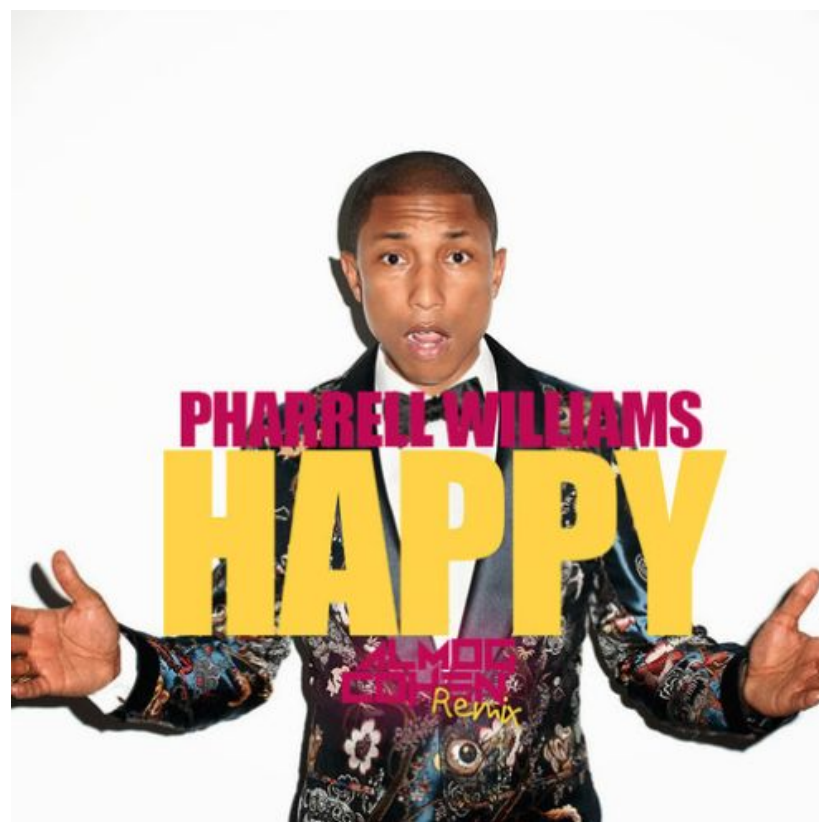
2. Wykorzystanie biblioteki **unittest.mock** (Python 3.3)

Cel: użycie “łatki” patch

- napisz funkcję usuwającą plik o podanej ścieżce
 - napisz test do tej funkcji (bez użycia mocków)
 - napisz test korzystający z biblioteki `unittest.mock`
-

Podsumowanie

1. Trzy przykładowe cechy testu jednostkowego to:
 - ...
 - ...
 - ...
 2. Co to jest mutacja?
 3. Co oznacza skrót TDD?
 4. Wyjaśnij *red, green, refactor*?
 5. Co to jest mock?
-



Python documentation

<https://docs.python.org/2/library/unittest.html>

Introduction

<http://cgoldberg.github.io/python-unittest-tutorial/>

<http://docs.python-guide.org/en/latest/writing/tests/>

Remove file tests

<http://www.toptal.com/python/an-introduction-to-mocking-in-python>
