Testowanie mutacyjne w języku Python

Konrad Hałas
konradhalas.blogspot.com

Testowanie mutacyjne (1/2)

- wyciągać wnioski z błędów aby uniknąć ich w przyszłości
- brak błędów? - trzeba je wygenerować
- dobre testy wyłapią błędy
- złe (niekompletne) ich nie zauważą – trzeba poprawić testy

Testowanie mutacyjne (2/2)

W szczególności:

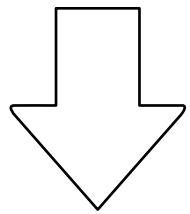
- obiekt testowany – **kod programu**
- testy – **testy jednostkowe**

Także:

- testy integracyjne
- specyfikacje i modele (np. maszyny stanowe)
- protokoły sieciowe
- obwody elektryczne

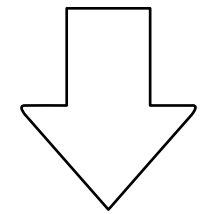
Przykład

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



```
1 def add(x):
2     return x * x
```

```
1 def test_add():
2     assertEqual(add(2), 4)
```



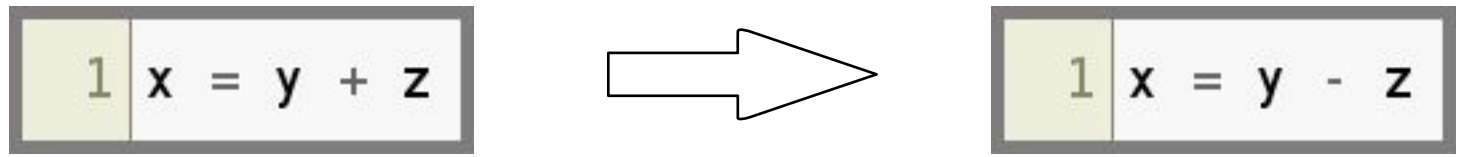
```
1 def test_add():
2     assertEqual(add(2), 4)
3     assertEqual(add(3), 6)
```

Nomenklatura

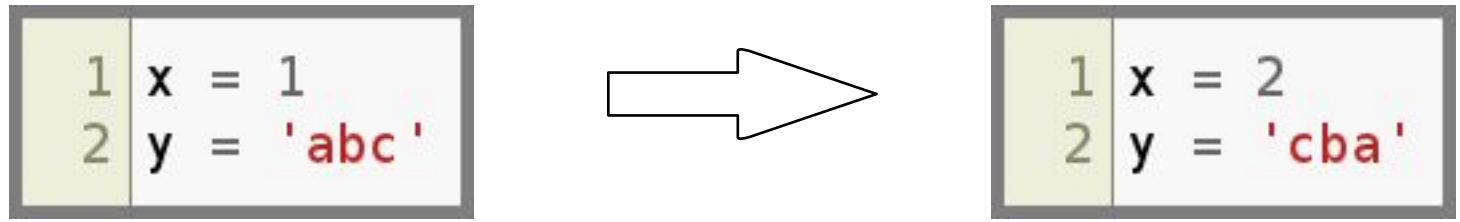
- **mutant** – program powstały poprzez wprowadzenie zmian symulujących błędy (mutacji) w programie oryginalnym
- **operator mutacji** – rodzaj wprowadzonej zmiany
- **zabicie mutanta** – wykrycie mutacji przez testy
- **przetrwanie mutanta** – niewykrycie mutacji przez testy
- **wynik mutacji** – stosunek mutantów zabitych do wszystkich mutantów

Operatory mutacji

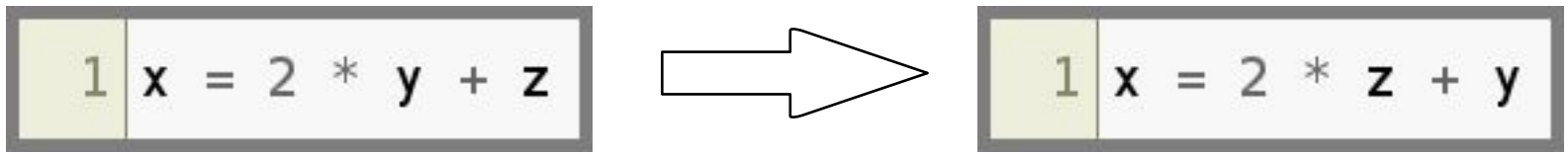
- zamiana operatorów



- zamiana stałych



- zamiana zmiennych



Mutanty równoważne

```
1 i = 0
2 while i < 10:
3       i += 1
4       (...)
```

==

```
1 i = 0
2 while i != 10:
3       i += 1
4       (...)
```

Duże koszty

Projekt	**Linie**	**Testy**	**Mutanty**	**Czas** CPU h
AspectJ Core	94902	321	47146	14
XStream	4837	137	17178	2
Joda-Time	25861	3447	13859	4.5

wyniki narzędzia Javalanche

Mutowanie języków dynamicznych

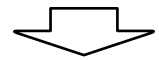
- nie znamy typu zmiennych do momentu wykonania programu

```
1 def add(x):
2     return x + x
3
4 add(2)       # 4
5 add('abc')   # 'abcabc'
6 add([1, 2])  # [1, 2, 1, 2]
```

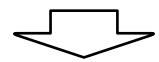
- mutacje mogą prowadzić do powstania mutantów niekompetentnych

Wprowadzanie mutacji w Pythonie

```
1 x = 10
2 y = x + 5
```



```
1  Module(body=
2      [Assign(
3          targets=[Name(id='x', ctx=Store())],
4          value=Num(n=10)),
5      Assign(
6          targets=[Name(id='y', ctx=Store())],
7          value=BinOp(
8              left=Name(id='x', ctx=Load()),
9              op=Add(),
10             right=Num(n=5)))])
```



```
1 d\x00\x00Z\x00\x00e\x00\x00d\x01
2 \x00\x17Z\x01\x00d\x02\x00S
```

Pester (1/2)

- działa na bazie Jestera (mutacje Javy)
- jedyne narzędzie do mutowania Pythona
- ostatnie aktualizacje w 2002 r.
- wykorzystuje technikę zamiany łańcuchów znaków i operator zamiany stałych

```
1 %true%false%false%true
2 %if %if 1 or %if(%if(1 or
3 %if %if 0 and %if(%if(0 and
4 %==%!=%!=%==
```

Pester (2/2)

```python
class Tested:
    # Jedynka w komentarzu 1
    def returnOneIfTrue(self, x):
        if x:
            return 1
        else:
            return 0
```

Changes to tested.py

```
class Tested:
        # Jedynka w komentarzu ~~1~~2
        def returnOneIfTrue(self, x):
                ~~if~~ if 1 or if 0 and x:
                        return ~~1~~2
                else:
                        return ~~0~~1
```

MutPy

- język implementacji – Python 3.x
- współpracuje z standardowym modułem testów jednostkowych
- wprowadza mutacje na poziomie drzewa składniowego
- interfejs tekstowy

Dziękuję!

pytania?