

B4B33RPH: Řešení problémů a hry  
Automatické testování softwaru. Vývoj řízený testy.

Petr Pošík

Katedra kybernetiky  
ČVUT FEL

<b>Úvod</b>	<b>2</b>
Připomenutí.....	3
<b>Testování</b>	<b>4</b>
Testování .....	5
FIRST .....	6
Modul doctest .....	7
xUnit Framework .....	8
<b>Vývoj řízený testy</b>	<b>9</b>
TDD: Vývoj řízený testy .....	10
TDD Ukázka .....	11
TDD Úvod .....	12
TDD Číslo 2 .....	13
TDD Číslo 3 .....	14
TDD Číslo 4 .....	15
TDD Číslo 5 .....	16
TDD Číslo 6 .....	17
TDD Číslo 8 .....	18
TDD Číslo 9 .....	19
TDD Čistý kód .....	20
TDD: Závěr .....	21

## Z minulé přednášky

### Testujte svůj kód!

- Dokud jej nevyzkoušíte (neotestujete) alespoň na několika příkladech, nevíte, zda funguje!
- Použijte nějaký framework pro automatické testování:
  - Snadná **tvorba** obsáhlé sady testů.
  - Snadné **přidávání** nových testů.
  - Snadné **opakované spouštění** všech testů.
  - Snadná **vizuální kontrola**, zda testy procházejí nebo selhávají.
- Spousta možností:
  - Náš vlastní modul `testing`.
  - Standardní modul `doctest`.
  - Standardní modul `unittest`.
  - `nosetest`, `pytest`, ...

## Automatizované testování

Zpracováno podle

Gerard Meszarosz: *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007.

### Testování

*Kvalita* softwaru z pohledu testování:

- Jak dobře kód splňuje specifikace?

Testování z pohledu QA týmu (acceptance tests, functional tests):

- Testujeme, protože jsme si jistí, že kód obsahuje chyby! (Nesplňuje specifikace zákazníka.)
- Testujeme poté, co je kód hotový.
- Obvykle black-box testování.
- Testování je spíš *měření* kvality softwaru, nikoli způsob, jak napsat kvalitní software.
- Zpětná vazba přichází příliš pozdě.
- V minulosti prováděny převážně ručně.

Testování z pohledu programátora (unit tests, integration tests):

- Testuji, protože si chci být jistý, že jednotka, na které právě pracuji, dělá to, co po ní chci. (Splňuje požadavky, které vznikly v důsledku designu architektury softwaru.)
- Obvykle white-box testování.
- V minulosti většinou dočasný kód, který se po otestování zahodil.

## Automatizované testy: F.I.R.S.T.

Automatizované testy by měly být F.I.R.S.T.

### Fast

- Pomalé testy → nebudete je spouštět často → chyby odhalíte pozdě

### Independent

- Jeden test by neměl nastavovat podmínky pro další test.
- Musí jít spustit každý test samostatně a celou sadu testů v jakémkoli pořadí.
- Závislé testy → jedna chyba spustí celý řetězec chyb v navazujících testech → složité hledání chyby.

### Repeatable

- Možnost *zopakovat* testy kýmkoli a kdekoli se stejným výsledkem.
- Testy lze spustit jen někde → budou se používat zřídka → chyby odhalíte pozdě

### Self-validating

- Dvoustavový výstup → snadné ověřit, zda test prošel nebo selhal.
- Složitý (dlouhý) výstup, který je nutno "ručně" zkontrolovat → málo časté testování → pozdní odhalení chyb.

### Timely

- Testy by měly být psány včas, ideálně před produkčním kódem.
- Testy psané po produkčním kódu → kód se špatně testuje → nebudete se chtít s jeho testováním zdržovat.

## Modul doctest

- Specialita Pythonu (opravte mě, pokud se pletu).
- Velmi vhodný pro jednoduché testy.
- Nevhodný pro složitější testy vyžadující přípravu a úklid.

```
class PrimesGenerator:
    """Prime numbers generator.

    >>> pg = PrimesGenerator()
    >>> pg.get_primes_up_to(1)
    []
    >>> pg.get_primes_up_to(2)
    [2]
    >>> pg.get_primes_up_to(3)
    [2, 3]
    >>> pg.get_primes_up_to(4)
    [2, 3]
    >>> pg.get_primes_up_to(5)
    [2, 3, 5]
    >>> pg.get_primes_up_to(20)
    [2, 3, 5, 7, 11, 13, 17, 19]
    """
    ...

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

## xUnit Framework

- Standardní testovací framework.
- Implementován v mnoha jazycích (naučte se ho, bude se vám hodit).
- V Pythonu implementován jako modul `unittest`.

```
import unittest
from primes3 import PrimesGenerator

class PrimesGeneratorTest(unittest.TestCase):

    known_values = ((0, []),
                    (1, [1]),
                    (2, [2]),
                    (3, [2,3]),
                    (4, [2,3]),
                    (5, [2,3,5]),
                    (7, [2,3,5,7]),
                    (20, [2,3,5,7,11,13,17,19]))

    def setUp(self):
        self.pg = PrimesGenerator()

    def test_get_primes_up_to(self):
        for limit, expected in self.known_values:
            observed = self.pg.get_primes_up_to(limit)
            self.assertEqual(observed, expected)
        ...

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

## Vývoj řízený testy

### TDD: Vývoj řízený testy

Tři zákony TDD (Test-driven development):

1. Nenapišeš ani kousek produkčního kódu, aniž bys předtím napsal selhávající test.
2. Nenapišeš větší část testu, než je potřebná k selhání (chybě).
3. Nenapišeš větší část produkčního kódu, než je potřebná ke splnění aktuálně selhávajícího testu.

Výsledek těchto pravidel:

- velmi krátký cyklus, v němž střídavě hrajete
  - roli zákazníka, který říká, co se má udělat (píšete test), a
  - roli programátora, který říká, jak se to má dělat (píšete kód, který splňuje aktuální specifikace).
- Testy a produkční kód se píší *společně* (testy o pár sekund napřed).
- Testy pak pokrývají všechny produkční kód!

## TDD Ukázka

Vytvořte funkci/metodu třídy na faktorizaci čísla na prvočíselné činitele.

- Vstup: číslo, které chceme rozložit
- Výstup: seznam prvočísel, jejichž součin je roven vstupnímu číslu

Jak byste postupovali?

P. Pošík © 2012

A4B99RPH: Řešení problémů a hry – 11 / 21

## TDD Ukázka: Úvodní fáze

Zakládáme test\_factorize.py

```
import unittest
from factorization import factorize
```

Po spuštění test\_factorize.py:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: No module named factorization
```

Zakládáme prázdný factorization.py

Po spuštění test\_factorize.py:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: cannot import name factorize
```

Upravujeme factorization.py:

```
def factorize():
    pass
```

Po spuštění test\_factorize.py:

```
--- Zadny vystup, kod bez chyby. ---
```

Upravujeme test\_factorize.py

```
import unittest
from factorization import factorize

class FactorizeTest(unittest.TestCase):
    pass

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

Po spuštění test\_factorize.py:

```
-----
Ran 0 tests in 0.000s
OK
builtins.SystemExit: False
```

P. Pošík © 2012

A4B99RPH: Řešení problémů a hry – 12 / 21

## TDD Ukázka: Test faktorizace čísla 2

Upravujeme test\_factorize.py

```
class FactorizeTest(unittest.TestCase):  
  
    def test_two(self):  
        observed = factorize(2)  
        self.assertEqual(observed, [2])
```

Upravujeme factorization.py:

```
def factorize(product):  
    pass
```

Upravujeme factorization.py:

```
def factorize(product):  
    return [2]
```

Po spuštění test\_factorize.py:

```
E  
=====  
ERROR: test_one (__main__.FactorizeTest)  
-----  
Traceback (most recent call last):  
  File "<wingdb_compile>", line 7, in test_one  
TypeError: factorize() takes no arguments (1 given)  
-----  
Ran 1 test in 0.000s
```

```
F  
=====  
FAIL: test_one (__main__.FactorizeTest)  
-----  
Traceback (most recent call last):  
  File "<wingdb_compile>", line 8, in test_one  
AssertionError: None != [2]  
-----  
Ran 1 test in 0.000s
```

```
.  
-----  
Ran 1 test in 0.000s
```

## TDD Ukázka: Test faktorizace čísla 3

Upravujeme test\_factorize.py

```
def test_three(self):  
    observed = factorize(3)  
    self.assertEqual(observed, [3])
```

Upravujeme factorization.py:

```
def factorize(product):  
    return [product]
```

Po spuštění test\_factorize.py:

```
F.  
=====  
FAIL: test_three (__main__.FactorizeTest)  
-----  
Traceback (most recent call last):  
  File "<wingdb_compile>", line 12, in test_three  
AssertionError: Lists differ: [2] != [3]  
  
First differing element 0:  
2  
3  
  
- [2]  
+ [3]  
-----  
Ran 2 tests in 0.016s
```

```
..  
-----  
Ran 2 tests in 0.000s
```

## TDD Ukázka: Test faktorizace čísla 4

Upravujeme test\_factorize.py

```
def test_four(self):
    observed = factorize(4)
    self.assertEqual(observed, [2,2])
```

Po spuštění test\_factorize.py:

```
F..
=====
FAIL: test_four (__main__.FactorizeTest)
-----
Traceback (most recent call last):
  File "<wingdb_compile>", line 16, in test_four
AssertionError: Lists differ: [4] != [2, 2]
[...snip...]
-----
Ran 3 tests in 0.000s
```

Upravujeme factorization.py:

```
def factorize(product):
    factors = []
    while product % 2 == 0:
        factors.append(2)
        product /= 2
    return factors
```

```
.F.
=====
FAIL: test_three (__main__.FactorizeTest)
-----
Traceback (most recent call last):
  File "<wingdb_compile>", line 12, in test_three
AssertionError: Lists differ: [] != [3]
[...snip...]
-----
Ran 3 tests in 0.016s
```

Upravujeme factorization.py:

```
def factorize(product):
    factors = []
    while product % 2 == 0:
        factors.append(2)
        product /= 2
    if product != 1:
        factors.append(product)
    return factors
```

```
....
-----
Ran 3 tests in 0.000s
```

## TDD Ukázka: Test faktorizace čísla 5

Upravujeme test\_factorize.py

```
def test_five(self):
    observed = factorize(5)
    self.assertEqual(observed, [5])
```

Po spuštění test\_factorize.py:

```
....
-----
Ran 4 tests in 0.000s
```

### TDD Ukázka: Test faktorizace čísla 6

Upravujeme test\_factorize.py

```
def test_six(self):  
    observed = factorize(6)  
    self.assertEqual(observed, [2,3])
```

Po spuštění test\_factorize.py:

```
.....  
-----  
Ran 5 tests in 0.000s
```

Test faktorizace čísla 7 vynecháváme, je to stejný případ, jako pro 3 a 5.

### TDD Ukázka: Test faktorizace čísla 8

Upravujeme test\_factorize.py

```
def test_eight(self):  
    observed = factorize(8)  
    self.assertEqual(observed, [2,2,2])
```

Po spuštění test\_factorize.py:

```
.....  
-----  
Ran 6 tests in 0.000s
```



## TDD Ukázka: Test faktorizace čísla 9

Upravujeme test\_factorize.py

```
def test_nine(self):
    observed = factorize(9)
    self.assertEqual(observed, [3,3])
```

Po spuštění test\_factorize.py:

```
...F...
-----
FAIL: test_nine (__main__.FactorizeTest)
-----
Traceback (most recent call last):
  File "<wingdb_compile>", line 32, in test_nine
AssertionError: Lists differ: [9] != [3, 3]
[...snip...]
-----
Ran 7 tests in 0.000s
```

Upravujeme factorization.py:

```
def factorize(product):
    factors = []
    for factor in range(2,product+1):
        while product % factor == 0:
            factors.append(factor)
            product /= factor
    return factors
```

```
.....
-----
Ran 7 tests in 0.015s
```

- Jsme schopni přijít na nějaký další test, kde by náš kód selhal?
- Nevadí náhodou, že jako faktory bereme všechna čísla a nikoli jen prvočísla? Jak by se kód lišil?

## TDD Ukázka: Je naše funkce napsaná čistě?

Stávající factorization.py:

```
def factorize(product):
    factors = []
    for factor in range(2,product+1):
        while product % factor == 0:
            factors.append(factor)
            product /= factor
    return factors
```

```
.....
-----
Ran 7 tests in 0.015s
```

Přepsaný factorization.py:

```
def factorize(product):
    factors = []
    for factor in range(2,product+1):
        product, factors_subset = factor_out(product, factor)
        factors.extend(factors_subset)
    return factors

def factor_out(product, factor):
    factors = []
    while product % factor == 0:
        factors.append(factor)
        product /= factor
    return product, factors
```

```
.....
-----
Ran 7 tests in 0.000s
```

Která z verzí se vám jeví přehlednější/čitelnější?

## TDD: Závěr

### Testy

- slouží jako specifikace.
- slouží jako dokumentace.
- pomáhají pochopit algoritmus.
- pomáhají předejít zbytečným složitostem v kódu.
- určují, kdy “je hotovo”.
- pomáhají zajistit, abychom úpravami do kódu nevnegli nové chyby.