A4B99RPH: Řešení problémů a hry
Unit testing. Vývoj řízený testy.

Petr Pošík

Katedra kybernetiky
ČVUT FEL

Testíky na cvičení

Zpětná vazba z prvního testíku

Zadání: Pokračujte s kódem, který máte z domácího cvičení a rozšiřte ho

- ✔ o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- ✔ o metodu `norm(self)` realizující výpočet velikosti vektoru.

Zpětná vazba z prvního testíku

Zadání: Pokračujte s kódem, který máte z domácího cvičení a rozšiřte ho

- ✔ o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- ✔ o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- ✔ Jak se měl jmenovat soubor, který se měl odevzdat?

Zpětná vazba z prvního testíku

Zadání: Pokračujte s kódem, který máte z domácího cvičení a rozšiřte ho

- ✔ o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- ✔ o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- ✔ Jak se měl jmenovat soubor, který se měl odevzdat?
- ✔ Jak se měla jmenovat třída definovaná v tomto souboru?

Zpětná vazba z prvního testíku

Zadání: Pokračujte s kódem, který máte z domácího cvičení a rozšiřte ho

- ✔ o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- ✔ o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- ✔ Jak se měl jmenovat soubor, který se měl odevzdat?
- ✔ Jak se měla jmenovat třída definovaná v tomto souboru?
- ✔ Jaké metody tato třída měla mít?

Zpětná vazba z prvního testíku

Zadání: Pokračujte s kódem, který máte z domácího cvičení a rozšiřte ho

- ✔ o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- ✔ o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- ✔ Jak se měl jmenovat soubor, který se měl odevzdat?
- ✔ Jak se měla jmenovat třída definovaná v tomto souboru?
- ✔ Jaké metody tato třída měla mít?
- ✔ Objekt jakého typu měla vracet metoda `__add__`?

Zpětná vazba z prvního testíku

Zadání: Pokračujte s kódem, který máte z domácího cvičení a rozšiřte ho

- ✔ o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- ✔ o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- ✔ Jak se měl jmenovat soubor, který se měl odevzdat?
- ✔ Jak se měla jmenovat třída definovaná v tomto souboru?
- ✔ Jaké metody tato třída měla mít?
- ✔ Objekt jakého typu měla vracet metoda `__add__`?
- ✔ Jaký význam má v Pythonu odsazení? Jak by se mělo správně realizovat?

Zpětná vazba z prvního testíku

Zadání: Pokračujte s kódem, který máte z domácího cvičení a rozšiřte ho

- ✔ o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- ✔ o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- ✔ Jak se měl jmenovat soubor, který se měl odevzdat?
- ✔ Jak se měla jmenovat třída definovaná v tomto souboru?
- ✔ Jaké metody tato třída měla mít?
- ✔ Objekt jakého typu měla vracet metoda `__add__`?
- ✔ Jaký význam má v Pythonu odsazení? Jak by se mělo správně realizovat?

Proč jste tyto chyby neodhalili?

Zpětná vazba z prvního testíku

Zadání: Pokračujte s kódem, který máte z domácího cvičení a rozšiřte ho

- ✔ o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- ✔ o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- ✔ Jak se měl jmenovat soubor, který se měl odevzdat?
- ✔ Jak se měla jmenovat třída definovaná v tomto souboru?
- ✔ Jaké metody tato třída měla mít?
- ✔ Objekt jakého typu měla vracet metoda `__add__`?
- ✔ Jaký význam má v Pythonu odsazení? Jak by se mělo správně realizovat?

Proč jste tyto chyby neodhalili?

Jak otestovat vlastní kód?

Python Shell

Spustit Python shell a zkusit svůj kód použít:

```
>>> from vectors import MyVector
>>> a = MyVector([1,1,1])
>>> b = MyVector([1,2,3])
>>> c = a+b
>>> type(c)
<class 'vectors.MyVector'>
>>> c.get_vector()
[2, 3, 4]
```

Python Shell

Spustit Python shell a zkusit svůj kód použít:

```
>>> from vectors import MyVector
>>> a = MyVector([1,1,1])
>>> b = MyVector([1,2,3])
>>> c = a+b
>>> type(c)
<class 'vectors.MyVector'>
>>> c.get_vector()
[2, 3, 4]
```

- ✔ Odhalíte snadno všechny výše zmíněné chyby.
- ✔ Někdy je nutné přepnout se do pracovního adresáře (`import os; os.chdir()`).
- ✔ Bývají problémy s re-importem již jednou importovaného modulu.
  - ✘ V Pythonu 2x: `reload(module)`
  - ✘ V Pythonu 3x: `import imp; imp.reload(module)`
  - ✘ . . . ale i tak to není nejlepší řešení

Manuální testy

Využít sekci `if __name__=='__main__':` ke spuštění testu:

```
if __name__=="__main__":
    from vectors import MyVector
    a = MyVector([1,1,1])
    b = MyVector([1,2,3])
    c = a+b
    print(type(c))
    print(c.get_vector())
```

Manuální testy

Využít sekci `if __name__=='__main__':` ke spuštění testu:

```
if __name__=="__main__":
    from vectors import MyVector
    a = MyVector([1,1,1])
    b = MyVector([1,2,3])
    c = a+b
    print(type(c))
    print(c.get_vector())
```

- ✔ Není nutné se starat o pracovní adresář.
- ✔ "Test" bude fungovat i bez importu modulu - nemusíte odhalit chybné jméno modulu.
- ✔ Import modulu ale můžete uvést. Nic tím nezkazíte a chybné jméno odhalíte.

Automatické testování

Využít metod automatického testování:

- ✔ doctest, unittest, či jiný framework
- ✔ lze spouštět i velké množství testů s automatickou kontrolou výsledků

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

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

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.

Programátorské testování

Doufejme, že sami vytváříte nějaký testovací kód ke svým funkcím/metodám:

```
if __name__ == "__main__":
    pg = PrimesGenerator()
    print("Primes up to 0: ", pg.get_primes_up_to(0))
    print("Primes up to 1: ", pg.get_primes_up_to(1))
    print("Primes up to 2: ", pg.get_primes_up_to(2))
    print("Primes up to 3: ", pg.get_primes_up_to(3))
    print("Primes up to 4: ", pg.get_primes_up_to(4))
    print("Primes up to 5: ", pg.get_primes_up_to(5))
    print("Primes up to 6: ", pg.get_primes_up_to(6))
    print("Primes up to 20: ", pg.get_primes_up_to(20))
```

a že jste kontrolovali výstup:

```
Primes up to 0:  []
Primes up to 1:  []
Primes up to 2:  [2]
Primes up to 3:  [2, 3]
Primes up to 4:  [2, 3]
Primes up to 5:  [2, 3, 5]
Primes up to 6:  [2, 3, 5]
Primes up to 20:  [2, 3, 5, 7, 11, 13, 17, 19]
Primes up to 100:  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 6
>>>
```

Automatizované testy: F.I.R.S.T.

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

Fast. Testy by měly být *rychlé*. Když nebudou rychlé, nebudete je chtít spouštět často. Když je nebudete spouštět často, neodhalíte chyby dostatečně brzo.

Automatizované testy: F.I.R.S.T.

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

Fast. Testy by měly být *rychlé*. Když nebudou rychlé, nebudete je chtít spouštět často. Když je nebudete spouštět často, neodhalíte chyby dostatečně brzo.

Independent. Testy by měly být *nezávislé*. Jeden test by neměl nastavovat podmínky pro další test. Měli byste být schopni spustit každý test samostatně a celou sadu testů v jakémkoli pořadí. Pokud testy nejsou nezávislé, chyba v jednom testu spustí celý řetězec chyb v navazujících testech. Hledání chyby je pak složitější.

Automatizované testy: F.I.R.S.T.

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

Fast. Testy by měly být *rychlé*. Když nebudou rychlé, nebudete je chtít spouštět často. Když je nebudete spouštět často, neodhalíte chyby dostatečně brzo.

Independent. Testy by měly být *nezávislé*. Jeden test by neměl nastavovat podmínky pro další test. Měli byste být schopni spustit každý test samostatně a celou sadu testů v jakémkoli pořadí. Pokud testy nejsou nezávislé, chyba v jednom testu spustí celý řetězec chyb v navazujících testech. Hledání chyby je pak složitější.

Repeatable. Testy by mělo být možné *zopakovat* kýmkoli a kdekoli se stejným výsledkem. Když jste schopni spustit testy se správným výsledkem jen někde, brání vám to v jejich častém spuštění a chyby neodhalíte dostatečně brzo.

Automatizované testy: F.I.R.S.T.

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

Fast. Testy by měly být *rychlé*. Když nebudou rychlé, nebudete je chtít spouštět často. Když je nebudete spouštět často, neodhalíte chyby dostatečně brzo.

Independent. Testy by měly být *nezávislé*. Jeden test by neměl nastavovat podmínky pro další test. Měli byste být schopni spustit každý test samostatně a celou sadu testů v jakémkoli pořadí. Pokud testy nejsou nezávislé, chyba v jednom testu spustí celý řetězec chyb v navazujících testech. Hledání chyby je pak složitější.

Repeatable. Testy by mělo být možné *zopakovat* kýmkoli a kdekoli se stejným výsledkem. Když jste schopni spustit testy se správným výsledkem jen někde, brání vám to v jejich častém spuštění a chyby neodhalíte dostatečně brzo.

Self-validating. Testy by měly mít dvoustavový výstup. Díky tomu je testy schopen ověřit, zda prošel nebo selhal. Neměli byste být nuceni procházet nějaký výpis výsledků, abyste rozhodli, zda test prošel nebo ne. Nejsou-li testy schopné rozhodnout sami, zda prošly nebo ne, nebudete chtít testovat tak často…

Automatizované testy: F.I.R.S.T.

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

Fast. Testy by měly být *rychlé*. Když nebudou rychlé, nebudete je chtít spouštět často. Když je nebudete spouštět často, neodhalíte chyby dostatečně brzo.

Independent. Testy by měly být *nezávislé*. Jeden test by neměl nastavovat podmínky pro další test. Měli byste být schopni spustit každý test samostatně a celou sadu testů v jakémkoli pořadí. Pokud testy nejsou nezávislé, chyba v jednom testu spustí celý řetězec chyb v navazujících testech. Hledání chyby je pak složitejší.

Repeatable. Testy by mělo být možné *zopakovat* kýmkoli a kdekoli se stejným výsledkem. Když jste schopni spustit testy se správným výsledkem jen někde, brání vám to v jejich častém spuštění a chyby neodhalíte dostatečně brzo.

Self-validating. Testy by měly mít dvoustavový výstup. Díky tomu je testy schopen ověřit, zda prošel nebo selhal. Neměli byste být nuceni procházet nějaký výpis výsledků, abyste rozhodli, zda test prošel nebo ne. Nejsou-li testy schopné rozhodnout sami, zda prošly nebo ne, nebudete chtít testovat tak často…

Timely. Testy by měly být psány včas, ideálně před produkčním kódem. Píšete-li testy až po produkčním kódu, často narazíte na to, že se kód testuje špatně. Rozhodnete se pak, že se s jeho testováním nebudete zdržovat.

Modul doctest

- ✔ specialita Pythonu (opravte mě, pokud se pletu)
- ✔ velmi vhodný pro jednoduché testy nevyžadující žádnou přípravu a úklid
- ✔ složitější testy lze dělat také, ale někomu to přijde . . . nepřirozené

```
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(7)
    [2, 3, 5, 7]
    >>> 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`.

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, [] ),
                     ( 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. Nenapíšeš ani kousek produkčního kódu, aniž bys předtím napsal selhávající unit test.

TDD: Vývoj řízený testy

Tři zákony TDD (Test-driven development):

1. Nenapíšeš ani kousek produkčního kódu, aniž bys předtím napsal selhávající unit test.
2. Nenapíšeš větší část unit testu, než je potřebná k selhání (chybě).

TDD: Vývoj řízený testy

Tři zákony TDD (Test-driven development):

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

TDD: Vývoj řízený testy

Tři zákony TDD (Test-driven development):

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

Výsledek těchto pravidel:

- ✔ velmi krátký cyklus, v němž střídavě hrajete
  - ✘ roli zákazníka, který říká, co se má dělat (píšete test), a
  - ✘ roli programátora, který říká, jak se to má dělat (upravujete kód).
- ✔ Testy a produkční kód se píší *společně* (testy o pár sekund napřed).
- ✔ Testy pak pokrývají všechen 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

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? Funkci na generování prvočísel už máme…

TDD Ukázka: Úvodní fáze

Zakládáme `test_factorize.py`

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

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
```

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`

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
```

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
```

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`:

```
--- Žádný výstup, kód bez chyby. ---
```

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`:

```
--- Žádný výstup, kód bez chyby. ---
```

Upravujeme `test_factorize.py`

```
import unittest
from factorization import factorize

class FactorizeTest(unittest.TestCase):
  pass

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

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`:

```
--- Žádný výstup, kód 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
```

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])
```

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])
```

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
```

Upravujeme `factorization.py`:

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

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])
```

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
```

Upravujeme `factorization.py`:

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

```
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
```

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])
```

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
```

Upravujeme `factorization.py`:

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

```
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
```

Upravujeme `factorization.py`:

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

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])
```

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
```

Upravujeme `factorization.py`:

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

```
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
```

Upravujeme `factorization.py`:

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

```
.
----------------------------------------------------------------------
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])
```

TDD Ukázka: Test faktorizace čísla 3

Upravujeme `test_factorize.py`

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

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
```

Upravujeme `factorization.py`:

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

TDD Ukázka: Test faktorizace čísla 3

Upravujeme `test_factorize.py`

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

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
```

Upravujeme `factorization.py`:

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

```
..
----------------------------------------------------------------------
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])
```

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

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(multiple):
  factors = []
  while multiple % 2 == 0:
    factors.append(2)
    multiple /= 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
```

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

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(multiple):
  factors = []
  while multiple % 2 == 0:
    factors.append(2)
    multiple /= 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(multiple):
  factors = []
  while multiple % 2 == 0:
    factors.append(2)
    multiple /= 2
  if multiple != 1:
    factors.append(multiple)
  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])
```

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])
```

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
```

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])
```

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])
```

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
```

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

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

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

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

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

- ✔ Jsme schopni přijít na nějaký další test, kde by náš kód selhal?

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(multiple):
  factors = []
  for factor in range(2,multiple+1):
    while multiple % factor == 0:
      factors.append(factor)
      multiple /= 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?

Testování třídy Game

Typické použití třídy Game:

```
>>> g = Game(playerA, playerB, payoff_matrix, n_iterations)
>>> g.run()
>>> g.get_players_payoffs()
```

Co můžeme na třídě Game testovat?

Testování třídy Game

Typické použití třídy `Game`:

```
>>> g = Game(playerA, playerB, payoff_matrix, n_iterations)
>>> g.run()
>>> g.get_players_payoffs()
```

Co můžeme na třídě `Game` testovat?

- ✔ Metoda `get_players_payoffs()` vrací `(None,None)` před spuštěním metody `run()`.

Testování třídy Game

Typické použití třídy `Game`:

```
>>> g = Game(playerA, playerB, payoff_matrix, n_iterations)
>>> g.run()
>>> g.get_players_payoffs()
```

Co můžeme na třídě `Game` testovat?

- ✔ Metoda `get_players_payoffs()` vrací `(None,None)` před spuštěním metody `run()`.
- ✔ Metoda `run()` volá metody `move()` a `record_opponents_move()` obou hráčů přesně `n_iterations`-krát.
- ✔ Metoda `run()` volá metody `move()` a `record_opponents_move()` střídavě, začíná se metodou `move()`.
- ✔ Metoda `run()` se chová k oběma hráčům rovnocenně, tj. nepředává jednomu z hráčů aktuální tah protihráče.
- ✔ ...

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 nevnesli nové chyby.