A4B99RPH: Řešení problémů a hry
Čistý kód. Vývoj řízený testy.

Petr Pošík

Katedra kybernetiky
ČVUT FEL

Zadání

Implementujte algoritmus *Eratostenova síta* na hledání prvočísel.

- ✔ Algoritmus implementujte jako funkci nebo jako metodu nějaké třídy.
- ✔ Chceme najít všechna prvočísla menší nebo rovna N (N bude argumentem funkce/metody).
- ✔ Algoritmus samozřejmě můžete rozdělit do několika funkcí/metod.
- ✔ Jména modulů/tříd/metod/funkcí zvolte sami.

Požadavek: odevzdaný kód by měl být *co nejčistší*,

- ✔ ať už si pod tímto termínem představujete cokoli.
- ✔ Kód "vyšperkujte" tak, abyste jej mohli ukazovat budoucím zaměstnavatelům jako příklad kvalitního kódu.

Cíl:

- ✔ Zjistit, co považujete za *čistý kód*.
- ✔ Na příští přednášce si ukážeme typické nedostatky.

Hodnocení:

- ✔ Nevyžadujeme, aby kód fungoval správně.
- ✔ Každá odevzdaná úloha bude odměněna 2 body, bude-li z ní znát, že se jedná o Eratostenovo síto.
- ✔ Termín: pondělí, 17.10.2011, 23:59

Odevzdaná řešení a jejich neduhy

Zaznamenáno:

- ✔ Několik výskytů poměrně čistého kódu!
- ✔ Asi 3 neobvyklá řešení.

Většinou ale:

- ✔ Algoritmus nalezený na webu přepsán do Pythonu:
  - ✘ Algoritmus v pseudokódu $\neq$ čistý kód!
  - ✘ Implementováno jako 1 velká funkce.
- ✔ Proměnné pojmenované jako `i`, `j`, `k`.
- ✔ Komentáře často nebyly vůbec, nebo byly naopak u každého řádku.
- ✔ Často byl použit seznam kandidátských čísel, který
  - ✘ byl procházen ve `for`-cyklu a zároveň
  - ✘ z něj byly odstraňovány prvky.

```
seznam = [...]
for prvek in seznam:
    if nejaka_podminka(prvek):
        seznam.remove(prvek)
```

  - ✘ Technika, která v jiných jazycích *zaručeně* způsobí problém!!!

Typický příklad řešení

```
# This function generates prime numbers up to
# a user specified maximum. The algorithm
# used is the Sieve of Eratosthenes.
#
# Eratosthenes of Cyrene, b. c. 276 BC,
# Cyrene, Libya -- d. c. 194 BC, Alexandria.
# The first man to calculate the circumference
# of the Earth. Also known for working on
# calendars with leap years and ran
# the library at Alexandria.
#
# The algorithm is quite simple.
# Given an array of integers starting at 2,
# cross out all multiples of 2.
# Find the next uncrossed integer,
# and cross out all of its multiples.
# Repeat until you have passed
# the maximum value.
#
# @author hugo
# @version 1
```

```
def generate_primes_up_to(max_value):
    """Find primes up to the max_value
    using the Sieve of Eratosthenes.

    """
    if max_value >= 2:  # There are some primes
        # Initialize the list (incl. 0)
        f = [False for i in range(max_value+1)]
        # Get rid of the known non-primes
        f[0] = f[1] = True
        # Run the sieve
        for i in range(2, len(f)):
            if not f[i]:   # i is uncrossed
                # cross out its multiples
                for j in range(2*i, len(f), i):
                    f[j] = True
        # Find the primes and put them in a list
        primes = [i for i in range(len(f)) if not f[i]]
        return primes
    else:   # max_value < 2
        return list() # no primes, return empty list
```


Clean Code

Zpracováno podle
Robert C. Martin: ***Clean Code: A Handbook of Agile Software Craftsmanship*****,**
Prentice Hall, 2008.

Co je čistý kód?

Bjarne Stroustrup, autor jazyka C++ a knihy "The C++ Programming Language":

> I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.

Grady Booch, autor knihy "Object Oriented Analysis and Design with Applications":

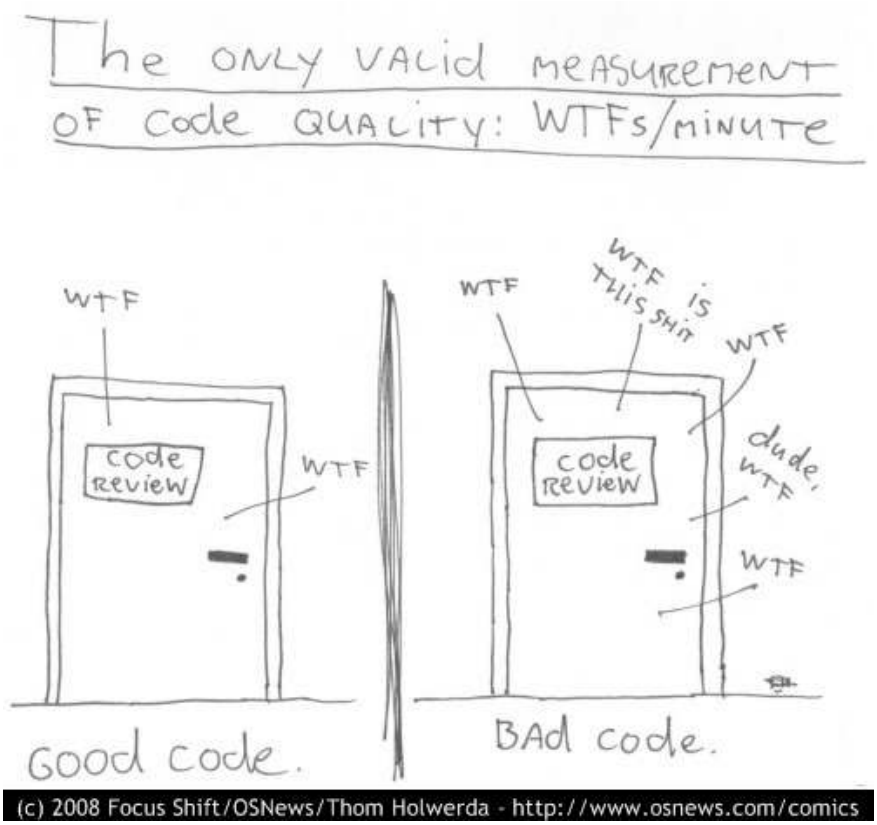
> Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.

Dave Thomas, zakladatel OTI, kmotr Eclipse:

> Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API.

Čistý kód v praxi

Jediné správné měřítko kvality kódu: Co-to-k-čerty za minutu



Smysluplná jména

- ✔ Vymyslet dobrá jména je **velmi těžké**! Věnujte tomu dostatečnou pozornost!
- ✔ Nebojte se jméno změnit, přijdete-li na lepší!
- ✔ Dobré jméno **odhaluje autorův záměr** (intention-revealing).
- ✔ Pokud jméno vyžaduje komentář, neodhaluje záměr. Porovnejte:

```
self.d = 0 # Elapsed time in days
```

versus

```
self.elapsed_time_in_days = 0
```

- ✔ Názvy tříd: **podstatná jména** (s přívlastky):
  - ✘ `Customer`, `WikiPage`, `AddressParser`
  - ✘ `Filter`, `StupidFilter`, `Corpus`, `TrainingCorpus`
- ✔ Názvy funkcí/metod: **slovesa** (s předmětem):
  - ✘ `post_payment`, `delete_page`, `save`
  - ✘ `train`, `test`, `get_email`
- ✔ Označujte jeden koncept vždy stejným slovem! Nepoužívejte stejné slovo k více účelům!
- ✔ Nebojte se dlouhých jmen!
  - ✘ Dlouhé popisné jméno je lepší než dlouhý popisný komentář.
  - ✘ Čím delší oblast platnosti proměnné, tím popisnější jméno by měla mít.
- ✔ Používejte pojmenované konstanty místo magických čísel v kódu!

Eratostenovo síto: smysluplná jména

```python
def generate_primes_up_to(max_value):
    """Find primes up to the max_value
    using the Sieve of Eratosthenes.

    """
    if max_value >= 2:  # There are some primes
        # Initialize the list (incl. 0)
        f = [False for i in range(max_value+1)]
        # Get rid of the known non-primes
        f[0] = f[1] = True
        # Run the sieve
        for i in range(2, len(f)):
            if not f[i]:   # i is uncrossed
                # cross out its multiples
                for j in range(2*i, len(f), i):
                    f[j] = True
        # Find the primes and put them in a list
        primes = [i for i in range(len(f)) if not f[i]]
        return primes
    else:   # max_value < 2
        return list() # no primes, return empty list
```

```python
def generate_primes_up_to(max_value):
    """Find primes up to the max_value
    using the Sieve of Eratosthenes.

    """
    if max_value >= 2:  # There are some primes
        # Initialize the list (incl. 0)
        crossed_out = [
            False for i in range(max_value+1)]
        # Get rid of the known non-primes
        crossed_out[0] = crossed_out[1] = True
        # Run the sieve
        for number in range(2, len(crossed_out)):
            if not crossed_out[number]:
                # cross out its multiples
                for multiple in \
                range(2*number, len(crossed_out), number):
                    crossed_out[multiple] = True
        # Find the primes and put them in a list
        primes = [i for i in range(len(crossed_out))
                  if not crossed_out[i]]
        return primes
    else:   # max_value < 2
        return list() # no primes, return empty list
```

Další smysluplná jména budou následovat!!!

Komentáře

Kdyby byly prog. jazyky dostatečně expresivní, nepotřebovali bychom komentáře!

- ✔ Čistý kód komentáře (skoro) nepotřebuje!
- ✔ Komentáře kompenzují naše selhání vyjádřit se v prog. jazyce. Porovnej:

```python
# Check to see if the employee is eligible for full benefits
if (employee.flags & HOURLY_FLAG) and (employee.age > 65):
```

versus

```python
if employee.is_eligible_for_full_benefits():
```

- ✔ Komentáře lžou! Ne vždy a ne záměrně, ale až příliš často!
- ✔ Nepřesné komentáře jsou horší než žádné komentáře!
- ✔ Komentáře nenapraví špatný kód!
- ✔ Dobré komentáře:
  - ✘ (do)vysvětlení, (do)upřesnění
  - ✘ zdůraznění, varování před následky
  - ✘ TODOs
- ✔ Špatné komentáře:
  - ✘ staré (už neplatné), bezvýznamné, nevhodné, redundantní, nebo zavádějící komentáře
  - ✘ komentáře z povinnosti
  - ✘ zakomentovaný kód
  - ✘ nelokální nebo nadbytečné informace

Eratostenovo síto: komentáře

```
# This function generates prime numbers up to
# a user specified maximum. The algorithm
# used is the Sieve of Eratosthenes.
#
# Eratosthenes of Cyrene, b. c. 276 BC,
# Cyrene, Libya -- d. c. 194 BC, Alexandria.
# The first man to calculate the circumference
# of the Earth. Also known for working on
# calendars with leap years and ran
# the library at Alexandria.
#
# The algorithm is quite simple.
# Given an array of integers starting at 2,
# cross out all multiples of 2.
# Find the next uncrossed integer,
# and cross out all of its multiples.
# Repeat until you have passed
# the maximum value.
#
# @author hugo
# @version 1
```

```
# This function generates prime numbers up to
# a user specified maximum. The algorithm
# used is the Sieve of Eratosthenes.
# Given an array of integers starting at 2,
# cross out all multiples of 2.
# Find the next uncrossed integer,
# and cross out all of its multiples.
# Repeat until you have passed
# the maximum value.
#
# @author hugo
# @version 1
```

Za chvíli se zbavíme dalších komentářů!

Funkce a metody

- ✔ Funkce by měly být krátké! (A ještě kratší!)
- ✔ Funkce by měla dělat právě 1 věc a měla by ji dělat dobře. (A bez vedlejších efektů.)
- ✔ Funkce dlouhé méně než 5 řádků:
  - ✘ Většinou dělají právě 1 věc.
  - ✘ Mohou mít přesné a výstižné jméno.
  - ✘ Nemohou obsahovat vnořené příkazy `if`, `for`, ...
  - ✘ Bloky uvnitř příkazů `if`, `for`, ... jsou pouze 1 řádek dlouhé
- ✔ Krátké funkce umožňují testovat dílčí části algoritmu!
- ✔ Sekce uvnitř funkcí/metod:
  - ✘ Jasná indikace toho, že funkce/metoda nedělá jen 1 věc a měla by být rozdělena.
- ✔ Argumenty funkcí/metod:
  - ✘ Udržujte jejich počet malý! 0, 1, 2, výjimečně 3.
  - ✘ Vytvořte jméno tak, aby evokovalo pořadí argumentů.
  - ✘ Boolovské argumenty funkcí často značí, že funkce nedělá 1 věc! Rozdělte ji.

Eratostenovo síto: funkce

```
def generate_primes_up_to(max_value):
    """Find primes up to the max_value
    using the Sieve of Eratosthenes.

    """
    if max_value >= 2:  # There are some primes
        # Initialize the list (incl. 0)
        crossed_out = [
            False for i in range(max_value+1)]
        # Get rid of the known non-primes
        crossed_out[0] = crossed_out[1] = True
        # Run the sieve
        for number in range(2, len(crossed_out)):
            if not crossed_out[number]:
                # cross out its multiples
                for multiple in \
                range(2*number, len(crossed_out), number):
                    crossed_out[multiple] = True
        # Find the primes and put them in a list
        primes = [i for i in range(len(crossed_out))
                  if not crossed_out[i]]
        return primes
    else:   # max_value < 2
        return list() # no primes, return empty list
```

```
def generate_primes_up_to(max_value):
    """Find primes up to the max_value
    using the Sieve of Eratosthenes.

    """
    if max_value < 2:
        return []
    else:
        crossed_out = uncross_integers_up_to(max_value)
        run_the_sieve_on(crossed_out)
        return get_uncrossed_numbers_in(crossed_out)

def uncross_integers_up_to(max_value):
    crossed_out = [False for i in range(max_value+1)]
    # Cross out 0 and 1, they are not primes.
    crossed_out[0] = crossed_out[1] = True
    return crossed_out

def run_the_sieve_on(crossed_out):
    for number in range(2, len(crossed_out)):
        if not crossed_out[number]:
            cross_out_multiples_of(crossed_out, number)

def cross_out_multiples_of(crossed_out, number):
    for multiple in range(2*number, len(crossed_out), number):
        crossed_out[multiple] = True

def get_uncrossed_numbers_in(crossed_out):
    uncrossed = [i for i in range(len(crossed_out))
                 if not crossed_out[i]]
    return uncrossed
```


Eratostenovo síto: převod na třídu

```
def generate_primes_up_to(max_value):
    """Find primes up to the max_value
    using the Sieve of Eratosthenes.

    """
    if max_value < 2:
        return []
    else:
        crossed_out = uncross_integers_up_to(max_value)
        run_the_sieve_on(crossed_out)
        return get_uncrossed_numbers_in(crossed_out)

def uncross_integers_up_to(max_value):
    crossed_out = [False for i in range(max_value+1)]
    # Cross out 0 and 1, they are not primes.
    crossed_out[0] = crossed_out[1] = True
    return crossed_out

def run_the_sieve_on(crossed_out):
    for number in range(2, len(crossed_out)):
        if not crossed_out[number]:
            cross_out_multiples_of(crossed_out, number)

def cross_out_multiples_of(crossed_out, number):
    for multiple in range(2*number, len(crossed_out), number):
        crossed_out[multiple] = True

def get_uncrossed_numbers_in(crossed_out):
    uncrossed = [i for i in range(len(crossed_out))
                 if not crossed_out[i]]
    return uncrossed
```

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

    def __init__(self):
        self.crossed_out = []
        self.max = None

    def get_primes_up_to(self, max_value):
        """Return list of primes up to the max_value."""
        if max_value < 2: return []
        self.max = max_value+1
        self.uncross_integers_up_to_max_value()
        self.run_the_sieve()
        return self.get_uncrossed_numbers()

    def uncross_integers_up_to_max_value(self):
        self.crossed_out = [
            False for i in range(self.max)]
        # Cross out 0 and 1, they are not primes.
        self.crossed_out[0] = self.crossed_out[1] = True

    def run_the_sieve(self):
        for number in range(2, int(self.max**0.5)+1):
            if not self.crossed_out[number]:
                self.cross_out_multiples_of(number)

    def cross_out_multiples_of(self, number):
        for multiple in range(2*number, self.max, number):
            self.crossed_out[multiple] = True

    def get_uncrossed_numbers(self):
        uncrossed = [i for i in range(self.max)
                     if not self.crossed_out[i]]
        return uncrossed
```

Eratostenovo síto: jiná jména

Možná vám kód bude připadat čitelnější, použijeme-li jiná jména.
V kódu jsou také použity privátní proměnné a metody třídy.

```
# This class generates prime numbers up to a user
# specified maximum using the Sieve of Eratosthenes.
# Given an array of integers starting at 2,
# mark all multiples of 2 as not prime.
# Find the next integer that still can be prime,
# and mark all of its multiples as not prime.
# Repeat until you have passed the maximum value.

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

  def __init__(self):
    self.__can_be_prime = []
    self.__max = None

  def get_primes_up_to(self, max_value):
    """Return list of primes up to the max_value."""
    if max_value < 2:
      return []
    else:
      self.__max = max_value+1
      self.__init_candidates_for_primes()
      self.__run_the_sieve()
      return self.__collect_remaining_candidates()
```

```
def __init_candidates_for_primes(self):
  self.__can_be_prime = \
    [True for i in range(self.__max)]
  self.__can_be_prime[0] = False
  self.__can_be_prime[1] = False

def __run_the_sieve(self):
  for number in range(2, int(self.__max**0.5)+1):
    if self.__can_be_prime[number]:
      self.__mark_as_not_prime_multiples_of(number)

def __mark_as_not_prime_multiples_of(self, number):
  for multiple in range(2*number, self.__max, number):
    self.__can_be_prime[multiple] = False

def __collect_remaining_candidates(self):
  primes = [i for i in range(self.__max)
            if self.__can_be_prime[i]]
  return primes
```


Unit Testing

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:

- ✔ 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:

- ✔ 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, 61, 67, 71, 73, 79, 83, 89, 97]
>>>
```


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…

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.

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)
```


Ukázka: zlepšení efektivity generátoru prvočísel

Není třeba vyškrtávat násobky všech prvočísel menších nebo rovno N, stačí násobky prvočísel menších nebo rovných $\sqrt{N}$.

- ✔ Označme prvočíslo x, $x > \sqrt{N}$.
- ✔ Chci vyškrtnout $2x$: $2x$ ale je x násobkem prvočísla 2, a tudíž již muselo být vyškrtnuto.
- ✔ Chci vyškrtnout $3x$: $3x$ ale je x násobkem prvočísla 3, a tudíž již muselo být vyškrtnuto.
- ✔ Chci vyškrtnout $4x$: $4x$ ale je $2x$ násobkem prvočísla 2, a tudíž již muselo být vyškrtnuto.
- ✔ ...
- ✔ Všechny násobky prvočísel větších než $\sqrt{N}$ už byly vyškrtnuty při vyškrtávání násobků prvočísel menších nebo rovných $\sqrt{N}$.

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

Jak byste postupovali? Funkci na generování prvočísel už máme…

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])
```

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])
```

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])
```

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])
```

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(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 to otestovat?

TDD Ukázka: Je naše funkce napsaná čistě?

Stávající `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
```

Přepsaný `factorization.py`:

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

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

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

V tomto případě je asi první verze přehlednější než druhá.

TDD: Závěr

- ✔ Testy slouží jako specifikace.
- ✔ Testy slouží jako dokumentace.
- ✔ Testy vám pomáhají pochopit algoritmus.
- ✔ Testy pomáhají předejít zbytečným složitostem v kódu.
- ✔ Jakmile kód projde všemi testy a my nejsme schopni přijít na žádný test, při němž by kód selhal, je hotovo.
- ✔ Při změnách v kódu umožňují testy zkontrolovat, zda jsme do kódu nezanesli nějakou nepředvídanou chybu.