

# Test Driven Development

## [Programování řízené testy]

Ondřej Bouda  
xbouda2@fi.muni.cz

- ▶ Chceme programovat. . .
  - ▶ správně, tj. s nízkým počtem chyb
  - ▶ pěkně
  - ▶ efektivně
- ▶ Cílem TDD je "čistý kód, který funguje" [Ron Jeffries]
- ▶ Eliminace strachu z refaktorování
  - ▶ "Tenhle kód se mi nelíbí, ale radši na to nebudu sahat. . ."  
⇒ zahrňování kódu
- ▶ Mimo jiné ideální metoda pro řešení domácích úkolů ☺

Cílem je:

- ▶ představit způsob vývoje software, který vede k nízké chybovosti, dobrému návrhu a přitom je zábavný
- ▶ ukázat konkrétní metodu, kterou můžete snadno nasadit ještě dnes, třeba ještě na 3. domácí úkol

Cílem *není*:

- ▶ ukázat, proč je dobré testovat 😊
- ▶ ukázat, jaké jsou všechny možné způsoby testování
- ▶ ukázat, jak testovat
  - ▶ IV113 Úvod do validace a verifikace
  - ▶ IA159 Formal Verification Methods
  - ▶ ...

Inspirováno knihou: Kent Beck: Programování řízené testy

- ▶ **testováním**

# Jak dosáhnout cílů

- ▶ testováním
  - ▶ to nikoho nebaví

# Jak dosáhnout cílů

- ▶ testováním
  - ▶ to nikoho nebaví
- ▶ **automatizovaným testováním**

# Jak dosáhnout cílů

- ▶ testováním
  - ▶ to nikoho nebaví
- ▶ automatizovaným testováním
  - ▶ na to není čas

# Jak dosáhnout cílů

- ▶ testováním
  - ▶ to nikoho nebaví
- ▶ automatizovaným testováním
  - ▶ na to není čas
- ▶ **automatizovaným testováním ještě *před* samotnou implementací**



- ▶ testováním
  - ▶ to nikoho nebaví
- ▶ automatizovaným testováním
  - ▶ na to není čas
- ▶ **automatizovaným testováním ještě *před* samotnou implementací**
  - ▶ může znít zvláště, ale...

# Psaní testů před vlastní implementací

- ▶ přirozený postup: nejdřív ujasnit, co chceme, pak to udělat
    - ▶ test = částečná formalizace problému
    - ▶ během psaní testu může vyplynout, že by se hodilo jiné rozhraní
    - ▶ ověření pochopení problému
  - ▶ programování je těžké – spousta věcí naráz:
    - ▶ kódování (syntaxe, API používaných knihoven, ...)
    - ▶ návrh (jaké má být veřejné rozhraní, interní fungování, ...)
- ⇒ rozdělení práce mezi psaní testů a implementování
- ▶ test psaný těsně před implementací je způsob, jak zaměřovat pozornost
  - ▶ při testování až po implementaci jsme ovlivněni implementací
  - ▶ usnadňuje následnou implementaci – časově efektivnější než testování po implementaci

Pouze dvě pravidla:

1. nový kód píšeme pouze tehdy, když automatizovaný test selže
2. eliminujeme duplicitu

Důsledky:

- ▶ přirozeně vede k nezatěžování se zbytečnostmi
- ▶ testy píše samotný programátor
- ▶ okamžitá zpětná vazba, potvrzení správnosti směřování

- ▶ udržování seznamu, co je třeba udělat
- ▶ `while (!seznam.empty()) {`
  1. vybrat test, který můžeme s jistotou implementovat, a implementovat jej
  2. spustit všechny testy – nový test by měl selhat
  3. implementace testovaného kódu; možné přidání dalších položek na seznam
  4. spustit všechny testy – vše by mělo projít`}`
- ▶ rychlý rytmus: Red – Green – Refactor
  1. Red: napsat test, který neprojde
  2. Green: rychle test zprovoznit, dovoleno přitom naprosto cokoliv
  3. Refactor: eliminace duplicit

# Unit Testing [Jednotkové testování]

- ▶ testování dílčích jednotek, každá testována samostatně
- ▶ jednotkou typicky jedna třída
- ▶ při TDD používáno nejčastěji
  - ▶ vynucuje dobrý návrh – vysoce soudržné, volně propojené komponenty<sup>1</sup>
  - ▶ TDD nevyžaduje vyložení unit testing – lze též např. blackbox testing (testování celého programu)

---

<sup>1</sup>jedna z klíčových zásad objektového návrhu – více v PA103 Objektové metody návrhu informačních systémů

# Ukázka jednotkového testu

```
#include <cppunit/TestFixture.h>
#include <cppunit/extensions/HelperMacros.h>

class VectorTest : public CppUnit::TestFixture
{
    CPPUNIT_TEST_SUITE(VectorTest);
    CPPUNIT_TEST(testEmpty);
    CPPUNIT_TEST(testPushBack);
    CPPUNIT_TEST_SUITE_END();

public:
    void testEmpty();
    void testPushBack();
};
```

# Ukázka jednotkového testu

```
void VectorTest::testEmpty()
{
    Vector vecEmpty;
    CPPUNIT_ASSERT(vecEmpty.empty());

    Vector vecOneItem(1, 42);
    CPPUNIT_ASSERT(!vecOneItem.empty());

    Vector vecManyItems(13, 42);
    CPPUNIT_ASSERT(!vecOneItem.empty());
}
```

# Ukázka jednotkového testu

```
void VectorTest::testPushBack()
{
    Vector vec;
    vec.push_back(42);
    CPPUNIT_ASSERT_EQUAL(1UL, vec.size());
    CPPUNIT_ASSERT_EQUAL(42, vec[0]);
    for (int i = 0; i < 1000; i++) {
        vec.push_back(i);
    }
    CPPUNIT_ASSERT_EQUAL(1001UL, vec.size());
    CPPUNIT_ASSERT_EQUAL(42, vec[0]);
    CPPUNIT_ASSERT_EQUAL(0, vec[1]);
    CPPUNIT_ASSERT_EQUAL(999, vec[vec.size()-1]);
}
```



- ▶ analogie knihovny JUnit
- ▶ základním kamenem je TestCase = třída obsahující testy dané jednotky
  - ▶ typicky obsahuje fixture = konkrétní testovaný objekt
  - ▶ jednotlivé testy jsou implementovány testovacími metodami
  - ▶ nezávislé spouštění testů – pro každý test se volá postupně:
    1. metoda setUp() (společná inicializace testů)
    2. testovací metoda
    3. metoda tearDown() (společný úklid)
- ▶ jednotlivé Test Cases organizovány do Test Suites (sad testů)
- ▶ technické info: <http://cecko.eu/public/pb161/tdd>

...

- ▶ testy by měly být maximálně konkrétní: testovat konkrétní vstup a očekávat konkrétní výstup
- ▶ začít hned asercemi, zbytek kódu doplňovat zpětně

*Kde byste měli začít s tvorbou systému? U historek, jež byste rádi vyprávěli o hotovém systému.*

*Kde byste měli začít psát funkce? U testů, které chcete zprovoznit u hotového kódu.*

*Kde byste měli začít psát test? U asercí, které uspějí, když je vše hotovo.*

— Kent Beck: TDD

# Jaké testy psát

- úvodní test:** může využívat několik různých operací nad objektem, ale pouze nezbytné rozumné minimum
- jednokrokový test:** měl by reprezentovat jeden krok k cíli (typicky jednu operaci; příliš složité  $\Rightarrow$  přidat jednodušší test)
- vysvětlovací test:** účelem je zejména ověření, že chápu fungování jednotky správně, nebo ukázat příklad užití
- test externí jednotky:** účelem ověřit si, že rozumím API externí jednotky; píše se před prvním použitím nového prostředku externí jednotky
- návratový test:** jako první krok k opravě hlášené chyby (proč chyběl test, který by chybu odhalil?)

# Oprava hlášené chyby v režimu TDD

1. spustit všechny testy – měly by všechny projít (chyba dříve nebyla známá)
2. přidat nový, minimální test, který chybu ukáže tím, že selže
3. opravit implementaci
4. spustit všechny testy – měly by všechny projít
  - ▶ skutečně ověříme, že jsme chybu reprodukovali a následně opravili
  - ▶ zkontrolujeme, že jsme nerozbili nic jiného
  - ▶ efektivní metoda proti vracejícím se chybám

- ▶ testy jsou milníky implementace dané jednotky
- ▶ snaha o pokrytí testy všech aspektů programované jednotky
- ▶ ve chvíli, kdy všechny testy projdou, máme hotovo
- ▶ implementace každého testu vždy až před implementací samotné funkce většinou výhodnější

- ▶ TDD není pro každého
- ▶ jednotkové testování se nehodí na všechno
- ▶ automatizovaně nelze testovat vše (bezpečnost, souběžnost, ...)

- ▶ Implementujte funkce až po napsání testů
- ▶ TDD je nástroj, míra jeho použití záleží na vás
- ▶ Zdroje: <http://cecko.eu/public/pb161/tdd>

Red – Green – Refactor

- ▶ 

```
while (!seznam.empty()) {
```

  1. vybrat test, který můžeme s jistotou implementovat, a implementovat jej
  2. spustit všechny testy – nový test by měl selhat
  3. implementace testovaného kódu; možné přidání dalších položek na seznam
  4. spustit všechny testy – vše by mělo projít

```
}
```

"Čistý kód, který funguje"



- ▶ Implementujte funkce až po napsání testů
- ▶ TDD je nástroj, míra jeho použití záleží na vás
- ▶ Zdroje: <http://cecko.eu/public/pb161/tdd>

Red – Green – Refactor

- ▶ 

```
while (!seznam.empty()) {
```

  1. vybrat test, který můžeme s jistotou implementovat, a implementovat jej
  2. spustit všechny testy – nový test by měl selhat
  3. implementace testovaného kódu; možné přidání dalších položek na seznam
  4. spustit všechny testy – vše by mělo projít

```
}
```

"Čistý kód, který funguje"

## Otázky? Diskuse?

*Aplikujte TDD alespoň na jednu domácí úlohu!*