Testování software

IVS 02

Martin Dočekal, David Kozák

Obsah

- **Prerekvizity**
  - OOP
  - Chyby v SW
- **Testování**
  - Základní pojmy
  - Stupně testů
  - Strategie testování
  - TDD
  - BDD
- **Zadání projektu**

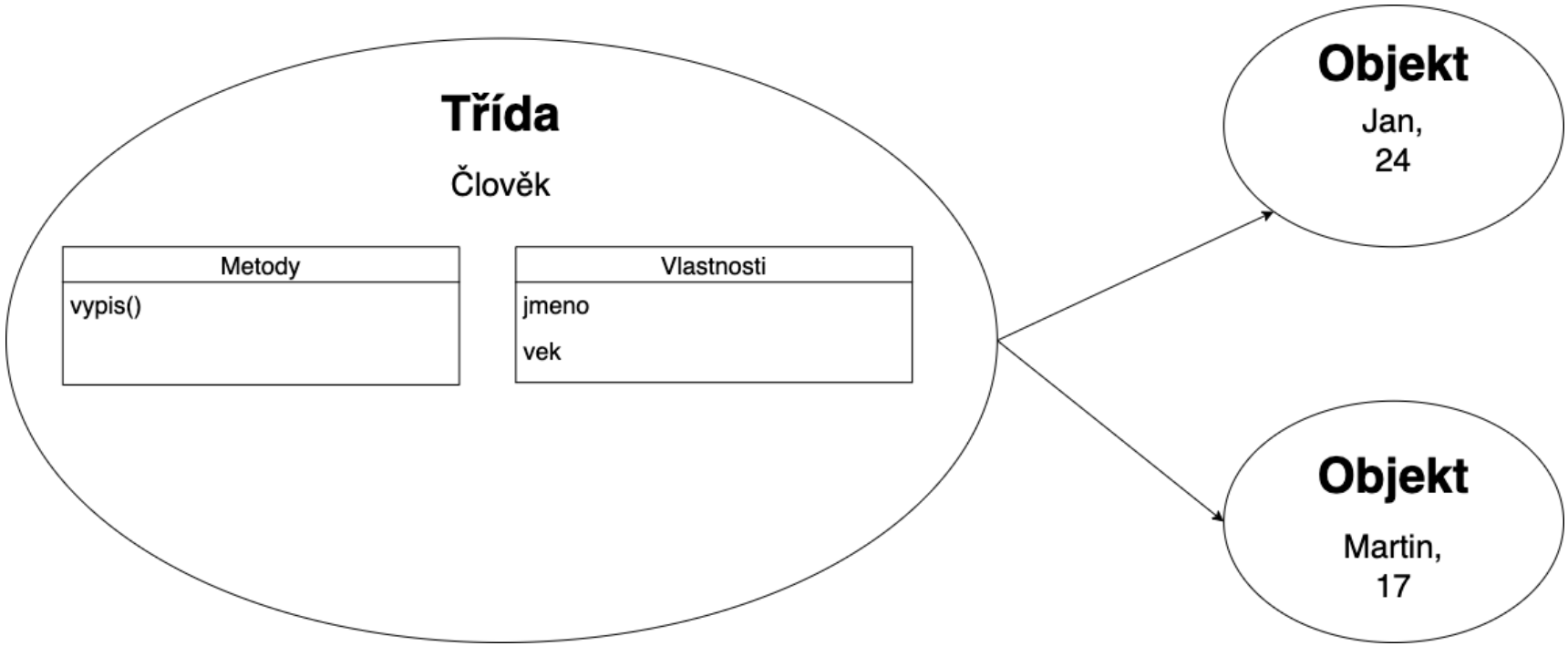
PREREKVIZITY – OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

Prerekvizity (OOP)

- **Objektově orientované programování (OOP)**
  - **Třída** – šablona podle níž vytváříme objekty (instance)
  - **Objekt** – jeden konkrétní jedinec (reprezentant, entita) příslušné třídy
    - **Proměnné**
    - **Metody**
      - **Konstruktor** – volá se při vytváření objektu
      - **Destruktor** – volá se při rušení objektu

Prerekvizity (OOP)

- Pro konkrétní objekt nabývají vlastnosti deklarované třídou konkrétních hodnot



Převod do světa objektů

Strukturovaný

```
typedef struct {
   char  *name;
   int age;
} Person;

Person initPerson(char *name, int age) {
   Person person;
   person.name = name;
   person.age = age;
   return person;
}

void destroyPerson(Person *p) {
   free(p->name);
   p->name = NULL;
}


void printPerson(Person *p) {
   printf("Hello, I am %s \n", p->name);
}

int main() {
   Person person = initPerson("Martin", 21);
   printPerson(&person);
   destroyPerson(&person);
}
```

Objektový

Person.cpp

```
Person::Person(char *name, int age) {
   this->name = name;
   this->age = age;
}


Person::~Person() {
   free(this->name);
   this->name = nullptr;
}

void Person::print() {
   std::cout << "Hello, I am " << this->name << std::endl;
}


int main() {
   Person person("Martin", 21);
   person.print();
   return 0;
}
```

Person.h

```
class Person {
   int age;
   char *name;

public:
   Person(char *name, int age);
   ~Person();
   void print();
};
```

Poznámky

- V hlavičkových souborem deklarujeme třídy a typy objektů
- **Konstruktor:**
  - Person(char *name, int age)
  - musíme je volat v implementačním souboru
- **Destruktor:**
  - ~Person();
  - volá se sám
- **Syntax:**
  - jmeno_tridy :: jmeno_metody(typ parametr) { … }

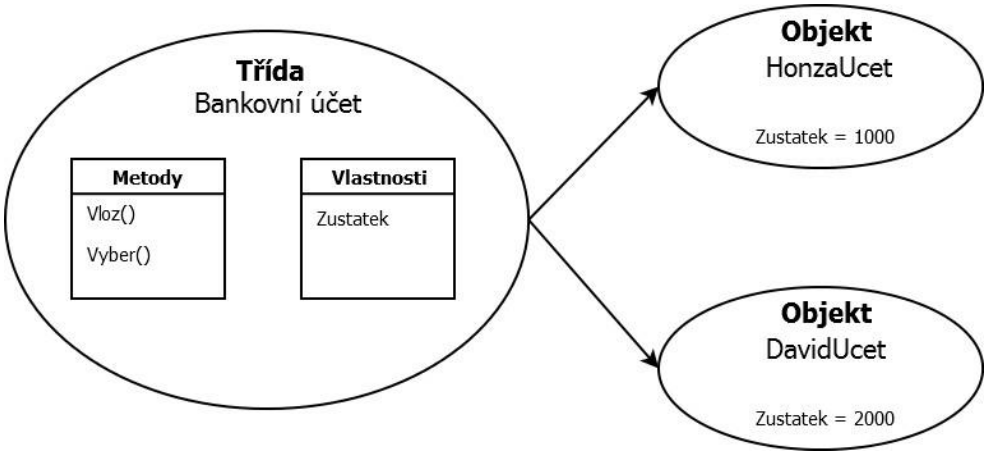
Prerekvizity - pokračování

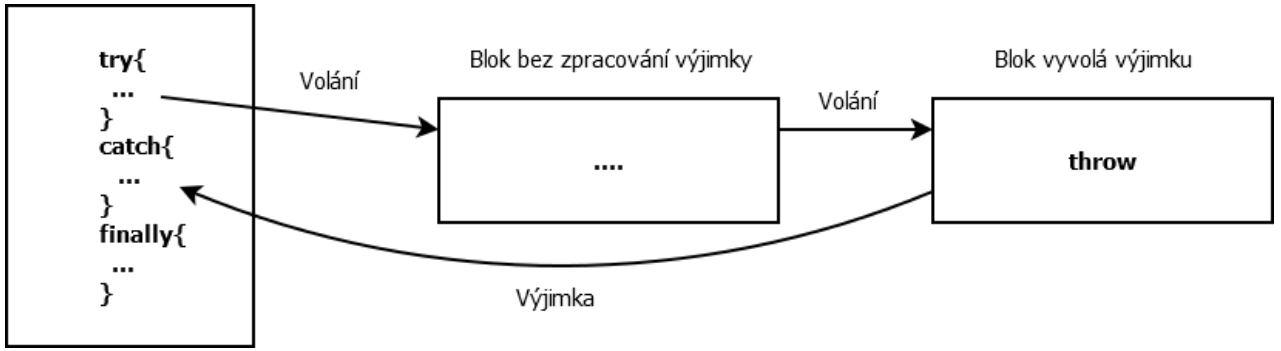
- **Výjimka** – výjimečná situace, která může nastat za běhu programu (chyba). Objekt, který nese informaci o chybě.
  - Obdoba chybového kódu (např.: v C errno)
  - **Vyhození výjimky (throw)**
    - Pokud nastane chybová situace, je vyhozena výjimka (neexistující soubor, nedostatek paměti, …)
  - **Zachycení výjimky (catch)**
    - Zpracování vyvolané výjimky; může existovat speciální objekt zpracovávající výjimky

Výjimky - příklad

```
void stepA() {}

void stepB() {}

void stepC() { throw

std::runtime_error("Oops"); } void

doSomething() {
   stepA();
   stepB();
   stepC();
}

int
   main()
   { try {
      doSomething();
   } catch (std::runtime_error &ex) {
      std::cout << "Error: " << ex.what() <<
      std::endl;
   }
}
```

OOP třídy a výjimky





Prerekvizity - příklad

- Třída:

```
class BankovniUcet
{
public:
    double zustatek;

    BankovniUcet();
    ~BankovniUcet();

    void vloz(double castka);
    void vyber(double castka);
};
```

```
BankovniUcet::BankovniUcet(){
    zustatek = 0;
}

BankovniUcet::~BankovniUcet(){
    zustatek = 0;
}

void BankovniUcet::vloz(double castka){
    if(castka <= 0){
        throw MyException1("castka musi byt vetsi nez 0!");
    }
    zustatek += castka;
}

void BankovniUcet::vyber(double castka){
    if(castka <= 0){
        throw MyException1("castka musi byt vetsi nez 0!");
    }
    else if((zustatek - castka) >= 0){
        zustatek -= castka;
    }
    else{
        throw MyException2("Nedostatek penez!");
    }
}
```

Prerekvizity - příklad

- Výjimky:

```
struct MyException1: public std::runtime_error
{
    MyException1(std::string const& message)
        : std::runtime_error(message)
    {}
};


struct MyException2: public std::runtime_error
{
    MyException2(std::string const& message)
        : std::runtime_error(message)
    {}
};
```

```
BankovniUcet ucet = BankovniUcet();

try{
  ucet.vyber(-100);
}
catch (MyException1& e){
  std::cout<<"vyjimka MyException1 zachycena: "<<e.what() <<std::endl;
}
catch (MyException2& e){
  std::cout<<"vyjimka MyException2 zachycena: "<<e.what() <<std::endl;
}
catch (std::exception& e){
  std::cout<<"vyjimka zachycena: "<<e.what() <<std::endl;
}
```

PREREKVIZITY – CHYBY V SOFTWARE

Co je to softwarová chyba?

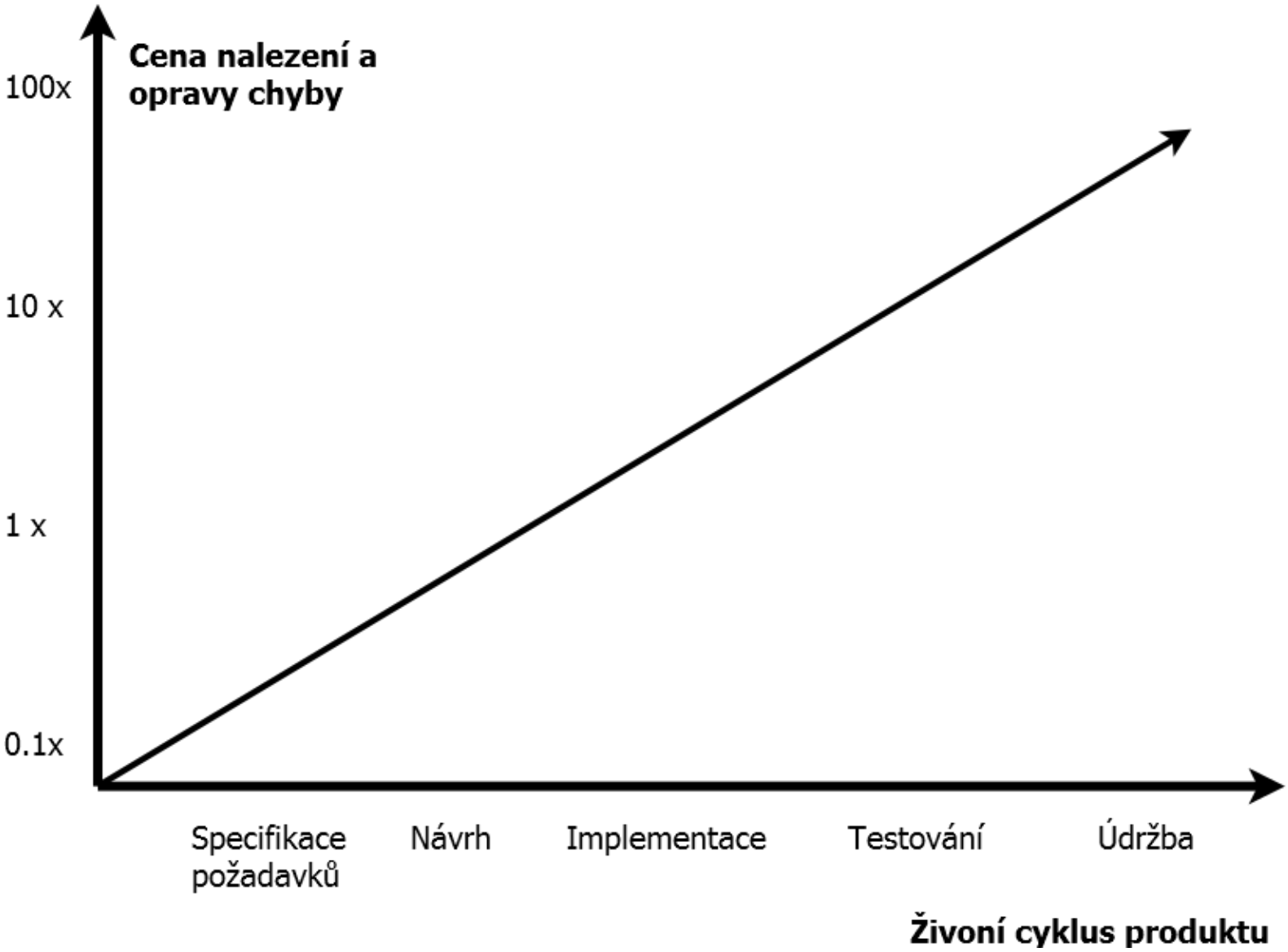
Platí alespoň jedno z následujících tvrzení:

1. SW nedělá něco, co by podle specifikace produktu dělat měl
2. SW dělá něco, co by podle specifikace produktu dělat neměl
3. SW dělá něco, o čem se produktová specifikace nezmiňuje
4. SW nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat
5. SW je obtížně srozumitelný, těžko se s ním pracuje, je pomalý, nebo – podle názoru testera SW – jej koncový uživatel nebude považovat za správný

- **Důsledky chyby:**
  - Od nepohodlí až po ztráty na životech

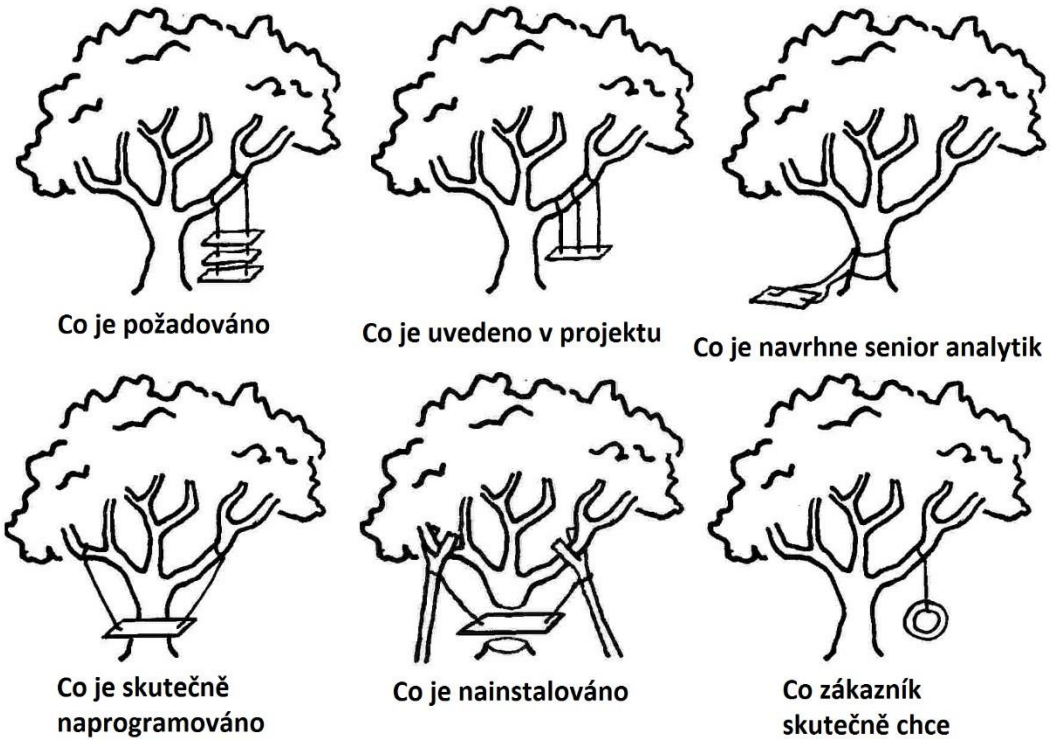
TESTOVÁNÍ – ZÁKLADNÍ POJMY

Náklady na chyby



Pojmy z testování SW

- **Verifikace** – „Stavíme tu věc správně, jak chce zákazník?“
  - kontrola, zda produkt odpovídá specifikaci
- **Validace** – „Postavili jsme správnou věc pro uživatele?“
  - kontrola, zda produkt odpovídá požadavkům uživatele



Axiomy testování

- **Fakta o testování (životní moudra)**
  - **Žádný program není možné otestovat kompletně**
    - Velký počet vstupů/výstupů
    - Velký počet cest v SW
  - **Velké riziko**
    - Co není pokryto testy? Co když je tam závažná chyba?
  - **Testování nikdy neprokazuje, že chyby neexistují**
    - Vždy existuje určitá kombinace vstupů, která není testována
  - **Čím více chyb najdete, tím více chyb je v SW**
    - Opakující se chyby, různě se projevující jedna chyba
  - **Ne všechny nalezené chyby se opraví**
    - Není čas, není to chyba, oprava je riskantní, nestojí za to

Dilema – Automatizovat?

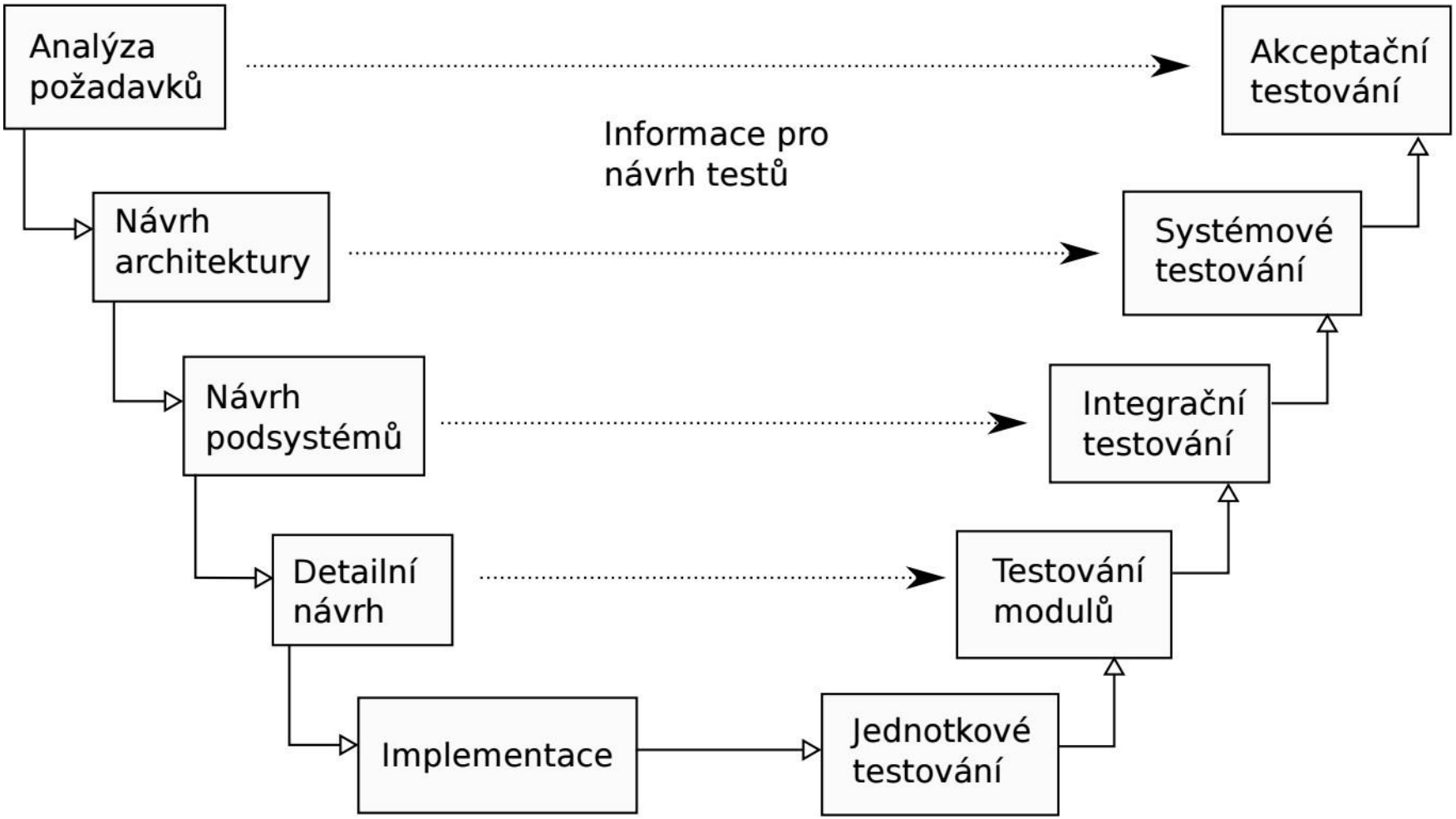
- **Automatické**
  - test je realizován programem – skriptem
  - jsou znovuspustitelné, lze navázat do procesu – **CI** (Continuous integration)
- **Manuální testování**
  - scénáře, vstupní a výstupní data
  - lidé realizují daný scénář – armáda testerů
- Vyhodnotit smysluplnost a proveditelnost automatizace!

TESTOVÁNÍ – STUPNĚ TESTŮ

Stupně testů

- **Jednotkové (Unit)**
  - Jednotlivá komponenta
- **Integrační**
  - Několik komponent jako skupina
  - Zaměřené na rozhraní podsystému
- **Systémové**
  - Celý systém s ohledem např. na spolehlivost či výkonnost
- **Akceptační**
  - Splňuje systém požadavky zákazníka?

Požadavky na testy



Jednotkové testy

- **Jednotka**
  - Záleží na paradigmatu daného jazyka – (funkce, objekt, …)
  - Komponenta třetí strany
  - Testy jednotky probíhají **v izolaci** od ostatních
  - Cílem je zjistit, zda **jednotka funguje podle specifikace**
- Koncept **xUnit**

xUnit

- Kolektivní název pro unit test frameworky pro různé jazyky (platformy) postavené na designu SUnit pro Smalltalk
- Poskytují konstrukce pro vytváření opakovaně spustitelných testů a ověřování očekávaného chování
- Příklady:
  - cppUnit, Google Test – C++
  - jUnit – Java, Scala
  - nUnit – .NET
  - pyUnit – Python
  - http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Koncepce xUnit

- **Test Case**
  - **Shlukuje sadu testů** (metod) testujících jednotku
  - **Metoda *setup*** – zajištění předpokladů (preconditions)
  - Spuštění samotného testu, který obsahuje jedno či více tvrzení (assertions), které mají platit.
  - **Metoda *teardown*** – uvedení do původního stavu tak, aby nebyly ovlivněny další testy
- **Test Suite**
  - **Sada testovacích případů** (test case)
  - Např. test všech jednotek v konkrétním systému

Setup

- **Zajištění předpokladů**
- **Fixture**
  - Fixture jsou všechna vstupní data pro xUnit taková, aby bylo možné provést test opakovaně a bez ohledu na aktuální kontext
    - Množina dat a objektů
    - Tvoří konfiguraci testů
    - Může obsahovat:
      - Mock objekty
      - Vstupní a očekávané hodnoty
      - Inicializace databáze vhodnými hodnotami
      - Vstupní soubory apod.

EXPECT vs. ASSERT

- **EXPECT**
  - Pokud test **selže**, **pokračuje dále**
    (nastane fail, ale selhání není tak kritické, abychom test ukončili)
  - **Příklad:** Testuji různé vstupy, pokud selže pro určitý vstup, můžu pokračovat testováním dalšího vstupu

- **ASSERT**
  - Pokud test **selže**, **ukončí ho**
    (nastane fail tak závažný, že by další průběh testu neměl smysl, tak ho ukončíme)
  - **Příklad:** Selhalo otevření souboru -> ukončím, nemůžu s ním pracovat

Příklad testů (Google test)

- Fixture

```
class osobniUcet : public ::testing::Test
{
protected:
    BankovniUcet *ucet;

    virtual void SetUp() {
        ucet = new BankovniUcet();
    }
    virtual void TearDown() {
      delete ucet;
    }

};
```

- Test case

```
TEST_F(TestFixtureName, TestName)
{
  ...
}
```

- Pokračování na dalším slidu …

Příklad testů (Google test) pokračování

```
TEST_F(osobniUcet, vlozeni){
    EXPECT_NO_THROW(ucet->vloz(100));
    EXPECT_EQ(ucet->zustatek, 100);
}

TEST_F(osobniUcet, vlozeni_err){
    EXPECT_ANY_THROW(ucet->vloz(-100));
}


TEST_F(osobniUcet, vyber_err){
    EXPECT_ANY_THROW(ucet->vyber(1000));
    EXPECT_EQ(ucet->zustatek, 0);
    EXPECT_ANY_THROW(ucet->vyber(-100));
    EXPECT_EQ(ucet->zustatek, 0);
}

TEST_F(osobniUcet, operace){
    EXPECT_NO_THROW(ucet->vloz(200));
    EXPECT_NO_THROW(ucet->vyber(100));
    EXPECT_EQ(ucet->zustatek, 100);
}
```

```
BankovniUcet::BankovniUcet(){
    zustatek = 0;
}

BankovniUcet::~BankovniUcet(){
    zustatek = 0;
}

void BankovniUcet::vloz(double castka){
    if(castka <= 0){
        throw MyException1("castka musi byt vetsi nez 0!")
    }
    zustatek += castka;
}

void BankovniUcet::vyber(double castka){
    if(castka <= 0){
        throw MyException1("castka musi byt vetsi nez 0!")
    }
    else if((zustatek - castka) >= 0){
        zustatek -= castka;
    }
    else{
        throw MyException2("Nedostatek penez!");
    }
}
```

Poznámky

- **osobniUcet** v hlavičce funkce je **fixture**
- **NO_THROW** očekáváme, že nebude vyhozena žádná výjimka
- **ANY_THROW** očekáváme vyhození jakékoliv výjimky

Izolace

- **Reálné funkce využívají jiné funkce**
- Analogicky to platí pro objekty v OOP
- Aby se zabránilo průniku chyb odjinud, nesmí se používat kód jiné jednotky
- Používají se **simulace okolí:**
  - ***Stub*** *(pahýl)* – simuluje **jiný kód** pro účely testů
    - Vrací předdefinovanou hodnotu
    - Může zaznamenávat své volání
  - ***Mock*** *(napodobenina)*
    - **Ověřuje chování** toho, kdo ho používá
      - Spíše než kontrolování předdefinované hodnoty sleduji jak byly (pořadí) volané metody.
    - Jsou ověřovány **interakční scénáře**

Integrační testy

- Jednotky tvoří **podsystémy** (množinu modulů)
- **Integrační testy** testují podsystémy
- **Napodobeniny** (Mock) jsou **nahrazeny** skutečnými implementacemi
- Cílem je **detekovat chyby na rozhraní** jednotek a jejich interakce
- **Resp.** testujeme, že spolu moduly komunikují správný způsobem
    - **Pozn. např.** převody jednotek, špatné předpřipravení modulů

Systémové testy

- Testování **systému jako celku**
- Ujištění se, že **funguje dle požadavků**

- **Typy systémových testů**
  - **Funkční**
    - GUI - Selenium
  - **Výkonnostní**
  - **Zátěžové**
  - **Bezpečnostní** (např. penetrační)

Akceptační testy

- Ověřují splnění **požadavků zákazníka**
- Ověřují se **uživatelské scénáře**
- Prakticky systémové testy **při předávání** zákazníkovi
- **Black box**

Regresní testování

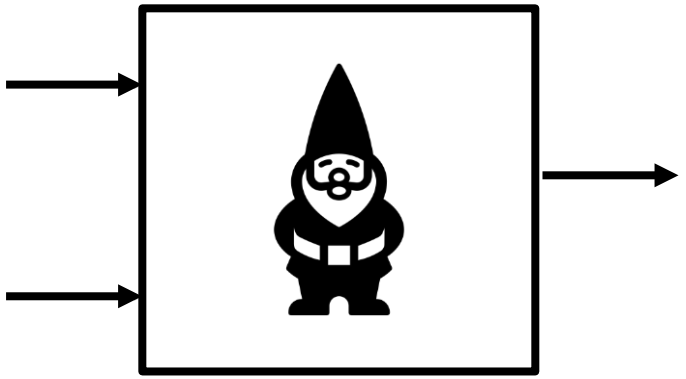
- **Testování prováděné po změnách** v kódu (Refaktorizace)
- Potvrzení, že jiná funkcionalita nebyla změnou ovlivněna
- Vhodné jsou **automatizované testy** – **CI**?
- Dobré provádět **po každém zásahu do kódu**
- **Postup**
  1. **Spustit testy** – ověřit, že systém funguje správně ještě než cokoliv uděláme
     - Není nezbytně nutné, ale může ušetřit spoustu bolesti
  2. **Provést změny v kódu**
  3. **Spustit znovu testy** – Ověřit, že nedošlo k nechtěným změnám funkcionality

TESTOVÁNÍ – STRATEGIE

Strategie testování

- **Černá skříňka** (black box)
  - Bez znalosti vnitřní struktury
  - Důležité jsou pouze vstupy / výstupy
  - Ohled pouze na specifikaci / funkci

- **Průhledná skříňka** (white / glass box)
  - S ohledem na vnitřní strukturu
  - Prakticky testování napsaného kódu
  - Snaha pokrýt všechny větve
  - Může být časově náročné; příliš detailní
  - Také musí odpovídat specifikaci



Testování moderními pojmy

- **Statická analýza** zkoumá vlastnosti SW bez jeho provádění
- **Dynamická analýza** ověřuje vlastnosti SW na základě provádění kódu

- **Formální verifikace** s pomocí formálních metod ověřuje, že systém odpovídá specifikaci
- **Testovaní** zkoumá SW jeho spouštěním za účelem zvýšení jeho kvality

Pokrytí kódu – Code Coverage

- **Sledování, jak je aplikace pokryta testy**
- *Kolik případů užití máme pokrytých?*
  Kolik zákaznízkových požadavků?
  Nejpřesnější je pokrytí kódu.
- Při zjišťování pokrytí kódu se zaznamenává, **jaký kód byl při testování spuštěn a jaký naopak otestován ještě nebyl**

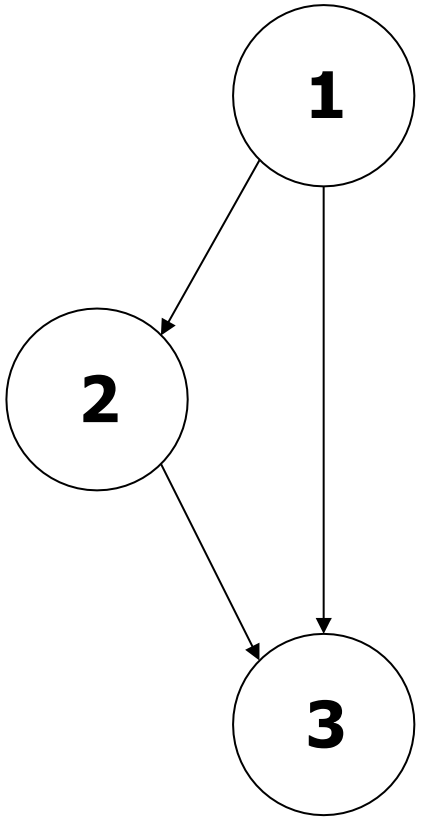
- Nejjednodušší, ale zároveň nedostačující a zavádějící, je **pokrytí příkazů – řádků**
- O stupeň pokročilejší je **pokrytí hran – rozhodnutí**
- Nejvyšší stupeň pokrytí je **pokrytí cest**

Pokrytí kódu – Control flow graph (CFG)

Program

```
1: if(x < y){
2:     y = 0;
   }
3: x -= y;
```

Control flow graph



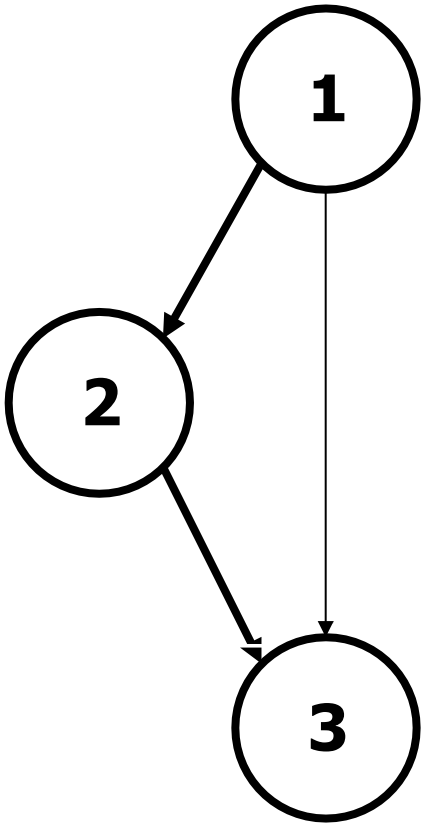
Pokrytí kódu – Control flow graph (CFG)

x=10, y=15

Program

```
1: if(x < y){
2 :   y = 0;
   }
3: x -= y;
```

Control flow graph



Poznámky

- Pokud bychom testovali **line-coverage**, tak bychom při zadání vstupu pro NEPLATNOU podmínku sice pokryly všechny řádky, ale nepokryli jsme cestu **z 1 -> 3**, tedy když by byla podmínka splněna
- Úplné **path-coverage testování** je v podstatě NEmožné, pokud program obsahuje cykly, jelikož každý průběh smyčky je jednou cestou

Časté výmluvy vývojářů – není to návod!

- „U mě na notebooku to funguje.“
- „Nemáme dostatek času to testovat.“
- „Je to v pohodě, jsme přece startup.“
- „Je to beta verze, chyby najdou uživatelé.“
- „Nemáme na testování dost peněz.“
- „Nic zásadního jsme nezměnili.“
- „Nemyslel jsem si, že by nás hackeři chtěli napadnout.“
- „Ale oni to používají špatně.“

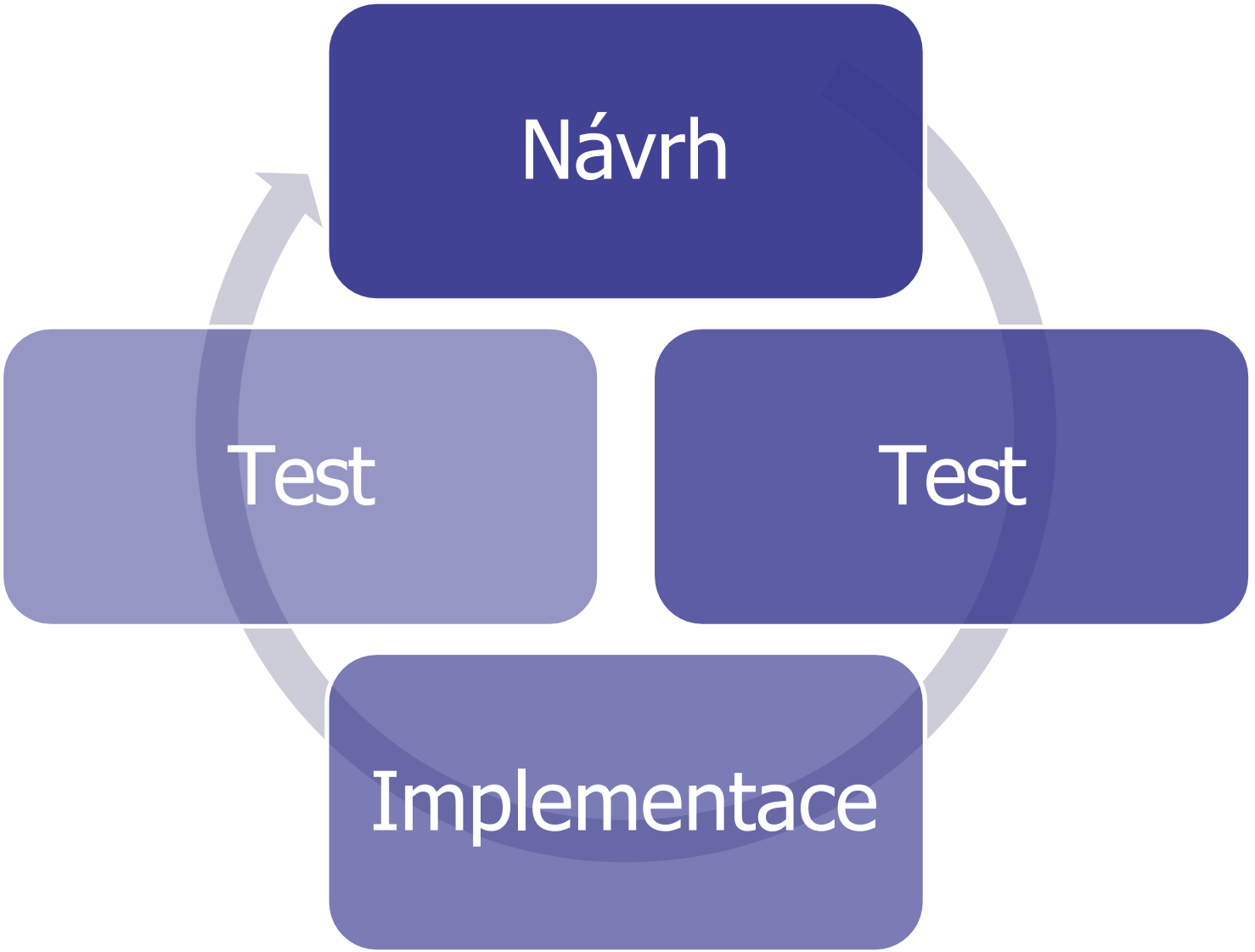
- https://www.americaninno.com/boston/8-common-_excuses-in-software-testing/

TESTOVÁNÍ –
TEST DRIVEN DEVELOPMENT

Test-driven development (TDD)

- **Testy řízený vývoj**, někdy také „programování řízené testy“
- Proces vývoje SW, kde je hlavní **důraz kladen na testy**
- **píšou se první**
- Řadí se k **agilním procesům**
- **Těží z automatizace spuštění testů** (typicky unit testy použitelné nejen pro ověření nové funkcionality, ale i jako regresní testy)

Konceptuální proces vývoje



Iterace v TDD

1. **Napsat sadu testů**
   - Prakticky návrh rozhraní (API)
2. **Spustit testy** a ujistit se, že **žádný test neprojde**
3. **Napsat kód** tak, **aby testy prošly**
   - Minimální kód pro splnění testů
4. **Ujistit se, že projde celá sada** testů
   - Aby způsob, jakým byl kód napsán, nenarušil kód ověřovaný jinými testy
5. **Refaktorizace** (pročistění) kódu **s průběžným ověřováním** pomocí testů

Výhody TDD

- **Všechen** produkční **kód je testovaný**
- Je implementována **pouze potřebná funkcionalita** *(tedy není implementováno nic navíc)*
- **Předem napsané testy nejsou ovlivněny** jakoukoliv **znalostí implementace**
- **Průběžným spouštěním** testů je možno **zamezit dlouhým úpravám kódu** bez znalosti stavu věcí
- **Občas** může být **výhodnější vrátit se do stavu, kdy testy prošly, než ladit**

TESTOVÁNÍ – BEHAVIOUR DRIVEN DEVELOPMENT

Behaviour Driven Development (BDD)

- **Rozšíření TDD**
- Vývoj je řízen **popisem chování systému**
  (příběh je blízký lidské přirozené řeči – tedy popis testu není kódem, ale spustitelným textem)
- **Slovní popis testů**, tudíž snadněji **srozumitelný**
- Testy **skutečně popisují očekávané chování**, tedy specifikaci kódu
- Testy **odpovídají „příběhům uživatelů“**, které jsou často výchozím bodem agilních metod vývoje
- **Cbehave**, Igloo
- RSpec (Ruby), NSpec (.Net), JBehave (Java)

Behaviour Driven Development (BDD)

- Používá behaviorální specifikaci zahrnující **poloformálně zapsané scénáře:**

1. Z pohledu koho,
2. co je cílem,
3. při počátečních podmínkách pro provedení scénáře,
4. ve kterém stavu systému se scénář provádí,
5. jaký je očekáváný výsledek nebo stav systému.

BDD - příklad

```
#include "cbehave.h"

FEATURE(1, "ucet")
  SCENARIO("Vlozeni 100,- na ucet")

     GIVEN("Vytvoreni ucetu pro noveho klienta.")
        BankovniUcet ucet = new BankovniUcet();
     GIVEN_END

     WHEN("Pouzijeme metodu vloz pro vlozeni 100,- na ucet")
         ucet.vloz(100);
     WHEN_END

     THEN("na uctu ma byt aktualni zustatek 100,-")
         SHOULD_INT_EQUAL(ucet.zustatek, 100);
     THEN_END
 SCENARIO_END
```

```
Features are as belows:
Feature: ucet
 Scenario: Vytvoreni ucetu pro noveho klienta.
     Given: Vytvoreni ucetu pro noveho klienta.
     When: Pouzijeme metodu vloz pro vlozeni 100,- na ucet
     Then: na uctu ma byt aktualni zustatek 100,-

 Summary:
 total features: [1]
 failed features: [0]
 total scenarios: [1]
 failed scenarios: [0]
```

Něco navíc…

- **Selenium**
  - Automatizace testování GUI aplikací
  - Ve webovém prohlížeči
  - Java API
- **RDD – Readme Driven Development** – Testování README
  - Testují se příklady v README
  - Vhodné zejména u veřejných knihoven
  - „Pokud nefunguje příklad v README, knihovnu většinou nepoužiji. “

Něco navíc…

- **Fuzz testování (fuzzing)**
  - Často automatizované
  - Chybné, neočekávané, náhodné vstupy
  - Výjimky, reakce na chybné vstupy, memory leak apod.
  - Neočekávané kombinace validních vstupů
  - Typické pro testování bezpečnosti systémů
    - Např. Odolnost proti náhodnému pádu systému

Continuous Integration – Postupná integrace

- Urychlení vývoje a hlídání spolupráce v týmu
- Principy:
  - **Centralizované repozitáře**
    - Zdrojové soubory na jednom místě (např. GIT)
  - **Automatická kompilace**
    - Po nahraní změn do repozitáře se provede build
  - **Automatické testování**
    - Automaticky se spustí testy
  - **Kontrola kvality kódu**
    - Kontrola pojmenování metod a tříd, definice komentářů, délka kódu tříd a metod, definice a použití proměnných apod.
  - **Automatické zveřejňování nových verzí**
  - A mnoho dalšího…

Literatura

- Prezentace – Petr Škoda, David Grochol, Tomáš Švec
- PATTON, Ron. *Testování softwaru*. Praha: Computer Press, 2002, xiv, 313 s. : il. ISBN 80-7226-636-5.
- BURNSTEIN, I. *Practical software testing*. New York: Springer, 2003, 709 s. ISBN 0-387-95131-8.
- BATH, Graham a Judy MCKAY. *The software test engineer´s handbook*. Santa Barbara: Rocky Nook, 2008, xviii, 397 s. ISBN 978-1-933952-24-6.
- STEPHENS, Matt a Doug ROSENBERG. *Testování softwaru řízené návrhem*. Brno: Computer Press, 2011, 336 s. : il., portréty. ISBN 978-80-251-3607-2.

- https://github.com/google/googletest
- https://github.com/bigwhite/cbehave