

Testování software

Martin Dočekal

Brno University of Technology, Faculty of Information Technology
Božetěchova 1/2, 612 66 Brno - Královo Pole
idocekal@fit.vutbr.cz



17.02.2021

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

PREREKVIZITY – OBJEKTIVĚ ORIENTO VANÉ PROGRAMOVÁNÍ

- **Objektově orientované programování (OOP)**
 - **Třída** – šablona podle níž vytváříme objekty (instance)
 - Existují i jazyky, které třídy nepoužívají (prototypově orientované)
 - **Objekt** je jeden konkrétní jedinec (reprezentant, entita) příslušné třídy
 - Spojuje data a funkcionalitu
 - Pro konkrétní objekt **nabývají vlastnosti** deklarované třídou **konkrétních hodnot**
 - **Proměnné** – datové typy zapouzdřené v objektech
 - **Metody** – funkce pracující převážně nad proměnnými objektu
 - **Konstruktor** – volá se při vytváření objektu
 - **Destruktor** – volá se při rušení objektu

Strukturovaný

```
unsigned level = 1;  
unsigned experience = 0;  
unsigned health = 100;  
  
experience += 200;  
lvlUp(experience, level, health);
```

Objektový

Třída

```
class Hero  
{  
    public:  
        Hero() : level(1), experience(0), health(100){};  
  
        void newExperience(unsigned exp)  
        {  
            experience += exp;  
            lvlUp();  
        }  
  
    protected:  
        unsigned level, experience, health;  
  
        void lvlUp(){}  
};
```

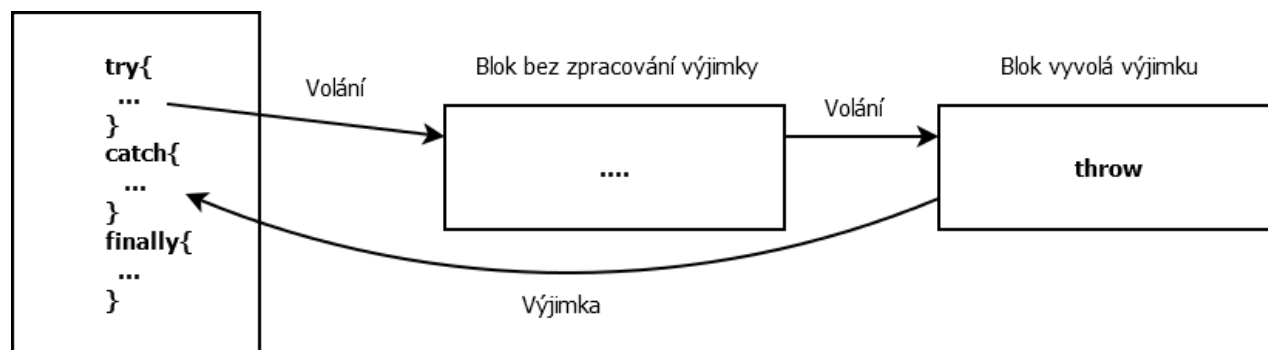
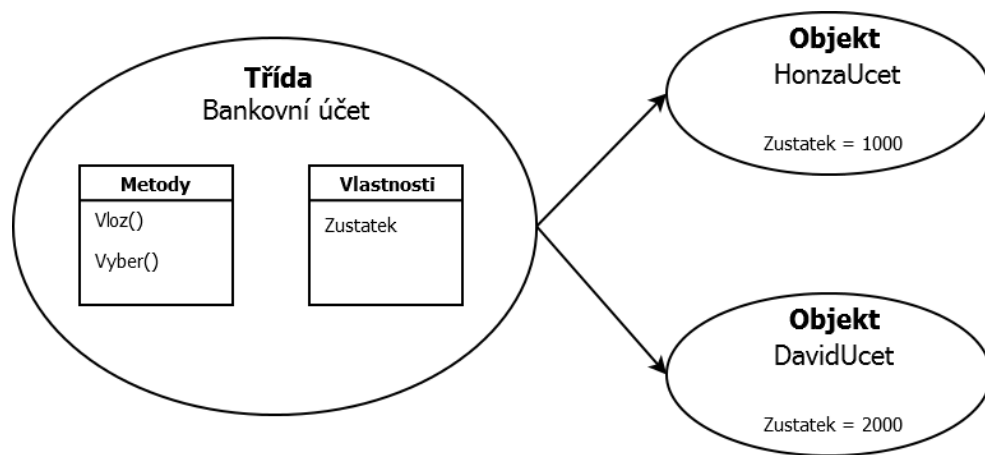
Tvorba objektu

```
Hero hero;  
hero.newExperience(200);
```

- **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
 - **Dokončení (finally)**
 - Uvolnění systémových zdrojů (i v případě výjimky)
 - Spustí se vždy ať už k výjimce došlo či nikoli

```
try {  
    ...  
}  
catch (...) {  
    // Však on si poradí...  
}
```





- 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!");
    }
}
```

- Výjimky:

```
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;
}
```

```
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)
    {}
};
```

PREREKVIZITY – CHYBY V SOFTWARE

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

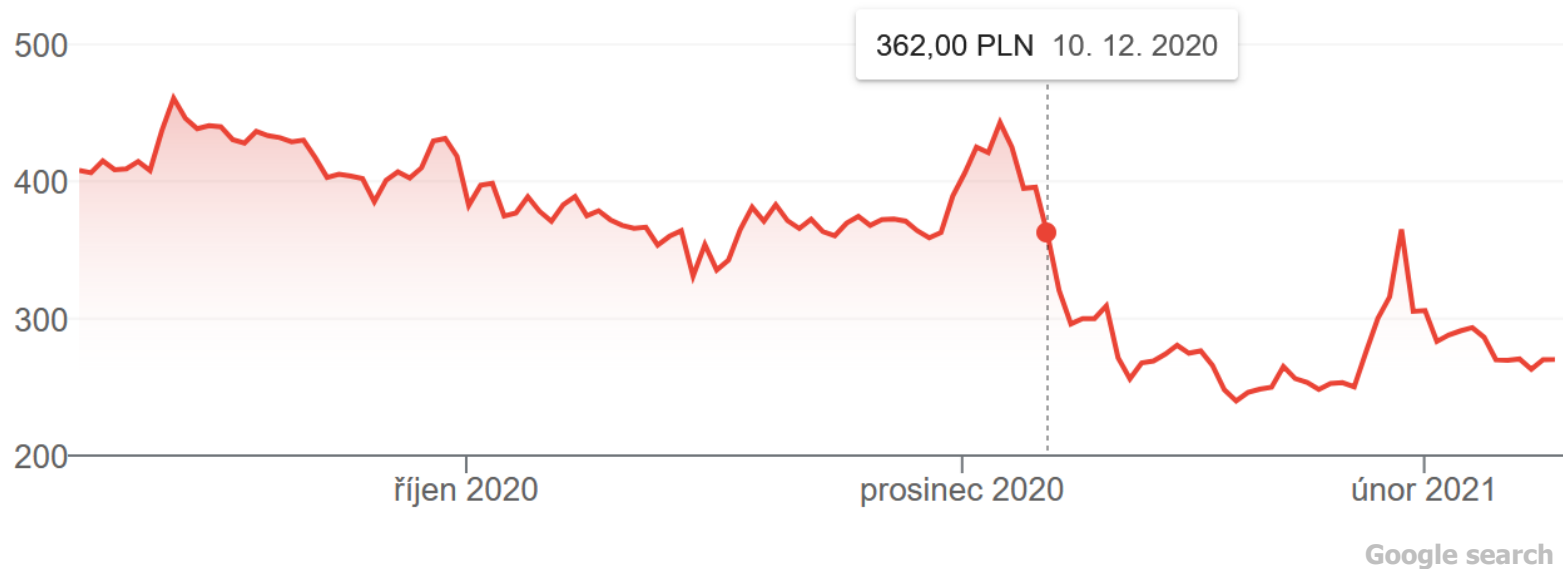
- **Důsledky chyby:**

- Od nepohodlí až po ztráty na životech

- **Dělení s pohyblivou řádovou čárkou Intel (1994)**
 - V některých případech vznikají neočekávané výsledky
 - Softwarová chyba, kterou se podařilo vypálit do počítačového čipu a v procesu výroby ji mnohokrát zopakovat
 - Náklady: přes 400 mil. dolarů
- **Přistávací modul na Mars (1999)**
 - Chybné nastavení jednoho datového bitu (signalizace dosednutí). Problém nezjištěn při testech
 - Nebyl testován systém jako celek, ale jen jeho části
- **Rok 2000**
 - Zkracování data 1989 -> 89, ?? 2000 -> 00 ??
 - Náklady: stovky mld. dolarů

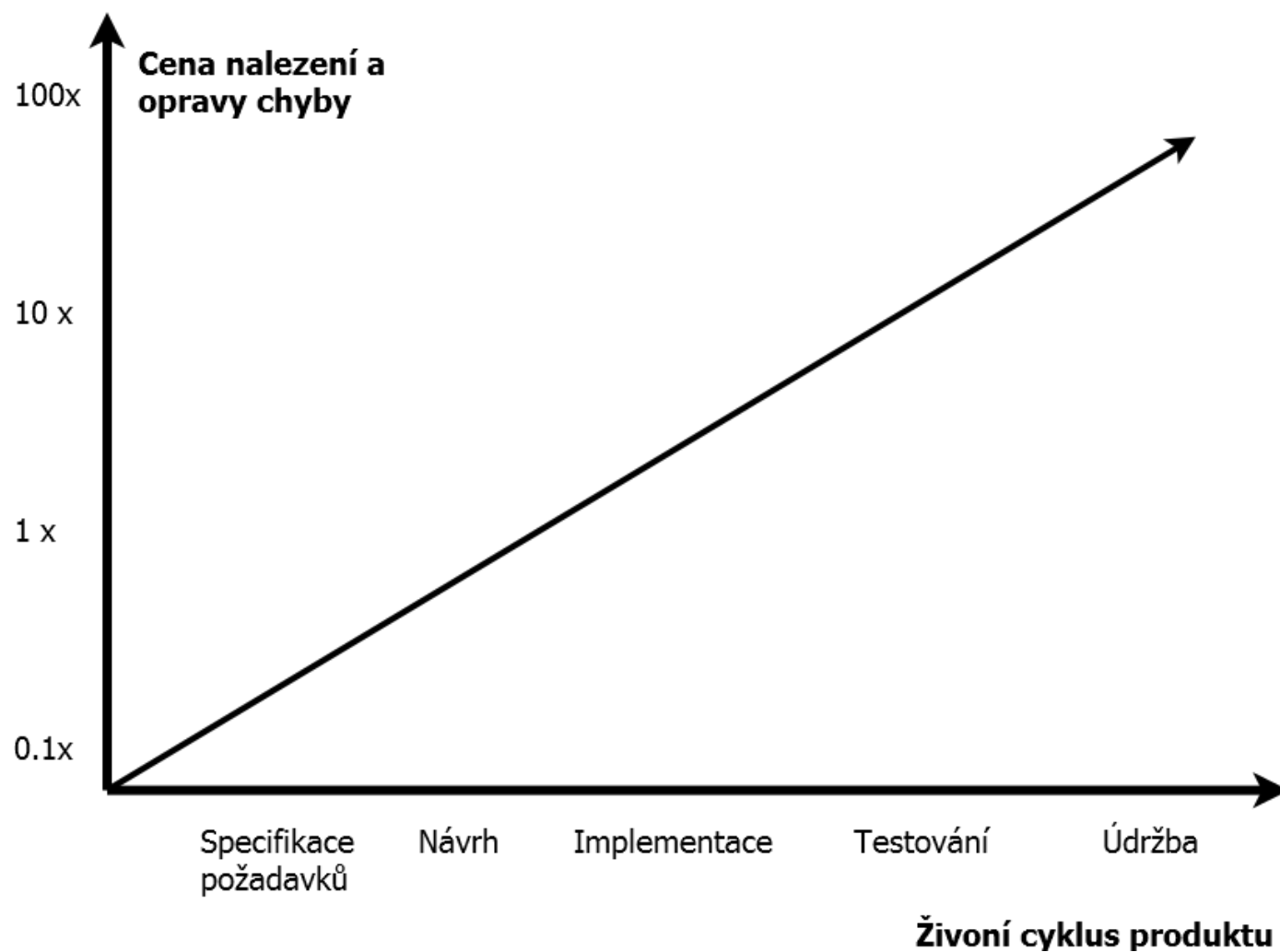


- Vydání Cyberpunk 2077
 - Hráči si stěžují na nedoladění
 - Propad akcií



TESTOVÁNÍ – ZÁKLADNÍ POJMY





- 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



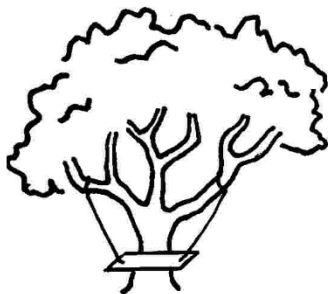
Co je požadováno



Co je uvedeno v projektu



Co je navrhne senior analytik



Co je skutečně
naprogramováno



Co je nainstalováno



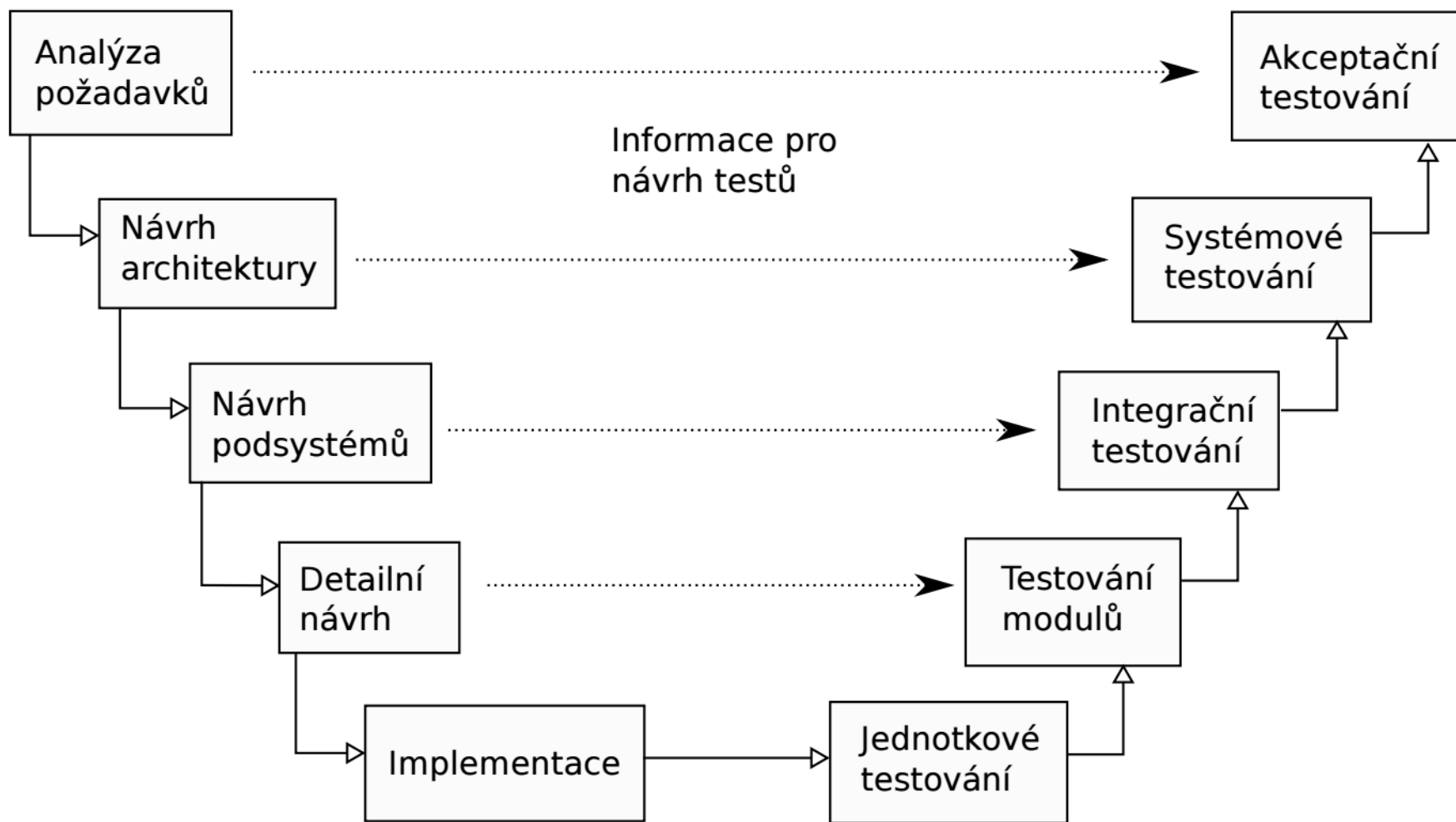
Co zákazník
skutečně chce

- 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 neproказuje, 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

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

TESTOVÁNÍ – 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?



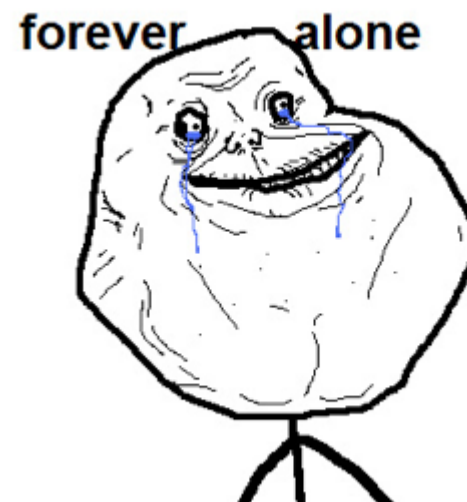
- **Jednotka**
 - Záleží na paradigmatu daného jazyka - (funkce, objekt, ...)
 - Komponenta třetí strany
 - Testy jednotky probíhají v izolaci od ostatních
 - Cílem je zjistit, zda jednotka funguje podle specifikace
- Koncept **xUnit**

- 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
 - Python – pyUnit
 - http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

- 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

- 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.

- 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 sledují jak byly (pořadí) volané metody.
 - Jsou ověřovány interakční scénáře



- EXPECT
 - Pokud test selže, pokračuje dále
 - Příklad: Testuji různé vstupy, pokud selže pro určitý vstup, můžu pokračovat testováním dalšího vstupu
- ASSERT
 - Pokud test selže, ukončí ho
 - Příklad: Selhalo otevření souboru -> ukončím, nemůžu s ním pracovat

- Fixture

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

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

};
```

- Pokračování na dalším slidu ...

- Test case

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

```
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!");
    }
}
```

- Jednotky tvoří podsystémy
- Integrační testy testují podsystémy
- Napodobeniny (Mock) jsou nahrazeny skutečnými implementacemi
- Cílem je detekovat chyby na rozhraní jednotek a jejich interakce

- Testování systému jako celku
- Ujistění se, že funguje dle požadavků
- Typy systémových testů
 - Funkční
 - Výkonnostní
 - Zátěžové
 - Bezpečnostní (např. penetrační)
 - Konfigurační
 - Různé OS, DB...
 - Zaměřené na zotavení (off-site backups?)

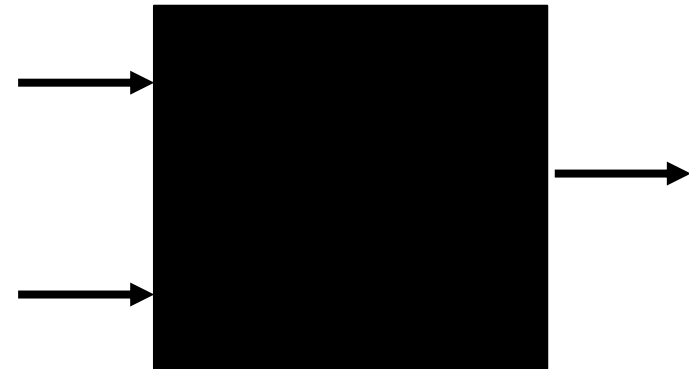
- Prakticky systémové testy při předávání zákazníkovi
- Ověřují splnění požadavků zákazníka
- Ověřují se uživatelské scénáře
- Black box
- „Testovací jízda“

- Testování prováděné po změnách v kódu (RefaktORIZACE)
- Potvrzení, že nový kód odpovídá specifikaci
- Potvrzení, že jiná funkcionality nebyla změnou ovlivněna
- „Black box“ testy zde vlastně hrají roli specifikace
- Vhodné jsou automatizované testy – CI?
- Obecně dobré provádět po větším zásahu do kódu

TESTOVÁNÍ – STRATEGIE

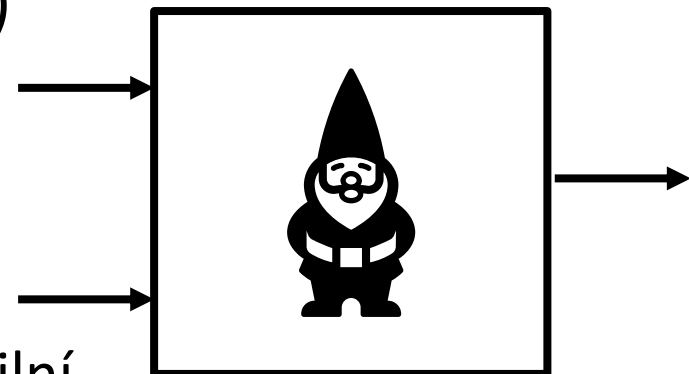
- **Černá skříňka (black box)**

- 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



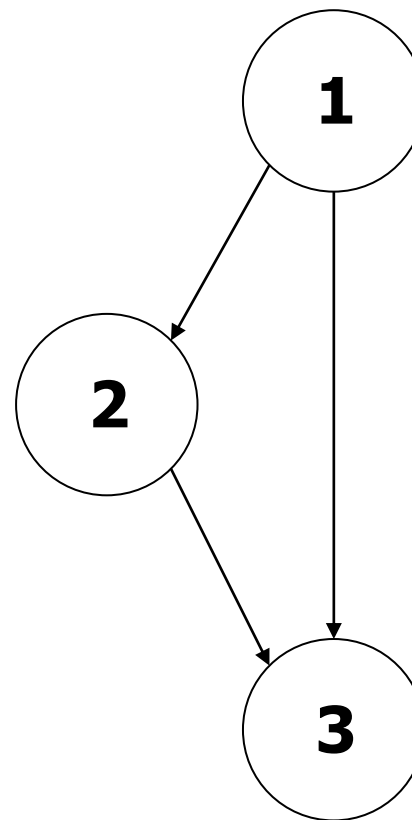
- **Statická analýza** zkoumá vlastnosti SW bez jeho provádění
- **Dynamická analýza** ověřuje vlastnosti SW na základě provádění kódu
- **Formální verifikace** s pomocí formálních metod ověřuje, že systém odpovídá specifikaci
- **Testování** zkoumá SW jeho spouštěním za účelem zvýšení jeho kvality

- Sledování, jak je aplikace pokryta testy
- Kolik případů užití máme pokrytých? Kolik zákaznických 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**

Program

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

Control flow graph

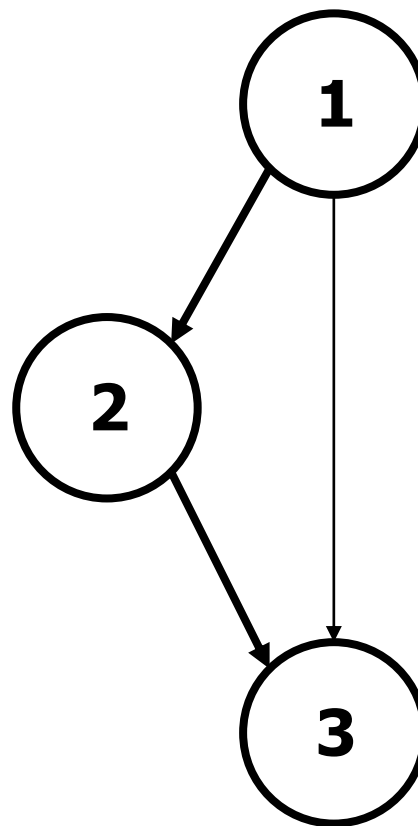


x=10, y=15

Program

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

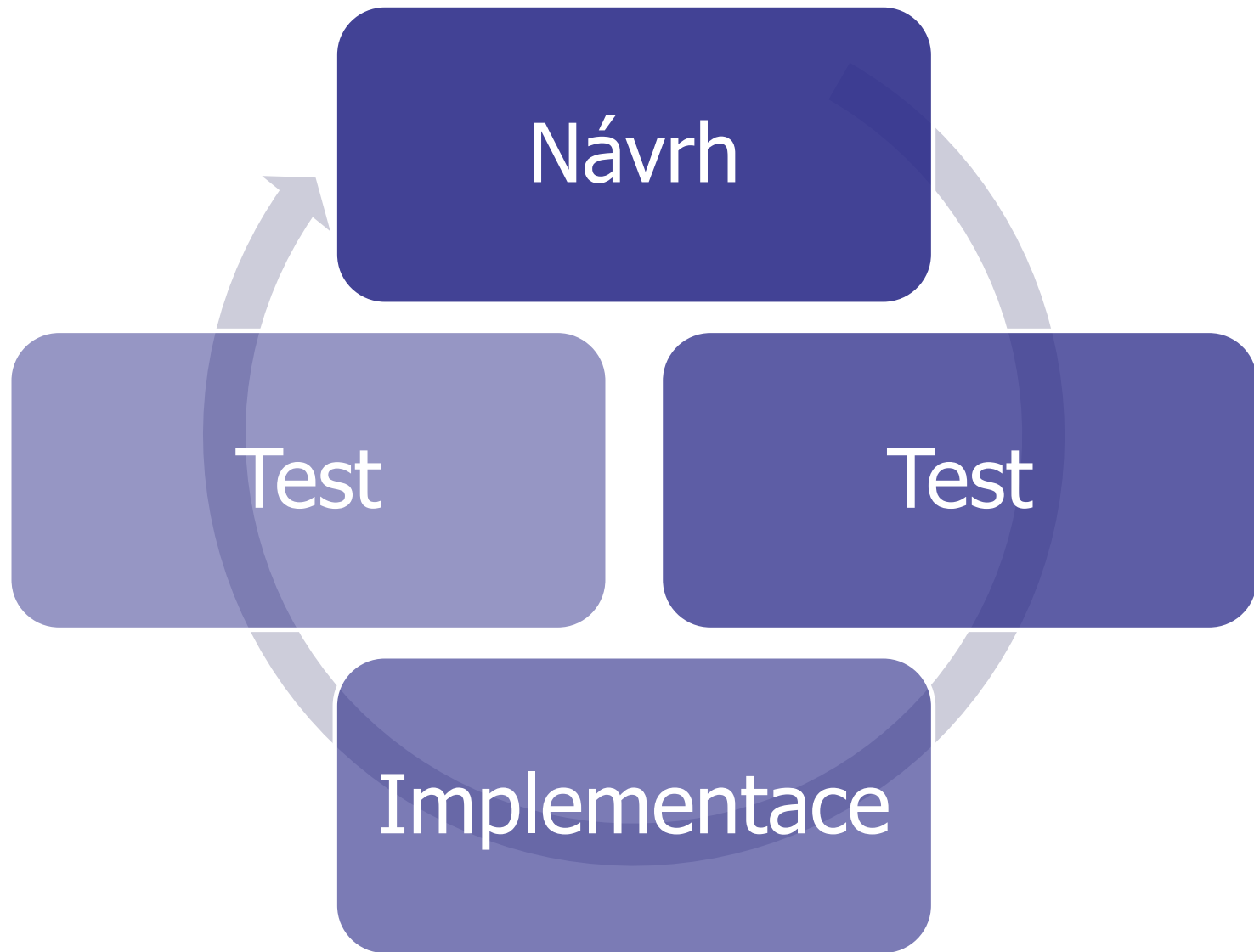
Control flow graph



- „Works fine on my machine.“
- „We didn’t have enough time to test.“
- „It’s OK, we are a start-up.“
- „It’s in beta, users will find the bugs.“
- „We don’t have enough money.“
- „We haven’t made any major changes.“
- „I didn’t think hackers would target us.“
- „They’re using it wrong.“
- <https://www.americaninno.com/boston/8-common-excuses-in-software-testing/>

TESTOVÁNÍ – TEST DRIVEN DEVELOPMENT

- Testy řízený vývoj, někdy také „programování řízené testy“
- Proces vývoje SW, kde je hlavní důraz kladen na testy
- Řadí se k agilním procesům
- Testování se snaží přenést co nejvíce na začátek projektu (nezapomene se na něj)
- Těží z automatizace spuštění testů (typicky unit testy použitelné nejen pro ověření nové funkcionality, ale i jako regresní testy)
- Předem napsané testy nejsou ovlivněny jakoukoliv znalostí implementace



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
 - Potom způsob, jakým byl kód napsán, nenarušil kód ověřovaný jinými testy
5. RefaktORIZACE (pročistění) kódu s průběžným ověřováním pomocí testů

- Všechny produkční kód je testovaný
- Je implementována pouze potřebná funkcionality
- 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

- Rozšíření TDD
- Vývoj je řízen popisem chování systému
- 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)

- 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 očekávaný výsledek nebo stav systému.

```
#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]

- Když je nutné testovat formálně:
 - Kosmický program, letectví, zdravotnictví,...
 - Formální analýza a verifikace (FAV)
 - Funkční verifikace číslíkových systémů (FVS)
- RDD – Readme Driven Development – Testování README
 - Testují se příklady v README
 - Vhodné zejména u veřejných knihoven
 - „Pokud nefunguje příklad v README, knihovnu většinou nepoužiji. “

- 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
- Selenium
 - Automatizace testování GUI aplikací
 - Ve webovém prohlížeči
 - Java API

- Urychlení vývoje a hlídání spolupráce v týmu
- Principy:
 - Centralizované repozitáře
 - Zdrojové soubory na jednom místě (např. GIT)
 - Automatická kompilace
 - Po nahrání 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...

- 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>

- <https://weblog.west-wind.com/posts/2018/Jun/16/Explicitly-Ignoring-Exceptions-in-C>
- <https://www.pinterest.cl/pin/421227371371087440/?lp=true>
- https://www.themarysue.com/wp-content/uploads/2010/10/foreveralone_justtheguy.png

ZADÁNÍ 1. PROJEKTU

http://ivs.fit.vutbr.cz/ivs_project_1_2021.zip