**Gymnázium Jablonec, U Balvanu 16, příspěvková organizace**

**2018**

**Evidence Rezervací**

Dokumentace k maturitní práci z informatiky

Kryštof Žucha

Vedoucí práce: Mgr. Pavel Červinka  
Oponent práce: Mgr. Radka Mrklasová

# Čestné prohlášení autora

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem v práci uvedl veškeré použité informační zdroje, které jsem při zpracování práce využil.

Kryštof Žucha, v Jablonci nad Nisou, 28. března 2018

# Obsah

Čestné prohlášení autora 2

Obsah 3

Úvod 4

Teorie 5

1. Obecná teorie programování 5

1.1. Úvod do problematiky 5

1.2. Vyšší jazyky 5

1.3. Jazyky s virtuálním strojem 8

1.4. Další jazyky související s programováním 8

2. Teorie objektově orientovaného programování (OOP) 9

2.1. Důvod vzniku 9

2.2. Základní programovací pojmy 10

2.3. Nevýhody Procedurálního programování 11

2.4. Fungování programu v OOP 12

Postup práce 14

Závěr 15

Zdroje 16

# Úvod

Výstupem této maturitní práce je program spustitelný v operačním systému Windows s nainstalovaným frameworkem .NET 4.5.2. či vyšším. Slouží k evidování rezervací a prodejů vstupenek na jakoukoliv akci, například na maturitní ples. Právě maturitní ples naší třídy mě inspiroval k vytvoření tohoto programu.

V této dokumentaci jsou popsány veškeré postupy, kterých jsem se držel při vytváření této práce. V teorii je mj. také popsána velmi stručně teorie objektově orientovaného programování. Podrobný postup vytváření je popsán v kapitole Postup práce. Zde uvádím, že jsem programoval v jazyce C# s frameworkem .NET, za pomoci IDE Visual Studio Community 2015. Veškeré obrázky jsem vytvořil v programu paint.NET, některé za použití volně stažitelných obrázků, jejichž zdroj je uveden na konci práce. Oba zmíněné programy jsou šířené zadarmo.

# Teorie

## 1. Obecná teorie programování

### 1.1. Úvod do problematiky

Nejprve rozeberu, co je to objektově orientované programování, a to proto, že se na naší škole na informatice neučí. Také vysvětlím běžné pojmy, které ve zbytku dokumentace používám. Začnu úplně zeširoka. Procesor počítače disponuje tzv. instrukční sadou, ta obsahuje příkazy, které procesor umí provádět. Jedná se o základní početní operace a operace pro práci s operační pamětí.

Programováním se rozumí psaní kódu pro procesor. Předáme-li procesoru kód (v binární podobě - např. instrukci č. 342 bychom procesoru předali jako 101010110), kde budou pod sebou vypsány jednotlivé instrukce z instrukční sady, procesor je přesně v tomto pořadí provede. Takovému kódu se říká strojový kód, a je to jediný kód, kterému procesor rozumí. Hovoříme tak o tzv. programovacím jazyce 1. generace - psaní strojového kódu. Chceme-li, aby procesor s jinou instrukční sadou vykonal stejný kód, musíme ho do této sady přepsat.

Nicméně takový strojový kód je velmi nepřehledný a nesrozumitelný, a aby se s ním pracovalo snadněji, vznikl tzv. Assembler, první programovací jazyk, který "zlidšťuje" příkazy z instrukční sady. Mluvíme o něm jako o jazyce 2. generace. V assembleru jsou naprogramovány základní funkce počítače, například jádro operačního systému. Kód z assembleru už je samozřejmě nutné procesoru "přeložit" do instrukcí z instrukční sady - do strojového kódu. Na to už slouží nějaký program, který ze jmen instrukcí udělá jejich číselné kódy.

### 1.2. Vyšší jazyky

Psal jsem, že Assembler je jakési "zlidštění" strojového kódu, nicméně i ten je velice složitý a vcelku nepochopitelný. Jiný způsob, jak začít práci s počítačem ovšem není. V assembleru je proto naprogramováno pouze jádro operačního systému, skrze který uživatel s počítačem komunikuje. V tomto jádře je již naprogramován jiný programovací jazyk, se kterým se pracuje dále. Typicky je to jazyk C. Říkáme o něm, že je "vyšší" než Assembler, protože se nachází jakoby v další vrstvě nad procesorem. Tento a všechny příbuzné nazýváme jako jazyky 3. generace. Kód v jazyce C je již lidsky vcelku snadno čitelný a není problém říci, co bude počítač dělat. Jazyk nám totiž poskytuje určitou úroveň abstrakce nad tím, co procesor přesně pro danou činnost udělá - již chápeme zdrojový kód z lidského pohledu na činnost, nikoliv z pohledu procesoru. O množině procesorů s nějakou instrukční sadou či přímo o nějakém operačním systému často hovoříme jako o tzv. platformě.

Na stejné či vyšší úrovni, jako je jazyk C, mohou samozřejmě existovat i další jazyky. Existují po té dva přístupy k tomu, jakým způsobem se k procesoru dostane strojový kód. Prvním způsobem je tzv. kompilace.

**1.2.1. Jazyky kompilované**

Kompilaci zprostředkovává program zvaný kompiler (z angl. compiler, česky též kompilátor). Tomu předáme zdrojový kód a on ho celý zkompiluje do podoby binárního spustitelného souboru obsahující strojový kód. Kompiler lze přirovnat k překladateli knih - dostane celý zdrojový kód (originál knihy), celý jej zkompiluje (přeloží) a výstupem je zkompilovaný soubor (přeložená kniha).

Výhody kompilace jsou následující:

* Rychlost vykonávání programu - procesoru je napřímo předán strojový kód, takže ho rovnou vykoná s maximálním možným výkonem bez jakýchkoli zdržení
* Snadná oprava chyb - je-li ve zdrojovém kódu chyba, kompiler ji odhalí během kompilace a sdělí, co je špatně
* Uzavřenost zdrojovému kódu - zkompilovaný kód již nemůže nikdo upravit nebo jej ukrást - chceme-li editovatelnost umožnit, dodáme program vč. zdrojových kódů. Zdrojový kód se dá ze strojového kódu získat pomocí tzv. reverzního inženýrství, což je velice náročný proces, vyžadující komplexní znalosti. "Odvozování zdrojového kódu" je obvykle zakázáno ve smlouvě s koncovým uživatelem o používání software.

Kompilace má i nevýhody:

* Nepřenositelnost - zdrojový kód je třeba zkompilovat zvlášť pro každou platformu, na které chceme program provozovat. Každá platforma tak vyžaduje svůj kompiler, takže mohou nastat určité odlišnosti mezi zkompilovanými programy pro dvě různé platformy, tedy se program může chovat pokaždé trochu jinak, což také komplikuje vývoj programu
* Správa operační paměti - procesor mechanicky vykonává strojový kód, bez toho, aniž by věděl, co bude následovat za instrukce, takže se může stát, že v operační paměti dojde k tzv. přetečení haldy (anglicky Stack Overflow), což znamená, že procesor do daného místa paměti uloží více dat, než kolik se do něj vejde, takže dojde k chybě, která změní hodnotu těchto dat. Např. rezervuje-li si procesor v paměti místo na 32 bitů dat a po té tam uloží číslo větší, než 2 147 483 647, dojde k přetečení a při příštím přečtení těchto dat z paměti procesor dostane jiné číslo, než to, které si tam původně uložil. Správu paměti tedy musí hlídat programátor, což zpomaluje a znesnadňuje vývoj
* Pomalý vývoj - kompilaci je třeba provést pro každé spuštění programu po změně zdrojového kódu, což zabírá čas. Problém to byl zejména dříve, kdy kompilace většího programu trvala vcelku dlouho, kvůli málo výkonným procesorům. Kompilace neumožňuje ladění programu za běhu, protože se provádí přímo ze strojového kódu. Zpomalení přináší také nutnost vlastní správy paměti. Veškeré vyšší datové struktury musí programátor navrhnout a naprogramovat, aby s nimi mohl procesor pracovat jako se základními datovými typy

Příkladem kompilovaných jazyků je výše zmíněné C, jeho následovník C++, Pascal nebo vcelku nový jazyk Dart.

**1.2.2 Jazyky interpretované**

Druhým způsobem je tzv. interpretace (z angl. interpretation, česky tlumočení - nepoužívá se). Zprostředkovává ji program interpreter, který je spuštěn při vykonávání kódu, řádek po řádku čte zdrojový kód a za běhu předává instrukce procesoru, který je vykonává. Jeho činnost lze přirovnat k opravdovému tlumočníkovi, dostane řádek kódu (mluvčí řekne větu), přeloží ji a sdělí instrukci procesoru (řekne překlad věty druhému mluvčímu) a takto dokola.

Výhody interpretace:

* Přenositelnost - díky tomu, že program je vykonáván de facto nepřímo na procesoru, ale přes interpreter, stačí, aby pro danou platformu existoval interpreter a náš program na ní poběží
* Správa operační paměti - vykonává ji přímo interpreter, který předejde možným chybám a tak je od této problematiky programátor odstíněn. V rámci interpreteru běží tzv. garbage collector (česky popelář, doslova "sběrač odpadu"), který programu rezervuje paměť a maže z ní data, která již nejsou potřeba (např. mezivýsledky početních operací).
* Rychlý vývoj - odpadá čekání na kompilaci, program lze ladit za běhu, protože víme, který řádek zdrojového kódu právě provádíme. Díky interpreteru máme k dispozici tzv. knihovny, což jsou hotové kusy kódu s připravenými datovými strukturami, které můžeme v rámci našeho programu využívat.

A nevýhody interpretace:

* Rychlost vykonávání programu - interpretace a automatická správa paměti zpomalují běh programu. U složitých a dlouhých výpočtů je interpretovaný program oproti kompilovanému nesmírně pomalý
* Obtížné hledání chyb - ladění za běhu sice zrychluje vývoj, ale není možné najít chybu ve zdrojovém kódu, dokud k ní nedojde při běhu programu. U komplexních programů se chyba projeví pouze někdy, při určitém sledu událostí, a to může být obtížné na opravu
* Otevřenost zdrojového kódu - program dodáváme přímo jako zdrojový kód, takže ho může kdokoli snadno upravit či ukrást

Příkladem interpretovaných jazyků jsou serverové jazyky PHP či Perl.

Dalšími jsou jednak skriptovací jazyky, jako je Lua, Python či Squirrel a dále prohlížečový jazyk Javascript. Tyto jazyky již patří do 4. generace jazyků a to proto, že již neběží přímo na procesoru či přes interpreter na procesoru, ale v rámci nějakého programu, který pro ně má vlastní interpretační část. U Javascriptu se jedná o webový prohlížeč, který podle javascriptového kódu vytváří dynamické doplňky webových stránek (např. různé kalendáře, mediální přehrávače, textové editory, galerie...). Zmíněné skriptovací jazyky slouží ke skriptování nějaké části programu, ve kterém běží - např. umělá inteligence či části úrovní v počítačových hrách nebo i způsob práce programu se soubory či zautomatizování nějaké činnosti pro lepší komfort uživatele. Tyto skripty pak dělají to, že spouští funkce zadefinované v jádře svého "mateřského" programu tak, jak je potřeba. Výhodou skriptů je, že jich můžeme snadno napsat větší množství bez nutnosti zásahu do programu - nemusíme přímo měnit funkce programu, jen je ve skriptu použijeme jiným způsobem.

### 1.3. Jazyky s virtuálním strojem

Vývoj jazyků šel samozřejmě dál, a proto vznikly jazyky s virtuálním strojem. To je v podstatě kombinace kompilovaných a interpretovaných jazyků. Komunikaci s procesorem zajišťuje onen virtuální stroj. Tato komunikace je rychlejší, než komunikace interpreta, a to proto, že zdrojový kód je nejprve zkompilován do binární podoby, ovšem jedná se o instrukční sadu virtuálního stroje, která je o mnoho jednodušší, než sada jakéhokoliv procesoru, proto je o hodně univerzálnější.

Virtuální stroj má tedy všechny výhody, které bylo možno vysledovat.

* Má možnost načíst připravené knihovny, což znamená velký programátorský komfort
* Kód se šíří v binární podobě
* Je nezávislý na platformě, protože stačí, aby pro danou platformu existoval virtuální stroj
* Disponuje automatickou správou operační paměti
* Chyby v kódu jsou odhaleny při kompilaci do mezikódu virtuálního stroje
* Program lze ladit za běhu, protože virtuální stroj ví, co vykonává, a umí se tak zpětně vrátit do daného bloku ve zdrojovém kódu

Rychlost vykonávání kódu je menší, než u kompilovaného jazyka, ale zase je díky kompilaci do virtuálního strojového kódu větší, než u jazyků čistě interpretovaných.

Existují dva masově používané virtuální stroje - Java a .NET. Oba mají přímou podporu Objektově orientovaného programování (více viz 2. kapitola teorie)

Java je v jádře multiplatformní jazyk, ve kterém vznikají programy pro počítače, chytré telefony, domácí mediatéky a mnoho dalších zařízení. Vyvíjí jej společnost Oracle. Ona podpora velkého množství platforem mu získala velkou popularitu, ovšem bohužel začíná zaostávat za .NETem.

.NET je virtuální stroj od Microsoftu, který je vyvíjen zejména pro platformu Windows, dále obsahuje framework ASP sloužící pro tvorbu webů a existuje také framework Xamarin sloužící k programování v .NET pro chytré telefony s Androidem a iOSem. Jeho primární jazyk je C#, ale umožňuje programovat i v C, C++ a ve Visual Basicu. V C# .NET je naprogramovaná i tato maturitní práce, takže se mu budu věnovat podrobněji dále (viz 3. kapitola teorie). Pro Linux existuje program Wine, jež umožňuje spouštět .NET programy dělané pro Windows.

### 1.4. Další jazyky související s programováním

Do předchozího výčtu se mi tyto jazyky nehodily, takže je zmíním nyní. První kategorií jsou jazyky databázové. V těchto jazycích se píší tzv. databázové dotazy. Tento dotaz je kód, který z databázové tabulky vytáhne určitá data, která přesně odpovídají určitým kritériím. Problematika databází je poměrně obsáhlé téma, které řeší efektivitu databází, způsoby, jak psát databázové dotazy apod. Nejpoužívanější databázové jazyky vycházejí z obecné definice jménem "SQL", např. MySQL ("moje", z angl. můj) či MS-SQL ("MicroSoft" - databázový jazyk od Microsoftu).

Další kategorií jsou tzv. značkovací jazyky. Jak jejich název napovídá, jejich účelem je cosi značkovat, a to text. Značkovací jazyk přiděluje nějakému prostému textu určitý význam. Nutno poznamenat, že práci se značkovacím jazykem nelze chápat jako programování, protože značkováním nevytváříme funkční kód.

Prvním značkovacím jazykem byl jazyk HTML, který vymyslel a zadefinoval v roce 1990 Tim- Berners Lee. Slouží k tvorbě statických webových stránek (dynamické prvky jsou vytvářeny v Javascriptu a serverovém jazyce, viz kapitola 1.2.2. teorie), tedy stránek přinášející pouze obsah, bez interaktivních prvků.

V HTML je možné text i stylovat, ale jak se technologie webů rozvíjela, přestalo stylování přímo v HTML stačit a vzniknul k tomu účelu jazyk CSS, který umožňuje styly interně dědit a tak je práce se stylováním mnohem snazší. Kolem těchto dvou jazyků se v roce 1993 zformovala organizace W3C, která spravuje webové jazyky, tedy HTML, CSS a Javascript. Na základě HTML pak ve W3C vznikl obecný značkovací jazyk XML, který používá k ukládání souborů mnoho programů (např. programy ze sady Microsoft Office - staré formáty .doc, .xls, atd. jsou ještě v proprietárním formátu Microsoftu, zatímco .docx, .xlsx, atd. jsou již interně ukládané jako XML - proto to "x" na konci přípony). Na základě XML vzniklo mnoho specializovaných značkovacích jazyků, jmenujme XHTML, mladší nástupce HTML a XAML, značkovací jazyk pro formuláře aplikací, viz 3. kapitola teorie.

Vedle XML existuje ještě JSON, který vyvíjí společnost Oracle. Oba formáty slouží ke stejnému účelu, každý z nich má své výhody a nevýhody, spočívající zejména v přístupu ke čtení a zapisování souboru, myslím ale, že není třeba je rozebírat.

## 2. Teorie objektově orientovaného programování (OOP)

### 2.1. Důvod vzniku

Nyní již rozeberu, co je to OOP a důvod jeho vzniku. Je samozřejmé, že každý kód ve finále běží tak, že se po sobě vykonávají jednotlivé instrukce, jinak to ani nejde. Nicméně programovat tímto způsobem je u většiny aplikací vzhledem k jejich komplexitě problém. Představme si jednoduchý textový editor. Je zde textové pole a několik tlačítek na uložení, načtení a stylování textu. Pokud uživatel nic nedělá, aplikace jen čeká a je vykreslena na monitoru. Jakmile uživatel klikne na některé tlačítko či naklikne textové pole, aplikace na to musí příslušným způsobem reagovat. V praxi to znamená, že v momentě, kdy na některém z jader procesoru běží vlákno s aplikací, vykonává se smyčka, která testuje, zdali nebylo stisknuto některé tlačítko, a pokud ano, spustí příslušnou akci tlačítka.

Od tohoto kódu je programátor odstíněn, a píše tak jen "obslužné bloky kódu", které se vykonají při stisku tlačítka či jiné události v aplikaci. Toto odstínění přináší v podstatě každý vyšší jazyk - jádro aplikace máme takto již připravené a definujeme tedy vzhled aplikace a jednotlivé její funkce.

"Klasické" programování, popsané výše, se nazývá procedurální - programátor napíše jednotlivé procedury aplikace, které se vykonávají při daných akcích. Jak se ale během vývoje programování rychle ukázalo, není to dostačující metoda programování. Na vývoj jednoduchých aplikací sice stačí, ale má několik vážných nedostatků a problémů, které brzdí a komplikují vývoj jakkoli komplexnějších programů. Tyto nevýhody odstranilo právě OOP, poprvé se objevilo zhruba v polovině 60. let 20. století. Jednotlivé nevýhody rozeberu dále.

### 2.2. Základní programovací pojmy

Nejprve však zavedu některé základní pojmy z oblasti programování.

Proměnná, je místo v operační paměti, do kterého si může program ukládat data pro pozdější využití. Proměnná je uchovávána jako sekvence bitů, tedy posloupnost jedniček a nul, které tak tvoří nějaké číslo binární soustavy. Velikost proměnné je délka binárního čísla, tedy počet bitů. Základní datové typy proměnných jsou boolean a byte. Boolean je "logická hodnota", nabývá tedy hodnot pravda a nepravda, resp. 1 a 0, má tedy velikost pouhý 1 bit. Byte je číslo z itervalu od 0 do 255 a má velikost 1 byte, tedy 8 bitů. Dalším běžným číselným typem je integer, velikost je 32 bitů, rozsah hodnot je tak cca. od -2 milionů do 2 milionů. Samozřejmě existují i větší číselné typy na uchovávání větších čísel. Práci s desetinnými čísly zajišťují speciální datové typy, které však interně pracují pomocí celočíselných.

Metoda nebo též funkce je blok kódu, který něco vykoná. Slouží k logickému členění kódu na funkční celky, např. v kalkulačce máme metodu pro získání čísel od uživatele a metody jednotlivých početních operací. Když chceme, aby se některá metoda vykonala, říkáme, že ji "zavoláme". Z hlavní kódové linie tedy voláme jednotlivé metody, které vždy vykonají část kýžené operace. Metody samozřejmě můžeme volat opakovaně. Každé metodě lze pak předat nějaká vstupní data, ta se označují jako parametry. Je to obdobné jako s parametry matematických funkcí (např. parametrem funkce sinus je úhel v radiánech), kupříkladu parametrem metody pro uložení souboru bude cílové jméno souboru (resp. cesta k tomuto souboru). Parametry se obvykle píšou do závorky za její název.

Smyčka nebo též cyklus je blok kódu, který se vykonává opakovaně. Z každé smyčky je samozřejmě třeba "vystoupit", tj. skončit vykonávání kódu ve smyčce, jakmile nastane nějaká podmínka, např. dosažení kýženého počtu cyklů (neboli iterací) či dosažení konce procházeného seznamu.

Bug, z anglického brouk, je chyba programu. Dělá-li program něco špatně, říkáme, že obsahuje bug. Procesu odstraňování bugů se říká debugging, jakým způsobem se dá provádět u jednotlivých typů jazyků viz. kapitoly teorie 1.2. a 1.3.

IDE, z angl. Integrated Development Environment, česky integrované vývojové prostředí, je program zjednodušující práci programátora. Obsahuje textový editor, který umí zvýrazňovat syntaxi programovacího jazyka a jedná-li se o kompilovaný jazyk, tak i compiler. Dále může disponovat celou řadou dalších funkcí, jako našeptávač syntaxe, nástroje pro debugging, automatické zálohování práce apod. Tato práce byla vytvořena ve velmi chytrém a profesionálním IDE od Microsoftu, jménem Visual Studio, konkrétně verze Community 2015, která je zcela zdarma. Toto IDE umí pracovat s celou řadou jazyků, a existují i rozšíření třetích stran na další jazyky.Mezi další známá ide patří např. Eclipse, NetBeans či PHPStorm.

Syntaxe nějakého programovacího jazyka je konkrétní množina slovních či znaménkových příkazů, které do kódu píšeme, a kterým rozumí kompiler/interpret. Každý jazyk se zapisuje trochu jinak, přestože kód ve finále dělá to samé. Většina moderních, běžně užívaných jazyků je tzv. C-like, z anglického "jako C", protože z jazyka C vycházejí. Jsou samozřejmě i jazyky, které mají syntaxi úplně odlišnou, např. MatLab či Fortran.

Pokud např. potřebujeme jen popsat nějaký algoritmus, nepoužijeme syntaxi nějakého konkrétního jazyka, protože je to zbytečně pracné, místo toho zapíšeme algoritmus v tzv. pseudo kódu. Tím se rozumí kód obsahující hypotetické příkazy, obvykle pojmenované tak, aby bylo jasné, co mají dělat. Pseudo kód tedy nemá syntaxi, každý programátor, který potřebuje něco v pseudo kódu napsat to udělá po svém. Každý programátor, který po něm bude tento kód číst, mu totiž porozumí, díky obecnosti přístupu. Je to jako, když např. napíšete jak nastartovat auto. Každý řidič tomu porozumí, přestože každé auto je trochu jiné, liší se kupříkladu dírkou startéru.

### 2.3. Nevýhody Procedurálního programování

Nyní již rozeberu, jaké nevýhody přináší procedurální programování.

Prvním velkým problémem jsou tzv. globální proměnné, tedy proměnné, ke kterým mají přístup všechny části programu. V běžném procedurálním programování je každá proměnná globální - jakmile je v kódu založena, můžeme s ní kdekoliv jinde v programu pracovat. Zní to sice jako výhoda, ale v komplexních aplikacích to přináší mnohá úskalí. Pro jednoduché jednorázové programy (často označované jako tzv. utility), které jen provedou nějaký výpočet či modifikují určité soubory je to i výhodnější, protože tento přístup představuje menší práci pro programátora.

U komplexních aplikací, které interně obsahují mnoho metod pro obsluhu všech funkcí aplikace je to velký problém, protože si tyto metody mohou vzájemně na proměnné sahat a modifikovat je. Čím více těchto metod existuje, tím větší nebezpečí hrozí. Např. metoda počítá s tím, že bude mít v proměnné očekávanou hodnotu, ale uživatel mezitím zavolá jinou funkci programu a hodnota proměnné se změní bez toho, aby se o tom metoda dozvěděla. Při vývoji je tedy třeba dávat si na toto pozor a zajistit správné chování pro všechny teoreticky možné případy, jinak program udělá chybu či spadne. Není možné si ve dvou metodách pojmenovat proměnnou stejně, což vede k nepřehlednosti kódu a přináší to příležitost dělat chyby spojené s užitím chybného názvu proměnné. Navíc pracuje-li na programu více lidí (což je u větších aplikací obvyklé), je prakticky nemožné se těmto chybám vyvarovat.

Další nevýhodou je uzavřenost struktur v kódu. Každá metoda dělá přesně to, k čemu byla naprogramována a není způsob, jakým by se dala její struktura znovupoužít. K sebemenšímu pozměnění jejího chování je třeba naprogramovat znovu velmi podobný kód nebo jí předat příslušný parametr, který chování pozmění, což ale mnohdy, s přibývajícím množstvím parametrů, zesložiťuje jejich strukturu.

Tyto nevýhody pak způsobovaly, že rozšiřování či upravování softwaru bylo velice náročné, a tudíž nákladné. V jistém momentu se tedy musel vývoj softwaru zastavit, protože již nebylo výhodné investovat do jeho vývoje nebo již dokonce nebyl udržitelný. Tím se rozumí, že vnitřní kód byl již tak složitý, spletitý a vzájemně provázaný, že jakýkoliv zásah by způsobil katastrofální rozbití celého programu.

### 2.4. Fungování programu v OOP

OOP vyřešilo všechny výše zmíněné problémy díky zavedení úplně nového přístupu k práci s kódem. Byla zavedena úplně nová filozofie, která změnila pohled na program jako celek. Ten nyní funguje na základě toho, že uvnitř něj spolu komunikují nějaké "objekty". Objekt je základní jednotka programu, nacházející se v operační paměti počítače. Objekt může obsahovat proměnné a metody.

Proměnné objektu se označují jako atributy, protože definují vlastnosti daného objektu, neboli vnitřní stav tohoto objektu. Např. objekt typu Člověk má atributy jméno, příjmení, datum\_narození, barva\_vlasů atd. Tyto proměnné již nejsou globálně a volně přístupné v kódu, lze se k nim dostat pouze přes objekt, pod který patří, protože je to logické. Dva objekty stejného typu budou mít stejné proměnné, jen v nich každý z nich bude mít jiné hodnoty. Navíc můžeme mít stejně pojmenovanou proměnnou i v objektu jiného typu, např. objekt typu Škola bude jistě mít také své jméno, a my tak tuto proměnnou také můžeme pojmenovat jméno. Je to výhodné, protože proměnná představuje data sloužící ke stejnému účelu, jen pro dva různé objekty.

Metody objektu představují schopnosti tohoto objektu. V OOP se metoda volá vždy tzv. "na objektu", protože ji vykonává daný objekt. Nemáme již globální metody, kterým musíme složitě sdělovat, přes spoustu parametrů, co se po nich chce. Pro názornou ukázku rozšířím výše zmíněný příklad. Chceme změnit jméno daného objektu. Proč se to v OOP musí dělat přes metodu viz níže. V procedurálním kódu bychom museli mít buďto metodu

*Změnit\_jméno(typ\_objektu, objekt, nové jméno)*

anebo dvě metody

*Změnit\_jméno\_člověka(člověk, nové\_jméno)* a *Změnit\_jméno\_školy(škola, nové\_jméno)*

Kdežto v OOP máme *Člověk.Změň\_jméno(nové\_jméno)* a *Škola.Zmeň\_jméno(nové\_jméno)*.

Což je o hodně přehlednější a logičtější. Navíc díky tomu, že víme, které metody lze na objektu volat, může je programátorovi zobrazit např. našeptávač v IDE, což zrychluje jeho práci, jelikož si snadno dohledá příslušnou metodu, u které přesně nezná syntaxi.

Kód programu je rozdělen do tzv. tříd.

# Postup práce

Popis, jak jste pracovali + printscreeny, obrázky, …

# Závěr

Zhodnotit, jestli jste zvládli vše, co jste si vytyčili.

# Zdroje

* How To ExplainRaster vs. Vector To YourClients: ucreative.com[online]. 2012, [cit. 2016-03-28]. Dostupný z: <http://www.ucreative.com/articles/how-to-explain-raster-vs-vector-to-your-clients>
* AUTOR, název článku: adresa webu[online, CD]. rok, [cit. rrrr-mm-dd]. Dostupný z: <odkaz na web>
* 130688-top\_foto2-0w8u1.jpg: novinky.cz[online]. 2008, [cit. 2017-11-11]. Dostupný z <https://media.novinky.cz/068/130688-top\_foto2-0w8u1.jpg?1277719201>
* BLÁHOVÁ, Renata. Literatura pro 1.ročník středních škol. Brno: Didaktis, 2008. ISBN 978-80-7358-115-2.
* PŘÍJMENÍ, Jméno. Název knihy. Místo vydání: Nakladatelství, rrrr vydání. ISBN 978-80-7358-115-2.