INFORMATIKA

Základní pojmy objektově orientovaného programování

TOMÁŠ PŘIBÁŇ – VÁCLAV VRBÍK

Západočeská univerzita v Plzni

Cílem tohoto článku je přivést čtenáře k pochopení elementárních základů objektově orientovaného programování.

K tomu, abychom správně pochopili a naučili se efektivně využívat objektově orientované programování, je nutností osvojit si „správný“ styl myšlení. Každý z nás vnímá a smýšlí o okolním světě trochu jinak. Matematik může okolí vnímat jako abstraktní numerické struktury, umělec jej vidí jako svět barev a tvarů. Jistě si teď položíte otázku, proč by nás mělo zajímat, jak kdo vnímá svět. Právě toto vnímání je pro programátora zásadní, protože on musí vytvořit počítačový program napodobující reálný svět. Z toho vyplývá i hlavní myšlenka objektově orientovaného programování (OOP) a tou je *co nejvěrněji a nejjednodušeji popsat svět skutečný.*

Asi se shodneme na tom, že většina z nás vnímá okolní svět jako věci. Hrnek, do kterého jste si ráno udělali čaj, je věc, židle, na které sedíte, je věc i auto, po kterém toužíte, je věc. V počítačové terminologii bychom pojem věc nenašli. Musíme hledat pod pojmem **Objekt**. To znamená, že hrnek je objekt, židle je objekt, auto je objekt, dokonce i zapomenutá peněženka v autě je objekt. Objektem je zde nejen věc, ale např. také člověk,

místo, pojem nebo třeba událost. Tento rozdíl nám dává určitou základní představu. V běžném životě „vnímáme svět jako věci“, v terminologii OOP „vnímáme svět jako objekty“.

Čtenáři, kteří jsou již seznámeni s procedurálním programováním, si jistě budou klást otázku, jaká je hlavní myšlenka tvorby programu při objektovém programování, respektive v čem je OOP tak převratné? Hlavní předností objektově orientovaného programu je, že se stává názornějším, srozumitelnějším a lze ho lépe spravovat. Pokud je totiž potřeba něco vylepšit, zpravidla se to týká jen několika objektů a zbytek programu lze nechat v původním stavu. To určitě přispívá k větší efektivitě návrhu a tvorby programu. Před nástupem objektově orientovaného přístupu se programy skládaly z globálních dat a z globálních funkcí, které s těmito daty pracují. Hovoříme o procedurálním stylu programování. Procedurální styl má řadu nevýhod plynoucích právě z globálního přístupu k datům. Vývoj takových programů byl však ze začátku velmi rychlý, ale v určitém okamžiku se takový program stává velmi těžce spravovatelným. Pokud je potřeba takový program upravit, je nutné zasáhnout na mnoha místech.

Objekt a instance

Nejlépe vše pochopíme, když si ukážeme názorný příklad z přírody – zvíře. Použijeme k tomu objekt *Zvire*. Zvíře můžeme popsat pomocí jeho **atributů** (popisují **vlastnosti** objektu *Zvire*) a jeho **metod** (popisují **chování** objektu *Zvire*). Každé zvíře patří k nějakému druhu a má jistou hmotnost a výšku. *Zvire* má samozřejmě mnohem více atributů, které by ho charakterizovaly, ale určitě se shodneme, že minimálně tři výše uvedené atributy charakterizují všechna zvířata. Metoda je činnost, kterou je schopen objekt vykonat. To znamená, že u našeho zvířete by to mohlo být jíst, běhat, spát.

Určitě už mnohé z vás napadlo, že se dosud bavíme spíše v abstraktní (teoretické) rovině než v rovině reálné (konkrétní). Abstraktní objekt si můžete představit jako zjednodušený popis reálného objektu (tj. bez podrobností). Když to převedeme na náš příklad, tak u abstraktního objektu *Zvire* se sice můžeme setkat s atributy *Druh*, *Hmotnost* a *Vyska*, ale tyto atributy stále nedefinují konkrétní hodnoty.

Usuzujete správně, abstraktní objekt slouží jen jako jakási šablona pro skutečný objekt. Skutečné zvíře má všechny atributy a metody definované v abstraktním objektu *Zvíře*, navíc je charakterizované podrobnostmi, které skutečné zvíře blíže specifikují. Abstraktní zvíře je tedy modelem

skutečného zvířete. Jak bylo již dříve uvedeno, abstraktní zvíře je definováno atributy *Druh*, *Hmotnost* a *Vyska* (neobsahuje však jejich konkrétní hodnoty). Skutečné zvíře je pak charakterizováno hodnotami přiřazenými k těmto atributům, například: *Druh = Pes*, *Hmotnost = 15 kg* a *Vyska = = 50 cm*.

Třída (class)

Ve většině objektově orientovaných programovacích jazycích je nutné předtím, než se vytvoří objekt, deklarovat třídu. **Třída** je šablona, pomocí níž se tvoří (skutečné) **objekty** (**instance** dané třídy) a definuje, jak budou nové objekty vypadat a pracovat. Použijeme-li opět předchozí příklad, můžeme říci, že *Pes* je instancí třídy *Zvire*.

Každý objekt z téže třídy má stejnou strukturu atributů a chování, které jsou definovány v této třídě. Je-li vytvořen objekt s názvem **o** ze třídy **T**, řekneme, že „**o** je instancí **T**“.

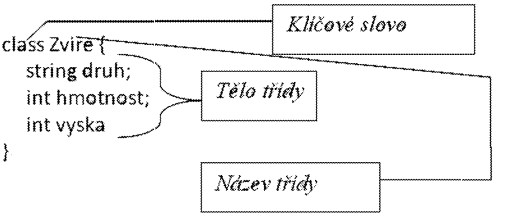
Na třídy a objekty se také můžeme dívat takto:

- Třída je to, co navrhujeme a programujeme.
- Objekt je to, co se z dané třídy vytváří za běhu aplikace.

Prozatím jsme se spíše bavili jen o atributech, objektech a třídách, ale ještě jsme se nezmínili o metodách. Ti z vás, kteří už prošli nějakým kurzem procedurálního programování, si mohou pojem metoda představit jako procedury a funkce. Pro ty, kteří žádným podobným kurzem neprošli, může posloužit představa, že metoda je činnost, kterou má objekt provést. Tato činnost je popsána pomocí skupiny příkazů, které se nacházejí v těle metody. Zároveň už jsme si uvedli, že činnosti, které by objekt Zvire mohl vykonávat, jsou jíst, běhat a spát; metody *Jist*, *Behat* a *Spat* jsou všeobecné (univerzální) a bude je obsahovat každé *Zvire*. Tyto všeobecné (univerzální) metody budou uvedeny bez parametru a to, které *Zvire* má jít spát, určíme tím, od které instance zvířete metodu Spat zavoláme. Konkrétní podoba a podrobnější popis toho, jak metodu vytvořit, je popsán dále v textu.

Definování třídy

Podoba definice třídy (obr. 1) závisí na programovacím jazyku. Z důvodu toho, že jedním z nejrozšířenějších programovacích jazyků pro OOP je Java, budou uváděny příklady kódu v jazyce Java. Čtenář musí ovšem dále pro sebe rozlišovat to podstatné, v čem spočívají základy OOP, od toho, co je jen ilustrativní, tj. způsob zápisu prvků OOP v jazyce Java.



Obr. 1

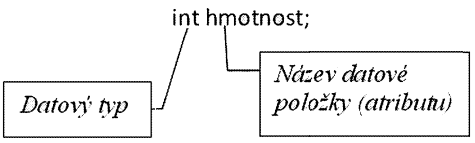
Uvedená definice třídy má tři části:

- klíčové slovo class
- název třídy
- tělo třídy

Deklarace atributu třídy

Z uvedeného příkladu jsou zřejmé základní části deklarace atributu třídy:

- datový typ
- název datové položky (atributu)



Obr. 2

Jak definovat metodu

Jak již bylo uvedeno, metody popisují chování objektu. Metodu si můžeme představit jako skupinu příkazů provádějících určitou činnost, která je definována následujícími položkami:

```
NávratovýTyp jménoMetody (parametry ) {
      // tělo metody
}
```

- `NávratovýTyp` musí být některý datový typ (např. int, float, boolean, ...). Návrat z metody a případné předání návratové hodnoty způsobí příkaz `return`. Je-li návratovým typem `void`, metoda nevrací hodnotu. Příkaz `return` provede návrat z metody a vrací řízení programu zpět za příkaz, který metodu vyvolal. Zápis příkazu `return` je

  `return výraz;`

  Tento příkaz vrací hodnotu výrazu „výraz“, který musí být shodného typu, jako je návratový typ metody [5].
  Příkaz `return` bychom využili například v případě, že při definování metody *Behat*, chceme, aby nám metoda po úspěšném „přeběhnutí“ *Zvirete* z bodu A do bodu B vrátila hodnotu *true* (logická 1), v opačném případě (nedojde k „přeběhnutí“ zvířete) hodnotu *false* (logická 0). Pozici bodů A a B zjistíme pomocí seznamu parametrů. Dále v metodě definujeme lokální proměnnou s názvem stav, která je logického datového typu. V takovém případě by zdrojový kód vypadal nějak takto:

```
boolean Behat (int bod_A_x,int  bod_A_y, int bod_B_x, int
               bod_B_y) {
     boolean stav;
     //zdrojový kód vykonávající přesun Zvirete z bodu
     A do bodu B, podle zadanych souradnic
     return stav;
}
```

- `jménoMetody` musí být platný identifikátor (Identifikátory v Javě mohou mít neomezenou délku. Malá a velká písmena se považují za rozdílné znaky, což znamená, že např. identifikátory chobot a Chobot jsou navzájem různé. Každý identifikátor musí začínat písmenem nebo podtržítkem. Zbylá část víceznakového identifikátoru může navíc obsahovat číslice „0“ ... „9“ a znak „$“) [5].
- `parametry` – často se setkáte s tím, že metody k tomu, aby provedly požadovanou činnost, potřebují další data. Těmto potřebným datům se říká `seznam parametrů`. Parametr je deklarován stanovením svého datového typu a názvu (identifikátoru). Velice se to podobá deklarování atributů jen s tím rozdílem, že deklarace je v kulatých závorkách a zobrazuje se vpravo od názvu metody. Ještě malá poznámka: deklarace parametrů musí být odděleny čárkami. Příklad:

```
void  metoda ( int x, int y ) {
      // tělo metody
}
```

- tělo metody je vymezeno počáteční a koncovou složenou závorkou – všechny příkazy uvnitř těchto závorek se nazývají *blok kódu* [1].

Jak to celé spustit

Dříve než se začneme zabývat náležitostmi, které musí program obsahovat pro to, abychom ho mohli spustit, si musíme říci, jaká je vůbec funkce objektů. Jak už jsme si uvedli výše, tak objekty v sobě zahrnují nejen data, ale i metody, které nad uvedenými daty pracují. Data i metody jsou navzájem svázány takovým způsobem, že objekt můžeme předat z jedné části programu do jiné a obě části mohou přistupovat nejen k datovým atributům, ale přístupné jsou i operace, které vykonávají metody. Takže například objekt typu řetězec (string) poskytuje nejen prostor pro uložení znaků řetězce, ale poskytuje i metody pro provádění operací nad uloženým řetězcem – vyhledávání, změnu malých písmen na velká, určení délky řetězce a podobně. Hlavní využití objektů je při tvorbě větších programů, které se skládají z několika objektů; jeden se stará o výstup na obrazovku a zpracovává pokyny od uživatele, další provádí výpočty, ještě jiný posílá data po síti atd. Každý tento objekt je naprogramován do značné míry samostatně nejen proto, aby se v něm autor po půlroce sám vyznal, ale zejména proto, že se již hotové a funkční objekty dají použít i v dalších programech.

Teď když jsme si nastínili funkci objektů a víme jak definovat třídu a její atributy a metody, musíme umístit třídu do programu. Pro lepší přehlednost si umístíme naši třídu *Zvire* mimo hlavní část programu. Hlavní program, přesněji řečeno metoda, která je vyvolána po spuštění programu jako první, se musí jmenovat *main* a musí být v programu vždy uvedena. Tato metoda musí být „zapouzdřena“, tj. musí být uvnitř v nějaké třídě (viz dále). Na jméně třídy nám zatím nezáleží, ale musí se tato třída bezpodmínečně jmenovat jako soubor, ve kterém je uložena, včetně dodržení stejných malých a velkých písmen v názvu [2]. Nejlépe celou situaci opět pochopíme z příkladu:

```
public class MujPrvniProgram {
public static void main(String argumenty[ ]) {
```

```
        Zvire zviratko = new Zvire();
        boolean vysledek;
        vysledek = false;
        vysledek = zviratko.Behat(0,0,15,15);
        if (vysledek)
            System.out.println("Zvire uspesne prebehlo.");
        else
            System.out.println("Zvire bohuzel neprebehlo,
                        zkontrolujte zadane body.");
        }
}
class Zvire {
        string druh;
        int hmotnost;
        int vyska;
        boolean Behat (int bod_A_x, int bod_A_y, int bod_B_x,
                       int bod_B_y) {
          boolean stav;
          //zdrojový kód vykonávající přesun Zvirete
            z bodu A do bodu B, podle zadanych souradnic
        return stav;
        }
}
```

Podívejme se, jak to funguje. První část zdrojového kódu definuje hlavní třídu *MujPrvniProgram*. Zbývající část obsahuje definici třídy *Zvire*. Půjdeme řádek po řádku a postupně si ukážeme, co jsme vlastně vytvořili. Hned v prvním řádku vás možná zarazí slovíčko *public* – toto slovíčko označuje veřejnou třídu, můžeme si to představit jako, že k ní má přístup „každý“. Po definici hlavní třídy *MujPrvniProgram* následuje definice hlavní části programu – metoda *main*. Pro metodu *main*() platí určitá pravidla:

- Musí být uvedena ve třídě, která je označena jako public.
- Musí mít přesně tuto podobu:

```
public static void main (String [ ] args) {
        tělo metody main
}
```

- Nesplňuje-li *main*() zmíněné požadavky, program nebude možné přeložit.

Dalším příkazem vytváříme objekt (instanci) třídy *Zvire* s názvem *zviratko*. Následuje deklarace proměnné s názvem *vysledek*, která je datového typu *boolean*. Poté do proměnné *vysledek* přiřadíme hodnotu *false*. V dalším řádku voláme metodu *Behat*() a předáváme jí čtyři parametry. Tato metoda se nachází v třídě *Zvire* a slouží k „přeběhnutí" zvířete z bodu A do bodu B a vrátí návratovou hodnotu datového typu boolean pomocí příkazu return. Všimněte si, že v příkazu volajícím metodu *Behat*() dojde ke dvěma věcem.

Nejdříve je volána metoda *Behat*(). Po dokončení činnosti metody *Behat*() je hodnota vrácená touto metodou uložena do proměnné *vysledek* pomocí operátoru přiřazení. Proměnná *vysledek* je poté použita v podmíněném příkazu if. Na obrazovce je následně zobrazena zpráva, kterou program vybral na základě logické hodnoty proměnné *vysledek*.

Dědičnost

Jedním ze základních kamenů OOP je bezpochyby dědičnost. Asi každý si už podle názvu dokáže udělat představu co to dědičnost je. A skutečně pojem dědičnost v OOP se velmi podobá reálné dědičnosti v běžném životě, tudíž tomu kdy dědíme určité vlastnosti (predispozice) od našich rodičů. Její užitečnost si ukážeme hned v následujícím příkladu.

Představte si, že nebudeme definovat objekt *Zvire*. Místo toho jeho atributy a metody umístíme do všech objektů představujících jednotlivá zvířata. Vytvoříme tedy například objekt psa, kočky, velryby, želvy a koně. Neboli atributy a metody se opakují pětkrát – jednou pro každý z pěti objektů.

Ale přišel za Vámi zoolog s prosbou, že by potřeboval evidovat ještě u každého zvířete jeho délku. Pro nás to tedy znamená, že potřebujeme přidat atribut délka. Protože jsme si na začátku nevytvořili objekt Zvire, který by „zastřešoval" námi vytvořené objekty, tak nám nezbývá nic jiného než pracně vložit atribut délka do každého z pěti objektů.

A nyní se ukáže síla dědičnosti, protože přemýšlivý programátor by si nejdříve vytvořil jeden objekt, do kterého by seskupil atributy a metody, které jsou pro všechny objekty společné (shodují se u většího počtu objektů). Konkrétně v našem případě by si vytvořil objekt *Zvire*, v kterém by byly všechny společné atributy a metody. Ostatních pět objektů zdědí objekt *Zvire* spolu s jeho atributy a metodami. A nyní by už realizace

žádosti zoologa nebyl takový problém, protože přidání atributu délka se provede v jediném objektu – *Zvire*. Zbývající objekty automaticky obdrží atribut délka zděděný z objektu *Zvire*.

Ještě si ukážeme názornou pomůcku pro rozpoznání aplikace dědičnosti. Vztah **generalizace-specializace**, který je klíčový pro volbu užití dědičnosti, můžeme často rozpoznat pomocí vztahu „**je**“. V našem příkladě si můžeme říci „*Pes* **je** *Zvire*“. To znamená, že každý *Pes* je také *Zvire*, je tedy zvláštním případem zvířete. *Zvire* je tedy generalizací, *Pes* specializací. Pokud si můžeme říci T2 **je** T1, pak v mnoha případech platí, že T2 je specializací T1 a T1 generalizací T2. Ve vztahu dědičnosti to znamená, že třída T1 je nadřízenou třídou třídy T2 [6]. Je důležité si uvědomit, že **vztah „je“ není symetrický**, to znamená, že neplatí obráceně. Nelze říci, že *Zvire* je *Pes*, neboť každé *Zvire* není *Psem*. Z tohoto tvrzení je zřejmé, že základní třída je chudší než odvozená – odvozená třída obsahuje všechny členy základní (viz vztah je), ale obráceně to neplatí. Díky tomuto vztahu si můžeme v programu vytvořit třeba pole zvířat a do něj umisťovat střídavě psy, kočky, koně, velryby, ...

Zapouzdření

Další velkou výhodou, kterou nám OOP poskytuje je zapouzdření. Co je vlastně na tom zapouzdření tak ohromujícího? Zjednodušeně se dá říci, že zapouzdření nám zajistí, aby s atributy nebo metodami, které se nacházejí v objektu, nám nemanipuloval jiný objekt, který k tomu nemá právo. Neboli slovy Rudolfa Pecinovského: „Nikdo nesmí mít šanci zjistit, jak to dělám, že umím to, co umím [3].“ Tuto schopnost oceníme především v případě složitějších programů, abychom ani omylem nemohli ovlivnit chod některé jiné části.

Zapouzdření si můžeme představit tak, že máme všechny související atributy a metody „pod jednou střechou“. Ačkoliv je to pravda, skutečným důvodem používání zapouzdření je prevence vzniku chyb. Pokročilejší čtenář, který má zkušenosti s procedurálním programováním při porovnání obou způsobů programování zjistí, že v procedurálním programování neexistuje dostatečná ochrana proti nesprávnému použití atributů a procedur. Pokud si ale pod atributy představíme jakékoliv datové položky, pak v procedurálním programování jejich ochrana před neoprávněnou manipulací „zvenku“ existuje – tou je lokální deklarace. Proměnné a procedury deklarované lokálně v nějaké proceduře slouží jen této proceduře a nejsou „zvenku“ nijak přístupné. Jedná se ovšem o jinou úroveň ochrany než u tříd

v OOP. To v OOP řeší právě zapouzdření, které umožňuje programátorovi umístit atributy a metody do třídy a následně stanovit pravidla, která budou sloužit ke kontrole přístupu [1].

Polymorfismus

Kdybyste se podívali do slovníku cizích slov a vyhledali si pojem polymorfismus, našli byste jako význam slova: mnohotvárnost, různorodost. Teď si pravděpodobně budete říkat co to má společného s OOP. Právě polymorfismus je vlastnost OOP, která programátorovi velice usnadňuje práci.

Vraťme se zpět k našemu příkladu se zvířaty. Představte si, že v naší aplikaci *MujPrvniProgram* budeme mít kromě hlavní třídy ještě dvě další třídy: třídu *Zvire* a třídu *Clovek*. V obou třídách budeme mít deklarovány určité atributy, které budou jednotlivé objekty popisovat. Dále tyto třídy budou obsahovat metodu Zobrazit(), která by měla zobrazit informace o objektech třídy *Zvire* a o objektech třídy *Clovek*. Pozornější čtenáře jistě napadne, vždyť se obě metody v obou třídách jmenují stejně! Mají sice podobné chování (úkoly), ale obě budou odlišně provedeny. A zde právě nastupuje polymorfismus.

Představte si, že by pro metodu *Zobrazit*() byl v každé třídě použit jiný název, například *ZobrazitZvire*() a *ZobrazitCloveka*(). Programátor, který by chtěl tyto třídy použít, by si musel zapamatovat názvy všech metod každého z objektů. To může být obtížné zvláště v případě, kdy třída obsahuje velké množství metod.

Programátoři se problémům s metodami a názvy metod vyhýbají prostřednictvím polymorfismu. Definují v každé třídě metodu se stejným názvem, která má podobné chování. Díky tomu si musí zapamatovat pouze jeden název, který je s tímto chováním spojen.

Pokud je to pro vás stále moc zamotané tak se pojďme podívat na příklad polymorfismu z reálného života. Asi každý ví, co to je dálkový ovladač, jak se ovládá a k čemu slouží. Jistě budete souhlasit s tvrzením, že dva dálkové ovladače, které vypadají úplně stejně mohou ovládat rozdílná zařízení (např. televizi a DVD přehrávač). Polymorfismus v tomto případě zahrnuje dvě položky se stejným názvem (dálkový ovladač), provádějící stejný úkol (ovládání „něčeho“), které se ovšem vnitřně odlišují.

Polymorfismus neslouží jenom pro usnadnění života programátora-sklerotika, ale ve spojení s dědičností hlavně k tomu, abychom mohli projít v cyklu námi vytvořené pole různých zvířat, každému zavolali jeho metodu

/Zarvi()/ a na tento pokyn by pes zaštěkal, kočka zamňoukala, atd. – tedy každý by udělal to, co je jemu vlastní, co mu ukládá jeho metoda.

Co říci závěrem

Existuje velké množství programovacích jazyků umožňujících objektově orientované programování, např. Smalltalk, Java, C#, C++, Object Pascal, Visual Basic. NET, Lisp, PHP, Python, Ruby. Z těch nejpoužívanějších je potřeba uvést především Javu, C#, C++ a PHP [4].

Samozřejmě nikdo neočekává, že po přečtení tohoto článku budete znát objektově orientované programování. Ale základní a nezbytný krok k tomu, abyste mohli začít tvořit svoje vlastní objektově orientované programy, je pochopit základní principy a myšlenky OOP. K tomu vám měl tento článek dopomoci.

Literatura

[1] *Keogh, J.* – *Giannini, M.*: OOP bez předchozích znalostí. Brno : Computer Press, 2006.

[2] *Herout, P.*: Učebnice jazyka Java. České Budějovice, Kopp, 2001.

[3] *Pecinovský, R.*: Myslíme objektově v jazyku Java 5.0. Grada, Praha 2004.

[4] Objektově orientované programování [online]. 2008 , 22. 11. 2008. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD>.

[5] Studentský informační server Dioné [online]. 4. 2. 2001. Dostupný z WWW: <http://v1.dione.zcu.cz/java/sbornik/11.html>.

[6] Programování se zaměřením na. NET a jazyk C# [online]. 27. 7. 2006. Dostupný z WWW: <http://projektysipvz.gytool.cz/ProjektySIPVZ/Default.aspx?uid=236>.

GeoGebra – více než dynamická geometrie

LUKÁŠ HONZÍK – MIROSLAV TICHÝ

Fakulta pedagogická ZČU v Plzni,
Střední škola aplikované kybernetiky Hradec Králové

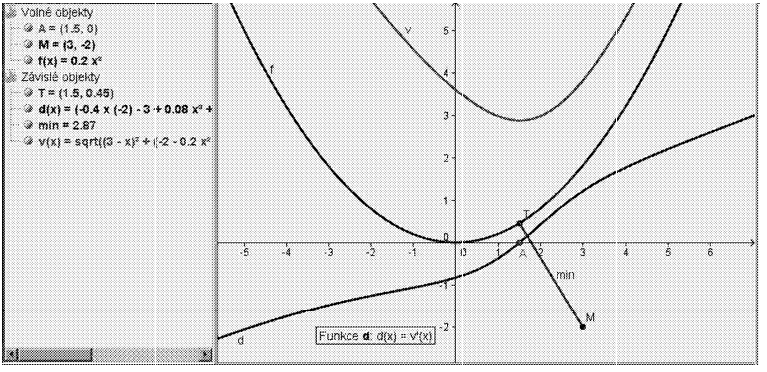
(*Dokončení*)

Příklad 2 – *Nalezení bodu na parabole splňujícího danou podmínku*

Zadání: Najděte bod paraboly $y = 0{,}2x^2$ nejbližší danému bodu $M = [3, -2]$.

Tento příklad se má řešit kombinací geometrických a algebraických prostředků.

Řešení **a)** Parabola je dána předpisem `f: y=0,2x^2`. Vytvoříme funkci $v(x)$ reprezentující vzdálenost bodu M od bodu $[x, f(x)]$ této paraboly. Extrém (minimum) této funkce je už hledaná vzdálenost. Extrém této funkce chceme hledat pomocí derivace. Problém je, že GeoGebra umí hledat extrémy jen u polynomů. Derivaci lze však snadno sestrojit jako funkci, úlohu převést na hledání průsečíku grafu této funkce a osy x, tím získáme nulový bod derivace, následně hledaný extrém. Celé řešení je vidět na obrázku 5a.

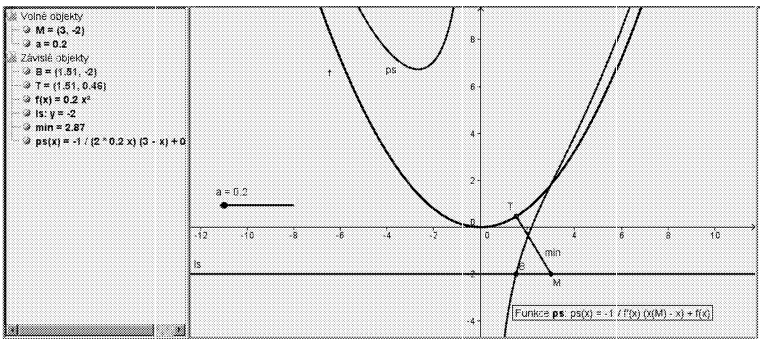


Obr. 5a Řešení pomocí derivace

Řešení **b)** Chceme úlohu řešit obecněji, nejen pro jednu kvadratickou funkci, ale pro libovolnou funkci $f : y = ax^2$ s parametrem a. Parametr budeme měnit posuvníkem, příslušně se budou dle jeho hodnoty měnit grafy v geometrickém okně i hodnoty v algebraickém okně. Předchozí řešení musíme upravit. Hledaným bodem T paraboly bude procházet normála jdoucí i bodem M.

Rovnice normály je $n(x) = -\frac{1}{f'(x_0)}(x - x_0) + f(x_0)$. Bod M na normále leží, bude v této rovnici figurovat takto: $M = [x, n(x)]$. Neznámá je x_0, tj. x-ová souřadnice hledaného bodu T na parabole.

Tuto rovnici lze řešit graficky hledáním průsečíku dvou křivek reprezentujících levou a pravou stranu rovnice normály. Levá strana této rovnice je rovna $n(x)$, tedy y-ové souřadnici bodu M, na obrázku 5b je popsána ls. Křivku ps reprezentující pravou stranu rovnice sestrojíme dle rovnice normály s tím, že se jedná o funkci proměnné x_0, x je v tomto předpisu x-ová souřadnice bodu M. Průsečík B grafů ls a ps pak už určuje x-ovou souřadnici hledaného bodu T na parabole. Ten je v konstrukci určen algebraicky pomocí svých souřadnic: $T[x(B), f(x(B))]$. Pak lze konstruovat (a samozřejmě měřit) hledanou minimální vzdálenost.



Obr. 5b Úsečka MT určuje hledanou minimální vzdálenost

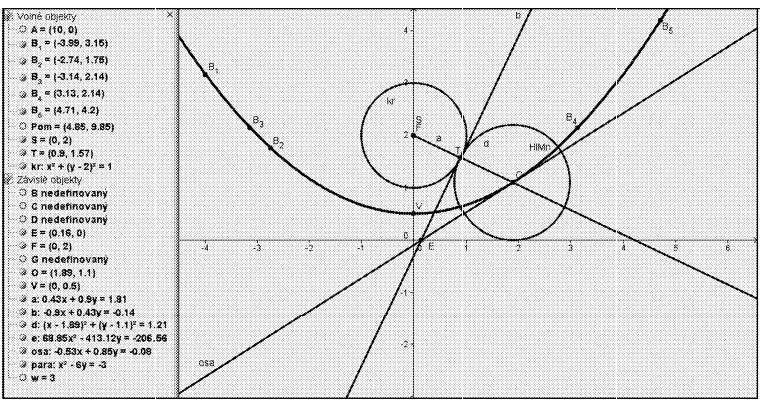
V této úloze je pěkně vidět kombinace použití algebraických a geometrických prostředků programu.

Příklad 3 – *Konstrukce množiny bodů dané vlastnosti*

Zadání: Je dána kružnice a přímka. Určete množinu středů všech kružnic, které se vně dotýkají kružnice dané a daná přímka je jejich tečnou.

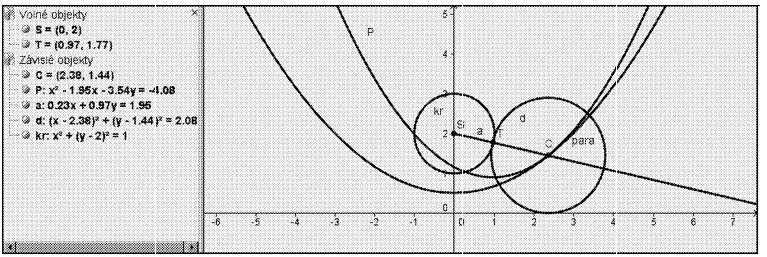
Řešení konstrukční: Bez újmy na obecnosti zadání úlohy můžeme volit kružnici k o rovnici $x^2 + (y-2)^2 = 1$, danou tečnou bude osa x. Volíme libovolný bod T na dané kružnici k. Střed příslušné hledané kružnice pro tento bod dotyku leží na polopřímce ST. Bod T zároveň určuje i společnou tečnu kružnice dané a hledané. Střed hledané kružnice pak leží na ose úhlu tvořeném touto tečnou a tečnou danou. Takto dostaneme jeden střed, jednu kružnici vyhovující zadání. Úlohu dokončíme a příslušnou hledanou množinu středů všech kružnic vyhovujících zadání sestrojíme pomocí nástroje „Množina bodů“, který je podobný stejnému nástroji Cabri Geometry (obr. 6).

Část „*algebraická*“: Získali jsme hledanou množinu bodů, chybí nám ale její analytické vyjádření. Na získanou množinu umístíme libovolných 5 bodů, „zkušebně“ jimi proložíme kuželosečku. Ta evidentně pokrývá celou získanou množinu bodů, jedná se o parabolu. Problém nastává s analytickým vyjádřením této paraboly. Při změně umístění některého z pětice bodů určujících parabolu se mění v okně algebry rovnice této paraboly. GeoGebra totiž nezjednodušuje výraz – rovnici získané paraboly. Potřebujeme ukázat, že se jedná o stále stejnou parabolu. To můžeme vyřešit pomocí algebraických možností programu, určíme parametr a ohnisko získané paraboly. Použijeme postupně příkazy `w = parametr [e]`, `F = ohnisko [e]`, `V = vrchol [e]`, kde e je námi proložená kuželosečka. Tyto hodnoty již nezávisejí na poloze bodů parabolu určujících. Pak už snadno zapíšeme rovnici paraboly v kanonickém tvaru: $(x - x(V))^2 = 2 \cdot w(y - y(V))$.



Obr. 6 Parabola jako množina bodů

Konstrukci můžeme provést i jinak s použitím prostředků, které nemáme při klasickém rýsování. Sestrojíme bod T a polopřímku ST jako v předchozím řešení a pak množinu všech bodů v rovině stejně vzdálených od bodu T i od dané tečny – osy x. Touto množinou je parabola, kterou snadno sestrojíme jediným příkazem – `Parabola [T, OsaX]`. Průsečík této paraboly a polopřímky ST je hledaný střed kružnice.



Obr. 7 Druhé řešení příkladu 3

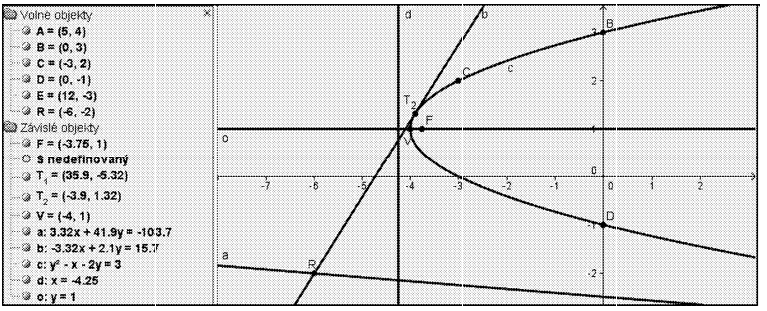
V této úloze, v níž je řešení jak geometrické, tak i částečně algebraické s využitím analytické geometrie, byly ukázány možnosti programu při práce s kuželosečkami.

Příklad 4 – *Rozbor kuželosečky*

Zadání: Jsou dány body $A = [5; 4]$, $B = [0; 3]$, $C = [-3; 2]$, $D = [0; -1]$ a $E = [12; -3]$, které určují kuželosečku. Zjistěte, o jakou kuželosečku se jedná, a nalezněte její střed, vrcholy, osy, ohniska a případné asymptoty nebo řídící přímku. Dále určete tečny z bodu $R = [-6; -2]$ i dotykové body.

Řešení: Nejprve zobrazíme zadané body (pomocí příkazu `A = (5, 4)` nebo můžeme udělat to, že bod umístíme libovolně kamkoliv na nákresnu a poté v algebraickém okně postupně přepíšeme jeho souřadnice na požadované hodnoty) a následně jimi příkazem `c = Kuzelosecka [A, B, C, D, E]` proložíme kuželosečku (obr. 8). V této chvíli je již splněn první úkol ze zadání, neboť po najetí kurzorem na vytvořenou kuželosečku se zobrazí informace o typu kuželosečky a její rovnici. V dalším kroku máme zjistit významné body a objekty svázané s kuželosečkou. V prostředí programu na vše dotazované existují příkazy: střed (pokud existuje) zjistíme zadáním příkazu `Stred [c]`, vrcholy příkazem `Vrchol [c]`, hlavní a vedlejší

osu příkazy `HlavniOsa [c]` a `VedlejsiOsa [c]`, atd. Stejně jednoduše nalezneme i tečny z bodu R pomocí příkazu `Tecna [R, c]` a dotykové body jako průsečíky kuželosečky a tečných přímek. Pokud požadovaný objekt pro zobrazenou kuželosečku neexistuje, je uveden v algebraickém okně jako „nedefinovaný“ nebo se ani v grafickém ani v algebraickém okně nezobrazí vůbec. Ovšem při změně polohy některého z bodů a přechodu kuželosečky v jinou kuželosečku, která již má daný objekt definovaný (například asymptoty u hyperboly), se tento objeví.



Obr. 8 Informace o typu kuželosečky se zobrazí po najetí kurzorem na její rovnici

Příklad 5 – *Apollóniova úloha*

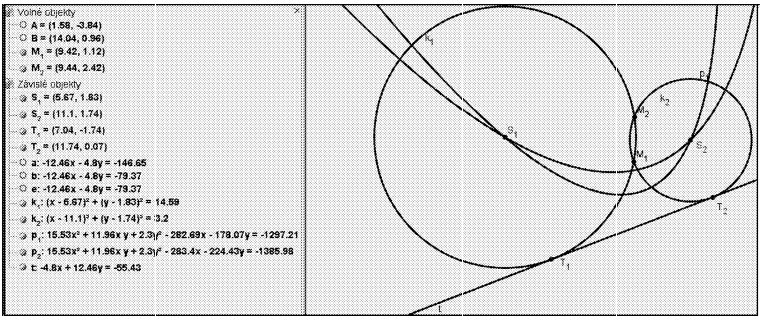
Zadání: Sestrojte kružnici procházející danými dvěma body M_1, M_2, dotýkající se dané přímky t.

Tento příklad je spíše zajímavost ukazující neklasické řešení klasické úlohy.

Řešení: Jedná se o Apollóniovu úlohu „bbp“. Tato úloha je obvykle řešena v prvním ročníku střední školy pomocí mocnosti bodu ke kružnici a Euklidových vět. Aparát programu GeoGebra dává možnost rychlejšího řešení, i když použitelného až po probrání kuželoseček (ne nutně analytických vlastností, ale „stačí“ syntetických vlastností např. v deskriptivní geometrii).

Hledáme střed kružnice splňující podmínky, že je stejně vzdálen od daného bodu M_1 a od tečny t. Leží proto na parabole těmito objekty určené. Bod M_1 bude jejím ohniskem, přímka t bude řídicí přímkou sestrojované paraboly. Její rovnici a graf získáme příkazem `Parabola [M_1, t]`

(obr. 9). Druhou parabolu určíme obdobně pomocí bodu M_2 a tečny t. Průsečíkem obou parabol je střed (středy) hledané kružnice, kterou následně sestrojíme. Hledané středy kružnic budou pochopitelně ležet i na ose úsečky M_1M_2, ale tu jsme při našem řešení nepoužili.



Obr. 9 Apollóniova úloha „bbp“

Příklad 6 – *Optimalizační úloha*

Nakonec si ukažme, že prostředí programu GeoGebra lze úspěšně využít také k řešení úloh, které mezi ostatní zdánlivě nezapadají. Kupříkladu pro jednoduché optimalizační slovní úlohy.

Zadání: Mléko z mlékárenských závodů ve městech A a B se vozí také do měst R, S a T. Denně se může z A dodat 250 přepravek s mlékem a z B 350 přepravek. Je třeba dodat denně 150 přepravek do R, 240 přepravek do S a 210 přepravek do T. Nalezněte způsob rozvozu, při kterém budou náklady na každodenní přepravu mléka nejnižší. Náklady na dopravu jedné přepravky z mlékárenských závodů do místa prodeje jsou v tabulce 1.

cena v eurech	**R**	**S**	**T**
A	4	3	5
B	5	6	4

Tab. 1

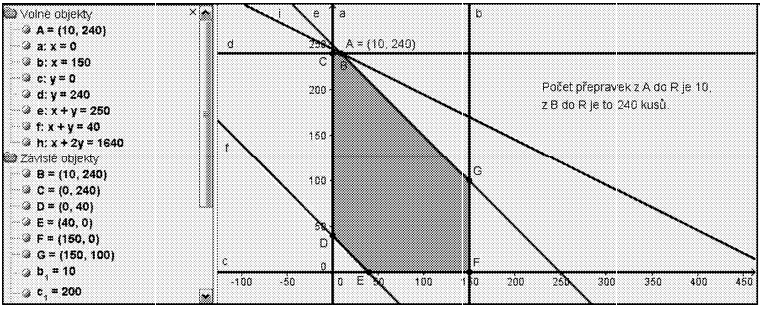
Řešení: V zásadě zde jde o nalezení extrému (minima) jisté funkce, která popisuje náklady na rozvoz přepravek. Pro přehlednost zaveďme tabulku

2 s údaji o počtu přepravek dovezených z jednotlivých podniků do jednotlivých cílových měst (počet přepravek převezených z A do R označíme neznámou x, z A do S neznámou y, ostatní hodnoty dopočítáme podle údajů ze zadání):

připravené přepravky	**R**	**S**	**T**
A	x	y	$250 - x - y$
B	$150 - x$	$240 - y$	$10 - (250 - x - y) = x + y - 40$

Tab. 2

Je zřejmé, že funkce, jejíž extrém chceme znát, bude dána předpisem $f(x, y) = 4x + 3y + 5(250 - x - y) + 5(150 - x) + 6(240 - y) + 4(x + y - 40)$ a budeme-li znát množství přepravek dovezené z A do R a do S (tedy hodnoty x a y), dokážeme vypočítat i celkové náklady. Dále pak platí, že tento extrém hledáme v určité omezené oblasti, která je ohraničena podmínkami vycházejícími z druhé tabulky: $0 \leq x \leq 150$, $0 \leq y \leq 240$, $x + y \leq 250$ a $x + y \geq 40$. Naším úkolem je tedy nalézt minimum funkce dvou přirozených proměnných (předpokládáme, že x a y mohou nabývat pouze přirozených hodnot) na omezeném definičním oboru. To bývá poněkud obtížné a zdlouhavé, můžeme se tomu však vyhnout, realizujeme-li grafické řešení. Pokud roznásobíme závorky na pravé straně předpisu funkce $f(x, y)$ a příslušné neznámé sečteme, dostaneme rovnici přímky, v našem případě $f(x, y) = -2x - 4y + 3280$, přičemž neznámé x a y představují souřadnice jistého bodu a hodnota $f(x, y)$ je určité zatím neznámé „posunutí“ této přímky, které chceme mít co nejmenší. Prostřednictvím příkazového řádku zadáme všechny omezující podmínky (program bohužel nezvládá zadávání a zobrazování polorovin, musíme se tedy spokojit pouze s hraničními přímkami, které zapíšeme příkazy `x = 0`, `x = 150`, `x + y = = 250`, atd.), sestrojíme jejich průnik a dostaneme mnohoúhelník, který zvýrazníme příkazem `Mnohouhelnik [B, C, D, E, F, G]` (body $B, C, \ldots, G$ jsou po řadě průsečíky výše zadaných omezujících přímek). Onen bod s hledanými souřadnicemi x a y se může nacházet pouze v této oblasti a je jasné, že extrémy funkce $f(x, y)$ se budou nacházet v některém z vrcholů mnohoúhelníku, které mají naštěstí celočíselné souřadnice. Kdyby tomu tak nebylo, způsobilo by to menší komplikace při interpretaci výsledku.



Obr. 10 Částečně grafické řešení slovní úlohy

Zbytek práce je již velmi snadný. Nejprve zavedeme přímku `h: -2x-4y+3280=0`, jejíž rovnice vychází z předpisu $f(x, y)$, libovolný bod A, jímž vedeme rovnoběžku s přímkou h (příkaz `i = primka [A, h]`), a zavedeme též vzorec pro výpočet nákladů, v němž ale nahradíme proměnné x a y souřadnicemi bodu A (příkaz `cena = -2x(A)-4y(A)+3280`). Pohybujeme-li s bodem A, vidíme, že se v algebraické části přepisuje údaj cena. Nyní již zbývá jen zjistit, ve kterém vrcholu mnohoúhelníku je tato hodnota nejmenší a které souřadnice bodu A tedy budou odpovídat hledaným neznámým x a y. Počty přepravek na ostatních trasách pak dopočítáme podle druhé tabulky, přičemž však musíme dávat pozor při zjišťování těchto hodnot, neboť pracujeme s počty přepravek, neboli celými čísly, zatímco GeoGebra určuje souřadnice bodů s přesností až na 15 desetinných míst. Je tedy nutné správně zaokrouhlovat.

Pro úplnost ještě vypišme zjištěné výsledky příkladu do tabulky 3.

připravené přepravky	**R**	**S**	**T**
A	10	240	0
B	140	0	210

Tab. 3

Příklady na internetu

Všechny příklady, které jsme zde představili, si čtenáři mohou stáhnout na adrese

http://www.geogebra.org/en/upload/index.php?direction0&order&
v sekci Czech nebo prostudovat v podobě dynamických pracovních listů na Cabri portálu Jihočeské univerzity na adrese
http://www.pf.jcu.cz/cabri/temata/geogebra1/index.htm

Závěr

V tomto článku jsme se snažili ukázat možnosti použití programu GeoGebra na podporu výuky matematiky, které jsou díky jeho algebraickému oknu bohatší než možnosti kteréhokoli z nám známých programů dynamické geometrie. Jeho velká síla je kromě klasické planimetrie i v analytické geometrii, v práci s kuželosečkami, s funkcemi, jak jsme předvedli v šesti uvedených řešených příkladech. Pro čistě geometrické programy tedy představuje zdatného konkurenta.

Nechceme však v žádném případě odmítat jiné programy dynamické geometrie používané ve škole, pouze k nim nabízíme vhodnou alternativu.

Program je velmi dobře využitelný při výuce matematiky na střední škole, jak ověřil ve všech ročnících druhý z autorů při výuce matematiky na SŠAK (specializované na aplikovanou kybernetiku). Žáci, když viděli předvedení programu GeoGebra při hodině, sami požadovali bližší informace o tomto programu, nainstalovali si ho a dnes tento program už běžně (nejen) při hodinách matematiky a při přípravě na tyto hodiny s oblibou používají.

Literatura byla uvedena za 1. částí článku.

Informace k semináři **Matematika, fyzika a podpora jejich výuky** (viz 1. str. obálky)

Seminář navazuje na dlouholetou tradici seminářů o filosofických otázkách matematiky a fyziky. Účastníkům nabízí zajímavé přednášky, ale i zamyšlení nad výukou a motivací žáků ke studiu matematiky a fyziky.

Výběr připravovaných přednášek: *J. Podolský*: Stacionární vesmír versus velký třesk, *J. Langer*: Éter a speciální teorie relativity, *P. Cejnar*: Vypouštění kvantového džina, *J. Šimša*: Výuka mocnin a logaritmů na SŠ, *D. Hrubý*: Jak jsem se seznámil s komplexními čísly, *D. Martišek*: Otazníky středoškolské informatiky

Seminář se bude konat v aule Gymnázia Velké Meziříčí. Ubytování je zajištěno v Domově mládeže Střední školy řemesel a služeb Velké Meziříčí. **Vložné účastníci platit nebudou**, uhradí si jen ubytování (přibližně 200 Kč za noc) a dopravu.

Pro účastníky bude vydána předseminární brožura s podrobným programem. Jako seminární materiál se připravuje: Sborník ze XIV. semináře o filosofických otázkách matematiky a fyziky.
