

KORESPONDENČNÍ SEMINÁŘ Z INFORMATIKY

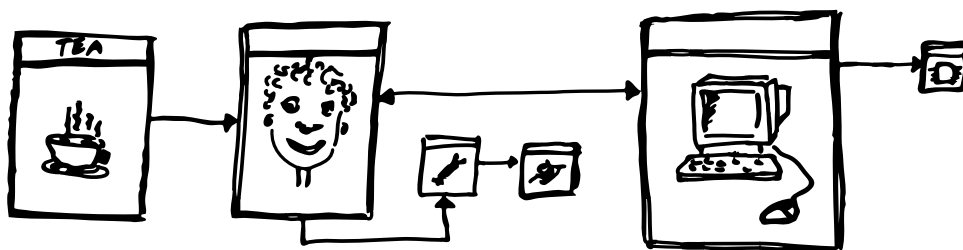
Milé řešitelky a milí řešitelé,

udeřila hodina há a poslední sada letošního ročníku je tu. Seznámíte se s principy, které na vás budou v profesním životě programátora číhat téměř na každém rohu. Naučíte se zcela novému pohledu na svět, navrhovat a vylepšovat informatické systémy a zaznamenat strukturu i vztahy vašeho systému pomocí mezinárodně používaných grafů a značek. Nebudu vás už dále napínat, v této sadě se naučíte objektivě orientovanému programování.

Objektově orientované programování

Objektově orientované programování (dále už jen OOP) je na světě asi nejrozšířenější programovací paradigma. Používají jej například jazyky jako Java, C# a C++. Základní myšlenkou této filozofie je abstraktní pojem *objekt*, který má nějaká data a k nim příslušné funkce, kterým říkáme metody. Čistá data jsou programátorovým očím většinou skryta avšak může se s nimi nepřímo manipulovat pomocí metod, které tedy definují určité rozhraní, jak s objektem zacházet.

Pokud bych tento přístup měl srovnat s procedurálním přístupem, který jste běžně používali, aniž byste to věděli, tak procedurální programování se snaží rozbít úkol programátora na jednotlivé proměnné, struktury dat a na funkce. Čistá data jsou tedy „vidět“ a existují hned vedle funkcí. OOP funkce i data zabalí do objektu a tento objekt poté existuje jako soběstačná datová struktura.



Objekt a třída

Základním prvkem OOP je objekt. Objekt je konkrétní instancí dané třídy. Třída je šablona pro vytváření objektů.

Je mi jasné že z předchozího jste toho moc nepochytili, takže se to pokusím vysvětlit na skutečném příkladu. V dědečkově garáži spinkají dvě krásná auta značky Ford Mustang. Sousedka má jen starý traktor značky Zetor. Počkat, a kde nám v tom příkladu vystupují objekty a třídy? Objekty jsou zde dědečkova auta a sousedčin traktor (samozřejmě, že objekty jsou i dědeček, babička i garáž, ale ty nás teď nezajímají). Dědečkova auta jsou třídy Ford Mustang a traktor zase třídy Zetor. Třidu si představíme jako jakousi výrobní tovární linku, která vyrábí objekty. Je jasné, že výrobní linka na traktory nemůže zničehonic vyrobit Mustanga. Dědečkovy dva Mustangy jsou tedy ze stejné výrobní linky, a přesto mezi nimi musíme rozlišovat. Když dědeček řídí prvního mustanga, tak určitě neřídí druhého. Jednoho může nabourat a to druhé bude pořád v celku.

Unified Modeling Language (UML)

Aby v tom všem nevznikl zmatek, tak na světě existuje jazyk *UML* (Unified Modeling Language), který pomocí diagramů graficky zaznamenává jak celý systém vypadá. UML specifikace zahrnuje spoustu druhů diagramů, které slouží k zaznamenávání celého návrhového procesu

programu, od Use Case Diagramů přes Package Diagramy až po Class Diagramy. My v této sadě budeme používat právě ty posledně zmiňované.

Do Class Diagramu zakreslujeme pouze třídy, jejich vlastnosti a vztahy mezi nimi, ke kterým se dostaneme později. Třída se znázorňuje obdélníčkem, ve kterém je napsané jméno třídy.

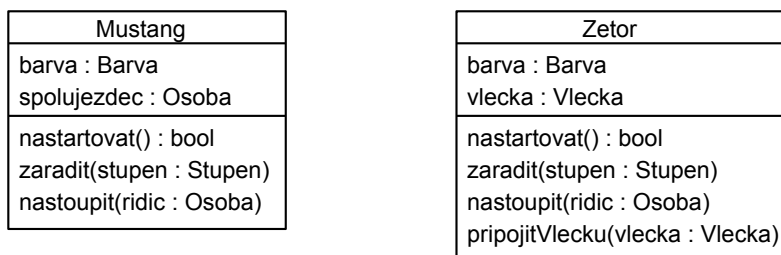


Obr. 1: Třídy Mustang a Zetor

Atributy a metody

Každá třída má své vlastní atributy. Atributy jsou vlastnosti, kterým umíme přiřadit hodnotu. Jednotlivé instance objektů stejné třídy se mohou odlišovat rozdílnými hodnotami atributů. Každá jiná třída se od druhé liší názvem, počtem nebo typem atributu. Např. každý Mustang i Zetor má nějakou barvu. Zetor, protože je traktor, má na rozdíl od Mustanga i vlečku. Mustang má oproti traktoru místo pro spolujezdce. Jeden dědeček Mustang v garáži je zelený, druhý je fialový, ale starosta má červeného.

Další nepostradatelnou vlastnost, kterou by objekt měl mít, je schopnost něco dělat. Představme si situaci kdy máme černou skříňku, která má nějaký vnitřní stav (hodnoty svých atributů) a my chceme, aby provedla nějakou činnost, která na základě svého vnitřního stavu něco vykonala a popřípadě změnila vnitřní stav objektu. Na našem příkladu s mustangy bychom mohli mít metody nastartovat, zařadit nebo nastoupit. Zetor by potom mohl mít metody nastoupit, zařadit, nastartovat a připojitVlečku.



Obr. 2: Zápis atributů a metod

V UML se atributy zapisují do již existujících tříd a to tak, že se název třídy oddělí čarou a pod sebe se píší názvy jednotlivých atributů. Typ atributu se v UML značí tak, že za název napíšeme dvojtečku a poté název typu. Metody se zase oddělí čarou od atributů a píšou se jednotlivé názvy. Návrátový typ metody se zapisuje stejně jako typ atributu. Jednotlivé argumenty metod se oddělují čárkami a píší se do závorek za název metody a před návratový typ v podobné formě jako atributy, tedy názevArgumentu : TypArgumentu.

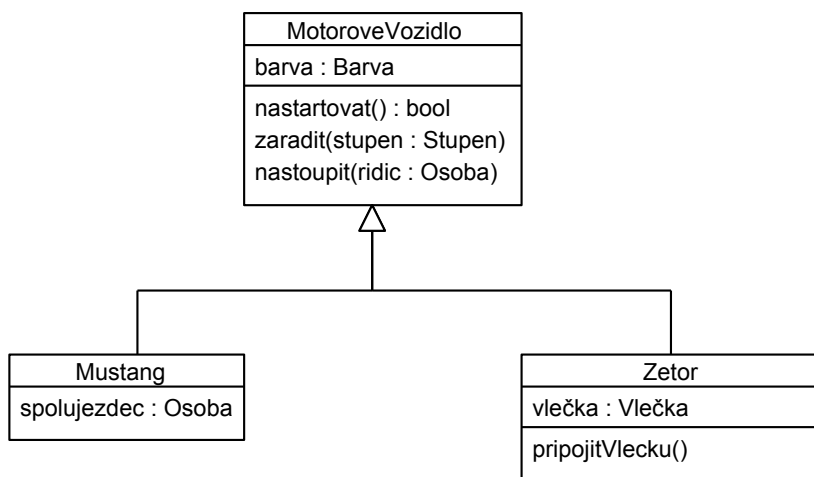
Pokud si výše uvedené pojmy převedeme do analogie procedurálního programování, tak atributy jsou něco jako proměnné, metody jako funkce a třídy jako jednotlivé typy.

Dědičnost

Určitě jste si všimli, že v našem návrhu s traktory a mustangy se spousty věcí opakuje. Jakkpak by ne, oboji jsou přeci motorizovaná vozidla a víme, že všechna motorová vozidla mají některé atributy a některé metody stejné. Každý vůz určitě bude mít nějakou barvu. Můžeme jej nastartovat, nastoupit do něj, nebo zařadit rychlost.

Pokud najdeme nějaké společné vlastnosti mezi třídami, tak je vždy dobré tyto vlastnosti určit, přesunout je do jiné třídy a říct, že tyto třídy *dědí* z právě vytvořené třídy. V praxi se snažíme v hierarchii přesunout co nejvíce věcí dolů a zároveň nepřidat žádnou vlastnost, která by byla ve třídě navíc.

Dědičnost v UML značíme šipkou s prázdnou hlavou, která vede ze třídy která dědí, do rodiče. Znamená to, že všechny atributy a všechny metody ze třídy rodiče se „přesunou“ do potomka, tj. funkcionality potomka je minimálně stejná jako rodiče.



Obr. 3: Dědičnost

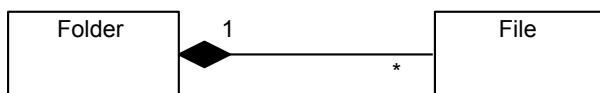
Dědičnost umožňuje znovuvyužitelnost kódu – nabalujeme stále více kódu „na sebe“. Dovoluje některé metody ponechat a některé změnit. Některé programovací jazyky povolují v krajních případech použití vícenásobné dědičnosti, kdy třída může dědit z více rodičů (např.: třída Netopýr může dědit ze třídy Savec a OkřídlenéZvíře), ale tato praktika by se měla využívat opravdu jen v krajních případech. S vícenásobnou dědičností se zatím nesetkáte, protože s sebou přináší spousty problémů, nicméně je dobré o ni vědět. Ve většině případů si vystačíte s jednonásobnou dědičností.

Asociace a kompozice

Asociace je nějaký vztah mezi jednotlivými třídami, který nám říká jestli a jak jsou jednotlivé objekty mezi sebou provázány (i mezi jinými třídami). Různých druhů asociací je mnoho, my se naučíme rozlišovat pouze kompozici

Kompozice je binární relace (vždy mezi dvěma objekty), která nám říká, že jeden objekt obsahuje druhý a životnost dílčího objektu je úzce spjata s životností nadřazeného, tj. pokud zanikne nadřazený objekt (když je mazán z paměti), tak zároveň s ním zaniknou i objekty, které obsahoval.

S kompozicí jsme se už vlastně setkali ve formě atributů. Můžeme říct, že třída Auto se skládá z barvy, motoru, kol, volantů atd. Kompozice¹ se v UML zapisuje pomocí čáry, která vede od dílčí třídy a je zakončena černě vyplněným kosočtvercem u třídy, která obsahuje dílčí třídu. Na oba konce vazeb se poté píše četnost, ve které se příslušná třída může vyskytovat. Četnost * (hvězdička) znamená neomezeně mnoho výskytů (0 a víc); četnost 2 znamená právě dva výskyty; četnost 1..4 – od jednoho do čtyř výskytů; četnost 3..* – 3 a více výskytů. Pokud četnost není uvedena, myslí se tím automaticky vazba 1:1.



Obr. 4: Vztah souboru a složky. Čteme: *jedna složka obsahuje 0 a více souborů a zároveň soubor je maximálně v jedné složce* (nikde není řečeno, že soubor nemůže existovat samostatně bez složky). Pokud složku smažeme, tak tím odstraníme i všechny obsažené soubory

¹Zápisu kompozice se užívá jen tam, kde tento vztah chceme zvýraznit

IS vs HAS

Velice často je pro začátečníky obtížné rozlišovat, kdy mají použít dědičností a kdy kompozicí. Ale nebojte se, na to existuje jednoduchá pomůcka. Dědičnost použijete tam, kde si můžete říct, že třída JE (IS) součástí podřazené třídy. Kompozici použijete zase tam, kde si můžete říct, že třída vlastní (HAS) třídu jinou. Například si zkuste vytvořit diagram pro třídy: růže, trn, kytka, okvěť, kytice, osoba a jméno. Řešení: růže HAS trn; růže IS kytka; růže HAS okvěť; kytice HAS kytka (mnoho – 1 : *); osoba HAS jméno.

Zapouzdření

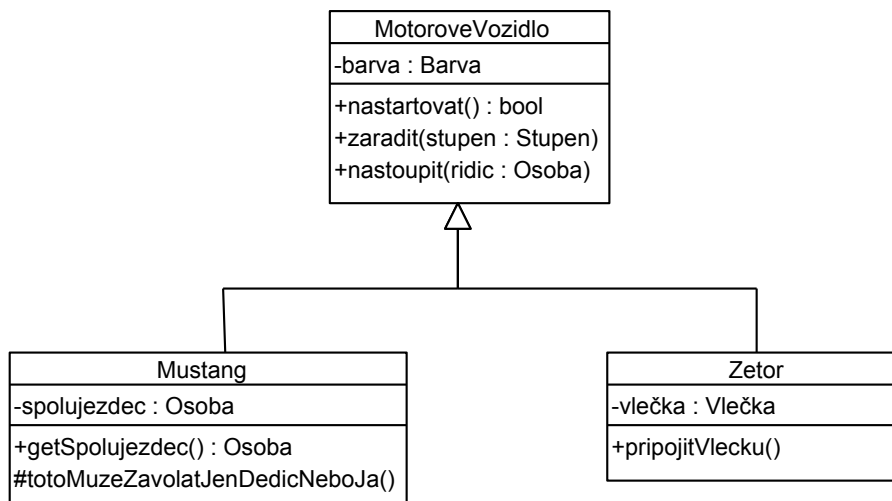
Na začátku jsem říkal, že data objektů (teď už víme, že se jim říká atributy) jsou většinou programátorovi skryta a on s objektem manipuluje pouze pomocí metod. Toho se docílí tak že se každému atributu i každé metodě určí viditelnost. Viditelností je několik druhů, ty nejpoužívanější jsou: *public*, *protected* a *private*.

K metodě nebo atributu deklarovanému jako *public* může přistupovat jakákoliv třída kdykoliv a odkudkoliv. Metody si tedy může volat kdokoli a atributy může kdokoli číst i měnit jak se mu zlíbí. V UML se takový atribut/metoda značí tak, že se před něj umístí symbol + (plus).

K atributu nebo metodě deklarované jako *private* může přistupovat pouze ta třída, které atribut/metoda to je. Ostatní třídy (ani ty které z této třídy dědí) ani neví, že takový atribut existuje. V UML je privátní položka označena symbolem - (minus).

Protected potom deklaruje metodu nebo atribut jako přístupnou pouze pro třídu, které atribut to je, a pro všechny třídy, které z této třídy dědí. Ostatní třídy, které z ní nedědí, položky označené jako *protected* nevidí. V UML tomu odpovídá značka # (mřížka).

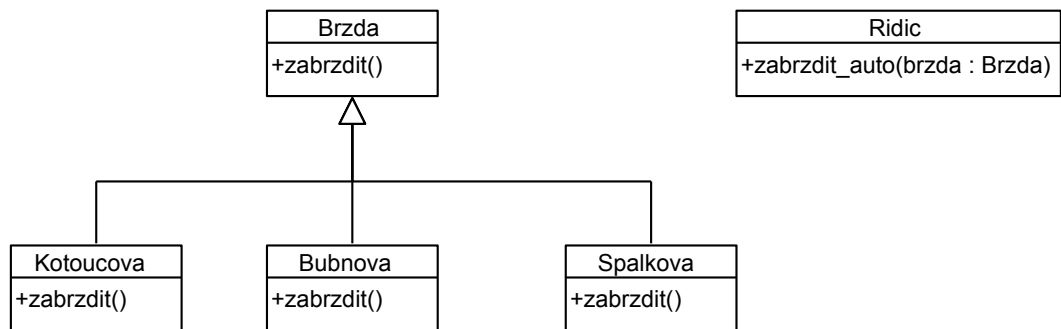
Základem dobrého návrhu je být přísný a vždy poskytovat jenom přístup, který je nezbytně nutný. Dokonce platí pravidlo, že všechny atributy, by měly být *private* a k nim se poté přistupuje přes metody (*public* nebo *protected*), které vrátí hodnotu atributu, nebo jej nastaví. Tím lze určit, kdo daný atribut může číst a kdo měnit. Takovým metodám se často říká gettery nebo settery, protože nedělají nic jiného, než jen vrátí nebo nastaví hodnotu atributu.



Obr. 5: Přidány modifikátory přístupu a příklad *protected* metody

Polymorfismus

Položme si filozofickou otázku. Jak je možné, že umíme v autě brzdit, když existují různé druhy brzd, bez toho, abychom se zvlášť učili brzdit na kotoučové, bubnové nebo na špalíkové brzdě? Odpověď je až nečekaně krátká. Protože mají stejné rozhraní. V autě toto rozhraní je většinou pedál, který když sešlápneme, tak zabrzdíme bez ohledu na to, jak je brzda uvnitř udělaná. V OOP toto chování označujeme pojmem *polymorfismus*, tedy že můžeme s různými třídami zacházet stejně, pokud dědí ze stejné třídy, která jim definuje společné rozhraní.



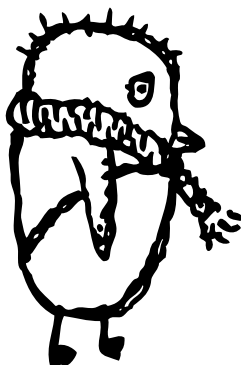
Obr. 6: Polymorfizmus

Třída Brzda (viz obr. 6) je rodič všech možných druhů brzd a definuje jim rozhraní (metoda zabrzdit). Každý druh brzdy poté implementuje svoji vlastní metodu zabrzdit. Polymorfizmus nám potom dovoluje zacházet s různými druhy brzd jako s každou jinou, protože mají stejné rozhraní. Metodě zabrzdit_auto poté stačí předat jakoukoliv třídu, která dědí ze třídy Brzda, a polymorfismus zajistí, že se zavolá správná metoda zabrzdit a vše bude fungovat tak, jak má.

Obecné rady na závěr

Při návrhu se vždy snažíme o nejlehčí systém. Cokoliv, co je v systému navíc, je zbytečné a komplikuje to program. Atributy vždy deklarujeme jako privátní a k nim přistupujeme pomocí getterů a setterů. Metody které volá jenom třída, která je vlastní, jsou vždy privátní resp. protected dle potřeby. Zájemcům určitě doporučuji samostudium, protože v této sadě jste se s OOP setkali jenom povrchně.

Teorie je za námi a mně nezbývá nic jiného, než ti popřát, ať se ti daří při řešení příkladu.



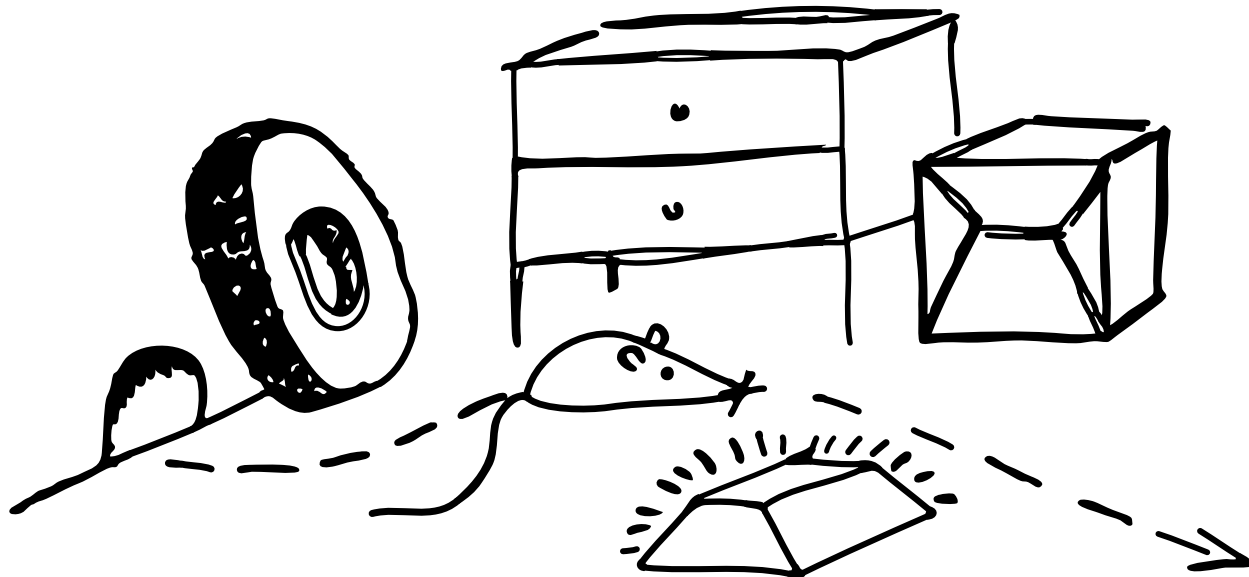
Ξ Zadání 5. sady úloh KSI (termín odevzdání: 14. 4. 2014)

Řešení zasílejte pomocí internetového systému na adrese <http://ksi.fi.muni.cz>.

Příklad 1: Pivnica (10 bodů)

Karlík sa rozhodol po zime skontrolovať pivnicu. Pivnica je o veľkosti 10×10 metrov. V pivnici sa nachádzajú tieto predmety: lopty (stred, priemer, váha, cena/lineárne závislá od priemeru/), skriňa so šuplíkmi (ľavý dolný roh, pravý horný roh, výška, šírka, hĺbka, počet šuplíkov, zoznam vecí v šuplíkoch), kocka zlata (veľkosť strany), kocka striebra (veľkosť strany), kocka polystyrénu (veľkosť strany), palica (dĺžka, šírka, druh dreva), škatule (veľkosť strany, zoznam predmetov, cena predmetov), pneumatiky (priemer, cena /lineárne závislá od priemeru/). Niektoré objekty obsahujú ďalšie objekty. Navrhните štruktúru odvodených tried, ich atribútov a metód (viditeľných navonok/len na vnútorné použitie) tak, aby ste vyriešili tieto úlohy:

- V pivnici býva Karlíkova kamarátka myš, ktorá sa potrebuje pohybovať medzi predmetmi. Navigujte ju po izbe bez toho, aby narazila. Na vstupe dostanete súradnice, kde sa nachádza, zoznam vecí a cieľové súradnice. Na výstupe sa očakáva najkratšia cesta.
- Keďže Karlík má v pivnici aj drahé veci, rozhodol sa, že umiestni alarm. Peňazí však nemá nazvyš (nechce predať cenné veci), preto si kúpil alarm, ktorý ochráni iba veci v obdĺžniku 100×200 cm. Pomôž mu umiestniť alarm čo najefektívnejšie, teda aby ochránil obdĺžnik, ktorý má najväčšiu cenu.
- Kým Karlík rozmýšľal nad alarmom, vonku sa roztopil sneh natoľko, že kanalizácia nestíha a voda sa valí do pivnice cez dvere. Navrhните metódu, ktorá pre zadanú výšku hladiny vráti zoznam objektov, pre ktoré sa ešte Karlík môže vrátiť a odnieť do bezpečia. Pod vodou nie je nič vidieť a Karlík nemá čas šmátrať naslepo pod hladinou, preto aspoň časť objektu musí byť ešte vidieť nad hladinou.



Příklad 2: Kolekce (10 bodů)

Kolekce (někdy též kontejnery) jsou objekty, které seskupují více objektů. Příkladem kolekce z reálného světa je ošatka jablek. Běžnými kolekcemi v informatice jsou zásobník, fronta, seznam, množina a slovník. Nejobecnější kolekcí je tzv. **iterovatelná kolekce** (angl. *Iterable*), která umožňuje provést nějakou specifikovanou operaci na všech jejích prvcích. Všechny následující kolekce jsou speciálním případem iterovatelné kolekce. **Seznam** je kolekce, v níž můžeme k jednotlivým prvkům přistupovat pomocí celočíselného indexu. Můžeme nový prvek kamkoliv přidat, přepsat již existující prvek nebo kterýkoliv prvek odebrat. **Zásobník** je speciální

variantou seznamu – umožňuje prvky vkládat a odebírat pouze z vrcholu zásobníku (princip LIFO: *Last In – First Out*). **Fronta** je kolekce, do které lze též prvky přidávat a odebírat, pořadí odebírání se však řídí principem FIFO (*First In – First Out*), tak jako v klasické frontě v obchodě. **Prioritní fronta** je podobná frontě, ale pořadí odebírání není určeno pořadím přidání, ale hodnotou nějakého klíče (priority). V **množině** nemají jednotlivé prvky své indexy, ale můžeme do ní prvky přidávat i odebírat a testovat, zda je nějaký prvek v množině. Základní varianta množiny nemůže obsahovat jeden prvek vícekrát (když do ní třikrát za sebou přidáme číslo 4, pořadí ho bude obsahovat pouze jedenkrát). Přestože nemůžeme přistupovat k jednotlivým prvkům pomocí indexů, můžeme projít celou množinu (a provést nějakou operaci pro každý prvek množiny). Existují dvě podobné kolekce: **multimnožiny**, které mohou obsahovat jeden prvek vícekrát a **uspořádané množiny**, které udržují prvky v nějakém pořadí, které si zvolíme (např. řetězce můžeme řadit alfanumericky nebo třeba vzestupně podle jejich délky). **Slovník** (asociativní pole) mapuje klíče na hodnoty. Je podobný seznamu, ale jeho prvky nejsou indexovány pomocí posloupnosti celých čísel, ale pomocí klíčů (klíčem může být např. číslo, řetězec nebo i složitější objekt). Ve slovnících můžeme iterovat přes množinu všech jejich klíčů. Existují také **uspořádané slovníky**, které umožňují procházet uložené položky ve vzestupném pořadí podle hodnoty klíče, a **multislovníky** (vícenásobné mapy), které umožňují mapovat jeden klíč na více hodnot.

Zatím jsme se bavili o kolekcích jako o *abstraktních datových typech*, tj. typech definovaných svými operacemi (někdy též mluvíme o rozhraní nebo specifikaci). Tento pohled je užitečný při návrhu programu, ale z hlediska efektivity může být otázka života a smrti (resp. otázka vteřin a hodin), kterou implementaci kolekce zvolíme. Každá z uvedených kolekcí přitom může být implementována různě, např. seznam může být implementován jako dynamické pole nebo spojovaný seznam. Dynamické pole bude efektivnější, pokud chci často přistupovat k jednotlivým prvkům na různých indexech. Spojovaný seznam bude lepší volbou tehdy, pokud výrazně převažuje přidávání a ubírání prvků, příp. pokud čteme pouze ze začátku seznamu.

OOP nám však umožňuje psát metody pracující třeba s libovolným seznamem, nezávisle na tom, jak je uvnitř implementovaný.

Vášim úkolem bude určit a zdůvodnit, kterou z výše uvedených kolekcí byste použili v následujících případech. Žádný kód uvádět nemusíte. Předpokládejte, že máte k dispozici iterovatelnou kolekci záznamů o účastnících KSI (pro jednoduchost bude mít každý záznam pouze 5 položek: id, jméno, příjmení, škola, celkový počet bodů).

1. Každý měsíc chceme náhodně vybrat jednoho řešitele, kterému společně s řešením pošleme malý dárek – plyšové logo KSI!
2. Na školy našich řešitelů hodláme zaslat propagační materiály, ale pouze jedny na každou školu. Potřebujeme tedy seznam všech škol našich řešitelů bez duplicit.
3. Na školy našich řešitelů zasíláme začátkem roku tolik výtisků první sady, kolik studentů z dané školy KSI řešilo loni. Potřebujeme tedy seznam všech škol našich řešitelů společně s počty řešitelů na dané škole.
4. Každý měsíc chceme náhodně vybrat jednu školu, na kterou pošleme plyšové logo KSI. Pravděpodobnost, že školu vybereme, by měla být přímo úměrná počtu jejích studentů řešících KSI.
5. Chceme vybrat vhodné jméno pro našeho nového maskota. Budeme ho vybírat z křestních jmen našich účastníků a tak bychom potřebovali jejich seznam, seřazený podle abecedy a bez duplicit.

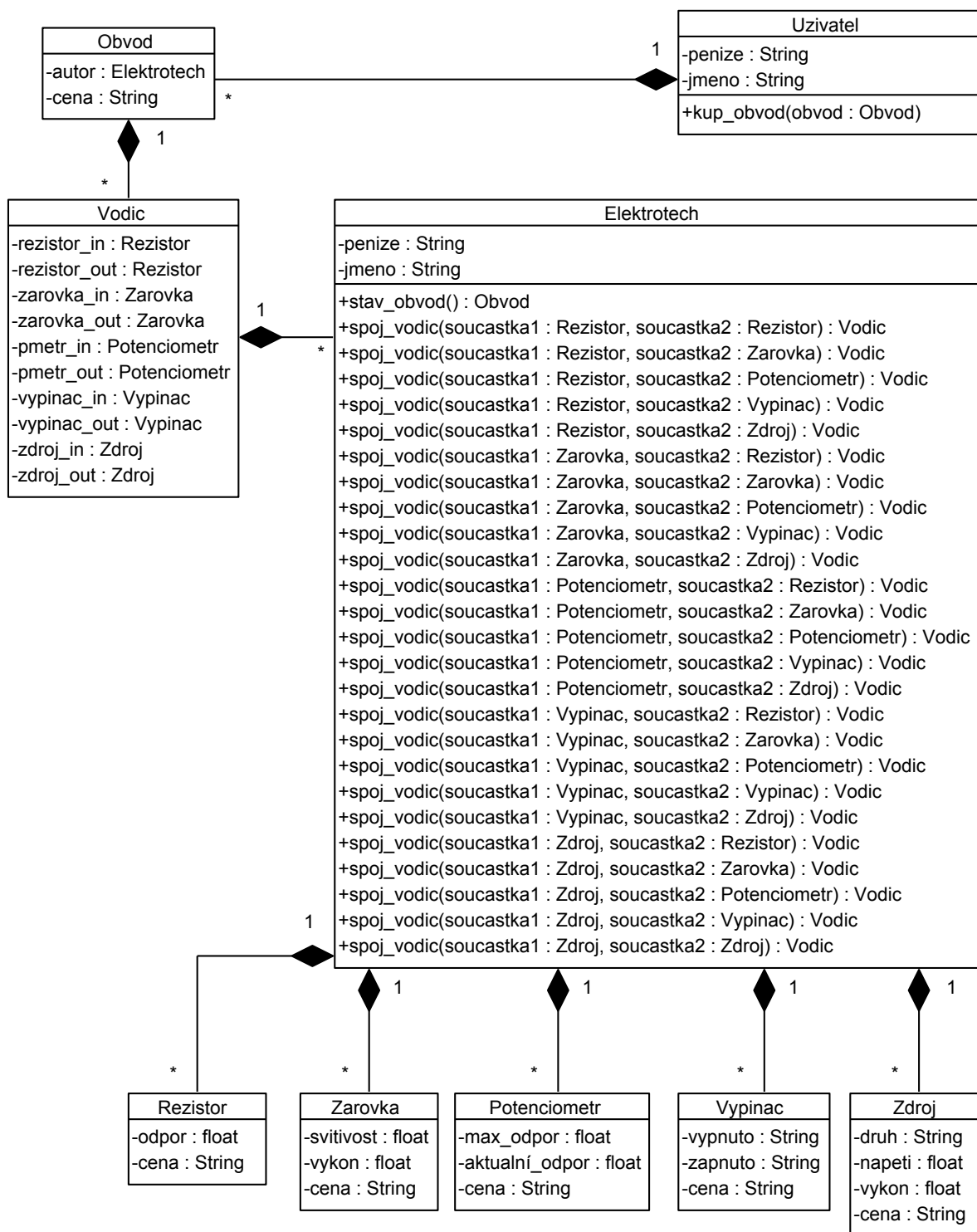
Příklad 3: Rozprávka (10 bodů)

Jednoho dne tučňákovi Karlíkovi volala jeho tučňáčí kamarádka Evka telefonem. Ukázalo se, že má problém se svým počítačem a žádá Karlíka o pomoc. Karlík jako dobrý kamarád samozřejmě souhlasil. Karlíkovi rodiče ho ale jen tak nepustili a na cestu mu zabalili tuňáka, kdyby mu vyhládlo. A tak se vydal z domovních dveří na cestu. Venku se pořádně chumelilo, i když byl Karlík zachumlaný v kožešinovém kabátu, tak mu byla čím dál tím větší zima. I přesto

se vytrvale blížil k Evčinu domu. Po dlouhé cestě konečně dorazil na místo a zaklepal na dveře. Evka mu otevřela a když viděla, jak je promrznutý, hned mu dala napít horkého čaje. Karlík došel do Evčina pokoje, kde stál ztichlý počítač. Prohlédl si ho, zamyslel se, vypořel ze zásuvky, zase zapojil a zapnul. Počítač se šťastně rozbzučel a pohádka tak šťastně skončila

Určitě se ptáte, proč jsme vám tuto krásnou pohádku vyprávěli. Vaší úlohou bude reprezentovat vztahy v pohádce grafem. Znázorněte vztahy IS/HAS (je/má) mezi jednotlivými postavami a předměty a dopište jejich schopnosti (jako metody nad objekty). Samozřejmě bude hodnocena efektivita návrhu, takže se vyvarujte duplicitám a zbytečností.

Příklad 4: Elektronické bastlení (10 bodů)



Obr. 7: Zadání k příkladu 4: Elektronické bastlení

Karlík si rád staví všelijaké elektrické obvody. Rozhodl se na nich vydělat pár korun a začal tím, že si vytvořil jednoduché schéma. Karlík (patřící do třídy *Elektrotech*) si koupí maximálně tolik součástek *Rezistor*, *Zarovka*, *Potenciometr*, *Vypinac* a *Zdroj*, kolik má peněz. Vodičů má doma spoustu, takže objekty třídy *Vodic* jsou zdarma. Součástky pospojuje pomocí metody `spoj_vodic()` a pak vytvoří objekt *Obvod* pomocí metody `stav_obvod()`. `spoj_vodic()` je přetížená metoda¹, jako argumenty bere dvě libovolné součástky a přidá do pole `vodice` vodič propojující tyto dvě součástky. Metoda `stav_obvod()` vytvoří objekt *Obvod* z pole vodičů, jména autora a spočítá jeho cenu. Cena obvodu je součet cen všech použitých součástek vynásobená koeficientem 1,2, aby z toho Karlík taky něco měl. *Uzivatel* si může *Obvod* koupit pomocí metody `kup_obvod()` pouze tehdy, má-li dost peněz. Argumentem metody `kup_obvod()` je objekt *Obvod*. Všechny vztahy v Karlíkově schématu jsou typu HAS.

Myslíte si, že za vás Karlík už udělal všechnu práci? Opak je pravdou. I když jeho systém funguje, je zbytečně složitý a vy ho tak budete muset opravit postupem známým jako *refaktoring* (<http://cs.wikipedia.org/wiki/Refaktorov%C3%A1n%C3%AD>). Ten spočítá v provádění jednoduchých úprav, které zjednodušují a zefektivňují návrh programu. Hlavní (ale zdaleka ne jedinou) věcí, kterou po vás budeme chtít, je úplné odstranění třídy *Vodic*. Cílem vašeho refaktorování by také mělo být odstranění duplicit. Podstatné je, aby v upraveném schématu fungovalo všechno, co funguje nyní. Vámi odevzdané řešení by mělo mít formát UML diagramu podobnému tomu ze zadání.

Příklad 5: Body, hejbejte se (10 bodů)

V této úloze budete sestavovat *rovnostanné* trojúhelníky následujícím způsobem: Máte zadáno n bodů ($n \geq 3$), každý bod se si náhodně vybere dva další různé body, se kterými se bude snažit vytvořit rovnostanný trojúhelník (toto neprogramujete, to se prostě stane). Bod je instance třídy *Bod*, která vypadá takto:

```
Class Bod {
    private Bod bod1; // první nahodne vybrany bod
    private Bod bod2; // druhy nahodne vybrany bod

    // otoci bod o zadaný pocet stupnu (kladne cislo - po smeru hodinovych
    // rucicek; zaporné cislo - proti smeru hodinovych rucicek)
    private void otocSe(int pocetStupnu);

    // otoci bod primo k zadanemu bodu
    private void otocSe(Bod bod);

    // vrati vzdalenost dvou zadanych bodu;
    // pokud jeden parametr vynechate, pouzije se misto nej aktualni bod
    private double zjistivZdalenostDvouBodu(Bod bod1, Bod bod2);

    // udela krok velikosti 1 tim smerem, kterym je bod otoceny
    private void udelejJedenKrok();

    // tuto funkci mate za ukol naprogramovat
    public void pohniSe();

    // tato funkce vraci true/false podle toho,
    // jestli tvori s dalsimi dvema body rovnostanny trojuhelnik
    public bool jsiSpokojeny();
}
```

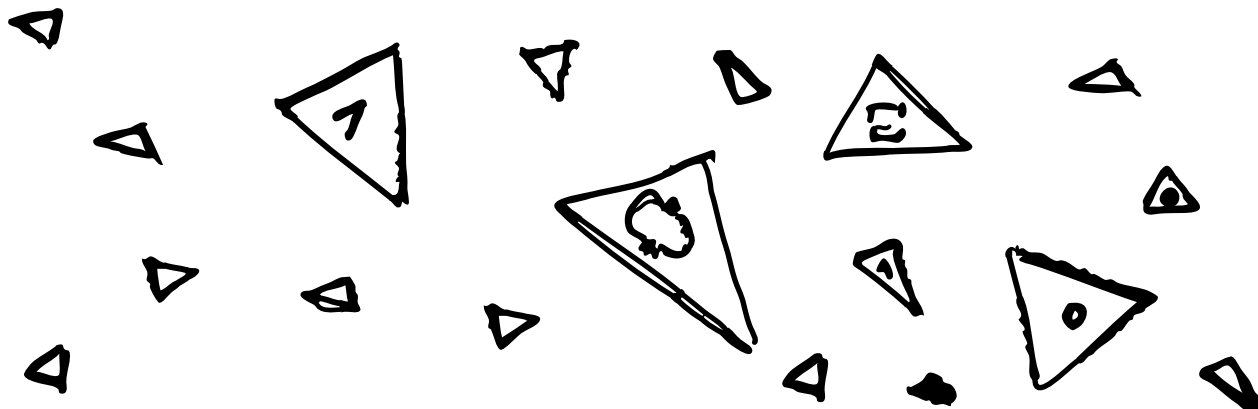
¹Přetížení metod znamená, že máme deklarováno více metod stejného názvu, ale odlišného počtu, typu nebo pořadí argumentů. Při volání funkce pak překladač podle argumentů rozhodne, která metoda se má zavolat.

Vaším úkolem je naprogramovat funkci `pohniSe` tak, aby po spuštění programu vznikly rovnostranné trojúhelníky. Nemáte povoleno přidávat žádné další veřejné metody ani atributy. Každý bod má navíc zakázané volat funkci `pohniSe` jiných bodů a kromě toho může udělat pouze jeden krok v jednom průchodu bodů. Výsledek nemusí být v žádném konkrétním programovacím jazyce, stačí pseudokód. Body se pohybují po neomezeně velké rovině, na které jsou na začátku náhodně rozmístěny (náhodně = tak, že to má řešení).

Kromě funkce `pohniSe` napište obslužný kód, který budou spouštět funkci `pohniSe` pro body a kontrolovat, jestli už je vše jak má. Můžete procházet všechny body, třeba nějak takto:

```
Loop body as bod
  bod.pohniSe();    // zavola funkci pohniSe nad aktualne prochazenym bodem
endOfLoop
```

Takovýto cyklus *Loop* postupně prochází všechny body, a když se dostane na konec, začne procházet znova. Skončí pouze v případě, že zavoláte speciální funkci `break()`. Nápověda: na vyřešení tohoto úkolu není potřeba složitá matematika, snažte se o jednoduché řešení. Nemusí být úplně optimální, důležité je, aby skončilo správně v rozumném čase (není tedy vhodné např. náhodně posouvat body a doufat, že to někdy vyjde :))



A to je z páté sady a zároveň i z celého ročníku KSI vše. Přejeme ti hodně úspěchů při řešení, a když budeš mít jakékoliv otázky, neváhej se na nás obrátit e-mailem na adresu `ksi@fi.muni.cz` nebo v diskuzním fóru na webových stránkách.

Termín odevzdání 5. sady úloh KSI: 14. 4. 2014

<http://ksi.fi.muni.cz>