KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 3 ◇ poznámky k přednášce

4. Dědičnost

verze z 21. října 2020

1 Problémy kódu z minulé přednášky

Na minulé přednášce jsme si všimli problémů obsažených ve zdrojových kódech naší grafické knihovny. Jsou to hlavně tyto tři:

1. Porušujeme základní programátorskou zásadu **nikdy nenechávat ve zdrojovém textu na více místech stejný nebo podobný kód**. Její dodržování má přitom dobré důvody. Snižuje možnost vnesení chyb do kódu při jeho úpravách a přispívá k udržitelnosti a rozšiřitelnosti programu do budoucna. Nedodržení zásady dříve nebo později povede k nepříjemným chybám. Problém se týká například metod pro vlastnosti `color` a `thickness` ve třídách `point`, `circle` a `polygon` a metod pro práci s nimi. Také definice příslušných slotů se ve třídách opakují. Dále jde například o metody `draw`, které jsou ve všech třídách grafických objektů kromě třídy `picture` totožné, a metody `set-mg-params`, které jsou podobné. Daly by se najít další příklady (například velká podobnost mezi třídami `polygon` a `picture`).

2. V metodě `check-item` třídy `picture` vyjmenováváme třídy, jejichž instance mohou být prvky obrázků. Tento seznam ale bude nutné aktualizovat, kdykoli definujeme novou třídu grafických objektů, na což může programátor snadno zapomenout.

3. V příkladě jsme si ukázali, jak vytvořit obrázek terče jako instanci třídy `picture`. S terčem můžeme různě manipulovat pomocí obecných zpráv (třeba geometrických transformací `move`, `scale`, `rotate`), nemáme ale jak definovat operace specifické pro terče, například zjištění a nastavení počtu kroužků nebo změnu používané dvojice barev. Terče je užitečné chápat jako zvláštní typ (třídu), nemáme ale nástroje, které by nám je takto umožnily definovat a přitom nepřijít o to, že jde současně o instance třídy `picture`.

V tomto předmětu se zabýváme **principy** programování. Proto se pojďme nejprve zamyslet, v čem je podstata uvedených problémů. Podíváme se na první dva, řešení třetího pak přijde samo.

První dva problémy jsou způsobeny tím, že **třídy grafických objektů mají něco společného** a my to nereflektujeme v našem zdrojovém kódu. Například všechny grafické objekty s výjimkou obrázků mají barvu (vlastnost `color`) a s ní souvisejícím slotem a metodami). Opakovaný kód vzniká tím, že tyto společné

prvky implementujeme u každé třídy zvlášť. Druhý problém je jiným projevem téže chyby: prvky seznamu `items` obrázku mohou být jen grafické objekty, tedy ty objekty, které mají to, co je společné všem třídám grafických objektů.

Chceme-li charakterizovat onu společnou část, můžeme říci, že grafické objekty jsou všechny **stejného typu**: jsou to právě a jen grafické objekty. Na rozdíl třeba od oken (instancí třídy `window`) mají společné to, co všechny grafické objekty sdílejí: barvu, tloušťku pera (tady jsou obrázky zajímavou výjimkou), schopnost nakreslit se do okna. V objektově orientovaném programování definujeme takový typ jako **třídu**.

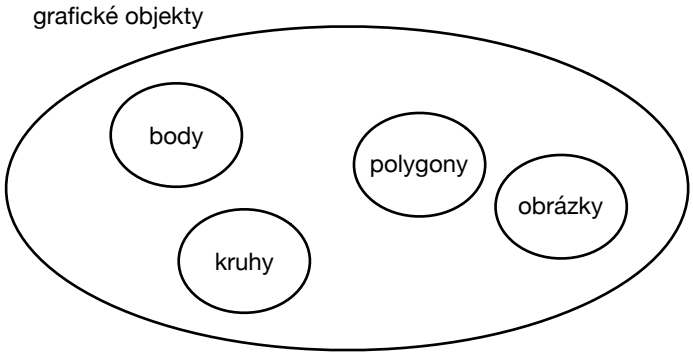
První problém pak vyřešíme tím, že společné rysy různých tříd grafických objektů přesuneme do této nové obecnější třídy. Co se týká druhého problému; s tím si poradíme tak, že místo požadavku, aby prvek obrázku byl bodem, kruhem, polygonem nebo obrázkem, budeme požadovat prostě aby byl grafickým objektem. Třetí problém vyřešíme v příkladu, až na něj přijde řada.

2 Princip dědičnosti

Objekty lze někdy účelně rozdělit podle jejich typu do skupin tak, že dvě různé skupiny buď nemají společný prvek (jsou disjunktní), nebo je jedna úplně obsažena ve druhé (je její podmnožinou).

Příklad: množiny grafických objektů

Takové rozdělení je možné zavést na grafických objektech, jak ukazuje Obrázek 1. Z množin všech bodů, kruhů, polygonů a obrázků jsou každé dvě disjunktní.



Obrázek 1: Množiny grafických objektů

Jejich společnou nadmnožinou je množina všech grafických objektů. Tu můžeme charakterizovat jako množinu všech našich objektů, které se umí vykreslit do okna (rozumí zprávě `draw`). Většina z nich (s výjimkou obrázků případně bodů) má také základní vlastnosti používané při kreslení: barvu, tloušťku čáry, vlastnost `filledp`. Mimo tuto nadmnožinu samozřejmě existují další objekty, které nejsou grafickými objekty. V našem malém systému jsou to okna.

Takové rozdělení můžeme chápat jako nový typ **datové abstrakce**: přechod od chápání konkrétního objektu jako kruhu k jeho chápání jako obecného grafického objektu obnáší přestat brát v úvahu jeho vlastnosti, které se vztahují konkrétně ke kruhům, tj. středu a poloměru, a uvažovat pouze o vlastnostech charakteristických pro grafické objekty obecně.

Tyto možnosti obecných datových typů zatím nemůžeme využít u tříd. Každá třída je datovým typem (určuje množinu objektů), ale neznáme způsob, jak definovat třídy hierarchicky, tedy aby jedna byla **podtypem** druhé (tedy aby jí určená množina objektů byla podmnožinou množiny určené druhou třídou). Naše znalosti nám zatím dovolovaly definovat třídy pouze tak, že žádný objekt nepatřil dvěma různým třídám (například množina všech kruhů a množina všech bodů nemají žádný společný prvek). Objektově orientované programování umožňuje také mezi třídami zavést hierarchický vztah, který jsme si před chvílí ukázali. Ke třídě C umožňuje definovat třídu D, která je jejím **podtypem**, tedy takovou, že každý objekt třídy D je i třídy C.

Taková třída D se pak nazývá *potomkem třídy* C a třída C *předkem třídy* D. Pokud mezi nimi není další třída, mluvíme o *přímém potomku* a *přímém předku*. Možnosti vytvářet ke třídám potomky říkáme **princip dědičnosti**.

Třídy jsou tedy uspořádány do stromové hierarchie podle vztahu předek–potomek (*ancestor–descendant*, *predecessor–successor*, nadtřída–podtřída). Hierarchie se nazývá *strom dědičnosti*.

Příklad: třída `shape`

Uspořádat třídy do stromové hierarchie se hodí právě kvůli před chvílí popsané datové abstrakci. Nyní definujeme třídu `shape`, která bude obsahovat všechny grafické objekty a jejímiž potomky budou konkrétní třídy grafických objektů jako (mírně upravené) `point` nebo `circle` (ty definujeme později). To nám pomůže vyřešit problémy uvedené na začátku, jak snadno pochopíte hned z této přednášky nebo později ze zdrojového kódu.

Co se týká vlastností, které bude vhodné do třídy `shape` přesunout, už jsme řekli, že by to měly být vlastnosti `color`, `thickness` a `filledp`. O všech se totiž v principu dá říci, že jsou vlastnostmi všech grafických objektů bez ohledu na jejich typ a že souvisejí s tím, co mají grafické objekty společného: kreslení do okna přes zprávu `draw`.

Rozeberme si to u vlastnosti `color`. Ta slouží k uložení barvy grafických objektů. Jistě má smysl, aby každý grafický objekt nesl informaci o barvě, kterou je kreslen.

Pro přesnost zopakujme: v některých speciálních případech nemusí být informace o barvě grafického objektu využita. To platí zejména pro třídu `picture`. Každý prvek obrázku se kreslí svou vlastní barvou. Kvůli zachování struktury stromu dědičnosti ale tuto výjimku nebudeme brát v úvahu. Pravidlo *is-a* musí dostat přednost před těmito drobnými výhradami.

Z podobných důvodů jako u informace o barvě přesuneme do třídy `shape` i

vlastnosti `thickness` a `filledp`.

```
(defclass shape ()
  ((color :initform :black)
   (thickness :initform 1)
   (filledp :initform nil)))
```

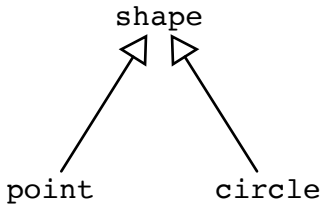
Metody pro vlastnosti:

```
(defmethod color ((shape shape))
  (slot-value shape 'color))

(defmethod set-color ((shape shape) value)
  (setf (slot-value shape 'color) value)
  shape)

(defmethod thickness ((shape shape))
  (slot-value shape 'thickness))

(defmethod set-thickness ((shape shape) value)
  (setf (slot-value shape 'thickness) value)
  shape)

(defmethod filledp ((shape shape))
  (slot-value shape 'filledp))

(defmethod set-filledp ((shape shape) value)
  (setf (slot-value shape 'filledp) value)
  shape)
```

Pozor, případné dřívější definice těchto metod pro jiné třídy zůstávají v platnosti. O tom se lze v prostředí LispWorks přesvědčit například pomocí nástroje „Generic Function Browser“, který umí ukázat všechny metody dané zprávy (generické funkce). Tam lze také nechtěné metody oddefinovat. Ke zrušení definice čehokoli se můžeme také dostat přes kontextovou nabídku definice v editoru.

Pokud zatím zůstaneme u uvedených tří tříd, bude strom dědičnosti vypadat jako na Obrázku 2.

Nyní rozšíříme pojem instance třídy z minulých přednášek. Pokud je objekt instancí některé třídy, bude současně i instancí všech jejích předků. Vztah „býti instancí třídy C“ je tedy týž jako vztah „býti typu C“. Pokud vytvoříme novou instanci třídy pomocí funkce `make-instance` s uvedením jména třídy v argumenty, nový objekt nazýváme *přímou instancí* této třídy. Každý objekt je tedy přímou instancí jedné třídy a instancí její a všech jejích předků.



Obrázek 2: Strom tříd grafických objektů, první verze

Princip datové abstrakce uvedený před chvílí naznačuje, jaký má být vztah mezi vlastnostmi a přijímanými zprávami třídy a jejích potomků. Postupujeme-li ve stromu dědičnosti zdola nahoru, dostáváme se od konkrétnějších tříd k obecnějším (například od konkrétnější třídy `point` k obecné `shape`). Tím postupně abstrahujeme, tedy vynecháváme některé konkrétní aspekty. To se odráží ve vztahu předek–potomek. Definováním potomka třídy zavádíme konkrétnější třídu. Její přímé instance budou tedy obsahovat obecně více vlastností a rozumět více zprávám.

To se odrazí i ve vnitřní konstrukci třídy: bude obsahovat všechny sloty, které obsahuje její předek, a případně nějaké další. A bude rozumět všem zprávám, kterým rozuměl její předek, a případně nějakým dalším.

Princip zapouzdření zakazuje přímý přístup ke slotům objektu (tj. pomocí funkce `slot-value`) odjinud, než z metod jeho třídy. Se zavedením dědičnosti se objekty stávají instancemi více tříd současně. Je tedy třeba princip zapouzdření upřesnit. Budeme používat nejpřísnější možnou variantu.

Princip zapouzdření s dědičností

Hodnoty slotů objektu smí přímo číst a měnit pouze metody třídy, jíž je objekt přímou instancí.

3 Pravidlo *is-a*

Při návrhu stromu dědičnosti nesmíme zapomínat, že má být zkonstruován podle vztahu podmnožina-nadmnožina mezi třídami. Základní princip správně navrženého stromu dědičnosti lze popsat následujícím pravidlem. Abyste mu porozuměli, uvědomte si, že správně vytvořený název typu (a tedy i třídy) je podstatné jméno (s přívlastky) v jednotném čísle, jako například *číslo*, *seznam* nebo *kruh* nebo *grafický objekt*. Název typu pak můžeme použít ve větě, jako například „každý *seznam* má nezápornou délku“ nebo „každý *kruh* je *grafický objekt*“. Druhá věta anglicky: Every `circle` *is a* `shape`.

Pravidlo *is-a*

Je-li třída D potomkem třídy C, pak věta „každé D je C“ musí dávat smysl.

Příklad: pravidlo *is-a*

Každý bod *je* grafickým útvarem (every point *is a* shape), každý kruh *je* grafickým útvarem. Proto strom na Obrázku 2 pravidlo *is-a* splňuje. Podobně například každý automobil je vozidlo, nebo každý pes je savec. Proto podle pravidla *is-a* lze definovat třídu automobilů jako potomka třídy vozidel a třídu psů jako potomka třídy savců.

Vztah volant–automobil, neodpovídá pravidlu *is-a* (volant *není* automobil), proto nelze definovat třídu volantů jako potomka třídy automobilů (volant by musel mít všechny vlastnosti automobilu, např. mít kola nebo výfuk). Kamión ale je automobil, takže pokud by se hodilo definovat třídu kamiónů, byla by potomkem třídy automobilů.

Je jasné, že pravidlo *is-a* se vztahuje především k předmětům reálného světa, které chceme našimi třídami modelovat. To je vidět z věty „Každý kruh je grafický útvar“ nebo „Žádný volant není automobil“. Obě věty nám poskytují dobré vodítko k tomu, jak dědičnost tříd navrhnout. Nepostihují ovšem problém dědičnosti úplně. K pravidlu *is-a* se proto ještě vrátíme na příští přednášce.

4 Určení předka v definici třídy

Při definici nové třídy makrem `defclass` lze určit místo definované třídy ve stávající hierarchii tříd. Konkrétně lze ke třídě stanovit jejího přímého předka. Obecnější možnost makra `defclass` je tato:

```
(defclass name parents slots)

  name: symbol
  parents: prázdný nebo jednoprvkový seznam
  slots: seznam
```

V druhém argumentu makra `defclass` (parametr ***parents***) lze kromě prázdného seznamu uvést i seznam obsahující jeden symbol. Pokud této možnosti využijeme, musí tento symbol být názvem nějaké již existující třídy. Nově vytvářená třída se pak stane jejím přímým potomkem.

(V seznamu ***parents*** lze uvést i více přímých předků nové třídy. Tato volba, kterou CLOS umožňuje, by měla za důsledek, že by hierarchie tříd měla složitější strukturu než strukturu stromu. Tento jev se nazývá *vícenásobná dědičnost*. V této části textu se budeme zabývat pouze jednoduchou dědičností.)

Příklad: třída `point` a `circle`

Nyní začneme měnit definice našich tříd a včleňovat je do postupně vznikajícího stromu dědičnosti.

Provedené změny budou zpětně kompatibilní. Veškerý kód napsaný pro třídy grafických objektů z předchozí kapitoly bude fungovat i v nové verzi.

Uživatel naší nové verze již ale bude počítat s námi zavedenou hierarchií tříd. Pokud budeme chtít zachovat zpětnou kompatibilitu, nebude ji možné v budoucnu měnit. Proto je třeba věnovat návrhu velkou pozornost.

Začneme u tříd `point` a `circle`. S těmito třídami nebude mnoho práce. Obě je třeba učinit bezprostředními potomky třídy `shape`. Dále vypustíme definice všech slotů a metod, které jsme do této třídy přesunuli.

Třída `point`. Metody pro práci s kartézskými a polárními souřadnicemi neuvádíme, protože se nezměnily. Metody pro vlastnosti `color` a `thickness` již nepotřebujeme, protože jsou přesunuty do třídy `shape`.

```
(defclass point (shape)
  ((x :initform 0)
   (y :initform 0)))
```

Třída `circle`. Metody pro vlastnosti `radius` a `center` jsou stejné jako dříve, proto je neuvádíme.

```
(defclass circle (shape)
  ((center :initform (make-instance 'point))
   (radius :initform 1)))
```

Příklad: třída `compound-shape`

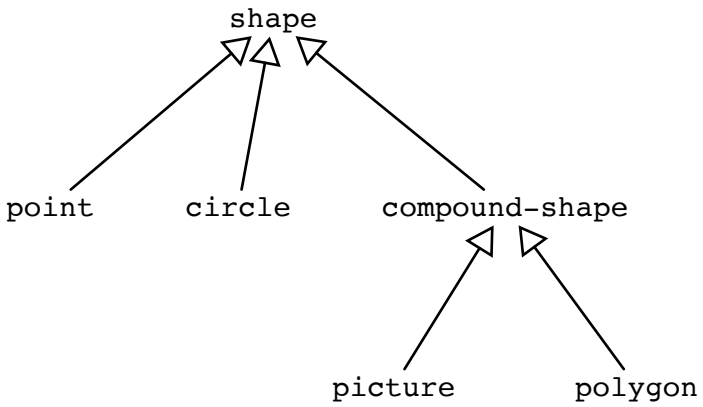
Co provést s vlastností `items` u tříd `picture` a `polygon`? Vlastnost slouží k uložení objektů, ze kterých se instance tříd skládají (v případě polygonů to mohou být body, u obrázků libovolné grafické objekty). U ostatních tříd nemá smysl tuto vlastnost definovat — jistě není vhodné a logické hovořit o seznamu podobjektů u bodů nebo kružnic.

Co tedy mají polygony a obrázky společného? V obou případech se jedná o objekty, které se skládají z jiných objektů, tedy o složené objekty (pravidlo *is-a*: polygon i obrázek *je* složený objekt). Má tedy smysl zavést společného předka tříd `polygon` a `picture` a metody společné těmto třídám přesunout do něj. Uvidíme, že těchto metod nebude málo a že se struktura našeho zdrojového kódu pročistí.

Struktura tříd ale ještě nebude konečná, příště ji dále zkomplikujeme.

Třídu složených grafických objektů nazveme `compound-shape`. Nový strom dědičnosti tříd grafických objektů je znázorněn na Obrázku 3.

Funkčnost tříd `polygon` a `picture`, která souvisí s tím, že se jedná o složené objekty (například, ale nejen, získávání a nastavování seznamu obsažených objektů) bude pomocí třídy `compound-shape` oddělena od funkčnosti pro tyto třídy specifické. To je příklad obecnějšího principu, podle něhož se různé třídy používají



Obrázek 3: Strom dědičnosti tříd grafických objektů

k implementaci různých funkčností. V dobře navrženém systému tříd je jasně a přehledně rozdělena zodpovědnost za různé úkoly mezi jednotlivé třídy.

Nově zavedená třída `compound-shape` obsahuje vlastnost `items`. Její metody ukážeme za chvíli, zatím uvedeme pouze definici třídy se slotem `items`. Hodnotu slotu neinicializujeme, takže při pokusu o jeho čtení dojde k chybě. Důvod vysvětlíme později:

```
(defclass compound-shape (shape)
  (items))
```

Ve třídách `polygon` a `picture` definici slotu opakujeme a tentokrát dodáváme i volbu `:initform`. V CLOSu se tím způsobí, že v těchto třídách se při vytváření instance slot inicializuje (je to ale týž slot, který byl definován už ve třídě `compound-shape`).

Třídě `polygon` zbyde ze všech ostatních slotů pouze slot `closedp`. Metody pro práci s ním budou stejné jako dříve.

```
(defclass polygon (compound-shape)
  ((items :initform '())
   (closedp :initform t)))
```

Nová definice třídy `picture`:

```
(defclass picture (compound-shape)
  ((items :initform '())))
```

5 Přepisování metod

Víme, že podle principu polymorfismu lze v každé třídě definovat jinou obsluhu téže zprávy. To může vést k tomu, že objekt, kterému je poslána zpráva, má na výběr mezi více metodami, které je možno jako obsluhu zprávy vykonat — každý objekt totiž může být instancí více tříd, z nichž každá může mít příslušnou metodu definovanou. V takové situaci **objektový systém vykoná metodu definovanou pro nejspecifičtější** (tedy ve stromu dědičnosti nejníže stojící) **třídu**, jíž je objekt instancí.

Konkrétněji: pokud je objektu zaslána zpráva, objektový systém hledá její obsluhu nejprve ve třídě, jíž je objekt přímou instancí. Pokud ji tam najde, zavolá ji. Nenajde-li ji, pokračuje ve hledání metody v bezprostředním předku třídy. Takto pokračuje, dokud metodu nenajde.

Každá třída má tedy k dispozici veškerou funkčnost svých předků. Říkáme, že třída *dědí* metody po svých předcích. Nejsme-li spokojeni se zděděnou metodou, můžeme ve třídě definovat metodu novou. V takovém případě říkáme, že ve třídě zděděnou metodu *přepisujeme*.

Příklad: geometrické transformace jednoduchých grafických objektů

Ukažme si nejprve uvedený princip na transformacích grafických objektů, tedy na metodách `move`, `rotate` a `scale`.

Především rozhodneme, jak tyto metody definujeme ve třídě `shape`. U instancí této třídy nemáme dostatek informací k tomu, abychom mohli uvést konkrétní definice. Máme tedy pouze dvě rozumné možnosti:

1. definovat metody tak, aby nic nedělaly,
2. definovat metody tak, aby vyvolaly chybu.

V našem případě volíme první možnost. Rozhodnutí podepřeme touto úvahou: zprávy `move`, `rotate` a `scale` mají za úkol měnit **geometrické vlastnosti** objektu. Ve třídě `shape` žádné takové vlastnosti nejsou, takže změnit je je možné a triviální. Není nutné měnit nic. Není to žádná chyba.

Metody `move`, `rotate` a `scale` pro třídu `shape` tedy definujeme tak, že pouze vrátí transformovaný grafický útvar jako svou hodnotu:

```
(defmethod move ((shape shape) dx dy)
  shape)

(defmethod rotate ((shape shape) angle center)
  shape)

(defmethod scale ((shape shape) coeff center)
  shape)
```

Ve třídách `point` a `circle` zůstaneme u původní implementace — metody třídy `shape` tedy **přepíšeme**. Například pro zprávu `move`:

```
(defmethod move ((pt point) dx dy)
  (set-x pt (+ (x pt) dx))
  (set-y pt (+ (y pt) dy))
  pt)

(defmethod move ((c circle) dx dy)
  (move (center c) dx dy)
  c)
```

Příklad: vlastnost `items`

Teď se podívejme na práci s vlastností `items` u složených grafických objektů. Metodu `items`, kterou jsme dříve definovali pro třídy `polygon` a `picture` zvlášť, můžeme beze změny přesunout do třídy `compound-shape`:

```
(defmethod items ((shape compound-shape))
  (copy-list (slot-value shape 'items)))
```

Metoda `set-items`:

```
(defmethod set-items ((shape compound-shape) value)
  (check-items shape value)
  (setf (slot-value shape 'items) (copy-list value))
  shape)
```

Jak víme, úkolem metody je otestovat, zda daný seznam splňuje požadavky kladené na seznam `items` daného složeného grafického objektu (pro polygony to má být seznam bodů, pro obrázky seznam libovolných grafických objektů) a v případě negativního výsledku vyvolat chybu.

Obvyklý způsob asi bude projít daný seznam a otestovat každý jeho prvek, jak jsme zatím dělali u tříd `polygon` a `picture`. Není to ale jediná možnost, obecněji může být potřeba zkontrolovat seznam samotný, například ověřit, zda má požadovaný počet prvků.

Navíc si teď musíme vybrat, zda metodu ve třídě `compound-shape` napsat jako prázdnou (tj. aby nedělala nic), nebo tak, aby vyvolala chybu. V tomto případě poprvé bez váhání použijeme druhou možnost. Pokud by totiž někdo navrhoval dalšího potomka třídy `compound-shape`, je nezbytné, aby tuto metodu přepsal — v případě, že by nekontrolovala nastavovaný seznam, mohla by způsobit nekonzistenci objektu.

Tento dvojí problém vyřešíme takto: metodu napíšeme tak, že vrátí chybu:

```
(defmethod check-items ((shape compound-shape) item-list)
  (error "Method check-items of compound-shape must be rewritten.")
  shape)
```

a dále napíšeme pomocnou metodu `do-check-items`, která otestuje prvky seznamu. Potomci ji budou moci použít ve své metodě `check-items`:

```
(defmethod do-check-items ((shape compound-shape) item-list)
  (dolist (item item-list)
    (check-item shape item))
  shape)
```

A konečně, metodu `check-item` napíšeme ze stejného důvodu jako před chvílí tak, aby vyvolala chybu:

```
(defmethod check-item ((shape compound-shape) item)
  (error "Method check-item of compound-shape must be rewritten."))
```

Po této reorganizaci můžeme ve třídách `polygon` a `picture` jednoduše přepsat metody `check-items` a `check-item`. (Všimněte si také výrazného zjednodušení metody `check-item` ve třídě `picture` možnému díky zavedení třídy `shape`.)

```
(defmethod check-items ((p polygon) items)
  (do-check-items p items))

(defmethod check-item ((p polygon) item)
  (unless (typep item 'point)
    (error "Items of polygon must be points.")))

(defmethod check-items ((p picture) items)
  (do-check-items p items))

(defmethod check-item ((p picture) item)
  (unless (typep item 'shape)
    (error "Items of picture must be shapes.")))
```

Příklad: geometrické transformace složených grafických objektů

Transformace složených grafických objektů nezávisí na tom, zda jde o obrázky, nebo polygony. Princip geometrických transformací je totiž dán obecně, bez ohledu na

konkrétní typ složeného grafického objektu. Proto metody zpráv `move`, `rotate` a `scale` patří do třídy `compound-shape`.

Než je napíšeme, připravíme si pomocnou metodu `send-to-items`, kterou mnohokrát využijeme a která slouží k poslání téže zprávy všem prvkům složeného grafického objektu (mohli jsme ji použít i před chvílí v metodě `do-check-items`).

```
(defmethod send-to-items ((shape compound-shape)
                          message
                          &rest arguments)
  (dolist (item (items shape))
    (apply message item arguments))
  shape)

(defmethod move ((shape compound-shape) dx dy)
  (send-to-items shape 'move dx dy)
  shape)

(defmethod rotate ((shape compound-shape) angle center)
  (send-to-items shape 'rotate angle center)
  shape)

(defmethod scale ((shape compound-shape) coeff center)
  (send-to-items shape 'scale coeff center)
  shape)
```

Příklad: kreslení ve třídě `shape`

U většiny tříd jsme postupovali v první verzi kreslení grafických objektů podle stejného vzoru: nejprve jsme metodou `set-mg-params` nastavili potřebné grafické parametry okna knihovny `micro-graphics` a potom jsme objekt metodou `do-draw` vykreslili. Tento postup je vhodný k tomu, aby byl definován obecně ve třídě `shape`:

```
(defmethod draw ((shape shape) mgw)
  (set-mg-params shape mgw)
  (do-draw shape mgw))
```

Autoři potomků třídy `shape` nyní nemusí přepisovat metodu `draw`, pouze, pokud je třeba, metody `set-mg-params` a `do-draw`.

Takto to funguje u jednodušších grafických objektů (například jednobarevných), u kterých lze kreslení na dvě fáze rozdělit. Jak víme, u třídy `picture` tímto způsobem kreslit nemůžeme. Proto metodu **přepíšeme**:

```
(defmethod draw ((pic picture) mg-window)
  (dolist (item (reverse (items pic)))
    (draw item mg-window))
  pic)
```

Doplňme ještě do třídy `shape` metody `set-mg-params` a `do-draw`. Pro druhou z nich nemůžeme ve třídě `shape` napsat implementaci, která by něco dělala, protože v této třídě nemáme žádné informace o tvaru objektu. Proto (ze stejného důvodu jako u geometrických transformací) metodu definujeme tak, aby nic nedělala:

```
(defmethod do-draw ((shape shape) mgw)
  shape)
```

Metodu `set-mg-params` napíšeme tak, že nastaví všechny parametry okna podle hodnot příslušných vlastností. Tento přístup zbaví některé třídy nutnosti metodu přepisovat:

```
(defmethod set-mg-params ((shape shape) mgw)
  (mg:set-param mgw :foreground (color shape))
  (mg:set-param mgw :filledp (filledp shape))
  (mg:set-param mgw :thickness (thickness shape))
  shape)
```

Všimněme si, že nám vznikají dva druhy metod: jedny jsou určené spíše k tomu, aby je uživatel volal (tj. aby objektům zasílal příslušné zprávy; to se týká metody `draw`), zatímco u druhých se to neočekává (`set-mg-params`, `do-draw`). Metody druhého typu jsou pouze připraveny k tomu, aby byly v nově definovaných třídách přepsány. Toto rozdělení bude na příští přednášce ještě výraznější.

Metodám, které nemají být explicitně volány, ale jsou určeny pouze k tomu, aby byly v potomcích tříd (případně) přepsány, se v některých jazycích říká *chráněné metody* (*protected methods*).

6 Volání zděděné metody

Víme, že díky dědičnosti je metoda definovaná pro třídu automaticky definovaná i pro všechny její podtřídy. Pokud ale v některé podtřídě tuto metodu přepíšeme novou metodou, spustí se při zaslání příslušné zprávy instanci této podtřídy tato nová metoda.

Metodu třídy můžeme v podtřídě přepsat. Uvnitř nové metody ale můžeme zařídit, aby se zavolala původní metoda definovaná v předkovi. V Lispu se k tomu využívá funkce `call-next-method`.

Příklad: kreslení u potomků třídy `shape`

Techniku si nejprve ukážeme na metodě `set-mg-params` třídy `polygon`. U instancí třídy je třeba kromě nastavování grafických parametrů `foreground`, `filledp` a `thickness` nastavit navíc parametr `closedp`. Ten totiž koresponduje s vlastností `closedp`, která je definována jen ve třídě `polygon` (v ostatních se nenachází). Proto metodu `set-mg-params` třídy `shape` sice přepíšeme, ale v těle nové metody ji zavoláme:

```
(defmethod set-mg-params ((p polygon) mgw)
  (call-next-method)
  (mg:set-param mgw :closedp (closedp p))
  p)
```

Po zaslání zprávy `set-mg-params` polygonu se zavolá tato metoda. Funkce `call-next-method` v těle metody způsobí zavolání metody `set-mg-params` třídy `shape` (třída `shape` je první předek třídy `polygon`, který metodu `set-mg-params` obsahuje). Tím se zařídí nastavení všech potřebných grafických parametrů metodou ve třídě `shape` i ve třídě `polygon`.

Metoda `do-draw` je ve třídě `polygon` stejná jako dříve.

U třídy `circle` není nutno přidávat ani metodu `draw`, ani metodu `set-mg-params`. Stačí pouze definice metody `do-draw` tak, jak byla uvedena na předchozí přednášce.

U třídy `point` je kreslení poněkud netypické — tato třída ignoruje obsah slotu `filledp` a před kreslením nastavuje hodnotu příslušného grafického parametru knihovny `micro-graphics` na `t`. To je vhodná příležitost k volání zděděné metody v metodě `set-mg-params`:

```
(defmethod set-mg-params ((pt point) mgw)
  (call-next-method)
  (mg:set-param mgw :filledp t)
  pt)
```

Metodu `draw` třídě `point` nedefinujeme, metoda `do-draw` zůstává stejná jako dříve.

Kreslení instancí třídy `picture` můžeme nechat beze změny.

7 Inicializace instancí

Nově vytvářené instance je někdy třeba inicializovat složitějším způsobem, než jak to umožňuje volba `:initform` v definici třídy. U většiny programovacích jazyků k tomu slouží zvláštní metody nazývané *konstruktory*. V Common Lispu je možné použít metodu `initialize-instance`.

Funkce `make-instance`, která slouží k vytváření nových instancí tříd, vždy nejprve novou instanci vytvoří a pak jí pošle zprávu `initialize-instance`. V obsluze

této zprávy tedy může nově vytvářený objekt provést inicializace, na které nestačí volba `:initform` v definici třídy. Zpráva `initialize-instance` patří ke zprávám, které objektům nezasíláme, ale pouze přepisujeme její metody.

Definice metody `initialize-instance` má následující tvar:

```
(defmethod initialize-instance ((var class) &key)
  ... kód metody ...)
```

Symbol `&key` v definici metody je třeba vždy uvést, protože metoda může mít klíčové parametry. Ty zatím nebudeme používat a za symbol nebudeme nic dalšího psát. V metodě `initialize-instance` pro každou třídu je vždy nutno umožnit inicializaci instance definovanou v rodičích třídy. Vždy je tedy nutno volat funkci `call-next-method`.

Metoda `initialize-instance`

Zprávu `initialize-instance` objektům nezasíláme, ale pouze přepisujeme její metody. V nich vždy voláme funkci `call-next-method`.

Příklady použití metody `initialize-instance` najdete v příkladech k této části textu.

Otázky a úkoly na cvičení

1. Upravte třídu `triangle` podle poznatků z této kapitoly. Všem zprávám, kterým instance rozuměly dosud, by měly rozumět i po úpravě. Jaký by měl být vztah této třídy a třídy `polygon`?
2. Totéž udělejte pro třídy `empty-shape` a `full-shape`.
3. Definujte třídu `triangle` znovu jako potomka třídy `polygon`. Nastavení vlastnosti `items` by nemělo trojúhelník uvést do nekonzistentního stavu.
4. Upravte třídu `ellipse` podle poznatků z této kapitoly. Jaký by měl být vztah této třídy a třídy `circle`?
5. Dodefinujte třídám grafických objektů nové metody `left`, `top`, `right`, `bottom` tak, aby po obdržení zprávy `left` (`top`, `right`, `bottom`) objekt vrátil souřadnici svého levého (horního, pravého, dolního) okraje. Nemusíte uvažovat tloušťku pera. Jak definovat metody pro třídu `shape`? U třídy `ellipse` výpočet vyžaduje určité geometrické znalosti, tuto třídu zatím můžete vynechat.
6. Definujte třídu `extended-picture`, která bude potomkem třídy `picture` a navíc bude mít vlastnost `propagate-color-p`. Pokud bude hodnota této vlastosti *Pravda*, obrázek při přijetí zprávy `set-color` nastaví na tutéž barvu i barvu všech svých prvků.