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

3. Polymorfismus

verze z 7. října 2020

1 Geometrické transformace

Nejprve uvedeme příklady operací, které má smysl provádět se všemi grafickými objekty bez ohledu na to, jakého typu objekty jsou: posunutí, rotaci a změnu velikosti.

Příklad: posunutí

Posunutí je operace, která změní polohu grafického objektu na základě zadaných přírůstků souřadnic. Objekt posuneme tak, že mu pošleme zprávu `move`, jejíž syntax je následující:

```
(move object dx dy)

  object:  grafický objekt, jemuž zprávu posíláme
  dx, dy:  čísla
```

Čísla ***dx*** a ***dy*** jsou přírůstky na ose x a y, o něž chceme objekt ***object*** posunout.

Je zřejmé, že zatímco z hlediska uživatele není podstatné, jaký grafický objekt posouváme, obsluha zprávy `move` bude u objektů různých tříd různá. Definice metod se tedy budou u různých tříd lišit:

```
(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)

(defmethod move ((poly polygon) dx dy)
  (dolist (item (items poly))
    (move item dx dy))
  poly)
```

U třídy `point` jednoduše přičítáme přírůstky `dx` a `dy` ke kartézským souřadnicím bodu. Kruh posouváme tak, že posouváme jeho střed. U polygonu posouváme všechny jeho vrcholy.

Příklad: otočení

Rotace je otočení objektu o daný úhel kolem daného bodu. Definujeme zprávu `rotate` s následující syntaxí:

```
(rotate object angle center)

  object:  grafický objekt
  angle:   číslo
  center:  instance třídy point
```

Zde *`angle`* je úhel, o který chceme objekt otočit, a *`center`* střed rotace.

V obsluze zprávy `rotate` u třídy `point` budeme postupovat tak, že bod nejprve posuneme, aby střed rotace splýval s počátkem souřadnic, pak změníme jeho polární souřadnice a posuneme jej zpět. Implementace tohoto postupu bude následující:

```
(defmethod rotate ((pt point) angle center)
  (let ((cx (x center))
        (cy (y center)))
    (move pt (- cx) (- cy))
    (set-phi pt (+ (phi pt) angle))
    (move pt cx cy)
    pt))
```

Metoda pro třídu `circle` již pouze správně otočí střed kruhu, u třídy `polygon` otočíme všechny vrcholy:

```
(defmethod rotate ((c circle) angle center)
  (rotate (center c) angle center)
  c)

(defmethod rotate ((poly polygon) angle center)
  (dolist (item (items poly))
    (rotate item angle center))
  poly)
```

Příklad: zvětšení

Další základní geometrickou transformací je *zvětšení*, které realizujeme pomocí zprávy `scale`. Podrobnosti najdete ve zdrojovém kódu k této přednášce.

2 Obrázky

Třída `picture` umožňuje spojit několik grafických objektů do jednoho a pracovat s nimi jako s jedním celkem. Tím usnadňuje operace prováděné na všech objektech současně. Instance třídy si můžeme představit jako skupinu grafických objektů spojených do jednoho (ve vektorových grafických editorech podobnou roli hraje příkaz *group*).

Implementace třídy je velmi podobná implementaci třídy `polygon`. Tady jsou základní definice:

```
(defclass picture ()
  ((items :initform '())))

(defmethod items ((pic picture))
  (copy-list (slot-value pic 'items)))

(defmethod set-items ((pic picture) value)
  (check-items pic value)
  (setf (slot-value pic 'items) (copy-list value))
  pic)

(defmethod check-items ((pic picture) items)
  (dolist (item items)
    (check-item pic item))
  pic)

(defmethod check-item ((pic picture) item)
  (unless (or (typep item 'point)
              (typep item 'circle)
              (typep item 'polygon)
              (typep item 'picture))
    (error "Invalid picture element type."))
  pic)
```

Jediný rozdíl vidíme v metodě `check-item`. Zatímco prvky seznamu `items` polygonu smějí být pouze body, u obrázku to může být libovolný grafický objekt (včetně obrázku). Poznamenejme, že pokud definujeme novou třídu grafických objektů (jako například třídy `triangle`, `full-shape`, `empty-shape`, `ellipse` z cvičení), nesmíme zapomenout změnit i metodu `check-item` třídy `picture`. Tuto zjevnou nevýhodu, která může být zdrojem chyb, odstraníme na příští přednášce.

Metody pro geometrické transformace můžeme napsat přesně stejně jako u třídy `polygon`. Například metoda pro zprávu `move`:

```
(defmethod move ((pic picture) dx dy)
  (dolist (item (items pic))
    (move item dx dy))
  pic)
```

3 Princip polymorfismu

Aniž bychom o tom hovořili, využili jsme v předchozích příkladech *princip polymorfismu*.

Princip polymorfismu pro jazyky založené na třídách
Různé třídy mohou mít definovány pro tutéž zprávu různé metody.

Princip samozřejmě vyplývá už ze základní charakteristiky objektově orientovaného programování, jak jsme ji uvedli na první přednášce. Na této přednášce jsme jej poprvé použili. Teď si jej trochu důkladněji rozebereme.

Princip polymorfismu jsme v předchozích příkladech využili dvakrát. Poprvé při definici metod pro zprávy geometrických transformací (zprávy `move`, `scale` a `rotate`), podruhé, když jsme je zasílali prvkům obrázků v metodách `move`, `scale` a `rotate` třídy `picture`. Pojďme si tyto případy rozebrat na příkladě zprávy `move`.

Při definici metod tříd `point`, `circle` a `polygon` nám princip polymorfismu umožnil definovat pro každou ze tříd metody téhož názvu `move`, ale s různou implementací — metody v těchto třídách mají různé definice. Lze říci, že systém umožňuje pojmenovat akce, které se liší provedením (implementací), ale nikoli významem, stejným názvem.

V některých staticky typovaných programovacích jazycích (C++, C#, Java) se setkáme s možností definovat různé funkce stejným názvem, pokud se liší počtem nebo typem parametrů. Této možnosti se říká *přetěžování funkcí* (*function overloading*). Principiálně jde ale o něco jiného. Hlavní rozdíl, který nás teď zajímá, je tento: u přetěžování funkcí jde ve skutečnosti o různé funkce, které sice z pohledu programátora mají týž název, ale překladač je vždy umí rozlišit podle typu a počtu parametrů. Za běhu programu už funkce nemají nic společného. Přetěžování funkcí lze tedy používat jen spolu se statickým typováním. S pohledu polymorfismu se někdy mluví o *statickém* či *nepravém polymorfismu*.

Dynamický nebo též *pravý polymorfismus*, což je ten, kterým se zabýváme zde, využívá dynamického typování. Třída každého objektu musí být známa za běhu programu, protože až za běhu programu se rozhoduje, která metoda zaslané zprávy se zavolá. U různých metod téže zprávy nejde o různé funkce bez jakéhokoli vzájemného vztahu (jako je tomu u statického polymorfismu), protože i za běhu programu musí být známo, že jde o různé metody téže zprávy.

Metoda `move` třídy `picture` z předchozího příkladu by nefungovala nebýt dynamického polymorfismu. O metodách zavolaných pro jednotlivé prvky obrázku se rozhoduje až za běhu a jinak to být nemůže. I staticky typované programovací jazyky musí tedy v podobných případech používat dynamické typování — musí znát typ (třídu) objektu za běhu programu. Takto konstruovaným metodám se pak v těchto jazycích obvykle říká *dynamické metody*.

Příklad: posunutí obrázku bez polymorfismu?

Kdyby nebylo principu polymorfismu, tj. kdyby neexistovaly metody, posouvání objektů bychom realizovali klasickými funkcemi. Ty by musely mít pro různé typy objektů různé názvy. Kdyby tyto názvy byly například `point-move` pro typ `point`, `circle-move` pro typ `circle`, `polygon-move` pro typ `polygon` a `picture-move` pro typ `picture`, musela by definice funkce `picture-move` vypadat takto:

```
(defun picture-move (pic dx dy)
  (dolist (item (items pic))
    (cond ((typep item 'point) (point-move item dx dy))
          ((typep item 'circle) (circle-move item dx dy))
          ((typep item 'polygon) (polygon-move item dx dy))
          ((typep item 'picture) (picture-move item dx dy)))))
```

Volat různé funkce podle typu objektu by musely i další funkce, například funkce pro rotaci a změnu velikosti objektu. To by přinášelo nežádoucí opakování kódu: každá funkce by musela obsahovat totéž větvení podle typu objektu (ve funkci `picture-move` realizované `cond`-výrazem).

Drobná poznámka na okraj. V Lispu by mohlo pomoci použití vhodného makra, ať už uživatelského, nebo vestavěného, např. `typecase`:

```
(defun picture-move (pic dx dy)
  (dolist (item (items pic))
    (typecase item
      (point (point-move item dx dy))
      (circle (circle-move item dx dy))
      (polygon (polygon-move item dx dy))
      (picture (picture-move item dx dy)))))
```

Bylo by to ale jen nedokonalé záplatování principiálního problému.

Kromě toho má tato definice ještě jednu nevýhodu: kdykoliv bychom definovali nový typ grafických objektů, museli bychom upravit i funkci `picture-move`. To se týká například tříd, jež jste definovali na cvičení (např. `triangle`).

Změna na jednom místě programu by tedy znamenala nutnost změny i na dalších místech. Této nutnosti nás princip polymorfismu zbavuje.

V objektově orientovaných programovacích jazycích bývá princip polymorfismu přítomen. Předchozí příklad proto není reálný. Při používání procedurálního programovacího stylu se s podobnými jevy ale setkáváme.

Jak jsme ale už viděli, po definici nové třídy (např. `triangle`) budeme stejně muset předefinovat jinou metodu třídy `picture`: metodu `check-item`. Tuto nepříjemnost vyřešíme lépe až na příští přednášce pomocí dědičnosti.

4 Kreslení pomocí knihovny `micro-graphics`

K vykreslování našich grafických objektů budeme využívat jednoduchou procedurální grafickou knihovnu `micro-graphics`, napsanou pro potřeby tohoto textu. Na této přednášce budeme používat jen některé její základní funkce: `mg:display-window`, `mg:get-param`, `mg:set-param`, `mg:clear`, `mg:draw-circle` a `mg:draw-polygon`. Jejich popis najdete v následující části.

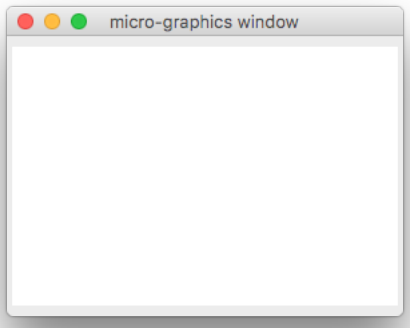
Příklad: použití knihovny `micro-graphics`

Vyzkoušejme některé ze služeb knihovny `micro-graphics`. Nejprve knihovnu načtěme do prostředí LispWorks volbou nabídky „Load...“ a souboru `load.lisp` (při volbě nabídky „Load...“ musí být aktivní okno s příkazovým řádkem, nikoliv okno s editovaným souborem, jinak se načte tento soubor).

Nejprve zavoláme funkci `mg:display-window` a výsledek uložíme do proměnné:

```
CL-USER 40 > (setf w (mg:display-window))
#<MG-WINDOW 20099673>
```

Otevře se nově vytvořené okno knihovny `micro-graphics`, jak vidíme na Obrázku 1.



Obrázek 1: Prázdné okno knihovny `micro-graphics`

Výsledek tohoto volání (v našem případě zapisovaný prostředím jako `#<MG-WINDOW 20099673>`) slouží pouze jako identifikátor okna, který budeme používat při dalších voláních funkcí knihovny. Žádný jiný význam pro nás nemá.

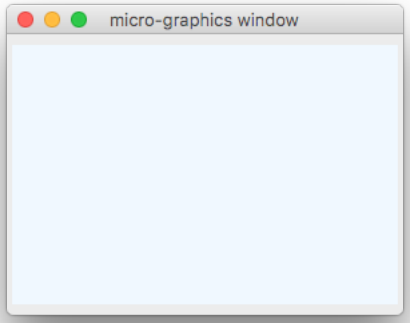
Pomocí funkce `mg:get-param` můžeme zjistit přednastavené kreslicí parametry okna:

```
CL-USER 41 > (mapcar
               (lambda (p)
                 (list p (mg:get-param w p)))
               '(:thickness :foreground :background
                 :filledp :closedp))
((:THICKNESS 1) (:FOREGROUND :BLACK) (:BACKGROUND :WHITE) (:FILLEDP
NIL) (:CLOSEDP NIL))
```

Kreslicí parametry můžeme nastavit funkcí `mg:set-param`. Zkusme tedy změnit barvu pozadí a potom pomocí funkce `mg:clear` okno vymazat:

```
CL-USER 42 > (mg:set-param w :background :aliceblue)
NIL

CL-USER 43 > (mg:clear w)
NIL
```

Pokud jsme to udělali dobře, pozadí okna se přebarví na světle modrou (Obrázek 2).



Obrázek 2: Okno knihovny `micro-graphics` po změně barvy pozadí

Poznámka: jak víme, v Common Lispu se symboly začínající dvojtečkou vyhodnocují na sebe. Proto vyhodnocení předchozího výrazu neskončilo chybou. Můžeme si to vyzkoušet konkrétně:

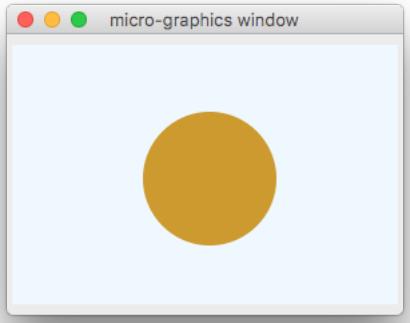
```
CL-USER 1 > :background
:BACKGROUND

CL-USER 2 > :aliceblue
:ALICEBLUE
```

Symbolům začínajícím dvojtečkou říkáme *klíče*.

Jako další krok otestujeme funkci `mg:draw-circle`. Víme, že tato funkce vyžaduje jako parametry údaje o středu a poloměru vykreslovaného kruhu. Kromě toho její výsledek ovlivňují kreslicí parametry `:foreground`, `:thickness` a `:filledp`. Nastavme tedy nejprve například parametry `:foreground` a `:filledp` a zavolejme funkci `mg:draw-circle`:

```
CL-USER 44 > (progn
               (mg:set-param w :foreground :goldenrod3)
               (mg:set-param w :filledp t)
               (mg:draw-circle w 148 100 50))
NIL
```

V okně se objeví vyplněný kruh barvy `:goldenrod3` (Obrázek 3).



Obrázek 3: Okno knihovny `micro-graphics` po nakreslení kruhu

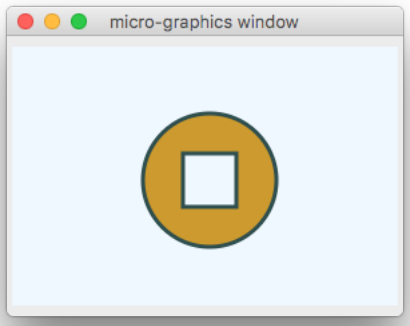
Nyní ještě změníme parametr `:foreground` a zkusíme pomocí funkce `mg:draw-polygon` nakreslit čtverec:

```
CL-USER 45 > (progn
               (mg:set-param w :foreground :aliceblue)
               (mg:draw-polygon w '(128 80 168 80
                                    168 120 128 120)))
NIL
```

Konečně, pro vylepšení efektu nastavíme novou barvu pera, parametry `:filledp`, `:thickness` a (kvůli uzavření čtverce) `:closedp` a znovu vykreslíme kruh a čtverec:

```
CL-USER 46 > (progn
               (mg:set-param w :foreground :darkslategrey)
               (mg:set-param w :thickness 3)
               (mg:set-param w :filledp nil)
               (mg:set-param w :closedp t)
               (mg:draw-circle w 148 100 50)
               (mg:draw-polygon w '(128 80 168 80
                                    168 120 128 120)))
NIL
```

Výslednou podobu okna vidíme na Obrázku 4.



Obrázek 4: Okno knihovny `micro-graphics` po nakreslení dvou kruhů a dvou polygonů

5 Základní funkce knihovny `micro-graphics`

Následuje dokumentace funkcí knihovny `micro-graphics` použitých v minulé části.

`(mg:display-window)` => *window*

window: odkaz na okno

Vytvoří a zobrazí nové grafické okno. Jako výsledek vrací odkaz na toto okno, který je třeba používat jako parametr v ostatních funkcích, jež s oknem pracují. Nové okno má několik **kreslicích parametrů**, které lze zjišťovat pomocí funkce `mg:get-param` a nastavovat pomocí funkce `mg:set-param`. Souřadnice v okně se

udávají v pixelech, jejich počátek je v levém horním rohu okna, hodnoty druhé souřadnice se zvětšují směrem dolů. Rozměry okna jsou 297 na 210 pixelů.

(mg:get-param *window param***)** => *value*

window: hodnota vrácená funkcí `mg:display-window`
param: symbol

Funkce `mg:get-param` vrací hodnotu kreslicího parametru ***param*** okna ***window***. Pro nás jsou důležité tyto parametry:

:thickness	Tloušťka čáry v pixelech. Ovlivňuje funkce na kreslení obrazců (např. `mg:draw-circle`), pokud není nastaven parametr `:filledp`. Počáteční hodnota: `1`.
:foreground	Barva inkoustu. Ovlivňuje funkce na kreslení obrazců (např. `mg:draw-circle`). Počáteční hodnota: `:black`.
:background	Barva pozadí. Ovlivňuje funkci `mg:clear`. Počáteční hodnota: `:white`.
:filledp	Zda kreslit obrazce vyplněné. Ovlivňuje funkce na kreslení obrazců (např. `mg:draw-circle`). Počáteční hodnota: `nil`.
:closedp	Zda spojit poslední a první vrchol polygonu. Ovlivňuje funkci `mg:draw-polygon`, pokud není nastaven parametr `filledp`. Počáteční hodnota: `nil`.

Přípustnými hodnotami parametrů `:foreground` a `:background` jsou všechny symboly, které v grafickém systému LispWorks pojmenovávají barvu. Jejich seznam lze zjistit funkcí `color:get-all-color-names`, nebo, pokud uvedeme část názvu barvy, kterou chceme použít, funkcí `color:apropos-color-names`. Vzorkovník barev je také ve zvláštním souboru.

Barvy lze také v LispWorks vytvářet z komponent pomocí zabudovaných funkcí `color:make-rgb`, `color:make-hsv`, `color:make-gray`. Zájemci se mohou na tyto funkce podívat do dokumentace.

Kreslicí parametry lze nastavovat funkcí `mg:set-param`.

(mg:set-param *window param value***)** => nil

window: hodnota vrácená funkcí `mg:display-window`
param: symbol
value: hodnota

Funkce `mg:set-param` nastavuje kreslicí parametr ***param*** okna ***window*** na hodnotu ***value***. Význam kreslicích parametrů je uveden u funkce `mg:get-param`. Nové kreslicí parametry ovlivňují způsob kreslení do okna od momentu, kdy byly nastaveny.

(mg:clear *window*) => nil

window: hodnota vrácená funkcí mg:display-window

Funkce mg:clear vymaže celé okno *window* barvou aktuálně uloženou v kreslicím parametru :background.

(mg:draw-circle *window x y r*) => nil

window: hodnota vrácená funkcí mg:display-window
x, y, r: čísla

Funkce mg:draw-circle nakreslí do okna *window* kruh se středem o souřadnicích *x*, *y* a poloměrem *r*. Kruh se kreslí barvou uloženou v kreslicím parametru :foreground okna *window*. Kreslicí parametr :filledp okna *window* udává, zda se bude kruh kreslit vyplněný. Pokud není nastaven, bude se kreslit pouze obvodová kružnice čarou, jejíž tloušťka je uložena v kreslicím parametru :thickness okna *window*.

(mg:draw-ellipse *window x y rx ry phi*) => nil

window: hodnota vrácená funkcí mg:display-window
x, y, rx, ry, phi: čísla

Funkce mg:draw-ellipse nakreslí do okna *window* elipsu se středem o souřadnicích *x*, *y* s poloosami *rx* a ry. Elipsa bude natočená (kolem středu) o úhel phi. Kreslí se barvou uloženou v kreslicím parametru :foreground okna *window*. Kreslicí parametr :filledp okna *window* udává, zda se bude elipsa kreslit vyplněná. Pokud není nastaven, bude se kreslit pouze obvod čarou, jejíž tloušťka je uložena v kreslicím parametru :thickness okna *window*.

(mg:draw-polygon *window points*) => nil

window: hodnota vrácená funkcí mg:display-window
points: seznam čísel

Funkce mg:draw-polygon nakreslí do okna *window* polygon s vrcholy danými parametrem *points*. Tento parametr musí obsahovat seznam sudé délky, jako prvky se v něm musí střídat $x$ové a $y$ové souřadnice vrcholů polygonu. Kreslí se barvou uloženou v kreslicím parametru :foreground okna *window*. Kreslicí parametr :filledp okna *window* udává, zda se bude polygon kreslit vyplněný. Pokud není nastaven, budou se kreslit pouze úsečky spojující jednotlivé vrcholy polygonu čarou, jejíž tloušťka je uložena v kreslicím parametru :thickness okna *window*. Kreslicí parametr :closedp okna *window* určuje, zda se má nakreslit i úsečka spojující poslední bod polygonu s prvním. Pokud je nastaven kreslicí parametr :filledp, kreslicí parametr :closedp se ignoruje.

6 Kreslení grafických objektů

Když jsme se naučili používat procedurální grafickou knihovnu `micro-graphics`, zkusíme ji využít ke kreslení našich grafických objektů.

Budeme pokračovat v objektovém přístupu; proto budeme grafické objekty kreslit tak, že jim budeme posílat zprávy a necháme je, aby vykreslení pomocí knihovny `micro-graphics` již provedly samy ve svých metodách.

Knihovna `micro-graphics` není objektová. Výsledek kreslení je vždy závislý na hodnotách, které je nutno předem nastavit. V objektovém přístupu se snažíme dodržovat princip nezávislosti objektů. Proto požadujeme, aby výsledky akcí prováděných s objekty byly pokud možno závislé pouze na vnitřním stavu objektů (a hodnotách parametrů zpráv objektům zasílaných). Proto budou informace o způsobu kreslení (barva, tloušťka pera a podobně) součástí vnitřního stavu objektů, stejně jako informace o okně, do nějž se objekty mají kreslit.

To je v souladu s principy objektového programovaní i s intuitivní představou: například barva kruhu je zjevně jeho vlastnost, kruh by tedy měl údaj o ní nějakým způsobem obsahovat a při kreslení by na ni měl brát ohled.

Příklad: třída `window`

Uveďme nejprve definici třídy `window`, jejíž instance budou obsahovat informace o okně knihovny `micro-graphics`, do něhož lze kreslit naše grafické objekty (vlastnost `mg-window`), a další údaje. Mezi ně patří:

- grafický objekt, který se do okna vykresluje (vlastnost `shape`),
- barva pozadí okna (vlastnost `background`).

Definice třídy:

```
(defclass window ()
  ((mg-window :initform (mg:display-window))
   (shape :initform nil)
   (background :initform :white)))
```

Je vidět, že při vytvoření instance této třídy se automaticky otevře nové okno.

Definice metod pro jednotlivé vlastnosti:

```
(defmethod shape ((w window))
  (slot-value w 'shape))

(defmethod set-shape ((w window) shape)
  (setf (slot-value w 'shape) shape)
  w)
```

```
(defmethod background ((w window))
  (slot-value w 'background))

(defmethod set-background ((w window) color)
  (setf (slot-value w 'background) color)
  w)
```

Vlastnost pro slot `mg-window` nedefinujeme. Hodnotu slotu (tedy odkaz na okno knihovny `micro-graphics`) považujeme za soukromou. Budeme ji prozrazovat jen grafickým objektům při jejich kreslení, aby věděly, do kterého okna se mají nakreslit (viz dále).

Chceme-li vykreslit obsah okna, pošleme mu zprávu `redraw`. Metoda `redraw` vykreslí obsah okna tak, že nejprve zjistí barvu pozadí (vlastnost `background`), touto barvou obsah okna vymaže (funkcí `mg:clear`) a nakonec pošle grafickému objektu ve slotu `shape` zprávu `draw`:

```
(defmethod redraw ((window window))
  (let ((mgw (slot-value window 'mg-window)))
    (mg:set-param mgw :background (background window))
    (mg:clear mgw)
    (when (shape window)
      (draw (shape window) mgw)))
  window)
```

Zpráva `draw` má následující syntax:

```
(draw object mg-window)
```

Po jejím zaslání grafickému objektu by se měl tento objekt nakreslit do zadaného okna knihovny `micro-graphics`. Obsluhu této zprávy bude tedy třeba definovat pro všechny třídy grafických objektů (zpráva využívá princip dynamického polymorfismu).

Příklad: rozšíření třídy `circle`

Nyní implementujeme kreslení pro třídu `circle`. Už víme, že je třeba definovat metodu pro zprávu `draw`. Po jejím obdržení by se měl kruh vykreslit do zadaného okna knihovny `micro-graphics`. To bude vyžadovat přidání vlastností a metod třídě `circle`:

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

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

(defmethod color ((c circle))
  (slot-value c 'color))

(defmethod set-color ((c circle) value)
  (setf (slot-value c 'color) value)
  c)

(defmethod thickness ((c circle))
  (slot-value c 'thickness))

(defmethod set-thickness ((c circle) value)
  (setf (slot-value c 'thickness) value)
  c)

(defmethod filledp ((c circle))
  (slot-value c 'filledp))

(defmethod set-filledp ((c circle) value)
  (setf (slot-value c 'filledp) value)
  c)
```

Význam vlastností `color`, `thickness` a `filledp` je jasný. Budeme je používat k určování vzhledu kresleného kruhu.

Příklad: kreslení ve třídě `circle`

Metoda `draw` třídy `circle` bude sestávat ze dvou částí:

1. Nastavení kreslicích parametrů okna podle hodnot slotů kruhu,
2. nakreslení kruhu (funkcí `mg:draw-circle`).

Bude rozumné definovat kód pro tyto dva úkony zvlášť. (Podle obecného principu: rozdělit složitější úkol na více jednodušších. Přestože to teď netušíme, rozhodnutí vedené tímto obecným principem se nám v budoucnu vyplatí.) Jelikož programujeme objektově, definujeme dvě pomocné zprávy, jejichž obsluha tyto úkony provede. První z nich nazveme `set-mg-params`. Příslušná metoda bude vypadat takto:

```
(defmethod set-mg-params ((c circle) mg-window)
  (mg:set-param mg-window :foreground (color c))
  (mg:set-param mg-window :thickness (thickness c))
```

```
  (mg:set-param mg-window :filledp (filledp c))
  c)
```

Metoda nastavuje kreslicí parametry zadaného okna knihovny `micro-graphics` podle vlastností kruhu.

Zprávu pro vlastní kreslení nazveme `do-draw`. Zde je její metoda:

```
(defmethod do-draw ((c circle) mg-window)
  (mg:draw-circle mg-window
                  (x (center c))
                  (y (center c))
                  (radius c))
  c)
```

K dokončení už zbývá pouze definovat vlastní metodu `draw`. Ta ovšem bude jednoduchá:

```
(defmethod draw ((c circle) mg-window)
  (set-mg-params c mg-window)
  (do-draw c mg-window))
```

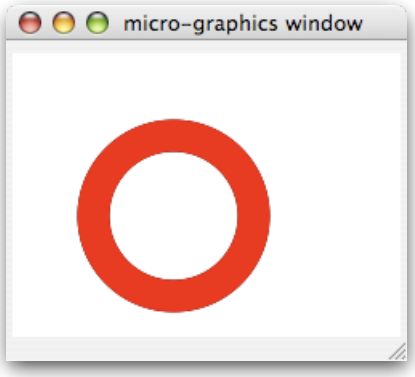
Příklad: test kreslení koleček

Test kódu z předchozích příkladů:

```
CL-USER 1 > (setf w (make-instance 'window))
#<WINDOW 217F04BF>

CL-USER 2 > (setf circ (make-instance 'circle))
#<CIRCLE 218359FB>

CL-USER 3 > (set-x (center circ) 100)
#<POINT 2183597B>

CL-USER 4 > (set-y (center circ) 100)
#<POINT 2183597B>

CL-USER 5 > (set-radius circ 50)
#<CIRCLE 218359FB>

CL-USER 6 > (set-color circ :red)
#<CIRCLE 218359FB>
```

```
CL-USER 7 > (set-thickness circ 20)
#<CIRCLE 218359FB>

CL-USER 8 > (set-shape w circ)
#<WINDOW 217F04BF>

CL-USER 9 > (redraw w)
#<WINDOW 217F04BF>
```

Výsledek by měl odpovídat Obrázku 5. Kruh v okně můžeme kdykoli překreslit



Obrázek 5: Červené kolečko

zavoláním `(redraw w)`.

Příklad: funkce na vytvoření objektu

K vytvoření kolečka z předchozího příkladu si můžeme napsat funkci:

```
(defun make-red-circle ()
  (let ((circ (make-instance 'circle)))
    (set-x (center circ) 100)
    (set-y (center circ) 100)
    (set-radius circ 50)
    (set-color circ :red)
    (set-thickness circ 20)
    circ))
```

Pak můžeme pomocí této funkce kolečko vytvořit a vykreslit v okně:

```
CL-USER 3 > (setf w (make-instance 'window))
#<WINDOW 21F18F5F>

CL-USER 4 > (setf c (make-red-circle))
```

```
#<CIRCLE 21EADBFB>

CL-USER 5 > (set-shape w c)
#<WINDOW 21F18F5F>

CL-USER 6 > (redraw w)
#<WINDOW 21F18F5F>
```

Příklad: kreslení ve třídě `picture`

Budeme pokračovat kreslením obrázků, tedy instancí třídy `picture`. Jak už víme, je třeba definovat metodu pro zprávu `draw`.

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

Metoda prochází všechny prvky obrázku `pic` od posledního k prvnímu (díky funkci `reverse`) a každému posílá zprávu `draw`. Metoda by tedy opravdu měla vykreslit všechny prvky obrázku, přičemž objekty, které jsou v seznamu prvků obrázku vpředu, by měly překrývat objekty více vzadu.

Příklad: test kreslení obrázků

Vyzkoušejme kreslení obrázku na příkladě. Vytvoříme instanci třídy `picture`, která bude obsahovat několik soustředných kruhů se střídajícími se barvami. Podobně jako v příkladě 6 si na to napíšeme pomocnou funkci:

```
(defun make-bulls-eye (x y radius count)
  (let ((result (make-instance 'picture))
        (items '())
        (step (/ radius count))
        (blackp t)
        circle)
    (dotimes (i count)
      (setf circle (set-filledp
                    (set-color
                     (set-radius (make-instance 'circle)
                                 (- radius (* i step)))
                     (if blackp :black :light-blue))
                    t))
      (set-y (set-x (center circle) x) y)
      (setf items (cons circle items)
```

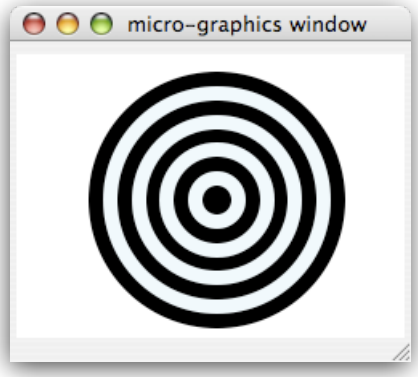
```
            blackp (not blackp)))
    (set-items result items)))
```

Funkce `make-bulls-eye` nejprve vytvoří obrázek (instanci třídy `picture`), pak v cyklu vytvoří zadaný počet kruhů, nastaví jim potřebné parametry a shromáždí je v seznamu. Tento seznam pak nastaví jako seznam prvků obrázku. Vytvořený obrázek vrátí jako výsledek.

Test:

```
CL-USER 17 > (setf w (make-instance 'window))
#<WINDOW 21C94ADF>

CL-USER 18 > (setf eye (make-bulls-eye 125 90 80 9))
#<PICTURE 200E038F>

CL-USER 19 > (set-shape w eye)
#<WINDOW 21C94ADF>

CL-USER 20 > (redraw w)
#<WINDOW 21C94ADF>
```

Výsledné okno je na Obrázku 6. Kruhy v obrázku jsou vyplněné (mají nastaveno



Obrázek 6: Terč

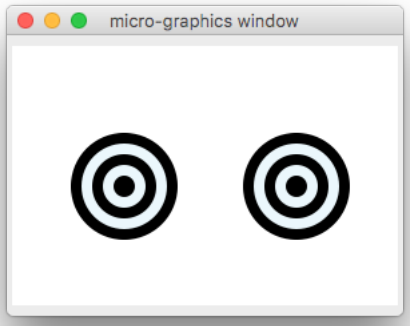
`filledp` na `t`), výsledného efektu je dosaženo jejich překrytím.

Příklad: druhý test kreslení obrázků

Jak těžké nyní bude nakreslit dva terče vedle sebe? Podívejme se na to:

```
CL-USER 26 > (setf w (make-instance 'window))
#<WINDOW 200E007F>

CL-USER 27 > (setf eye1 (make-bulls-eye 84 105 40 5))
#<PICTURE 200D248B>

CL-USER 28 > (setf eye2 (make-bulls-eye 213 105 40 5))
#<PICTURE 200BC66B>

CL-USER 29 > (setf pic (make-instance 'picture))
#<PICTURE 200A1D4B>

CL-USER 30 > (set-items pic (list eye1 eye2))
#<PICTURE 200A1D4B>

CL-USER 31 > (set-shape w pic)
#<WINDOW 200E007F>

CL-USER 32 > (redraw w)
#<WINDOW 200E007F>
```

A výsledek je na Obrázku 7. Samozřejmě i na vytvoření dvou terčů vedle sebe



Obrázek 7: Dva terče vedle sebe

bychom si mohli napsat funkci.

Příklady k této přednášce obsahují vylepšenou funkci `make-bulls-eye`.

Příklad: kreslení polygonů

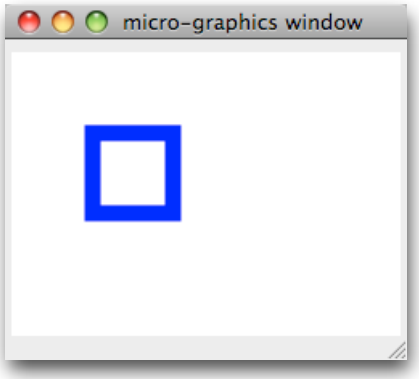
K třídě `polygon` přidáme nové vlastnosti, které budou obsahovat informace potřebné ke kreslení. Metodu `draw` a pomocné metody navrhneme podobně jako u třídy `circle` (podrobnosti ve zdrojovém kódu k přednášce).

Příklad: zkouška polygonu

Jednoduchý příklad práce s polygonem:

```
CL-USER 1 > (setf w (make-instance 'window))
#<WINDOW 200A5D9F>

CL-USER 2 > (setf p (make-instance 'polygon))
#<POLYGON 216FBD3B>

CL-USER 3 > (set-items
              p
              (list (move (make-instance 'point) 50 50)
                    (move (make-instance 'point) 100 50)
                    (move (make-instance 'point) 100 100)
                    (move (make-instance 'point) 50 100)))
#<POLYGON 216FBD3B>

CL-USER 4 > (set-color (set-thickness p 10) :blue)
#<POLYGON 216FBD3B>

CL-USER 5 > (set-shape w p)
#<WINDOW 200A5D9F>

CL-USER 6 > (redraw w)
#<WINDOW 200A5D9F>
```

Pokus by měl skončit stejně jako na Obrázku 8.



Obrázek 8: Modrý čtverec

Funkce na vytvoření modrého čtverce:

```
(defun make-blue-square ()
  (let ((p (make-instance 'polygon)))
    (set-items
     (set-color (set-thickness p 10) :blue)
     (list (move (make-instance 'point) 50 50)
           (move (make-instance 'point) 100 50)
           (move (make-instance 'point) 100 100)
           (move (make-instance 'point) 50 100)))))
```

A její použití:

```
CL-USER 8 > (setf sq (make-blue-square))
#<POLYGON 200A3F13>

CL-USER 9 > (setf w (set-shape (make-instance 'window) sq))
#<WINDOW 200B157B>

CL-USER 10 > (redraw w)
#<WINDOW 200B157B>
```

Příklad: kreslení bodů

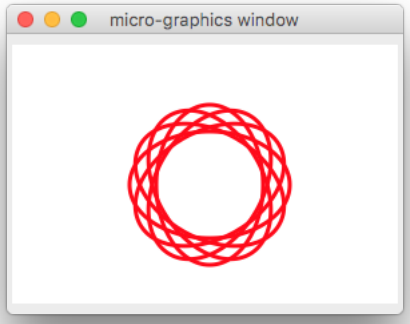
Body kreslíme jako malá kolečka. Podrobnosti najdete ve zdrojovém kódu k této přednášce.

Otázky a úkoly na cvičení

1. Přidejte do třídy `triangle` z minulých cvičení metody `move`, `scale`, `rotate` pro geometrické transformace.
2. Podobně jako u základních tříd této kapitoly dodejte do třídy `triangle` vlastnosti a metody používané při kreslení.
3. Definujte třídy `full-shape` a `empty-shape`. Instance třídy `full-shape` by měly představovat geometrické objekty, které vyplní celou rovinu. U třídy `empty-shape` to budou naopak objekty neobsahující žádný bod. Pro tyto třídy definujte všechny metody, které jsme v této kapitole definovali pro ostatní třídy, pokud to má smysl. Má například smysl definovat metodu `draw` pro třídu `empty-shape`? Jak tuto metodu definovat pro třídu `full-shape`?
4. Doplňte do třídy `ellipse` vlastnosti potřebné ke kreslení.
5. Definujte ve třídě `ellipse` metody pro kreslení a otestujte je.
6. Definujte metody `move`, `rotate` a `scale` pro třídu `ellipse`.

7. Napište funkci podobnou funkci `make-bulls-eye` pro elipsy. Ke změně velikosti elipsy ale použijte zprávu `scale` a k posunutí do bodu `x y` zprávu `move`.

8. Napište funkci, která vrátí instanci třídy `picture`, která se vykreslí zhruba jako na Obrázku 9. K otočení elipsy použijte zprávu `rotate`. V případě po-



Obrázek 9: Několik elips

třeby třídu `picture` upravte.