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

2. Zapouzdření

verze z 30. září 2020

1 Princip zapouzdření

Začneme několika příklady.

Příklad: problém s nastavováním slotu

Vytvořme nejprve novou instanci třídy `point`:

```
CL-USER 1 > (setf pt (make-instance 'point))
#<POINT 21817363>
```

a předpokládejme, že na nějakém místě programu omylem nastavíme hodnotu slotu `x` této instance na `nil`:

```
CL-USER 2 > (setf (slot-value pt 'x) nil)
NIL
```

Po nějaké době, na jiném místě našeho programu, pošleme objektu `pt` zprávu `r`. (Než budete číst dál, zkuste odhadnout, co přesně se stane a proč.)

Dojde k chybě:

```
CL-USER 3 > (r pt)

Error: In * of (NIL NIL) arguments should be of type NUMBER.
  1 (continue) Return a value to use.
  2 Supply new arguments to use.
  3 (abort) Return to level 0.
  4 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :? for
other options
```

Dostali jsme se do chybového stavu, k němuž došlo po zaslání zprávy `r` objektu `pt`. Hlášení o chybě, „`In * of (NIL NIL) arguments should be of type`

`NUMBER`“, je třeba číst takto: *při volání funkce `*` s argumenty `(NIL NIL)` došlo k chybě, protože argumenty volání nejsou čísla.* Jinými slovy, pokoušíme se násobit symbol `nil`.

Než chybový stav opustíme (což se v LispWorks, jak víme, dělá napsáním `:a`), můžeme jej využít k nalezení příčiny chyby. Pomocí debuggeru LispWorks můžeme zjistit, že k chybě došlo, protože bod má ve slotu `x` místo čísla symbol `nil`. Jak a kdy se tam symbol dostal, ovšem nezjistíme. Mohlo to být na libovolném místě programu, ve kterém slot `x` nastavujeme, o několik řádků výše, nebo o hodinu dříve. Tady může začít hledání, které může u velkých a příliš spletitých programů trvat dlouho.

Příklad: polární souřadnice problém nemají

Porovnejme tuto situaci s následující. Nejprve ale opravme náš objekt `pt` a nastavme jeho slot `x` na nějakou přípustnou hodnotu:

```
CL-USER 4 > (setf (slot-value pt 'x) 1)
1
```

Teď se pokusme udělat podobnou chybu s tím rozdílem, že nyní nastavíme na `nil` jednu z polárních souřadnic bodu `pt`, řekněme úhel. Co se stane?

```
CL-USER 5 > (set-phi pt nil)

Error: In COS of (NIL) arguments should be of type NUMBER.
  1 (continue) Return a value to use.
  2 Supply a new argument.
  3 (abort) Return to level 0.
  4 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :? for
other options
```

Chybové hlášení sice není příliš srozumitelné (i když jeho smysl už chápeme — došlo k pokusu počítat kosinus z hodnoty `nil`), ale pomocí debuggeru jsme schopni velmi rychle odhalit příčinu chyby, kterou je zaslání zprávy `set-phi` s nepřípustnou hodnotou argumentu.

Rozdíl mezi uvedenými dvěma případy: zatímco v tomto jsme byly upozorněni na problém v momentě, kdy jsme jej způsobili, v předchozím proběhlo nastavení nepřípustné hodnoty bez povšimnutí a upozorněni jsme byli až na druhotnou chybu — tedy chybu způsobenou předchozí chybou.

Přitom, z pohledu zvenčí by člověk řekl, že koncepčně není mezi uvedenými případy podstatný rozdíl; koneckonců, ať se jedná o kartézské nebo polární, vždy jsou to prostě jen souřadnice. Rozdíl je v tom, jak jsou tyto případy implementovány

(v jednom se hodnota nastavuje přímo, ve druhém zasíláním zprávy a nějakým výpočtem). Chyba je tedy na straně programátora.

Závěr: není dobré umožnit uživateli nastavovat hodnoty slotů v objektech bez kontroly jejich konzistence.

Poznamenejme, že ve staticky typovaném jazyce by uvedený problém vůbec nenastal, protože kompilátor by nedovolil do proměnné číselného typu uložit nečíselnou hodnotu. To je sice pravda, ale pro příklad, kdy by uložit nepřípustnou hodnotu do slotu umožnil i kompilátor staticky typovaného jazyka, nemusíme chodit daleko. Například u instancí třídy `circle` má smysl jako hodnoty slotu `radius` nastavovat pouze nezáporná čísla. Pokus uložit do tohoto slotu v průběhu programu vypočítané záporné číslo ale žádný kompilátor neodhalí.

Příklad: problém se změnou implementace

Předpokládejme, že změníme reprezentaci geometrických bodů tak, že místo kartézských souřadnic budeme ukládat souřadnice polární. Nová definice třídy `point` by vypadala takto:

```
(defclass point ()
  ((r :initform 0)
   (phi :initform 0)))
```

Položme si otázku: Co všechno bude nutno změnit v programu, který tuto třídu používá? Odpověď je poměrně jednoduchá. Pokud změníme definici metod `r`, `phi` a `set-r-phi` následujícím způsobem:

```
(defmethod r ((point point))
  (slot-value point 'r))

(defmethod phi ((point point))
  (slot-value point 'phi))

(defmethod set-r-phi ((point point) r phi)
  (setf (slot-value point 'r) r
        (slot-value point 'phi) phi)
  point)
```

nebudeme muset v programu měnit už žádné jiné místo, ve kterém pracujeme s polárními souřadnicemi bodů. Horší to bude s kartézskými souřadnicemi. S těmi jsme dosud pracovali pomocí funkce `slot-value`. Všechna místa programu, na kterých je napsáno něco jako jeden z těchto čtyř výrazů:

```
(slot-value pt 'x)
(slot-value pt 'y)
(setf (slot-value pt 'x) value)
(setf (slot-value pt 'y) value)
```

bude třeba změnit. Budeme tedy muset projít celý (možná dost velký) zdrojový kód programu a všude, kde to bude potřeba, provést příslušnou úpravu.

Pokud používáme k práci s objekty mechanismus zasílání zpráv a nepřistupujeme k jejich vnitřním datům přímo, bude náš program lépe připraven na změny vnitřní reprezentace dat.

Příklad: jednoduchost rozhraní

Představme si, že píšeme uživatelskou dokumentaci ke třídám `point` a `circle`, které jsme zatím naprogramovali. Při našem současném řešení bychom museli v dokumentaci popisovat zvlášť práci s kartézskými a polárními souřadnicemi bodů, protože s každými se pracuje jinak: Kartézské souřadnice se zjišťují pomocí funkce `slot-value` s druhým parametrem rovným symbolu `x` nebo `y`, zatímco ke čtení souřadnic polárních používáme zprávy `r` a `phi`.

Podobně, ke čtení poloměru kružnice používáme funkci `slot-value` s druhým parametrem rovným symbolu `radius`. Výsledkem by bylo, že by si uživatel musel pamatovat dvojí způsob práce s našimi objekty, což by mělo obvyklé nepříjemné důsledky (musel by častěji otvírat dokumentaci, asi by dělal více chyb a podobně), které by se zmnohonásobily u většího programu (který by neobsahoval dvě třídy, ale sto, které by obsahovaly mnohem více slotů a metod atd.).

V našem případě si musí uživatel našich tříd pamatovat, která data se čerpají přímo ze slotů a která jsou vypočítaná a získávají se zasláním zprávy. Když odhlédneme od toho, že tato vlastnost dat se může časem změnit (viz předchozí příklad), nutíme uživatele, aby se zabýval detaily, které pro něj nejsou podstatné.

Závěr: Při návrhu třídy je třeba myslet na jejího uživatele a práci mu pokud možno co nejvíce zpříjemnit a usnadnit.

Uvedené tři příklady nás motivují k pravidlu zvanému *princip zapouzdření*, které budeme vždy důsledně dodržovat.

Princip zapouzdření

Hodnoty slotů objektu smí přímo číst a měnit pouze metody tohoto objektu. Ostatní kód smí k těmto hodnotám přistupovat pouze prostřednictvím zpráv objektu zasílaných.

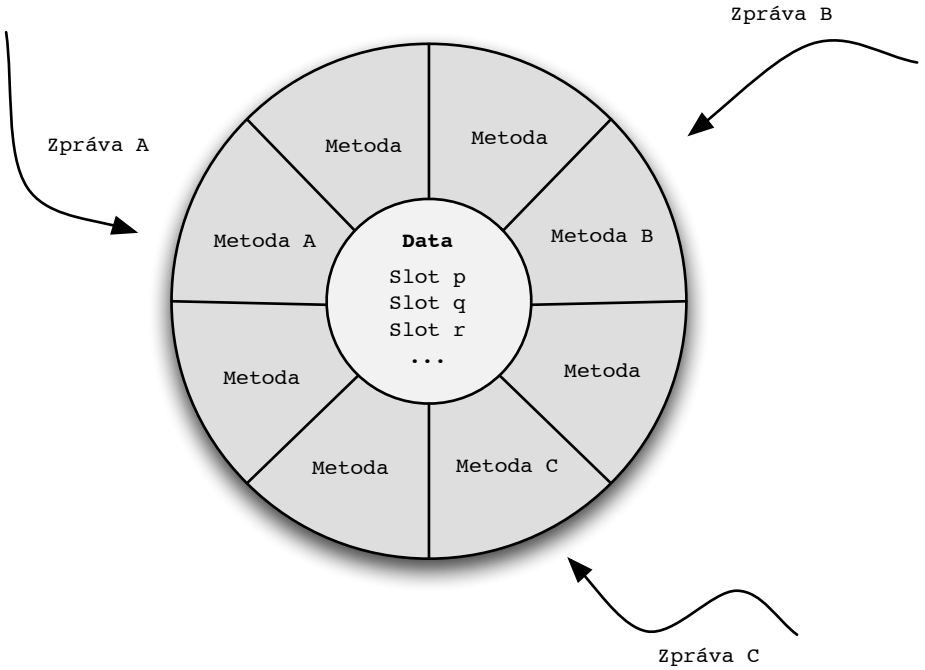
K principu zapouzdření nás motivovaly tři základní důvody. Zopakujme si je:

1. Uživateli je třeba zabránit modifikovat vnitřní data objektu, protože by jej mohl uvést do nekonzistentního stavu.

2. Změna vnitřní reprezentace dat objektu by měla uživateli přinášet co nejmenší komplikace.

3. Rozhraní objektů by mělo být co nejjednodušší a nejsnadněji použitelné. Uživatele nezajímají implementační detaily.

Data uvnitř objektu smíme tedy měnit pouze pomocí zpráv objektu zasílaných. Momentální hodnota dat v objektu je také nazývána jeho *vnitřním stavem*. Princip zapouzdření je znázorněn na Obrázku 1.



Obrázek 1: Princip zapouzdření

Nepřímý přístup k hodnotám slotů přes metody umožňuje kontrolovat správnost nastavované hodnoty, čímž se zabrání problémům ukázaným v prvním příkladě. Ukážeme si to v následujících příkladech.

2 Úprava tříd `point` a `circle`

Upravme tedy definice našich tříd `point` a `circle`, aby byly v souladu s principem zapouzdření.

Příklad: třída `point` s přístupovými metodami

Nová definice třídy `point` (metody pracující s polárními souřadnicemi zde neuvádíme, protože žádnou změnu nevyžadují):

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

(defmethod x ((point point))
  (slot-value point 'x))

(defmethod y ((point point))
  (slot-value point 'y))

(defmethod set-x ((point point) value)
  (unless (typep value 'number)
    (error "x coordinate of a point should be a number"))
  (setf (slot-value point 'x) value)
  point)

(defmethod set-y ((point point) value)
  (unless (typep value 'number)
    (error "y coordinate of a point should be a number"))
  (setf (slot-value point 'y) value)
  point)
```

Příklad: třída `circle` s přístupovými metodami

Nová definice třídy `circle`. U slotu `radius` budeme postupovat stejně jako u slotů `x` a `y` třídy `point`. Zveřejníme jeho obsah tím, že definujeme zprávy pro čtení a zápis jeho hodnoty:

```
(defmethod radius ((c circle))
  (slot-value c 'radius))

(defmethod set-radius ((c circle) value)
  (when (< value 0)
    (error "Circle radius should be a non-negative number"))
  (setf (slot-value c 'radius) value)
  c)
```

U slotu `center` nedovolíme nastavování hodnoty a dáme mu pouze možnost čtení:

```
(defmethod center ((c circle))
  (slot-value c 'center))
```

Různé změny u kruhu tak bude možné dělat pomocí jeho středu, například nastavit polohu:

```
CL-USER 9 > (make-instance 'circle)
#<CIRCLE 200CB05B>

CL-USER 10 > (set-x (center *) 10)
10
```

3 Třída `picture`

Další využití zapouzdření ukážeme na příkladě třídy `picture`. Třída umožňuje spojit několik grafických objektů do jednoho a usnadnit tak operace prováděné na všech těchto objektech současně.

Příklad: třída `picture` s typovou ochranou

Instance třídy `picture` budou obsahovat seznam podřízených grafických objektů:

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

Metody ke čtení a nastavování hodnoty slotu `items`:

```
(defmethod items ((pic picture))
  (slot-value pic 'items))

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

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

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

Metoda `set-items` podobně jako předtím například metoda `set-radius` třídy `circle` nejprve testuje, zda jsou nastavovaná data konzistentní, v tomto případě, zda jsou všechny prvky seznamu `items` správného typu (že proměnná `items` obsahuje seznam, otestuje makro `dolist` v metodě `check-items` — můžete vyzkoušet sami):

```
CL-USER 1 > (setf list (list 0 0 0))
(0 0 0)

CL-USER 2 > (setf pic (make-instance 'picture))
#<PICTURE 200E83CB>

CL-USER 3 > (set-items pic list)

Error: Invalid picture item type.
  1 (abort) Return to level 0.
  2 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :? for
other options
```

Pokud z této chybové hlášky nepochopíme, o jakou chybu jde, můžeme si opět pomoci debuggerem LispWorks.

Snažíme se dodržovat osvědčenou programátorskou zásadu nepsat neúměrně dlouhé funkce (v našem případě metody); každá by se měla věnovat řešení jednoho problému. Proto jsme část funkčnosti metody `set-items`, která se zabývá kontrolou typu hodnot v seznamu, izolovali a přemístili do jiné metody (se kterou jsme pak udělali totéž).

Z tohoto kroku budeme těžit už na této přednášce a příště kapitole se nám na nečekaném místě také vyplatí. Z toho bychom si měli vzít příklad:

> Dodržováním obecných programátorských zásad vzniká zdrojový kód, ve kterém se v budoucnu snadněji dělají změny.

Všimněte si, že metodu `check-item` bude třeba přepracovat, kdykoliv definujeme novou třídu grafických objektů. To je jistě nešikovné. Nápravu sjednáme později.

Takto definovaná třída `picture` bude fungovat správně, ale nebude ještě dostatečně odolná vůči uživatelským chybám. Uživatel má stále ještě možnost narušit konzistenci dat jejích instancí. Jak? To si ukážeme v dalším příkladě.

Příklad: problém třídy `picture`

Pokračujme tedy s testováním třídy `picture`. Vložme do proměnné `list` seznam složený z grafických objektů a uložme jej jako seznam prvků již vytvořenému obrázku `pic`:

```
CL-USER 5 > (setf list (list (make-instance 'point) (make-instance
'circle)))
(#<POINT 2008A1DB> #<CIRCLE 2008A1BF>)

CL-USER 6 > (set-items pic list)
(#<POINT 2008A1DB> #<CIRCLE 2008A1BF>)
```

Jaké jsou prvky obrázku `pic`?

```
CL-USER 7 > (items pic)
(#<POINT 2008A1DB> #<CIRCLE 2008A1BF>)
```

To nás jistě nepřekvapí. Nyní upravme seznam `list`:

```
CL-USER 8 > (setf (first list) 0)
0
```

Co bude nyní tento seznam obsahovat?

```
CL-USER 9 > list
(0 #<CIRCLE 2008A1BF>)
```

A co bude nyní v seznamu prvků obrázku `pic`?

```
CL-USER 10 > (items pic)
(0 #<CIRCLE 2008A1BF>)
```

(Jste schopni vysvětlit, co se stalo? Než budete pokračovat dále, je to třeba pochopit.) Obrázek `pic` nyní obsahuje nekonzistentní data.

Příklad: úprava třídy `picture`

Abychom problém odstranili, změníme metodu `items` tak, aby nevracela seznam uložený ve slotu `items`, ale jeho kopii. Podobně, při nastavování hodnoty slotu `items` v metodě `set-items` do slotu uložíme kopii uživatelem zadaného seznamu. Tímto dvojím opatřením zařídíme, že uživatel nebude mít k seznamu v tomto slotu přístup, pouze k jeho prvkům.

```
(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)
```

Není třeba dodávat, že pokud uživatel nedodrží princip zapouzdření, budou tato bezpečnostní opatření neúčinná.

Několik testů:

```
CL-USER 12 > (setf list (list (make-instance 'point) (make-instance
'circle)))
(#<POINT 200D7A23> #<CIRCLE 200D7A07>)

CL-USER 13 > (set-items pic list)
(#<POINT 200D7A23> #<CIRCLE 200D7A07>)

CL-USER 14 > (setf (first list) 0)
0

CL-USER 15 > (setf (second (items pic)) 0)
0

CL-USER 16 > (items pic)
(#<POINT 200D7A23> #<CIRCLE 200D7A07>)
```

Vidíme, že nyní je všechno v pořádku — ani následná editace seznamu posílaného jako parametr zprávy `set-items`, ani editace seznamu vráceného zprávou `items` nenaruší vnitřní data objektu `pic`.

U dalších tříd už nebudeme ochranu hodnot slotů dělat tak důsledně, jako zde. Bude to hlavně proto, aby se zdrojový kód příliš nenatahoval a byl stále dostatečně srozumitelný. V praxi mají na rozhodnutí do jaké míry hlídat konzistenci vnitřního stavu objektu vliv různé faktory.

4 Vlastnosti

V předchozích příkladech jsme uvedli několik zpráv sloužících ke čtení a zápisu dat objektů. V souladu s principem zapouzdření tyto zprávy nezávisí na tom, jakým způsobem jsou data v objektech uložena (např. souřadnice `x` a `y` u bodů jsou uložena ve slotech, zatímco souřadnice `r` a `phi` nikoliv).

Datům objektů, která můžeme zasíláním zpráv číst a nastavovat, říkáme *vlastnosti objektů* (anglicky *properties*). Každá vlastnost má své jméno, od kterého jsou odvozena jména příslušných zpráv. Zpráva pro čtení vlastnosti jménem ***property*** se jmenuje ***property***, obvykle nemá žádný parametr a v reakci na ni by objekt měl vrátit hodnotu této vlastnosti. Jméno zprávy pro nastavení této vlastnosti je `set-`***property***. Tato zpráva má obvykle jeden parametr — novou hodnotu vlastnosti. Vrací modifikovaný objekt.

Některé vlastnosti není zvenčí povoleno měnit. U nich není k dispozici zpráva pro jejich nastavování. Objekt samotný takovouto vlastnost měnit může, obvykle tak činí pomocí `slot-value`.

Příklad

Pro třídy `point`, `circle` a `picture` jsme zatím definovali následující vlastnosti: `x`, `y`, `r`, `phi` pro třídu `point`; `center` a `radius` pro třídu `circle` (první je jen ke čtení) a `items` pro třídu `picture`.

5 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`. Popis těchto funkcí spolu s popisem ostatních funkcí rozhraní knihovny najdete v jiném dokumentu.

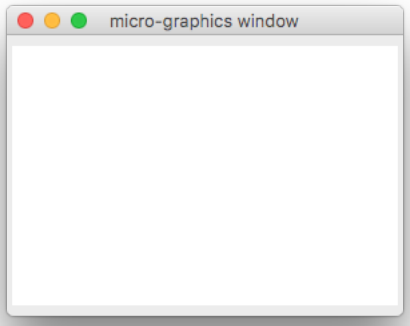
Příklad: použití knihovny `micro-graphics`

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 2.



Obrázek 2: 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.

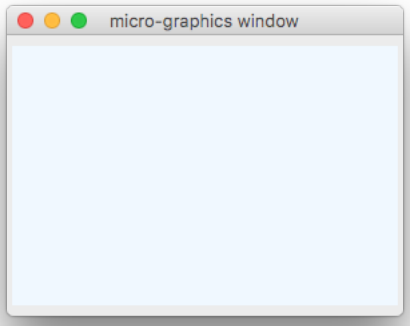
Pomocí funkce `mg:get-param` můžeme zjistit přednastavené kreslicí parametry okna:

```
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 3).



Obrázek 3: 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
```

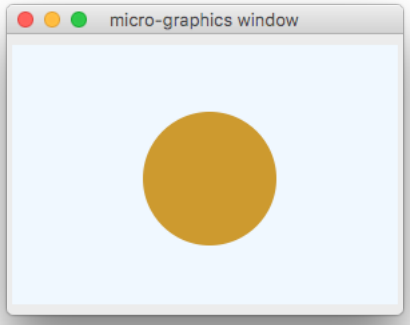
```
CL-USER 2 > :aliceblue
:ALICEBLUE
```

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 4).



Obrázek 4: Okno knihovny `micro-graphics` po nakreslení kruhu

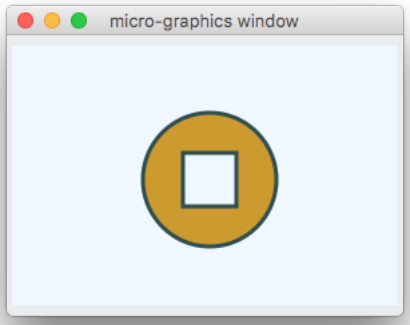
Nyní ještě změníme parametr `:foreground` a zkusíme pomocí funkce `mg:draw-polygon` nakreslit čtverec:

```
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 5.



Obrázek 5: Okno knihovny `micro-graphics` po nakreslení dvou kruhů a dvou polygonů

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. 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 mg-window ((window window))
  (slot-value window 'mg-window))

(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 `mg-window` je pouze ke čtení.

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 (mg-window 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 vykreslit do zadaného okna knihovny `micro-graphics`. Obsluhu této zprávy bude tedy třeba definovat pro všechny třídy grafických objektů.

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 pro určování vzhledu vykresleného 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. vykreslení 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 vykreslení nazveme `do-draw`. Zde je metoda pro tuto zprávu:

```
(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:

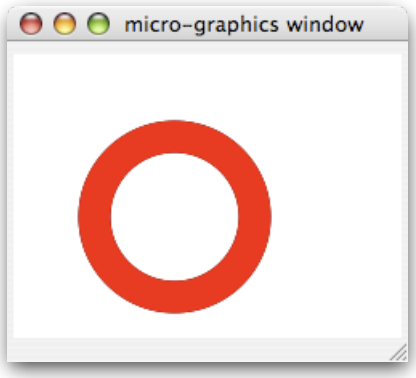
```
CL-USER 1 > (setf w (make-instance 'window))
#<WINDOW 217F04BF>

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 6. Kruh v okně můžeme kdykoli překreslit 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)))
```



Obrázek 6: Červené kolečko

```
    (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>
```

Otázky a úkoly na cvičení

Úkoly k této kapitole budou ještě přidány.

1. Definujte třídu `polygon`, jejíž instance budou podobné instancím třídy `picture`, ale prvky vlastnosti `items` mohou být jen body. Instance budou reprezentovat polygony. Definujte i kreslení instancí třídy pomocí funkce `mg:draw-polygon` knihovny `micro-graphics`.

2. 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 a pro které to má smysl.

Úkolem následujících úloh je doplnit třídu `ellipse` z minulé kapitoly, aby se dala používat stejně jako třídy definované v této kapitole.

3. Přesvědčte se, že definice třídy `ellipse` splňuje podmínku principu zapouzdření. Pokud ne, upravte ji. Třída by měla mít čtyři vlastnosti: `focal-point-1`, `focal-point-2`, `major-semiaxis` a `minor-semiaxis`. První dvě by měly být jen ke čtení, druhé dvě i k zápisu. Dodejte také testy správnosti zapisovaných hodnot.

4. Upravte také metodu `to-ellipse` třídy `circle`, aby vyhovovala principu zapouzdření.

5. Podobně jako v příkladu pro třídu `circle` doplňte do třídy `ellipse` vlastnosti potřebné ke kreslení.

6. Podobně jako v příkladu pro třídu `circle` definujte ve třídě `ellipse` metody pro kreslení a otestujte je.