
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

6. Zpětná volání

verze z 23. října 2020

1 Zpětná volání v knihovně `micro-graphics`

Naše objektová grafická knihovna má jednu slabinu, se kterou jste se jistě už setkali. Než se pustíme do vylepšování knihovny, zaměříme se na odstranění slabiny. To uděláme na této přednášce.

Při experimentování s okny knihovny `micro-graphics` jsme rychle zjistili, že v nich nakreslené obrázky nikdy dlouho nevydrží. Operační systém (resp. jeho grafická část) totiž kreslí do oken tak, že jen rozsvítí pixely na obrazovce barvou, kterou se dozví od knihovny `micro-graphics`. O tom, co mají pixely dohromady znázorňovat, nic neví. Pokud tedy změníme rozměry okna nebo okno překryjeme jiným (tohle neplatí v Mac OS), pixely se přemažou jinou barvou. Systém si nepamatuje, jakou barvu měly původně nebo jakou mají mít při změně rozměru okna.

Operační systém (jeho grafická část) ovšem umí zpozorovat, že je potřeba okno překreslit, a umí o to požádat aplikaci, které okno patří. V našem případě to udělá tak, že řekne naší knihovně `micro-graphics`, která o tom informuje nás prostřednictvím tzv. *zpětných volání* (*callbacks*).

Zpětná volání knihovny `micro-graphics` jsou uživatelské (tj. námi napsané) funkce, které knihovna zavolá, aby náš program informovala, že došlo k situaci, na kterou může chtít reagovat.

Knihovna rozlišuje několik typů takových situací, například že je třeba překreslit okno nebo že uživatel do okna kliknul myší či stiskl klávesu, když bylo okno aktivní. Každý z těchto typů má své jméno a pro každý z nich můžeme definovat funkci, kterou knihovna zavolá, kdykoli k dané situaci dojde. Těmto funkcím říkáme zpětná volání.

Informace o různých typech zpětných volání knihovny `micro-graphics` se dozvíte postupně. Na této přednášce se budeme zabývat zpětnými voláními typu `:display`, který se týká funkcí, jež knihovna volá, když je třeba překreslit okno. Knihovna `micro-graphics` tyto funkce volá s jedním argumentem, a to odkazem na příslušné okno.

K zaregistrování zpětného volání nebo k jeho zrušení slouží funkce `mg:set-callback`, k zjištění aktuální hodnoty funkce `mg:get-callback`. Syntax funkce `mg:set-callback` je následující:

```
(mg:set-callback mg-window callback-type function)
  => nil

  mg-window: hodnota vrácená funkcí mg:display-window
  callback-type: symbol
  function: funkce nebo nil
```

Parametr *callback-type* určuje typ zpětného volání, které chceme nastavit nebo zrušit. Pokud je chceme nastavit, uvedeme v parametru *function* funkci, která bude novým zpětným voláním, pokud je chceme zrušit, uvedeme v tomto parametru hodnotu `nil`.

Syntax funkce `mg:get-callback`:

```
(mg:get-callback mg-window callback-type) => result

  mg-window: hodnota vrácená funkcí mg:display-window
  callback-type: symbol
  result: funkce nebo nil
```

Parametr *callback-type* určuje typ zpětného volání, které chceme získat. (Funkce `mg:get-callback` je určena pro zvláštní situace, obvykle ji k ničemu nepotřebujeme.)

Příklad: test zpětného volání `:display`

Tímto postupem zaregistrujeme do nového okna zpětné volání, které vytiskne daný text, kdykoliv je třeba okno překreslit:

```
CL-USER 3 > (setf mgw (mg:display-window))
#<MG-WINDOW 200CA85B>

CL-USER 4 > (mg:set-callback
              mgw
              :display
              (lambda (mgw)
                (format t "~%Překresli mě!")))
NIL
```

Nyní můžeme manipulovat s oknem a dívat se, co se tiskne do standardního výstupu (vyzkoušejte si to).

Tisk vypneme zrušením zpětného volání:

```
CL-USER 5 > (mg:set-callback mgw :display nil)
NIL
```

2 Překreslování oken po vnější změně

Nyní využijeme zpětného volání `:display` k automatickému překreslení okna po změně vyvolané z vnějšku.

Příklad

Upravíme třídu `abstract-window` tak, aby její instance (tj. jak instance třídy `window`, tak dalších tříd oken) měly instalovánu jako zpětné volání `:display` funkci, která zajistí překreslení okna kdykoliv o ně knihovna `micro-graphics` požádá. Instalaci provedeme při vytváření okna, tedy v metodě `initialize-instance`. Definice třídy bude stejná jako dříve:

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

Stejné zůstanou i definice metod na zjišťování a nastavování hodnot vlastností `shape` a `background` a metoda `redraw` na překreslování. Ostatní metody postupně přepíšeme.

Nyní napíšeme novou metodu `install-callbacks`, která nastaví oknu zpětná volání (zatím jen na překreslení):

```
(defmethod install-display-callback ((w abstract-window))
  (mg:set-callback (slot-value w 'mg-window)
                   :display (lambda (mgw)
                              (declare (ignore mgw))
                              (redraw w)))
  w)

(defmethod install-callbacks ((w abstract-window))
  (install-display-callback w)
  w)
```

(Řádek `(declare (ignore mgw))` nemá význam pro běh programu. Pouze potlačí upozornění překladače o nepoužité proměnné `mgw`.)

Tuto úpravu je možné si ihned vyzkoušet. Načtěte si zdrojový text k předchozí kapitole (soubor `05.lisp`) a vyhodnoťte definice metod `install-display-callback`

a `install-callbacks`. Následujícím kódem vytvoříme nové okno a vložíme do něj jednoduchý útvar (funkci `make-test-circle` si napíšeme bokem, protože ji budeme potřebovat v dalších příkladech):

```
(defun make-test-circle ()
  (move (set-radius (set-thickness (set-color
                                    (make-instance 'circle)
                                    :darkslategrey)
                                   5)
                    55)
        148
        100))
```

```
CL-USER 7 > (setf w (set-background
                     (set-shape (make-instance 'window)
                                (make-test-circle))
                     :ghostwhite))
#<WINDOW 20109733>
```

Jak už jsme zvyklí, okno se nevykreslí a pokud ho vykreslíme ručně zasláním zprávy `redraw`, obrázek po manipulaci s oknem brzy zmizí. Pokud ale do okna nainstalujeme zpětné volání na vykreslení:

```
CL-USER 8 > (install-callbacks w)
#<WINDOW 20109733>
```

bude se už překreslovat automaticky. (Všimněme si, jak jsme hezky využili dynamičnosti Lispu — nemuseli jsme program ukončit, zkompilovat a znovu spustit. Dokonce jsme nemuseli ani zavírat okno, jen jsme mu dodali novou funkčnost.)

Instalaci zpětného volání nyní zautomatizujeme zavoláním metody `install-callbacks` při vytváření nové instance:

```
(defmethod initialize-instance ((w abstract-window) &key)
  (call-next-method)
  (install-callbacks w)
  w)
```

Nyní se již budou všechna nově vytvořená okna sama překreslovat. S novou definicí třídy `abstract-window` by měly fungovat i všechny dříve napsané příklady.

3 Překreslení při změně okna

Teď se pustíme do řešení ambiciózněji úkolu: naučit okna automaticky se překreslovat při každé změně jejich obsahu. Začneme překreslováním i po změně barvy pozadí a nastavení grafického objektu (vlastnosti `shape`). Současně se naučíme jeden důležitý princip.

Změny vlastností `background` a `shape` okna zatím nevedou k jeho překreslení, protože nezpůsobují zpětné volání `:display`. V těchto případech se o překreslení musí okno postarat samo. Udělá to tak, že zavolá funkci `mg:invalidate` knihovny `micro-graphics`. Tato funkce zaznamená, že okno potřebuje překreslit, ale samotné překreslení neprovede. Jedná se o obvyklý postup, který grafické knihovny používají, aby se vyhnuly zbytečnému několikanásobnému překreslování okna po každé jeho změně (má to i jiné důvody).

`(mg:invalidate` *`mgw`***`)`** `=> nil`

`mgw:` hodnota vrácená funkcí `mg:display-window`

Oznámí knihovně `micro-graphics`, že okno *`window`* má neaktuální obsah. Knihovna pak ve vhodný moment poté, co naše práce s oknem skončí, zavolá zpětné volání `:display`. Funkci lze bez obav volat několikrát za sebou. Sama zajistí, že se zpětné volání vykoná pouze jednou.

Překreslování okna knihovny `micro-graphics`.

Okna zásadně nepřekreslujeme přímo, ale pomocí funkce `mg:invalidate`.

Pravidlo respektujeme nejen abychom zabránili zbytečnému vícenásobnému překreslování, ale také proto, že jindy než během zpětného volání `:display` nemusí jít do okna vůbec kreslit, případně to může vést k chybám.

Příklad: Překreslení okna po změně `background` a `shape`

Nejprve zapouzdříme volání funkce `mg:invalidate` do nové metody `invalidate`:

```
(defmethod invalidate ((w abstract-window))
  (mg:invalidate (slot-value w 'mg-window))
  w)
```

Tuto metodu pak zavoláme kdykoli je třeba:

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

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

Nyní se sami můžete podívat, že se okna automaticky překreslují jak při nastavení `background`, tak `shape`:

```
CL-USER 24 > (setf w (make-instance 'window))
#<WINDOW 21CC8377>

CL-USER 25 > (set-background w :khaki)
#<WINDOW 21CC8377>

CL-USER 26 > (set-shape w (make-test-circle))
#<WINDOW 21CC8377>
```

4 Překreslování při změnách objektů

Nakonec naučíme okna automatickému překreslení při jakékoliv změně obsažených objektů. Tento úkol bude poněkud náročnější, protože bude vyžadovat přepracování všech metod grafických objektů, po jejichž volání je potřeba je překreslit.

Především je ale třeba, aby objekty měly jak zařídit, že se okno, ve kterém jsou uloženy, dozví o jejich změně. K tomu se bude hodit, aby si objekty své okno pamatovaly.

Příklad: okno grafického objektu

Zavedeme každému grafickému objektu vlastnost `window`, ve které bude uloženo okno, jež objekt obsahuje.

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

(defmethod window ((s shape))
  (slot-value s 'window))

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

Při nastavování okna obrázku je třeba zařídit, aby bylo okno správně nastaveno i prvkům. Je to hezký příklad na volání zděděné metody:

```
(defmethod set-window ((p abstract-picture) w)
  (call-next-method)
  (send-to-items p 'set-window w))
```

Objektu je třeba okno nastavit, když je do něj ukládán. Změníme proto znovu metodu `do-set-shape` třídy `abstract-window`:

```
(defmethod do-set-shape ((w abstract-window) s)
  (setf (slot-value w 'shape) s)
  (set-window s w)
  (invalidate w))
```

A teď se pustíme do řešení našeho problému automatického překreslování okna po změně obsaženého objektu. Stále se budeme držet pravidla, že objekty nepřekreslujeme hned po jejich změně, ale nepřímo pomocí funkce `mg:invalidate` (zapouzdřené v metodě `invalidate` třídy `abstract-window`).

Příklad: překreslení po změně barvy

Začneme jednoduchým případem: metoda `set-color` třídy `shape` zatím pouze nastavuje příslušný slot:

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

To nám nyní již nestačí. Po nastavení slotu by objekt měl nějak nahlásit, že u něj došlo ke změně, která vyžaduje překreslení okna.

Především napíšeme novou metodu `change`, kterou budou objekty volat po každé své změně. Metoda provede všechno potřebné: podívá se, jestli objekt má nastavené okno, a pokud ano, zašle mu zprávu informující o jeho změně.

```
(defmethod change ((shape shape))
  (when (window shape)
    (ev-change (window shape) shape))
  shape)
```

Zprávu zasílanou objektem oknu jsme nazvali `ev-change`. Předpona „`ev`“ znamená *event* neboli *událost*. Události jsou důležitým prvkem našeho systému; budeme se jim věnovat později. Zatím nám stačí vědět, že jde o zvláštní druh zprávy.

Zpráva `ev-change` má jeden parametr, kterým je objekt, jenž zprávu zasílá. Okno bude tedy informováno nejen o tom, že se nějaká část jeho obsahu změnila, ale bude také vědět která.

Nyní je tedy třeba napsat metodu `ev-change` pro třídu `abstract-window`. Ta nemusí dělat nic jiného než zavolat metodu `invalidate`:

```
(defmethod ev-change ((w abstract-window) shape)
  (invalidate w))
```

Teď nám tedy zbývá změnit metodu `set-color` třídy `shape`, aby po provedení změny změnu ohlásila zasláním zprávy `change`:

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

Napsaný kód bychom opět měli hned vyzkoušet (nezapomínejme na to!). Třeba takto:

```
CL-USER 5 > (setf w (make-instance 'window))
#<WINDOW 2009F157>

CL-USER 6 > (set-shape w (make-test-circle))
#<WINDOW 217141CF>

CL-USER 7 > (set-color (shape w) :purple)
#<CIRCLE 20099E53>
```

Po posledním vyhodnocení by se mělo kolečko v okně přebarvit.

Později uvidíme, že zvolené řešení není zcela vyhovující, a ještě je upravíme.

Příklad: hlášení, že změna nastane

Někdy se oknu může hodit, aby se dozvědělo o změně objektu v momentě, kdy se změna teprve chystá (například pro zapamatování stavu objektu před změnou). Předchozí příklad jasně ukazuje, jak takovou věc naprogramovat například pro zprávu `set-color`. Jediný podstatný rozdíl bude, že okno v reakci na informaci o chystané změně neudělá nic. Třída `abstract-window` ale bude připravena na to, aby její potomci přepsáním příslušných metod nějakou netriviální reakci umožnili. Zde je tedy příslušná obecná úprava ve třídách `shape` a `abstract-window`:

```
(defmethod changing ((shape shape))
  (when (window shape)
```

```
    (ev-changing (window shape) shape))
  shape)

(defmethod ev-changing ((w abstract-window) shape)
  w)
```

A například metoda `set-color` třídy `shape` bude upravena takto:

```
(defmethod set-color ((shape shape) value)
  (changing shape)
  (setf (slot-value shape 'color) value)
  (change shape))
```

Stále ještě nejde o verzi, se kterou se spokojíme. Jak uvidíme v dalších příkladech, bude dobré metodu ještě změnit.

Příklad: překreslení po posunutí

Jak upravit metody `move`, `scale` a `rotate`? Snadno nahlédneme, že problém těchto metod je v tom, že jsou v potomcích třídy `shape` přepisovány. Například metoda `move` je ve třídě `shape` zatím definována tak, že nic nedělá:

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

a ve třídě `point` přepsána tak, aby vykonala konkrétní akci vhodnou pro body:

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

Nyní by se nám hodilo definovat tyto metody tak, aby ve třídě `shape` pomocí zpráv `changing` a `change` signalizovaly změnu grafického objektu a mezitím provedly akci specifickou pro příslušnou podtřídu.

Člověka by napadlo, že by se v této situaci hodilo místo funkce `call-next-method` použít nějakou jinou, která by nezavolala metodu implementovanou u předka, ale naopak u potomka třídy `shape`:

```
(defmethod move ((shape shape) dx dy)
  (changing shape)
  (call-previous-method)
```

```
  (change shape))

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

Tento způsob volání metod v opačném pořadí (od nejobecnější třídy k potomkům) není v objektově orientovaných jazycích používán (výjimkou je zajímavý programovací jazyk *Beta*, a také Common Lisp, jak se dozvíme na poslední přednášce). Běžné objektové jazyky takové možnosti neposkytují, proto se jimi nebudeme (kromě poslední přednášky) zabývat.

Rozumným řešením tohoto problému, které lze použít v běžných objektových jazycích, je definovat pomocné metody nového názvu, jež budou metodami původními ve třídě `shape` volány:

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

(defmethod move ((shape shape) dx dy)
  (changing shape)
  (do-move shape dx dy)
  (change shape))

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

Toto řešení použijeme pro všechny tři uvedené metody, takže kromě metody `do-move` definujeme pomocné metody `do-scale` a `do-rotate`.

Je zřejmé, že metody `move` a `do-move` hrají rozdílnou úlohu. Zatímco u metody `move` očekáváme, že ji uživatel bude volat, ale asi ji nebude v potomcích třídy `shape` přepisovat, u metody `do-move` to bude přesně obráceně: uživatel ji nebude nikdy volat (na to je tady metoda `move`), ale v případě, že bude definovat vlastního potomka třídy `shape`, bude ji možná potřebovat přepsat. Pro to by se v běžném objektovém jazyce použila pro metodu `move` ochrana typu *public* a pro metodu `do-move` typu *protected*.

Metody podobného použití už jsme programovali na předchozích přednáškách, jsou to například metody `set-mg-params`, `do-draw`, `do-check-items`.

Příklad: nastavení barvy naposledy

Inspirováni předchozím příkladem uděláme poslední změnu metody `set-color`. Ta se totiž rovněž dělí na dvě části: hlášení o změně a vlastní nastavení barvy. Přestože

v případě této metody se obě části dějí ve stejné třídě (na rozdíl od předchozího příkladu), je účelné je rozdělit do dvou metod. Poslední verze metody `set-color` bude tedy vypadat takto:

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

(defmethod set-color ((shape shape) value)
  (changing shape)
  (do-set-color shape value)
  (change shape))
```

Jako v předchozím příkladě, zpráva `set-color` je určena k volání uživatelem, zpráva `do-set-color` by mu měla být utajena. Je nachystána na případné přepsání v potomcích třídy `shape`, pokud by se v nich barva objektu ukládala jinak než do slotu nebo pokud by se kromě uložení barvy do slotu měla provést i další akce.

Otázky a úkoly na cvičení

1. Vyzkoušejte překreslování okna po zaslání zprávy `move` středu instance třídy `circle`. V čem je problém? Jak byste ho vyřešili? Je ještě nějaká třída, kterou je třeba upravit?

2. Vyzkoušejte, zda instance třídy `bulls-eye` z příkladu k minulé přednášce fungují i s verzí knihovny z této přednášky. Překreslují se správně za všech okolností? Navrhněte opravu případných nedostatků.

3. Napište třídu oken, jejíž instance budou mít následující vlastnost: po změně libovolného objektu v okně se pozadí okna přebarví na barvu podobnou barvě objektu. Podobnou barvu k dané barvě můžete vytvořit například touto funkcí (její definici nemusíte zkoumat; využívá možností LispWorks, nikoli Common Lispu):

```
(defun darken-color (color)
  (let ((color-spec (color:get-color-spec color)))
    (color:make-hsv (color:color-hue color-spec)
                    (color:color-saturation color-spec)
                    (/ (color:color-value color-spec) 2))))
```

4. Napište třídu oken, jejíž instance budou mít následující vlastnost: po změně libovolného objektu v okně se barva tohoto objektu změní na červenou.

5. Vylepšete řešení předchozího příkladu tak, aby se barva libovolného dříve změněného objektu vrátila na původní. V okně bude tedy vždy nejvýše jeden červený objekt, a to ten, který se změnil naposledy.

6. U příkladů, ve kterých jste definovali novou třídu oken, se zamyslete, zda bylo správné ji definovat jako potomka třídy `abstract-window`, nebo `window`. Zkuste pro novou třídu zformulovat kontrakt (invarianty, prekondice a postkondice) a zvážit, zda splňuje pravidla z minulé přednášky.

7. Napište třídu oken, jejíž instance budou mít následující vlastnost: před změnou libovolného objektu v okně se objeví dotaz, zda se má změna opravdu provést. Pokud uživatel zvolí možnost NE, dojde k chybě. Dialog s dotazem můžete vytvořit voláním

```
(capi:confirm-yes-or-no "Umožnit změnu?")
```

   Proč se dialog někdy objeví vícekrát?

   (U tohoto příkladu o kontraktu neuvažujte.)