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

8. Princip vlastnění

verze z 20. listopadu 2020

1 Problém nesamostatnosti objektů

Principy z minulé přednášky stačily k tomu, abychom mohli naprogramovat okna (obecně instance třídy `abstract-window`), která nějak reagují na kliknutí do svého obsahu. Je to dáno tím, že okna mohou reagovat na události posílané objekty uvnitř svého `shape`. Vysvětlili jsme si, že implementovat reakci na kliknutí do objektu přímo na úrovni tohoto objektu (tj. v metodě `mouse-down`) není obvykle vhodné, takže nám opravdu zbývají jen události.

To omezuje naše možnosti a nečiní zcela zadost základnímu principu **nezávislosti objektů** — objekty ke své plné funkčnosti potřebují okno, ve kterém jsou umístěny.

Kdybychom například chtěli kolečko s šipkou, které jsme minule programovali, chápat jako samostatnou a nezávislou jednotku, setkali bychom se s potížemi. Uložit například ne jedno, ale třeba dvě kolečka se šipkou do téhož okna by znamenalo změnit třídu okna a podstatnou část kódu napsat znovu. Kolečko s šipkou totiž **není nezávislá část programu**.

Na této přednášce tento problém vyřešíme a tím také dokončíme první verzi naší objektové grafické knihovny.

2 Vlastnické vztahy mezi objekty, delegát

V našem grafickém objektovém systému se kromě stromové hierarchie tříd založené na dědičnosti (tj. vztahu *potomek–předek*) setkáme ještě s hierarchií jinou, danou vztahem vlastnickým (tj. vztahem *prvek–kontejner*). Tento vztah se vytváří až za běhu aplikace, je méně formalizován a je volnější (ne všechny objekty mu musí podléhat) a dynamičtější (za běhu aplikace může prvek změnit svého vlastníka a podobně).

Například instance třídy `abstract-window` vlastní (lze říci i *obsahují*) jako vlastnost `shape` instanci třídy `shape`, která sama bývá instancí třídy `abstract-picture`, a tedy obvykle vlastní (obsahuje) jako vlastnost `items` své prvky, které opět mohou být instancemi třídy `abstract-picture` nebo jiné třídy. Polygony obsahují své body, kruh obsahuje svůj střed (vlastnost `center`). Tak se v každém okně vytváří stromová struktura objektů organizovaná podle vlastnických vztahů.

Vlastnický vztah můžeme využít k *předávání zodpovědnosti* za některé činnosti z podřízených objektů na jejich kontejnery. U grafických aplikací totiž často dochází k situacím, kdy objekt není sám schopen rozhodnout, jaká akce se má vyvolat následkem nějakého uživatelského zásahu (třeba kliknutí myši), a proto by bylo nešikovné mu tuto akci definovat jako metodu. Už jsme se zmínili minule, že definice nových metod by také znamenala nutnost definice nových tříd (například pro každé tlačítko v rámci okna). To by vedlo k nepřijatelnému rozrůstání zdrojového textu programu.

A také jsme si na příkladě elektrického vypínače vysvětlili, že by to nebylo v souladu s tím, jak věci fungují v realitě: elektrický vypínač na zdi nepotřebuje vědět, jestli slouží k rozsvěcení žárovky, zářivky, nebo třeba k zapínání elektrického topení. Pouze posílá signál, že byl zapnut, a o správnou reakci se postará někdo jiný.

Na rozdíl od minulé přednášky nebudou objekty v okně předávat zodpovědnost přímo oknu, ale jen svému bezprostřednímu vlastníku. Tím se vyřeší problém nesamostatnosti objektů popsaný na začátku přednášky.

Přímého vlastníka daného objektu nazveme jeho *delegátem*. Delegátovi bude objekt zasílat různé zprávy o změně svého vnitřního stavu a podobně. Delegát objektu bude zodpovědný za případné ošetření situací, které objekt sám neřeší. Zareaguje, jak potřebuje, a pokud je třeba, událost přepošle dále svému delegátovi.

U třídy `shape` delegát nahradí předchozí vlastnost `window`.

Každý objekt může a nemusí mít delegáta a v některých výjimečných případech, zejména když objekt nemá vlastníka (třeba u oken), může jeho delegátem být objekt, který jeho vlastníkem není.

Stran popsaného systému událostí mají tedy všechny objekty možnost mít delegáta, posílat mu události a přijímat události od objektů, jichž je sám delegátem. To se týká všech našich objektů, tedy grafických objektů i oken. Proto definujeme novou třídu `omg-object` (zkratka „`omg`“ neznamená nic jiného než „Object Micro-Graphics“), která bude předkem jak třídy `abstract-window`, tak třídy `shape`, a v ní funkčnost systému událostí definujeme. Původní definice ze tříd `abstract-window` a `shape` vypustíme.

Příklad: třída `omg-object` a delegát

Každý objekt bude mít vlastnost `delegate`, ve které bude umístěn jeho delegát (nebo `nil`, pokud objekt delegáta nemá). Jak uvidíme, třída `omg-object` se nám bude hodit i k dalším účelům. Obecně bude realizovat funkčnost společnou všem objektům našeho systému, ať už jsou to okna, nebo grafické objekty.

První definice třídy `omg-object` a vlastnosti `delegate`:

```
(defclass omg-object ()
  ((delegate :initform nil)))

(defmethod delegate ((obj omg-object))
```

```
  (slot-value obj 'delegate))

(defmethod set-delegate ((obj omg-object) delegate)
  (setf (slot-value obj 'delegate) delegate))
```

Příklad: okna a grafické objekty jsou omg-objekty

Rovnou upravíme třídy `abstract-window` a `shape`, aby byly potomky třídy `omg--object`. U třídy `shape` vypustíme slot `window` a příslušné metody.

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

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

Při vytváření nových objektů a nastavování jejich obsahu je třeba správně podřízeným objektům nastavovat delegáta. V budeme postupovat podobně jako dříve s nyní už smazanou vlastností `window`. Nezapomeneme přitom ani na střed kruhu a vrcholy polygonů (delegáta obecně nastavujeme ve třídě `compound-shape`).

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

(defmethod initialize-instance ((c circle) &key)
  (call-next-method)
  (set-delegate (center c) c)
  c)

(defmethod do-set-items ((shape compound-shape) value)
  (setf (slot-value shape 'items) (copy-list value))
  (send-to-items shape 'set-delegate shape))
```

V minulosti jsme také nastavovali okno prvkům složeného grafického objektu (obrázku), když bylo okno nastavováno jemu. Nic takového teď nemusíme dělat.

V následující části budeme podrobně mluvit o událostech. Zatím si jen řekněme, že zatímco na předchozích přednáškách objekty posílaly události svému oknu, nyní je budou posílat svému delegátovi.

Následující pravidlo je důležité kvůli udržení přehledu i ve složitém programu. Budeme se snažit je poctivě dodržovat.

Pravidlo zodpovědnosti

Každý objekt je zodpovědný za svůj obsah a obsah podřízených objektů. Sobě a jim tedy může posílat jakékoli zprávy. Naopak za objekty, které mu nepatří, zodpovědný není, a nijak s nimi nekomunikuje. Jediný povolený způsob komunikace s vnějšími objekty je zaslání události delegátovi. Smyslem události není delegátovi něco nařizovat, ale pouze ho informovat. Objekt nikdy nesmí předpokládat, že delegát na přijetí události nějak zareaguje.

Z logiky věci vyplývá, že žádný objekt by neměl sám nastavovat svou vlastnost `delegate`. Měl by to dělat výhradně nadřízený objekt, když se delegátem objektu stává.

Na události se podrobně podíváme teď.

3 Základní události

Základní informace o událostech jsou až na zavedení delegátů stejné jako z minula. Události se proti obecným zprávám vyznačují následujícími zvláštnostmi:

1. Událost objekt posílá výhradně svému delegátovi.
2. Účelem události je delegáta o něčem informovat, nikoli mu něco přikazovat.
3. Objekt tedy nesmí očekávat, že a jak delegát na přijetí události zareaguje (nemusel by udělat vůbec nic).

Jména událostí

Kvůli odlišení událostí od ostatních zpráv musí jejich jméno vždy začínat předponou `ev-`.

K zaslání události používá objekt stejně jako minule zprávu `send-event`. Základní metoda je nyní definována ve třídě `omg-object`. Její první varianta je velmi podobná původní metodě z minulé přednášky. Později ji ještě upravíme.

```
(defmethod send-event ((object omg-object) event &rest event-args)
  (let ((delegate (delegate object)))
    (when delegate
      (apply event delegate object event-args))
    object))
```

Jak je vidět, událost se vždy posílá delegátu objektu (pokud ho objekt má) a prvním argumentem je vždy objekt, takže delegát se vždy dozví, kdo mu událost posílá.

Základní události: `ev-changing` a `ev-change`

Události objekt posílá, když dochází nebo došlo k jeho viditelné změně, která vyžaduje překreslení okna. Události nemají kromě odesílatele žádné další argumenty. Příslušné metody jsou definovány ve třídě `omg-object` a zařizují poslání zpráv `changing` a `change` (protože tyto zprávy se posílají vždy při změně objektu a změnu podřízeného objektu chápeme i jako změnu majitele). Ty přepošlou událost dalšímu delegátovi. Jak bylo řečeno na začátku přednášky, tím se zařídí, že událost dostane vždy i okno. Metody `changing` a `change` ještě později vylepšíme, tady uvedeme jejich první verzi. Metody `ev-changing` a `ev-change` už zůstanou beze změny.

```
(defmethod ev-changing ((obj omg-object) sender)
  (changing obj))

(defmethod ev-change ((obj omg-object) sender)
  (change obj))

(defmethod changing ((object omg-object))
  (send-event object 'ev-changing))

(defmethod change ((object omg-object))
  (send-event object 'ev-change))
```

Zprávy `changing` a `change` tedy objekt dostává, kdykoli se změní jeho obsah. Samozřejmě si je nadále také sám posílá, když se mění on (tj. ne když se mění jeho vlastněný objekt). Například:

```
(defmethod set-x ((point point) value)
  (unless (typep value 'number)
    (error "x coordinate of a point should be a number"))
  (changing point)
  (do-set-x point value)
  (change point))
```

Metodu `change` přepisujeme ve třídě `abstract-window`:

```
(defmethod change ((w abstract-window))
  (call-next-method)
  (invalidate w))
```

Proti předchozí verzi volá zděděnou metodu (tj. metodu `change` třídy `omg-object`), aby okno posílalo událost případnému delegátovi. Metody `changing`, `ev-changing` a `ev-change` ze třídy vymažeme, protože jsou přesunuty do třídy `omg-object`.

Základní události: `ev-mouse-down`

Tuto událost objekt posílá, když do něj uživatel klikne myší. Událost má kromě odesílatele další tři argumenty: původní objekt, na který uživatel klikl, symbol značící tlačítko myši a bod určující v souřadnicích okna místo kliknutí. Zpráva má tedy o argument více, než na předchozí přednášce, aby se odlišil odesílatel od skutečně kliknutého objektu.

Odesílání události grafickým objektem po kliknutí, tedy v metodě `mouse-down` (všimněte si nového argumentu `shape` události):

```
(defmethod mouse-down ((shape shape) button position)
  (send-event shape 'ev-mouse-down shape button position))
```

Přeposílání události v metodě `ev-mouse-down` (tedy poté, co ji objekt přijal od podřízeného objektu):

```
(defmethod ev-mouse-down
           ((obj omg-object) sender clicked button position)
  (send-event obj 'ev-mouse-down clicked button position))
```

Zde si všimněte parametru `clicked`. Ten se přeposílá beze změny delegátovi, takže na libovolné úrovni (třeba až u okna) je stále známo, na který objekt se původně kliklo.

Původní metodu `ev-mouse-down` třídy `abstract-window` vymažeme.

Přeposílání, které jsme napsali pro události `ev-changing`, `ev-change` a `ev--mouse-down`, je u nich třeba vždy dodržovat. Formulujeme si to jako pravidlo.

Přeposílání událostí

Před a po změně objektu (tedy v metodách `changing` a `change`) a po kliknutí do objektu (tedy v metodě `mouse-down`) musí objekt vždy poslat příslušnou událost (`ev-changing`, `ev-change`, resp. `ev-mouse-down`).
Informace o změně a kliknutí, které od objektu delegát dostane prostřednictvím události, musí sám vždy přeposlat svému delegátovi.

Na toto pravidlo pamatujeme, především když některou z uvedených šesti metod přepisujeme v potomkovi třídy. Většinou to znamená nezapomenout volat zděděnou metodu.

4 Definice vlastních událostí

Události `ev-changing` a `ev-change` slouží k informování o obecné změně objektu, která vyžaduje překreslení okna. Událost `ev-mouse-down` zase k informování o obecném kliknutí a předává informaci o tlačítku myši a souřadnicích bodu kliknutí.

Někdy se hodí definovat vlastní událost, která sice může (ale nemusí) mít svůj původ ve změně objektu nebo kliknutí, ale vyjadřuje nějakou konkrétnější situaci. Jako příklad můžeme uvést třídu `button`, která je naprogramována jeko příklad k této kapitole. Instancemi třídy jsou obrázky, které mají fungovat jako tlačítka. Skládají se ze zarámovaného obdélníka a textu. Velikost tlačítka se sama upravuje podle rozměrů textu. Text se nastavuje a čte pomocí vlastnosti `button-text`.

Tlačítka samozřejmě po kliknutí myší posílají událost `ev-mouse-down`. Pokud ale uživatel klikne levým tlačítkem, posílají ještě novou událost `ev-button-click`. Smyslem této události není informovat o obecném kliknutí, ale o tom, že se má spustit akce, se kterou je tlačítko spojeno. Událost jednak neobtěžuje programátora nepotřebnými hodnotami (tlačítko, poloha) a jednak by ji v budoucnu mohlo tlačítko generovat i za jiných okolností, například když je tlačítko aktivní a uživatel stiskne mezerník.

Událost tlačítko posílá ze své metody `mouse-down`. Všimněte si, že opravdu nemá žádný parametr:

```
(defmethod mouse-down ((b button) mouse-button position)
  (call-next-method)
  (when (eql mouse-button :left)
    (send-event b 'ev-button-click))
  b)
```

Podobně je třeba přidávat události např. u grafických objektů, které zobrazují nějakou hodnotu, jako třeba počítadlo, display hodin a podobně. Po změně hodnoty by objekt neměl posílat jen událost `ev-change` (pokud vůbec vedla k viditelné změně) ale novou událost, která by se posílala jen po této změně a obsahovala vhodnější argumenty.

Události nového typu mohou také objekty posílat, když dojde k jinému vstupu od uživatele, než je kliknutí, například k dvojkliku, pohybu myši, stisku klávesy atd.

5 Přejmenování události

Pokud uživatel klikne na prvek obrázku, obrázek se o tom dozví v události `ev--mouse-down`. Chce-li obrázek zjistit, na který prvek uživatel klikl, může se podívat na parametr události, který obsahuje původní kliknutý objekt. Tak to děláme

například v první verzi příkladu kolečka s šipkou k této přednášce (jde o druhý parametr, zde pojmenovaný `origin`):

```
(defmethod ev-mouse-down ((p circle-with-arrow-1)
                          sender origin button position)
  (when (eql origin (cwa-arrow p))
    (move (cwa-circle p) 0 -10))
  (call-next-method))
```

Druhou možností je, aby odesílatel událost `ev-mouse-down` **přejmenoval**. Postupem, který popíšeme za chvíli, se stane, že metoda `mouse-down` odesílatele sice zavolá `send-event` s argumentem `ev-mouse-down`, ale skutečná událost, která se pošle, se bude jmenovat jinak (v příkladě níže bude nový název `ev-arrow-click`). Odesílatel si tedy myslí, že posílá událost `ev-mouse-down`, příjemce (delegát) ale dostane událost `ev-arrow-click`. Pokud se takto bude chovat jen jeden objekt, delegát tedy už nemusí zjišťovat, o který objekt jde.

Tak je to zařízeno ve druhé verzi příkladu kolečka s šipkou. Šipka ve své metodě `mouse-down` samozřejmě odesílá událost `ev-mouse-down`. Delegát (celý obrázek s šipkou a kolečkem jako prvky) ale dostane událost `ev-arrow-click`. Obrázek tedy nemusí zkoumat, kdo je odesílatelem události `ev-arrow-click`, protože jím může být jedině šipka. Zato ale — podle pravidla, které jsme si řekli dříve — musí manuálně událost `ev-mouse-down` přeposlat svému delegátovi:

```
(defmethod ev-arrow-click ((cwa circle-with-arrow-2)
                           sender origin button position)
  (move (cwa-circle cwa) 0 -10)
  (send-event cwa 'ev-mouse-down origin button position))
```

Výhody tohoto řešení se projeví především v situacích, kdy jako tlačítko má fungovat větší množství objektů. V příkladě s editorem polygonů, který je také naprogramován k této přednášce, jsou čtyři tlačítka, instance třídy `button` (viz předchozí část). Jak víme, tlačítka po kliknutí levým tlačítkem myši generují zvláštní událost `ev-button-click`. Ta je ovšem v příkladě editoru polygonů pro každé tlačítko přejmenována na zvláštní název. Tak může editor obsahovat pro kliknutí na tlačítko čtyři různé obsluhy:

```
(defmethod ev-closedp-click ((e polygon-editor) sender)
  (set-closedp (editor-polygon e)
               (not (closedp (editor-polygon e)))))

(defmethod ev-filledp-click ((e polygon-editor) sender)
  (set-filledp (editor-polygon e)
               (not (filledp (editor-polygon e)))))
```

```
(defmethod ev-color-click ((e polygon-editor) sender)
  (set-color (editor-polygon e)
             (random-color)))

(defmethod ev-clear-click ((e polygon-editor) sender)
  (set-items (editor-polygon e) '()))
```

Kdyby se události nepřejmenovávaly, všechna čtyři tlačítka by posílala stejnou událost `ev-button-click` a její obsluha by pak musela sama zjišťovat, které tlačítku událost poslalo:

```
(defmethod ev-button-click ((e polygon-editor) sender)
  (cond ((eql sender (closedp-button e))
         (set-closedp (editor-polygon e)
                      (not (closedp (editor-polygon e)))))
        ((eql sender (filledp-button e))
         (set-filledp (editor-polygon e)
                      (not (filledp (editor-polygon e)))))
        ((eql sender (color-button e))
         (set-color (editor-polygon e)
                    (random-color)))
        ((eql sender (clear-button e))
         (set-items (editor-polygon e) '()))))
```

Ještě musíme popsat, jak je přejmenování událostí zařízeno. Především je třeba si položit obvyklou principiální otázku: *Kdo za to bude zodpovědný?* Odpověď není příliš složitá. Delegát rozhoduje, jakou událost má odeslat který jeho podřízený objekt, když do něj např. klikne uživatel. Na druhou stranu, informace o tom, pod jakým jménem má objekt kterou událost poslat, je nejjednodušší uložit přímo do objektu.

Každý objekt (instance třídy `omg-object`) bude tedy obsahovat novou vlastnost `events`, ve které tyto informace budou obsaženy, a hodnotu vlastnosti bude nastavovat jeho delegát.

Vlastnost bude obsahovat seznam dvojic `(`***`událost . překlad`***`)`. Pokud tedy delegát bude např. chtít, aby objekt událost `ev-button-click` odeslal pod názvem `ev-clear-click`, uloží do jeho seznamu `events` pár `(ev-button-click . ev-clear-click)`.

Odeslání zařídí nová verze metody `send-event` třídy `omg-object`.

```
(defmethod event-translation ((obj omg-object) event)
  (cdr (find event (events obj) :key 'car)))
```

```
(defmethod send-event ((object omg-object) event &rest event-args)
  (let ((delegate (delegate object))
        (real-event (event-translation object event)))
    (when (and delegate real-event)
      (apply real-event delegate object event-args))
    object))
```

Metoda `event-translation` posílá objektu zprávu `event`. To je samozřejmě zpráva zjišťující hodnotu stejnojmenné vlastnosti. Práce s vlastností je napsána tak, aby s ní šlo dělat to, co je obvykle třeba: zjistit celý seznam (metoda `event`) a přidat, změnit, či odebrat jeden údaj.

```
(defmethod events ((obj omg-object))
  (slot-value obj 'events))

(defmethod add-event ((obj omg-object) event translation)
  (let ((new-list (remove event (events obj) :key 'car)))
    (setf (slot-value obj 'events)
          (cons (cons event translation)
                new-list))
    obj))

(defmethod remove-event ((obj omg-object) event)
  (setf (slot-value obj 'events)
        (remove event (events obj) :key 'car)))
```

Metoda `send-event` také nemusí událost poslat vůbec. To se stane jednak pokud objekt nemá žádného delegáta nebo pokud nemá událost v seznamu `events` uvedenu. Druhá možnost výhodná tím, že delegát není nucen implementovat obsluhu pro všechny události, které by kdy objekt mohl posílat. (Tak například můžeme uložit instanci třídy `button` do obecného obrázku bez obav, že dojde k chybě, že obrázek nerozumí zprávě `ev-button-click`; pokud událost nebude v seznamu `events` uložena, prostě se neodešle.)

Definici třídy `omg-object` rozšíříme o slot obsahující seznam událostí a jejich překladů. Každá nově vytvořená instance třídy `omg-object` vždy odesílá události `ev-changing`, `ev-change` a `ev-mouse-down`, a to pod původními názvy. Proto slot rovnou příslušným způsobem při vytvoření objektu naplníme. Tím dostaneme novou (ale stále ještě ne konečnou) verzi třídy:

```
(defclass omg-object ()
  ((delegate :initform nil)
   (events :initform '((ev-changing . ev-changing)
                       (ev-change . ev-change)
                       (ev-mouse-down . ev-mouse-down)))))
```

6 Odstranění přebytečných hlášení změn

Nejprve vyzkoušíme náš systém hlášení změn. Následující testy si můžete snadno vyzkoušet sami; mezi soubory k přednášce máte i kompletní zdrojový kód předposlední verze knihovny, což je ta, ve které se v našem výkladu právě nacházíme.

Vytvoříme pomocnou třídu `test-delegate`, jejíž instance budou přijímat zprávy o změně od jiného objektu a tisknout o nich informaci:

```
(defclass test-delegate (omg-object) ())

(defmethod ev-changing ((d test-delegate) sender)
  (format t "~%Proběhne změna.")
  d)

(defmethod ev-change ((d test-delegate) sender)
  (format t "~%Proběhla změna.")
  d)
```

Dále vytvoříme instanci této třídy a nastavíme ji jako delegáta nové instanci třídy `point`:

```
CL-USER 1 > (setf delegate (make-instance 'test-delegate))
#<TEST-DELEGATE 4020004A93>

CL-USER 2 > (setf pt (make-instance 'point))
#<POINT 4020008F23>

CL-USER 3 > (set-delegate pt delegate)
#<TEST-DELEGATE 4020004A93>
```

Nyní by objekt `delegate` měl při každé změně objektu `pt` vytisknout zprávu. Například:

```
CL-USER 4 > (set-x pt 5)

Proběhne změna.
Proběhla změna.
#<POINT 42002FF5DB>
```

Zkouška dopadla podle očekávání. Další zkouška:

```
CL-USER 5 > (set-r pt 10)
```

```
Proběhne změna.
Proběhne změna.
Proběhla změna.
Proběhla změna.
#<POINT 42002FF5DB>
```

Zde už něco nehraje. Ještě jedna zkouška:

```
CL-USER 6 > (setf c (make-instance 'circle))
#<CIRCLE 402000B1B3>

CL-USER 7 > (set-delegate c delegate)
#<TEST-DELEGATE 42002FF5C3>

CL-USER 8 > (move c 10 10)

Proběhne změna.
Proběhne změna.
Proběhne změna.
Proběhla změna.
Proběhne změna.
Proběhla změna.
Proběhla změna.
Proběhla změna.
#<CIRCLE 402000B1B3>
```

Změny se tedy hlásí zbytečně často.

Pokud bychom se chtěli co nejlépe dopátrat příčin tohoto opakování, bylo by dobré události `ev-changing` a `ev-change` vylepšit, aby obsahovaly více informací o změnách (který objekt se změnil a v důsledku jaké zprávy). Nebudeme to dělat, protože příčina problému je i bez toho celkem jasná: každá změna objektu vyvolává rekurzivně změnu téhož i dalších objektů. Tomuto rekurzivnímu hlášení teď zabráníme tak, že omezíme hlášení jen na první a poslední.

Je tedy třeba potlačit všechna posílání zprávy `changing` kromě prvního a zprávy `change` kromě posledního. To platí jak pro volání z metod, které mění stav objektu (jako např. `set-r`), tak pro volání z metod `ev-changing` a `ev-change`, tj. těch, která jsou způsobena změnou podřízených objektů.

Ve třídě `omg-object` zavedeme vlastnost `change-level`, která bude mít číselnou hodnotu indikující, zda má objekt hlásit změny. Hodnota `0` bude znamenat, že hlášení je povoleno, vyšší hodnoty hlášení potlačí:

```
(defclass omg-object ()
  ((delegate :initform nil)
   (change-level :initform 0)
```

```
   (events :initform '((ev-changing . ev-changing)
                       (ev-change . ev-change)
                       (ev-mouse-down . ev-mouse-down)))))

(defmethod change-level ((obj omg-object))
  (slot-value obj 'change-level))

(defmethod inc-change-level ((obj omg-object))
  (incf (slot-value obj 'change-level))
  obj)

(defmethod dec-change-level ((obj omg-object))
  (decf (slot-value obj 'change-level))
  obj)
```

Potlačení hlášení změn zařídíme v metodách `changing` a `change`:

```
(defmethod changing ((object omg-object))
  (when (zerop (change-level object))
    (send-event object 'ev-changing))
    (inc-change-level object))

(defmethod change ((object omg-object))
  (dec-change-level object)
  (when (zerop (change-level object))
    (send-event object 'ev-change)))
```

Nyní si už můžete sami otestovat, že se změny zbytečně už nehlásí.

Důležitá poznámka. Pokud by během metody, která mění stav objektu, došlo k chybě, inkrementované počítadlo, by se v metodě `change` už nedekrementovalo. třeba tady:

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

Metoda `do-move` v některých třídách může být celkem složitá (třeba u třídy `compound-shape`) a pokud by během ní došlo k chybě, počítadlo inkrementované v metodě `changing` by se už nedekrementovalo. Objekt `shape` by se ocitl v **nekonzistentním stavu**. Zodpovědný programátor takovou situaci nedopustí (a to ani v případě chyby) a použije vhodný nástroj svého jazyka, který zařídí zavolání `change` i v případě, že v `do-move` došlo k chybě (v Lispu je takovým nástrojem operátor `unwind-protect`).

7 Omezení knihovny `omg`

Naše malá objektová grafická knihovna (která dnes dostala i název) je hotová. Mějme ale na paměti, že i když funguje dobře, nebyla nikdy navržena k praktickému využití. V tom jí brání (kromě toho, že je příliš jednoduchá) významné nedostatky. V této části o nich v budoucnu něco napíšu.

Otázky a úkoly na cvičení

Pokud není řečeno něco jiného, nikdy při řešení neupravujte knihovnu `omg`. Chovejte se čistě jako její uživatel. V případě, že to není možné, zdůvodněte to a knihovnu upravte co nejjednodušším způsobem. V Lispu je možné dodefinovávat existujícím třídám vlastní metody. V jiných jazycích to většinou buď nejde, nebo to není doporučeno. Vy tuto možnost použijte jedině v dobře odůvodněných případech.

1. Vylepšete třídu `button` tak, aby tlačítko po změně svého textu poslalo událost `ev-button-text-change`. Argumenty události by byly: odesílatel, tlačítko, původní text, nový text. (Argumenty s odesílatelem a tlačítkem by byly u tlačítka vždy stejné, ale pokud by delegát událost chtěl přeposlat svému delegátovi, už by se lišily. Přeposílání ovšem neprogramujte.)

2. Zkuste udělat totéž bez toho, že byste třídu `button` jakkoli měnili. Místo toho své tlačítko definujte jako novou třídu, která bude jejím potomkem.

3. Abyste se přesvědčili, že počet hlášení změn v předposlední verzi naší knihovny je opravdu neúnosně velký, vyzkoušejte si pomocí testovacího delegáta otočit čtverec. Nejprve zkuste co nejpřesněji odhadnout, kolik řádků o změnách se vytiskne, a potom poslední výraz vyhodnoťte.

```
CL-USER 9 > (setf square
                  (set-items
                    (make-instance 'polygon)
                    (list
                      (make-instance 'point)
                      (move (make-instance 'point) 10 0)
                      (move (make-instance 'point) 10 10)
                      (move (make-instance 'point) 0 10))))
#<POLYGON 402000157B>

CL-USER 10 > (set-delegate square delegate)
#<TEST-DELEGATE 42002FF5C3>

CL-USER 11 > (rotate square pi (make-instance 'point))

Proběhne změna.
```

```
...

Proběhla změna.

#<POLYGON 420034084B>
```

4. Instance třídy `rating-stars` by se měly chovat jako známé komponenty uživatelského rozhraní sloužící k označení určitého počtu hvězdiček. Ten by byl uložen v nastavitelné vlastnosti `max-rating`. Aktuální hodnocení by bylo uloženo ve vlastnosti rating nastavitelné kliknutím i programově. Změny hodnocení by měl objekt hlásit nově definovanou událostí `ev-rating-change`.

5. Třídu `rating-stars` otestujte na vhodném příkladě. Můžete například definovat okno, které bude obsahovat v řádcích několik objektů třídy `rating--stars` nadepsaných nějakým textem (může to být třeba název filmu vždy s hvězdičkami s možností nastavit jeho hodnocení). Pod nimi by v instancích třídy `text-shape` bylo zapsáno minimální, průměrné a maximální hodnocení, které by se vždy automaticky měnilo. Fantazii se meze nekladou; okno by třeba dále mohlo obsahovat tlačítka na přidání a odebrání řádku s hvězdičkami.

6. Události `ev-mouse-enter` a `ev-mouse-leave`, které jste definovali v jednom z úkolů k minulé přednášce, použijte k vylepšení třídy `rating-stars`: pokud uživatel vjede myší do instance, hvězdičky se vhodně označí, ještě než uživatel klikne. Pokud ji opustí bez kliknutí, zobrazení přejde do původního stavu.

7. Napište třídu `radio-group`. Instance se budou chovat jako skupiny radiobuttonů známé z grafických uživatelských rozhraní. Třída by měla přinejmenším obsahovat vlastnost na počet tlačítek ve skupině. Definujte vhodné pomocné třídy (např. pro jednotlivé tlačítko) a nezapomeňte na zasílání vhodných událostí. Můžete použít třídu `text-shape` z příkladů. Třídu otestujte na vhodném příkladě.

8. Definujte třídu `menu-window` jako potomka třídy `window`, jejíž instance se budou chovat stejně jako obyčejné okno, ale kliknutí pravým tlačítkem myši zobrazí kontextovou nabídku k objektu s nastavenou vlastností `solidp` pod myší nebo celému oknu. Ta může vypadat jakkoli jednoduše, ale musí alespoň obsahovat několik řádků s textem, představujícím jednotlivé položky nabídky. Nabídka se musí alespoň částečně lišit pro jednotlivé typy objektů. Příklady položek nabídky:

   - Pro instanci třídy `shape`: nastavit barvu na červenou, posunout o (10,10).
   - Pro instanci třídy `circle`: zvětšit poloměr o 10, zmenšit poloměr na polovinu.
   - Pro polygon: přidat bod na místo kliknutí, změnit `closedp`.

- Při kliknutí mimo objekt: změnit barvu pozadí okna na náhodnou barvu, smazat `shape`, nastavit `shape` na nové kolečko.

Nabídka by měla napodobovat obvyklé chování kontextových menu: po kliknutí do položky zmizet a vykonat požadovanou akci, po kliknutí mimo nabídku zmizet bez vykonání jakékoli akce. Ve druhém případě by se neměla vykonat ani žádná akce případně naprogramovaná pro obsah okna.

Nabídka by neměla být součástí obsahu okna! Uživatel nečeká, že se oknu změní jeho `shape`, když se objeví kontextová nabídka.

9. Napište třídu `select-window` (potomek třídy `window`) s přidanou funkčností označování položek. Požadavky na řešení:

   (a) Po kliknutí na objekt s nastavenou vlastností `solidp` se tento objekt orámuje obdélníkem. Orámování případného dříve označeného objektu se zruší.

   (b) Označený objekt bude uložen ve vlastnosti `selection` okna. Pokud není nic označeno, vlastnost bude obsahovat nil.

   (c) Po kliknutí mimo všechny objekty se označení zruší.

   (d) Grafický objekt okna musí být nezávislý na tom, který objekt je označen. Rámovací obdélník v něm nebude nikde uložen.

10. Definujte třídu `inspector-window`, jejíž instance (nazývané *prohlížeče*) budou okna, zobrazující informace o zvoleném objektu jiného (*prohlíženého*) okna. Informace o objektu by měla obsahovat název třídy objektu a dále názvy a hodnoty základních vlastností (např. `color`, `thickness`, `radius`, `x`, `y`, `r`, `phi`, `closedp` atd., podle třídy prohlíženého objektu). Prohlížená okna budou instancemi vámi definované třídy `inspected-window`.

    Základní funkčnost:

    (a) Prohlížeč má nastavitelnou vlastnost `inspected-window`, která obsahuje okno, jež má být prohlíženo.

    (b) Po nastavení prohlíženého okna prohlížeč zobrazí informace o tomto okně.

    (c) Po kliknutí do prohlíženého okna zobrazí prohlížeč informace o pevném objektu (tj. s vlastností `solidp` rovnou *Pravda*), na který uživatel klikl. Po kliknutí mimo všechny objekty se zobrazí opět informace o okně.

    (d) Prohlížený objekt je uložen ve vlastnosti `inspected-object` prohlížeče. Tato vlastnost nemusí být zapisovatelná.

    (e) Při změně aktuálně prohlíženého objektu prohlížeč automaticky aktualizuje zobrazenou informaci.

    (f) Po dvojkliku na hodnotu vlastnosti zobrazí prohlížeč dialog s dotazem na novou hodnotu. Po potvrzení vlastnost na tuto hodnotu změní.

(g) Pro třídy grafických objektů, které definuje někdo jiný (třeba pro třídu `bulls-eye`), musí být specifikován (a v dokumentaci popsán) způsob, jakým prohlížeč zjistí a nastaví jejich nové vlastnosti. (Bez úpravy třídy `inspector-window` a `inspected-window` musí tedy autor třídy `bulls-eye` být schopen zařídit, aby prohlížeč zobrazoval a nastavoval hodnotu vlastnosti `squarep`.)

Je možné využít následující možnosti Lispu a LispWorks:

- Libovolnou hodnotu (typicky symbol, číslo, text, třídu apod.) lze čitelně vytisknout do textového řetězce voláním `(format nil "~a" hodnota)`.
- Třídu objektu lze zjistit vyhodnocením výrazu `(type-of objekt)`.
- Dialog s dotazem na novou hodnotu můžete zobrazit voláním

```
(multiple-value-list
 (capi:prompt-for-value "Zadejte novou hodnotu"))
```

  Výsledkem je seznam s prvním prvkem rovným uživatelem zadané hodnotě a druhým informaci (logické hodnotě), zda uživatel dialog stornoval.
- K názvu vlastnosti lze zjistit název zprávy, která vlastnost nastavuje, pomocí této funkce:

```
(defun setter-name (prop)
  (values (find-symbol (format nil "SET-~a" prop))))
```

  Například:

```
CL-USER 74 > (setter-name 'radius)
SET-RADIUS
:INTERNAL
```

  (druhou hodnotu `:internal` můžete ignorovat).

Můžete také použít třídu `text-shape` z příkladů, případně i další příklady. Pokud je to vhodné, použijte i zatím nepoužité služby knihovny `micro-graphics`.

Neupravujte knihovnu `omg`; pokud dojdete k závěru, že na ní potřebujete něco změnit, přepište příslušné metody ve vašich třídách. Výjimka z tohoto pravidla: existujícím třídám knihovny `omg` můžete dodefinovávat vlastní nové metody.

11. Upravte knihovnu `omg` tak, aby objekty mohly mít více delegátů. Kdy se to může hodit?