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 5. října 2022

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 CIS of (NIL) arguments should be of type REAL.
  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 hodnotu funkce `cis` 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

```
(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.

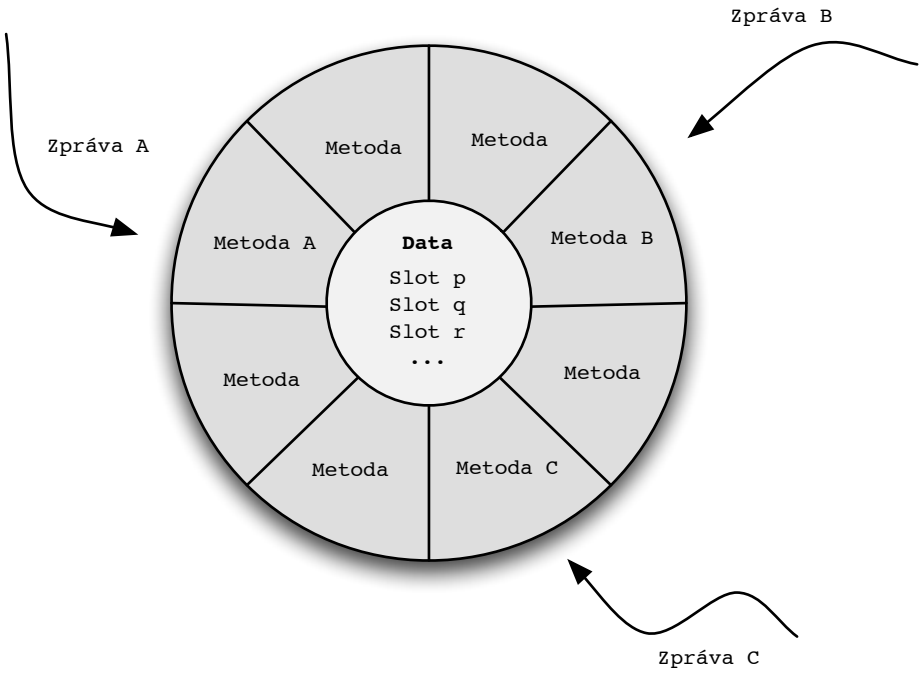
Princip zapouzdření

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

(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 bude i tak 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 polygon

Další využití zapouzdření ukážeme na příkladě třídy `polygon`. Z našeho pohledu jsou polygony grafické objekty, které obsahují jiné grafické objekty jako své prvky: polygon je tvořen seznamem bodů. Kreslí se jako lomená čára, tyto body spojující, nebo plocha lomenou čarou omezená. Obdélníky, čtverce i trojúhelníky jsou polygony.

Příklad: třída `polygon` s typovou ochranou

Instance třídy `polygon` budou obsahovat seznam vrcholů polygonu, které budeme reprezentovat instancemi třídy `point`. Z pohledu uživatele bude seznam uložen ve vlastnosti `items`, z pohledu interního jej budeme ukládat do slotu `items`.

Název vlastnosti i slotu jsem záměrně zvolil obecnější než se nabízí (vhodnější by bylo třeba `vertices`). Je to proto, že vím, že se nám obecnější název bude v budoucnu více hodit.

Definice třídy:

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

Metody pro vlastnost `items`:

```
(defmethod items ((poly polygon))
  (slot-value poly 'items))

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

(defmethod check-items ((poly polygon) items)
  (dolist (item items)
    (check-item poly item))
  poly)

(defmethod set-items ((poly polygon) value)
  (check-items poly value)
  (setf (slot-value poly 'items) value)
  poly)
```

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` body (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 p (make-instance 'polygon))
#<POLYGON 402000AD63>

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

Error: Items of polygon should be points.
  1 (abort) Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template 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 později 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.

Takto definovaná třída `polygon` 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 `polygon`

Pokračujme tedy s testováním třídy `polygon`. Vložme do proměnné `list` seznam bodů a uložme jej jako seznam prvků již vytvořenému polygonu `p`:

```
CL-USER 5 > (setf list (list (make-instance 'point) (make-instance
'point)))
(#<POINT 40200ED6AB> #<POINT 40200ED6EB>)

CL-USER 6 > (set-items p list)
#<POLYGON 402000AD63>
```

Jaké jsou vrcholy polygonu?

```
CL-USER 7 > (items p)
(#<POINT 412029443B> #<POINT 412029477B>)
```

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 #<POINT 412029477B>)
```

A co bude nyní v seznamu vrcholů polygonu?

```
CL-USER 10 > (items p)
(0 #<POINT 412029477B>)
```

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

Příklad: úprava třídy `polygon`

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 ((poly polygon))
  (copy-list (slot-value poly 'items)))

(defmethod set-items ((poly polygon) value)
  (check-items poly value)
  (setf (slot-value poly 'items) (copy-list value))
  poly)
```

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

Několik testů:

```
CL-USER 11 > (setf list (list (make-instance 'point) (make-instance
'point)))
(#<POINT 4020001353> #<POINT 4020001393>)

CL-USER 12 > (set-items p list)
#<POLYGON 412028A203>

CL-USER 13 > (setf (first list) 0)
0

CL-USER 14 > (setf (second (items p)) 0)
0

CL-USER 15 > (items p)
(#<POINT 41202ACB53> #<POINT 41202ACB6B>)
```

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 `p`.

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 funkce `slot-value`.

Vlastnosti versus sloty

Zatímco sloty jsou vnitřní záležitostí třídy a uživatel o nich nemá informace, vlastnosti jsou součástí rozhraní, které uživatel používá.

Příklad

Pro třídy `point`, `circle` a `polygon` 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 `polygon`.

Otázky a úkoly na cvičení

1. Popište princip zapouzdření a vyjmenujte jeho tři hlavní výhody.
2. Co je to vlastnost objektu a jaký je rozdíl mezi vlastností a slotem?
3. Uveďte příklad vlastnosti, jejíž hodnota není uložena ve slotu.
4. Upravte třídu `triangle` z úloh k minulé přednášce, aby vyhovovala principu zapouzdření.
5. Napište metodu `to-polygon` třídy `triangle`, která vrátí polygon s vrcholy stejných souřadnic (nikoli totožnými!), jako má trojúhelník, jenž je příjemcem zprávy.
6. Definujte třídu `picture`, jejíž instance budou podobné instancím třídy `polygon`, ale prvky vlastnosti `items` mohou být nejen body, ale libovolné grafické objekty (body, kruhy, polygony a jiné instance třídy `picture`). Instance budou reprezentovat skupiny grafických objektů.
7. Přesvědčte se, že definice třídy `ellipse` z minulého cvičení 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.
8. Upravte také metodu `to-ellipse` třídy `circle`, aby vyhovovala principu zapouzdření.