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

5. Návrh stromu dědičnosti

verze z 21. října 2020

1 Potíže s dědičností

Při návrhu stromu dědičnosti se programátor nachází v obtížné situaci. Musí navrhnout strukturu použitelnou pro dnes neznámé účely uživatelem, který nebude mít možnost ji měnit. Jedinou šancí, jak se tohoto úkolu dobře zhostit, je dodržovat osvědčené programátorské zásady.

Hlavní zásadou, kterou jsme už uvedli, je pravidlo *is-a*. Jeho dodržováním zajistíme, že námi navrhovaná struktura tříd bude (víceméně věrně) kopírovat strukturu typů předmětů reálného světa, které se snažíme modelovat. (Ony typy předmětů reálného světa jsou ovšem také něco uměle definovaného; i zde je třeba být obezřetný.)

Připomeňme si, jak jsme pravidlo formulovali.

Pravidlo *is-a*

Je-li třída D potomkem třídy C, pak věta „každé D je C“ musí dávat smysl.

Následující příklad ukazuje, do jakých potíží se můžeme dostat, pokud pravidlo nedodržujeme.

Příklad: je úsečka bod?

Uvažme tuto definici třídy `segment` (úsečka):

nesprávně

```
(defclass segment (point)
  ((x2 :initform 0)
   (y2 :initform 0)))
```

Vedla nás následující úvaha: Úsečka se skládá ze dvou bodů. Je tedy třeba definovat čtyři sloty, vždy dva pro dvě souřadnice jednoho bodu. Jednu takovou dvojici slotů už máme ve třídě `point`, je tedy třeba třídu rozšířit ještě o další dva. Použijeme dědičnost, sloty `x` a `y` zdědíme z třídy `point` a zbylé definujeme v naší třídě.

Nové sloty budou sloužit k uložení hodnot nových vlastností `x2` a `y2`, pro ně tedy samozřejmě napíšeme přístupové metody. Dále vhodně přepíšeme ostatní metody třídy `point`. Kde to bude účelné, zavoláme zděděnou metodu. Například:

```
(defmethod move ((seg segment) dx dy)                    nesprávně
  (call-next-method)
  (set-x2 seg (+ (x2 seg) dx))
  (set-y2 seg (+ (y2 seg) dy))
  seg)
```

Podobně přepíšeme metody `rotate` a `scale`.

Další metody, které budeme muset přepsat, jsou metody `set-mg-params` a `do-draw`. Zejména u první narazíme na nepříjemnost: zjistíme, že metoda ze třídy `point` se nám nehodí. Raději bychom ji zrušili a funkcí `call-next-method` volali přímo metodu třídy `shape`. Metoda `do-draw` ze třídy `point` nám zase nebude vůbec k ničemu, budeme ji muset kompletně přepsat.

Při práci na třídě postupně zjistíme, že musíme přepsat v podstatě všechny metody třídy `point`. To, že jsme třídu `segment` definovali jako jejího potomka, nám nic užitečného nepřineslo, spíše komplikace. A kdyby autoři třídy `point` k ní v budoucnu připsali nové metody, museli bychom neustále naši třídu upravovat. Neustále by hrozilo, že se bude chovat nekorektně.

To platí například pro nové metody `left`, `top`, `right`, `bottom`, které jste měli za úkol doplnit k našim třídám v úlohách k minulé přednášce. Snadno zjistíte, že instance třídy `segment` nebudou na nové zprávy reagovat správně, dokud třídu neupravíte. Tohle jistě není účelem principu dědičnosti.

Příčinu problémů najdeme v nedodržení principu *is-a*. **Úsečka totiž není bod.** V reálném světě není množina úseček podmnožinou množiny bodů. Proto není většina předpokladů o instancích třídy `point` splněna pro instance třídy `segment` a my musíme neustále přepisovat existující metody.

Další příklady nás přesvědčí o tom, že bychom si měli přesněji stanovit, co pravidlo *is-a* má říkat.

Příklad: je bod úsečka?

Úsečka tedy není bod. Ale co naopak? Bod se dá chápat jako úsečka, jejíž koncové body splývají. Co kdybychom tedy udělali třídu `segment` *předkem* třídy `point`? Pravidlo *is-a* by pak mělo být splněno.

Bohužel i zde narazíme na problém. Pokud je u úsečky povoleno měnit její koncové body (což by mělo být), může se stát, že úsečka se splývajícími koncovými body se stane úsečkou, jejíž koncové body nesplývají. Na podobný problém narazíme v dalším příkladě.

Příklad: je trojúhelník polygon?

Při návrhu třídy `triangle` bychom se jistě měli zamyslet, zda ji neudělat potomkem třídy `polygon`. Podle pravidla *is-a* by to zdá se mělo být možné. Navíc by nám to přineslo mnohé výhody, zejména v tom, že většinu metod by třída mohla zdědit

z třídy `polygon`: jednak metody pro geometrické transformace a jednak metody pro kreslení.

Aby to fungovalo, byly by samozřejmě vrcholy trojúhelníka prvky seznamu `items`. To by nám ovšem nebránilo zachovat i vlastnosti `vertex-a`, `vertex-b` a `vertex-c`, které by své hodnoty braly z tohoto seznamu.

Na nepříjemnost ovšem narazíme při nastavování vlastnosti `items`: trojúhelník má vždy tři vrcholy, nastavením vlastnosti na seznam s jiným počtem prvků bychom trojúhelník uvedli do nekonzistentního stavu (nehledě na to, že vrcholy trojúhelníka nepotřebujeme nikdy nastavovat).

Podrobnější rozbor nás tedy donutil změnit názor. V našem pojetí **trojúhelník není polygon**, protože polygonu lze libovolně nastavovat seznam `items`, což u trojúhelníka nejde.

Příklad: je `bulls-eye` obrázek?

Podobný problém má i třída `bulls-eye`, kterou jste dostali jako příklad k minulé přednášce. Ve třídě `picture` je možno libovolně měnit seznam `items`, což by ovšem ve třídě vedlo k nekonzistentním objektům.

Příklad: neposlušná kolečka

Následující potomci třídy `circle` dělají každý něco jinak než třída `circle`. Můžete se zamyslet, zda splňují pravidlo *is-a*.

První kolečko je opravdu neposlušné:

```
(defclass my-circle-1 (circle)
  ())

(defmethod move ((c my-circle) dx dy)
  c)
```

Instance třídy `my-circle-1` se chovají přesně stejně jako instance třídy `circle` s jedinou výjimkou: nereagují na zprávu `move`.

Instance druhé třídy na zprávu `move` zareagují, jak mají, navíc ovšem při každém posunu změní barvu:

```
(defclass my-circle-2 (circle)
  ())

(defmethod move ((c my-circle-2) dx dy)
  (call-next-method)
  (set-color c :green))
```

Třetí třída:

```
(defclass my-circle-3 (circle)
  ((move-count :initform 0)))

(defmethod move-count ((c my-circle-3))
  (slot-value c 'move-count))

(defmethod move ((c my-circle-3) dx dy)
  (call-next-method)
  (incf (slot-value c 'move-count))
  c)
```

Instance počítá, kolikrát jsme ji posunuli. Hodnotu ukládá do vlastnosti `move-count`.

Čtvrtá třída:

```
(defclass my-circle-4 (circle)
  ())

(defmethod initialize-instance ((c my-circle-4) &key)
  (call-next-method)
  (set-color c :red))
```

Hodnota vlastnosti `color` nově vytvořených instancí třídy `shape` je `:black`. To v této třídě měníme.

A konečně pátá třída:

```
(defclass my-circle-5 (circle)
  ((default-dx :initform 0)
   (default-dy :initform 0)))

(defmethod default-dx ((c my-circle-5))
  (slot-value c 'default-dx))

(defmethod set-default-dx ((c my-circle-5) value)
  (setf (slot-value c 'default-dx) value)
  c)

(defmethod default-dy ((c my-circle-5))
  (slot-value c 'default-dy))

(defmethod set-default-dy ((c my-circle-5) value)
  (setf (slot-value c 'default-dy) value)
  c)
```

```
(defmethod move ((c my-circle-5) dx dy)
  (when (eql dx t) (setf dx (default-dx c)))
  (when (eql dy t) (setf dy (default-dy c)))
  (call-next-method c dx dy)
  c)
```

Metoda `move` třídy `my-circle-5` využívá nám dosud neznámé možnosti volat funkci `call-next-method` s jinými argumenty než těmi, se kterými byla metoda původně zavolána. Tím přidáváme možnost poslat objektu zprávu `move` s argumenty `t`, `t`, které vedou k posunu o defaultní hodnoty.

Ve všech příkladech neposlušných koleček zpráva `move` nezanechá instance v nekonzistentním stavu. To je rozdíl proti předchozím příkladům. U některých příkladů máme pochybnosti o korektnosti. Otázka, zda třídy splňují pravidlo *is-a*, tedy ani zdaleka není jednoduchá. Zřejmě budeme muset upřesnit samotné pravidlo.

2 Princip substituce

Upřesněné pravidlo *is-a* vychází z důslednějšího pochopení vztahu třídy a jejího potomka, pokud jsou chápány jako množiny objektů. Víme, že instance potomka je vždy i instancí původní třídy. Pokud tedy zapomeneme, že jde o instanci potomka, **musí se chovat jako instance původní třídy**.

Princip substituce

Pokud kdekoli v programu nahradíme instanci třídy instancí jejího potomka, musíme dostávat stejné výsledky.

Uvedený princip se ve formalizované podobě nazývá *principem Liskovové* podle významné informatičky Barbary Liskov. Zajišťuje, že pokud s instancemi dané třídy pracujeme jen s použitím znalostí o této třídě, máme jistotu, že budeme docházet vždy ke správným výsledkům, i když instance není přímou instancí třídy (tj. je instancí nějakého potomka).

Za chvíli si povíme konkrétněji, jak princip budeme používat v tomto předmětu.

Příklad: metoda `left` pro úsečku?

Jedna z vlastností instancí třídy `point` se týká metody `left`. Pokud bychom neuvažovali tloušťku pera (vlastnost `thickness`), je zřejmé, že každý bod by měl v reakci na zprávu `left` vrátit hodnotu své `x`-ové souřadnice (tak jste také jistě metodu `left` ve třídě `point` naprogramovali). Pokud metodu implementujeme v naší nesprávné třídě `segment` (kterou jsme napsali jako potomka třídy `point`) přirozeným způsobem, budeme dostávat jiné výsledky; rovnost vlastnosti `left` a vlastnosti `x` bude porušena. Třída tedy **porušuje princip substituce**.

3 Kontraktové programování

Český překlad anglického **Design by Contract** je to způsob programování zavedený Bertrandem Meyerem a implementovaný v jeho objektově orientovaném jazyce *Eiffel*.

Kontraktové programování je založeno z našeho pohledu na tom, že než začneme programovat třídu a její metody, stanovíme podmínky, které musí splňovat, tzv. *kontrakt*. Podmínky jsou následujících tří typů:

Invarianty třídy jsou podmínky, které instance třídy musejí neustále splňovat. Chápeme je tak, že pokud instance nesplňuje invariant, není v konzistentním stavu. Příklady: Poloměr kruhu je kladné číslo. Hodnota `items` polygonu je seznam bodů. Počet vrcholů trojúhelníka je 3.

Prekondice metody jsou podmínky na stav instance a hodnoty parametrů metody při jejím volání. Prekondice většinou zajišťují, že se instance v metodě nedostane do nekonzistentního stavu (tj. neporuší invariant). Zatím jsme se nesetkali s případem, že by v prekondici vystupovala i instance (vždy to byly pouze hodnoty parametrů), ale to se změní. Příklad: Hodnoty parametrů metody `move` třídy `shape` musí být čísla. Hodnota parametru metody `set-radius` třídy `circle` musí být kladné číslo.

Postkondice metody jsou podmínky na stav instance po vykonání metody ve vztahu k jejímu stavu na začátku metody a k hodnotám parametrů (splňujícím prekondice). Postkondice stanovují, co má metoda dělat. Příklad pro metodu `set-color` třídy `shape`: hodnota vlastnosti `color` se po provedení metody musí rovnat hodnotě parametru. Pro metodu `move` třídy `point`: hodnota vlastnosti `x` po provedení metody musí být rovna součtu hodnoty této vlastnosti před provedením metody a parametru `dx`.

Invarianty třídy bychom měli vždy stanovit při její definici. Prekondice a postkondice metody bychom měli stanovit při definici metody. Nebudeme to dělat příliš formálně, ale vždy bychom se nad nimi měli alespoň zamyslet, aby naše definice nevedly k problémovému chování ukázanému v předchozích příkladech.

Vztah invariantů, prekondic a postkondic předka a potomka musí být následující:

1. Pro potomka platí všechny **invarianty** předka (sám může další přidat).
2. **Prekondice metody** potomka musí být slabší než prekondice metody předka pro tutéž zprávu. (Potomek tedy může odebrat podmínky, nesmí žádné přidat.)
3. **Postkondice metody** potomka musí být silnější než postkondice metody předka pro tutéž zprávu. (Potomek tedy může přidat podmínky, nesmí žádné odebrat.)

Uvedené tři podmínky znamenají naši přesnou konkrétní formulaci pravidla *is-a*.

Poznamenejme, že princip Liskovové je přísnější, protože se vztahuje ke **všem myslitelným vlastnostem**, které třídy a jejich metody mají. V našem přístupu ověřujeme jen invarianty, prekondice a postkondice, které si sami stanovíme. Míra benevolence se v tomto ohledu u různých přístupů a jazyků liší.

Náš přístup si můžete vyzkoušet na příkladech (zatím sem napíšu jen několik málo).

Příklad: kontrakt ve třídě `segment`

Třída nesplňuje invariant „`left` = `x`“ třídy `point`.

Příklad: kontrakt ve třídě `triangle`

Ve třídě je přirozené zavést invariant „Počet prvků vlastnosti `items` je 3“. Ten ale není splněn po zaslání zprávy `set-items` s argumentem `()`.

Příklad: kontrakt ve třídě `my-circle-1` (blokování posunu)

Metoda `move` třídy `circle` splňuje postkondici „Souřadnice `x` středu kruhu na konci metody je rovna této souřadnici na začátku metody zvýšené o hodnotu parametru `dx`“. Metoda `move` třídy `my-circle-1` ji nesplňuje.

Příklad: kontrakt ve třídě `my-circle-2` (změna barvy po posunu)

Toto je hraniční případ. Podle striktně chápaného principu substituce je třída nekorektně definována, protože pro třídu `shape` (a též `circle`) platí, že po posunu objektu se nemění jeho barva. My se ale domluvíme na následujícím volnějším pravidle:

Pravidlo pro postkondice

Postkondice metody budeme obvykle formulovat pouze pro vlastnosti objektu, se kterými metoda pracuje, a vlastnosti příbuzné.

Kdyby tedy kruh po posunu změnil poloměr, nebyl by náš kontrakt dodržen, protože poloměr kruhu chápeme jako vlastnost příbuznou souřadnicím středu (obojí se týká geometrie kruhu). Barva se souřadnicemi středu nesouvisí, takže třídu `my-circle-2` budeme považovat za korektně zavedenou.

Toto rozhodnutí je ovšem diskutabilní a v praxi pak záleží na pravidlech daných programovacím jazykem, případně dalších okolnostech.

Když už jsme u diskutabilních rozhodnutí, uděláme ještě jedno, později je ale zpřísníme:

Kontrakt pro kreslení

Způsob vykreslení objektu do okna metodou `draw` nezahrnujeme do kontraktu.

Příklad: kontrakt ve třídě `my-circle-3` (počítání posunů)

Třída je naprogramována korektně. Splňuje všechny invarianty třídy `circle` (navíc přidává invarianty pro vlastnost `move-count`, což je povoleno. Prekondice i postkondice pro metodu `move` jsou tytéž jako ve třídě `circle`.

Příklad: kontrakt ve třídě `my-circle-4` (jiná barva nových instancí)

Stav nově vytvořené instance by mohl být součástí našeho kontraktu o třídě. Mohl by být dán například postkondicí funkce `make-instance`. Takové omezení nebudeme vynucovat. Zavedeme tedy pravidlo, že potomci mohou definovat jiný stav nově vytvořeného objektu než původní třída. Uživatel se tedy např. nesmí spoléhat na to, že vlastnost `color` nově vytvořené instance třídy `shape` má vždy hodnotu `black`.

Příklad: kontrakt ve třídě `my-circle-5` (nové typy argumentů)

Třída je naprogramována korektně. Prekondice metody `move` jsou v ní proti prekondicím předka **oslabeny** (umožňují jiné hodnoty parametrů `dx` a `dy` než jen čísla), což je povoleno. Pokud uživatel s kolečkem bude pracovat jen jako s instancí třídy `circle`, nedojde k žádnému problému.

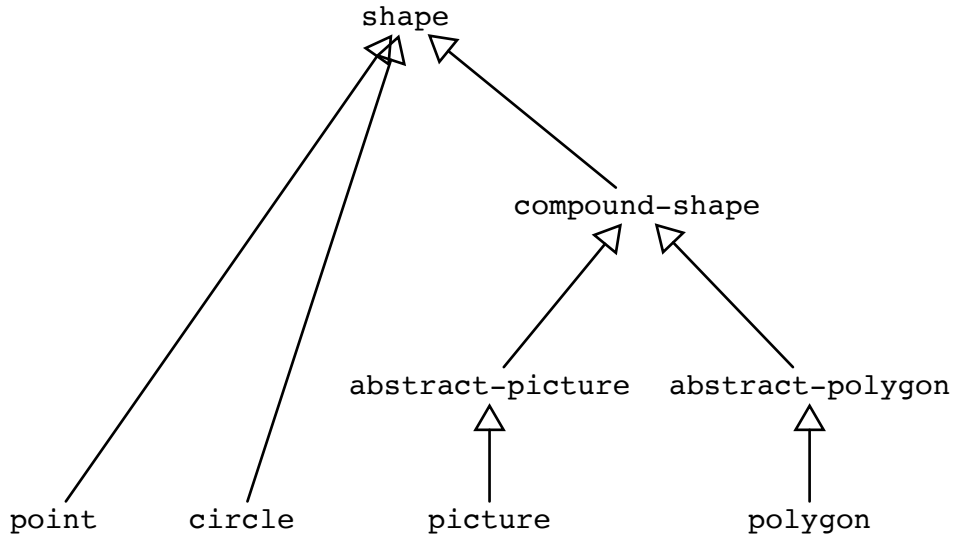
4 Kontrakty v naší grafické knihovně

Abychom v knihovně dodržovali přirozené kontrakty a současně se vyhnuli problémům uvedeným na začátku přednášky, definujeme nové třídy `abstract-polygon`, `abstract-picture` a `abstract-window`. Podrobnosti najdete ve zdrojovém kódu a příkladech v přednášce. Nový strom dědičnosti tříd grafických objektů je na Obrázku 1.

Otázky a úkoly na cvičení

U úloh, kde to dává smysl, se zamyslete nad kontraktem pro novou třídu a její metody.

1. Zkuste definovat metody `left`, `top`, `right`, `bottom` k chybně zavedené třídě `segment` ze začátku této přednášky.



Obrázek 1: Nový strom dědičnosti tříd grafických objektů

2. Rozhodněte, zda vaše třída `extended-picture` z úkolů z minulé přednášky splňuje princip substituce v přísné formě a zda splňuje naše volnější podmínky (kontrakt).

3. Splňuje třída `light` z příkladů k minulé přednášce naše podmínky (kontrakt)?

4. Definujte třídu `triangle` jako potomka třídy `abstract-polygon`. Ošetřete, aby se v `set-items` mohly nastavovat jen tříprvkové seznamy.

5. Navrhněte třídu `circle-picture`, jejíž instance budou obrázky, které v seznamu `items` mohou obsahovat jen kolečka. Jakou třídu stanovíte jako předka?

6. Navrhněte třídu `circle-window`, jejíž instance jsou okna, která jako svůj `shape` mohou kromě hodnoty `nil` obsahovat jen instanci třídy `circle`.

7. Upravte třídu `ellipse` podle poznatků z této kapitoly. Jaký by měl být vztah této třídy a třídy `circle`?

8. Totéž udělejte pro třídy `empty-shape` a `full-shape`.

9. Navrhněte třídu `disc`, jejíž instance se budou skládat ze dvou různobarevných plných soustředných kruhů. Instance budou mít vlastnosti `radius`, `inner-radius` (poloměr vnějšího a vnitřního kruhu) a `color` a `inner-color` (barva vnějšího a vnitřního kruhu). Všechny vlastnosti budou jak ke čtení tak k zápisu. Je vhodné také napsat vlastnost `center` (jen ke čtení) obsahující střed kruhů.

   Zvažte, zda je vhodné ukládat hodnoty těchto vlastností do slotů definovaných ve třídě `disc`. Nebo je lepší zvolit jiné řešení? Proč?

   Třída `disc` by měla být potomkem třídy `abstract-picture` nebo `circle`. (U `abstract-picture` k nastavení vlastnosti `items` na dva kruhy při vytváření

nové instance použijte metodu `initialize-instance`.) Která možnost je lepší?

10. Uvažme třídu `hideable-picture`, jejímiž instancemi by byly obrázky, jejichž některé prvky by bylo možné skrýt, takže by se nevykreslovaly. Třída by byla potomkem třídy `abstract-picture` nebo přímo `picture`. Která z možností je správně?

    Která z následujících možností návrhu třídy je lepší a proč? Možnosti prodiskutujte a třídu naprogramujte. Případně navrhněte jiný, vlastní způsob.

    1. Třída bude mít novou vlastnost `hide-items`. Ta bude obsahovat seznam stejné délky jako seznam `items`, který bude obsahovat hodnoty `t` nebo `nil`. Prvky obrázku s příslušnou hodnotou `t` se budou vykreslovat, ostatní se skryjí. Vlastnost bude nastavitelná uživatelem.

    2. Všechny prvky obrázku budou uloženy v nové vlastnosti `all-items`. Obrázek bude mít i vlastnost `hide-items` jako v předchozím případě, ta se ale nyní bude týkat vlastnosti `all-items`. Vlastnost `items` bude obsahovat pouze prvky obrázku, které nejsou skryté.