Programozási Nyelvek Összehasonlító Elemzése

Objektum-orientált programozás és kivételkezelés Common Lispben

2007. szeptember, Salvi Péter

Áttekintés

1) Objektum-orientáltság
   - Részletes tárgyalás:
     - Általános ismertető, az osztály ill. objektum fogalma
     - Öröklődés, többszörös öröklődés
     - Sablonfüggvények, metódusok, polimorfizmus
     - Metódusok összekapcsolása
     - Objektumok létrehozása és megszüntetése
     - Egységbezárás, adatrejtés
   - Bemutató egy példán keresztül
   - Többszörös kötés megvalósítása Javában és CLOSben
2) Kivételkezelés
   - A “hagyományos” kivételkezelés problémája
   - Kivételek Common Lisp módra

Common Lisp Object System: CLOS

- Eredetileg kiegészítés, később ANSI szabvány
- Filozófiája lényegesen eltér a megszokott OOP-tól
- Rendszerint Lispben implementálják (mint a nyelv nagyrészét, makrók segítségével)

- A CLOS összetevői:
  - Osztályok
  - Példányok (objektumok)
  - Sablonok
  - Metódusok
- Az osztályok és metódusok nincsenek egybekötve

Az osztály

- Az osztály definíciójában megadhatóak:
  - Az ősosztályok (többszörös öröklődés megengedett)
  - Tulajdonságok (lehetnek statikusak (*shared*) is)

- A definíció formája:
  `(`**`defclass`** *`osztálynév`* `(`*`ősök*`)`
  `(`*`tulajdonságok*`)`
  *`osztály-beállítások*`)`

- Az “osztály-beállítások” rész tartalmazhat dokumentációt ill. alapértelmezett értékeket

A tulajdonságok

- Egy tulajdonság megadható:
  - Csak a nevével (alapértelmezett beállítások)
  - Kulcsszavakkal kiegészített listával
- A felhasználható kulcsszavak:
  - *:reader* / *:writer* / *:accessor* megadja a függvény nevét, amivel a tulajdonság írható / olvasható / mindkettő
  - *:allocation* – értéke *:instance* vagy *:class*, azaz hogy a tulajdonság a példányoké vagy az osztályé
  - *:initarg* – a tulajdonság kulcsszava a konstruktorban
  - *:initform* – alapértelmezett érték
  - *:type* – a tulajdonság típusa
  - *:documentation* – a tulajdonság dokumentációja

Öröklődés

- Az osztályok precedenciájának két szabálya van:
  1) A származtatott osztály megelőzi a szüleit
  2) A származtatás meghatározza a szülők sorrendjét
- Ezek alapján minden osztályhoz meghatározható egy precedencia-lista, ami a legspecifikusabbtól a legáltalánosabbig megy
- A legáltalánosabb osztály a *T*
- A saját osztályoknak szintén őse a *standard-object*
- A precedencia-lista nem mindig egyértelmű, de a nem-egyértelmű esetekben determinisztikus (és érdektelen)

Nem egyértelmű precedencia

- Vizsgáljuk meg a következő programrészletet:

```
(defclass stream () ())

(defclass buffered-stream (stream) ())
(defclass disk-stream (buffered-stream ()))

(defclass char-stream (stream) ())
(defclass ascii-stream (char-stream) ())

(defclass ascii-disk-stream
    (ascii-stream disk-stream)
  ())
```

- *(ascii-disk-stream ascii-stream ... stream standard-object t)*

Öröklött tulajdonságok

- A származtatott osztály lokális tulajdonságokkal kibővítheti és felüldefiniálhatja az öröklöttet
- A tulajdonságok beállításai újbóli megadásnál felülíródnak, kivéve a következőket:
  - *:initarg* – a kulcsszó hozzáadódik az eddigiekhez és bármelyiket lehet használni
  - *:type* – a típus-megszorítást konjunkcióval kapcsoljuk, értelemszerűen csak szűkebb típust lehet megadni
  - *:reader* / *:writer* / *:accessor* – ezek nem öröklődnek, de az általuk definiált függvények a metódushívás szabályai szerint természetesen használhatóak

Sablonok és metódusok

- Az osztályok segítségével specializálhatunk függvényeket (polimorfizmus)
- Először egy sablont kell definiálni:

```
(defgeneric függvény-név (paraméterek*)
  beállítások*)
```

- A beállítások megadhatnak:
  - Dokumentációt
  - A specializáció módjának hangolását (ld. később)

A metódusok kiválasztása

- A metódus definíciójában egy vagy több paraméter osztályát specifikálni lehet
- Lehet specializálni konkrét példányra is
- A specifikált paraméter egy lista:
  - *(paraméter osztály)*
  - *(paraméter (eql objektum))*
- A nem specifikáltaknál az alapértelmezés a *T*
- A definíció formája a következő:

```
(defmethod függvény-név mód* (paraméterek*)
  ...)
```

A kiválasztás egyértelműsége

- A konkrét példány specifikusabb az osztálynál
- A specializáció egyértelmű, ha egyszeres
- Többszörös esetben befolyásolja a specializált változók összehasonlítási sorrendje
- Alapesetben balról jobbra halad, de a sablon definíciójában az *:argument-precedence-order* kulcsszóval beállítható
- Például itt az első függvény fog meghívódni:

```
(defmethod func ((p1 class1) p2) () ...)
(defmethod func (p1 (p2 class2)) () ...)
(func object-of-class1 object-of-class2)
```

A metódusok összekapcsolása

- A legspecifikusabb (“fő”) metódus hívódik meg
- Mégis, más metódusok is beleszólhatnak
- A metódus *módja* lehet:
  - *:begin* – a főmetódus előtt fut le (“előmetódus”)
  - *:after* – a főmetódus után fut le (“utómetódus”)
  - *:around* – a főmetódus körül fut le (“körülmetódus”)
- Tehát metódushíváskor:
  - Lefut a körülmetódusok eleje, majd az előmetódusok, először a specifikusabbak, aztán az általánosabbak
  - Lefut a főmetódus
  - Lefutnak az utómetódusok, majd a körülmetódusok vége, először az általánosabbak, majd a specifikusabbak

A metódusok felépítése

- A “specifikusságban következő” metódus a *call-next-method* függvénnyel érhető el
- Ezt kell a körbemetódusokban meghívni, hogy sorra kerüljenek a többi metódusok is
- Az elő-, utó- és körbemetódusok a főmetódushoz kapcsolása több módon történhet (beállítható a sablonban a *:method-combination* kulccsal):
  - standard, progn, and/or, +, min/max, list/append/nconc
  - Újfajta kapcsolási mód is definiálható a *define-method-combination* függvénnyel

Objektumok létrehozása

- Objektumokat a *make-instance* fügvénnyel kell létrehozni:

```
(make-instance 'osztály :kulcs paraméter...)
```

- Érdemes (szokás) azonban *make-osztály* nevű függvényt létrehozni, ami egyben a konstruktor feladatát is ellátja
- Konstruktort lehet írni az *initialize-instance* metódus utófüggvényével is
- Az objektumok megszüntetéséért a GC felelős

Egységbezárás

- Nem teszi kötelezővé
- Megoldható modulokon keresztül:
  - A modulban definiálható az export felület
  - Példa:

```
(in-package :cl-user)
(defpackage :csomag (:use :common-lisp)
  (:export :exportált-függvény1
           :exportált-függvény2
           ...))
(in-package :csomag)
...
```

Amiről nem lesz szó...

- CLOS MOP:
  - Meta-Object Protocol
  - Az osztályokat metaosztályok objektumainak tekinti
  - Nem része az ANSI szabványnak
  - A legtöbb fordító azonban támogatja, azaz
    - A CLOS-t a MOP-on keresztül definiálják
    - A MOP függvényei elérhetőek valamilyen fordító-függő csomagban (pl. SBCL esetén az SB-MOP csomag)
- Az alábbiak nincsenek Common Lispben:
  - Absztrakt osztályok, interfészek
  - Beágyazott, ill. tagosztályok

Példa

Többszörös kötés (Java, CL)

Kivételkezelés

- A “hagyományos” kivételkezelés
  - A hiba helyén kivételt dob
  - A kivétel addig adódik tovább felfele a hívási láncon, amíg le nem kezelik
  - A lekezelő gyakran több szinttel magasabban helyezkedik el (a magasszintű programrészek tudják, hogy mi a teendő)
  - Ekkor viszont a verem már visszafejtődött, nem igazán lehet értelmesen folytatni / javítani
- Common Lispben is kezdetben ez volt
  - *throw* / *catch* (ma már elavult, nem használják)

A Common Lisp kivételkezelése: “Condition System”

- Kivételeket a *define-condition* függvénnyel adhatunk meg, ugyanúgy, mint egy sima osztályt
- Kivételeket lehet dobni a következő parancsokkal: *error*, *cerror*, *warn*, *signal* (mindegyik más)
- A kivételek “elkaphatóak” három módon:
  - *handler-case* – sima lekezelés
  - *restart-case* – egy folytatási lehetőséget definiál, de továbbdobja a kivételt
  - *handler-bind* – lekezelés a verem visszafejtése nélkül
- A *restart*ok az *invoke-restart* függvénnyel hívhatóak meg

Megfeleltetés Javával

- Java:

```
try {
  vmi();
  megVmi();
}
catch (Kivetel k) {
  helyrehoz(k);
}
```

- Common Lisp:

```
(handler-case
    (progn
      (vmi)
      (meg-vmi))
  (kivetel (k)
    (helyrehoz k)))
```

A *restart-case* és *handler-bind*

- A *restart-case* használata:
```
(restart-case kifejezés
  (név1 (paraméterek1*) test1)...)
```
- A *handler-bind* használata:
```
(handler-bind ((kivétel1 függvény1)...)
  (kifejezés)...)
```
- Ezek alkalmazásával megoldható, hogy az alacsonyszintű kezelési módok közül magas szinten válogathassunk

Példa: restart-case / handler-bind

Kivételkezelés összefoglaló

- A kivételek (örökölhető) osztályok (paraméterezhető) példányai
- Függvények (makrók) segítségével használható
- Megadható *restart* függvény, a program folytatódhat az eredeti és magasabb szintről is
- A kezeletlen kivétel típusától függően kezelődik (*error* esetén meghívódik a debugger stb.)
- Kivételes és normális esetben egyaránt végrehajtandó részekhez *unwind-protect*:

```
(unwind-protect kifejezés utána*)
```

Források

- S. E. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1989
- P. Seibel, *Practical Common Lisp*, Apress, 2005
- P. Seibel @ Google TechTalks (2006.05.10.)
- Más fontos könyvek a (Common) Lisp nyelvvel kapcsolatban:
  - H. Abelson, G. J. Sussman, *Structure and Interpretation of Computer Programs, 2nd Ed.*, MIT Press, 1996
  - P. Graham, *ANSI Common Lisp*, Prentice Hall, 1995
  - P. Graham, *On Lisp*, Prentice Hall, 1993
  - G. Kiczales, *The Art of the Metaobject Protocol*, MIT Press, 1991
  - P. Norvig, *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann, 1992
  - Ch. Queinnec, *Lisp in Small Pieces*, Cambridge University Press, 1996
  - G. L. Steele, *Common Lisp the Language, 2nd Ed.*, Digital Press, 1990