Kelemen Dávid

Multimetodok Objektum-Orientált Környezetben

Bevezetés

A multimetodok azonos nevű függvények csoportja, melyek csupán az argumentumok típusában különböznek, a megfelelő változat meghívása pedig az argumentum dinamikus típusától függ. Bár a multimetodok használata növeli a programozási nyelv kifejezőerejét, csökkenti a kódismétlés mértékét és alapvető problémák megoldása lehet (pl. bináris metódusok), költsége, a modularitás esetleges megbontása és használatának bonyolultsága miatt a mai modern programozási nyelvek ritkán támogatják – főleg nem az objektumorientált programozási nyelvek.

Ezen tanulmány célja bemutatni a multimetodok előnyeit és korlátait. Először bemutatok néhány gyakorlati feladatot, mely gyakran visszatér a multimetodok tárgyalása során. Ezután betekintést nyerünk a különböző *dispatching* változatokba. Utána megnézzük, mik lehetnek a multimetodok előnyei, hátrányai, illetve, hogy miért nem terjedtek el a ma széleskörben használt programnyelvek esetén. Az ezt követő részben, mint cseppben a tengert ismerjük meg a multimetodok működését a gyakorlatban a Cecil programozási nyelv esetén. Végül néhány gyakorlati példán keresztül kerül bemutatásra, hogy a multimetodok milyen formában vannak jelen a ma használt programozási nyelvekben.

Példa feladatok

Aszteroidák és űrhajók

Készítsünk egy *SpaceObject* osztályt, melyből két alosztály származik: *Asteroid* és *Spaceship*. Hozzunk létre egy publikus, összeütközést szimuláló *collide* metódust, mely a két objektum típusától függően viselkedik.

A feladat nehézsége, hogy bár az objektum dinamikus típusa jellemzően kiderül az objektumorientált programozási nyelvekben, viszont az argumentumok statikus típusa ismert csak futási időben is. Ebből kifolyólag mindegy, hogy egy aszteroidát mivel ütköztetünk, úgy fog viselkedni, mintha űrobjektummal találkozott volna.

```
public static void main(String[] args) {
    SpaceObject ship1 = new SpaceShip();
    SpaceObject ship2 = new SpaceShip();

    SpaceObject asteroid1 = new Asteroid();
    SpaceObject asteroid2 = new Asteroid();

    ship1.collideWith(ship2);          // SpaceShip <=> SpaceObject
    ship1.collideWith(asteroid2);      // SpaceShip <=> SpaceObject
    asteroid1.collideWith(ship2);      // Asteroid <=> SpaceObject
    asteroid1.collideWith(asteroid2);  // Asteroid <=> SpaceObject
}
```

Pontok és színes pontok

Készítsünk egy *Point* és egy belőle leszármazó *ColorPoint* osztályt – előbbi két privát argumentummal rendelkezik (koordináták), utóbbi emellett egy sztring attribútummal is (szín). Definiáljuk a pontok egyenlőségét ellenőrző *equals* metódust, mely paraméterül egy másik pontot kap.

A feladat rámutat a bináris metódusok problémájára [1]. Egy *equals* metódus elméleti szinten szimmetrikus operátort fejez ki, viszont az objektumorientáltság jellemzői miatt szükségszerűen aszimmetrikus lesz, hiszen az egyik objektum hívja meg, a másikkal paraméterül. A másik probléma az, hogy egy ilyen operátor elképzelhető, hogy privát argumentumok egyenlőségét is ellenőrizni kívánja, ez pedig megszegné az OOP elveket, vagy pedig csak ennek a függvénynek a kedvéért írunk gettereket az argumentumokhoz.

Formarajzolás

Készítsünk egy formákat reprezentáló *Shape* osztályt, melyből leszármaznak a kört (*Circle*), háromszöget (*Triangle*) és négyzetet (*Square*) reprezentáló osztályok. Vegyük különböző implementációit a kimeneti eszközöket reprezentáló *OutputDevice* objektumnak, pl. *ColoredOutputDevice* és *BlackAndWhiteOutputDevice*. Írjuk meg a draw metódust, mely két paramétert vár: egy alakzatot és egy kimeneti eszközt vár paraméterül. A metódus hívása az alakzat és az eszköz típusától függjön.

Ebben az esetben mindkét argumentum dinamikus típusára szükség van a megfelelő működéshez, melyet a széleskörben használt *single dispatch* módszer nem tesz lehetővé.

Dispatching

Mint láttuk, változatos problémák fogalmazhatók meg, melyek megoldása nem, vagy csak nehézkesen oldhatók meg az elterjedt programozási nyelvek eszköztárával. Például a Java programozási nyelvben megtalálható *instanceof* operátor segítségével ellenőrizhető egy argumentum dinamikus típusa. Viszont, ahogy a procedurális programozási nyelvek esetén is szimulálható az objektumorientáltság, a hangsúly a kényelmes és elegáns használaton van. Többszörös elágazásban használva az *instanceof* operátor látszólag felesleges „spagettikóddal” jár, holott az argumentumok dinamikus típusának használata olvashatóbb és átláthatóbb kódot eredményezne.

Az utóbbi példában említésre kerül a *single dispatching*. Ennek kifejtése a következő alfejezetben történik meg.

Single dispatching

Az objektumorientált programozási nyelvek leggyakrabban a metódusválasztást csak az üzenetfogadó objektum futási idejű típusának figyelembevétele alapján történik. Ez a megközelítés gyakran jól megállja a helyét, hiszen általában az objektum, aminek egy metódusát hívjuk, „érdekesebb” a metódusnak átadott argumentumoknál. Másrészről előfordulnak olyan esetek, amikor nincs okunk kitüntetett figyelmet fordítani egy argumentumnak. Erre egy példa a fent említett *equals* metódus, ahol természetellenes csak az egyik pontnak figyelembe venni a dinamikus típusát. Emellett az objektumrajzoló programunk esetén is, nincs okunk sem az alakzat, sem a kimeneti eszköz típusát kevésbé fontosnak tekinteni, ahogy két űrobjektum ütközésénél is a résztvevő példányok azonos jelentőséggel bírnak.

Double dispatching

Az aszimmetria problémájának egyik megoldása kíván lenni a *double dispatching*. Ebben az esetben a programozó minden „érdekesnek” ítélt argumentumra sorra használhatja a *single dispatching* módszerét. Például, amikor használni szeretnénk a *Point* osztály *equals* metódusát egyéb típusú objektumokkal is, ezt az alábbi módon tehetjük meg (forrás: [2]):

```
-- in Point:
   self = aPoint { return aPoint.equalsPoint(self); }
   self.equalsPoint(originalSelf) {
      return originalSelf.x = self.x && originalSelf.y = self.y; }
-- in Object:
   self.equalsPoint(originalSelf) { return false; }
```

A megközelítés legnagyobb problémája, hogy minden *dispatched* argumentumhoz legalább két metódust kell készíteni – egy, ami megkezdi a *double dispatch* folyamatát azáltal, hogy tovább küldi az argumentumot egy specifikus függvénynek adott típus esetén; valamint egy másik, ami az alapértelmezett működést biztosítja egyéb típusok esetén. Jól látszik, hogy ez a módszer nem igazán karbantartható, emellett pedig nagy a hibázás kockázata is.

Multiple dispatching

A *multiple dispatching* programozási nyelvekben több argumentum is részt vehet a metódusválasztásban, ezzel nagyobb kifejezőerőt kölcsönözve. Ezen programozási nyelvekben írt metódusokat multimetodoknak hívjuk. Itt válik lehetővé, hogy több argumentum is „érdekesnek" legyen tekinthető. Például az összeadás operátor definiálható két egész szám, két lebegőpontos szám, vagy vegyes típusú argumentumok esetén is.

Az alábbi példa mutatja a pontok egyenlőségét vizsgáló multimetodot (példa: [2]). Alapesetben két objektum egyenlősége hamis, viszont, ha mindkét argumentum típusa *Point*, akkor a koordináták egyenlőségén múlik a visszatérési érték.

```
-- the default implementation of equality returns false (don't dispatch on either argument):
x = y { return false; }

-- implementation of equality for a pair of points
-- (v@obj means dispatch on argument v, and match only for actuals that are equal to or inherit from obj):
p1@Point = p2@Point {
   return p1.x = p2.x && p1.y = p2.y; }
```

Ezzel a módszerrel elkerülhető a *double dispatching*, a problémáival együtt. A kevesebb „spagettikód" miatt a programozónak több idő marad az üzleti logikát elkészíteni. Az előbbi módszerhez képest egyszerűbb átlátni az argumentum kombinációkat, így kisebb a hibázás lehetősége is.

Multimetodok és objektum-orientáltság

A *multiple dispatching* a korábbiak alapján nagyobb kifejezőerővel bír, természetesebb, olvashatóbb, ls kisebb hiba-rizikóval bír a *single dispatching* módszeréhez képest. Ennek ellenére a ma népszerű programozási nyelvek jellemzően nem építették eszköztárukba ennek a lehetőségeit. Ennek legfontosabb oka, hogy bonyolultabb módszer, és kevésbé hatékony az implementációja. Viszont, ha ettől eltekintünk, akkor is azt látjuk, hogy a *single dispatching* használói jellemzően nem érzik objektum-orientáltnak a multimetodokat.

A *single dispatching* programozási nyelveknél a programozó definiál objektumokat, melyekhez tartoznak argumentumok és metódusok. Az objektumok öröklődési hierarchiába rendeződnek. *Multiple dispatching* esetén a metódusok kevésbé kötődnek hozzá az objektumhoz. Például a CLOS programozási nyelvben generikus függvények vannak, ahol az azonos nevű multimetodok csoportosulnak. Ebből kifolyólag a program decentralizálódik, az implementáció inkább a funkcionális programnyelveknél ismert

mintaillesztéshez hasonlít. Távolabbról nézve, a multimetodok bizonyos szempontból integrálja a funkcionális és az objektumorientált programozási stílust.

Kérdéses viszont, hogy a multimetodok hogy illeszthetők be az objektumorientált megközelítésbe. A generikus függvények megközelítése szerint a multimetod nem tartozik egy objektumhoz sem. Ezzel szemben, az adatabsztakció-orientált programozási megközelítés alapján a multimetodot több objektum is tartalmazhatja. Ez utóbbi sajnos szintén problematikus, hiszen megfelelő keretrendszer kell a program szervezéséhez, viszont van rá esély, hogy objektumorientált környezetbe is beilleszthető legyen, amennyiben a programozási nyelv magáévá teszi az adatabsztakció-orientált programozást.

További probléma a generikus függvény megközelítéssel az enkapszuláció: ha a multimetodot objektumon kívülinek tekintjük, nincs lehetőség priváttá tenni, és az osztály privát metódusait sem érheti el. Emiatt is az adatabsztrakció-orientált szemlélet a preferált, hiszen ekkor minden releváns objektum részének tekinthető a multimetod – amikre illeszkedik az argumentum megkötés – így ezekhez kiváltságos hozzáférés is adható.

Cecil [2]

A multimetodok objektumorientált környezetben való használatára tett kísérletet a Cecil programozási nyelv, mely az alábbi tervezési megközelítéseket követte: a metódusválasztásban minden argumentum részt vesz; a multimetodok több osztályhoz is „tartozhat”; a programozási környezet támogatja az adatabsztrakció-orientált programozást; a multimetod hozzáfér a metódusválasztásban részt vevő argumentumok típusainak privát adattagjaihoz.

Az osztályok és azok öröklődése az objektumorientált nyelvekhez hasonló módon lehetséges, bővebb bemutatása: [2].

Metódusok

Cecilben írt metódusok esetén minden argumentumhoz megadhatunk bizonyos típuskorlátozásokat, melyek hatására az illeszkedés csak akkor teljesül, ha az átadott argumentum típusa megegyezik a korlátozóéval, vagy pedig annak leszármazottja. Korlátozás nélküli argumentum bármely típusra illeszkedik. Az alábbi szabályok alapján bármennyi argumentum korlátozható:

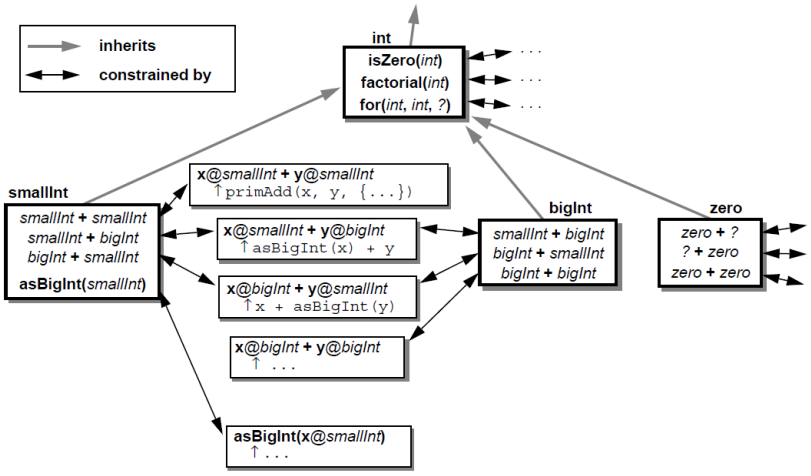
- A korlátozást nem tartalmazó metódusok egyszerű eljárások.
- Az egy argumentumot korlátozó metódus a *single dispatching* szabályai szerint működik
- Amennyiben több metódus is korlátozott, valódi multimetodról beszélünk

A metódust meghívó objektum nem tud a korlátozásokról, így arról sem, hány multimetod vesz részt a funkcionalitás létrehozásában. Például:

```
x@smallInt + y@smallInt {
    -- primAdd performs primitive arithmetic of (children of) primInt
    -- primAdd takes a failure block which is invoked if an error (e.g., overflow) occurs
    ↑ primAdd(x, y, { &errorCode | -- code to handle failure (e.g., retry as bigInts) -- } ) }
```

A fent látható összeadás operátor multimetodnak tekinthető, mivel két argumentumra is megkötést teszünk: mindkettő típusa *smallInt* vagy annak leszármazottja kell legyen ahhoz, hogy a metódus kiválasztásra kerüljön. Ahogy a következő alfejezetben látni fogjuk, ezen argumentum megkötések szerves részét fogják képezni a metódusválasztásnak.

Mivel a multimetodok több osztályhoz is kapcsolódhatnak, az átláthatóság érdekében szükség van olyan vizualizációra, ami segít jól áttekinteni az összeköttetéseket. Erre egy példa az alábbi ábra:



Metódus választás

A multimetodok egy sarkalatos kérdésköre, hogy miként válasszuk ki a lefuttatni kívánt metódusvariációt. A probléma hasonló a többszörös öröklődést megvalósító programozási nyelveknél fellépőnél: ha két metódus eltérő argumentumkorlátozásokkal rendelkezik, de mindkettő alkalmazható? Hogyan válasszunk közülük, ha egyik metódus sem egyértelműen specifikusabb a másiknál?

Ezt a problémát különböző programozási nyelvek eltérően oldják meg. A CLOS programozási nyelvben az argumentumok pozíció szerinti prioritási sorba rendeződnek: egy argumentum teljes mértékben dominálja a tőle jobbra levőket. Ezzel a linearizációval a futás idejű kétértelműségek sikeresen eltüntethetők, viszont benne foglaltatik a kétértelműségek automatikus feloldásából származó lehetséges hibalehetőség is, melyekről a programozó nem feltétlenül szerez tudomást.

A Cecil programozási nyelv egy eltérő megoldást kínál a problémára, ahol a programozó visszajelzést kap az esetleges hibákról. Ehhez az alábbi parciális rendezést definiálja a multimetodokhoz:

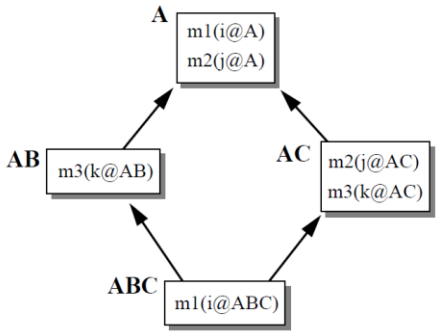
- Egy C objektum nagyobb egy P objektumnál akkor és csak akkor, ha C egy leszármazottja P-nek
- Egy M metódus nagyobb egy azonos nevű N metódusnál akkor és csak akkor, ha minden *i* argumentumpozícióra az M i. argumentumának korlátozása nagyobb N i. argumentumának korlátozásánál
- A korlátozott argumentumok nagyobbak a nem korlátozottaknál

Akkor választjuk N metódust M-nel szemben, ha N a rendezés szerint nagyobb M-nél. Viszont a módszer veszélye a kétértelműség, amennyiben két metódus a parciális rendezés alapján nincs relációban. A megfelelő metódus kiválasztása az alábbiak alapján történik:

- Gyűjtsük össze az összes metódust, aminek a neve és argumentumszáma megegyezik a hívásban használt metódussal, valamint az argumentum megkötések is illeszkednek
    - Amennyiben ez a halmaz üres, hiba történik (*„Message not understood”*)
- Kiválasztjuk az egyetlen, a parciális rendezés szerinti legnagyobb metódus változatot, ezzel térünk vissza

- Amennyiben a parciális rendezés alapján nincs legnagyobb, hiba történik (*„Message ambiguous”*)

Az alábbi példa segít belelátni a metódusválasztás logikájába:



Itt az A objektumból öröklődik az AB és az AC objektum, melyekből az ABC objektum öröklődik. Tehát ABC > AB, ABC > AC, ABC > A, AB > A, és AC > A. Az AC és az AB viszont nincsenek rendezve egymáshoz képest. Például, ha meghívjuk az *m1* metódust, paraméterül egy ABC objektumot megadva, az az *m1(i@ABC)* hívódik meg, hiszen dominálja az *m1(i@A)* metódust. Az *m2* metódus ABC objektummal hívásakor viszont a legjobban illeszkedő változat az *m2(j@AC)*, így az kerül meghívásra. Továbbá, ha az *m3* metódus ABC objektumot kap paraméterül, hiba történik: a paraméter illeszkedik az AB és az AC lehetőségekre is, rendezés viszont nincs köztük, kétértelműség miatt a program hibát dob.

Enkapszuláció

Az adatabsztrakció-orientált programozásban, a multimetodokra úgy tekintünk, mint amik erősen kötnek az argumentumban megszorítottan átadott paraméterek típusaihoz, és csak gyengébben egymáshoz. Az enkapszuláció viszont nem kerülhető meg, ha objektumorientált környezetben szeretnénk egy módszert alkalmazni. A legtöbb ilyen programozási nyelvben az adattagok publikus és privát láthatóságra vannak osztva. A publikus operációk kívülről és belülről is elérhetők, a privát metódusokat viszont csak az objektum maga hívhatja meg. Viszont a multimetodok nem kapcsolódnak kivételes mértékben egyik osztályhoz sem, mégis fontos lenne megoldani, hogy a releváns objektumok privát adattagjaihoz hozzáférjen, valamint azt, hogy egy multimetod privát legyen.

A probléma egy megoldása lehet az a megközelítés, hogy a multimetod minden objektumhoz tartozik, ami megjelenik argumentum megszorításban. Nem-megszorított argumentumok típusaihoz nincs kitüntetett hozzáférése a multimetodnak. Ez a megközelítés logikusnak tűnhet a gyakorlati programozás szemszögéből. Ha egy argumentum érdekel minket annyira, hogy korlátozást definiáljak rá, természetesnek tűnik, hogy hozzáférhessünk az objektum implementációjához is. Például:

```
cons = object inherits list
private head(c@cons) { field }
private tail(c@cons) { field }
size(c@cons) { ↑ 1 + size(c.tail) }
do(c@cons, block) {
   eval(block, c.head)
   do(c.tail, block) }
pairDo(c1@cons, c2@cons, block) {
   eval(block, c1.head, c2.head)
   pairDo(c1.tail, c2.tail, block) }
```

Az előbbi példában egy láncolt lista megvalósítása látható. Bár a *head* és a *tail* privát argumentumok, a *do* és a *pairDo* metódusok hozzáférnek ezekhez is, hiszen a *cons* objektumra van korlátozva egy (illetve kettő) argumentum. További előny, hogy ha az objektum implementációja változik, könnyű megtalálni azokat a multimetodokat, amelyekre hatással lehet: azok, amelyek argumentuma arra az objektumra van korlátozva.

A privát multimetodok esetén egy újabb filozófiai kérdéssel kerülünk szembe: a privát multimetod privát minden minden argumentum-megszorításra külön-külön, vagy pedig privát, mint argumentumok csoportja? Az alábbi példa szemlélteti a problémát:

```
displayOn(shape@filledPolygon, device@fancyGraphicsHardware) {
   setUpDisplay(shape, device)
   ... rest of code ... }
private setUpDisplay(shape@filledPolygon, device@fancyGraphicsHardware) {
   ... initialize graphics hardware for filled polygon displays ... }
```

A *setUpDisplay* metódus privátnak lett deklarálva, tehát *filledPolygon* és *fancyGraphicsHardware* típusú attribútumokkal meghívva elérhető. De mi történik, ha az alábbi függvényt definiáljuk?

```
draw(shape@filledPolygon) {
   setUpDisplay(shape, standard_display())
   ... more code ... }
```

Ha a *standard_display()* visszatérési értékének dinamikus típusa nem *fancyGraphicsHardware*, akkor a privát *setUpDisplay()* függvény elérhető, vagy nem? Az első megközelítés alapján, mivel a *draw* metódus argumentumot korlátoz a *filledPolygon*-ra, ezért a hívás valid, a *draw* metódusnak kitüntetett hozzáférése van a privát *setUpDisplay* multimetodhoz. A második megközelítés ezzel szemben azt mondja, hogy a *draw* metódus nem érheti el a *setUpDisplay* metódust, hiszen nem minden szükséges argumentumra tesz korlátozásokat. A Cecil ez utóbbi megközelítést követi.

A fenti szabályok viszont egy nagy veszélyt is hordoznak magukban: az enkapszuláció könnyen megbontható azáltal, hogy definiálunk egy megfelelő multimetodot, ami azután hozzáfér a privát adattagokhoz egy objektumban. Bár a Cecil egyik legnagyobb erőssége az egyszerű bővíthetőség, nem feltétlen üdvözölendő, hogy ilyen egyszerűen hozzá lehessen férni objektumok belső implementációjához. Ennek kiküszöböléséhez lehetőség van explicit felsorolni azokat a multimetodokat, melyek kitüntetett hozzáférést kaphatnak az adott objektumhoz. Ezzel egy „hozzáférésszabályozó listával” bővül az enkapszulációs szabályhalmaz. További multimetodok készíthetők az objektumhoz, viszont azok nem férhetnek hozzá a privát adattagokhoz.

Az enkapszulációs szabályok megengedik, hogy egy multimetod egy szülő objektum illeszkedése esetén hozzáférjen egy leszármazott privát metódusaihoz. Bár ez a jellegzetesség szintén megkérdőjelezhető, az alábbi okokból mégis indokolt:

- Egy objektum kivételes hozzáféréssel kell rendelkezzen a saját privát adattagjaihoz
- Egy leszármazottnak lehetősége kell legyen, hogy felüldefiniálja az őstől örökölt privát metódusokat, és ez a saját verzió szintén lehet privát láthatóságú
- Ha egy objektum hozzáfér a saját metódusaihoz, akkor a leszármazott felüldefiniált verzióját is meg kell tudnia hívni

Tehát ahhoz, hogy a szülő hozzáférhessen a leszármazott által felüldefiniált metódusváltozathoz, kitüntetett hozzáféréssel kell rendelkeznie a leszármazott metódusaihoz. Viszont ez a lehetőség is korlátozható: az enkapszulációs szabály kibővíthető azzal a megszorítással, hogy az ős csak azokhoz privát metódusokhoz férhetnek hozzá a leszármazott osztályban, amelyek felüldefiniáltak, és az eredeti változathoz az ős hozzáférhetne. Amennyiben az ős nem rendelkezik megfelelő implementációval, egy absztakt metódus bevezetésével fenntarthatja a kitüntetett hozzáférést. Ezt a lehetőséget a Cecil jelenleg nem támogatja.

Gyakorlati felhasználás

Mint láttuk, a multimetodok növelhetik a programozási nyelv kifejezőerejét, és kényelmes, intuitív használatot tesznek lehetővé. Azt is láttuk, hogy az objektumorientált programozási paradigmába beilleszthető a módszer, bár bizonyos új problémákat hoz, mely a fordítóprogramok íróinak kihívást jelenthetnek, illetve a fejlesztőktől is körültekintő használatot várnak el.

Bár széles körben nem terjedtek el a multimetodok, lehetőségünk van ma is széleskörben használt programozási nyelvekben használni őket. Ebben a fejezetben betekintést nyerünk néhány gyakorlati felhasználási lehetőségbe. Először Python programozási nyelvben használható multimetodokat nézzük meg, majd a Java egy bővítését, a Relaxed MultiJava-t. Végül egy példát mutatok arra, hogyan szimulálhatók a multimetodok a C programozási nyelvben.

Python

A Python programozási nyelv kibővíthető a *multimetod* nevű csomaggal, melynek segítségével annotációval és a paramétertípusok megadásával egyszerűen használhatók a multimetodok. Példámban a fent bemutatott úrobjektum feladata segítségével mutatom be ezt a funkcionalitást.

```
from multimethod import multimethod


class SpaceObject:
    pass


class Asteroid(SpaceObject):
    pass


class Spaceship(SpaceObject):
    pass
```

Ahogy látható, a *SpaceObject* ősosztály két leszármazottja a *Spaceship* és az *Asteroid*. Definiáljunk emellett egy *Universe* világegyetemet reprezentáló osztályt, mely szabályozza az objektumok ütközését:

```
class Universe:
    @multimethod
    def collide(self, x: SpaceObject, y: SpaceObject):
        print("SpaceObject <--> SpaceObject")
    @multimethod
    def collide(self, x: Asteroid, y: Asteroid):
        print("Asteroid <--> Asteroid")

    @multimethod
    def collide(self, x: Asteroid, y: Spaceship):
        print("Asteroid <--> Spaceship")

    @multimethod
    def collide(self, x: Spaceship, y: Asteroid):
        print("Spaceship <--> Asteroid")

    @multimethod
    def collide(self, x: Spaceship, y: Spaceship):
        print("Spaceship <--> Spaceship")
```

Az *Universe* osztály azt a *collide* metódust fogja alkalmazni, amely a legjobban illeszkedik az argumentumtípusokra. Tehát az alábbi példakód az alábbi kimenetet fogja adni:

Kód	Kimenet

`space_objects = [`
`  (Asteroid(), Asteroid()),`
`  (Asteroid(), Spaceship()),`
`  (Spaceship(), Asteroid()),`
`  (Spaceship(), Spaceship())`
`]`
 	
`universe = Universe()`
`for i in space_objects:`
`  universe.collide(i[0], i[1])`
	`Asteroid <--> Asteroid`
	`Asteroid <--> Spaceship`
	`Spaceship <--> Asteroid`
	`Spaceship <--> Spaceship`

További lehetőség például az *overload* annotáció, ahol nem az illeszkedés mértéke, hanem a metódusok sorrendje határozza meg, melyik fusson le:

```
from multimethod import isa, overload


@overload
def func(obj: isa(str)):
    print(f'{obj} is a string')


@overload
def func(obj: str.isalnum):
    print(f'{obj} is alphanumeric')


@overload
def func(obj: str.isdigit):
    print(f'{obj} is digit')


def islong(x):
    return len(x) > 10


@overload
def func(obj: islong):
    print(f'{obj} is long')
```

A metódusválasztás lentről felfelé történik, és az fog lefutni, amelyik először illeszkedik. Az illeszkedés egyedi függvény segítségével is ellenőrizhető, ahogy az *isLong* metódus is átadható – ez akkor ad igazat, ha a paraméter hossza 10-nél nagyobb.

```
func('12345678912345')
func('123')
func('Cat33')
func('$5g b|!@')
```

Példámban az első hívás ellenőrzi, hogy az *isLong* metódus igazat ad-e a paraméterül adott sztringre. Mivel igen, ezért az fut le, a válasz: *„12345678912345 is long”*. A második esetben az *isLong* metódust hamissal tér vissza, így az illeszkedés nem történik meg. Viszont lentről a második metódusváltozat illeszkedik, hiszen csak számjegyekből áll a sztring, a kimenet: *„123 is digit”*. Hasonlóan a harmadik ill. negyedik példa kimenete *„Cat33 is alphanumeric”* ill. *„$5g b|!@ is a string”*.

MultiJava [3]

A MultiJavaegy kiterjesztése a Java programozási nyelvnek, mely lehetővé teszi externális metódusok hozzáadását meglévő osztályokhoz, valamint multimetodok készítését is. Ennek egy bővítést a Relaxed MultiJava, amely rugalmasabb működést tesz lehetővé, és nagyobb szabadságot ad a programozónak.

A legfontosabb különbség a két bővítés között, hogy potenciális illesztési hibák csupán fordítási idejű figyelmeztetést ad, és nem omlik össze a program, valamint további ellenőrzéseket futtat betöltés-időben. A rugalmasság nem jelent felületességet, a MultiJava esetén hibaként jelzett esetek a Relaxed MultiJava esetén is látható, legfeljebb csak figyelmeztetésként.

A nagyobb engedékenység természetesen nagyobb kifejezőerőt jelent. Például, ha van egy potenciális kétértelműség, amiről a programozó tudja, hogy futás közben sosem fog előfordulni, a MultiJava nem engedte futni, a Relaxed Multijava viszont egy figyelmeztetést követően futtatja a kódot. Természetesen a MultiJava pedig nagyobb kifejezőerejű, mint a klasszikus Java programozási nyelv.

A MultiJava és a Relaxed MultiJava ugyanarra a Java bájtkódra fordul, mint a Java, és tökéletesen együttműködik más Java forrásfájlokkal.

A multimetod választásnál az egyetlen legspecifikusabb, alkalmazható változat kerül meghívásra, ami felüldefiniálja a többi változatot. Ha nincs alkalmazható változat, vagy nincs olyan egyetlen változat, ami a legspecifikusabbnak tekinthető, hiba lép fel. Emellett lehetnek *bevezető metódusok*, melyek nem írnak felül semelyik másik metódust.

A multimetodok itt hasonló szintaktikával rendelkeznek, mint a korábban látott Cecil programozási nyelvnél: az argumentum megkötések a ’@’ jelet követően kerülnek definiálásra, ahogy az alábbi példa is mutatja:

```
package RectanglePackage;
import ShapePackage.*;
import OutputPackage.*;
public class Rectangle extends Shape {
  ... implementations of Shape methods ...
  public void draw(OutputDevice@BWPrinter p)
  { ... code for drawing a Rect. on a b&w printer ...
  }
}
```

```
package CirclePackage;
import ShapePackage.*;
import OutputPackage.*;
public class Circle extends Shape {
  ... implementations of Shape methods ...
  public void draw(OutputDevice@BWPrinter p)
  { ... code for drawing a Circle on a b&w printer ...
  }
}
```

A fenti példában a *Rectangle* és a *Circle* osztály *draw* metódusának egy speciális változata akkor fog csak lefutni, ha a paraméterül kapott *OutputDevice* futás-idejű típusa *BWPrinter*.

A Relaxed MultiJava végrehajtási időben mérsékelten rosszabb hatékonyságú, mint a Java programozási nyelv hasonló feladatokon.

C – multimetodok szimulálása [4]

Végül, érdekességképp nézzük meg, mit tehetünk, ha a programozási nyelvünk nem támogatja a multimetodokat. Példámban ez a C nyelv lesz, a megoldandó feladat pedig ismét az űrobjektumok ütköztetése.

```c
#include <stdio.h>

typedef void (*CollisionCase)(void);

void collision_AA(void) { printf("Asteroid <--> Asteroid\n"); };
void collision_AS(void) { printf("Asteroid <--> Spaceship\n"); };
void collision_SA(void) { printf("Spaceship <--> Asteroid\n"); };
void collision_SS(void) { printf("Spaceship <--> Spaceship\n"); };

typedef enum {
    asteroid = 0,
    spaceship,
    num_thing_types /* not a type of thing itself, instead used to find number of things */
} Thing;

CollisionCase collisionCases[num_thing_types][num_thing_types] = {
    {&collision_AA, &collision_AS},
    {&collision_SA, &collision_SS}
};

void collide(Thing a, Thing b) {
    (*collisionCases[a][b])();
}

int main()
{
    collide(asteroid, asteroid);
    collide(asteroid, spaceship);
    collide(spaceship, asteroid);
    collide(spaceship, spaceship);
    return 0;
}
```

Ebben az esetben a típusok gyakorlatilag enumerátorként vannak reprezentálva, a különböző viselkedéseket pedig eltérő függvények képviselik, melyek egy mátrixba kerülnek. A *collide* metódus ezután a mátrix megfelelő helyén tárolt metódust futtatja, így szimulálva a multimetod jellegzetes működését.

Összefoglalás

Bár a multimetodok elsősorban a funkcionális programozási paradigmába illenek bele, megfelelő diszciplínát követve nemcsak helyt állnak az objektumorientált környezetben, de az ott jelenlevő bizonyos alapvető problémák megoldásaként is szolgálhatnak. A megvalósításuk új problémák bevezetésével jár a metódusválasztás és az enkapszuláció területén is, de ezek megoldása nem lehetetlen, és feloldásuk várhatóan több előnnyel járna a programozási nyelvre nézve, mint amennyi hátránnyal, például a komplexitást illetően.

Hivatkozások

[1] Bruce, K., Cardelli, L., Castagna, G., The Hopkins Group, Leavens, G. T., Pierce, B. (1995). On Binary Methods. *Theory and Practice of Object Systems*, 1(*3*), pp 221 – 242.

[2] Chambers, C. (1992). Object-Oriented Multi-Methods in Cecil. *ECOOP '92 Conference Proceedings*, Ultrecht, Netherlands.

[3] Millstein, T., Reay, M., Chambers, C. (2003). Relaxed MultiJava: Balancing Extensibility and Modular Typechecking. *OOPSLA'03*, October 26-30, 2003, Anaheim, California, USA.

[4] https://en.wikipedia.org/wiki/Multiple_dispatch#C