A formázás. Az adattárolás megkezdése előtt a merev- és a hajlékonylemezt is formázni kell. A formázás két műveleti szakaszból áll: fizikai- és logikai formázásból. Fizikai formázással a lemezen sávokat és szektorokat hoznak létre. Logikai formázással a lemezt az operációs rendszer adattárolási szabványaihoz illesztik. Ugyanis a DOS vagy Windows operációs rendszer ill. az UNIX vagy LINUX operációs rendszer lemezkezelése különböző. Adatokat tartalmazó lemez esetében figyelembe kell venni, hogy ezeket az adatokat formázás után elveszítjük.

Irodalom

1] *Abonyi Zs.*: PC hardver kézikönyv, Computer Books, Budapest, 1996
2] *Markó I.*: PC Hardver, LSI Oktatóközpont, Budapest, 2000
3] *M. Brain*: How Hard Disks Work, www.howstuffworks.com

Kaucsár Márton

Objektumorientált paradigma

VI. rész

Statikus és dinamikus objektumok

(*kulcsszavak:* statikus objektum, dinamikus objektum, Heap, garbage collection, copy constructor, virtual constructor)

Mint a változóknál általában, az objektumok esetében is beszélhetünk *statikus* és *dinamikus objektumokról*, annak függvényében, hogy a számukra lefoglalt memóriahely melyik memóriazónában van, és mikor történik a helyfoglalás és felszabadítás. Két lényegesen különböző memóriazónáról beszélhetünk: a *Heap*-ről, amelyben a helyfoglalás dinamikusan történik és a statikus részről, amelyben a változók, objektumok élettartamuktól függően vagy az *adatszegmens*ben (*Data Segment* – globális változók), vagy a veremben (*Stack* – lokális változók, paraméterek) találhatók. A statikus objektumoknak szánt helyet az illető objektumot tartalmazó programmodul memóriába töltésekor foglalja le a rendszer és az alkalmazás, és program vagy modul futásának befejezésekor szabadul fel. A dinamikus objektumok helyének lefoglalása pedig a helyfoglaló kódrész végrehajtásakor történik, és a felszabadításáról is teljesen dinamikusan lehet gondoskodni.

Ilyen értelemben beszélhetünk *statikus* és *dinamikus* példányosításról. A statikus példányosítás statikus objektumot hoz létre (egyszerű deklaráció) és a láthatósági terület függvényében az objektum az adatszegmensben vagy a veremben lesz, a dinamikus példányosítás dinamikus objektumot hoz létre (egyszerű deklaráció + dinamikus példányosítás) és a dinamikus objektum számára a Heap-ben foglalódik hely.

Az objektumorientált programozásban az objektumokat általában a program futása közben hozzuk létre, majd mikor már nincs szükségünk rá, szüntetjük meg. Ezért minden objektumorientált nyelv kell, hogy rendelkezzen olyan mechanizmussal (kulcsszavak vagy eljárások szintjén), amely lehetővé teszi az objektumok dinamikus példányosítását és megszüntetését. A leggyakoribb dinamikus példányosító a **new** operátor vagy eljárás. A dinamikus példányosítás pillanatában ajánlatos a konstruktort is meghívni, így biztos, hogy a példányosított objektum inicializálva lesz és a VMT, DMT mezők értékei is jól lesznek kitöltve. Számos programozási nyelv ezt támogatja. Hasonlóan a leépítés, felszabadítás esetén történjen meg a destruktor meghívása is.

Dinamikus objektumok esetén egyik érdekes kérdéskör a *típuskényszerítés*. Dinamikus objektumok esetén megtörténhet az, hogy egy objektumot valamilyen ősosztály típusúnak deklarálunk és egy leszármazott típusúnak példányosítunk. Ekkor típuskényszerítést kell végrehajtanunk ahhoz, hogy az így példányosított objektum saját metódusait meg tudjuk hívni. Az ilyen objektumokat *polimorfikus objektumoknak* nevezzük. Ez azt is jelenti, hogy fordítási időben nem kell ismerni az illető osztály típusát, hanem ez csak futás közben, a dinamikus példányosítás pillanatában derül ki. Ha polimorfikus objektumokat akarunk létrehozni, az ilyen objektumok konstruktorai is virtuálisak kell, hogy legyenek, mert csak így valósítható meg a futás alatti típuskötés (*virtual constructors*). A programozási nyelvek biztosítanak olyan mechanizmusokat, amelyek segítségével meg lehet oldani a típuskényszerítést.

Egy másik kérdéskör az *értékadás kérdésköre* a dinamikus objektumok esetén. Statikus objektumok esetén egyértelmű: egy objektum felveheti egy másik objektum értékét, ha a két objektum ugyanolyan típusú, vagyis vagy megegyezik az osztályuk, vagy az értékadás jobboldalán szereplő objektum osztálya leszármazottja az értékadás baloldalán lévő objektum osztályának. Ekkor az értékadás baloldalán lévő objektum felveszi az értékadás jobboldalán lévő objektumnak az állapotát. Dinamikus objektumok esetén ez nem ennyire egyértelmű, hisz ha egy dinamikus objektum felveszi egy másik dinamikus objektum értékét, akkor ez azt jelenti, hogy mindkét objektum ugyanarra a Heap-beli zónára fog mutatni, így bármelyiknek változtatjuk az állapotát, változik a másik is. Ezek tehát össze vannak kötve, nem két példányban vannak jelen. Létezik egy speciális konstruktor: a *másoló konstruktor* (*copy constructor*), ami megoldja azt, hogy a dinamikus objektumok értékadásakor létrejöjjön még egy példány, lemásolva az értékadás jobboldalán lévő objektum állapotát. Így az objektumok dinamikusan is két példányban lesznek jelen. A másoló konstruktornak kell legyen mindig egy paramétere és ez a megfelelő osztályú másolandó dinamikus objektum lesz.

A harmadik érdekes kérdéskör azt tárgyalja, hogy a dinamikus objektumok felszabadítását nem lehetne-e esetleg automatikusan elvégezni. Számos – inkább értelmező jellegű – nyelv rendelkezik saját automatikus *szemétgyűjtő mechanizmus*sal (*garbage collection*). A Heap használaton kívüli memóriazónáit a szemétgyűjtő mechanizmus deríti fel és szabadítja fel. A szemétgyűjtő a programmal párhuzamosan futó, kis prioritású szálon fut és amikor minden hivatkozás megszűnik egy dinamikus objektumra, automatikusan felszabadítja az általa lefoglalt memóriaterületet, miután automatikusan meghívta az objektum destruktorát.

Ha egy dinamikus objektumot felszabadítottunk (akár manuálisan, akár szemétgyűjtő segítségével) felszabadul a lefoglalt Heap zóna, így erre többet már nem hivatkozhatunk.

Napjaink programozási nyelveiben egyre inkább csak dinamikus objektumokról beszélhetünk (visszafele kompatibilitás céljából megmaradtak a statikus objektumok is, ezek használata azonban kerülendő).

Ha össze akarjuk foglalni a statikus és dinamikus objektumok jellemzőit, akkor a következő táblázatot kapjuk (*A szürkével jelzett rész a szemétgyűjtő mechanizmust jelöli.*):

	Statikus	**Dinamikus**	
Létrehozó	Rendszer	Programozó	
Létrehozás pillanata	Blokkba való belépés	Amikor szükség van rá	
Példányosítás	Deklarálás	Deklarálás + helyfoglalás	
Értékadás	Ugyanolyan típusúak	Másolás vagy referencia átadás	
Felszabadító	Rendszer	Programozó	Rendszer
Felszabadítás pillanata	Blokkból való kilépés	Nincs rá szükség	Időnként

Mitől objektumorientált egy program?

(*kulcsszavak:* objektumokat használó, objektum alapú, objektumorientált, hibrid, eseményorientáltság, szórás, kivételkezelés)

A fejezet elején azt mondtuk, hogy:

Egy objektumorientált program egymással kommunikáló objektumok összessége, melyben minden objektumnak megvan a jól meghatározott feladata.

Ez a definíció, azonban nem annyira egyértelmű, csak elméletben igaz. A gyakorlat más irányvonalakat is megszabott, ezeket próbáljuk most összefoglalni.

a.) Objektumokat használó program

Léteznek olyan programozási nyelvek (általában a szkript nyelvek, makró nyelvek), amelyek nem biztosítanak lehetőséget osztályok definiálására, nem használják ki az öröklődés, a polimorfizmus által nyújtott lehetőségeket, viszont lehetőség van arra, hogy előre definiált objektumokat használhassunk. Ezek a nyelvek tehát kizárólag az egybezártság tulajdonságát használják fel, lehetőséget biztosítva az adat és kódrejtésre. Az objektumokat nem kell példányosítani (hisz nem létezik az osztály fogalma), ezek önmaguktól léteznek, csak használni kell őket: módosíthatjuk állapotukat, meghívhatjuk metódusaikat.

b.) Objektum alapú

Az objektum alapú nyelvek már ismerik az osztály fogalmát, használható az öröklődés, a polimorfizmus. Objektumokat példányosíthatunk és használhatjuk őket. Az objektumok lehetnek statikusak és dinamikusak. Minden objektumnak külön üzenetet kell küldeni, csak így lehet „megszólítani" őket. Az objektumok másképp nem kommunikálnak egymással.

c.) Objektumorientált

Az objektumorientált program lényeges eltérése a fent említett objektum alapú programtól az, hogy az objektumok kommunikálnak egymással. Az objektumok általában dinamikusak, kihasználják a típuskényszerítést és a helyettesíthetőséget. Maga a főprogram is egy objektum (az Alkalmazás, Application objektum) és ő példányosít, indítja el útjukra és felügyeli a többi objektum működését. Minden objektum képes arra, hogy dinamikusan, szükség szerint más objektumokat hozzon létre és szüntessen meg. Az üzenetekkel történő kommunikáció jól ki van építve és jól működik. Fontos szerep jut a polimorfizmusnak és a polimorfikus objektumoknak.

d.) Hibrid nyelvek

Számos programozási nyelv azonban „nem kötelezi el magát" egyik kategória mellett sem, lehetőséget biztosítva arra, hogy mindhárom elvet, módszert kihasználva lehessen programokat írni, alkalmazásokat fejleszteni. Ezeket hibrid nyelveknek nevezzük.

e.) Eseményorientált

Az esemény egy olyan történés, amely megváltoztatja valamely objektum állapotát. Az események lehetnek automatikus vagy manuális események.

Automatikus események:

- *jel*: egy objektum egyértelmű jelt küld egy másik objektumnak valamilyen kommunikációs protokoll segítségével
- *hívás*: egy objektum meghívja egy másik objektum valamelyik metódusát, üzenetet küld
- *őrfeltétel*: egy előre kijelölt feltétel beteljesedik
- *kivétel*: valamilyen kivétel lép fel, az objektum működése eltér a normális működéstől
- *idő*: letelik a műveletre szánt kijelölt idő, vagy elérkezik egy megjelölt időpillanat
- *visszajelzés*: valamelyik periféria automatikusan visszajelez

Manuális események:

- a felhasználó billentyűzet segítségével egy karaktersort, parancsot visz be
- a felhasználó egér vagy fényceruza segítségével parancsod ad a rendszernek
- a felhasználó menüben vagy más parancsosztó rendszerben navigálva parancsot ad

Az eseményvezérelt programozás azt jelenti tehát, hogy a program futása során események keletkeznek, amelyeket valamilyen kontroll objektum kap meg, és vagy feldolgozza egy eseménykezelő segítségével ezeket az eseményeket, vagy megfelelő szabályok szerint szétosztja, szétszórja az eseményeket a program objektumai között. Ezt a mechanizmust *események szórásának* nevezzük. Az egyes objektumok fogadni tudják az eseményeket és reagálni tudnak az eseményekre. A reakció lehet egy feladat végrehajtása (*eseménykezelő metódus* segítségével), vagy lehet egy újabb esemény kiváltása. Ahhoz, hogy egy objektum fogadni tudjon egy eseményt, a következő feltételek kell, hogy teljesüljenek:

- az objektum fel kell legyen készítve az esemény fogadására
- az objektumhoz el kell jusson az esemény

Az *eseményorientált programozás* tehát olyan programozás, amely egy eseménybegyűjtő és szóró mechanizmuson alapszik, és az objektumok a hozzájuk beérkezett eseményeket lekezelik. Az eseményorientált programozás nagyon jól illeszkedik az objektumorientált paradigmához és teljes mértékben kihasználja az objektumok közötti kommunikációt és kapcsolatokat.

Kivételkezelés

Minden programozó rémálma talán az, hogy az általa írt alkalmazás, program minden különösebb ok nélkül, egyszerre csak kiír valamilyen furcsa hibaüzenetet és „lefagy". Valamilyen végzetes hiba lép fel az alkalmazásban, elérkezett egy olyan pontba, ahonnan nincs normális kiút, a futás megszakad és egyszerűen „kidobja" a felhasználót.

A hibák okainak sokfélesége miatt a hibavizsgálat gyakran több időt és energiát igényel, mint maga az alkalmazás fejlesztése. A programozó tulajdonképpen minden lehetséges futási módot, minden kombinációt végig kellene, hogy próbáljon ahhoz, hogy meggyőződjön a programkód hibamentességéről. Ez nem „egyszerű" megoldás. Az igazi megoldás az, ha a programozási nyelv biztosít valamilyen mechanizmust a hibák elhárítására, lekezelésére. Az objektumorientált hibakezelés a *kivételkezelésen* alapszik.

A kivétel egy esemény vagy feltétel, melynek bekövetkezése megszakítja a program normális futását.

Amikor valamilyen hiba lép fel egy metódus futása során, automatikusan létrejön egy kivételobjektum, amely információkat tartalmaz a kivétel típusáról és az alkalmazás pillanatnyi állapotáról. Az a metódus, amelyben a hiba fellépett, *kiváltja* a kivételt (*throws the exception*). A kivétel kiváltása után a metódus működése megszakad. A kiváltott kivételt kezelni kell, ezt a *kivételkezelő* kódblokk végzi el (az a blokk, amely által kezelt kivétel típusa megegyezik a kiváltott kivétel típusával). A kivételkezelő blokkok egymásba ágyazhatók, a nyelv mindig megkeresi a legalkalmasabb kivételkezelőt. Ezt a tevékenységet a kivétel *elkapásának* (*catching the exception*) nevezzük. A kivétel elkapása után a kivételkezelő kapja meg a vezérlést és ez értelmes módon feldolgozza a kivételt: vagy elhárítja a hibát, vagy visszaállítja a rendszert egy előző, stabil állapotba.

A kivételkezelés nagyon jól illeszkedik az objektumorientált paradigmához. A hibát kezelő kód jól elkülönül a tényleges kódtól, a hiba könnyen eljut arra a helyre, ahol ezt kezelni kell. A kivételosztályok hierarchiába szerveződnek.

E megoldás nagy előnye az, hogy a program szerkezete lényegesen egyszerűbbé válik, a futtatás normál esetei szétválaszthatók a hibás esetektől.

Kovács Lehel