Objektumorientált programozás

Az **objektumorientált programozás** (angolul *object-oriented programming*, röviden **OOP**) egy programozási módszer. Ellentétben a korábbi programozási nyelvekkel, nem a műveletek megalkotása áll a középpontban, hanem az egymással kapcsolatban álló programegységek hierarchiájának megtervezése. Az objektumorientált gondolkodásmód lényegében a valós világ lemodellezésén alapul – például egy hétköznapi fogalom, a *„kutya”* felfogható egy osztály (a kutyák osztálya) tagjaként, annak egyik objektumaként. Minden kutya objektum rendelkezik a kutyákra jellemző tulajdonságokkal (például szőrszín, méret stb.) és cselekvési képességekkel (például futás, ugatás). Az objektumorientált programozásban fontos szerep jut az úgynevezett *öröklődésnek*, ami az osztályok egymásból való származtatását teszi lehetővé: a kutyák osztálya származhat az állatok osztályából, így megörökli az állatok tulajdonságait és képességeit, valamint kibővítheti vagy felülírhatja azokat a kutyák tulajdonságaival, képességeivel.

Megközelítések

Analízisszintű gondolkodás

Egy szoftver fejlesztésének korai fázisaiban a megvalósítandó rendszer feladatait szeretnénk feltérképezni: a funkcionális és egyéb jellegű követelményeket. Más szóval, a kérdés ilyenkor az, hogy a rendszernek mit kellene tennie. Ilyenkor határozzuk meg a szoftver (formális és informális) specifikációját, majd abból kiindulva kezdjük magasabb szintű absztrakciók segítségével előállítani a rendszer modelljét, amely a konkrét megvalósítás alapját fogja képezni.

Tervezésszintű gondolkodás

A meglévő modell alapján a szoftver konkrét implementációjához (megvalósításához) vezető utat tervezzük meg. Ilyenkor arra keressük a választ, hogy a meghatározott specifikációt hogyan valósítsa meg a rendszer. Ezen a szinten már képbe kerülnek különböző alacsony szintű technikák is, mint például a kommunikációs protokollok, programozási nyelvek és technológiák.

Az OOP és a szekvenciális nyelvek különbségei

A hagyományos, ún. szekvenciális programozási nyelvekben az utasítások sorról sorra, egymást követően hajtódnak végre. Ez annyit jelent, hogy a program egyes sorai egymásra épülnek: Ha az egyik sorban definiálunk egy változót, akkor a következő sorban is használhatjuk. Ezt az egyszerűséget megtöri az előre definiált rutinok és függvények használata, de még a ciklusokkal, illetve más vezérlési szerkezetekkel együtt is könnyen átlátható a forráskód struktúrája. Az ilyen nyelveknél ez felfogható előnyként is, azonban nagy projektekben kényelmetlenséget okozhat, hogy egy-egy alapvető algoritmust többször, többféleképpen is le kell írnunk, holott lényegi változtatás nincs benne. Ezt a problémát részben lehet szubrutinokkal és függvényekkel orvosolni, de az igazi megoldást az objektumok használata jelenti.

Az objektumközpontú problémamegközelítés alapjaiban megváltoztatja a programozással kapcsolatos gondolkodásmódunkat. Az objektumorientált programozás alapja, hogy az adatokat és az őket kezelő függvényeket egy egységbe „zárjuk” (*encapsulation* – egységbezárás). Az OOP másik fontos sajátossága az öröklődés *(inheritance)*. Az öröklés azt jelenti, hogy egy osztályból kiindulva újakat hozunk létre, amelyek öröklik az „ősosztály” *(baseclass)* adattagjait és metódusait. Az új osztályt származtatott *(derived)* osztálynak nevezzük.

Mik is az objektumok valójában?

Ha egy programkódban objektumokról beszélünk, először az osztályok felépítését kell megvizsgálni, ugyanis minden objektum egy előre definiált osztály példánya. Egy osztályban általában egy feladat elvégzéséhez szükséges kódrészleteket gyűjtik össze funkciójuk szerint, tagfüggvényekbe rendezve. Egy osztályt létrehozása után példányosítani kell, hogy használhatóvá váljanak a benne összegyűjtött rutinok. Az osztály példányát nevezzük objektumnak.

```
class A { }; //egy üres osztály
```

Az osztályok felépítése

A használt programozási nyelvtől függően, de legtöbbször a *class* előtétszóval definiálunk osztályokat. Egy osztály csak függvényeket (*metódusokat*, *tagfüggvényeket*) és változókat *(adattagokat)* tartalmazhat. Mivel az osztályokban található függvényeket az objektumon keresztül lehet elérni, ezért hívják őket tagfüggvényeknek, vagy metódusoknak. Bizonyos nyelvek megengedik, az osztály függvényeinek használatát példányosítás nélkül is (például a C++-ban a *statikus* függvények). Ilyen esetekben azonban a függvény meghívásához meg kell adni annak az osztálynak a nevét, amelynek a metódus tagja.

Minden osztálynak van legalább egy *konstruktora*. Ez egy különleges függvény, amely az osztály példányosításakor hívódik meg. Feladata az osztály kezdőértékeinek beállítása, helyfoglalás a memóriában stb. Több konstruktort is megadhatunk, a fordító a szignatúrától függően választja ki a megfelelőt. A konstruktor párja a *destruktor*, ami az elfoglalt memóriát szabadítja fel, eltakarít maga után. Destruktorból pontosan egy darab van az osztályban. Néhány nyelvben (C#, Java) ún. „garbage collector”, „szemétgyűjtő” van, ami a program futása közben automatikusan szabadítja fel a nem használt memóriát.

Nyelvektől függően léteznek különböző osztályváltozók, melyek többnyire – a tagfüggvényekhez hasonlóan, csak az objektumon belül használhatóak. Például a C++ nyelvben ilyen a *this* változó (mutató), mely a példányosított osztály nevétől függetlenül mindig saját objektumára fog hivatkozni.

```
class A
{
 public:
  A() { this->msg = "Helló, világ!";} //konstruktor
 private:
  std::string msg;
};

A* obj = new A(); //példányosítás
```

Öröklődés

Egy osztály definiálása után előfordulhat, hogy az osztályban szereplő kódokat más osztályokban is használni szeretnénk. Például egy adatbázist, vagy mondjuk egy fájlkezelő függvényeket tartalmazó osztályt, valószínűleg más objektumokból is szeretnénk használni. Ehhez csak arra van szükség, hogy egy adott osztályt amiből a másik osztály tagfüggvényeit el akarjuk érni, abból származtassunk. Ezt nevezzük öröklődésnek.

```
class Base
{
 public:
  Base(){ };
  void f(){ };
```

```
};

class Derived : public Base
{
 public:
  Derived(){ };
};

Derived* der = new Derived();
der->f(); //A Derived osztály örökölte az f függvényt
```

Belátható, hogy az öröklődés használata rendkívül leegyszerűsíti a gyakran kellő függvények integrálását az egyes osztályokba anélkül, hogy meg kellene változtatni az osztályok struktúráját. Azokban a nyelvekben ahol esetleg nem valósították meg az öröklődést, van egy alternatív megoldás a problémára: Abban az osztályban, ahol használni szeretnénk egy másik osztály tagfüggvényét, egyszerűen példányosítanunk kell egy ősosztálybeli objektumot. Ennek a módszernek hátránya lehet, hogy egyes objektumokból esetleg több példány is lesz, ezáltal felesleges memóriát használ a programunk. Ezt kiküszöbölendő megtehetjük, hogy a használt osztályon kívül példányosítjuk a másik osztályt, és paraméterként adjuk át.

Nyilvánosság (PPP)

A PPP rövidítés az objektum orientált nyelvekben többnyire alkalmazott adattulajdonságok rövidítése. Angolul, sorrendben: *public*, *protected*, *private* – azaz: nyilvános, védett és saját. Ezen tulajdonságokat az objektum tagfüggvényei, illetve változói egyaránt felvehetik. Az egyes tulajdonságok az elemek nyilvánosságát, azaz hozzáférhetőségét határozzák meg a következő módon:

- `public`: Az objektumot használó bármely kód számára közvetlenül hozzáférhető.
- `protected`: Közvetlenül nem férhető hozzá, de a származtatott osztályok használhatják.
- `private`: Csak abban az osztályban érhetők el, amelyikben meghatározták őket.

Felületek

A felületek, vagy más néven interfészek biztonsági segítséget jelentenek a nagyobb projektekben. Képzeljük csak el, mi történne, ha *A* objektum példányosításakor paraméterként átadunk neki egy másik, *B* objektumot, de *B* objektum nem valósítja meg hozzá fűzött reményeket, azaz nem tartalmaz egy változót, vagy függvényt, amire *A* osztálynak szüksége van! Az ilyen hibák felkutatása az OOP összetettsége miatt nem egyszerű, de az ilyen hibák felületek használatával elkerülhetőek.

A felület valójában előre meghatározza egy osztály felületét: Megadja a benne található tagfüggvényeket, azok nyilvánossági tulajdonságait, bemenő paramétereit, egyszóval a függvénytörzs kivételével mindent.

```
interface IF
{
 void f();
}
```

Egy ilyen felület úgy véd meg a fent említett hibáktól, hogy az osztály definiálásakor megadjuk, milyen felületet kell megvalósítania. Ha a definiált osztály nem illeszkedik a felületre a fordító hibát jelez, de elkerüljük a sokkal kellemetlenebb futásidőben bekövetkező hibát. A fenti interfész az alábbi osztály „csontváza”:

```
interface ItestInterface
{
```

```
 void f();
}

class ImplementationClass : ItestInterface
{
 void IF.f(){;}
}

ImplementationClass ic = new ImplementationClass();
ItestInterface itf = (ItestInterface) ic;
itf.f();
```

Elvont osztályok

Az elvont, vagy absztrakt osztályok átmenetet képeznek a hagyományos osztályok és az osztályokat meghatározó felületek között. Ez azt jelenti, hogy egy absztrakt osztály egyaránt tartalmaz kidolgozott, törzzsel rendelkező tagfüggvényeket, és előre nem meghatározott tagfüggvényeket, melyeket majd az osztályból származtatott alosztály fog részletesen definiálni. Az absztrakt osztály ily módon hagyományos értelemben vett osztályként és a tőle öröklő alosztályok interfészeként egyszerre tölt be funkciót.

```
class AbstractBase
{
 public:
  void printMsg() = 0;
  virtual ~AbstractBase();
};

class Derived : public AbstractBase
{
 public:
  Derived(){ };
  ~Derived(){ };
  void printMsg(){ std::cout << "MSG\n"; }
};
```

Tervezési minták

A tervezési minták lényegében azokra a problémákra nyújtanak általános megoldást, melyekkel a szoftverfejlesztők gyakran találkoznak. Ilyen probléma például, amikor egy adott környezetben már bevált kódot alkalmassá kell tennünk egy másik interfészen keresztül történő használatra is. Bár a probléma minden feladatnál egyedinek tűnik, egy idő után felismerjük, hogy a megoldási módszerek lényegében azonos sémát követnek. Így tehát, ha felfedjük egy probléma lényegét és összevetjük hasonló, már megoldott problémákkal, ugyanazokkal a lényegi lépésekkel találkozunk.

Lásd még

- Objektumorientált programozási nyelvek listája
- UML

Irodalom

- Angster Erzsébet: **Az objektumorientált tervezés és programozás alapjai** (*magánkiadás* 1999; ISBN 9789636508180)
- Raffai Mária: **Objektumtechnológia sorozat** kötetei: (1) *Objektumok az üzleti modellezésben* (ISBN 963-9056-28-6; 2001) (2) *Egységesített megoldások a fejlesztésben* (ISBN 963-9056-29-4; 2001) (3) *Objektumorientált alkalmazásfejlesztés* (ISBN 963-9056-31-6; 2002) (4) *UML 2' modellező nyelvi kézikönyv' (ISBN 963-7692-01-0; 2005, 2007)*
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: **Programtervezési minták** (*Kiskapu* 2004; ISBN 9639301779)
- Raffai Mária: **Információrendszer-fejlesztés** (*Novadat* 1999; ISBN 9639056197)

Külső hivatkozások

- Weblabor.hu – Az objektumorientált programozás előnyei a kódújrahasznosítás jegyében [1]
- Codex Magazin – Programozzunk objektumokkal [2]
- Prog.hu – Öröklés és konstruktorok C++-ban [3]
- Lásd még: Prog.hu – Aspektus-orientált programozás [4]

Hivatkozások

[1] http://weblabor.hu/cikkek/oopkodujrahasznositas
[2] http://www.codexonline.hu/CodeX4/alap/month/cikkoo.htm
[3] http://prog.hu/cikkek/539/Orokles+es+konstruktorok+C+-ban.html
[4] http://www.prog.hu/cikkek/905/Aspektus-orientalt+programozas.html

Szócikk forrásai és közreműködői

Objektumorientált programozás *Forrás*: http://hu.wikipedia.org/w/index.php?oldid=7656317 *Közreműködők:*: Civertan, Csörföly D, December, GaborLajos, Glanthor Reviol, Hkoala, JagdTiger, Juhasz peter, Messiah, Mr Steve, Nagytibi, Nyenyec, Opa, Pifta, Qorilla, Simon69, Szajci, 2 névtelen szerkesztések

Licenc