ELTE IK - Programtervező Informatikus BSc

Záróvizsga tételek

11. Objektumelvű programozási nyelvek

Objektumelvű programozási nyelvek

Osztály és objektum. Egységbe zárás, tagok, konstruktorok. Információ elrejtése. Túlterhelés. Memóriakezelés, szemétgyűjtés. Öröklődés, többszörös öröklődés. Altípusosság. Statikus és dinamikus típus, típusellenőrzés. Felüldefiniálás, dinamikus kötés. Generikusok. Altípusos és parametrikus polimorfizmus. Objektumok összehasonlítása és másolása.

1 Osztály és objektum

Az osztály egy felhasználói típus, amelynek alapján példányok (objektumok) hozhatók létre. Az osztály alapvetően attribútum és metódus (művelet) definíciókat tartalmaz. Az osztály írja le az objektum típusát: megadja a tulajdonságait és azok lehetséges értékeit (azaz a típusértékeket), valamint az objektumon végrehajtható műveleteket (típusműveletek).

Példa C++-ban:

```
//definition of Point
class Point {
private:
    int xCoord;
    int yCoord;
public:
    //constructor
    Point(int xCoord, int yCoord) : xCoord(xCoord),yCoord(yCoord) {}

    void Translate(int dx, int dy) {
        xCoord+=dx;
        yCoord+=dy;
    }

    void Translate(Point delta) {
        xCoord+=delta.xCoord;
        yCoord+=delta.yCoord;
    }

    int getX() { return xCoord; }

    int getY() { return yCoord; }
```

```
};

int main(int argc, char* argv[])
{
    Point point(0,0);
    point.Translate(5,-2);
    cout<<point.getX()<<","<<point.getY()<<endl; //5,-2
    Point delta(-2,1);
    point.Translate(delta);
    cout<<point.getX()<<","<<point.getY()<<endl; //3,-1

    return 0;
}
```

Megjegyzés: A nem objektum-orientált nyelvekben nincsenek osztályok, pl. régebbi nyelvekben, mint a C, ADA, Pascal, vagy funkcionális nyelvekben (Haskell, Clean, stb.).

2 Egységbe zárás

Az egységbe zárás azt fejezi ki, hogy az összetartozó adatok és függvények, eljárások együtt vannak, egy egységbe tartoznak. További fontos fogalom az adatelrejtés, ami azt jelenti, hogy kívülről csak az férhető hozzá közvetlenül, amit az objektum osztálya megenged. Ez fontos ahhoz, hogy megelőzze a nem kívánt kapcsolatok kialakulását, megkönnyítse a kód értelmezését, és elkerülje az adatok ellenőrizetlen terjedését (lásd objektumtobzódás).

Ha az objektum, illetve osztály elrejti az összes adattagját, és csak bizonyos metódusokon keresztül férhetnek hozzá a kliensek, akkor az egységbe zárás az absztrakciót és információelrejtés erős formáját valósítja meg. Egyes nyelvek, mint a Java vagy a C++, C# ezt ki is kényszerítik (public: nyilvános, private: csak az adott osztályú objektumok számára, protected: csak az adott osztály, vagy leszármazott osztályok példányai számára), míg mások, mint a Python nem, itt csak konvenciókkal valósítható meg hasonló (kérlek ne piszkáld közvetlenül azt, aminek aláhúzással kezdődik a neve). A Java és a C# ismeri a csomagnyilvánosságot is, ez Javában alapértelmezett. Ezeket a jellemzőket adattagokhoz és metódusokhoz is hozzá lehet rendelni.

Az adatelrejtés támogatja a refaktorálást, azaz az osztály belső reprezentációja szabadabban átírható, a klienseket ez nem érinti, egészen addig, amíg a meglévő publikus metódusokat ugyanazzal a paraméterezéssel hívhatják. Továbbá bátorítja a programozókat, hogy egy helyre tegyék az összetartozó adatokat és az őket feldolgozó függvényeket, eljárásokat, amely szerveződést a programozó társai is megérthetnek. A kapcsolatok lazítását is megkönnyíti.

3 Konstruktor, destruktor

3.1 Konstruktor

A konstruktor az objektumok inicializáló eljárása, akkor fut le, ha egy osztályból új objektumot példányosítunk. Alapértelmezett konstruktor alatt a paraméter nélküli konstruktort értjük, a legtöbb programozási nyelv esetén ezt a fordítóprogram automatikusan generálja üres törzzsel, amennyiben nem lett megadva egy konstruktor sem egy osztályban. Többek között a konstruktorban szokás gondoskodni arról, hogy az objektum dinamikus élettartamú változói számára tárhelyet foglaljunk.

3.2 Destruktor

A destruktor a konstruktor ellentétes párja, ez az eljárás az objektumok felszabadításakor fut le. Meghívása automatikusan megtörténik, attól függetlenül, hogy az objektum felszabadítása automatikusan történik a hatókör

végeztével (lokális objektumok esetén), manuálisan a programozó által (dinamikus élettartamú objektumok esetén) vagy a szemétgyűjtő által (szintén dinamikus élettartamú objektumok esetén).

A desktruktorban szokás többek között az objektum dinamikus helyfoglalású adattagjait és a lefoglalt erőforrásokat felszabadítani.

4 Információ elrejtése

Az objektum elrejti az adatait és bizonyos műveleteit. Ez azt jelenti, hogy nemtudjuk pontosan, hogy egy objektumban hogyan vannak az adatok ábrázolva, sőt aműveletek implementációit sem ismerjük. Az információk elrejtése az objektumbiztonságát szolgálja, amelyeket csak a ellenőrzött műveleteken keresztül érhetünkel.

5 Túlterhelés és felüldefiniálás

5.1 Túlterhelés

Több azonos nevű, különböző szignatúrájú függvény. A függvényhívás aktuálisparaméterei meghatározzák, hogy melyik függvény fog meghívódni. Ezt már afordításidőben eldől (statikus, fordításidejű kötés).

A túlterhelés (overloading) segítségével azonos nevű alprogramokat hozhatunk létre eltérő szignatúrával. A szignatúra a legtöbb programozási nyelvben az alprogram nevét és a formális paraméterek számát és típusát jelenti, de egyes nyelvekben (például Ada) a visszatérési érték típusa is beletartozik. A túlterhelés elsődleges felhasználási területe, hogy ugyanazt a tevékenységet különböző paraméterezéssel is elvégezhessük.

A fordító az alprogramhívásból el tudja dönteni, hogy a túlterhelt változatok közül melyiket kell meghívni. Ha egyik sem illeszkedik vagy több is illeszkedik, akkor fordítási hiba lép fel.

5.2 Felüldefiniálás

Egy osztályhierarchián belül az utódosztály újradefiniálja az ősosztály metódusát.(azonos név, azonos szignatúra). Ha ősosztály típusú mutatón vagy referenciánkeresztül érjük el az osztályhierarchia példányait és ezen keresztül meghívjuk afelülírt metódust, akkor futási időben dől el, hogy pontosan melyik metódus kerülmeghívásra. (dinamikus, futásidejű kötés).

6 Objektumok másolása, összehasonlítása

Az objektumok másolása egy speciális konstruktorral, az úgynevezett másoló konstruktorral (copy constructor) történik. Ez paraméterül az adott osztály egy példányát kapja meg, és azt a programozó által megadott működési logika szerint lemásolja az éppen inicializált objektumba.

Több programozási nyelv (például C++) fordítóprogramja automatikusan elkészít egy másoló konstruktort, ha a programozó nem definiál sajátot. Ez az alapértelmezett másoló konstruktor lemásolja a forrásobjektum összes adattagjának értékét, ezt nevezzük sekély másolatnak (shallow copy). Ha az objektum dinamikus foglalású adattagokat is tartalmaz, akkor azoknak nem az értéke, hanem csak a hivatkozása lesz lemásolva, ami általában nem a kívánt működés. Ez esetben saját másoló konstruktor írása szükséges, ami mély másolatot (deep copy) készít.

7 Automatikus, statikus és dinamikus élettartam, szemétgyűjtés

Élettartam: A változók élettartama alatt a program végrehajtási idejének azt a szakaszát értjük, amíg a változó számára lefoglalt tárhely a változóé.

7.1 Automatikus élettartam

A blokkokban deklarált lokális változók automatikus élettartamúak, ami azt jelenti, hogy a deklarációtól a tartalmazó blokk végéig tart, azaz egybeesik a hatókörrel. A helyfoglalás számukra a végrehajtási verem aktuális aktivációs rekordjában történik meg.

7.2 Statikus élettartam

A globális változók, illetve egyes nyelvekben a statikusként deklarált változók (például C/C++ esetén a `static` kulcsszóval) statikus élettartamúak. Az ilyen változók élettartama a program teljes végrehajtási idejére kiterjed, számukra a helyfoglalás már a fordítási időben megtörténhet.

7.3 Dinamikus élettartam

A dinamikus élettartamú változók esetén a programozó foglal helyet számukra a dinamikus tárterületen (heap), és a programozó feladata gondoskodni arról is, hogy ezt a tárterületet később felszabadítsa. Amennyiben utóbbiról megfeledkezik, azt nevezzük memóriaszivárgásnak (memory leak). Mint látjuk, a dinamikus élettartam esetén a hatókör semmilyen módon nem kapcsolódik össze az élettartammal, az élettartam szűkebb vagy tágabb is lehet a hatókörnél.

7.4 Szemétgyűjtő

A szemétgyűjtő másik neve a hulladékgyűjtő, az angol Garbage Collector név után pedig gyakran csak GC-nek rövidítik. Feladata a dinamikus memóriakezeléshez kapcsolódó tárhelyfelszabadítás automatizálása, és a felelősség levétele a programozó válláról, így csökkentve a hibalehetőséget.

A szemétgyűjtő figyeli, hogy mely változók kerültek ki a hatókörükből, és azokat felszabadíthatóvá nyilvánítja. A módszer hátránya a számításigényessége, illetve a nemdeterminisztikussága. A szemétgyűjtő ugyanis nem szabadítja fel egyből a hatókörükből kikerült változókat, és a felszabadítás sorrendje sem ugyanaz, amilyen sorrendben a változók felszabadíthatóvá váltak.

Azt, hogy a hulladékgyűjtő mikor és mely változót szabadítja fel, egy programozási nyelvenként egyedi, összetett algoritmus határozza meg, amelyben rendszerint szerepet játszik a rendelkezésre álló memória telítettsége, illetve a felszabadításhoz szükséges becsült idő. (Például ha egy objektum rendelkezik destruktorral, akkor általában a GC később szabadítja csak fel.)

Összességében a szemétgyűjtő csak annyit garantál, hogy előbb-utóbb (legkésőbb a program futásának végeztével) minden dinamikusan allokált változót felszabadít.

Szemétgyűjtést használó nyelvek pl. Java, C#, Ada. C/C++-ban nincs szemétgyűjtés, a programozónak kell gondoskodni a dinamikusan allokált memóriaterületek felszabadításáról.

8 Öröklődés

Egy osztály legegyszerűbben adattagjainak és metódusainak felsorolásával hozható létre. Azonban az objektumorientált paradigma lehetőséget ad egy másik, hatékonyabb módszerre is, az öröklődésre. Az öröklődés az újrafelhasználhatóságo szem előtt tartva arra ad lehetőséget, hogy már meglévő (szülő-, ős-) osztályból kiindulva hozzunk létre új (gyermek-, leszármazott-, al-) osztályt. Az öröklés két osztály között fennálló olyan kapcsolat, amely során a leszármazott osztály rendelkezik a szülő osztály majdnem összes tulajdonságával (nem privát adattagjait és metódusait sajátjaként kezeli), s ezeket újabbakkal egészítheti ki. Az így létrehozott osztály is lehet más osztályok őse (kivéve pl. Java-ban a `final` kulcsszóval ellátott osztályok, ezekből már nem származtathatunk), így ezek az osztályok egy öröklési hierarchiába szerveződnek. Attól függően, hogy egy osztálynak egy- vagy több őse van, beszélünk egyszeres- ill. többszörös öröklődésről. A Java az egyszeres öröklődést támogatja, de pl. C++-ban van többszörös öröklődés.

A Java-ban az osztályhierarchia legfelső eleme az Object osztály, amelyből minden más osztály (közvetve vagy közvetlenül) származik.

Egy alosztály az örökölt metódusokat újraimplementálhatja. Ilyenkor az adott metódus ugyanolyan néven, de más, módosított (alosztályra specifikált) tartalommal kerül megvalósításra. Az ilyen metódusokat polimorfnak nevezzük. Java-ban minden olyan metódust, ami nincs ellátva a `final` kulcsszóval, újra lehet definiálni. (C++-ban mindent, ami nem privát)

Példák:

- C++:

```
class RegularPolygon
{
protected:
    const double radius;
public:
    RegularPolygon(double radius) : radius(radius) {}
    virtual double area() = 0;
};

class EquilateralTriangle : public RegularPolygon
{
public:
    EquilateralTriangle(double radius) : RegularPolygon(radius) {}
    virtual double area()
    {
         return 0.75*sqrt(3.0)*radius*radius;
    }
};
```

- Java:

```
public abstract class RegularPolygon{
    protected final double radius;

    public RegularPolygon(double radius){
        this.radius = radius;
    }
    public abstract double area();
}

public class EquilateralTriangle extends RegularPolygon{
    public EquilateralTriangle(double radius){
        super(radius);
    }
    public double area(){
         return 0.75*Math.sqrt(3.0)*radius*radius;
    }
}
```

9 Statikus és dinamikus típus és kötés

Nem a kliens, hanem az objektum feladata megválasztani, hogyan reagáljon egy metódushívásra. Ezt tipikusan futás időben végzi el, és a metódushívást a hozzá társított táblából választja ki. Ez dinamikus kötés néven ismert, és megkülönbözteti az objektumot az absztrakt adattípustól és a modultól, amelyek rögzített megvalósítással bírnak minden példány számára. Ha a metódus kiválasztásába beleszól a többi paraméter, akkor többszörös kötésről van szó (lásd kettős metódus, multimetódus, többszörös metódus).

A metódushívást tekintik üzenetátadásnak is, ahol a kliens a kötésben részt vevő objektumnak küld üzenetet.

- statikus típus: A változó deklarációjában megadott típus. Fordítás során egyértelműen eldől, nem változhat futás során. A statikus típus határozza meg, hogy mit szabad csinálni az objektummal (pl. hogy milyen műveletek hívhatók meg rá).
- dinamikus típus: A változó által hivatkozott objektum típusa. Vagy a statikus típus leszármazottja, vagy maga a statikus típus. Futás során változhat.

Példa(Java):

```
Object o = new String("Hello");
```

Itt az o változó statikus típusa `Object`, dinamikus típusa pedig `String`.

Kötések:

- statikus kötés: A változó statikus típusa szerinti adattagokra lehet hivatkozni.
- dinamikus kötés: A változó dinamikus típusa szerinti adattagok használhatók.

A kötés akkor fontos, ha a hívott művelet, vagy a hivatkozott változó a statikus és a dinamikus típusban különbözik, vagy esetleg a statikus típusban nem is létezik (ekkor a dinamikus típusban sem hivatkozhatunk az adattagra).

Többféleképpen lehet a kötéseket meghatározni a különböző nyelvekben:

- Java : Minden esetben dinamikus kötés van (az örökölt metódusok törzsében is).
- C++ : A művelet definiálásakor lehet jelezni, ha dinamikus kötést szeretnénk (`virtual`).
- Ada : A híváskor lehet jelezni, ha dinamikus kötést szeretnénk.

10 Generikusok

Generikus programozás: Algoritmusok, adatszerkezetek általánosított, több típusra is működő leprogramozása (pl. generikus rendezés, generikus verem, ...). Ezt az általános kódot nevezzük sablonnak (template).

Bizonyos nyelvekben (Ada, C++) egy sablont példányosítani kell – ekkor a kívánt típusokat, objektumokat (a sablon definíciónak megfelelően) a sablonnak paraméterként megadva példányosul a szóban forgó sablon. (Ugyanúgy, mint pl. amikor egy függvénynek megadjuk az aktuális paraméterét.) Ezt nevezzük generatív programozásnak: a program a megadott sablon alapján létrehoz egy "igazi" programegységet (program generál programot).

Példa: Egy verem sablon példányosításakor megadjuk, hogy milyen típusú elemeket tároljon a verem (típus paraméter) és hogy hány elem fér a verembe (objektum paraméter). C++ és Ada esetén egy sablon paramétere alprogram is lehet.

Példa: Egy rendezésnek megadjuk, hogy milyen művelet alapján rendezzen. Természetesen lehet alapértelmezett sablonparaméter is.

Egy sablon definiálása esetén természetesen a sablont nem lehet minden típusra használni. Egy sablon attól lesz sablon, hogy több típusra is működik. Azonban megszorításokat tehetünk (és tennünk is kell) arra, hogy milyen típusokra lehessen azt használni, hogyan lehessen a sablont példányosítani.

Jó példa erre az Ada nyelv, ahol a sablon specifikációja egy "szerződés" a sablon törzse és a példányosítás között:

- A sablon törzse nem használhat mást, csak amit a sablon specifikációja megenged neki. (A törzset nem feltétlenül kell, hogy ismerjük példányosításkor.)

```
generic
    type Element_T is private;
    with function "*" (X, Y: Element_T) return Element_T is <>;
function Square (X : Element_T) return Element_T;
```

  Itt a `with function` kezdetű sor végén az `is <>` azt jelenti, hogy ha az adott típusra már létezik * művelet (pl. egész számokra), akkor nem kell külön megadni példányosításkor, a program automatikusan azt használja.

  A törzs:

```
function Square (X: Element_T) return Element_T is
begin
    return X * X;   -- The formal operator "*".
end Square;
```

- A példányosításnak biztosítania kell mindent, amit a sablon specifikációja megkövetel tőle. A következő példában a négyzetre emelő függvényt mátrixokra alkalmazzuk, feltéve, hogy definiáltuk a mátrixszorzást.

```
with Square;
with Matrices;
procedure Matrix_Example is
    function Square_Matrix is new Square
        (Element_T => Matrices.Matrix_T, "*" => Matrices.Product);
    A : Matrices.Matrix_T := Matrices.Identity;
begin
    A := Square_Matrix (A);
end Matrix_Example;
```

Például a C++-ban a sablonszerződés nem így működik. Ott a sablon specifikációja az egész definíció (emiatt sablonosztályokat csak teljes egészében header fájlokban definiálhatunk). Példányosításkor ezt is ismerni kell, hogy tudjuk, hogyan példányosíthatunk. Az információelrejtés elve tehát sérül.

Más nyelvekben (pl. Java, funkcionális nyelvek) nem kell a sablonokat példányosítani – mindig ugyanaz a megírt kód hajtódik végre, csak épp az aktuális paraméterekkel. Pl. Java-ban a típusparaméter fordításkor "elveszik", csak futási időben derül ki.

11 Polimorfizmus

A programozási nyelvekben és a típuselméletben a polimorfizmus egy egységes interfészre utal, amit különböző típusok valósítanak meg. Többalakúságot is jelent. A polimorf típuson végzett műveletek több különböző típus értékeire alkalmazhatók. Polimorfizmus többféleképpen is megvalósítható:

- **Ad-hoc polimorfizmus:** Egy függvénynek sok különböző implementációja van, amelyeket egyenként specifikálnak néhány különböző típus és kombinációja számára. Megvalósítható túlterheléssel.
- **Paraméteres polimorfizmus:** A kódot általánosan írják meg különböző típusok számára, és alkalmazható az összes típusra, amely megfelel bizonyos, a kódban előre megadott feltételeknek. Objektumorientált környezetben sablonnak vagy generikusnak nevezik. Funkcionális programozási környezetben egyszerűen polimorfizmusnak hívják.
- **Altípusosság:** A név több különböző osztály példányait jelöli, amelyeknek a függvényt deklaráló közös őse van.[3] Objektumorientált környezetben többnyire erre gondolnak, amikor polimorfizmusról beszélnek.

11.1 Paraméteres polimorfizmus

A parametrikus polimorfizmus jellemzője, hogy a függvényt egyszer kell megírni, és az minden típusra egységesen fog működni. A függvény paramétereinek típusát a függvény használatakor paraméterben kell megadni, innen a paraméteres név. Szokták generikusnak is hívni, az egységes viselkedés miatt. Használata a teljes típusbiztonság megőrzésével növeli a nyelv kifejezőerejét.

A paraméteres polimorfizmus nemcsak függvényekre, hanem adattípusokra is értelmezhető, tehát adattípusokra is lehetnek paraméteresek, más néven generikusok. A generikus típusok polimorfikus adattípusok néven is megtalálhatók.

A parametrikus polimorfizmus nagyon gyakori a funkcionális programozásban, emiatt gyakran csak polimorfizmusnak nevezik. Az alábbi Haskell példa egy paraméteres polimorf lista adattípust mutat be, két paraméteresen polimorf függvénnyel.

```
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}

List<B> map(Func<A, B> f, List<A> xs) {
    ...
}
```

Objektumorientált nyelvekben kevésbé gyakori, de legtöbbjükben szintén van lehetőség paraméteres polimorfizmusra. Ezt C++-ban és D-ben template-ek, Javában genericek biztosítják.

11.2 Altípusosság

Egyes nyelvekben az altípusosság használható a polimorfizmus korlátozására. Ezekben a nyelvekben az altípusosság lehetővé teszi, hogy egy függvény egy bizonyos T típusú objektumot használjon, de ha S altípusa T-nek, akkor a függvény S típusú objektumokra is használható a Liskov-féle helyettesíti elv szerint. Az altípus jelölése néha S ¡: T. Megfordítva, T szupertípusa S-nek, ennek jele T :¿ S. Az altípusos polimorfizmus rendszerint dinamikus.

A következő példában a kutya és a macska az állatok altípusa. A letsHear() eljárás állatot fogad, de altípusos objektumokkal is működik.

```
abstract class Animal {
    abstract String talk();
}
```

```
class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}

void letsHear(final Animal a) {
    println(a.talk());
}

int main() {
    letsHear(new Cat());
    letsHear(new Dog());
}
```

A következő példában Number, Rational, és Integer típusok úgy, hogy Number :¿ Rational és Number :¿ Integer. Egy függvény, aminek paramétere Number, úgy működik Integer vagy Rational paraméterrel, mint Number paraméterrel. Az aktuális típus elrejthető a felhasználó elől, és objektumazonossággal férhető hozzá. Valójában, ha a Number típus absztrakt, akkor nem is lehetnek olyan objektumok, amelyek aktuális típusa Number. Lásd: absztrakt osztály, absztrakt adattípus. Ez a típushierarchia számtoronyként is ismert a Scheme programozási nyelvből, és rendszerint még több típusból áll.

Az objektumorientált nyelvek az altípusos polimorfizmust öröklődéssel valósítják meg. Tipikus implementációkban az osztályok tartalmaznak virtuális táblát, ami hivatkozza az osztály interfészének polimorf részét megvalósító függvényeit. Minden objektum tartalmaz egy referenciát erre a táblára, amire mindig szükség van, ha valami egy polimorf függvényt hív. Ez a mechanizmus példa a:

- Késői kötésre, mert a virtuális függvényhívások csak a híváskor kapcsolódnak;
- Egyszeri küldésre, mivel a virtuális függvényhívások csak az első argumentumuk virtuális tábláját veszik figyelembe, így a többi argumentum dinamikus típusa nincs figyelembe véve.

Néhány objektumrendszer, például a Common Lisp Object System többszörös küldésre is képes, így a hívások az összes argumentumokban polimorfak lesznek.