Objektumorientált paradigma

Az öröklődés

(*kulcsszavak:* öröklődés, öröklődési hierarchia, egyszeres, többszörös, előnyök, interface, kiküszöbölés, is_a reláció, protected, öröklési módok)

Ha már definiáltunk egy osztályt, bármikor lehetőségünk van arra, hogy az adott osztályt más osztályok definiálására felhasználjuk, azzal a céllal, hogy a már meglévő kódot újra fel tudjuk használni, illetve azzal a céllal, hogy működésében kibővítsük, testre szabjuk a már meglévő osztályt. Ez a mechanizmus úgy valósul meg, hogy a második osztályt *leszármaztatjuk* az első osztályból. Ezt *öröklődésnek* nevezzük, és az osztályok ilyenképpen *öröklődési hierarchiába* szervezhetők. Ilyen értelemben beszélhetünk *ősosztályokról* és *leszármazottakról*, *gyerek osztályokról*. Természetesen egy leszármazott a maga során lehet ősosztálya egy másik osztálynak vagy más osztályoknak.

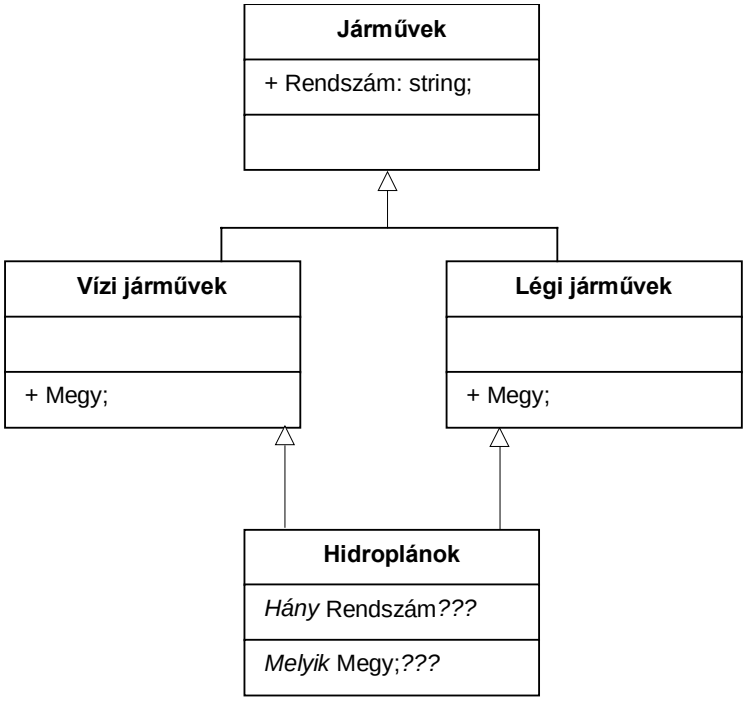
Az öröklődés tulajdonképpen két síkon nyilvánul meg: a leszármazott kiterjeszti az ős interfészét a behozott új attribútumokkal, metódusokkal (az osztály, a típus szintjén), de ugyanakkor leszűkíti az objektumok fogalmi szintjét (példányosítás). Például, ha az **Emberek** osztályt az **Élőlények** osztályból származtatjuk, akkor természetesen kibővítjük az **Élőlények** osztályt új attribútumokkal, metódusokkal: *intelligencia*, *kultúra*, *beszéd* stb., de az is nyilvánvaló, hogy sokkal kevesebb **ember** van, mit **élőlény**. Az **Emberek**, mint halmaz, részhalmaza az **Élőlények**nek, mit halmaznak.

Az öröklődés lehet *egyszeres* vagy *többszörös*. Egyszeres öröklődésről akkor beszélünk, ha a leszármazott osztálynak **pontosan egy** ősosztálya van. Ha kettő vagy ennél több ősosztálya van a leszármazottnak, akkor többszörös öröklődésről beszélhetünk. Más kifejezésekkel élve az egyszeres öröklődést *egyágúnak*, a többszörös öröklést *többágú öröklődésnek* is nevezzük.

1. ábra

A. Egyszeres *B. Többszörös öröklődés. Öröklődési hierarchiák*

A leszármazott osztály *örökli* az ősosztály adatait és metódusait. Ilyen szempontból a többszörös öröklődés nem egyértelmű, mivel ha, például, két ősosztályban szerepel egy-egy ugyanolyan nevű adat vagy metódus, akkor kérdéses, hogy a leszármazott vajon melyik osztálytól örökli át, mert mindkettőtől lehetetlen. A másik anomália az úgynevezett *rombusz-öröklődés*. Ha egy ősosztályban létezik legalább egy adat vagy egy metódus, az osztály *minden leszármazottja* örökli ezeket. Tegyük fel, hogy az illető ősnek van két leszármazottja, és létezik még egy harmadik leszármazott, amely többszörös öröklődést használva a két leszármazottból öröklődik, *hány példányban* jelenik meg az örökölt adat vagy metódus?



2. ábra
A többszörös öröklődés anomáliái

A két kérdés lényegében ugyanazt a problémát veti fel: *ha kétértelműség van, hogyan válasszunk?* Elméletileg erre három megoldás létezik.

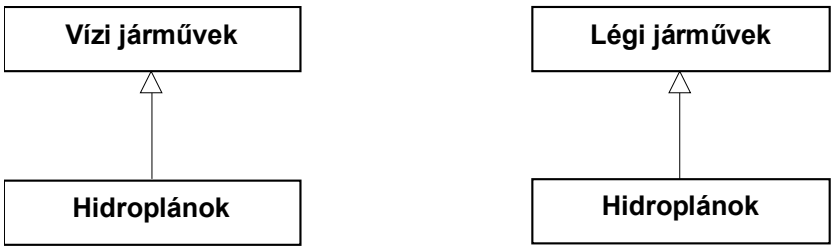
- A legtöbb esetben az ilyen kódot nem lehet lefordítani, a fordító, vagy a futtató környezet kétértelműségre (*ambiguous*) hivatkozva hibajelzéssel leáll.
- A származtatott osztály mondja meg, hogy melyiket szeretné használni.
- Az ősosztály mondja meg, hogy mit szeretne tenni ilyen esetben.

A fent említett kétértelműségre hivatkozva számos programozási nyelv nem is engedi meg a többszörös öröklődést, és a gyakorlott programozók is azt tanácsolják, hogy *kerüljük a többszörös öröklődést!* A későbbiekben számos módszert fogunk látni a többszörös öröklődés kiküszöbölésére.

1.1. Az öröklésről általában. Miért jó az öröklődés?

Ha öröklődésről beszélünk, definiálnunk kell a *helyettesíthetőség fogalmát is.* A helyettesíthetőség azt jelenti, hogy a leszármaztatott osztály objektumai bármilyen körülmények között helyettesíteni tudják az ősosztály objektumait, vagyis a leszármazott osztály felveheti az ősosztály szerepét, viselkedését, és nem lehet megkülönböztetni az ősosztály valamelyik példányától, ha hasonló környezetben használjuk. Ez a folyamat természetes, mivel a leszármaztatott osztályban szerepel az ősosztály minden adata és metódusa, így bármikor úgy viselkedhet, mint maga az ősosztály. Vagy azt is mondhatjuk, hogy az ősosztály szerepelhet formális paraméterként bárhol, ahol a leszármazott aktuális paraméterként előfordulhat.

A helyettesíthetőség fogalmát még *is_a relációnak* is szoktuk nevezni. Ez kifejezi azt, hogy az őstől a leszármazott felé *specifikálás*, a leszármazottól az ős felé pedig *általánosítás* történik. A gyakorlatban, azonban gyakran azért is használjuk az öröklődést, hogy leszűkítsük, testreszabjuk az ős működését. Vagy azért is, mert a már meglévő osztályon a konstrukció, idősporlás szempontjából csak keveset kell módosítanunk és máris egy új leszármazottat nyertünk. Ilyen esetekben nem áll fenn az *is_a reláció*, nem áll fenn a helyettesíthetőség. Fogalmi szinten is elkülönítjük ezeket az öröklődési típusokat. Ha fennáll az *is_a reláció*, akkor a leszármazott *altípusa* (*sub-type*) az ősnek, ha nem áll fenn, akkor *alosztálya* (*sub-class*) az ősnek.



3. ábra

A. Nem áll fenn az is_a reláció (logikailag nem helyes). B. Fennáll a reláció (logikailag helyes)

A fenti példában a **Hidroplánok** osztályt egyszer a **Vízi járművek** osztályából, egyszer pedig a **Légi járművek** osztályából származtatjuk. Nyilvánvaló, hogy az első esetben nem áll fenn az *is_a reláció*, mivel a **hidroplán** nem a kifejezés általános értelmében vett **Vízi jármű**, hanem egy olyan repülőgép, amely le tud szállni a vízre is, de nem rendelkezik más semmilyen, vízi járművekre vonatkozó jellegzetességgel, pl. *vasmacskával*, *mentőcsónakkal* stb. A második esetben fennáll a reláció, hisz a **hidroplán** egy speciális **Légi jármű**, olyan, amely le tud szállni a vízre is, és minden helyzetben helyettesíteni tudja a **Légi járműveket**. Elvileg mondhatjuk azt is, hogy a hidroplán egy olyan hajó, ami repülni tud, és azt is, hogy olyan repülőgép, ami le tud szállni a vízre. Nyilvánvaló, hogy a második állítás a helyes logikailag, habár az is nyilvánvaló, hogy konstrukció szempontjából az elsőt is meg lehet valósítani.

A gyakorlatban mégis mindkettő használható, attól függően, hogy melyik előnyösebb, melyik biztosít gyorsabb kódmódosítást és újrahasználást. *De vigyázzunk, mert ha nem áll fenn az is_a reláció, akkor bajok történhetnek (akár fogalmi, akár fizikai szinten - mint a példából is láthatjuk) a helyettesítésekkor.*

Összesítve, öröklődést a következő estekben használhatunk:

a.) Specializálás

Specializáljuk az ősosztályt. Nem változtatjuk meg a meglévő metódusokat, adatokat, de behozhatunk újakat. Ebben az esetben fennáll az *is_a releció*. Az öröklődés leggyakrabban használt, ideális esete, amely jó programstruktúrát eredményez. Például a **Halászhajó** a **Hajó**nak egy speciális altípusa, egy olyan hajó, amely rendelkezik a **Hajó**k összes tulajdonságával, de pluszban még **halászni** is tud. Vagy pl. a **TextEditWindow** (olyan ablak, amelyben szöveget tudunk szerkeszteni) a **Window** (általános ablak) speciális esete.

b.) Specifikálás

Ez abban az esetben áll fenn, amikor az ős egy általános osztály, a leszármazottak pedig konkrét implementációk. Ezt az esetet használjuk fel a *homogén interfészek* létrehozására is. Minden leszármazott ugyanúgy viselkedik, ugyanolyan nevű metódusokat tartalmaz. Nem hoz be Ebben az esetben is fennáll az *is_a reléció*. Például a **Vonat**nak, mit általános ősosztálynak specifikált leszármazottjai a **Személyvonatok**, **Gyorsvonatok**, **InterCity**-k. Semmilyen új metódust nem hoznak be, csak a menetidő változik, és persze a jegy ára.

c.) Konstrukció

Az ős biztosítja a gyerek felépítését, de logikailag más kontextust nem biztosít. Ez a módszer logikailag nem a leghelyesebb és az *is_a reláció* sem áll fenn. Például **Hidroplán** és **Vízi járművek**, vagy ha a **Halmaz** osztályt a **Lista** osztályból származtatjuk (a halmaz egy olyan lista, amiben minden elem csak egyszer fordul elő – konstrukció szempontjából jó, logikailag helytelen). Hasonlóan gyakran előfordul például, hogy a grafikus objektumokat a **Pont** osztályból származtatjuk: a **Kor** az **x**, **y** középpontot tartalmazó **Pont**ot kiterjeszti úgy, hogy behoz egy **r** sugarat (konstrukció szempontjából kényelmes megoldás, de matematikailag helytelen, mert a **Kör** nem **Pont**!).

d.) Általánosítás

Általánosítjuk az őst. Újrafelhasználjuk a kódot, újabb metódusokat, adatokat hozhatunk be. Bizonyos esetekben nem lesz helyettesíthető az ős, bizonyos esetekben igen. Például az **Vitorlás motorcsónak** általánosítása a **Vitorlás**nak, hisz szükség eseten, ha szélcsend van, motorral is mehet.

e.) Kibővítés

Kibővítjük az ősosztályt, de megtartjuk az összes jellegzetességét. Nem hozunk be új metódusokat, hanem a meglévő metódusok funkcionalitásait kibővítjük. Helyettesíthető lesz. Például **Vonat** és **Tehervonat**, olyan vonat, amely árút szállít, vagy a **StringLista** olyan **Lista**, amely stringeket, karakterláncokat tartalmaz.

f.) Leszűkítés

Konstrukció szempontjából egy már meglévő osztály bizonyos funkcióitól eltekintünk, és így új osztály jön létre, nem lesz helyettesíthető. Például, ha a **Repülőgép**et úgy definiáljuk, mit egy olyan **Hidroplán**, amely nem tud a vízre szállni. Vagy a **Pingvin** egy olyan **Madár**, amely nem tud repülni.

g.) Egyezés

A hasonló jellegű, hasonló feladatokat megoldó osztályokat egymás alá helyezzük. Logikailag nem teljesen helyes és nem helyettesíthető. Helyette az a megoldás használható, hogy egy közös, általános ősből származtatjuk le őket. Például, ha a **Teherautót** a **Személygépkocsi**ból származtatjuk, abból a meggondolásból, hogy hasonló jellegűek. Helyette az a megoldás javasolható, hogy hozzunk létre egy közös őst, például **Szárazföldi járművek** és mindkettőt ebből származtassuk.

h.) Kombinálás

Tipikus példája a többszörös öröklődés. Összekombinál két vagy több meglévő osztályt. Mint már említettük vigyázni kell vele.

Az öröklődés számos előnnyel rendelkezik. Ilyenek például a kód újrafelhasználhatósága, a kód megoszthatósága, a hasonló interfészek elkészítésének lehetősége, szoftver komponensek, szoftver könyvtárak felállítása és emiatt gyorsabban lehet alkalmazásokat fejleszteni. Nagy előny az is, hogy minden nagyon jól strukturálva, osztályozva van jelen, és az információ-rejtést meg lehet tartani az öröklődési hierarchián belül is. Sajnos az öröklődésnek ára is van, a program lassúbb lesz, hisz meg kell keresni a hierarchián belül a megfelelő metódus-előfordulást, a generált tárgykód mérete is nagyobb lesz, mert a gépi kód nem támogatja az objektumorientált programozást, és a program forrásszövege is komplexebb lesz.

Az öröklődés az OOP második tulajdonsága.

1.2. A közös ős fogalma

Nagyos sokan úgy definiálják elméletileg az öröklődést és az öröklődési hierarchiát, mint *egyetlen gyökérrel rendelkező osztály-fát*. Ez azt jelentené, hogy létezik egy közös ős, egy ős-gyökér, és minden osztály ebből vagy más, már meglévő osztályokból öröklődne. A gyakorlatban, azonban bizonyos programozási nyelvek megengedik azt, hogy az osztályok *„lógjanak a levegőben"*, vagyis semmilyen más osztályból ne öröklődjenek. Kétségkívül, a gyakorlati megoldásnak is vannak előnyei, például az osztályok kisebbek lesznek, hisz eleve nem öröklődik át számos, a közös ősben definiált metódus.

Maradjunk azonban az elméleti, közös ős fogalmánál. Az objektumorientált programozás egyik, messzemenően fontos lényege az, hogy az objektumok kommunikálni tudjanak egymással. Az objektumok között relációk legyenek felállítva. Ilyen értelemben a közös ős fogalma meghatározó, hisz ide lehet csoportosítani az összes olyan metódust, amely a kommunikáció, a jól működés megvalósítása érdekében minden osztály kell, hogy tartalmazzon. Hasonlóan ide lehet csoportosítani az összes olyan metódust, amelyek, például konverziós műveleteket stb. hajtanak végre, vagy minden olyan adatot, amelyekre minden egyes objektumnak szüksége lehet. Ezek a metódusok *absztrakt metódusok* is lehetnek, vagyis olyan metódusok, amelyek csak deklarálva vannak egy osztályban, implementálva nem. A metódust átöröklik a leszármazottak és minden egyes leszármazott írja meg a metódus törzsét, implementálja a viselkedést, az ősosztályból csak a metódus *aláírását* (*signature*), vagyis nevét és paraméterlistáját használják fel. Az így megvalósított egy gyökeres hierarchia könnyebbé, explicitebbé teszi az objektumok működését.

Egy másik nagy előnye a közös ősnek, épp a már említett helyettesíthetőségből származik. Azt mondtuk, hogy minden leszármazott osztály előfordulhat ott, ahol az ősosztály szerepelt, vagyis a leszármaztatott osztályok helyettesíthetik az ősöket. Ez nagyon-nagy előnyünkre válhat a metódusok paraméterezéseinél. Nyugodtan deklarálhatunk, például metódusokat olyan formális paraméterekkel, amelynek típusa az ősosztály, és, amikor szükség van a metódus tényleges meghívására, az aktuális paraméterek lehetnek valamelyik leszármazott osztály példányai. A közös ős minden formális paraméter típusát felveheti.

Ilyen értelemben minden olyan programozási nyelv, amely támogatja a közös ős fogalmát, rendelkezik egy ősosztállyal (ez általában az **Object** vagy a **TObject** nevet viseli), amely minden osztály közös őse, a hierarchia gyökéreleme. Ha egy osztály definíciójakor nem adjuk meg az osztály ősét, automatikusan ez az osztály lesz az ős.

1.3. Absztrakt osztályok, interfészek

Mint már említettük, az öröklődési hierarchia során egyes osztályok tartalmazhatnak absztrakt metódusokat, vagyis olyan metódusokat, amelyek csak deklarálva vannak, implementálva nem. Általánosítva, ha egy osztálynak csak absztrakt metódusai vannak, akkor azt az osztály *absztrakt osztálynak* nevezzük. Absztrakt osztályokat nagyon gyakran használunk, mikor bizonyos általános elveket szögezünk le, csoportosítunk egy osztályba, és a konkrét implementációt a leszármazottakra bízzuk. Absztrakt metódusok vagy absztrakt osztályok esetén az osztály-diagrammunkban ki lehet tenni az **abstract** direktívát, így is ábrázolva azt, hogy a metódusok csak bevezetve vannak, implementálva nem. Némely programozási nyel az absztrakt metódusokat csak olyan szinten engedi meg használni, hogy az implementációs részben leírjuk a metódus fejlécét, de nem írjuk meg a törzset, pontosabban üres törzzsel hagyjuk. Más programozási nyelvekben viszont elég az is, ha csak az osztálydeklarációnál adjuk meg a metódust és szerepel utána az **abstract** direktíva. Beszélhetünk *félig-absztrakt* osztályokról is, ezekben az osztályokban léteznek absztrakt metódusok, de nem mindegyik metódus absztrakt.

Az *interfész* (*interface*) fogalma, olyan absztrakt osztályt fed, amelyből hiányoznak a példányváltozók. Tehát az interfész csak osztályváltozókat és metódusok deklarációit tartalmazza. Az interfész – mint nevéből is látszik – egy felületet biztosít, egy olyan felületet, amely a programban egy absztrakciós szint bevezetésének lehetőségét rejti: a feladat megvalósításának egy bizonyos szintjén el lehet vonatkoztatni a konkrét implementációtól. Ez nagymértékben megkönnyíti a tervezést és növeli a módosíthatóságot.

Egy interfész tényleges használata az implementációján keresztül valósul meg. Egy osztály akkor implementál egy interfészt, ha az összes, az interfész által deklarált metódushoz implementálást ad. Ezáltal az absztrakt program konkréttá válik. Mindenütt, ahol az illető interfész szerepelt, szerepelhet bármilyen, az interfészt implementáló osztály.

Az interfészek között is létezik az öröklődés, tehát az interfészeket is öröklődési hierarchiába lehet szervezni, sőt interfészek esetén a többszörös öröklődésnek semmilyen anomália, akadálya sincs, hiszen, ha hiányoznak a példányváltozók és a metódus implementációk, minden egyértelművé válik. Egy osztály tetszőleges számú interfészt implementálhat. Ha adott egy feladat, amely, például két jól elkülöníthető részfeladatra bontható, és ezt egy osztálynak kell megvalósítania, akkor ez megoldható úgy, hogy a két részfeladathoz tartozó metódusok absztrakt módon bekerülnek két interfészbe, és az osztály mindkettőt implementálja.

A későbbiekben azt is látni fogjuk, hogy az interfészek tulajdonképpen sokkal többek, mint gondolnánk, hisz az a tény, hogy a feladat egy adott pontján el tudunk tekinteni az implementáláshoz, ez oda is vezethet, hogy akár az illető implementálás más nyelven is megtörténhet. Például a **COM** (Component Object Modell) standardra épülő nyelvek át tudják egymásnak adni, egy-egy interfészen keresztül, az osztályokat, metódusokat. Az illető nyelvben csak az interfészt kell deklarálni, a konkrét implementálás más nyelvben történik meg. Ebben az esetben minden osztálynak, interfésznek kell legyen egy – a rendszeren belül – egyedi azonosítója, hogy tudják egymást azonosítani. Windows rendszer alatt, például, ezek az azonosítók a *Windows Registry* adatbázisban vannak nyilvántartva a rendszer által. Egy másik felvetődő probléma a tényleges implementációban rejlő metódushívások vagy paraméterezések, hisz nem minden nyelv oldja meg ezeket egyformán. A **COM** standard azonban lehetőséget biztosít direktívák szintjén ezeknek az egységesítésére. Ilyen értelemben az interfészek programozási nyelvek közötti hidakká váltak és jelentős szerepük van a különböző programozási nyelvekben megírt kódok összehangolásában.

Kovács Lehel