Objektumorientált elvek

A fejezetben áttekintjük az objektumorientált gondolkodás kialakulását, a mögötte álló hajtóerőket és szerepét a tervezésben és a fejlesztésben.

Az 1990-es évek elején kitört az első szoftverkrízis. Hirtelen megnőtt ugyanis a szoftverek iránti igény, de kevés volt a programozó az igények kielégítéséhez. Vezető szoftverfejlesztők végül előjöttek a megoldással, és olyan elveket fektettek le, amelyek alkalmazásával a kód jelentős mértékben újrahasznosítható lett. Ezzel együtt adott szoftver fejlesztésén párhuzamosan több ember is dolgozhatott a granularitásnak megfelelően és az áttekinthetőség, a módosíthatóság is jóval egyszerűbb lett. Ezeket az elveket nevezték el objektumorientált elveknek, amelyek betartása ma már elvárás a szoftverfejlesztési folyamat minden résztvevőjétől. Így nem csak a tényleges kódolás során alkalmazhatóak, hanem a tervezés során is, így a terv már elvekben is megfelel a kódolás során alkalmazandó objektumorientált technikáknak.

Azt ne felejtsük el viszont, hogy az objektumorientált gondolkodásra akkor van elsősorban és feltétlenül szükségünk, ha nagyobb programokat hozunk létre. Egy apró, például 10-15 óra alatt összedobott programocska esetén lehet, hogy inkább akadálynak tűnnek ezek az elvek, úgy érezhetjük, feleslegesen bonyolítjuk az életünket. Ilyen lehet egy programozási beadandó feladat az oktatásban, de ezek az apró gyakorlóprogramok is arra lettek szánva, hogy az objektumorientált elveket elmélyítsék. És egyébként is, ki az, aki egész életében ilyen apróságokat akarna írni?

Az objektumorientáltság paradigma akkor teljesül, ha mind a tervezés, mind a kód elkészítése során betartjuk a megfogalmazott elveket. Ezek ismerete így nélkülözhetetlen a siker érdekében. Jelenleg nagyon sok szoftverfejlesztő cég még elválasztja a tervezés és a kódolás során történő alkalmazásukat, a tervezésnél nem, vagy csak korlátozott mértékben veszik figyelembe ezeket az elveket, azzal az indokkal, hogy ezek csak a kódolásra vonatkoznak. Így viszont gyakori, hogy a klasszikus szoftvertervezés megvalósulása és a kódolás megkezdése között egyfajta újratervezés történik, ahol a jelenleg többségében alkalmazott objektumorientált programnyelvek követelményeinek megfelelően átfogalmazzák a már elkészült terveket. Célszerű lenne ezért mind a tervezésben, mind a megvalósításban következetesen alkalmazni ezeket az elveket. A következő fejezetekben ezeket fogjuk - példákon keresztül - megismerni.

Még egy fontos dolog, amelyről beszélni kell. A minap elém került egy cikk, amelynek címe az volt: Melyek a legnehezebb elemek az objektum orientált programozás megtanulása során? A cikk sok mindent felsorolt és kicsengése az volt, hogy roppant nehéz dolog ennek elsajátítása.

Szerintem ez egy csöppet sem igaz, csak a jó oldaláról kell megközelíteni a tanulást. Nem az a lényeg, hogyan programozzunk objektumorientáltan, hanem az, hogy szét tudjuk választani a tervezést és a kódírást. Az utóbbi talán könnyebb, mert bármely objektum orientált programnyelv megszabja a kereteket, vezeti a kezünket, hogy hol és mit lehet megvalósítani. Nézzünk erre egy példát!

Egy autóverseny játékhoz autókat kell programoznunk. Az gondolom nyilvánvaló, hogy az autók

objektumok lesznek, de milyen attribútumokkal (adatokkal) és milyen operációkkal (funkciókkal) rendelkeznek? A sokféle funkciót a tervezés során határozzuk meg, ebből kettő eléggé nyilvánvaló, mert az autóverseny lényegét adják, ez a gyorsítás és a lassítás (fékezés).

Meghatározásuk után az objektumorientált elveknek megfelelően a funkciók metódusok (függvények) lesznek, az autó pillanatnyi sebessége pedig attribútum, azaz az autó objektum belső adata. Ez mindig leolvasható, de beállítása csak a gyorsítás és a fékezés metódusokon keresztül történhet. Persze ez nem lehet ugrásszerű, de ezt meg tudjuk valósítani a metódusokon belül, csupán ötletes programozó kell hozzá.

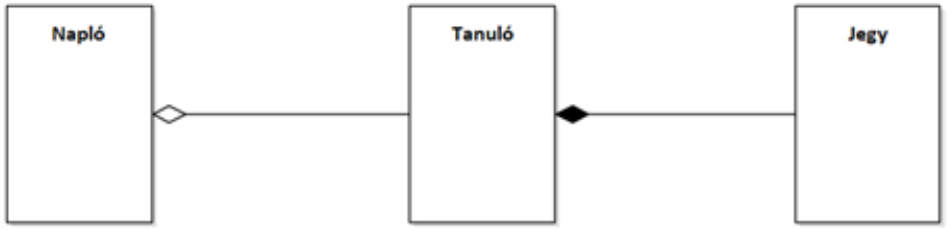
Így rögtön látható, hogy a tervezés (mit lehet csinálni az autóval?) és a programozás (hogyan valósítsuk meg a gyorsítás és fékezés funkciókat?) nem keveredik, a tervező csak a játék egészének működésével foglalkozik, míg a programozó megvalósítja a tervező által megálmodott funkciókat.

Osztály és példánya, az objektum

A fejezetben az objektumorientált elvek alapfogalmait, az osztály és az objektum szerepét járjuk körül.

Az objektumorientált elvek megvalósításához be kellett vezetni néhány új fogalmat és ki kellett alakítani a fogalmaknak megfelelő programnyelvi konstrukciókat. Az alapvetően új konstrukció az osztály volt (angolul class).

Tervezési szinten ezt úgy tudjuk kezelni, hogy a szoftverrendszer összefüggő adatait, mint az egyes osztályok adattagjait jelenítjük meg. Megállapítjuk a követelmények alapján, hogy ezek az adatcsoportok milyen kapcsolatban vannak egymással. A kapcsolatok lehetnek rész-egész tartalmazási kapcsolatok, mint például egy elektronikus osztálynapló esetén az osztály tanulóit tartalmazó Napló osztály, vagy - több más adattal együtt - a tanuló jegyeit tartalmazó Tanuló osztály.



A tartalmazási kapcsolat minden esetben objektumok tartalmazását jelenti a gyakorlatban, azok az adatok, amelyek leírnak egy érdemjegyet, történetesen a Jegy osztály adattagjai és az ebből képzett objektumok kerülnek tárolásra. Jelezhetjük az eltérő jellegű tartalmazásokat is, az ábrán a Tanuló kompozíció jelleggel tartalmazza jegyeit, míg a Napló aggregáció jelleggel tartalmazza a tanulókat. Tervezési szinten érdemes a hétköznapi nyelven megfogalmazható dolgokat ténylegesen is hétköznapi módon kezelni. Ennek megfelelően kimondhatjuk a konkrét esetben, hogy a Napló Tanulókat tartalmaz, a Tanuló pedig Jegyeket, ami - lássuk be - minden papíralapú osztálynapló hagyományos felépítése.

Az objektumorientált elvek értelmében a valós életben felismerhető objektumok adják a szoftver

objektumait is. Úgy is mondhatjuk, hogy hűen leképezzük a valós rendszert a szoftverben. Ennek a megközelítésnek óriási előnye, hogy rögtön érthetővé teszi a szoftvert az átlagos felhasználó számára, hiszen az általa ismert szakterületi fogalmak változtatás nélkül jelennek meg a szoftverben.

Ezt programozási, kódírási szinten úgy képzelhetjük el, hogy egy kódblokkban, amit a class kulcsszó jelez, változókat (azaz attribútumokat) és metódusokat (azaz függvényeket, eljárásokat, egységes névvel operációkat) deklarálunk. Arra kell törekednünk, hogy a metódusok csak a kódblokkban deklarált változókon dolgozzanak. Persze elképzelhetők olyan osztályok is, amelyek nem tartalmaznak változókat, csupán metódusokat – ezek úgy néznek ki, mint a programnyelvek korábban is meglévő függvény gyűjteményei.

A szoftver működtetése során a megírt osztályokból hozunk létre objektumokat. Ez az elektronikus osztálynapló esetében például azt jelenti, hogy a tanuló felelete esetén létrehozunk egy Jegy típusú objektumot, azt feltöltjük a jegy megszerzésének körülményeivel (dátum, tantárgy, amiből kapta, osztályzat értéke, tanár, aki adta, stb.). Ezt az objektumot átadjuk a konkrét Tanuló objektumnak megőrzésre, ahogy a papír alapú naplóba beírnánk a tanuló lapjára az érdemjegyet.

Persze az objektum csak addig él a memóriában, amíg a szoftver fut. Elmentése általában adatbázisba történik, amelynek felépítése tükrözi a szoftver objektumokkal leírható adatszerkezetét, így megfeleltethető annak. Ezt hívjuk objektum-relációs leképezésnek (ORM, azaz Object-Relational Mapping), amely nagy felfedezése az objektumorientált gondolkodásnak és lehetővé teszi, hogy az objektumorientált világot és a relációs adatbázisok világát programozói eszközökkel össze tudjuk kapcsolni.

Ellenőrző kérdések:

Mi az objektumorientáltság elvi megvalósításának az alapvető konstrukciója?

Mit értünk rész-egész tartalmazási kapcsolat alatt?

Mit értünk objektumrelációs leképezés alatt?

Egy tervezés alatt levő szoftverben hogyan állapítjuk meg annak objektumait?

Az objektumorientált világban mi az attribútum, és mi az operáció?

Interfész (interface)

A fejezet az objektumorientált tervezésben az absztrakciót biztosító interfészt tárgyalja, ami lehetővé teszi a megvalósítástól független funkcionális leírást.

Az osztályok típusokat írnak le – ez igaz minden objektumorientált alapon felépített programozási nyelvben – és az ismert osztálynak megfelelő típusú változó létrehozható. Az öröklődés lehetősége biztosítja azt, hogy a leszármazott rendelkezzen az ős típusával is, akár a teljes öröklődési hierarchián keresztül. Így biztosak lehetünk abban, hogy a leszármazott osztály objektumának viselkedése

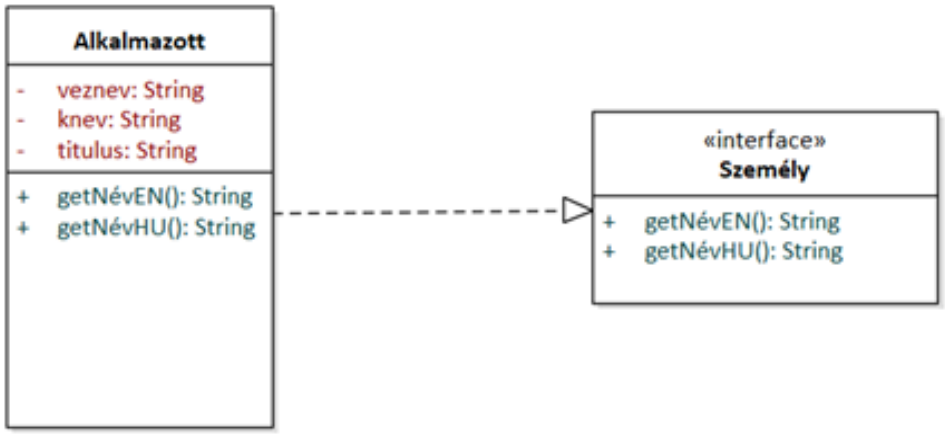
megegyezik az őseinek a viselkedésével, azaz ugyanazokat a funkciókat kínálja fel.

Előfordulhat azonban olyan igény is a tervezés vagy a programozás során, hogy az öröklési rokonságba nem hozható objektumoknál is szeretnénk közös viselkedést, funkcionalitást megvalósítani. Elvben lehetőségünk lenne - és egyes programnyelvek támogatják is – a többszörös öröklődésre. Ezzel mintegy össze tudnánk szedni a szükséges viselkedéseket több, rokonságban nem álló osztályból, ez azonban számos problémát felvet. Az egyik ilyen probléma, hogy nehezen tudnánk olyan osztályokat találni, amelyek teljes mértékben örökíthetők, azaz összes funkciójukat érdemes megvalósítani a leszármazottban. Itt is megjegyezzük, hogy az öröklődés alapelve az "olyan, mint" szabály, de ez nem helyettesíthető egy elméleti "olyan, mint, de nem mindenben" szabállyal.

Az ellentmondás feloldására született meg az objektumorientált rendszerekben az interfész, mint tervezési és programozási elem. Feladata az, hogy leírjon egyfajta viselkedést, amely viselkedést az interfész implementálásával, vagy más szóval megvalósításával egy-egy osztály átvesz az interfésztől. Úgy foghatjuk fel, hogy ezzel tulajdonképpen kiegészíti a saját funkcionalitását. Mivel interfészt egy osztály tetszőleges számban implementálhat, a kívánt funkcionalitást és így az öröklődési viszonyban nem álló osztályok összes objektumára vonatkozó viselkedést tetszés szerint összeválogathatjuk.

Természetesen nem csak a programnyelvek "gyári" interfészei léteznek, bármikor írhatunk saját interfészt, amikor közös viselkedésre van szükség a fejlesztés alatt álló szoftverben. Interfész alapján elkezdhetjük megtervezni a tesztelést is, anélkül, hogy a kód rendelkezésre állna.

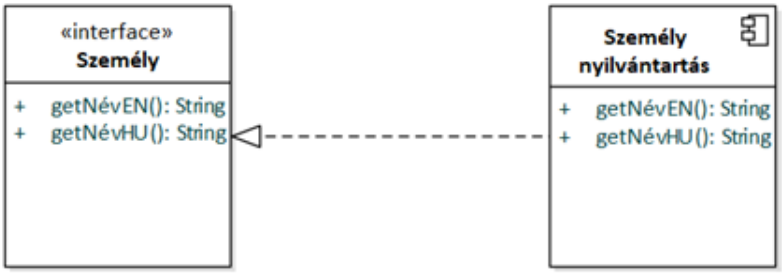
A szoftverek tervezése során interfészekkel leírhatunk olyan viselkedéseket is, amelyeket nem egy-egy konkrét osztály valósít meg, hanem például a szoftver egy komponense vagy modulja. Szolgáltatásokat, mint például webszolgáltatásokat is megadhatunk egy-egy interfész formájában.



Az ábrán egy osztályt és egy interfészt mutatunk be, ahol az interfész határoz meg funkciókat – jelen esetben azt, hogy az őt implementáló osztály a saját név adatelemeiből össze kell fűzzön (konkatenáljon) egy magyar elrendezésű teljes nevet, illetve egy angolszász elrendezésű teljes nevet. Ezeket a funkciókat itt a baloldalt látható Alkalmazott osztály valósítja meg, de biztosak lehetünk benne, hogy bármely, az interfészt implementáló többi osztály rendelkezni fog ezekkel a funkcionalitásokkal. A köztük feltüntetett

nyíl pontosan ezt a kapcsolatot jelzi, így bármely fejlesztő számára egyértelmű lesz.

Tervezés során modulra is vonatkoztathatjuk ezt az implementációt, ahogy ez a következő ábrán látható:



Ellenőrző kérdések:

Miért hasznos az interfész az OO-világban?

Hogyan nevezzük az interfésznek megfelelő függvények kidolgozását?

Mire vonatkozhat egy interfész?

Mi a közös egy osztályban és egy interfészben?

Mi az öröklődés alapelve?

Egységbezárás (encapsulation)

Ebben a fejezetben az objektumorientált gondolkodás egyik alapvető ellentmondásának feloldását kíséreljük meg. Egyrészt az osztálynak el kell rejtenie belső szerkezetét, másrészt az így elrejtett adatokra ugyanakkor szükségünk van a szoftver működéséhez.

Egy szoftverben számos, egymástól adat struktúrájukban és üzleti logikájukban eltérő objektum lehet, amelyek egymással kommunikálnak, azaz kéréseket indíthatnak (szolgáltatásokat nyújtanak) egymás felé. Ennek során az objektumok az állapotukat, azaz az adataik értékeinek összességét, elrejtik egymás elől. Adataik természetesen változhatnak a kérések hatására, de ez sohasem közvetlenül történik, hanem csak publikus funkció (metódus) hívásokon keresztül. Ehhez az szükséges, hogy az adatok (attribútumok) csak az objektumon belül legyenek láthatóak, elérhetőek, úgy mondjuk, hogy privát láthatósággal rendelkeznek. Lehetnek olyan metódusok is (többnyire ezek úgynevezett segédmetódusok), amelyek szintén privát láthatóságúak, mert csak az objektumon belül van rájuk szükség.

Mielőtt továbbmennénk, jegyezzük meg, hogy ennek megvalósítása már programozói feladat, a tervező

munkája véget ér a szakterület osztályainak megtervezésével, az adatszerkezet leírásával. Arra viszont gondolnia kell, hogy az adatok kívülről nem érhetők el, tehát nem tervezhet olyan folyamatokat, ahol az egyik objektum közvetlenül használja fel a másik adatait.

Úgy is mondhatjuk, hogy az objektum az állapotát, azaz adatai értékét saját maga menedzseli, más objektum ezekhez nem fér hozzá közvetlenül. Amennyiben kommunikálni akarunk az objektummal, csak az objektum által biztosított publikus metódusokat használhatjuk, az állapotát (adatai értékét) közvetlenül nem befolyásolhatjuk.

Nézzük meg ezt egy játékos példán, ami ugyanakkor jól bemutatja ezt a koncepciót.

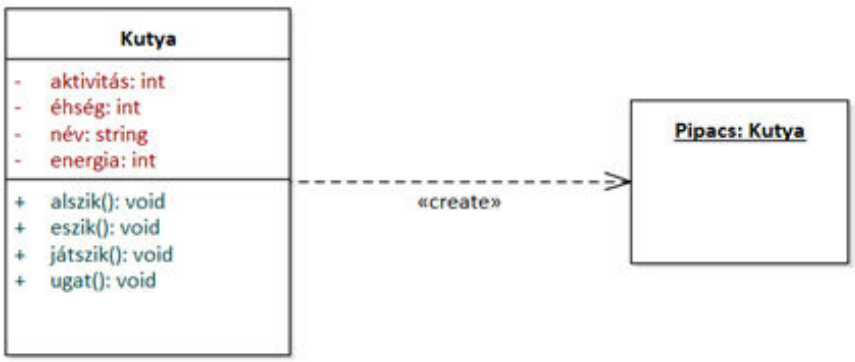
Családunkba egy beagle-vizsla-golden retriver keverék kiskutya érkezett, aki a Pipacs nevet kapta. Természetesen lánykutya, amit a neve is sugall. Nagyon szeret játszani, főleg ugrálni, amit boldog ugatással kísér, és amikor elfárad, nagyokat alszik. Mikor felébred és éhes, ugatással kér enni, az élelem hatására aktivitása ismét a csúcson van. Ezt nagyon jól tudjuk modellezni objektumorientált alapokon.

A Kutya osztály egy példánya (objektuma) Pipacs nevet kapott (ez a név attribútum értéke lesz), továbbá az aktivitás, éhség, energia attribútumok tárolják a kiskutya állapotát. A privát ugat() metódus pedig a természetes hangját állítja elő. A játszik() metódus hatására az aktivitása nő, azaz boldogan ugrál és ugat(), de az energiái fokozatosan lemerülnek.

Az alszik() metódus megnöveli energiáját, de az éhségét is fokozza, természetesen álmában nem, vagy csak halkan ugat.

Az eszik() metódus csökkenti az éhségérzetet (csökken az attribútum értéke), és megnöveli az aktivitását, valamint ezt a tevékenységet a lelkes ugat() metódus kíséri.

Bár közvetlenül nem férünk hozzá az állapotot alkotó attribútumok értékeihez, a publikus játszik(), alszik() és eszik() metódusok, a privát attribútumok változtatásain keresztül mindig beállítják az állapotot a tevékenységnek megfelelően. Ezt a kapcsolatot osztály és objektum között grafikusan is meg tudjuk jeleníteni:



Lehet, hogy a példa komolytalannak tűnik, de játékprogramjaink mind így épülnek fel! Gondoljunk bele kedvenc játékprogramjainkba és könnyen felismerhető lesz ez a fajta működés. Amikor egy karakter

életereje nő vagy csökken a különböző játékhelyzetekben, akkor a növekedésnek vagy a csökkenésnek megfelelően a háttérben tulajdonképpen a karaktert képviselő objektum belső állapota módosul.

Ne féljünk az emberi gondolkodás megjelenítésétől! Nem akkor lesz egy program jól használható, ha elvont változókat vezetünk be, amelyeket valószínűleg csak a kód szerkesztője fog érteni, hanem az úgynevezett beszédes változókkal és metódusokkal mindenki számára egyértelművé tesszük, mit is csinál a program.

Ellenőrző kérdések:

Mit nevezünk az OO gondolkodás alapvető ellentmondásának?

Mit értünk egy objektum állapota alatt?

Mi a közös az OO elveken felépülő szoftverben és az emberi gondolkodásban?

Hogyan kommunikálnak egymással az objektumok?

Miért nem előnyös, ha az objektum adatai közvetlenül elérhetők?

Absztrakció (abstraction)

Objektumorientált gondolkodásunk egyik alaptétele, hogy igyekszünk egymástól elválasztani a funkciók leírását és a funkciók megvalósítását – ezt nevezzük absztrakciónak. Ezt a problémakört járjuk be ebben a fejezetben.

Amikor az objektumorientált tervezés feltétlenül szükséges, a programjaink általában nagyok, bonyolultak, és sok objektum együttműködésén alapulnak. Ezek a szoftverek hosszú időn át használatban vannak, így az elkerülhetetlen módosítások miatt a karbantartásuk nem egyszerű feladat. Az absztrakció elve ezt a problémát célozza meg, azt írja elő, hogy mindegyik objektum, és ezt kiterjesztve mindegyik modulja a szoftvernek csupán magas szintű funkcionalitásokat adjon közre önmaga felhasználására.

Lényegében ezen a mechanizmuson keresztül el lehet rejteni a felhasználók számára nem nyilvános megvalósítási részleteket, és csak azokat a funkciókat tesszük láthatóvá, amit az egyes felhasználók – és ezek lehetnek személyek, de más rendszerbeli objektumok is - fel tudnak használni a saját tevékenységük elvégzéséhez.

Így a funkcionalitás használata egyszerű, mert nem változik, vagy csak ritkán változik a szoftver élettartama során. Ugyanakkor a megvalósítás szabadon változtatható – úgy mondjuk szakmai szóhasználattal, hogy az implementáció cserélhető. Úgy gondolhatunk erre, mint egy kisméretű, publikus funkciócsoportra, amelyet a felhasználó nyugodtan kezelhet anélkül, hogy tudná, hogyan van megvalósítva az adott funkciócsoport, mi az aktuális működés a háttérben. Sok példát felhozhatunk erre, ha belegondolunk, praktikusan a minket körülvevő gépek mind ilyenek. Tekintsünk például egy

gépkocsira, lehet benzines vagy dízel, vagy elektromos, de a kezelése ugyanolyan elven és kontrollokkal történik. Az informatikai területhez közelebbi példa lehet a különböző szövegszerkesztő programok változása az idők során. Bár a tárolási módjuk, az eredményül kapott fájlok mérete jelentősen megváltozott, a használatuk változatlan és talán nem is akarjuk tudni, hogyan épül fel egy adott fájl karakterszinten.

Kimondhatunk tehát egy fontos szabályt. Számunkra többnyire az a jó, ha egy szoftver frissítése, azaz az implementáció megváltozása nem, vagy csak ritkán változtatja meg az adott szoftver kezelését. Nem vonatkozik ez persze a várva várt hibajavításokra.

Ellenőrző kérdések:

Mit értünk a szoftverfejlesztésben absztrakció alatt?

Milyen terhet jelentenek az elkerülhetetlen módosítások a szoftverek esetében?

Mit értünk az implementáció cserélhetősége alatt?

Kik lehetnek a felhasználói egy adott funkciónak vagy funkciócsoportnak? Mondj példát ezekre!

Miért jó az, ha az implementáció megváltozása rejtve marad?

Öröklődés (inheritance)

Talán ez a legismertebb objektum orientált fogalom, hiszen az objektumorientált elvek bevezetésének a fő hajtóereje az újrafelhasználás lehetőségének kialakítása volt. Ezt a fogalmat tárgyaljuk részleteiben ebben a fejezetben.

Az volt az eredeti elképzelés, hogy alakítsunk ki olyan osztályokat, adat és kód kombinációkat, amelyek önmagukban vagy úgynevezett leszármazottak útján ismételten felhasználhatók. Felmerülhet, miért előnyös a közös ős, hogyan léphetünk előre ennek következetes alkalmazásával?

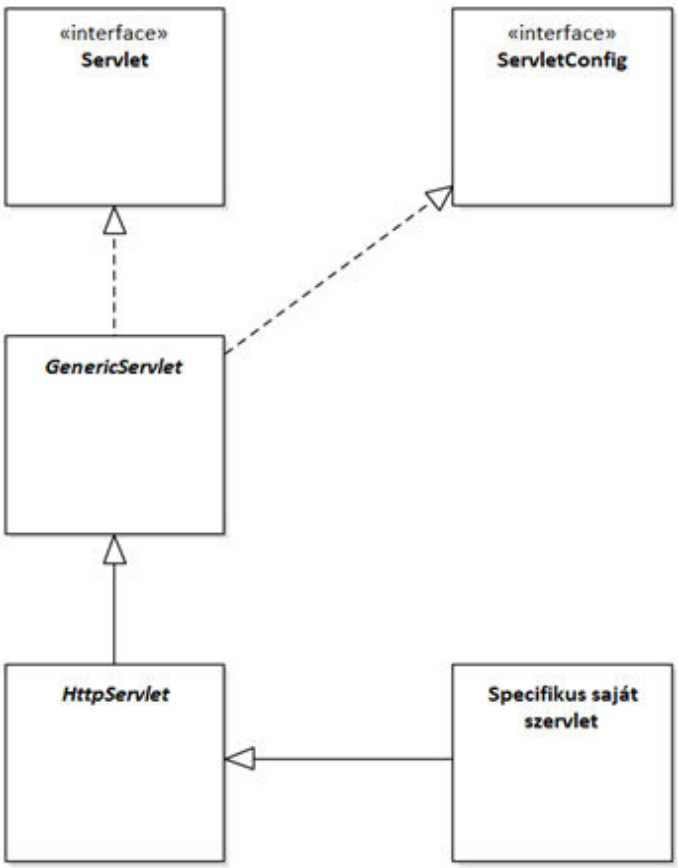
Erre többféle példát is megadhatunk, kezdjük a programozási keretrendszerek kérdésével!

Első példánk valós alapokon nyugszik, a Java nyelvben pontosan így került megvalósításra! Szeretnénk egy keretrendszert felállítani, ahol a konkrét tevékenység ugyan változhat, de az adat bemenet és a kimenet mindig állandó. Természetesen leírhatjuk a követelményeket, és a programozóra bízzuk ezek megvalósítását, de az ember, még ha programozó is, hibázhat. Így nem lehetünk biztosak abban, hogy minden esetben a követelményeknek teljes mértékben megfelelő kód készül. A szoftverben természetesen nem engedhetünk meg semmiféle lazaságot, a keretrendszert működtető metódusoknak és a közös adatoknak még az elnevezése is esetről esetre pontosan azonos kell legyen.

Készítsünk hát egy osztályt, amelynek példányai, az objektumok egy webszerver keretében futnak, és képesek a kliens által küldött kérést teljesíteni és választ adni – plusz igény, hogy ehhez a kéréshez

adatokat is kell tudnunk csatolni. A kérés a használt protokoll után a HTTP-kérés (HttpRequest) nevet kapja, a válasz a HTTP-válasz (HttpResponse) nevet. Minden általunk készített osztály és így annak objektumai egy-egy adott tevékenységet végeznek – ezek a szervletek – de adott webszerver számos ilyen objektumot tartalmazhat. A tevékenységek megkülönböztetésére különböző elnevezéseket, címeket használunk, ezek a szervletek egyedi URL azonosítói.

Ahhoz, hogy ez biztonságosan megvalósítható legyen, és emberi tévedések ne tegyék tönkre a működést, a megfelelő Application Programming Interface, azaz röviden API, keretein belül egy absztrakt ősosztályt hoztak létre a nyelv fejlesztői a szükséges metódusokkal. Minden programozói speciális szervlet ennek leszármazottjaként készül el. Így a keretrendszer biztos lehet abban, hogy minden szervlet rendelkezik a működéshez szükséges metódusokkal. Arról a fordítóprogram, a compiler gondoskodik, hogy ezek felül legyenek írva, azaz működőképesek legyenek.

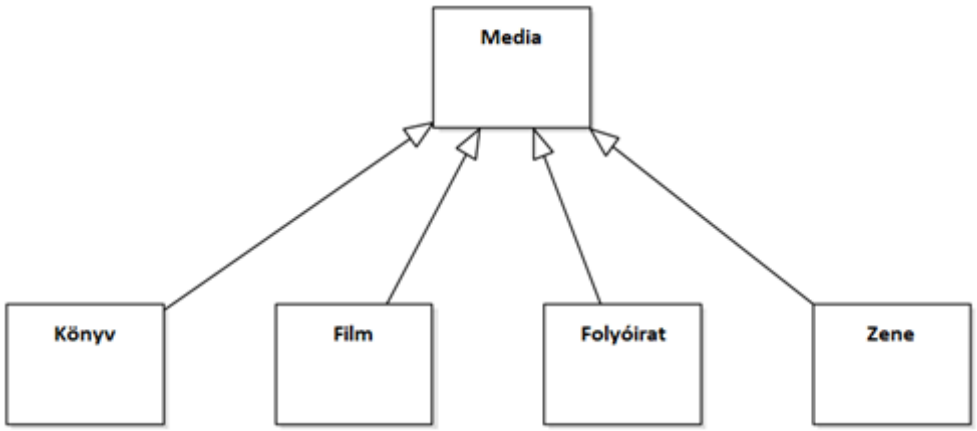


Látható az ábrán, hogy a saját szervletünk egy öröklődési és implementációs hierarchiában kapja meg az összes szükséges funkcionalitást. Ennek a konkrét felépítése a nyelvet kialakító programozók döntése volt, de a gyakorlati felhasználáshoz nekünk csak annyit kell tudnunk, hogy melyik osztály leszármazottjaként kell megvalósítani.

A másik példánk már inkább tervezés, és az adatkezelés köréből származik. Feladatunk, hogy egy könyvtári alkalmazást tervezzünk, amely különböző nyilvántartott, kölcsönözhető vagy helyben olvasható

– nézhető, hallgatható – médiatermékeket kezel. Bár ezek a könyvtári tételek jelentősen eltérhetnek egymástól, mégis van bennük számos közös adat és viselkedés. A viselkedéseket objektumorientált szoftvert tervezve az egyes metódusok, funkciók reprezentálják.

Azzal tehetjük az alkalmazásunkat rugalmassá és bővíthetővé, ha első lépésben meghatározzuk, milyen is legyen általánosságban a könyvtári elemek adatszerkezete, mi az az adatkör és funkcionalitás, amit bármely, a rendszerben kezelt médiának biztosítania kell. Ha ezt sikerül egy absztrakt ősosztályban (Media) összefoglalni, akkor ezen a szinten bármely új médiatípus egyformán kezelhető. A speciális adatokat és viselkedést viszont az ősosztály leszármazottjai tartalmazzák, mint például film esetén a kereshető alkotói listát, vagy a film, videó játékidejét.



Amennyiben a későbbiekben az alkalmazásba új médiatípust kell bevezetni, akkor csupán új média osztályt kell létrehozni örökítéssel, de a teljes alkalmazás egyéb funkcióihoz, mint például a leltározás, kölcsönzés, nem kell hozzányúlnunk. Az ehhez szükséges adatok, metódusok ugyanis az ősosztályban már megfogalmazásra kerültek.

Ellenőrző kérdések:

Miért vezették be az öröklődést az OO elvek kidolgozása során?

Hogyan biztosíthatjuk azt, hogy egy keretrendszerben minden objektum biztosan azonos módon viselkedjen? (Azonosak legyenek a fő funkcionalitásai.)

Hogyan illeszkednek bele az interfészek az öröklődési hierarchiába?

Hogyan valósul meg az absztrakció az öröklődés során?

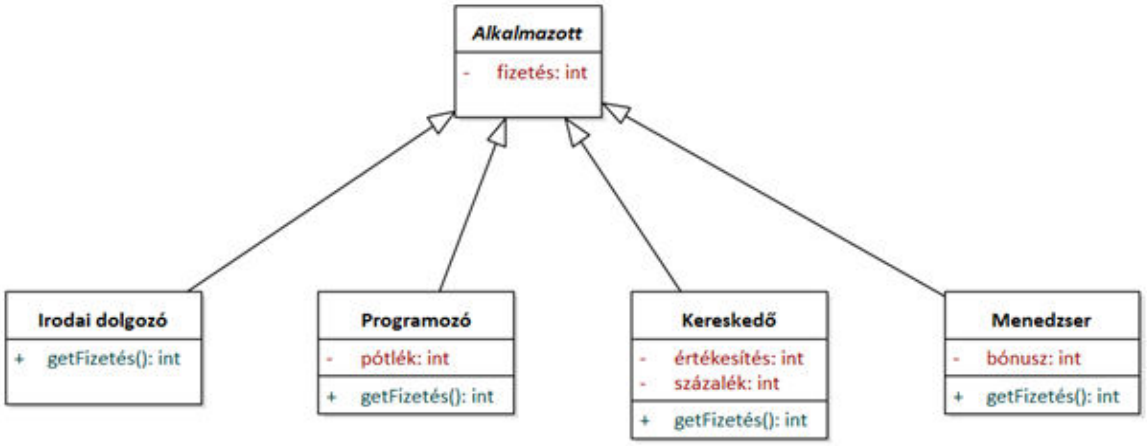
Mi a szerepe egy absztrakt ősosztálynak az öröklődési hierarchiában?

Polimorfizmus

A fogalom könnyen lefordítható, többalakúságot jelent és a görög nyelvből származik – de ezzel persze még nem jutunk közelebb a kifejezés megértéséhez. Ebben a fejezetben azért teszünk egy próbát erre.

Az objektumorientált elvek talán legkevésbé megfogható – és így a legkevésbé érthető – fogalma a polimorfizmus, de azért megpróbáljuk közelebb hozni a megértését. Ehhez a legjobb egy elméleti, bár nem elvont példát létrehozni és annak viselkedésén bemutatni a fogalmat.

Nézzük meg egy szoftverfejlesztő cég (de persze bármilyen cég lehet, csak az alkalmazottak elnevezése változik) személyi nyilvántartását, ahol sok egyéb mellett az alkalmazottak fizetését, illetve az alapbér mellett kifizetett egyéb juttatásokat is nyilvántartjuk. Az objektumorientált elvek értelmében az alkalmazottak mind egy közös ősosztály (Alkalmazott) leszármazottai, és az őket reprezentáló osztályok adják meg az alkalmazottak pontos típusát (ezek lehetnek például: Irodai dolgozó, Programozó, Kereskedő, Menedzser).



Az ábrán csak a számunkra, a példa szempontjából érdekes attribútumokat tüntettük fel. Látható, hogy fizetése mindenkinek van, de ez a Programozó esetén munkaköri pótlékkal egészül ki, a Kereskedő a fizetése mellett az értékesítései alapján százalékos járulékot kap, a Menedzser pedig havi szintre lebontva rendszeres bónuszt kap a céges eredmények alapján.

Számunkra viszont az a legfontosabb, hogy minden alkalmazottat meg tudjunk szólítani azzal a kérdéssel, hogy mennyi a havi fizetése. A válasz független kell legyen attól, hogy mely juttatásokból, milyen számítás alapján tevődik össze ez az összeg. Persze nem az egyes személyeket kérdezzük, hanem a megfelelő objektumot szólítjuk meg. Azt várjuk el, hogy minden objektum a típusának belső üzleti logikája szerint összeszámolja a különböző juttatásokat, és nem nekünk kell a típust figyelembe véve, annak megfelelő külső üzleti logikával ezt a számítást elvégezni.

Így például, ha az összes alkalmazotti objektumot egy kollekcióba gyűjtjük, akkor a gyűjtemény elemein végigfutva akár elemenként, akár összegezve is megkapjuk a kifizetendőket.

A lényeg az, hogy minden osztály és ebből következően minden objektum ismeri a getFizetés() metódust, és a kérdésre típus specifikusan válaszol. Szakmailag precízen megfogalmazva a típustól függő külső algoritmus helyett egy típus specifikus belső algoritmust alkalmazunk. Belátható, hogy egy újabb alkalmazotti típus felvitele a szoftverbe csupán egy újabb, Alkalmazottból leszármaztatott osztály

megírását és tesztelését jelenti és nem kell az üzleti logikát újrafogalmazni.

Ez a megoldás a polimorfizmust használja ki, és jelentősen egyszerűsíti a szoftver módosítását, átalakítását az új követelményeknek megfelelően. Minden objektum típusnak saját getFizetés() metódusa van, és megszólítás esetén értelemszerűen ez a metódus hozza a típusfüggő üzleti logikát, amely alapján az objektum elvégzi a számítást.

Ellenőrző kérdések:

Keress gyakorlati példát a polimorfizmus alkalmazására egy szoftver tervezése során!

Mi a különbség a belső és a külső üzleti logika között?

Miért előnyös a polimorfizmus alkalmazása egy időszakosan változó szoftver esetén?

Mi a szerepe az absztrakt osztálynak a polimorfizmus esetén?

Milyen kapcsolat van egy általános típus és a specifikus típusok között?

Kohézió (cohesion)

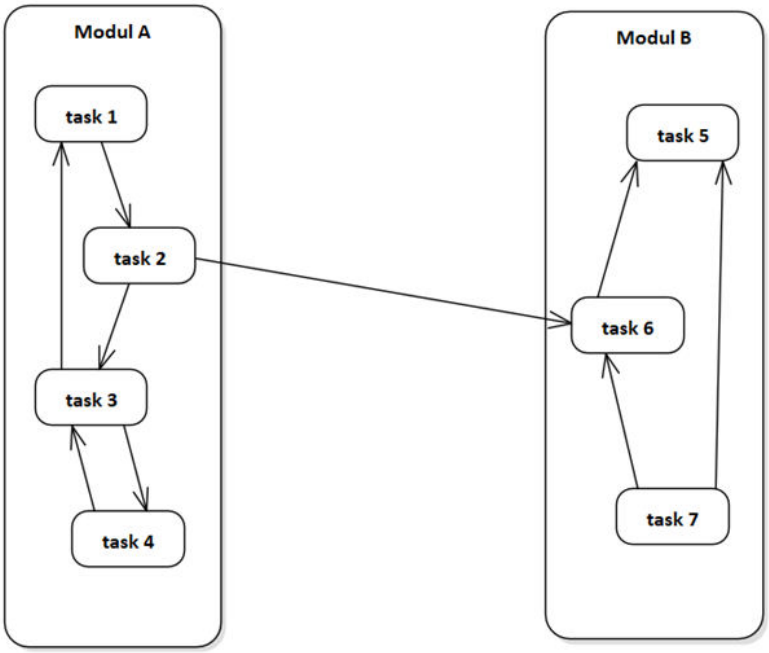
Röviden megfogalmazva a kohézió annak a foka, hogy egy adott modulon belül az elemek milyen mértékben tartoznak össze. Úgy is mondhatjuk, mennyire erős a viszony adott osztály metódusai és adatai, és valamely, az osztály által kiszolgált cél vagy feladat között. A kohéziót vonatkoztathatjuk arra is, hogy az osztály metódusai és adatai milyen mértékben kapcsolódnak egymáshoz. Az erős és a gyenge kohézió hatásait elemezzük ebben a fejezetben.

A kohézió mértékére általában két szélsőséges értékkel hivatkozunk, erős kohézióról (high cohesion) vagy gyenge kohézióról (low cohesion) beszélünk. Nem számszerűsíthető, és így a köztes értékeknek nincs jelentősége. Számunkra az erős kohézió a lényeges, minden más előnytelen megoldást jelent. Ennek oka az, hogy az erős kohézióhoz olyan szoftver jellemzők társulnak, mint a robosztusság, megbízhatóság, újrafelhasználhatóság (ne feledjük, az objektumorientált elvek az újrafelhasználhatóság érdekében lettek megfogalmazva!). Nem utolsó sorban a kód érthetősége függ tőle. Tipikus esetben egy szoftver teljes életciklusa során a módosításokba több munkát fektetünk, mint a szoftver elkészítésébe, így a kód érthetősége elsőrendű fontosságú. A gyenge kohézió mindaz, amely a fenti igényeknek nem felel meg, tehát gyenge minőségű szoftvert jelent.

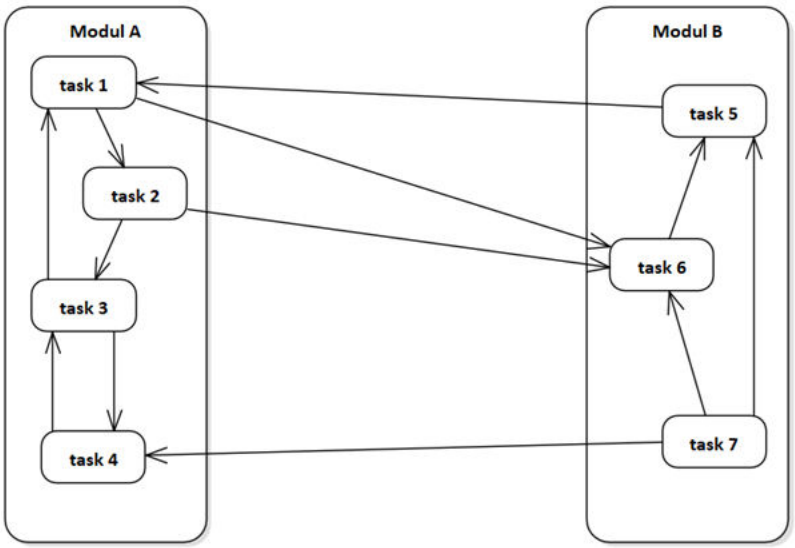
A gyakorlatban például az adatbázis elérést megvalósító osztályok esetében akkor beszélhetünk erős kohézióról, ha egy-egy úgynevezett szerviz osztály csupán egy vagy esetleg egymással szorosan összefüggő adatcsoportokkal kapcsolatos kéréseket szolgál ki, és a szorosan nem kapcsolódó adatcsoportokkal kapcsolatos kéréseket külön osztályok szolgálják ki. Ennek előnye azonnal belátható,

mert ha egy adatcsoporthoz tartozó szolgáltatások módosításra kerülnek, mindig csupán az adott osztály kódját kell bővíteni. Ez a bővítés, vagy módosítás biztosan nem fogja befolyásolni a többi adatcsoport lekérdezéseit, így a kód átláthatóbb, jobban kezelhető és nem utolsó sorban kevesebb tesztelés is elegendő az üzembe állításhoz.

Ezt a koncepciót jól kiemeli az alábbi ábra is, két erős kohéziót megvalósító modul kommunikációját bemutatva:



A következő ábrán két hasonló, de gyenge kohéziót megvalósító modul látható. Gondolom, első pillantásra is eldönthető, melyik a követendő példa:



Összefoglalva az alábbiakat jelenthetjük ki a kohézióval kapcsolatban:

A kohézió foka emelhető, ha az egy-egy osztályban megvalósított metódusok, amelyek összességükben az osztály funkcionalitását adják ki, sok közös vonást mutatnak. Gondoljunk egy szövegdarabokat – idegen szóval stringeket – kezelő osztályra, ahol az osztály minden metódusa az objektum létrehozásakor paraméterként kapott adaton dolgozik (itt persze ez az adat szöveg, azaz string típusú). Lehetnek olyan metódusok, amelyek információt adnak a stringről, olyan metódusok, amelyek a stringet átalakított formában adják vissza (például kisbetű, nagybetű konverzió), de mindegyikben közös, hogy az attribútumként tárolt stringet nem változtatják meg. A kohézióra törekvés értelmében egy külön osztályt érdemes létrehozni arra, hogy a stringet tovább tudjuk építeni, manipulálni és az így módosított stringet tároljuk a továbbiakban.

Fentiekből következik, hogy mindegyik metódus csak egy erősen korlátozott mennyiségű tevékenységet végez. Amennyiben a tevékenységek kombinációjára van szükség, ezt az osztályon kívül valósítjuk meg, vagy olyan metódust írunk az osztályban, amely ezeket belső hívások segítségével összekapcsolja.

Az erős kohézió megvalósítása több előnnyel jár. Csökkenti az adott modul (osztály, komponens) komplexitását, azaz ezek egyszerűbbek, kevesebb műveletet végeznek el.

Hatására megnövekszik a rendszer karbantarthatósága, mert az adott rendszerben végzett változtatások kevesebb modult érintenek. Úgy is fogalmazhatunk, hogy az adott modulban elvégzett változtatások nem igényelnek módosítást a többi modulban.

Nem utolsó sorban a modulok újrahasznosíthatósága növekszik, mert a fejlesztők könnyebben megtalálhatják a számukra szükséges funkciókat egy logikusan szervezett, összefüggő funkcionalitásokkal rendelkező modulban. Gondoljunk csak az egyes osztálykönyvtárakra, amelyet akár az adott objektumorientált programnyelv hivatalos fejlesztője, akár fejlesztői közösségek biztosítanak.

A gyakorlatban sokszor persze nem ilyen egyszerűen oldható meg a kohézió kérdése. Meg kell találni az egyensúlyt az egy funkcióba összezsúfolható, de még áttekinthető elemi tevékenységek és a tevékenységek funkciókba szétosztása között. Egyensúlyt kell tehát teremteni az egység komplexitása és a további elemekhez való kapcsolódás között.

Az egyensúly fontossága miatt a tananyagban külön fejezet foglalkozik a kapcsolás kérdéseivel.

Ellenőrző kérdések:

Hogyan határoznád meg a kohézió fogalmát?

Melyek az erős kohézió (high cohesion) előnyei?

Hogyan segíti elő az erős kohézió megvalósítása a kód érthetőségét? Miért fontos az érthetőség?

Egy moduláris felépítésű rendszer terve alapján hogyan lehet felmérni a kohézió mértékét?

Mit értünk egyensúly alatt a kohézióval kapcsolatban?

Kapcsolódás (coupling)

A kapcsolás és a kohézió az objektumorientált gondolkodásban látszólag ellentétes fogalmak, pontosabban egy alapvető probléma kétféle megközelítését adják. Ebben a fejezetben a kapcsolás nézőpontjából vizsgáljuk meg a problémát, de röviden emlékeztetünk a kohézió fogalmára is.

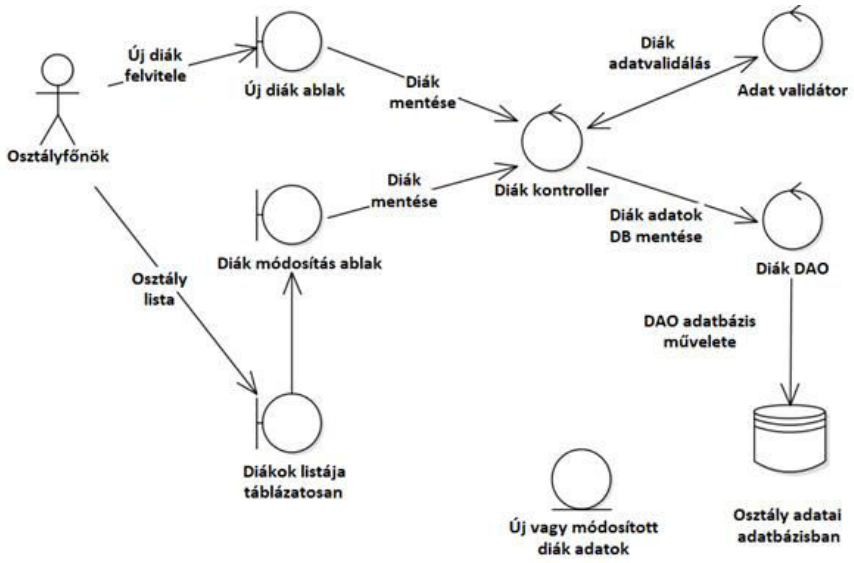
A kohézió annak mértéke, hogy egy-egy objektum milyen mértékben fókuszál adott feladat elvégzésére, mennyire tudja kizárni a feladat végrehajtásához nem szükséges egyéb tevékenységeket. Félretéve az antropomorf megközelítést, nem az objektum ilyen okos, hanem a fejlesztője vette sikeresen ezt az akadályt, és olyan objektumot alkotott, ahol ez megvalósul.

Van egy másik fontos alapfogalom is az objektumorientált gondolkodásban, és ez a kapcsolás (angolul coupling). Gyakran úgy tekintünk rá, mint a kohézió ellentétjére, mert a laza kapcsolás többnyire erős kohézióval jár együtt. Ennek megfordítottja is igaz, gyenge kohézió esetén gyakran tapasztalunk erős kapcsolást.

A kapcsolás annak mértéke, hogy egy-egy objektum a feladata elvégzése során mennyire veszi igénybe más objektumok operációit, mennyire függ a többi objektumtól. Az elég könnyen belátható, hogy ha egy funkció megvalósítása során több objektum szoros együttműködésére van szükség, a szoftverrendszerünk hamar monolitikussá válhat. Ennek következtében viszont minden szoftvermódosítás és -karbantartás nagy erőfeszítést és magas anyagi ráfordítást igényel, hiszen a módosítások számos kapcsolódó objektumot egyszerre érintenek.

Sem a kohézió, sem a kapcsolás nem új fogalmak, az objektumorientált koncepció már a strukturális programtervezéstől örökölte ezeket. Megfogalmazásuk az 1960-as évek második felében, részletezésük és publikálásuk az 1970-es években történt. Ma úgy tekintünk rájuk, mint szoftverfejlesztési standardokra, amelyek betartása automatikusan jól karbantartható, könnyen módosítható szoftverekhez vezet.

Nézzük meg egy példán, hogyan valósul ez meg egy szoftver tervezése során. A példánkban szereplő szoftver adatbázisban tárolt adatokat kezel (azaz ment, módosít, lekér és töröl).



Az ábrán látható, hogy az egyes adatmentési, adatmódosítási lépések külön elemeket igényelnek, és mindegyik elem csak a saját feladatával törődik. Még az olyan, beépülőnek képzelhető tevékenység is, mint az adatellenőrzés, külön komponens feladata lett. Az ábrán ezt a tevékenységet az általánosan elfogadott validálás kifejezéssel jelöljük. Mivel laza kapcsolásról van szó, bármely elem a többitől függetlenül módosítható, továbbfejleszthető, és ezek a változtatások nem érintik a kliens működését.

Ellenőrző kérdések:

Hogyan határozhatunk meg egy laza kapcsolást megvalósító szoftver rendszert? Melyek a jellemzői?

Hogyan határozhatjuk meg a kapcsolás mértékét?

Mondj példát a laza kapcsolás megvalósítására adatbázis elérés esetén!

Mi az előnye annak, ha követjük a laza kapcsolás megvalósításának elvét?

Miért hátrányos az, ha szoftverrendszerünk monolitikus?

Demeter törvényei

Sok próbálkozás van és volt már arra, hogy a szoftverfejlesztést tudományos keretek közé szorítsák. Ezek több-kevesebb sikerrel jártak, vannak olyanok, amelyek fennmaradtak, mások eltűntek az idők során. Az egyik ilyen, jelenleg is ismert és javasolt megoldással foglalkozunk ebben a fejezetben.

Demeter törvényeit 1987-ben alkották meg, pontosabban javasolták alkalmazásukat a szoftverfejlesztéseben. Szülőhelye a Demeter projekt volt, amely elsősorban az úgynevezett adaptív

programozás és aspektus-irányult programozás elveinek kidolgozására koncentrált.

A törvényre, mint a legkevesebb tudás elvére is hivatkozhatunk, amely hétköznapi nyelven azt jelenti, hogy egy modul vagy komponens (modulként érthetünk itt bármely objektumot is) bármely egyéb komponens szerkezetéről vagy tulajdonságairól a lehető legkevesebb információval rendelkezik.

Ha belegondolunk, ez az elv tulajdonképpen a laza kapcsolás egy specifikus esete, tehát ugyanazokkal az előnyökkel és hátrányokkal kell járnia. Másként megfogalmazva, minden egyes egység csak a "barátokkal" kommunikál, semmi esetre sem az idegenekkel – ez egyenesen adódik abból a tényből, hogy még a barátait sem ismeri azok minden részletében, az idegeneket viszont egyáltalán nem ismeri.

Természetesen minden modul rendelkezik elegendő információval ahhoz, hogy a feladatát el tudja látni, de csak annyival. Ez az elv a tervezésben is érvényesíthető. Nézzünk meg erre egy példát a mindennapi életünkből.

Tegyük fel, hogy egy, a posta munkáját támogató szoftvert kell megterveznünk. A rendszer felépítése moduláris, és azt tűztük ki célul, hogy a tervezésben követni kell Demeter törvényeit. Modulokat tervezünk, ahol az elképzelés szerint lesz kézbesítő modul, felvevő modul, szortírozó modul és szállító modul. Minden modul csak az üzleti folyamatokban meghatározott módon léphet kapcsolatba más modulokkal, és csak azokkal, amelyek szomszédosak vele. Itt persze nem fizikai közelséget értünk, hanem a tevékenységek sorrendisége által meghatározott közelséget.

Így például a szállító modul, amelynek feladata a települések, postai körzetek közötti többszintű postaforgalom biztosítása, a megfelelő postai körzet elérése után a küldemények kézbesítésére a kézbesítő modult kéri meg, és nincsen információja arról, hogy az adott postai körzetben milyen más tevékenységek zajlanak, mely modulok aktívak még.

Ezt továbbgondolva adódik, hogy a felvevő modul, amelynek feladata a küldemények postai felvétele, beleértve az árazást és egyéb felvételi ellenőrzéseket, speciális szolgáltatások hozzákapcsolását, a küldeményeket ömlesztve átadja a szortírozó modulnak, amely csoportosítja azokat célirány szerint, majd az egyes küldemény csomagokat átadja a szállító modulnak.

Minden modul a Demeter törvények értelmében csak a folyamatban következő modul szolgáltatásait veszi igénybe a minimális tudás értelmében. Amennyiben a küldemények nyomkövetését is meg kell oldani, akkor ehhez egy új modult érdemes tervezni. Ez végig tudja nézni minden modul "küldemény tartalmát", de nem rendelkezik információval azok sorrendiségéről, vagy egymás felé nyújtott szolgáltatásairól.

Demeter törvényeinek betartása könnyebben karbantartható és módosítható szoftvert eredményez. Mivel a törvények következetes betartásával az egyes objektumok kevésbé függnek a többi objektum belső szerkezetétől, az egyes objektumok belső szerkezetének megváltoztatása nem jár együtt a kifelé mutatott viselkedés megváltozásával.

Sok esetben a többrétegű architektúrák, felépítési elvek úgy is tekinthetők, mint Demeter törvényeinek szisztematikus megvalósítása a szoftver rendszerekben. Ebben az architektúrában az egyes rétegek kódja csak a rétegen belül hívhat meg funkciót, illetve a következő réteg kódjához fordulhat. Egy esetleges

átnyúlás a rétegeken magának a többrétegű architektúrának az elvét sértené meg, ezért szigorúan kerülendő még akkor is, ha egyszerűsítő megoldásnak látszik.

Ellenőrző kérdések:

Mit értünk Demeter törvényének másik megfogalmazása alatt?

Hogyan függ össze a laza kapcsolás elve és Demeter törvénye?

Egy példán mutassa be Demeter törvényének gyakorlati megvalósulását adott szoftverben?

Mit eredményez Demeter törvényének betartása?

Hogyan valósul meg a többrétegű architektúrákban Demeter törvénye?

Kompozíció

Az öröklődés kitűnő megoldásnak látszik a régi funkciók újrahasznosításában, azok kiterjesztésében. Vannak azonban hátrányai, és helyette a kompozíciót javasolják a szoftverfejlesztésben. Erről a megoldásról lesz szó ebben a fejezetben.

Az öröklődés alapelve, hogy minden ős funkciónak élnie kell a leszármazottakban, ami sokszor nem kívánatos a gyakorlatban. Funkciók letiltása az elvek megsértése miatt nem jöhet számításba, és a felesleges funkciók jelenléte állandó problémát jelenthet az újrafelhasználás során. Ne feledjük el, hogy nincs behatásunk az általunk tervezett, kivitelezett szoftver modulok (mint például objektumok, komponensek) további felhasználására. Ezért úgy kell mindent megtervezni, hogy ne okozhasson problémát az újrafelhasználás során.

Van azonban egy, újabban nagyon népszerűvé vált lehetőség az újrafelhasználásra, a kompozíciók létrehozása. Ennek lényege az, hogy a már korábban ismert, többször felhasznált objektumot egy új osztály építőelemeként hasznosítjuk. Ezeket az új osztályokat gyakran burkolóosztályoknak nevezzük, mert a régi objektum köré húzott burkolatként tekintünk rájuk.

Ez nem öröklődés, tehát nem vonatkoznak rá annak megkötései. A már meglévő, alaposan tesztelt és bevált objektum az új osztály privát adattagja lesz, és szolgáltatásait az új osztály funkciói fogják hasznosítani. Az objektum el lett rejtve az osztály funkciói mögött, senki nem tudhatja a kód ismerete nélkül, hogy milyen szerkezeti felépítés van a háttérben.

Ez a felépítés azonban nem korlátozódik az osztályokra és objektumokra. A szolgáltatás alapú architektúrákban ugyanez az elv jelenik meg, amikor egy-egy új funkció, vagy akár szolgáltatás mögött régi, bevált szolgáltatások, mint például korábban már létező webszolgáltatások állnak. Ebben az esetben az új szolgáltatás ugyanúgy a régi szolgáltatásokat elfedő burkolatként jelenik meg, mint az új osztály és a

régi objektumok viszonyában.

A szolgáltatás alapú architektúrák terjedésével, a mikroszolgáltatások népszerűségének növekedésével, várhatóan a kompozíció elvének további terjedése és általánossá válása lesz megfigyelhető a szoftverrendszerekben.

Ellenőrző kérdések:

Mit tekintünk az öröklődés alapelvének?

Mi a kompozíció előnye az öröklődéssel szemben?

Webszolgáltatások esetén hogyan értelmezhető a kompozíció fogalma?

Mit nevezünk burkolóosztálynak?

Tesztelés szempontjából mi az előnye a kompozíciók alkalmazásának?

S.O.L.I.D.-elvek

Úgy gondolták, hogy a szoftverfejlesztésre is ráfér egy kis marketing, ezért a SOLID betűszót alkották meg a fejlesztés minőségét fenntartani hivatott elvek terjesztésére. Ebben a fejezetben megvizsgáljuk, hogy ez a marketing mennyire volt korrekt a problémákra adott válaszokban.

Marketing szempontból elfogadható, hogy a "szilárd, stabil" jelentésű betűszó már csupán a jelentése révén egyfajta pozitív visszajelzés, és feltételezhető, ha valaki ezzel jelzett elveket követ, biztosan jó szoftvert alkot. Itt a jó jelző a szoftver bővíthetőségét, elemeinek újrafelhasználását, egyszerű módosíthatóságát jelzi. Az elvek egy része tervezésnél is hasznos, de vannak olyanok, amelyek többnyire a programozás, kódírás során kerülnek elő. Összességükben azonban az elvek betartásának fontossága megnőtt a mikroszolgáltatásokon alapuló architektúrák terjedésével, mert ezek az elvek azokra is kitűnően alkalmazhatók.

Nézzük meg egyenként az öt elvet.

S mint Single Responsibility, azaz egyetlen felelősség. Értelme az, hogy minden objektum egy és csak egy felelősséggel rendelkezzen, egyfélét csináljon. Az elv nagyon hasonlít a kohézió elvéhez, lényegében annak újrafogalmazása és összekapcsolása egyéb elvekkel. Fontos, hogy már tervezési szinten ajánlatos figyelembe venni. Vizsgáljuk meg ezt az állítást egy közigazgatási szoftverből származó példán.

Személyi adatokat kell a rendszerben tárolnunk, ezt objektumorientált gondolkodás esetén egy Személy osztályban fogjuk összefoglalni. A személynek persze van neve, és van címe is, egész pontosan egy mai magyarországi közigazgatási szoftverben két neve van (a jelenleg használt és a születési neve). Címe is gyakran több van, az állandó lakcím, az ideiglenes lakcím és a levelezési címe. Ezek persze lehetnek azonosak az adatok felvételének pillanatában, de a tárolásuk mindig külön történik. Tervezéskor érdemes

a név adatelemeit egy külön osztályba rendezni, amelynek műveletei - másképpen operációi - végzik majd a név elemek összekapcsolását igény szerint, akár többféle módon. Hasonlóképpen a cím elemeit is kiszervezzük egy külön osztályba. Az előnye ennek az, hogy a Név és Cím osztályokat bármely további alkalmazásban változatlanul felhasználhatjuk, és megvalósult az egyetlen felelősség elve is, ami a szoftvert könnyen módosíthatóvá teszi.

O mint Open/Closed elv, azaz nyitott-zárt elv. Értelme az, hogy rendszerünk legyen nyitott a bővítésre, továbbfejlesztésre, de zárt a már meglévő objektumok módosításaira. Azt javasolja, hogy ne írjuk át a már meglévő kódot, hanem öröklődéssel, kompozíciók, új szolgáltatások létrehozásával vigyünk bele új funkciókat. Kevesebbet kell tesztelni, és a már bevált megoldások változatlanul megmaradnak. A gyakorlatban ezt szolgálhatják az interfészek, amelyek zártak a módosításra (nem adunk hozzá új funkciókat), de könnyen bővíthetők azzal, hogy új interfészt hozunk létre.

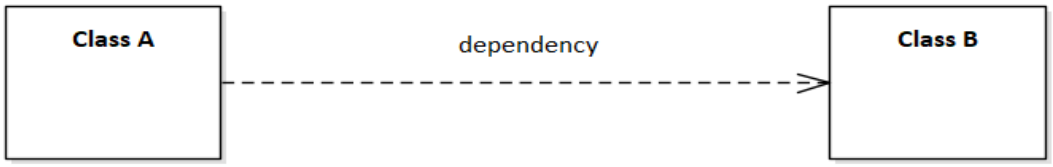
L mint Liskov helyettesítés. Az öröklődés alapelve, hogy ha szükségünk van osztályok örökítésére, a leszármazottak mindig hordozzák az ős funkcionalitását. Másként megfogalmazva azt értjük alatta, hogy legyenek képesek az ős objektumot bármely szituációban helyettesíteni. Ne felejtsük el, hogy az öröklődés egy "olyan, mint" kapcsolat, megszorítások nélkül. Az elv betartása főleg programozás során kerül elő.

I mint Interface Segregation elv, azaz az interfészek szétválasztásának elve. A cél az, hogy csökkentsük a szükséges változtatások gyakoriságát, vagy mellékhatásait azzal, hogy a szoftvert több, független részre bontjuk fel. Ez csak akkor érhető el viszont, ha a funkcionalitást leíró interfészek meghatározásakor minden egyes interfész specifikus klienst vagy feladatot szolgál ki. Senki sem tervez túlbonyolított, komplex interfészt a kezdetekben, de a funkcionalitások hozzáadása során a legegyszerűbbnek tűnik, ha a már meglévő interfészt egészítjük ki újabb és újabb funkcióval. Ez a kiegészítési stratégia viszont hamar átláthatatlanná teszi az interfészt és adott kliens szempontjából sok felesleges funkció kerül bele. Ezt hivatott megoldani az elv értelmében az interfészek átdolgozása a továbbfejlesztések során.

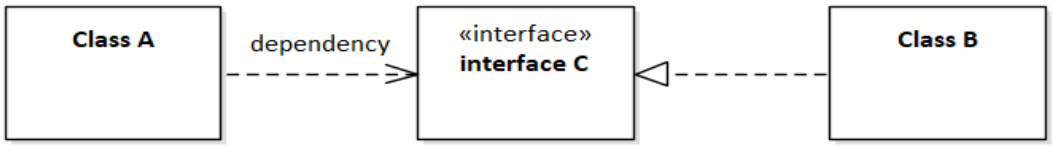
D mint Dependency Inversion elv, azaz a függőség megfordításának elve. Egy szoftver tipikusan magas szintű modulokból - amelyek a komplex üzleti logikát hordozzák - és alacsony szintű modulokból áll, amelyek segédfukciókat biztosítanak. Ideális esetben a magas szintű modulok könnyen újrahasznosíthatók, és nem befolyásolják a működésüket az alacsony szintű modulok.

Ennek elérése csak úgy lehetséges, ha ezeket a modulokat szét tudjuk választani, valamilyen absztrakció bevezetésével. Ez az absztrakció ne függjön a részletek megvalósításától, hanem fordítva, a részletek függjenek az absztrakciótól, azaz a részletek implementálják az absztrakciót. Azt kell elérnünk, hogy mind a magas szintű és az alacsony szintű modulok, egymástól függetlenül, de ugyanattól az absztrakciótól függjenek.

Ez első olvasásra nagyon elvontnak tűnik, de az absztrakció, amiről beszélünk, nem más, mint a jól ismert interfész. Így a különböző szintű modulok nem egymástól kerülnek függésbe, hanem ugyanazon interfészen keresztül (tehát ugyanazt az interfészt valósítják meg), ami viszont könnyű megoldást kínál módosítások esetén, hiszen csupán egy új implementációt kell létrehozni.



Nincs dependency inversion, direkt függőség



Van dependency inversion, közvetett függőség absztrakción keresztül

Két korábbi elv, a nyitott/zárt elv és a Liskov helyettesítés elveinek következetes betartása automatikusan biztosítja ennek az elvnek az érvényesülését is.

Ellenőrző kérdések:

Mit értünk a Single Responsibility elve alatt?

Mit értünk az Open/Closed elv alatt?

Mit értünk a Liskov-helyettesítés elve alatt?

Mit értünk az Interface Segregation elve alatt?

Mit értünk a Dependency Inversion elve alatt?

Referenciák:

Alexander Petkov:
https://www.freecodecamp.org/news/object-oriented-programming-concepts-21bb035f7260/

Law of Demeter: https://en.wikipedia.org/wiki/Law_of_Demeter

Thorben Janssen: https://stackify.com/solid-design-principles/