Aspektus-orientált programozás

LENGYEL LÁSZLÓ, LEVENDOVSZKY TIHAMÉR

BME, Automatizálási és Alkalmazott Informatikai Tanszék
lengyel@aut.bme.hu, tihamer@aut.bme.hu

Reviewed

Kulcsszavak: aspektus-orientált programozás, átszövő vonatkozások

Az aspektus-orientált programozás (AOP) egy szerencsés kiegészítése a ma leginkább elterjedt objektum-orientált paradigmának. Jelen cikkben bemutatjuk az AOP fontosabb koncepcióit, a legelterjedtebb megvalósítás, az AspectJ szemléletét alapul véve. Bevezetést adunk az átszövő vonatkozások problémájába, majd megoldásként sorra vesszük az AOP adta lehetőségeket. Röviden megemlítjük a legnépszerűbb megvalósításokat (AspectJ, HyperJ, kompozíciós szűrők) is.

Bevezetés

Napjaink domináns programozási paradigmája az objektum-orientáltság (OO). Az objektum-orientált programozás (OOP) jól használható megoldást adott az áttekinthető és ilyen módon biztonságosabb programok írásához, valamint a kód-újrafelhasználhatóság követelményére. Ez okozta széleskörű elterjedését, és viszonylagos egyeduralmát. Az objektum-orientáltság mögött az a szemlélet húzódik meg, hogy az elkészítendő program önálló entitások, úgynevezett objektumok összessége, melynek működését ezen objektumok kommunikációja valósítja meg. Ez a módszer a tapasztalatok szerint áttekinthető viszonylag komplex feladatok esetén is [1,2].

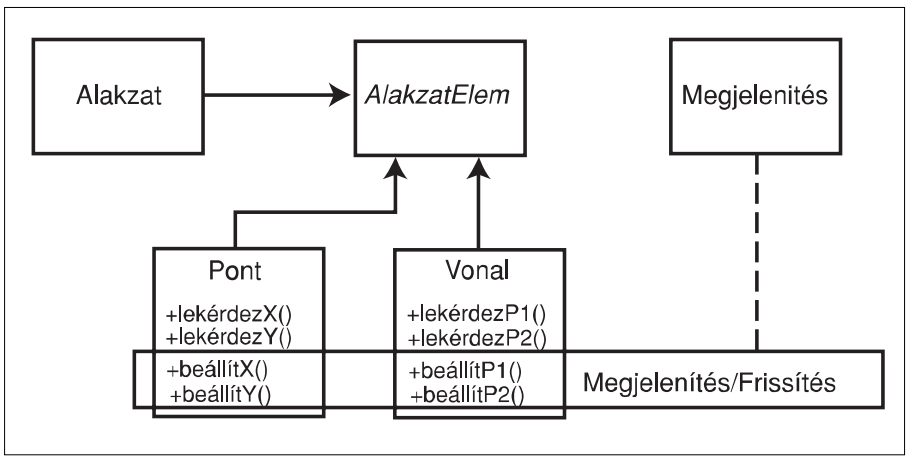
Ha egy komplex feladatot objektumokra dekomponálunk, elsősorban az önálló entitások létrehozására koncentrálunk, valamit arra, hogy az adatokat és a hozzájuk kapcsolódó műveleteket egységbe zárjuk. Így azonban figyelmen kívül kell hagynunk sokkal fontosabb logikai rendező elveket, csoportosítási szempontokat, mint például a perzisztencia, az elosztottság, a hibakövetés, amelyek így a programkódban jellemzően elszórtan jelennek meg, nehézkessé téve a rendszer megértését és így a karbantartását is. Ezeket a szétszóródott és a program különálló egységein áthúzódó, de logikailag egybe tartozó kódrészleteket *átszövő vonatkozásoknak* (croscutting concerns) nevezzük.

Egy példa az átszövő vonatkozásokra a program futásának nyomon követése. Elosztott rendszereknél az alkalmazás gyakran ír naplófájlt, amely az alkalmazás hibás működése esetén segíti a nyomkövetést azáltal, hogy minden függvényhívást és kivételt tartalmaz. A napló fájl írásához minden osztálynak tartalmaznia kell naplózást megvalósító programsorokat, általában szétszórva, holott a naplózást végrehajtó kódrészletek nagyon szoros kapcsolatban állnak egymással: egy funkciót valósítanak meg.

Egy másik lehetséges példához [3] vegyük szemügyre az *1. ábrán* látható egyszerű alakzatszerkesztő UML osztálydiagramját. Az *AlakzatElem* két konkrét leszármazott osztályát láthatjuk: az egyik a *Pont*, a másik, pedig a *Vonal*. Az osztályokra bontás ígéretesnek bizonyul: mindkét osztály jól definiált interfésszel rendelkezik, valamint az adatok és a rajtuk végzett műveletek egységbezárását is sikerült megvalósítanunk. De gondoljuk át a képernyőkezelő szemszögéből a dolgot, aminek értesítést kell kapnia minden elem mozgásáról. Ez megköveteli, hogy minden függvény, amely mozgatást végez, a mozgatás után értesítse a képernyőkezelőt.

Az ábrán a *Megjelenítés/Frissítés* téglalap azokat a függvényeket keretezi be, amelyeknek ezt a feladatot kell megvalósítaniuk. Hasonlóan a *Pont* és *Vonal* téglalapok azokat a függvényeket keretezik be, amelyek a hozzájuk tartozó vonatkozást valósítják meg. Látható, hogy a *Megjelenítés/Frissítés* téglalap nem illik az ábra semelyik másik téglalapjába, hanem átvágja, átszövi azokat.

1. ábra Átszövő vonatkozások



Az OO lehetőségeket kiegészítve, az átszövő vonatkozások problémájára egy megoldást kínál az *aspektus-orientált programozás* (Aspect Oriented Programming, AOP) [4], amely a programot alkotó kódot az alapján osztja részekre, hogy azok milyen szempontból járulnak hozzá a program működéséhez. Aspektus-orientáltan megközelítve a problémát, az egyes szempontokat *aspektusokba* tömörítve külön, egymástól viszonylag függetlenül kódoljuk, majd egy *aspektus-szövő alkalmazás* (aspect weaver) segítségével egyesítjük. Az egybeszövés az adott megközelítéstől függően történhet akár futási időben dinamikusan, akár a fordítás előtt vagy során statikusan.

Objektum-orientált programozást használva az átszövő vonatkozások megvalósítása szétszóródik az egész rendszerben. Az AOP mechanizmusait használva a *Megjelenítés/Frissítés* téglalap által határolt vonatkozás megvalósítása egyetlen aspektusban elvégezhető, emellett tervezési szinten is lehetővé válik az aspektusokban való gondolkodás, és így a modularitás megvalósítása is.

A modularitás lényege, hogy programunk logikailag egy egységet képező részeit egy fizikai egységben adhassuk meg. Általános elv, hogy a modulok belső kohéziója minél erősebb legyen, míg a modulok egymással lazán kapcsolódjanak. Az absztrakció segítségével különböző elemekből ki tudjuk emelni azok közös jegyeit.

Míg a vonatkozások szétválasztása és a modularitás segítségével horizontális módon szervezhetjük a programunkat, addig az absztrakció vertikális jellegű. Egy programozási paradigmát illetve technológiát legfőbb mértékben az határoz meg, milyen típusú absztrakciót használ fel. A gyakori, ismétlődő kódrészletek, illetve kódminták az adott programozási technika absztrakciós képességeinek hiányát jelentik. A redundancia azért kellemetlen, mert egy kisebb változtatás a programtervben azt eredményezheti, hogy számos, egymással kapcsolatban nem álló modulban kell változtatást elvégezni.

Átszövő vonatkozások

A vonatkozások (concerns) elkülönítése azt a képességet jelenti, mellyel azonosítjuk és kiemeljük a szoftver azon részeit, melyek egy adott szándékot, célt valósítanak meg. A vonatkozások elkülönítésének elsődleges célja, hogy a szoftvert kezelhető és könnyebben érthető részekre bontsuk szét. Természetes kérdés, hogy miként végezzük ezt a felbontást. Mely funkciók kerüljenek osztályokba, melyek aspektusokba?

Fontos látni, hogy az átszövés egyéni felbontásokra vonatkozik, ugyanis az átszövő vonatkozásokat nem lehet teljesen elkülöníteni egymástól. Az alapvető tervezési szabály, hogy vegyük az alapvető vonatkozásokat, mint első szintű absztrakciót, valósítsuk meg őket osztályokkal, és majd később elvégezzük a kiterjesztésüket aspektusokkal, ha szükséges lesz. Az alakzatszerkesztő példánál két fontos tervezési vonatkozás van: a grafikai elemek reprezentálása és a grafikai elemek mozgásának a követése.

Az 1. ábrán látható osztályok az első vonatkozást képviselik. Minden grafikai osztály magában foglalja saját belső adatstruktúráját, amelynek kiterjesztésére az aggregáció és az öröklés biztosít lehetőséget. A második vonatkozást, az elemek mozgásának a nyomon követését ugyancsak egy külön osztályként kellene ábrázolni, de az első vonatkozás megakadályozza ezt, mert az egységbezárás miatt a mozgást megvalósító függvények a grafikai osztályok részei. A rendszert a mozgás nyomon követése köré is tervezhetjük, de abban az esetben a grafikai funkcionalitás fogja átszőni a nyomon követés osztályait. Melyik megoldást érdemes alkalmaznunk?

A probléma tisztázható például a domináns felbontás segítségével. A szoftvert – a könyvekhez hasonlóan – folyó szövegként írjuk, ahogy a könyv is fejezetekből és bekezdésekből áll össze, a szoftvernek is vannak moduljai, mint például az osztályok. A modulok, amelyek a domináns felbontást alkotják, egységes vonatkozásokat tartalmaznak és legtöbbször *önállóan futtathatók*. Egy domináns modul nem tartalmazhat olyan vonatkozást, amely számos más modult sző át. Ezek átszövő vonatkozások lesznek.

Jellegzetes átszövő vonatkozások például a szinkronizáció, a monitorozás, a pufferelés, a tranzakciókezelés vagy a környezetfüggő hibakezelés. Az átszövő vonatkozások egész magas szintűek is lehetnek, mint például az adatbiztonság vagy a szolgáltatásminőség aspektusai.

Aspektus-orientált programozás

Az aspektus-orientált programozási paradigma a kilencvenes évek közepén született meg, ma már a programozási nyelvekkel kapcsolatos kutatások egyik nagyon fontos területe, és várhatóan a közeljövőben széleskörűen el fog terjedni.

Joggal mondhatjuk, hogy a Fortran óta minden programozási nyelv rendelkezik azzal a képességgel, hogy alprogramokba különítse el a vonatkozásokat. Az alprogramok máig jól használhatók, és mint ahogy az objektum-orientált programozás nem tud elszakadni a blokk struktúráktól és a strukturált programozástól, ehhez hasonlóan az AOP sem veti el a már meglévő technológiákat. Gyakori, hogy a vonatkozások nem valósíthatók meg egy egyszerű eljáráshívással, ugyanis egy vonatkozás összekeveredik más strukturális elemekkel, kuszasággá válik. Az alprogramok másik hátránya, hogy megköveteli, hogy a hívó komponenst programozók is tudatában legyenek a vonatkozásoknak, tudják, hogyan kell őket beiktatni, használni. Az AOP az alprogramok mellett olyan hívási mechanizmust kínál, amely lehetővé teszi, hogy a hívó komponens fejlesztőinek ne kelljen tudniuk a kiegészítő vonatkozásokról, vagyis az alprogram meghívásáról.

Az aspektus-orientált programozás két legfontosabb alapelve az átszövő vonatkozások elkülönítése és a modularitás [5]. Az AOP egyik felismerése, hogy a moduláris egységek határai ritkán esnek egybe a vonatkozások határvonalaival. A modularizálás bizonyos vonatkozásoknál megoldható, de az átszövő vonatkozásokat megvalósító kód szétszórva található meg a programban, átszőve az egyes moduláris egységeket. Az AOP célkitűzése, hogy ezeket az átszövő vonatkozásokat is önálló moduláris egységekbe lehessen szervezni, a programtermék (kód és/vagy terv) bonyolultságát csökkentse, a karbantarthatóságát, olvashatóságát javítsa, újrafelhasználhatóságát megkönnyítse.

Egy aspektus a megvalósítás egy moduláris egysége, olyan viselkedést foglal magába, amely több osztályra is hatással van. Az AOP segítségével először elkészítjük az alkalmazást egy tetszőleges objektum-orientált nyelven (például C++, Java vagy C#), majd ezt követően külön foglalkozunk az átszövő vonatkozásokkal, amely az aspektusok beépítését jelenti. Végül az aspektusszövőt alkalmazva a kód és az aspektusok kombinációjaként készül el a futtatható alkalmazás, melynek eredményeként az aspektus számos függvény, modul vagy objektum megvalósításának lesz a része, és ezzel növeli mind az újrafelhasználhatóságot, mind a karbantarthatóságot.

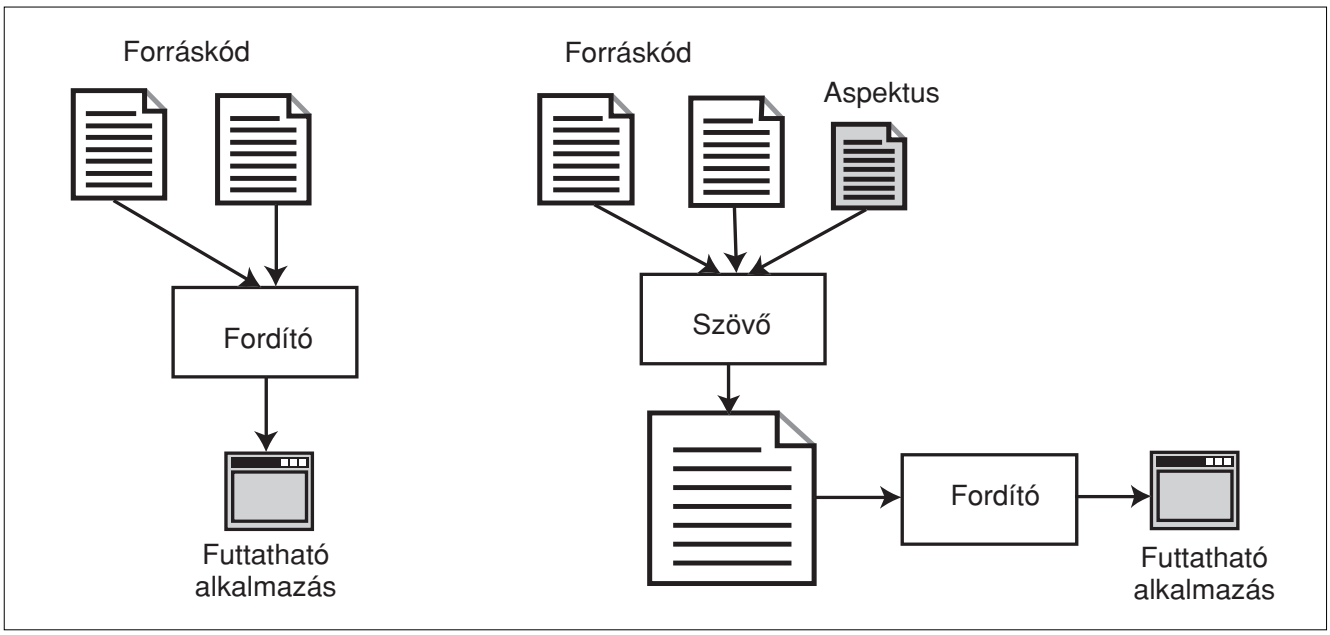
A szövő folyamat a *2. ábrán* látható. Fontos, hogy az eredeti kódnak semmit sem kell tudnia a hozzáadott aspektusról, az aspektusszövő nélkül lefordítva az eredeti alkalmazást kapjuk, a szövőt és az aspektusokat felhasználva pedig az aspektusok funkcionalitásával kiegészített alkalmazást. Ez azt jelenti, hogy az eredeti kódon semmit sem kell változtatni, mindkét esetben ugyanabban a formában használható.

Az AOP kiegészíti, és nem helyettesíti az objektumorientált programozást. Egy más típusú felbontást kínál, az osztályok mellett új modularizációs elemeket vezet be, melynek feladata, hogy az osztályoktól külön, azokból kivonva valósítsa meg az egyes aspektusokat. Az AOP az OOP-re épül, nem tekintjük idejétmúltnak az objektumokat, függvényeket, hanem az aspektusokkal együtt tovább használjuk őket, mindig a legmegfelelőbbet.

Az AOP azáltal nyújt többletet, hogy a függvény jelöli ki a meghívás helyét és nem a hívó, amely nem is tud az egészről. Ha a hívott programrészletnél megadjuk a hívás helyét, akkor ez automatikusan hozzászövődik és a megvalósításban természetesen függvényhívás lesz belőle, de ezt a szövő a programozó beavatkozása nélkül, automatikusan kezeli). Ez azért hasznos, mert, ahogy az már a fentiekben szerepelt, a hívó kód magától a kiterjesztésétől függetlenül is le tud futni. Például egy operációs rendszer a memórialapozáshoz egy úgynevezett piszkos bitet (dirty bit) használ, amely azt jelzi, hogy egy memórialapon történt-e valamilyen változás, ugyanis amennyiben történt, akkor el kell menteni a megváltozott adatokat a háttértárolóra. AOP technikákkal a piszkos bit kezelése szétválasztható a memórialapozástól: a rendszer futtatható piszkos bit kezeléssel, vagy anélkül, valamint a piszkos bit kezelését végző függvények fizikailag egy helyen vannak, és nem a hívás helyétől függően a forráskódban elszórva.

Felmerülhet a kérdés: hogyan adjuk meg, hogy egy adott kódrészlet hol hívódjon meg, és mindezt úgy, hogy a hívó kódrészletben ez egyáltalán ne jelenjen meg, a megoldás a csatlakozási pont. A csatlakozási pontok a programnak azokat a jól meghatározott pontjait jelentik, amelyekben az aspektus interakciója történik a rendszer többi részével. Attól függően, hogy a csatlakozási pontok a program szövegének vagy futásának pontjait jelentik, statikus illetve dinamikus csatlakozási pontokról beszélünk. Statikus kapcsolódási pontok például a naplózás esetén minden publikus függ-

2. ábra Aspektusszövő



vény első utasítása, amely eredménye egy olyan napló fájl lesz, ahol nyomon követhető a függvényhívások sorrendje. A dinamikus kapcsolódási pontok a programfutás olyan eseményeihez kötődnek, mint metódus-hívás, hívás fogadása, futása, továbbá az attribútum lekérdezés, kivétel dobása, osztály inicializáció, objektum inicializáció.

Az aspektusok újrahasználhatósága az AOP egyik alapvető eredménye. Az egyszerű, kisméretű aspektusok készítése még inkább újrafelhasználhatóvá teszi az egyes kódrészleteket. A tapasztalatok felhasználásával, az aspektusok összegyűjtésével aspektuskönyvtár készíthető. Ehhez kapcsolódó kérdés, hogyan bánjunk a sok aspektussal, hogyan készítsük el őket, milyen jelölést használjunk a leírásukra. Ezek és a hasonló kérdések azok, amelyeket a felhasználók és a kutatócsoportok fognak felderíteni az elkövetkező néhány évben.

Az egyik legfontosabb, még jelenleg is nyitott kérdés az aspektusok szemantikai helyessége. A komponens alapú rendszereknél mindig is kérdés volt, hogy miként biztosítsuk az egyes komponensek helyes működését. Az aspektus alapú megközelítés által kínált csatlakozási pont alapú összetétel sokkal gazdagabb mechanizmusokat kínál, mint az interfész vagy üzenet alapú kapcsolatok.

Az egyes aspektusokat mind a specifikáció, mind pedig a komponens teszt szempontjából széleskörűen kell megvizsgálni. Amennyiben egy aspektust újrafelhasználunk, nincs garancia arra, hogy minden helyen, ahol ezután újrafelhasználjuk, az elvárásoknak megfelelően működik. Meg kell találni a módját, hogy leírjuk az újrafelhasználásra szánt aspektusok speciális környezetbeli működését.

Az AOP ismertetése után röviden megemlítjük a három legelterjedtebb AOP megvalósítást.

AspectJ

Az AspectJ (http://www.aspectj.org) a Java nyelv természetes kiterjesztése: minden Java program egyben AspectJ program is. Az AspectJ a következő új programozási konstrukciókat vezeti be a Javába az AOP alapfogalmak számára [6,7]:

- *aspektus* (aspect): Új programozási egység, amely kiemeli az átszövő vonatkozásokat. Az aspektusok tartalmazzák a vágási pontok, a tanácsok és bevezetések definícióit, emellett az osztályokhoz hasonlóan lehetnek adattagjaik és metódusaik is.
- *vágási pont* (pointcut): Dinamikus csatlakozási pontok halmazát jelöli ki egy logikai kifejezés segítségével. Ezeket vágásipont-leíróknak (pointcut descriptor) hívjuk. A vágási pontok paraméterezhetők, a paramétereken keresztül a vágási pont környezetének objektumait tudjuk átadni a tanácsnak.
- *tanács* (advice): A metódushoz hasonló programozási egység. A tanácsok mindig egy adott vágási ponthoz kapcsolódnak. Törzsük tartalmazza azt a viselkedést, melyet az adott vágási pont által leírt csatlakozási pontokban automatikusan végrehajt.
- *bevezetés* (introduction): A bevezetéssel új adattagokat és metódusokat tudunk definiálni az osztályokban. (A csatlakozási pontot itt az osztály jelenti.)

Annak érdekében, hogy futás közben egyértelműen eldönthető legyen, hogy egy belépési pontra kapcsolódó több aspektus elemei milyen sorrendben futhatnak, az aspektusok között és az aspektusokon belül jól definiált precedencia relációk állnak fenn [8].

Az AspectJ-ben [9,10] az osztályok és az aspektusok nem azonos szintű programozási egységek. Míg az osztályok önálló entitásoknak tekinthetők, addig az aspektusok csak azokkal az osztályokkal együtt értelmezhetőek, amelyeknek az átszövő kódját tartalmazzák. Úgy tekinthetjük, hogy az aspektusok az eredeti program kódjában végzett változtatásokat tartalmaznak. Az aspektusok ennek megfelelően önállóan nem fordíthatóak, újrafelhasználásuk csak forrásszinten lehetséges. A jelenlegi AspectJ megvalósításnál az egybeszövés a fordítás során történik, és szükség van az alapprogram forráskódjára is [11].

Hiperterek – HyperJ

A hipertér-technológia (http://www.research.ibm.com/hyperspace/index.htm) megközelítésének alapelvéül a vonatkozások többdimenziójú szétválasztása szolgál. Az elv szerint egy szoftverben egyszerre különböző típusú vonatkozások vannak, ezzel szemben a ma elterjedt nyelvek és módszerek csak egy fajta vonatkozás szerinti dekompozícióra nyújtanak lehetőséget. Ezt a jelenséget nevezzük a domináns dekompozíció egyeduralmának.

A domináns dekompozíció alapja OO nyelveknél az osztály, funkcionális nyelveknél a függvény, szabályalapúaknál a szabály.

A hiperterek lehetővé teszik vonatkozások tetszőleges dimenziójának explicit azonosítását a szoftver életciklusának tetszőleges szakaszában. Ehhez a hipertérmodell a következő fogalmakat használja:

- *hipertér*: A vonatkozások azonosítását szolgálja. A hipertér a szoftvert felépítő egységek egy halmazát jelenti. Ilyen egység lehet például OO környezetben egy osztály, egy metódus vagy egy adattag. Ezeket az egységeket a hipertér egy többdimenziójú mátrixba szervezi. A hipertér képzeletbeli koordinátarendszerében a vonatkozások jellegük szerinti felosztását (a vonatkozások dimenzióit) jelentik, a koordináták pedig a dimenzión belüli konkrét vonatkozásokat. A hipertérben szereplő egységek koordinátái kijelölik, hogy az adott egység mely vonatkozások leírásában játszik szerepet.
- *hipersík*: A vonatkozások egységbe zárását szolgálja. Az egy vonatkozáshoz tartozó egységek egy hipersíkon helyezkednek el. A hipersíkok megadásával foghatjuk össze a vonatkozásokat érintő egységeket

• *hipermodul*: A vonatkozások integrálására szolgál. Egy hipermodul magában foglalja az integrálandó hipersíkok egy halmazát, valamint az integrációs kapcsolatokat, melyek a hipersíkok egymáshoz való viszonyát és az integrálás módját írják le.

A HyperJ a vonatkozások többdimenziójú szétválasztásának Java nyelvű megvalósítása [12]. A HyperJ az integrációt nem forráskódon, hanem a lefordított hipersík csomagokon végzi, így lehetőség van már létező alkalmazások újrafelhasználására újra modularizálással. A csatlakozási pontok statikusak, helyüket a hipermodul specifikációja tartalmazza.

Kompozíciós szűrők

Mivel az objektum-alapú rendszerekben a viselkedést az objektumok közötti üzenetváltás valósítja meg, az objektumok bejövő és kimenő üzeneteinek (tipikusan függvényhívásainak) manipulálásával a viselkedésbeli változtatások rendkívül széles skálája elvégezhető.

A *kompozíciós szűrők* (http://trese.cs.utwente.nl/composition_filters/) modelljében az üzenetek vizsgálatát és manipulációját erre a célra tervezett szűrők végzik. A modell az átszövő vonatkozásokat az objektumok moduláris és független kiegészítéseiként fejezi ki. A modularitás itt abban nyilvánul meg, hogy a szűrők jól definiált interfésszel rendelkeznek, és koncepcionálisan függetlenek az objektum megvalósításától [13,14]. A szűrők függetlenek egymástól, vagyis a specifikációjukban nem szerepel hivatkozás egyéb szűrőkre.

Összegzés

Az AOP egy koncepció, egy olyan elképzelés, amely nem kötődik egyik programozási nyelvhez sem. Valójában, egyszerű és hierarchikus felbontást alkalmazva, a már létező programozási nyelvek elemeit megtartva és felhasználva képes kiküszöbölni a (nem csak objektumorientált) programozási nyelvek hiányosságait.

Az AOP koncepcióit már több különböző nyelvre kidolgozták: C, C++, Java, Perl, Python, Ruby, SmallTalk és C#. (A Java az a nyelv, amely iránt a kutató közösségekben a legnagyobb érdeklődés mutatkozik, a legfejlettebb AOP környezetek ezen a nyelven állnak rendelkezésre.)

A szoftverfejlesztés területén manapság egyre nagyobb szerepet kapnak a szoftver hosszú távú életciklusát érintő kérdések. Ilyen a fejleszthetőség, a karbantarthatóság, a követelmények gyakori változásához való alkalmazkodóképesség vagy az újrafelhasználhatóság. Az AOP ezeket a célokat egy, az OO-nál rugalmasabb, magasabb absztrakciós szintű modell biztosításával segíti.

A téma irodalomjegyzéke igen bőséges. Az érdeklődő olvasóknak további információk végett a CACM AOP-nek szentelt különszámát [15] ajánljuk, illetve az egyes megvalósítások web oldalait.

Irodalom

[1] Czarnecki, K. and Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications. Addison Wesley, Boston, 2000.

[2] Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, 1976.

[3] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, Harold Ossher: Discussing Aspects of AOP, CACM Volume 44, Issue 10 (October 2001)

[4] Aspect-Oriented Software Development. http://www.aosd.net/

[5] Tzilla Elrad, Robert E. Filman and Ataf Bader: Aspect-oriented Programing, CACM Volume 44, Issue 10 (October 2001)

[6] Gregor Kiczales, Erik Hilsdale, Jim Hungunin, Mik Kersten, Jeffrey Palm and William G. Griswold: An Overview of AspectJ. Jorgen Lindskov Knudsen (Ed.): Proceedings of the 15th ECOOP, Budapest 2001, pp.327–353.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold: Getting started with AspectJ, CACM Volume 44, Issue 10 (October 2001)

[8] Kiczales, G., et al.: An overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP). Springer, 2001.

[9] The AspectJ Programming Guide, http://www.aspectj.org

[10] Bill Griswold, Erik Hilsdale, Jim Hugunin, Wes Isberg, Gregor Kiczales, Mik Kersten: Aspect-Oriented Programming with AspectJ, http://www.aspectj.org

[11] The AspectJ Primer; http://www.aspectj.org/doc/primer.

[12] Peri Tarr, Harold Ossher: Hyper/J User and Installation Manual http://www.research.ibm.com/hyperspace http://www.math.klte.hu/~espakm/GOF/hires/Pictures/

[13] Mehmet Aksit, Bedir Tekinerdogan: Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters, 1998.

[14] Mehmet Aksit, Lodewijk Bergmans: Software evolution problems using composition filters. ECOOP 2001, Budapest

[15] Communications of the ACM Volume 44, Issue 10 (October 2001)