Objektum-orientált megvalósítások

Előadásvázlat „Szolgáltatásbiztonságra tervezés” tárgyból

Majzik István
BME Méréstechnika és Információs Rendszerek Tanszék

Tartalomjegyzék:

1 Az alapfeladatok áttekintése

A gyakorlatiban elterjedt megoldások általában olyan hibamódokra készülnek fel, amik az elosztott rendszerben könnyen detektálhatók (pl. ”I am alive” üzenetekkel). Ezek nagy hibafedésű lokális hibadetektálást feltételeznek.

- *Fail silent* működés: Nincs hibás üzenet vagy szolgáltatás (a meghibásodott résztvevő nem küld a hiba után üzeneteket).
- *Omission* (elnyelés): Kimaradhatnak üzenetek (a vett üzenetek mindig hibamentesek)

A hibatűrés módszerei:

- Fizikai hibák ellen: Objektumok többszörözése (replikálása) különböző szervereken (hostokon).
- Szoftver (tervezési) hibák ellen: Az objektum replikákban eltérő tervezés (*design diversity*):
  - Metódus szinten: Eltérő módon megvalósított metódusok
  - Objektum szinten: Objektumok némiképpen eltérő belső állapottal (*data diversity*, amennyiben a hibák adatfüggők, viszont kis változások a kimenetben elfogadhatók)
  - Osztály szinten: Függetlenül tervezett és implementált osztályok (teljesen OO kompatibilis), pl. egy absztrakt osztály eltérő leszármazottai
  - Rendszer szinten: Ld. N-verziós programozás a teljes rendszerre.

Objektum életciklus és a hibatűréshez szükséges többletfeladatok kapcsolata:

Objektum életciklus	Többletfeladatok hibatűréshez
Inicializálás (konstruktor)	Replikák létrehozása
	Replika csoport kezelése
Metódushívás	Kérések sorrendezése
	Többszörös kérések szűrése
Hívott metódus indulása	Kérés ellenőrzés
	Kérés tárolása (üzenettár)
Hívott metódus befejeződése	Objektum állapot mentése
	Szinkronizáció replikákkal
Törlés (destruktor)	Kilépés a replika csoportból
Elsődleges objektum hiba	Helyreállítás
	Újrakonfigurálás

Alapvető szolgáltatások az objektum replika alapú hibatűréshez:

- *Inter-Replica Protocol* (IRP): Állapot-szinkronizáció az elsődleges és a tartalékok között, az állapotmentés és helyreállítás indítása
- *Group Communication Service* (GCS): Csoportkommunikáció a hívók, az elsődleges(ek) és a tartalékok között; üzenet sorrendezés megvalósítása.

Ehhez szükséges alacsonyabb szintű (tipikusan köztesréteg) szolgáltatások:

- Set: objektumok egy halmazának elérése egy referenciával
- Multicast: a címzettek halmaza alapján az üzenet továbbítása a halmaz minden tagjához
- Ordering: az üzenetek sorrendezése
- Reliable messaging: az üzenet eljuttatása minden elérhető címzetthez

- Atomic: a kérések atomi kezelése
- Membership: tagsági kép szolgáltatása az elérhető illetve kiesett objektumokról

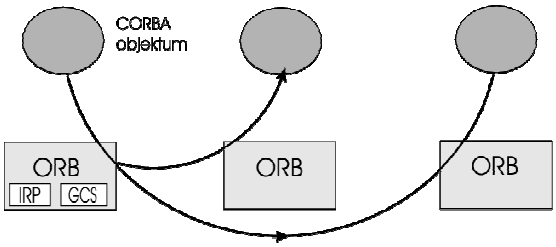
2 Köztesréteg megoldások

Alapfeladat: Az objektum replikáció támogatása köztesréteg segítséggel:

- Inter-Replica Protocol (IRP): Állapot-szinkronizáció az elsődleges és a tartalékok között, az állapotmentés és helyreállítás indítása.
- Group Communication Service (GCS): Csoportkommunikáció a hívók, az elsődleges(ek) és a tartalékok között; üzenet sorrendezés megvalósítása.

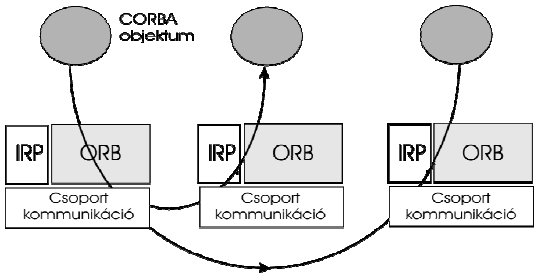
Példaként a CORBA ORB (Object Request Broker) példáján mutatjuk be a köztesrétegen alapuló hibatűrő architektúrák kialakításának lehetőségeit:

- **Integráció** (*integration*): az ORB módosítása a replika kezelés érdekében (pl. Orbix+Isis). Csoportkommunikáció és hibatűrési stratégiák az ORB-be ágyazva.



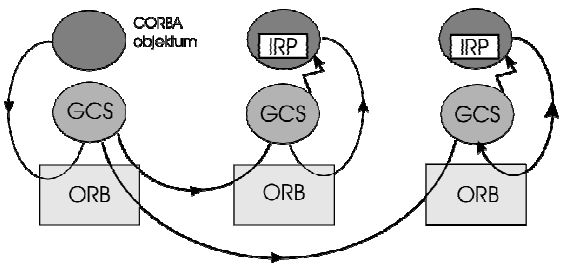
Integráció a köztesrétegbe

- **Beavatkozás** (*interception*): a CORBA üzenetek eltérítése (pl. Eternal). Csoportkezelő köztesréteg veszi át az üzeneteket és kezeli a replikákat.



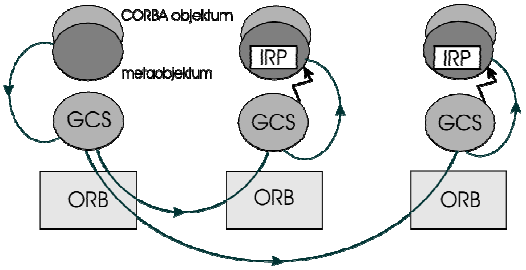
Beavatkozás: Az üzenetek eltérítése

- **Szolgáltatás** (*service*): CORBA szolgáltatás kialakítása a replikáció támogatására, ami a hívó alkalmazás szintjéről érhető el (pl. OpenDreams).
  A CORBA üzeneteket egy csoportkommunikáció szolgáltatáshoz (GCS) kell irányítani, ennek feladata a replika kezelés.



Szolgáltatás a csoportkommunikáció megvalósításához

- **Reflektivitás** (*reflection*): a hibatűréshez szükséges specifikus funkciók megvalósítása külön modulba kiemelve, ez reflektív programozással (pl. Open C++ metaobjektumok segítségével) vagy aspektus-orientált programozással (pl. AspectJ, ld. később) illeszthető a hívó alkalmazáshoz.



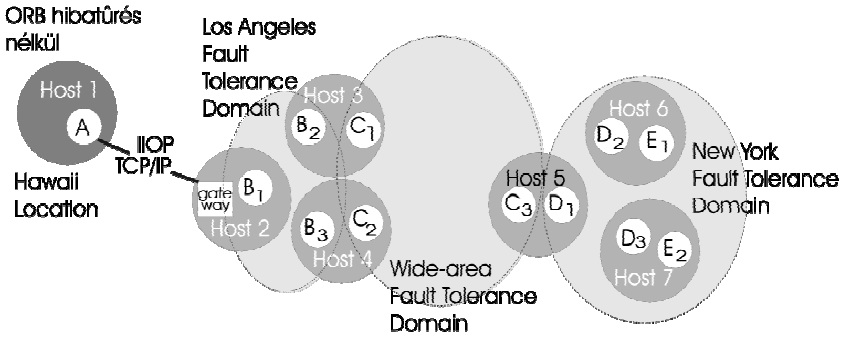
Reflektivitáson alapuló megvalósítás

3 Egy megvalósítási példa hibatűrő OO rendszerek kialakítására

Mintapéldaként a hibatűrő CORBA (FT-CORBA) szabványos architektúráját tekintjük át.

3.1 Alapelvek

- Hibamodell: objektum összeomlás (crash);
  Nincs kezelve: kommunikációs hálózat particionálása, nem fail-silent működés (hibás válaszok, bizánci hibák), korrelált hibák
- Hibatűrés: Ne legyen egyszeres hibapont (single point of failure); ennek érdekében alkalmazott objektum redundancia:
  - „térbeli" redundancia: objektum *replikáció* (host elhelyezkedéstől független);
  - „időbeli" redundancia: *többszörös hívások* a kliens által (ugyanahhoz a szerver objektumhoz, *at most once* szemantikával)
- Az objektum replikák egy *objektumcsoportot* képeznek; ezek menedzselhetők (pl. passzív vagy aktív replikáció)
  - Referencia: IOGR (Interoperable Object Group Reference), ezt teszi közzé a replikált aktív objektum, a továbbiakban a normál objektumhoz hasonlóan működik (kérések fogadása és kiszolgálása ez alapján)
  - Rejtett belső koordináció: replika kezelés (aktív, passzív) illetve hibakezelés
- Hibatűrő tartomány (Fault Tolerance Domain, FTD): több host, több objektumcsoport menedzselése
  - Egy FTD-n belül minden objektumcsoportot egy közös replikáció menedzser (Replication Manager) kezel
  - A hibatűrés tulajdonságai objektumcsoportra vagy FTD-re is állíthatók, ezek dinamikusan változhatnak



Az FT-CORBA hibatűrő tartomány

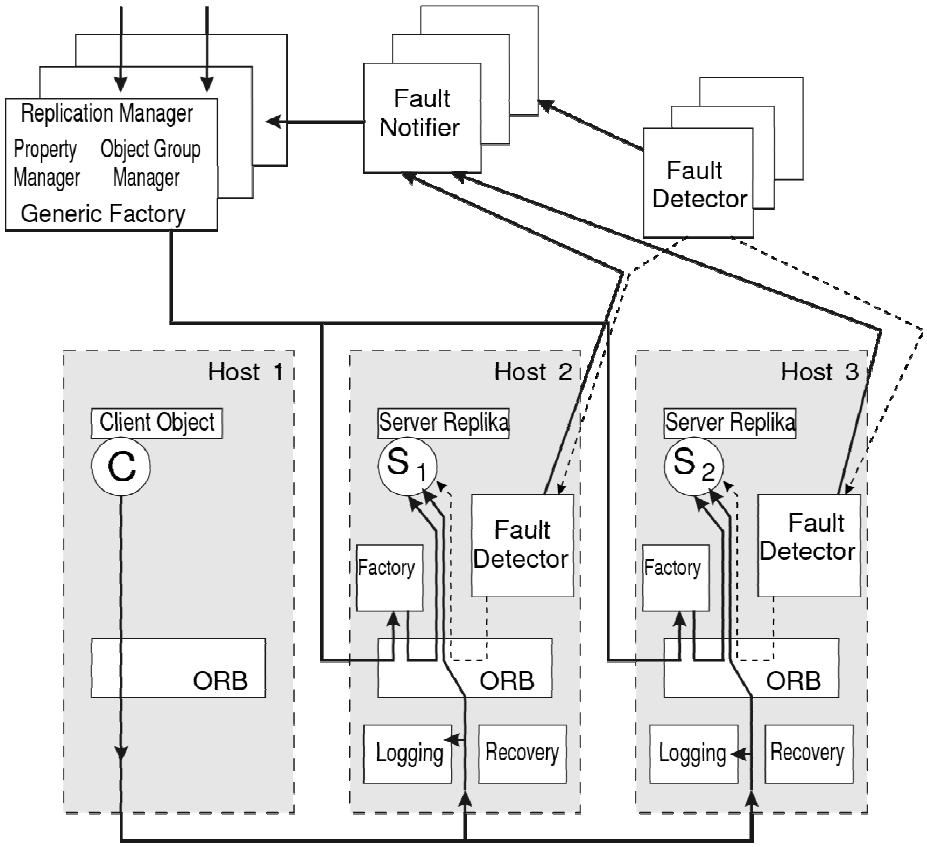
3.2 Egy-egy hoston megvalósított objektumok

- *Factory* objektum minden hoston (nem többszörözött): objektumok létrehozására
- *Fault Detector* objektumok minden hoston (nem többszörözött): lokális hibadetektálás; ennek érdekében az alkalmazás objektumok meg kell valósítsák a PullMonitorable interfészt, ami tartalmazza az is_alive() metódust; ezt hívja a Fault Detector
- ORB-hez csatolt *mechanizmusok* az ORB és az OS között: Logging, Recovery és Message Handling Mechanism
    - *Logging Mechanism*: minden üzenetet automatikusan tárolni tud; periodikusan hívja az alkalmazás objektum get_state() metódusát (ez a Checkpointable interfész része, amit az objektumnak implementálnia kell), így az állapotot is menteni tudja
    - *Recovery Mechanism*: a tárolt üzeneteket vissza tudja játszani; helyreállításkor az objektum set_state() metódusának hívásával tölti vissza az elmentett állapotot (ez az Updateable interfész része, amit az objektumnak implementálnia kell)
    - *Message Handling Mechanism*: többszörös kérések és válaszok kiszűrése, csoportkommunikáció támogatása

3.3 A hibatűrő infrastruktúra

A hibatűrő infrastruktúra (Fault Tolerance Infrastructure) a hibatűréshez szükséges „szolgáltató” objektumokat tartalmazza (ezek is replikálhatók) minden FTD-ben:

- Replication Manager,
- Fault Notifier,
- Fault Detector (globális is a lokálisak mellett),
- Fault Analyzer (opcionális).



A hibatűrő CORBA architektúra

3.3.1 Replication Manager: Az objektumcsoport menedzselése

A Replication Manager jellegzetes szolgáltatásai az alábbi interfészeken érhetők el:

PropertyManager interface: az objektumcsoport hibatűrő tulajdonságainak beállítása

- *MembershipStyle*: objektumcsoport tagság kezelése
    - Infrastruktúra vezérelt: a Replication Manager hozza létre az objektumcsoportot, fenntartja a minimális számú replikát
    - Alkalmazásvezérelt: az alkalmazás hozza létre a replikákat, kéri a Redundancy Managert, hogy adja hozzá ezeket az objektumcsoporthoz; a minimális számú replika fenntartását az alkalmazás végzi
- *ConsistencyStyle*: replika objektumok állapotának kezelése
    - Infrastruktúra által vezérelt: garantált a szigorú replika konzisztencia (az objektumcsoport viselkedése megfelel egy egyszerű objektum viselkedésének, hiba esetén is: aktív replikáció esetén minden hívás után konzisztens állapotba kell kerüljenek a replika objektumok; passzív replikáció esetén pedig állapot-transzfer után) – megköveteli a Checkpointable illetve Updateable interfész implementációját az alkalmazás objektumban (állapotmentés, üzenettárolás és helyreállítás)
    - Alkalmazásvezérelt: nem garantált a szigorú replika konzisztencia, alkalmazásfüggő mentés és helyreállítás valósulhat meg
- *ReplicationStyle*: a replikáció típusának beállítása
    - STATELESS: a szerver objektum működését a hívások sorrendje nem befolyásolja;

    - COLD_PASSIVE: csak az aktív (elsődleges) objektum hajtja végre a kéréseket; ennek állapotát és az üzeneteket periodikusan menteni kell; hiba esetén egy tartalék veszi át az aktív szerepét; ennek állapota a mentés és az üzenet naplózása alapján állítható helyre;
    - WARM_PASSIVE: csak az aktív (elsődleges) objektum hajtja végre a kéréseket; ennek állapotát és az üzeneteket periodikusan menteni kell; a tartalék objektum(ok) rendszeresen átveszi(k) az aktív objektum állapotát;
    - ACTIVE: valamennyi objektum (azonos sorrendben) végrehajtja a kéréseket; a többszörözött válaszokat és kéréseket a Message Handling Mechanism szűri ki; ha egy objektum meghibásodik, a tartalékok még működnek;
    - ACTIVE_WITH_VOTING (kiterjesztés): az aktívan többszörözött objektumok szavaznak a kérésekről/válaszokról; csak a többség egyetértése esetén van kérés/válasz (adathibák esetén is alkalmazható így a hibatűrés)
- *CheckpointInterval*: az objektum állapotmentés periódusideje
- *FaultMonitoringStyle*: hibamonitorozás (detektálás) módja
    - PULL: lekérdezés, a Fault Detektor hívja az objektumot, hogy ellenőrizze az elérhetőséget (az objektum implementálja a PullMonitorable interfészt)
    - PUSH (kiterjesztés): az alkalmazás objektum periodikusan jelenti a Fault Detektornak, hogy elérhető (alkalmazásfüggő megvalósítás lehet)
- *FaultMonitoringGranularity*: a hibamonitorozás felbontása
    - MEMB: az objektumcsoport minden tagját monitorozni kell
    - LOC: hostonként csak egy tagját kell monitorozni az objektumcsoportnak
    - LOC_AND_TYPE: egy típus esetén hostonként csak egy objektumot kell monitorozni
- *FaultMonitoringInterval*: az objektum monitorozás periódusideje
- *Factories*: a Factory objektumok adatai
- *InitialNumberReplicas*: a replika objektumok kezdeti száma (létrehozáskor)
- *MinimumNumberReplicas*: a replika objektumok minimális száma (működés közben); ha ez alá csökken a számuk és nem pótolhatók, akkor fel kell számolni az objektumcsoportot

GenericFactory interface: Objektumcsoport létrehozása

- az alkalmazás create_object() üzenettel kéri egy objektumcsoport létrehozását (mintha csak egy objektum létrehozását kérné, de itt a Factory helyett a ReplicationManager-hez megy a kérés);
- ennek hatására a ReplicationManager create_object() üzenetet küld a különböző hostok Factory objektumainak, amelyek létrehozzák a replika objektumokat.

ObjectGroupManager interface: Objektumcsoport tagság

- alkalmazásvezérelt MembershipStyle esetben az alkalmazás menedzselheti az objektumcsoportot a create_member(), add_member(), remove_member() üzenetekkel

3.3.2 Fault Detector: globális hibadetektálás

- A lokális Fault Detector az adott hoston monitorozza az objektumokat,
- A globális Fault Detector a lokális Fault Detectorokat monitorozza (így egy-egy host kiesése detektálható)

3.3.3 Fault Notifier: hibajelzések kezelése

- Ennek jelentik a hibákat a lokális és globális Fault Detector objektumok, valamint egyéb hibadetektáló objektumok
- A hibát a Replication Manager felé terjeszti, valamint minden további objektum felé is, amely a hibajelzésre regisztrálta magát
- Hibaanalízis (és -előrejelzés) lehetséges alkalmazás-specifikus Fault Analyzer (hibaanalízis) objektumokkal, amiket a Fault Notifierhez regisztrálni kell

3.4 Az alkalmazott mechanizmusok összefoglalása

- Objektumcsoport létrehozás: alkalmazás hívja a ReplicationManager GenericFactory interfész create_object() metódusát, ez a hostok lokális Factory objektumának create_object() metódusát hívja a beállított paraméterek (ld. PropertyManager interface) szerint
- Objektumcsoport hívása: a Redundancy Manager által visszaadott csoportreferencia alapján hívható az objektumcsoport, mintha egy egyszerű objektum volna
- Hibadetektálás: a lokális Fault Detector monitorozza az objektumokat azok is_alive() metódusának hívásával, ezeket a globális Fault Detectorok monitorozzák (szintén is_alive() metódushívással)
- Hibajelzés: a Fault Detector objektumok a Fault Notifier felé jelzik a hibát; a Fault Notifier-hez speciális hibaanalízis objektum is csatlakozhat.
- Hibakezelés: Redundancy Manager által
  - a ReplicationStyle függvényében újrakonfigurálás,
  - a ConsistencyStyle függvényében helyreállítás.
- Újrakonfigurálás:
  - Aktív replikáció esetén a hibás objektum helyreállítása vagy pótlása szükséges;
  - Passzív replikáció esetén csak egy aktív objektum lehet;
    ha a Fault Detector ennek hibáját jelzi, akkor a Redundancy Manager ezt újraindítja, ha ez sikertelen, akkor egy másik objektumot jelöl ki aktívnak a tartalékok közül
- Helyreállítás:
  - Ha alkalmazásvezérelt a ConsistencyStyle, akkor az új aktív objektum feladata a saját állapot helyreállítása
  - Ha infrastruktúravezérelt a ConsistencyStyle, akkor a helyreállítás automatikus: az alkalmazás objektumokban felhasználható a Logging Mechanism (üzenetek tárolása, állapotlekérdezés és -tárolás a Checkpointable interfész get_state() metódusa segítségével), és Recovery Mechanism (üzenetek újrajátszása, állapot betöltése a set_state() metódus segítségével).

4 Aspektus-orientált megvalósítás

4.1 Bevezető

Szoftverfejlesztés néhány általános problémája:

- *Szempontok* szétválasztása (separation of *concerns*):
  - produkciós: pl. üzleti logika, perzisztencia, biztonság
  - fejlesztési: pl. tesztelhetőség, adaptálhatóság, naplózás
  - hibakezelés, QoS: pl. hibadetektálás, redundanciakezelés, időbeliség
- Alapmegfigyelés: Szempontok és moduláris egységek határai nem esnek egybe
  - code tangling: egy modul több szempontnak kíván eleget tenni
  - code scattering: egy szempont sok modulban valósul meg
- Ebből eredő hatékonyságromlás:
  - követelmények követhetősége rossz
  - újrafelhasználhatóság korlátozott az átszövő megvalósítás miatt
  - nehezebb a továbbfejlesztés
  - meglévő technikák csak részben oldják meg (ld. Visitor, Template tervezési minta)
- Célkitűzés: Adott szempontot megvalósító részek elkülönítése, kiemelése.

Megoldások:

- Statikus osztálystruktúra + öröklés: redundancia sémák megvalósítására használható (nem igazán átszövő)
- Reflektív architektúrák: alapszintű objektumok kiterjesztése metaszintű objektumokkal
- *Aspektus–orientált programozás*: nyelvi támogatás a problématér eltérő szempontú elemeinek moduláris egységekbe (aspektus konstrukciókba) szervezésére; majd ezek egybeszerkesztése egy aspektus-szövő (aspect weaver) segítségével.

4.2 Az aspektus-orientált programozás

Az aspektus-orientált programozás (aspect oriented programming, AOP) tervezésének lépései:

- aspektusok dekompozíciója (átszövő aspektusok azonosítása)
- aspektusok implementációja (elkülönítve lehetséges, külön nyelven vagy nyelveken)
- aspektusok összeállítása (AOP nyelvi szabályok alapján)

AOP elemei:

- alap (üzleti logika): komponens nyelv $\rightarrow$ komponens program
- járulékos szempontok: aspektus nyelv $\rightarrow$ aspektus program
- aspektus-szövő (aspect weaver): alap és aspektus programok $\rightarrow$ teljes program (fordítható); futásidejű (run-time) vagy fordítási idejű (compile-time) lehet

AOP nyelvi elemek szerepe:

- mit kell tenni a beavatkozáshoz: *tanács* vagy *beavatkozás* (advice)
- hol kell beavatkozni: *vágási pont* vagy *eseményleíró* kifejezés (pointcut)
- hol történik a tényleges beavatkozás: *kapcsolódási pont* (join point):
  - egy-egy eseményleíró több kapcsolódási pontot is meghatározhat
  - a kapcsolódási pontot nem kell felkészíteni a beavatkozáshoz (meglévő kódhoz illeszthető).

4.3 Az AspectJ megvalósítás: A Java kiterjesztése

4.3.1 Kapcsolódási pont (joint point)

Aspektus interakciójának a helye

- statikus: programszöveghez kapcsolódik
- dinamikus: programfutáshoz kapcsolódik
    - metódushívás és végrehajtás
    - konstruktor hívás és végrehajtás
    - attribútum írás/olvasás
    - kivételkezelés

4.3.2 Vágási pont, vagy eseményleíró kifejezés (pointcut)

Kapcsolódási pontok kijelölése és kontextus összegyűjtése

- Kapcsolódási pontok kijelölése: dinamikus kapcsolódási pontok halmazai, ezekből logikai műveletekkel képzett halmazok;
    - név alapú (name-based): explicit felsorolás
    - tulajdonság alapú (property-based): wildcards: metódusnév, paraméterek, visszatérési érték stb.
- Kontextus kijelölése: paraméterek (this, target, args)
- Absztrakt vágási pont: leszármazott aspektus definiálja
- Konkrét kulcsszavak a kapcsolódási pont megadására:
    - call(method), execution(method),
    - get(field), set(field),
    - handler(exception),
    - staticinitialization(class),
    - within(class), withincode(method),
    - cflow(pointcut), cflowbelow(pointcut),
    - this(object), target(object), args(object)
    - if(BooleanExpression)
    - ||, &&, és ! operátorok vágási pontokon
    - kontextus megadás: target(), this(), és args()
    - reflektív tulajdonság: speciális objektum, thisJoinPoint, tartalmaz információt a kapcsolódási pontról

4.3.3 Tanács, vagy beavatkozás (advice)

Eseményleíró kifejezéshez rendelt, paraméterezhető, esemény előtt, után, vagy közben végrehajtott metódus-szerű műveletek:

- before() advice: a kapcsolódási pont előtt fut le
- after() advice: a kapcsolódási pont után fut le, normál és kivételkezelés esetén is
- around() advice: a kapcsolódási pont helyett fut le
- A vágási pont és tanács együtt az *aspect weaver* egy szabályát határozza meg

Bevezetés (introduction): Új adattagok és metódusok definiálása az osztályokban (a csatlakozási pontot itt az osztály jelenti)

- osztály metódusának és attribútumának bevezetése

- osztály öröklődési hierarchiájának megváltoztatása
- kivételkezelés megváltoztatása

4.3.4 Aspektus (aspect)

Moduláris egység (osztályhoz hasonlóan: metódusok, konstruktor, inicializáló, eseményleíró és tanács alkotja);

- Fenti elemek valamint bármely osztályon belüli tag tetszőleges kombinációja
- Aspektusok kiterjeszthetik egymást, interfészeket valósíthatnak meg
- Aspektusok asszociációi (szeparált aspektus példány az objektumban, a cél objektumban, a vezérlési folyamban, vagy a vezérlési folyamban a vágási pontot kivéve)
- Egy-egy belépési ponthoz kapcsolódó aspektusok között precedencia relációk állhatnak fenn
- Aspektusok között öröklődési relációk állhatnak fenn
- Aspektusok csak azokkal az osztályokkal együtt értelmezhetőek, amelyeknek az átszövő kódját tartalmazzák (önállóan nem fordíthatók); mintegy változtatásokként értelmezhetők
- Specialitások:
    - wildcard használata az eseményleíró kifejezésben
    - absztrakt aspektusok: absztrakt eseményleíró deklarációk helyezhetők el, melyek definícióját (az adott osztályhoz igazítva) az aspektus egy leszármazottjában adjuk meg

4.3.5 Példák

- Üzenettárolás és állapot-szinkronizáció aspektus-orientált megvalósítása
- Aktív redundáns szerverek menedzselése az aspektus programban
    - Az elsődleges szerverhez érkező hívások eljuttatása a tartalékokhoz
    - Kérések sorrendezése
    - Állapot-szinkronizáció a hívások után
    - Hibadetektálás (monitorozás)
    - Állapotmentés és helyreállítás támogatása