JavaEE – konténer szolgáltatások

Transactions, security, timer service, interceptors + AOP: AspectJ

Simon Károly
simon.karoly@codespring.ro

JavaEE: Tranzakciók

Simon Károly
simon.karoly@codespring.ro

ACID

- Munka-egység (unit-of-work) – a tranzakción belüli műveletek (task-ok).
- Atomic: ha valamelyik művelet (task) nem sikeres, a teljes munka-egység megszakításra kerül (az adatok visszaállításra kerülnek) (commit vagy rollback).
- Consistent: az adatok integritása (pl. átutalás: ugyanannyi összeg kerül levonásra az egyik számláról, amennyi a másikhoz hozzá lesz adva).
- Isolated: nincs interferencia más folyamatokkal/tranzakciókkal (a tranzakció által használt adatokat nem befolyásolhatják a rendszernek más részei a munka-egység befejezéséig).
- Durable: a tranzakción belüli minden adatváltozást tárolni kell (a változások nem veszhetnek el pl. rendszer-összeomlásnál).
- Példák:
  - Pénzügyi szoftverek, szolgáltatások (pl. ATM)
  - On-line rendelések
  - Egészségügyi szoftverek (pl. gyógyszeradagolás)
  - Stb.

Declarative Transaction Management

- Tranzakció hatóköre (Transaction Scope): a tranzakcióban résztvevő menedzselt erőforrások összessége (EJB-k, entitások).
- EJB: tasks → methods, unit-of-work → a tranzakción belül meghívott EJB metódusok, transaction scope → a unit-of-work-ban résztvevő EJB-k
- Egy tranzakció akkor továbbítódik egy EJB-hez, amikor annak egy metódusa meghívásra kerül, és így az EJB bekerül a tranzakció hatókörébe. A tranzakció az EntityManager perzisztencia kontextusához szintén továbbításra kerül.
- EJB tranzakció attribútumok használata (javax.ejb.TransactionAttribute):
  - EJB szintjén vagy metódusok szintjén.
  - Annotációval vagy XML leíróállománnyal.
  - NotSupported, Supports, Required, RequiresNew, Mandatory, Never.
  - Amennyiben nincs másképpen meghatározva az alapértelmezett érték Required (a legtöbb esetben az EJB metódusok tranzakciót igényelnek, főleg, ha együttműködnek az Enity Manager-el).

Transaction Attributes

- NotSupported: egy ilyen metódus meghívása a tranzakció felfüggesztését eredményezi (a metódus visszatéréséig). A tranzakció hatóköre nem kerül továbbításra ehhez az EJB-hez, illetve azokhoz, amelyek ebből lesznek meghívva. A visszatérés után az eredeti tranzakció folytatódik.
- Supports: ha tranzakcióból hívják meg, az EJB hozzáadódik az illető tranzakció hatóköréhez (az általa meghívott EJB-k szintén). Nem kötelező, hogy az EJB része legyen egy tranzakció hatókörének, olyan EJB-kkel (kliensekkel) is együttműködhet, amelyek nem részei tranzakció hatókörének.
- Required (alapértelmezett): a metódust kötelező egy tranzakció hatókörén belül meghívni. Ha a meghívó része egy tranzakció hatókörének, akkor az EJB ennek a hatókörnek válik részévé. Ha a hívó nem része tranzakciónak, akkor az EJB saját tranzakciót indít, létrehozza a saját tranzakció hatókörét (amelynek az általa hívott EJB-k is részei lesznek). Az új tranzakció hatóköre addig érvényes, ameddig a metódushívás vissza nem tér.
- RequiresNew: függetlenül attól, hogy a hívó fél része-e egy tranzakciónak vagy sem, a metódushívás mindig új tranzakció indítását eredményezi. Ha a hívó fél már része volt egy tranzakciónak, az a tranzakció felfüggesztésre kerül, mindaddig amíg a metódus vissza nem tér (ameddig az újonnan létrehozott hatókör érvényes).

Transaction Attributes

- Mandatory: az EJB-nek mindig a hívó fél tranzakció hatókörébe kell kerülnie. A metódust nem hívhatja olyan kliens, amelyik nem része tranzakciónak. Az ilyen esetekben EJBTransactionRequiredException típusú kivételt kapunk.
- Never: a metódus nem hívható meg egy tranzakció hatókörén belül. Ha a hívó fél része egy tranzakciónak EJBException típusú kivételt kapunk.
- Megjegyzések:
  - Az EntityManagerekhez történő hozzáférés tranzakció hatókörön belül javasolt. A Required, RequiresNew, illetve Mandatory attribútumokat alkalmazzuk az adathozzáférésért felelős EJB-k esetében.
  - A MDB-k esetében értelemszerűen csak a NotSupported és Required attribútumok alkalmazhatóak (mivel a többi attribútumnak mindig a hívó fél kontextusában van jelentése).
  - EJB Endpoint-ok (webszolgáltatások) esetében a Mandatory attribútum nem használható (a jelen specifikáció szerint az endpoint-ok nem továbbítják a kliens tranzakció hatókörét).
- Példa: BankBean
  - ```
    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public BigDecimal getBalance(long accountId) throws IllegalArgumentException
    ```
  - ```
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void transfer(long accountIdFrom, long accountIdTo, BigDecimal amount)
      throws IllegalArgumentException, InsufficientbalanceException
    ```

Tranzakciók és perzisztencia

- Szabályok:
  - Amikor egy transaction-scoped entity manager kerül meghívásra egy tranzakció hatókörén kívülről, akkor létrehozott perzisztencia környezete (persistence context) a metódushívás visszatértéig lesz érvényes (a menedzselt objektumok ezután lecsatolásra kerülnek).
  - Amikor egy transaction-scoped entity manager egy tranzakció hatókörén belül kerül meghívásra, akkor, amennyiben még nem létezik, létre fog jönni egy perzisztencia környezet, amely az illető tranzakcióhoz lesz hozzárendelve. Ha már létezik perzisztencia környezet, amely az illető tranzakcióhoz hozzá van rendelve, az lesz használva. A perzisztencia környezet továbbítódik a tranzakción belüli entity manager hívások között.
  - Ha egy EJB egy tranzakció hatókörön belüli perzisztencia kontextust használ, és meghív egy állapottal rendelkező EJB-t, amely kiterjesztett perzisztencia környezetet használ, hiba keletkezik.
  - Ha egy kiterjesztett perzisztencia környezetet alkalmazó állapottal rendelkező session bean meghív egy másik EJB-t, amely tranzakció hatókörű kontextust használ (injection), a kiterjesztett hatókör továbbítódik.
  - Ha egy EJB meghív egy másik EJB-t, amelynek különbözik a tranzakció-hatóköre, a perzisztencia kontextus (akár kiterjesztett, akár nem) nem kerül továbbításra.
  - Ha egy állapottal rendelkező, kiterjesztett perzisztencia kontextust alkalmazó session bean meghív egy másik (nem injektált) kiterjesztett kontextust alkalmazó beant, hibát kapunk. Ha egy statefull session bean-t egy másikba helyezünk az injection mechanizmus segítségével, a két bean megosztja a kiterjesztett perzisztencia környezetet. Ha manuálisan hozunk létre statefull session-t, a perzisztencia környezetek nem kerülnek megosztásra.

Isolation and DB Locking

- Dirty reads: tranzakció egy másik tranzakció által végrehajtott nem véglegesített (commit) változásokat olvas.
- Repeatable reads: tranzakción belüli többszöri kiolvasásnak ugyanazt az eredményt kell adnia. Megoldás lehet az adat zárolása, vagy snapshot alkalmazása. Non-repeatable reads: az adat változhat a tranzakción belüli két egymás utáni kiolvasás között.
- Phantom reads: az insert művelet előtt indított tranzakciók látják az indítás után hozzáadott adatbázis bejegyzéseket (a query-k a más tranzakciók által utólag beírt bejegyzésekre is érvényesek).

Database Locks

- Read locks: megakadályozza, hogy más tranzakciók módosítsák a tranzakció alatt beolvasott adatokat a tranzakció befejezése előtt (a nonrepeatable reads kivédése). Más tranzakciók olvashatják az adatokat, de nem módosíthatják azokat. Az aktuális tranzakció sem eszközölhet változtatásokat. Az, hogy a lock csak egy bejegyzésre, több bejegyzésre, vagy az egész táblára érvényes a használt adatbázisrendszertől függ.
- Write locks: megakadályozza, hogy más tranzakciók változtassák az adatokat, de megengedi a "dirty read" helyzet előfordulását mind más tranzakciók számára, mind az aktuális tranzakció esetében (a saját nem véglegesített változtatásait olvashatja).
- Exclusive write locks: megakadályozza, hogy más tranzakciók írják, vagy olvassák az adatokat a tranzakció befejezéséig. Nem engedi a "dirty read" helyzet előfordulását más tranzakciók esetében (néhány adatbázisrendszer az aktuális tranzakció esetében is megakadályozza az olvasást).
- Snapshots: a tranzakció elkezdésekor az adatokról egy "pillanatkép" készül. Néhány rendszer esetében minden tranzakció részére saját snapshot jön létre. A snapshot-ok segítségével kiküszöbölhetőek a dirty reads, nonrepeatable reads és phantom reads helyzetek, de bizonyos esetekben problémát jelenthet, hogy így az adatoknak nem egy valós idejű változatával dolgozunk.

Elkülönítési szintek

- Transaction Isolation Levels:
  - Read Uncommited: a tranzakció olvashat nem véglegesített adatokat (más tranzakciók által commit hívása előtti módosításokat). Mind a dirty reads, mind a nonrepeatable reads, mind a phantom reads helyzetek előfordulhatnak.
  - Read Committed: a tranzakció nem olvashat nem véglegesített adatokat (commit előtti módosítások). A dirty reads helyzeteket kivédi, de előfordulhatnak nonrepeatable és phantom reads helyzetek.
  - Repeatable Read: a tranzakció nem módosíthat más tranzakción belül beolvasott adatokat. A dirty reads és nonrepeatable reads helyzetek kivédve, phantom read előfordulhat.
  - Serializable: a tranzakciónak kizárólagos olvasási és írási jog adott. Más tranzakciók nem olvashatnak vagy módosíthatnak a tranzakción belül használt adatokat. Mind a dirty, mind a nonrepeatable mind a phantom read helyzet kivédett, a legmegkötőbb elkülönítési szint.
- A szintek a JDBC-n belül definiáltakkal azonosak, a konkrét megvalósítás függ az adatbázisrendszertől, illetve az általa alkalmazott zárolási (locking) mechanizmustól.
- Cél: hatékonyság – konzisztencia egyensúly megteremtése (pl. egy nagy osztott rendszer esetében a Serializable elkülönítési szint gyakori alkalmazása a hatékonyság kárára mehet). A körültekintő elemzés mellett a helyes/optimális szint meghatározásához a háttérben használt adatbázisrendszer ismerete is szükséges lehet.

Elkülönítési szintek beállítása

- A különböző Java EE szerverek különböző módon oldják meg.
- Lehetőség van a JDBC API segítségével az adatbázis kapcsolat szintjén meghatározni az elkülönítési szintet:
  - ```
    DataSource source = (javax.sql.DataSource)
                        jndiCntxt.lookup("java:comp/env/jdbc/titanDB");
    Connection con = source.getConnection();
    con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
    ```
- Az erőforrásokra vonatkozó elkülönítési szint különbözhet különböző tranzakciókon belül, de egy adott tranzakción belül a szintnek azonosnak kell lennie.
    ```

Optimista zárolási stratégia

- a túl megszorító megkötés az elkülönítési szinttel kapcsolatban a hatékonyság rovására mehet (ha sok különböző kliens akar hozzáférni ugyanahhoz az erőforráshoz ki kell várniuk a "sorukat" → skálázhatósággal kapcsolatos problémák).
- Optimistic locking tervezési minta: feltételezzük, hogy ugyanabban az időpontban különböző kliensek nem akarnak ugyanahhoz az adathoz hozzáférni, és a tranzakció véglegesítésekor (commit) az adatbázisrendszerre hagyjuk, hogy eldöntse, másik tranzakció módosította-e közben az adatokat. Ha igen, kivételt dobunk és visszaállítjuk a tranzakciónk előtti állapotot (rollback) ("optimistán" feltételezzük, hogy más nem foglalkozik az adatokkal a tranzakciónk lefutása alatt).
- Megoldás: Version mezők használata az adatbázis bejegyzések esetében → @javax.persistence.Version annotáció az entitásokon belül:
  - ```
    @Entity
    public class BankAccount {
      …
      @Version
      private long version;
      …
    }
    ```
- A tranzakció előtt az EntityManager beolvassa az értéket, a végén ellenőrzi, hogy az aktuálisan tárolt érték megegyezik-e a beolvasottal. Amennyiben nem, kivételt dob (javax.persistence.OptimisticLockException) és visszaállítja az eredeti állapotot (rollback), ellenkező esetben növeli a verziószámot és hagyja lefutni a tranzakciót.
    ```

Zárolás a programból

- Programmatic locking: az EntityManager lock metódusának alkalmazása.
- A metódus egy entitás példányt kap paraméterként, és read, vagy write lock-ot alkalmaz rá:

```
void lock(Object entity, LockModeType type);
```

- LockModeType.READ – nem fordulhat elő dirty vagy nonrepeatable read az illető entitás esetében.
- LockModeType.WRITE – azonos hatás a READ-el, de a verziószám növelését is eredményezi.
- Megjegyzés: egyes megvalósítások nem garantálják, hogy verziószámmal nem rendelkező entitások esetében is működjön ez a zárolási mechanizmus.

JavaEE: Biztonság

Security - authentication, authorization

Simon Károly
simon.karoly@codespring.ro

Authentication

- Felhasználók azonosítása: helyzet/alkalmazásszerver-specifikus módon történik.
- Egyszerű eset: JNDI API-alapú azonosítás:
  - …

```
properties.put(Context.SECURITY_PRINCIPAL, "username");
properties.put(Context.SECURITY_CREDENTIALS, "password");
…
Context ctx = new InitialContext(properties);
Object ref = jndiContext.lookup("SecureBean/remote");
SecureRemoteBusiness remote = (SecureRemoteBusiness) ref;
```

- Alkalmazásszerverek által biztosított lehetőségek (pl. Java Authentication and Authorization Service (JAAS)).

Authorization

- Users, user groups → roles
- Role-based authorization
- Példa:
  - ```
    public interface SecureSchoolLocalBusiness {
      void open();
      void close();
      void openFrontDoor();
      void openServiceDoor();
      boolean isOpen();
    }
    ```
  - ```
    public interface Roles {
      String ADMIN = "Administrator";
      String STUDENT = "Student";
      String JANITOR = "Janitor";
    }
    ```

Authorization

```
@Singleton
@Local(SecureSchoolLocalBusiness.class)
@DeclareRoles({Roles.ADMIN, Roles.STUDENT, Roles.JANITOR})
@RolesAllowed({})
@Startup
public class SecureSchoolBean implements SecureSchoolLocalBusiness {

        @PermitAll
        public boolean isOpen() {…}

        @RolesAllowed(Roles.ADMIN)
        public void close() {…}

        @PostConstruct
        @RolesAllowed(Roles.ADMIN)
        public void open() {…}

        @RolesAllowed({Roles.ADMIN, Roles.JANITOR})
        public void openServiceDoor() {…}

        @RolesAllowed({Roles.ADMIN, Roles.STUDENT, Roles.JANITOR})
        public void openFrontDoor() {…}

}
```

Programmatic Security

- …

```
@Resource private SessionContext context;
…
final String userName = context.getCallerPrincipal().getName();
if (!this.isOpen()) {
        if (!context.isCallerInRole(Roles.ADMIN)) {
                throw SchoolClosedException.newInstance(
                        "Front door access is denied for " + userName
                        + ". Only admins can open the door after …");
        }
}
```

RunAs Security Identity

- A metódusok szerepköre.
- MDB-k esetében kötelező, amikor más "védett" bean-ekkel kommunikálnak (mivel itt a hívó fél szerepköre nem továbbítódik).
- Példa:

```
public interface FireDepartmentLocalBusiness {
      void declareEmergency();
}
```

-
```
@Singleton
@RunAs(Roles.ADMIN)
@PermitAll
public class FireDepartmentBean implements FireDepartmentLocalBusiness {

      @EJB SecureSchoolLocalBusiness school;

      public void declareEmergency() {
            …
            school.close();
      }
}
```

JavaEE: Timer Service

Simon Károly
simon.karoly@codespring.ro

Timer Service

- Timed-event API: időzítőt rendelhetünk hozzá EJB-ekhez, amelyek a megfelelő időpontokban meghívják a bean ejbTimeout() metódusát, vagy egy @javax.ejb.Timeout annotációval ellátott metódusát.
- Példa: Batch Credit Card Processing System
- ```
  public interface CreditCardTransactionProcessingLocalBusiness {
        List<CreditCardTransaction> getPendingTransactions();
        void process();
        void add(CreditCardTransaction) throws IllegalArgumentException;
        Date scheduleProcessing(ScheduleExpression exp)
                                        throws IllegalArgumentException;
  }
  ```
- A ScheduleExpression felhasználásával hozhatunk létre a TimerService segítségével egy Timer objektumot. Alternatív megoldásként, deklaratív módon, a Schedule annotáció segítségével is hozhatunk létre Timer-eket.
- @javax.ejb.Schedule attribútumok: second (0-59), minute (0-59), hour (0-23), dayOfMonth (1-31 vagy –x, ahol x a "Last" előtti napok száma), month (1-12 vagy {"Jan", "Feb"…}), dayOfWeek (0-7 vagy {"Sun", …}, a 0 és a 7 is "Sun"), year (négy számjegy).
- Az időzítést meghatározó kifejezések szintaxisa flexibilis, wildcard karakterek (*), listák, intervallumok is használhatóak.
  ```

Scheduling

- Ha egy EJB TimerService-t használ, implementálnia kell a TimedObject interfészt, illetve annak ejbTimeout(Timer timer) metódusát, vagy rendelkeznie kell egy Timeout annotációval ellátott metódussal.
- Időzítés beállítása a bean-en belül - a klienstől kapott ScheduleExpression felhasználásával:

```
…
@Resource private SessionContext context;
@Resource private TimerService timerService;
…
public Date scheduleProcessing(final ScheduleExpression expression)
                                                    throws IllegalArgumentException {
      if (expression == null) throw new IllegalArgumentException;
      final TimerService timerService = context.getTimerService();
      final Timer timer = timerService.createCalendarTimer(expression);
      final Date next = timer.getNextTimeout();
      log.info("Created " + timer + ", next fire is at " + next);
}
```

- Vagy automatikusan, a Schedule annotáció segítségével, timeout metódus létrehozásával:

```
@Timeout
@Schedule(dayOfMonth = EVERY, month = EVERY, year = EVERY,
                  second = ZERO, minute = ZERO, hour = EVERY)
public void processViaTimeout(final Timer timer) {
      this.process();
}
```

TimerService

- Két időzítő típus: single-action és interval
- ```
  package javax.ejb;
  public interface javax.ejb.TimerService {
        public Timer createTimer(long duration, java.io.Serializable info);
        public Timer createTimer(java.util.Date expiration, java.io.Serializable info);
        public Timer createSingleActionTimer(long duration, TimerConfig timerConfig);
        public Timer createSingleActionTimer(java.util.Date expiration, TimerConfig
                              timerConfig);
        public Timer createTimer(long initialDuration, long intervalDuration,
                              java.io.Serializable info);
        public Timer createTimer(java.util.Date initialExpiration, long intervalDuration,
                              java.io.Serializable info);
        public Timer createIntervalTimer(long initialDuration, long intervalDuration,
                              java.io.Serializable info);
        public Timer createIntervalTimer(java.util.Date initialExpiration, long
                              intervalDuration, TimerConfig timerConfig);
        public Timer createCalendarTimer(ScheduleExpression schedule);
        public Timer createCalendarTimer(ScheduleExpression schedule, TimerConfig
                              timerConfig);
        public Collection<Timer> getTimers();
  }
  ```
- A létrehozás után alapértelmezetten a TimerService perzisztensé teszi a Timer objektumokat. Az időzítők rendszerösszeomlás esetén, annak visszaállítása után érvényesek maradnak (az összeomlás alatt lejárt időzítők "sorsát" nem rögzíti a specifikáció, ez implementáció függő, de általában lejártnak minősülnek a visszaállítás után).
  ```

Timer

- ```
  package javax.ejb;
  public interface javax.ejb.Timer {
        public void cancel();
        public long getTimeRemaining();
        public java.util.Date getNextTimeout();
        public javax.ejb.ScheduleExpression getSchedule();
        public javax.ejb.TimerHandle getHandle();
        public java.io.Serializable getInfo();
        public boolean isPersistent();
        public boolean isCalendarTimer();
  }
  ```

- Az időzítők beazonosítására használhatjuk az info objektumot, amely bármilyen szerializálható objektum lehet (egyszerű esetben String).
- A TimerHandle objektum egy a Timer-re mutató referenciát tartalmaz, egyszerűsíti például a mentést, majd későbbi felhasználást. A handler getTimer metódusa téríti vissza a Timer referenciát, mindaddig, amíg érvényes az időzítő, egyébként NoSuchObjectLocalException típusú kivételt dob.
- A createTimer metódus mindig az aktuális tranzakció hatókörében lesz végrehajtva (rollback esetén az időzítő nem jön létre). A timeout callback metódus tranzakció attribútumának általában RequiresNew-nek kell lennie.
  ```

Stateless Session Bean Timers

- Az időzítő lejártakor a rendszer az instance pool-ból tetszőleges példányt választ és végrehajtja annak timeout metódusát. Ha nincs még példány a pool-ban, létrehoz egyet.
- Az injektált TimerService példányhoz a bean a SessionContext-en keresztül hozzáférhet bármely business metóduson belül, illetve a @PostConstruct és @PreDestroy metódusokon belül is (nem férhet hozzá a setter injection metódusokon belül, a kliensnek kezdeményeznie kell valamilyen művelet végrehajtását ahhoz, hogy az időzítő érvényessé váljon).
- @PostConstruct metóduson belül időzítőt beállítani nem ajánlott. Például nem biztos, hogy a konténer létrehoz példányt mielőtt konkrét kérés érkezik egy klienstől, így az időzítő sem lesz érvényes. Másik probléma, hogy a metódus minden példány létrehozása után meghívásra kerül, és ki kell védeni a helyzetet, hogy ugyanannak az időzítőnek több példánya jöjjön létre. Osztályszintű kontrollváltozó használata megoldás lehetne, de ez problémás lehet, ha a program klaszteren fut, és/vagy több VM-et vagy több classloadert használ (különböző VM-eken belül különböző értéke lesz a változónak). Ha a metóduson belül lekérdezzük az aktuálisan érvényes időzítőket, csökken a hatékonyság, mivel mindig meghívásra kerül a getTimers() metódus. Ezen kívül klaszterek esetében nem biztos, hogy minden gépen minden időzítő látható lesz.
- @PreDestroy metóduson belül nincs értelme időzítőt létrehozni, vagy kikapcsolni (cancel). A metódus szintén különálló példányokra vonatkozik.

Message-Driven Bean Timers

- Rendeltetésük a Stateless Session Bean Timer-ekével megegyező, beérkező üzenetek hatására lesznek létrehozva (vagy ritkábban konfigurációs állomány alapján).
- A createTimer metódust az üzenetkezelő metóduson belül hívjuk meg, JMS használata esetén az onMessage() metóduson belül:

```
@MessageDriven
public class JmsTimerBean implements MessageListener {
      @Resource TimerService timerService;
      public void onMessage(Message message) {
                MapMessage mapMessage = (MapMessage) message;
                long expirationDate =
                          mapMessage.getLong("expirationDate");
                timerService.createTimer(expirationDate, null);
      }
      @Timeout
      public void timeout() {
                …
      }
}
```

Aspektusorientált programozás

AspectJ – rövid bevezető

Simon Károly
simon.karoly@codespring.ro

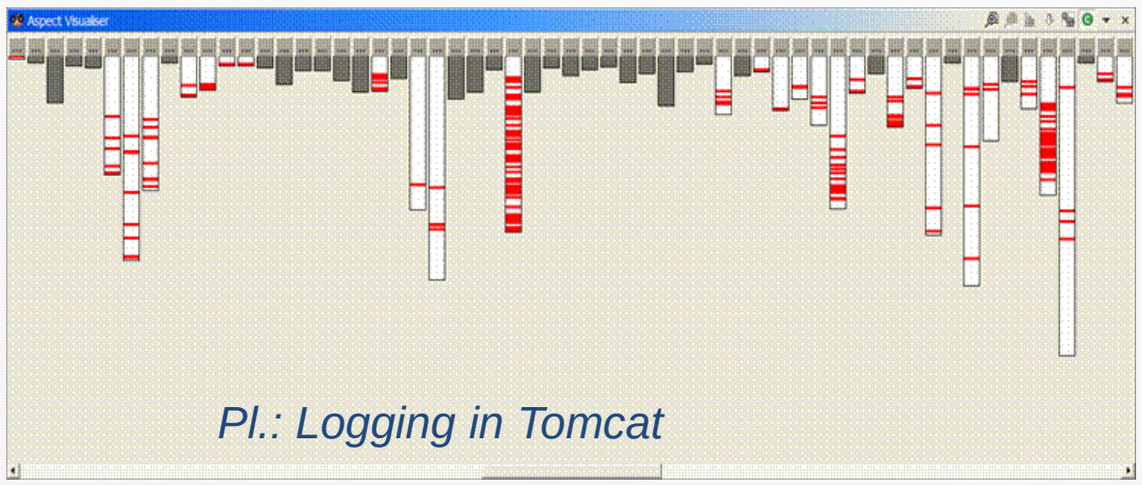
AOP

- Áttekinthetőség, karbantarthatóság, újrafelhasználhatóság → modularizáció
- Objektumorientált programozás
- **Átmetsző követelmények (crosscutting concerns)**
  - Nem implementálhatóak egy modul által.
  - Nem funkcionális követelmények, vagy olyan funkcionális követelmények, amelyek az eredeti dekompozíciónál még nem voltak ismertek.
  - Példák: naplózás (több különböző ponton van szükség ilyen műveletekre), biztonság (több különböző ponton szükséges ellenőrzések) stb.
- **Aspektus:** egy átmetsző követelmény reprezentációja
  - Megváltoztatják a komponensek működését egy jól meghatározott módon.
  - A megváltoztatandó programrészek kontextusában léteznek (nem különállóan) (Paller Gábor: „az objektum valami, az aspektus valaminek a valamije”).
  - A programrészek megváltoztatása az aspektusok által: "szövés" (**weaving**) → az aspektus programhelyeket azonosít, és a beazonosított helyeken megváltoztatja a kódot.
- **Aspektusorientált programozás:** aspektusok modularizálása.
- Komponensnyelv (általában OO, pl. Java) + AO nyelv (pl. AspectJ).

Példa: scattered/tangled code

- Scattered (szétszórt) kód: pl. naplózás → a naplózási mechanizmus esetleges változtatása több modult érint.
- Tangled (összekuszált) kód: pl. egy modulon belül biztonság+naplózás → egy ilyen modul implementálásához mindenik mechanizmust ismerni kell.
- Példa:

```
void transfer(Account fromAccount,
    Account toAccount, int amount) {
  if (fromAccount.getBalance() < amount) {
     throw new InsufficientFundsException();
  }
  fromAccount.withdraw(amount);
  toAccount.deposit(amount);
}
```



Pl.: Logging in Tomcat

```
void transfer(Account fromAccount,
    Account toAccount, int amount) {
  if (!getCurrentUser().
      canPerform(OP_TRANSFER)){
    throw new SecurityException();
  }
  if (amount < 0) {
    throw new NegativeTransferException();
  }
  if (fromAccount.getBalance() < amount) {
    throw new InsufficientFundsException();
  }
  Transaction tx = database.newTransaction();
  try {
    fromAccount.withdraw(amount);
    toAccount.deposit(amount);
    tx.commit();
    systemLog.logOperation(OP_TRANSFER,
        fromAccount, toAccount, amount);
  } catch(Exception e) {
    tx.rollback(); }
}
```

Join point modellek

- **Join point:** az aspektus által beazonosított programpont, ahová az aspektus kódrészletet szúr be.
- **Pointcut:** pontszűrő, a programpontokat kiválasztó kifejezés, amely meghatározza azokat a programhelyeket, ahol egy adott aspektust alkalmazni kell.
- **Advice:** tanács, a beszúrandó kódrészlet, illetve a beszúrás szabályai.
- Pointcut példák (AspectJ):
  - `execution(* set*(*) )` - olyan metódusok végrehajtása, amelyeknek neve set-tel kezdődik és egy (bármilyen típusú) paraméterrel rendelkeznek
  - `this(Point)` – az aktuális objektum a Point osztály példánya
  - `within(com.company.*)` – minden join point a com.company csomagon belül (osztály esetében ugyanígy alkalmazható)
  - Kombinálva:

```
pointcut set():execution(*set*(*)) && this(Point)&& within(com.company.*)
```

- Advice példa:
  -

```
after() : set() {
   Display.update();
}
```

A set() nevű pontszűrővel találó join point –okhoz tartozó kód végrehajtása után meghívja a Display.update() metódust.

Példa

```
public class Hello {
  public void saySomething() {
    System.out.println("Something");
  }
  public int saySomethingElse(String msg) {
    System.out.println("Something else: " + msg);
    return 0;
  }
  public static void main(String[] args) {
    Hello h = new Hello();
    h.saySomething();
    h.saySomethingElse("hello");
  }
}
```

```
public aspect Example1 {
  pointcut p1() : execution( * *.say*( .. ) );
  before(): p1() {
    System.out.println("This");
  }
  after(): p1() {
    System.out.println("That");
  }
}
```

This
Something
That
This
Something else: hello
That

Pontszűrő példák

- **call**: hasonló az execution programpont-kiválasztóhoz, csak, míg az execution a metódus törzsébe szúrja be a kódot, a call a hívás elé/után (általában nincsen jelentősége, de pl. a main esetében nem alkalmazható a call).
- **execution(Hello.new()):** a Hello osztály paraméter nélküli konstruktora
- **int *.say*(String):** a metódus bármelyik tulajdonságára rákereshetünk
- **MyInterface.new():** az összes interfészt implementáló osztály konstruktora
- ***Hello+.method():** a Hello osztálynak és leszármazottainak metódusa
- **execution(public !static * *(..)):** minden publikus nem statikus metódus
- Paraméteres pontszűrő (hozzáférhet a metódus paramétereihez):

```
pointcut p1(String parms) : execution(int saySomethingElse(String)) && args(parms);
  before(String parms): p1(parms) {
    System.out.println( "saySomethingElse called with parameter " + parms );
  }
```

További példák

- További pontszűrő példák:
  - handler(ArrayIndexOutOfBoundsException) - ha a megadott kivétel kezelője fut
  - target(SomeType) - ha a metódushívás célobjektuma bizonyos típusú
  - withincode(* say*()) - illeszkedik a metódus törzsében levő utasításokra
  - set(int T.x) – a T osztály x egyedváltozójának (int típusú) értékadó műveleteinek elkapása
  - set(public * *) - az összes osztály mindenik publikus attribútumának írása
  - get(T.x) - az egyedváltozók elérésének keresése
  - Stb.

- Inter-type declarations: egy osztály adattagjait vagy metódusait egy másik helyen deklaráljuk (hogy egy adott aspektushoz tartozó kód egy helyen legyen):

```
aspect DisplayUpdate {
      void Point.acceptVisitor(Visitor v) {
                v.visit(this);
      }
      // other crosscutting code...
}
```

- Kritikák:
  - Kód védelme (az aspektusokon keresztül egy már létező kód viselkedés módosítható).
  - Biztonsági megfontolások (pl. aspect, amely egy jelszóellenőrzésnél mindig true-t visszatérítő kódrészletet helyez a kódba).
  - Stb.

További példák

- ```
  public class Hello2 {
    public int saySomethingElse(String msg) {
      if (msg.equals("hello"))
          throw new IllegalArgumentException();
      System.out.println("Something else: " + msg);
      return 0;
    }
    public static void main( String args[] ) {
      Hello2 h = new Hello2();
      h.saySomethingElse("hallo");
      h.saySomethingElse("hello");
    }
  }
  ```

```
Something else: hallo
saySomethingElse returned
saySomethingElse threw exception
Exception in thread "main"
java.lang.IllegalArgumentException
        at Hello2.saySomethingElse(Hello2.java:5)
        at Hello2.main(Hello2.java:13)
```

- ```
  public aspect Example3 {
    pointcut p1() : execution(int saySomethingElse(String));
    after() returning: p1() {
      System.out.println("saySomethingElse returned");
    }
    after() throwing: p1() {
      System.out.println("saySomethingElse threw exception");
    }
  }
  ```

További példák

- A saySomething metódus helyettesítése egy teljesen új implementációval:

```
public aspect Example4 {
 pointcut p1(String s) :
     execution(int saySomethingElse(String)) && args(s);
 int around(String s) : p1(s) {
   System.out.println("New implementation printing " + s);
   return 1;
   }
}
```

- Új attribútum hozzáadása és használata az aspektuson belül:

```
public aspect Example5 {
 public static int Hello.callCtr = 0;
 pointcut p1() : execution( * *.say*( .. ) );
 before() : p1() {
   ++Hello.callCtr;
   System.out.println(
     "say* method executed " + Hello.callCtr + " times" );
 }
}
```

További példák

- Az osztály leszármazási hierarchiájának megváltoztatása:

```
public aspect Example6 {
  declare parents: Hello extends java.applet.Applet;
  public void Hello.init() {
    saySomething();
    saySomethingElse("hello");
  }
```

- Interfész „implementáltatása”: a Point, Line és Square osztályok implementálják az aspektuson belül definiált HasName interface-t, és az aspektus az interfész metódusainak implementációját is szolgáltatja:

```
public aspect A {
  private interface HasName {}
  declare parents: (Point || Line || Square) implements HasName;

  private String HasName.name;
  public  String HasName.getName()  {return name;}
}
```

JavaEE: Interceptors

Simon Károly
simon.karoly@codespring.ro

Intercepting Methods

- Egyfajta (egyszerűsített/alapvető) AOP támogatás.
- Interceptor osztály létrehozása: egyszerű POJO, egy @javax.interceptor.AroundInvoke annotációval ellátott metódussal:

```
@AroundInvoke
Object <any-method-name>(javax.interceptor.InvocationContext invocation)
                                                    throws Exception;
```

-
```
@AroundInvoke
public Object audit(final InvocationContext context) throws Exception {
        …
        try {
                log.info(context + " intercepted");
                return context.proceed();
        } finally {
                log.info("done");
        }
}
```

InvocationContext

- ```
  package javax.interceptor;
  public interface InvocationContext {
        public Object getTarget();
        public Method getMethod();
        public Object[] getParameters();
        public void setParameters(Object[] newArgs);
        public java.util.Map<String, Object> getContextData();
        public Object proceed();
        public Object getTimer();
  }
  ```
  ```

Applying Interceptors

- Interceptorok alkalmazása: annotációk vagy XML leírók segítségével.
- Az @Interceptors annotáció alkalmazható különálló metódusok szintjén, vagy a bean osztály szintjén:

```
@Stateless
@Interceptors({Auditor.class})
public class MyBean implements MyLocalBusiness {
        …
}
```

- XML leíró alkalmazása esetén az <interceptor-binding> elem alkalmazható, az <ejb-name>, <interceptor-class>, <method-name>, <method-params> belső elemekkel. Az <ejb-name> elem esetében wildcard is alkalmazható (default interceptor-ok bean-ekhez rendelése).
- Interceptorok inaktiválása az @ExcludeDefaultInterceptors annotáció, vagy az <exlude-default-interceptors> elem segítségével lehetséges a bean-ek szintjén. Metódusok szintjén rendelkezésünkre áll az @ExcludeClassInterceptors annotáció.

Interceptors

- EJB lifecycle eseményekre is alkalmazhatóak.
- Segítségükkel testreszabott Injection annotációk is létrehozhatóak.
- Kivételkezelésnél alkalmazhatóak. Lehetőség van bizonyos kivételek felléptekor a metódushívás megszakítására, vagy adott kivétel kezelésére, és más kivételtípus továbbdobására.
- Életciklusuk megegyezik az érintett EJB-ek életciklusával (a bean példány egyfajta kiterjesztésének tekinthetőek, vele együtt jönnek létre, vele együtt lesznek aktiválva, passzivá téve, törölve).
- Az @AroundInvoke metódusok nem csak interceptor osztályokon belül, hanem a bean osztályon belül is alkalmazhatóak (például, amennyiben egy dinamikus implementációt szeretnénk, vagy az interceptor bean-specifikus). Az ilyen esetekben a beanen belül használt interceptor lesz utolsóként meghívva a konkrét metódushívás előtt.