ASPEKTISUUNTAUTUNUT OHJELMOINTI JA OHJELMISTON MODULARISOINTI

Juhani Sannikka
Erikoistyö
Tietojenkäsittelytiede
Kuopion yliopiston
tietojenkäsittelytieteen laitos
Maaliskuu 2005

KUOPION YLIOPISTO, informaatioteknologian ja kauppatieteiden tiedekunta
Tietojenkäsittelytieteen koulutusohjelma

JUHANI SANNIKKA, Aspektisuuntautunut ohjelmointi ja ohjelmiston modularisointi
Erikoistyö, 73 s., 12 liitettä (62 s.)
Erikoistyön ohjaaja: FM Harri Karhunen
Erikoistyön tarkastaja: FM Heli Lintula
Huhtikuu 2005

Avainsanat: aspectj, modularisointi, autentikointi, auktorointi, transaktio

AspectJ-ohjelmointikieli on Java-kielen laajennos, joka tarjoaa myös aspektisuuntautuneen ohjelmoinnin työkalun. Se sisältää AspectJ-kääntäjän, jonka tuottamaa tavukoodia tavallinen Javan virtuaalikone ymmärtää. Aspektien koodauksessa käytetään hyväksi luokissa toteutettua perustoiminnallisuutta.
Join point tarkoittaa luokissa toteutettua ohjelman kohtaa, kuten metodia, konstruktoria tai ohjelman kontrollivirtaa, jossa niitä suoritetaan. Pointcut on aspektissa esitelty ja määritelty ohjelmalohko, joka sieppaa halutut ohjelman join point –kohdat suoritettavaksi. Sieppaaminen voi perustua luokan tai metodin nimeen tai tyyppiin. Niiden kuvaamiseen voidaan käyttää reflektointia. Pointcut –lohkojen valitsemien join point -kohtien suorituksia voidaan ohjeistaa advice-määrittelyillä. Advice-lohkon koodi täydentää join point -kohtien toimintoja. Advice-lohkon koodi voidaan suorittaa niitä ennen, niiden jälkeen tai se voi korvata alkuperäisen toiminnon kokonaan.
AspectJ-ohjelmointikieli on aspektisuuntautunut ohjelmointikieli, joka mahdollistaa luokkien toiminnallisuuden koostamisen ja käsittelemisen kokonaisuuksina, jotka muodostavat ohjelman kannalta poikkileikkauksellisia tehtäviä tai ominaisuuksia. Pelkkänä luokkatoteutuksena näiden tehtävien koodi on vaikeasti hallittavissa, koska se levittäytyy laajalle ohjelmistoelementtien koodin sekaan. Aspekti auttaa poikkileikkauksellisten tehtävien modularisoinnissa, jolloin ohjelman rakenne selkiytyy. Ohjelmiston perustoiminnallisuus on paremmin hallittavissa ja myös muutokset aspektien muodostamaan modulaariseen rakenteeseen on helpompi toteuttaa, kuin luokkiin levitetyn koodin muutokset. Aspekteja voidaan määritellä myös ohjaamaan ohjelmistokehityksen koodaustapoa. Tällaiset aspektit saavat kääntäjän tulostamaan varoituksia tai estämään väärän koodauksen.
Tässä raportissa kerrotaan aspektisuuntautuneen ohjelmoinnin keinoista toteuttaa poikkileikkauksellisten tehtävien, kuten autentikointi, auktorisointi ja transaktiot modulaarisina uudelleen käytettävinä ratkaisuina. Raportissa kerrotaan myös ratkaisuissa käytettävistä suunnittelumalleista.
Luvussa 1 esitellään aspektisuuntautuneen ohjelmointitavan historiaa ja käsitteistöä. Luvussa 2 esitellään AspectJ-kieli, joka toteuttaa aspektisuuntautuneen ohjelmoinnin ympäristön. Luvussa 3 käsitellään esimerkkejä suunnittelumallien toteuttamiseksi aspekteina ja luvussa 4 säikeiden käsittelyä modulaarisesti aspektien avulla. Luvussa 5 käsitellään järjestelmän autentikoinnin ja auktorisoinnin modulaarisia ratkaisuja. Luvussa 6 esitetään erilaisia aspektisuuntautuneen ohjelmoinnin mahdollisuuksia transaktion modulaariseen toteutukseen. Lukujen 3-6 esimerkit perustuvat AspecJ in Action –teokseen, jonka on kirjoittanut R. Laddad.

Sisällysluettelo

1 ASPECT ORIENTED PROGRAMMING

1.1 Oliosuuntautuneisuuden ohjelmoinnin kehittyminen

Tietojenkäsittelyjärjestelmät ja ohjelmointikielet ovat kehittyneet *konekielistä* (*assembly*) käsitteiden ja kaavojen muodostamiseen ja kääntämiseen, proseduurien, rakenteiden, toimintojen, logiikan ohjelmointiin ja abstraktien tietotyyppien käyttöön. Tämä ohjelmointitekniikan kehitys on parantanut mahdollisuuksia käsitellä lähdekoodissa erillisiä *tehtäväkonaisuuksia* (*concerns*).

Nykyisin vallitsevassa oliosuuntautuneessa ohjelmoinnissa (OOP, Object Oriented Programming) ohjelman käsitteet ja toiminnot on ratkaistu olioiden ja niiden toimintoja ohjaavan koodin avulla. Olioiden käyttäytyminen ja data muodostavat abstrahoidut (ja fyysiset) käsitteet ohjelman suorituksessa. Olio-ohjelmoinnilla pysytään ratkaisemaan hyvin monimutkaisia ongelma-alueita, mutta silläkin on eräitä rajoitteita. Esimerkiksi joitakin ohjelman suorituksessa vaadittuja toimintoja ei voida koodata yhden ohjelman osan suoritettavaksi, vaikka ne sille luontevasti kuuluisivatkin. OO-teknologiassa on huomioitava globaaleja rajoitteita (constraints) ja kaikkialla yleisesti vaikuttavaa (pandemic) käyttäytymistä, joita on vaikeaa hallita paikallisesti. Tämä voi vaikeuttaa ongelma alueiden eriyttämistä ja edellyttää käyttökohteen toiminnan alan mukaista soveltamista (domain-specific).

Post-object programming (POP) on olio-ohjelmoinnin seuraava kehitysvaihe, joka laajentaa sen ilmaisu kykyä. POP on edesauttanut esimerkiksi seuraavien alojen kehittymistä:

- toiminnan alan erityiskielet (domain-specific language),
- generatiivinen eli muodostava ohjelmointi (generative programming),
- geneerinen eli yleistävä ohjelmointi (generic programming),
- sääntöjä ja rajoitteita käyttävät sääntökielet (constraint languages),

- luokkien peilaus (reflection) ja meta-ohjelmointi (metaprogramming) ,
- ominaisuus-suuntautuneiden ohjelmistojen kehittäminen (feature-oriented development),
- näkymät ja näkökulmat (views, viewpoints), sekä
- epäsynkroninen viestin välitys (asynchronous message brokering).

Seuraavassa luvussa kerrotaan AOP:sta (Aspect-Oriented Programming), joka on eräs POP-teknologia. AOP tekniikka parantaa ohjelmistojen toteutuksen hallittavuutta, koska ohjelmiston eri kokonaisvaatimuksia tai -huolenaiheita (concerns) määritellään erillisinä ohjelman osina. Näille osille määritellään kuvauksia niiden yhteyksistä (relationship), joiden mukaisesti ohjelman osia koostetaan (weave) yhdeksi kokonaisuutena toimivaksi sovellukseksi.

1.2 AOP-tekniikan tavoitteet

Ohjelman toiminnot (concerns) voivat huolehtia tehtävistä, jotka muodostuvat käsitteistöltään laajoista päättelysäännöistä (high-lewel notions), kuten ohjelmiston turvallisuutta ja laatua palvelevat tehtävät, tai ne voivat huolehtia myös pienemmistä tehtävistä (low-lewel notions), kuten ajureista ja puskuroinnista. Ohjelman osilla voi olla toiminnallisia piirteitä, esimerkiksi ne voivat toteuttaa liiketoiminnallisia palveluja (business rules). Ohjelman osilla voi olla myös ei-toiminnallisia piirteitä, jotka ohjaavat järjestelmää, kuten synkronointi ja transaktioiden hallinta. Vertauksellisesti voidaan sanoa, että kun OOP pyrkii hyödyntämään luokkien yhteyttä ja perintäpuuta, AOP pyrkii toteuttamaan ohjelman toiminnot erillisistä primääri-luokkaisista (first-class) elementeistä, jotka käynnistyvät suoraan horisontaalisesta oliorakenteesta.

Rakenteellisesti jotkin tietojärjestelmän toiminnot voidaan toteuttaa kätevästi yhdessä rakenneosassa, kun taas joidenkin toimintojen toteutus levittäytyy monelle rakenne-

osalle. AOP keskittäytyy mekanismeihin, joilla näitä eri rakenneosille levittäytyviä toimintoja voidaan yksinkertaistaa. AOP:n päämääränä on modularisoida toteutettavien ohjelmistojen rakenne, jolloin pienen muutoksen tekeminen loogisella tasolla ei aiheuta koodin tai suunnittelun tasolla muutoksia moniin eri paikkoihin ohjelmistossa [ElF01].

Ohjelmiston *näkymävaatimukset* (*aspectual requirements*) ovat ongelma-alueita (concerns), joita voidaan ratkaista levittämällä toteutus osissa usealle rakenneosalle. Esimerkkejä tällaisista ongelma-alueista ovat:

- Synkronointimenetelmät, jotka vaativat melkoisen joukon operaatioita varmistamaan yhdenmukaisen lukitusprotokollan,
- Monimutkaisten oliograafien läpikäynti, johon tarvitaan globaalia informaatiota.
- Laskutus mekanismit, joilla hallitaan veloituksia.
- Vikasietoiset mekanismit, jotka luovat toistuvasti yhdenmukaisia kopioita.
- Sekä palvelun laatua varmistavat toiminnot, jotka vaativat järjestelmän prioriteettien määrittelyä, kuten autentikointi, jonot ja käyttöoikeudet.

Nämä edellä luetellut toiminnot on perinteisesti toteutettu aliohjelmilla, eikä AOP:kään hylkää tätä ratkaisua täysin. Perinteisessä toteutuksessa poikkileikkauksellista toimintoa käsittelevän aliohjelman koodista tulee suttuista ja vaikeasti hallittavaa, kun sitä joudutaan toteuttamaan useiden ohjelmistoelementtien koodin seassa. Tämän ongelman korjaamiseksi AOP tarjoaa aspektin. Se on mekanismi, joka kätkee alirutiinit ja perinnän sekä paikallistaa käsiteltävän toiminnon koodin ohjelmiston useista eri osa-alueista. AOP -kehitysympäristö sisältää punontamekanismin (weaver), joka punoo aspektit ja sovelluksen peruskoodin koossa pysyväksi järjestelmäksi.

AOP-kehitysympäristö sisältää sisäisen (implisiittisen) kutsumekanismin, joten lähdekoodin toteuttajan kanssa ei tarvita yhteistyötä, jotta tiedetään kuinka lähdekoodia voidaan kutsua, kuten aliohjelmia käytettäessä. Kun erilaisia toimintoja voidaan eriyttää muusta ohjelmiston kehittämisestä, voidaan yksinkertaistaa ja nopeuttaa järjestelmän

kehittämistä. Järjestelmistä saadaan myös helpommin ymmärrettäviä, muunneltavuus paranee myös asiakaskohtaisesti samoin kuin uudelleen käytettävyys.

Aspektin muodostava koodi yhdistää poikkileikkaavasti ohjelmassa huolehdittavia asioita (crosscutting concerns). Tämä modulaarinen koodi on helpompi käsitellä, kuin perinteisen ohjelmiston eri osiin tehty koodi. Myös ohjelmiston varsinainen kohdetta käsittelevä koodi on helpommin ymmärrettävissä, koska ohjelmoija voi kutsua aspektin koodissa esimerkiksi erikseen toteutettuja hajautusalgoritmeja, autentikointia, avauskäskyjä (access control), synkronointia, salausta, automatisointia (redundancy). Aspektit voivat olla uudelleen käytettäviä ja ne voivat olla liittimiä (connector) muihin komponentteihin.

1.3 Näkymien muodostaminen ja sovelluksen ominaisuudet

Aspekti voi määrittelynä sisältää assosiaation muista ohjelman käsitteistä, kuten muuttujat ja metodit, tyyppien esittelyt, poincut-määrittelyt ja advice-määrittelyt. AOP-tekniikassa aspekti muodostetaan valitsemalla ohjelman perustoiminnallisuuden luokista join point -kohdat, joiden suoritus palvelee kollektiivisesti ohjelman suoritusta [ElF01].

Join point –kohdat ovat ohjelman ydintoteutuksen koodia, jonka toiminnallisuutta aspekti hyödyntää. Ne voidaan ryhmitellä luonteensa mukaisesti suoritus- , kutsu- ja kentän käytön tyyppisiin join point –kohtiin:

Aspektissa voidaan viitata join point-kohtaan, joka suorittaa metodin, muuttujan alustajan, konstruktorin, staattisen alustajan, käsittelijän tai objektin alustajan. Samoin aspektissa voidaan viitata join point -kohtaan, joka kutsuu metodin, konstruktorin ja objektin esi-alustuksen. Join point -kohta voi myös viitata kentän arvoon tai asettaa sen [LaC03].

Pointcut-määrittelyt voivat sisältää viittauksia useisiin join point -kohtiin, jotka vaikuttavat aspektin muodostukseen eri tavoin. Pointcut- määrittelyn kautta aspektien koodi

on vuorovaikutuksessa järjestelmän muun koodin kanssa. Pointcut voidaan parametrisoida haluttua käyttötarvetta tai käyttäjää varten.

Pointcut -kohdat ovat useimmiten dynaamisia eli voivat viitata ohjelmassa meneillään oleviin tapahtumiin ja muuttavat ohjelman käyttäytymistä. Pointcut-lohkot sisältävät *määrittäjän* (*pointcut designator* eli *pcd*). Niissä voidaan määritellä halutut join point – kohdat ja ehtoja niiden suorittamiselle.

Pointcut -kohdan lisäksi aspektissa määritellään *ohjeita* (*advice*), nämä yhdessä vaikuttavat ohjelman etenemiseen käyttäen aspektissa toteutettuja palveluja järjestelmän kohdetoimintojen toteuttamiseen. Kun ohjelma etenee advice-kohtaan, niiden koodi suoritetaan. Advice-lohkon suoritus voi tapahtua ennen, jälkeen tai sen toiminnon sijaan mitä lohkossa määritelty koodi kutsuu.

Tätä prosessia, joka saa aikaan sopivan advice-lohkon suorittamisen kussakin liitoksessa kutsutaan näkymän koostamiseksi (weaving). Pointcut-määrittelyt voivat olla myös staattisia, jolloin ne vaikuttavat lähinnä vain sovelluksen rakenteen muodostumiseen ja tukevat dynaamisten poikkileikkauksellisten tehtävien toteuttamista [LaC03].

Tulkin (*interpreter*) suorittaessa ohjelmaa, se kutsuu saavuttamassaan pointcut-kohdassa *koostajaa* (*weaver*), joka huolehtii join point-, advice- ja pointcut-määrittelyjen mukaisten toimintojen liitosten valinnoista, niiden vaatimien ohjeiden kutsumisesta ja ohjeiden mukaisesti valittujen ohjelman osien suorittamisesta [ElF01].

Koostesäännöt (*weaving rules*) määrittelevät ohjelman etenemisen vaiheissa ohjelmassa suoritettavia toimintoja. AspectJ-kielessä näitä sääntöjä ei kirjoiteta ydintoimintojen toteutukseen, vaan ne sisältyvät näkymäkoodiin ja kääntäjän sääntöihin. [Lad03, ElF01].

1.4 AOP:n vaikutus modularisointiin

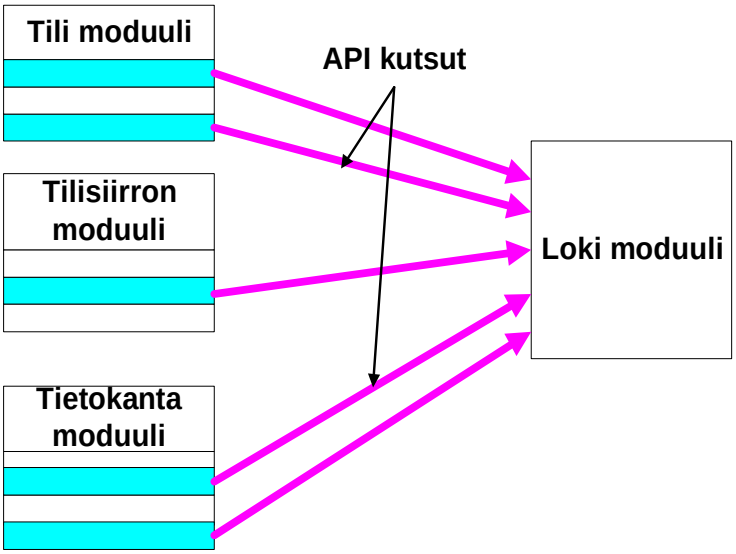
Perinteisessä olio-ohjelmoinnissa selkeästi eriytettyjä ydintoimintoja suorittavat moduulit voidaan toteuttaa rajapintojen avulla ja kytkeä löysästi toisiinsa. Sovellukseen

pitää kuitenkin toteuttaa toimintoja, jotka vaikuttavat sen useissa osissa, niin palvelimella kuin asiakassovelluksessa. Näitä toimintoja kutsutaan poikkileikkauksellisiksi tehtäviksi (crosscutting concern).

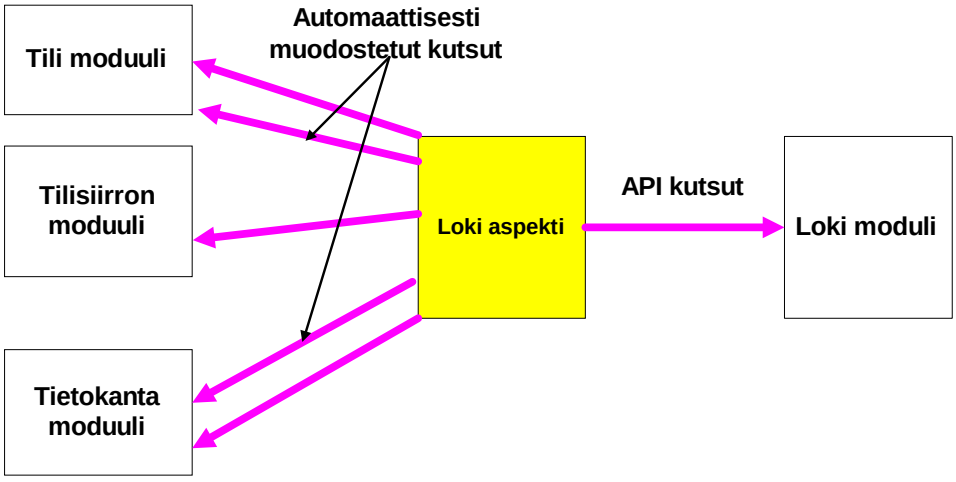
Esimerkiksi käyttäjän auktorisointi (*authorization*) -moduuli voidaan toteuttaa rajapinnalla, jonka käyttö ei sido tiukasti asiakassovelluksen toteuttamista. Ohjelmoijan ei tarvitse edes tietää auktorisointipalvelun tarkkaa toteutusta käyttäessään sitä ja palvelu voidaan myös vaihtaa toiseen toteutukseen. Kuitenkin jokaisen toteutukseen sisältyy palvelun käyttöön vaadittavaa koodia, mikä sekoittuu peruslogiikan toteuttamaan koodiin, joten palvelun vaihtaminen toiseen ei onnistu ilman suuria muutoksia järjestelmään.

Kuva 1 esittää perinteisen olio-ohjelmoinnin tekniikalla toteutettuja pankkijärjestelmän moduuleita; Tili-, Tilinsiirronhallinta- ja Tietokantamoduulit, sekä niiden toimintoja kirjoittava Lokimoduuli. Lokimoduuli voidaan toteuttaa abstraktin API:n, joka piilottaa yksityiskohtaisen toteutuksen. Kaikkien muiden moduuleiden täytyy kuitenkin sisältää tuota API:a kutsuvaa koodia.

AOP-tekniikkaa käytettäessä ydintoimintojen moduulit eivät sisällä API-kutsuja, eikä niissä tarvitse olla tietoisia Loki-moduulista, joka Loki-aspektin kanssa huolehtii lokin kirjoittamisesta, katso kuva 2. Jos tähän sovelluksen poikkileikkaukselliseen lokinkirjoitus tehtävään syntyy uusia vaatimuksia, tarvitsee muutoksia tehdä vain Loki-aspektiin, joka huolehtii tietojen sieppaamisesta ja kutsuu lokia kirjoittavaa API:a. Asiakasmoduulit eivät sisällä lokin kirjoittamiseen liittyvää koodia.



Kuva 1. Moduulien toteutus perinteisellä tekniikalla. API-kutsut kytkevät moduulit toisiinsa [Lad03].



Kuva 2. AOP-tekniikalla toteutettu pankkijärjestelmän lokinkirjoitus. Aspekti määrittelee sieppauskohdat ja kutsuu niitä suoritettaessa lokia kirjoittavaa API:a [Lad03].

Sovellusten poikkileikkauksellisten tehtävien modularisointi tekniikoita on useita. Esimerkiksi Enterprise JavaBeans (EJB) –arkkitehtuuri yksinkertaistaa hajautettujen palvelinpuolen sovellusten luomista ja laajojen tehtävien käsittelyä, kuten turvallisuuden (*security*), hallinta (*administration*), suorituskyky (*performance*) sekä säiliöinnin pysyvyys (*container-managed persistence*). Java-papujen kehittäjät voivat keskittyä liiketoimintalogiikkaan tai toisaalta asennusnäkymässä pavun datan kartoittamiseen tietokannan datan kanssa. Näin EJB-kehyksessä voidaan eriyttää erillisiksi tehtäviksi liiketoimintalogiikka ja asennuskuvauksen (deployment descriptor) avulla datan kartoitus pavun kentistä tietokannan sarakkeisiin. Asennuskuvaus on kartoituksen XML-tiedosto [Lad03].

Toinen poikkileikkauksellisten tehtävien modularisointitekniikka on dynaamisten proxien käyttäminen proxy-suunnittelumallin mukaisesti. Tämä on monimutkainen tekniikka, jota ei käsitellä tässä raportissa.

Modularisoimattomuus aiheuttaa järjestelmän koodin sekaisuuden, jos yksittäinen moduuli on toteutettu niin, että se huolehtii useista eri tehtävistä samanaikaisesti. Yhteen tehtävään liittyvää koodia joudutaan myös levittämään useisiin moduuleihin, joten ohjelmiston muutokset ovat työläitä ja alttiita virheille.

1.5 AOP-kielinen implementointi

AOP-metodologian mukainen järjestelmän kehittäminen etenee periaatteeltaan tavanomaisesti. Tunnistetaan tehtävät (concern), implementoidaan ne ja muodostetaan niistä yhdistämällä lopullinen järjestelmä. Tehtävien tunnistamiseksi vaatimukset jaotellaan yksittäisiin ydintehtäviin (core concerns) ja järjestelmätason poikkileikkauksellisiin tehtäviin (crosscutting concerns). Ensin mainittuja ovat esimerkiksi ydin liiketoiminnan tehtävät ja järjestelmätasolla monen moduulin alueelle vaikuttavia ovat esimerkiksi authorisointi, lokin kirjoitus, pysyvyys, säieturvallisuus ja puskurointi.

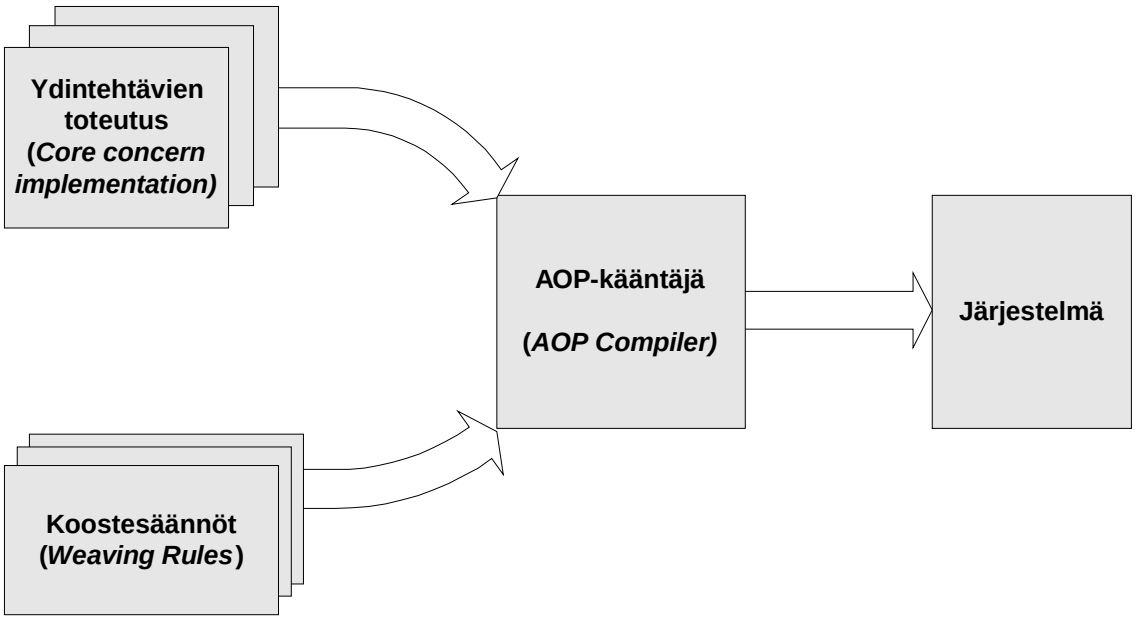
Seuraavassa vaiheessa jokainen näistä tehtävistä implementoidaan itsenäisesti. Ydin tehtävien toteutukseen voidaan käyttää tavanomaista olio-ohjelmointi tekniikkaa, eli

koodataan perustoiminnallisuuden ja rakenteet sisältävät ohjelmointikieliset rajapinnat ja luokat.

Lopuksi määritellään koostamissäännöt aspekteille, jotka muodostavat modularisoinnin yksiköt. Aspektien sisältämän informaation perusteella muodostuu lopullisen järjestelmän toiminnallisuus. Esimerkiksi aspektissa voidaan määritellä, että jokaisen sovelluksen operaation pitää varmistua asiakkaan oikeuksista ennenkuin sovellus etenee liiketoimintalogiikkaan [Lad03].

Kieliriippumaton AOP-toteutus muodostuu kaksivaiheisesti: ensin yhdistetään erilliset tehtävät koostesääntöjen (*weaving rules*) mukaisesti. Koostesäännöt määritellään aspekteissa. Ne voivat olla eri ohjelmointikielisiä toteutuksia kuin ydintehtävien toteutus. Kääntävälle prosessorille voidaan antaa ohjeeksi eri lähdekoodisia käännöksiä lopullisen suoritettavan koodin muodostamiseksi, mikä vaatii kuitenkin työläitä käännösvaiheita.

Kuva 3 esittää aspektien ja ydintehtävien toteutuksen yhdistämisen koostamissääntöjen mukaisesti järjestelmäksi. Tässä koostaja (weaver) sisältyy AOP-kääntäjään (AOP Compiler). Koostesäännöt (weaving rules) määrittelevät poikkileikkaukselliset tehtävät ja antavat lisätoiminnallisuutta ydintehtävien toteutukseen, joten niiden lähdekoodiin ei tarvitse tehdä muutoksia. Koostaminen ilmenee vain kääntäjän tuottamassa tavukoodissa. Ydintehtävien toteutus voi sisältää java-lähdekoodia ja -luokkatiedostoja. Järjestelmä sisältää koostetun ryhmän luokkatiedostoja.



Kuva 3. AOP kääntäjä toimii lopullisen järjestelmän koostajana (wiever) [Lad03].

Tässä raportissa perehdytään aspektien toteutukseen vain Java-ympäristössä käyttäen valmista AspectJ-kääntäjää. Ympäristön tarkempi selvitys on liitteessä 1.

2 ASPECTJ

AspectJ-ohjelmointikieli on aspektisuuntautunut Java-kielen laajennus, joten Java-kielinen ohjelma on myös kelvollinen AspectJ-ohjelma. AspectJ-kääntäjä (compiler) kääntää lähdekoodin luokkatiedostoiksi, jotka ovat yhdenmukaisia Javan tavukoodin kanssa, joten Javan virtuaalikone pystyy suorittamaan nämä luokkatiedostot.

AspectJ-käännöksessä toteutuvat dynaaminen ja staattinen poikkileikkauksellisuus. Dynaaminen poikkileikkauksellisuuden (*dynamic crosscutting*) koostaminen tarkoittaa uuden toiminnon tai käytöksen kutomista (weaving) ohjelman suoritukseen. Näin voidaan ottaa käyttöön tai korvata ydintoteutuksen moduulien toimintoja ohjelman suorituksessa [Lad03].

Staattinen poikkileikkauksellisuuden (*static crosscutting*) koostaminen ei muuta ohjelman käytöstä suorituksen aikana, vaan sen staattista rakennetta, kuten luokkia, rajapintoja ja aspekteja. Esimerkiksi luokkiin ja rajapintoihin voidaan lisätä uusia metodeja tai dataa, jotta luokkien tilan ja käytöksen avulla voidaan toteuttaa dynaamista poikkileikkauksellisuutta. Staattista poikkileikkauksellisuutta ovat myös käännöksen aikaiset varoitukset ja virheilmoitukset, joilla ohjataan moduulien toteuttamista.

Java-kielen AspectJ-laajennukset määrittelevät dynaamisen ja staattisen poikkileikkauksellisuuden koostesääntöjä (weaving rules). AspectJ-laajennuksissa käytetään join point-, pointcut- ja advice-määrittelyjä, jotka ovat ohjelmallisia koostesääntöjä. Nämä määrittelyt muodostavat moduuleja, jotka toteuttavat jonkin poikkileikkauksellisen tehtävän sovelluksessa [Lad03].

2.1 Join point

Join point on tunnistettava kohta ohjelman lähdekoodissa ja suorituksessa. Se voi olla metodin kutsu tai objektin jäsenen määrääminen (assignment). AspectJ-kielessä join

point on hyvin keskeinen, koska ne ovat ohjelman luokkatoteutuksen kohtia, joista poikkileikkaukselliset toiminnot kerätään. Esimerkiksi seuraavassa luokan määrittelyssä voi löytää kaksi join point –kohtaa: suoritettava `credit()`-metodi ja `_balance`-instanssijäsenen käyttö.

```
public class Account {
…
void credit(float amount) {
_balance += amount;
}}
```

Koska join point on jokin kohta ajettavan ohjelman dynaamisten kutsujen graafissa, sen käyttäytymistä voidaan muuttaa, esimerkiksi aspektin advice-lohkon ohjeella. Jokaista dynaamista join point-kohtaa vastaa jokin lähdekoodin nimetty osa. Erilaiset join point-kohtien lajit lähdekoodissa ja niitä vastaavat tunnisteet (signature) on lueteltu seuraavassa taulukossa. Näitä kohtia voidaan määrittää lähdekoodista joko tunnisteen perusteella suoraan tai reflektoimalla [Lad03].

Taulukko 1. Join poin lajit ja niiden tunnisteet

Join point-laji	Tunniste lähdekoodissa (signature)
Metodin suoritus	Metodi
Metodin kutsu	Metodi
Konstruktorin suoritus	Konstruktori
Kentän saanti (get)	Kenttä
Kentän asetus (set)	Kenttä
Advice-lohkon suoritus	-
Alustus	Vastaava konstruktori
Staattinen alustus	Tyypin nimi

Esi alustus	Vastaava konstruktori
Poikkeuksen käsittelijä	Poikkeuksen tyypin nimi
Poikkeuksen heitto	Poikkeuksen tyypin nimi
Synkronointi	Lukitusobjektin tyypin nimi

Taulukko 1 esittää join point –lajit sekä tunnisteet, joiden perusteella näitä kohtia voidaan määrittää lähdekoodista. Advice-lohko määritellään aspektissa, jossa se ohjeistaa join point –kohdan suoritusta [HiH04].

2.2 Pointcut

Pointcut on ohjelmarakenne, joka valitsemalla join point-kohdat ja kokoaa näiden kohtien kontekstin. Esimerkiksi pointcut voi valita metodin kutsun, ja voi myös siepata metodin kontekstin, kuten metodin kutsuman kohdeobjektin sekä metodin argumentit. Näin pointcut määrittelee koostesäännön (weaving rule), jonka joint point-kohta tilannekohtaisesti täyttää. Edellisen luvun 2.1 esimerkin jatkona voidaan kirjoittaa pointcut, joka sieppaa `credit`()-metodin suorituksen:

```
execution(void Account.credit(float))
```

AspectJ:n pointcut-mekanismi sisältää join point –kohtien valintaoperaattoreita, joita kutsutaan *primitiivisiksi pointcut määrittäjiksi* (*primitive pointcut designator, pcd*). Pointcut-lohkoissa nämä määrittäjät valitsevat sellaisia joinpoint-tyyppejä, joiden metadatan kuvaus täsmää määrittäjän argumentteihin, katso taulukko 1.

Nämä argumentit voivat perustua joinpoint-kohdan *nimeen* (*signature*) tai *tyypin kuvaukseen* (*type pattern*), joiden ilmaisemiseen voidaan käyttää jokerimerkkejä (wildcards). Pointcut-lohkon määrittäjät voivat tunnistaa myös toisen pointcut-lohkon määrittelyn joinpoint-kohdat ja liittää ne kontrollivirtaansa. Määrittäjä voi valita joinpoint-kohdat myös niiden suorituksen kontekstissa käytettävien objektien ja argumenttien perusteella. Näin ollen primitiiviset pointcut määrittäjät voidaan jakaa argumenttiensä perusteella kolmeen categoriaan, katso taulukot 2, 3 ja 4. [LaC03]

Taulukko 2. Tyypin kuvaukseen tai nimeen perustuva pointcut-lohkon määrittäjä

Määrittäjä (designator)	Valitut join point -kohdat
`call(Signature)`	metodien ja konstruktorien kutsut
`execution(Signature)`	metodien ja konstruktorien suoritus
`initialization(Signature)`	objektin alustuksen suoritus
`get(Signature)`	kentän viittaus
`set(Signature)`	kentän asetus
`staticinitialization(TypePattern)`	staattisen alustajan suoritus
`within(TypePattern)`	kaikki tyypin mukaiset join point-kohdat
`withincode(Signature)`	metodien ja konstruktorien määrittelyt

Taulukossa 2 luetellaan määrittäjät, jotka valitsevat joinpoint-kohdat nimen tai tyypin kuvauksen perusteella [LaC03].

Taulukko 3. Pointcut-perusteinen määrittäjä

Määrittäjä (designator)	Valitut join point -kohdat
`cflow(pointcut)`	kaikki suorituksessa käytetyt joinpoint-kohdat jotka pointcut määrittää
`cflowbelow(pointcut)`	kuten `cflow`, mutta ei sisällä argumenttinä oleva pointcut-lohkon join point-kohtia

Taulukossa 3 luetellaan määrittäjät, jotka valitsevat joinpoint-kohdat toisen pointcut-lohkon määrittelyn perusteella [LaC03].

Taulukko 4. Konteksti-perusteinen määrittäjä

Määrittäjä (designator)	Valitut join point -kohdat
`this(TypePattern or Id)`	joinpoint-kohdat joissa `this`-määreellä sidottu objekti on tietyn tyypin ilmentymä
`target(TypePattern or Id)`	joinpoint-kohdat joissa kutsu tai kenttäoperaatio kohdistuu tietyn tyyppisen objektin ilmentymään
`args(TypePattern or Id, ...)`	joinpoint-kohdat joiden argumentit täsmäävät määrittäjän argumetteihin

Taulukossa 4 luetellaan määrittäjät, jotka valitsevat joinpoint-kohdat niiden suorituksessa käytettävien objektien ja argumenttien perusteella. Nämä sisältävät myös suorituksen kontekstin [LaC03].

2.3 Advice

Advice –koodi suoritetaan siinä join point-kohdassa, joka pointcut-määrittelyssä on valittu. Advice-koodi voidaan suorittaa ennen join point-kohtaa tai sen jälkeen, se voi myös ohittaa kohdan tai korvata sen. Advice-koodi voi esimerkiksi kirjoittaa viestin ennen kuin eri moduleissa olevat join point -kohdat suoritetaan. Advice-määrittelyn runko muistuttaa metodin runkoa, sillä se kapseloi suoritettavan logiikan metodiin siirryttäessä. Esimerkiksi seuraava advice tulostaa viestin ennenkuin suorittaa Account-luokan `credit()`-metodin.

```
before() : execution(void Account.credit(float)) {
System.out.println(“ Veloitetaan”);}
```

Pointcut ja advice yhdessä muodostavat dynaamisen poikkileikkaussäännön. Pointcut tunnistaa vaaditun join point –kohdan, ja advice-lohkon ohjeet täydentävät join point –kohtien toimintoja. Ohjeita voidaan määritellä suorituksensa ja käyttötarkoituksensa mukaisesti seuraavasti:

- `before()`-advice suoritetaan ennenkuin ohjelman suoritus etenee käsiteltävänä olevaan join point -kohtaan. Esimerkiksi:

```
before(): move() {
     System.out.print(”lähtee”);}
```

- `after()returning` ja `after()throwing` –advice suoritetaan, kun join point -kohta on palauttanut jotain tai heittänyt poikkeuksen. Pelkkä `after()`-advice voidaan suorittaa molemmissa tapauksissa. Esimerkki palautuksesta:

```
after(FigureElement fe, int x, int y) returning:
   call (void FigureElement.setXY(int, int))
   && target(fe)
   && args(x,y) {
   System.out.println(fe + “ moved to (“ + x +
“,”+y+”)”); }
```

- `around()`-advice suoritetaan käsiteltävänä olevan join point -kohdan sijaan. Join-point-kohdan alunperäistä toiminnallisuutta voidaan kutsua käyttämällä `proceed()` –kutsua [HiH04].

Esittely (*Introduction*) on staattinen poikkileikkaava käsky, joka esittelee muutoksia sovelluksen luokkiin, rajapintoihin ja aspekteihin. Se saa aikaan staattisia muutoksia moduuleihin, mutta ei vaikuta suoraan niiden käytökseen. Esimerkiksi luokkaan voidaan lisätä metodi tai kenttä. Esimerkiksi seuraavassa esittely ilmoittaa, että Account-luokka toteuttaa BankingEntity-rajapinnan:

```
declare parents: Account implements BankingEntity;
```

Käännöksen aikaiset ilmoitukset (*compile-time declarations*) ovat staattisia poikkileikkaavia ohjeita, joilla voidaan esimerkiksi ehkäistä väärää koodaustapaa. Esimerkiksi voidaan ilmoittaa, että on virhe kutsua Abstract Window Toolkit (AWT)-koodia

EJB:stä. Seuraava käännöksen aikainen ilmoitus saa kääntäjän varoittamaan, jos jostain järjestelmän osasta kutsutaan Persistence-luokan `save`()-metodia. Tällöin `call()`-pointcut sieppaa metodin kutsun.

```
declare warning : call(void Persistence.save(Object))
:  “  Consider using Persistence.saveOptimized()  “ ;
```

Aspekti (*aspect*) on yhtä keskeinen yksikkö AspectJ-kielessä kuin luokka on Java-kielessä. Aspektin koodi ilmaisee sekä dynaamisen että staattisen poikkileikkaavuuden koostesäännöt. Aspekti yhdistää pointcut- ja advice-määrittelyt, sekä ilmoitukset ja esittelyt. Näiden AspectJ-elementtien lisäksi aspekti voi sisältää dataa, metodeja ja sisäluokan jäseniä, samalla tavalla kuten java-luokkakin. Edellä olevien esimerkkejä käyttäen voidaan kirjoittaa seuraavanlainen aspekti:

```
public aspect EsimerkkiAspekti {
   before() : execution(void Account.credit(float)) {
   System.out.println(“ Tulostettava viesti”);
   }
   declare parents: Account implements BankingEntity;
   declare warning: call(void Persistence.save(Object))
     : “Consider using Persistence.saveOptimized()“ ;
}
```

Aspektissa voidaan määritellä sisäluokkia samoin kuin luokissa sisäaspekteja. Staattinen aspekti voi esitellä luokalle uusia jäseniä. Aspektin instanssi voi ohjeistaa (advice) sekä luokan että aspektin instanssin jäseniä [Ken99].

3 SUUNNITTELUMALLIT

Tässä luvussa tutkitaan Worker-objektin luonti -, Madonreikä -, Poikkeuksen esittely- ja Osapuoli –suunnittelumalleja. Malleja käytetään tässä raportissa kehitettävän sovelluksen toteutuksessa. Suunnittelumallien lähdekoodit ovat raportin sähköisessä liitteessä.

3.1 Worker-objektin luonti - suunnittelumalli

Worker-objekti on luokanilmentymä, joka kapseloi metodin niin, että metodia voidaan käyttää kuten objektia. Tuota metodia kutsutaan Worker-metodiksi. Worker-objektia voidaan kutsua, välittää toiseen paikkaan ja varastoida. Yleinen tapa toteuttaa tällainen Worker-objekti on `Runnable` –rajapinnan käyttö, jonka toteuttava luokka sisältää `run()`-metodin joka puolestaan kutsuu Worker-metodia. Worker-objektin suoritus (execution) edelleen suorittaa Worker-metodin.

Tätä suunnittelumallia voidaan käyttää sieppamaan operaatioita pointcut-määrittelyjen kanssa. Voidaan myös automaattisesti generoida Worker-objekteja jotka kapseloivat nuo operaatiot. Näitä objekteja voidaan välittää sopiville metodeille suoritettavaksi.

Tässä esiteltävässä mallissa käytetään aspektia luomaan automaattisesti anonyymien worker-luokkien objekteja. Nämä objektit ovat Worker-objekteja. Aspektiin kirjoitetaan pointcut, joka sieppaa kaikki ne join point-kohdat, jotka on ohjattava Worker-objekteille. Seuraavaksi kirjoitetaan advice joka suorittaa nämä siepatut join point-kohdat anonyymien worker-luokkien `run()`-metodissa.

Tavallisesti around advice –lohkossa suoraan kutsuttu `proceed()` –metodi suorittaa siepatut join point –kohdat. Tässä mallissa `proceed()` –metodia kutsuu around advice joka on `Runnable`-rajapinnan implementoivan anonyymin luokan sisältämässä `run()`-metodissa, ja saa Worker-objektin palautteena. Tämän objektin `run()`-metodia voidaan käyttää esimerkiksi säikeissä suorittamaan siepatut join point –kohdat.

Seuraavaa mallin kaavaa voi käyttää implementointiin. Ensin kirjoitetaan pointcut joka sieppaa halutut join point -kohdat. Koska kaavassa ei tarvita kontekstia, voidaan käyttää nimettyä tai anonyymiä pointcut-määrittelyä. Seuraavaksi pointcut ohjeistetaan kirjoittamalla siihen around advice, jonka rungossa luodaan anonyymin luokan Worker-objekti. Se luodaan kutsumalla `run()` -metodissa `proceed()` -metodia sen sijaan, että kutsuttaisiin jotain tiettyä metodia. Tämän jälkeen Worker-bjektia voidaan käyttää halutulla tavalla.

```
void around() : <pointcut> {
   Runnable worker = new Runnable () {
      public void run() {
               proceed();
}}
... Lähetä worker-objekti johonkin jonoon suoritettavaksi,
... tai lähetä worker-objekti jollekin alijärjestelmälle
    suoritettavaksi,
... tai kutsu pelkästään run() –metodia suoraan.
}
```

Esimerkki kaavan käyttämisestä [Lad03]:

Liitteessä 2 on uudelleen käytettävän abstraktin `Asynchronous-ExecutionAspect` -aspektin lähdekoodi. Aspekti suorittaa erillisessä säikeessä kaikki join point -kohdat, jotka `asyncOperations()` –niminen pointcut määrittelee. Aspekti sisältää abstraktin pointcut-esittelyn, sekä around advice -ohjeen sille.

Aspektin advice -runko luo anonyymin luokan objektin, joka implentoi `Runnable`-rajapinnan. `Run()`-metodi kutsuu proceed()-metodia, joka suorittaa join point –kohdat. `worker` –objekti suorittaa ohjeistetut joint point –kohtien operaatiot. Advice käyttää sitä uuden säikeen luomiseen ja aspekti käynnistää luodun säikeen.

`AsynchronousExecutionAspect` -aspektissa esitelty abstrakti pointcut määritellään konkreettisessa aliaspektissa.

Liitteen 2 `SystemAsynchronousExecutionAspect`-aspekti sisältää tarvittavan pointcut –määrittelyn. Aspektissa määritellään ohjelmallisesti varmuuskopion teko ja nouto välimuistista tapahtumaan eri säikeissä. `SystemAsynchronousExecutionAspect` mahdollistaa `CachePreFetcher`- ja `Projectsaver`- luokkien metodien asynkronisen suorituksen. Näiden luokkien ja `Test`-luokan lähdekoodi on liitteessä 2.

Nämä voidaan kääntää ajc-komennolla, joka vastaa Java-kielen javac-komentoa, sekä ajaa normaalisti java-komennolla:

```
> ajc CachePrefetcher.java ProjectSaver.java Test.java
AsynchronousExecutionAspect.java SystemAsynchronousExe-
cutionAspect.java

> java Test
```

Tulosteena saadaan `CachePrefetcher` ja `ProjectSaver` –luokkien viestit ja säikeiden nimet.

3.1.1 Worker-metodin palauttama arvo

Edellisen kappaleen esimerkkin ohjatut kutsut voivat palauttaa arvon kutsujalle. Tällöin `proceed()` palauttaa metodin arvon, kun operaatio on suoritettu loppuun. Tämän arvon voi pitää Worker-objektissa tai palauttaa around advice –lohkosta. Jotta arvo olisi järkevä, kutsujan pitää odottaa kunnes Worker-objektin suoritus on päättynyt. Edellä kerrotussa asynkronisessa esimerkissä kutsuvan säikeen palauttama arvo ei vastaa operaation arvoa, koska säie palauttaa arvon välittömästi ja operaatio voidaan suorittaa myöhemmin.

Paluuarvo voidaan ottaa talteen yleisellä tavalla, esimerkiksi kirjoittamalla abstrakti luokka `RunnableWithReturn`, joka implementoi `Runnable` –rajapinnan. `RunnableWithReturn`-luokkaa implementoivissa luokkissa `run()` –metodin pitää aset-

taa `_returnValue`-jäsen `proceed()`-metodin palautusarvoksi, koska se on suoritetun join point –kohdan palautusarvo. RunnableWithReturn-luokan lähdekoodi on liitteessä 2.

Tätä `RunnableWithReturn`-luokkaa voidaan käyttää perusluokkana anonyymille luokalle advice-lohkossa, kuten liitteen 2 `SynchronousExecutionAspect`-aspektissa. Around advice –lohkon `proceed()`-metodin paluuarvo on asetettu `_returnValue`-jäsenen arvoksi. `proceed()`-metodin palauttama objekti on käärre-objekti (*wrapper*), joten paluuarvon tyypistä ei tarvitse huolehtia.

Worker-objektin luonti -suunnittelumallia voi hyödyntää erilaisissa tilanteissa, kuten auktorisoinnissa ja säieturvallisuudessa myös Swing-komponenttien kanssa. Worker-objektia voi joissakin tilanteissa käyttää heti luonnin jälkeen, koska se ilmentää silloin kontekstia. Näin tehdään esimerkiksi tämän raportin transaktionhallinnan esimerkissä.

3.1.2 Worker-objektin kontekstin hallinta

Worker-objektin luonti –suunnittelumallin soveltamisessa voi olla tarpeellista koota pointcut-kohdan konteksti talteen muuta käyttöä varten. Muuttumattoman kontekstin saa talteen ja välitettäväksi advice-lohkossa `proceed()` –metodin avulla seuraavasti:

```
void around([context]) : <pointcut> {
    Runnable worker = new Runnable() {
      Proceed([context]);
 }}
   ... voit käyttää worker-objektia
}
```

3.2 Madonreikä suunnittelumalli

Madonreikä suunnittelumalli (Wormhole pattern) välittää kutsujan informaation kontekstin kutsunvastaanottajan käytettäväksi ilman, että informaatiota tarvitsisi välittää *kontrollivirrassa (control flow)* kullekin metodille parametrijoukkona. Esimerkiksi auktorisoinnissa järjestelmän usean metodin on tiedettävä, kuka niitä kutsui, jotta ne voisivat päätellä onko kutsujalla lupa tähän operaatioon. Madonreikäsuunnittelumalli sallii käyttää kutsuvaa objektia ja sen kontekstia tämän tiedon hankkimiseen.

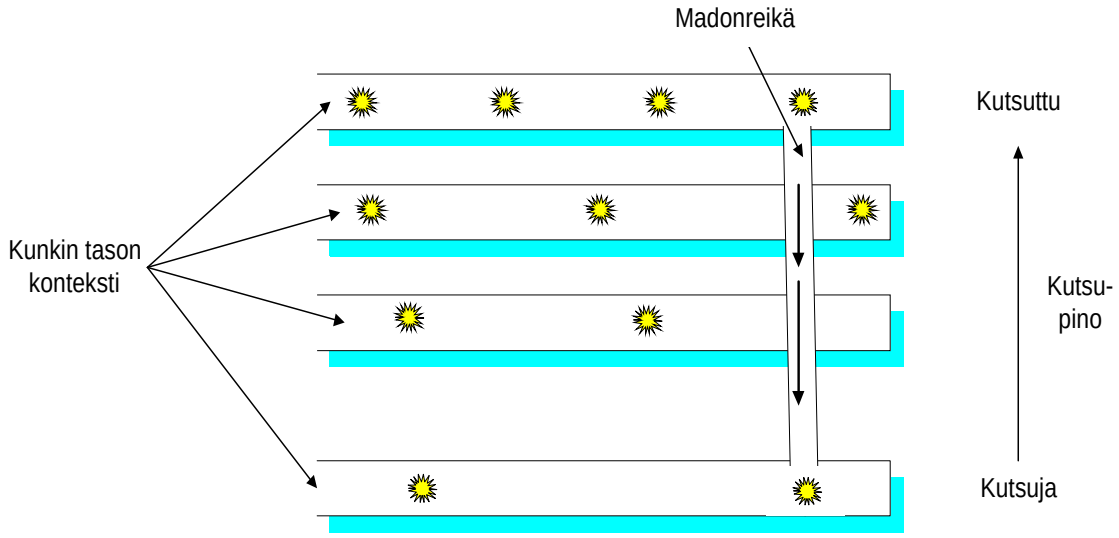
Malli luo suoran reitin kahden tason välille kutsupinossa (call stack), jolloin vältytään kaikkien tasojen läpikäymiseltä. Tämä säästää kutsuketjun modifioinnilta kun halutaan välittää kontekstin informaatiota, se myös ehkäisee niin sanottua API-saastetta.

Jos AspectJ ei ole käytössä, voi monisäikeisessä ympäristössä välittää kutsujan kontekstin kahdella tavalla. Voit välittää kontekstin informaation lisäparametrien avulla, mikä aiheuttaa API –saastetta, koska kaikille metodeille pitää määritellä ylimääräisiä parametrejä. Toinen tapa on käyttää kutsujan luomia säiekohtaisia varastoja kontekstin asettamiseksi ja käyttämiseksi. Tässäkin tavassa useat moduulit ovat mukana informaatiota välittävässä logiikassa.

3.2.1 Madonreikä suunnittelumallin periaate

Malli voidaan luoda määrittelemällä kaksi pointcut-kohtaa: yksi kutsujalle ja yksi kutsun vastaanottajalle. Kutsuja kokoaa kontekstin mikä välitetään tässä madonreiässä. Madonreikä on määriteltävä vastaanottajan join point-kohtien suorituskohdista kutsujan join point-kohtien kontrollivirtaan.

Kuva 4 esittää kutsupinoa tasoittain. Normaalisti konteksti voidaan siirtää taso kerrallaan, kunnes haluttu taso saavutetaan. Madonreikä mallissa tehdään polku suoraan haluttujen tasojen välille.



Kuva 4. Madonreikämalli välittää kutsuttavan objektin kontekstin kutsupinon tasojen ohi suoraan kutsujalle. Vaakatasot esittävät kutsun syvyyttä.

3.2.2 Madonreikämallin kaava

Mallin kirjoittamiseen voidaan käyttää seuraavaa kaavaa, jossa määritellään kutsuvan tilan pointcut, joka kokoaa siihen liittyvän kontekstin. Samoin määritellään kutsun vastaanottajan tilan pointcut. Koottu konteksti voi molemmissa tiloissa olla kohdeobjekti tai muu parametri tarvittaville metodeille ja niiden suoritus.

Seuraavaksi luodaan pointcut joka toteuttaa madonreiän. Sekä kutsujan että kutsun vastaanottajan pointcut-lohkot sieppaavat omassa tilassaan join point –määrittelyistä kontekstinsa. `pointcut wormhole` -lohkossa kutsun vastaanottajan `calleeSpace()` pointcut määritellään samaan kontrollivirtaan kutsujan `callerSpace()` pointcut-kutsujen kanssa. Näin saadaan molempien join point –määrittelyjen konteksti käyttöön.

Tätä mallia käytetään tämän raportin transaktio-esimerkissä myöhemmin.

Madonreikämallin kaava:

```
  public aspect WormholeAspect {
     pointcut callerSpace(<caller context>)
         : <caller pointcut>;
     pointcut calleeSpace(<callee context>)
         : <callee pointcut>;
     pointcut wormhole(<caller context>,
                                  <callee context>)
         : <cflow(callerSpace(<caller context>))
           && <calleeSpace(<callee context>);
  // advices to wormhole
  around (<caller context>, <callee context>)
    : wormhole(<caller context>, <callee context>){
         … advice body
  }
}
```

Seuraava `AccountTransactionAspect` -aspekti on yksinkertainen esimerkki tämän mallin käytöstä. Aspekti luo madonreiän transaktion osapuolten kuten tilinhaltijan, kassanhoitajan tai verkkopankin ja todellisten operaatioiden välille. AccountTransactionAspect.java:

```
public aspect AccountTransactionAspect {
// 1. transaktiojärjestelmän operaatioiden käyttö
  pointcut transactionSystemUsage(TransactionSystem ts)
       : execution(*TransactionSystem.*(..))&&
this(ts);
// 2. tilioperaatioiden käyttö
  pointcut accountTransactions (Account account,
                                  float amount)
       : this(account) && args(amount)
        && (execution(public * Account.credit(float))
        || (execution(public * Account.debit(float)));
// 3. luo madonreiän kohtien 1 ja 2 välille
   pointcut wormhole (TransactionSystem ts,
                      Account account,  float amount)
       : cflow(transactionSystemUsage (ts))
        && accountTransactions (account, amount);
// 4. käyttää madonreiän mahdollistamaa kontekstia
   before (TransactionSystem ts,
               Account account,  float amount) returning
        : wormhole(ts, account, amount)  {
// esim. kirjoitetaan loki transaktion operaatioiden
// etenemisestä, authorisoinnista jne.
}}  // AccountTransactionAspect.java päättyy
```

Kohtien selitykset:

Kohta 1: `transactionSystemUsage()`-poincut sieppaa kaikki suoritettavat join point -kohdat `TransactionSystem`-objektiin. Näin poincut voi kerätä transaktion kontekstin ja säilyttää sen tässä objektissa.

Kohta 2: `accountTransaction()`-pointcut sieppaa Account -luokan `credit()`- ja `debit()` -metodien suorituksen. Tämä pointcut säilyttää tiliä ja summaa koskevan kontekstin.

Kohta 3: `wormhole()` -pointcut luo madonreiän transaktiojärjestelmän operaatioiden ja tililuokan operaatioiden välille sieppaamalla kaikki ne join point –määrittelyt kohdasta 2 jotka tapahtuvat `transactionSystemUsage()` kontrollivirrassa. Näin `wormhole()` -pointcut asettaa saataville kohdissa 1 ja 2 siepatun kontekstin.

Kohta 4: `before` advice voi nyt käyttää kontekstia, jonka `wormhole()`tarjoaa. Tämä advice voi tunnistaa tilin, summan, sekä tilinkäytöstä vastuussa olevan transaktiojärjestelmän.

3.3 Poikkeuksien käsittely

AspectJ-kielinen advice ei voi heittää tarkistettua poikkeusta, elleivät sen join point –kohdat voi heittää samaa poikkeusta. Tarkistettu poikkeus laajentaa suoraan tai epäsuorasti `Exception` –luokan, sen sijaan että laajentaisi `RuntimeException` –luokan. Usein kuitenkin järjestelmän tehtävien koostaminen (weaving) vaatii, että aspekti käsittelee tarkistettuja poikkeuksia. Esimerkiksi JDBC tietokantayhteyksissä tai JAAS autentikoinnissa aspektit voivat käyttää yleisiä ohjelmakirjastoja ja palveluja. Näistä heitetyt poikkeukset ovat yleensä tarkistettuja poikkeuksia, joita alusta pystyy käsittelemään.

Lisäksi tietyn tyyppiset poikkileikkaukselliset tehtävät, kuten virheistä toipuminen ja transaktionhallinta vaativat, että aspekti sieppaa kaikentyyppiset poikkeukset joita suoritettavat joint point –kohdat heittävät. Näitä varten aspektin catch-lohkoon on määriteltävä poikkeuksen käsittelyn logiikka. Tilannetta hankaloittaa vielä se, että aspektit ovat uudelleenkäytettäviä, eivätkä ne osaa käsitellä liiketoimintakohtaisia poikkeuksia.

Poikkeuksien käsittely -malli perustuu siihen, että suorituslogiikassa syntyvä alkuperäinen, tarkistettu poikkeus poimitaan ja heitetään uusi ajonaikainen poikkeus, joka käärii alkuperäisen poikkeuksen. Kun ajonaikainen poikkeus tapahtuu, siitä ilmoitetaan poikkeuksena ylemmän tason kutsujalle. Sen on myös voitava käsitellä poikkeus, vaikka se ei olisi tietoinen järjestelmän aspektien määrittelyistä.

Mallin mukaisesti voi käsitellä liiketoimintakohtaisia poikkeuksia uudelleenkäytettävissä aspekteissa. Toisin sanoen on luotava tehtäväkohtainen (concern-specific) ajonaikainen poikkeus, joka voi asettaa tarkistetun poikkeuksen aiheuttajakseen. Asettamisen voi tehdä joko välittämällä poimitun poikkeuksen ajonaikaisenpoikkeuksen konstruktorille tai käyttämällä `Exception`-luokan `initCause()` –metodia. Esimerkiksi implementoitaessa pysyvyysaspektia JDBC:tä käyttäen, voi heittää `PersistenceRuntimeException` -poikkeuksen, joka käärii alkuperäisen tarkistetun `SQLException` -poikkeuksen.

3.3.1 Tehtäväkohtaisen poikkeuksen käsittely aspektissa

Esimerkkinä liitteen 2A `ConcernAspect` –aspekti toteuttaa poikkeuksen käsittelykaavan. Kaava esittää mallin, jossa tehtäväkohtainen `ConcernCheckedException` –tyyppinen poikkeus siepataan ja kääritään `ConcernRuntimeException` –tyyppiseksi poikkeukseksi. Tämä säilyttää alkuperäisen poikkeuksen niin, että sen vastaanottaja voi nähdä alkuperäisen poikkeuksen tyypin. Mallia käytettäessä sen käsitteiden nimet korvataan tehtäväkohtaisilla käsitteillä.

Aspektissa käytetään `ConcernRuntimeException`-luokkaa, jonka konstruktori saa alkuperäisen heitetyn poikkeuksen parametrikseen. Tästä parametristä ilmenee

poikkeuksen syy, mikäli kutsuja tarvitsee sitä. Aspektin before advice sieppaa alkuperäisen tarkistetun poikkeuksen, käärii sen tarkistamattomaksi poikkeukseksi ja heittää sen. Tässä advice saa alkuperäisen poikkeuksen `void concernLogic()` –metodilta. ConcernAspect-aspektin ja mallin muu lähdekoodi on testausluokkineen liitteessä 2A.

3.3.2 Liiketoimintokohtaisen poikkeuksen käsittely aspektissa

Samaa poikkeuksen esittely –mallia voi käyttää tehtäväkohtaisten tarkistettujen poikkeusten lisäksi myös liiketoimintokohtaisten tarkistettujen poikkeusten käsittelyyn. Joissain tilanteissa jonkin aspektin advice-lohkon on kyettävä poimimaan kaikki poikkeukset. Tällöin aspektin pitää pystyä heittämään edelleen liiketoimintokohtaisia poikkeuksia eli kun poikkeus on siepattu, sitä pitää pystyä käsittelemään jollain tavoin.

Edellä esitettyä `ConcernAspect` –aspektia voidaan muuttaa niin, että se ottaa kiinni kaikki poikkeukset, jotka sen pointcut sieppaa. Lisätään aspektiin around advice, joka käsittelee siepatut operaatiot try/catch-lohkossa. Kun catch-lohko on suoritetaan, se muodostaa tehtäväkohtaisen virheenkäsittelyn ja heittää kutsujalle uuden `ConcernRuntimeException` –tyyppisen poikkeuksen, joka on kääre alkuperäiselle tarkistetulle poikkeukselle. Muutettu ConcernAspect-aspektin ja mallin muu lähdekoodi on testausluokkineen liitteessä 2B.

3.3.3 Poikkeuksen taltiointi aspektissa

Liiketoimintometodien suoritukseen tarvitaan liiketoimintakohtaisten poikkeusten käsittelyä. Kun kaikki poikkeukset siepataan ja kääritään ajonaikaisiksi poikkeuksiksi, kutsujan on saatava käyttöönsä poikkeuksen todellinen syy, jotta sitä voidaan käyttää toteutettavan sovelluksen tarpeiden mukaan. Siepatusta poikkeuksesta syy voidaan ottaa talteen ja lähettää se edelleen kutsujalle.

Edellisen kappaleen esimerkkiin voidaan lisätä `PreserveBusinesException` –aspekti, joka ottaa talteen ajonaikaisen poikkeuksen syyn. Tämä aspekti ei ole uudelleenkäytettävä vaan se on toteutettava järjestelmäkohtaisesti. Aspektin lähdekoodi on käyttöesimerkkeineen liitteessä 2C.

`PreserveBusinessException`-aspektin toiminta: kun jokin metodi - joka pystyy heittämään liiketoimintakohtaisen poikkeuksen - on heittänyt `ConcernRuntimeException`-tyyppisen poikkeuksen, kutsutaan `PreserveBusinessException`-aspektissa after advice, joka voi myös heittää liiketoimintakohtaisen poikkeuksen. Advice-määrittelyn rungossa tarkistetaan onko ajonaikaisen poikkeuksen aiheuttaja sellainen liiketoimintapoikkeus joka pitäisi säilyttää. Jos on, sen tyyppi pakotetaan `BusinessException` –tyypiksi ja poikkeuksen syy heitetään edelleen kutsujalle.`PreserveBusinessException`-aspektin lähdekoodi on liitteessä 2 C.

3.4 Osallistuja-suunnittelumalli

Monilla järjestelmän operaatioilla on yhteisiä erityispiirteitä, kuten yksinkertaiset transaktio-ominaisuudet, methodin kutsujen ajallinen kesto, IO-toiminnon ominaisuudet, etäkäytön ominaisuudet, tai muu ominaisuus jonka perusteella jotkin operaatiot ovat osallisena johonkin järjestelmään laajalti vaikuttavaan tehtävään. Koska nämä operaatiot ovat hajotettuna useissa moduuleissa, operaatioiden käytöksen laajentaminen ja muuttaminen on poikkileikkauksellinen tehtävä. Osallistuja-suunnittelumalli tarjoaa erään tavan modularisoida tällaista erityistuntomerkkeihin pohjautuvaa poikkileikkauksellista tehtävää. Malli auttaa sieppaamaan join point –kohdat erityispiirteiden perusteella silloin kun nimeen perustuva sieppaus ei riitä.

Kun halutaan hallita järjestelmän hitaita operaatioita ja niiden aiheuttamaa odotusaikaa, voidaan näitä operaatioita suorittavia metodeja hakea metodien nimeen (signature) perustuvilla pointcut –määrittelyillä. Esimerkiksi kun metodi heittää `IOException` –poikkeuksen, voidaan päätellä että se suorittaa hitaan IO-operaation. Näin voidaan tun-

nistaa joitain hitaita metodeja. Myös metodin nimestä voidaan päätellä sen ominaisuuksia. Esimerkiksi kun nimi on `set` -alkuinen, kyseessä on todennäköisesti tilaan vaikuttava operaatio. Näin ei kuitenkaan voida määritellä kaikkien operaatioiden erityistuntomerkkejä. Operaatiohan voi olla esimerkiksi hidas, koska se suorittaa monimutkaisia laskutoimituksia, eikä metodin nimi paljasta sen ominaisuuksia. Siksi ominaisuuksiin perustuvaa pointcut-määrittelyä ei voida koostaa reflektoimalla jokerimerkeillä (wildcards).

Jotta tällaiset erityispiirteitä omaavat join point –kohdat voidaan siepata pointcut-määrittelyyn, tarvitaan niitä toteutettavien luokkien yhteistyötä. Eräs mahdollisuus on täydentää implementointia metadatalla, joka ilmaisee erityispiirteet silloin kun se ei ole johdettavissa nimeämiskäytännöstä. Tätä ei kuitenkaan voida käyttää vielä AspectJ-kielisessä toteutuksessa.

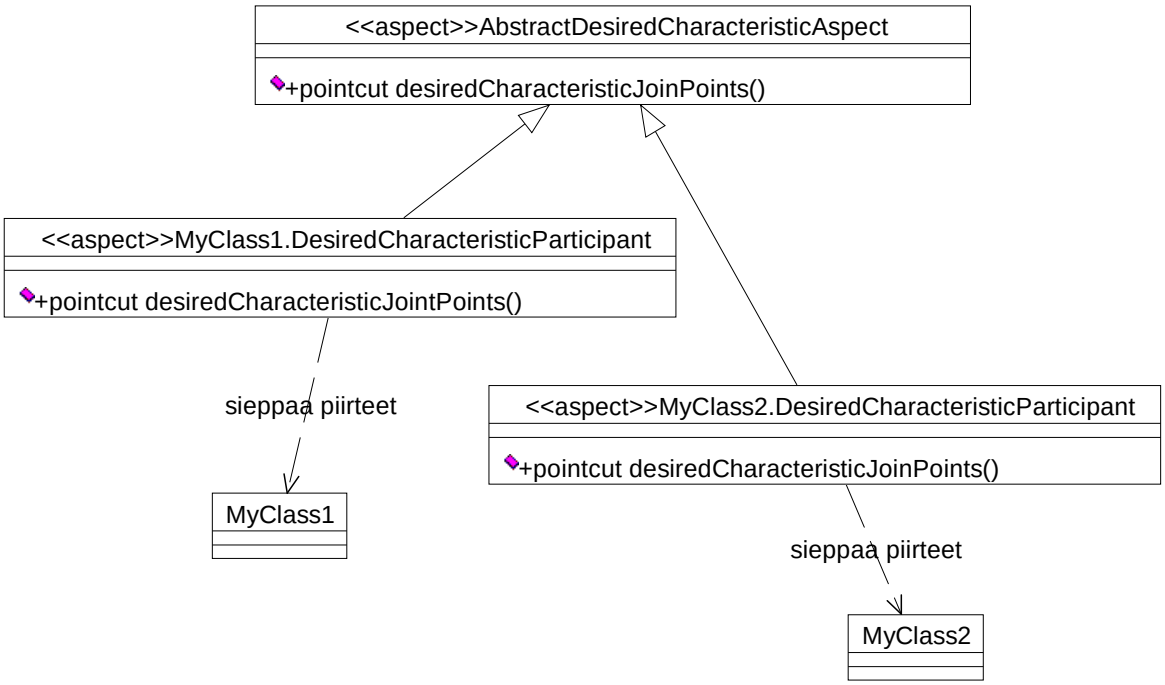
AspectJ-toteutuksessa *Osallistuja-suunnittelumalli* (*The Participant pattern*) auttaa sieppaamaan metodit niiden erityispiirteiden perusteella. Malli kuitenkin vaatii ydintoteutuksen modifiointia, joten on mahdollista että osa muutoksia tarvitsevista operaatioista jää tunnistamatta. Siksi usein on turvallisempaa käyttää vakiintuneita nimeen tai ominaisuuteen perustuvia pointcut-määrittelyjä.

Osallistuja-suunnittelumallissa ohjelmaluokkien sisältämä poincut viittaa edelleen joihinkin erityispiirteisiin. Sen sijaan että poincut-määritelmä sisällytettäisiin kuhunkin luokkaan ja käytettäisiin aspektien advice-lohkossa, luokkiin itseensä määritellään aliaspekti (subaspect). Tämä aliaspekti laajentaa ohjeistavaa aspektia (advising aspect) ja sisältää pointcut-määritelmän. Tavallaan tämä määritelmä kääntää aspektin ja luokan roolit päinvastaisiksi. Sen sijaan että tehtäisiin aspektit tietoisiksi luokista ja metodeista, nyt tehdään luokat tietoisiksi aspekteista.

Malli rakentuu seuraavasti. Ensin kirjoitetaan abstrakti aspekti, joka sisältää yhden tai useampia abstrakteja pointcut-määrittelyjä, jotka viittaavat haluttuja erityispiirteitä omaaviin join point –kohtiin. Nämä pointcut-määrittelyt muodostavat eräänlaisen ’aspektuaalisen rajapinnan’. Aspektin advice-lohkossa jokaista pointcut-määrittelyä –tai

niiden kompinaatioita - ohjeistetaan käyttäytymään halutulla tavalla. Tätä aspektia voidaan pitää kutsuvana aspektina (inviting aspect), jonka kutsuu muita osallistumaan advice-lohkossaan. Kutsu voi olla pelkkä yksittäinen kutsu, tai se voi sisältää valinnaisia ehtoja.

Jokaisen luokan, jota halutaan käyttää osallistujana, pitää sisältää konkreettinen aliaspekti, joka laajentaa kutsuvaa aspektia. Tämä aliaspekti toteuttaa luokassaan abstraktin pointcut-määrittelyn konkreettisesti. On huomattava että konkreettisen aliaspektin ei tarvitse olla luokan sisäaspekti, vaan se voi olla esimerkiksi vertaisaspekti (peer aspect). Aliaspekti on määriteltävä jokaiseen luokkaan, jota halutaan käyttää osallistujana yhteistyössä. Kuva 5 esittää Osallistuja suunnittelumallin tyypillistä rakennetta UML-kaaviona.



Kuva 5. Osallistuja-suunnittelumallin esimerkki UML notaationa

3.4.1 Osallistuja-suunnittelumallin kaava

Seuraava mallin kaava (template) sisältää abstraktin aspektin sekä kaksi luokkaa, joissa on sisäaspekti. Näissä sisäaspekteissa toteutetaan abstrakti pointcut-määrittely konkreettisesti.

AbstractDesiredCharcteristicAspect.java perustaa yhteistyön. Mallissa on käytetty around advice -määrittelyä, yhtä hyvin voi olla myös before- tai after advice.

```
abstract aspect AbstractDesiredCharcteristicAspect {
  public abstract pointcut desiredCharacteristicJoinPoints();
  Object around() : desiredCharacteristicJoinPoints() {
          // tähän advice koodi
          }
  }
```

Haluttu poikkileikkauksellinen käytös kirjoitetaan around advice -lohkossa, joka määrittelelee abstraktin pointcut -esittelyn.

MyClass1.java on osapuolena yhteistyössä. Luokkaan pitää määritellä sisäaspekti, joka on `AbstractDesiredCharcteristicAspect`-perusaspektin aliaspekti. Abstrakti `desiredCharacteristicJoinPoints` -pointcut toteutetaan tässä MyClass-luokassa. Aliaspekti julistaa, että tämän luokan metodeilla haluttuja piirteitä, joten ne voivat olla osapuolena perusaspektin tarjoamassa yhteistyössä.

```
  public class MyClass1 {
  // MyClass1-luokan koodi
  public static aspect DesiredCharacteristicParticipant
          extends AbstractDesiredCharcteristicAspect {
          public pointcut desiredCharacteristicJoinPoints() :
          call(* MyClass1. desiredCharacteristicMethod1())
          || call(* MyClass1. desiredCharacteristicMethod2())
```

```
        /* || jne...*/;
}}
```

Samaan tapaan voidaan määritellä muita yhteistyöhön osallistuvia luokkia. Osallistujamallissa luokat voidaan täsmällisesti määritellä poikkileikkauksellisen tehtävän toteuttajiksi yhteistyön osapuolina. Samoin on mahdollista määritellä osapuoleksi luokkahierarkia tai pakkaus. Silloin aliaspekti on määritelty luokan ulkopuolelle, ja tässä osapuolen aliaspektissa on pointcut-lohkossa ylläpidettävä listaa tarvittavista halutunpiirteisistä metodeista. Listan toteutus voi perustua myös metodien nimen mukaiseen reflektiiviseen valintaan.

4 SÄIETURVALLISUUS

Säieturvallisuus vaatii järjestelmän oikean käyttäytymisen ylläpitoa tilanteessa, jolloin järjestelmässä useat säikeet käyttävät sen tilaa. Koska säikeet vaikuttavat eri osissa sovellusta, niiden toteutus, muuttaminen ja testaus voi aiheuttaa ongelmia. Mikäli tässä ei onnistuta, järjestelmässä esiintyy käyttöliittymätasolla epämääräistä tulostumista, datan eheys on puutteellista ja järjestelmä voi hidastella tai lukittua kokonaan.

Näihin ongelmiin on käytettävissä ratkaisumalleja. Niissä määritellään objektien lukituksia jotta vältyttäisiin järjestelmän lukkiutumiselta ja pyritään tehokkaaseen moniajoon sekä välttämään ylikuormitus. Jokainen ongelma on kuitenkin analysoitava erikseen, jotta voidaan valita oikea ratkaisumalli.

Tässä raportissa esitetään kaksi AspectJ-kielistä ratkaisua. Swing-toteutuksen mallissa pyynnöt välitetään suojatuille objekteille määrätyssä säikeessä. Toisessa mallissa käytetään lukemisen ja kirjoittamisen lukitusta (read-write lock pattern).

4.1 Swing ja yhden säikeen sääntö

Swing, joka on hyvin yleinen Javan GUI -kirjasto, käyttää yhden säikeen turvallisuussääntöä. Se vaatii, että kaikkien Swing-komponenttien on käytettävä vain yhtä *tapahtumasäiettä* (*event-dispatching thread*). Rajoittamalla käytön vain yhteen määrättyyn säikeeseen, malli siirtää turvallisuus ongelmat pois komponentin toteutuksesta. Kun toinen säie tarvitsee Swing-komponentin palveluja, sen pitää pyytää tapahtumasäiettä suorittamaan operaatio, koska ainoastaan tapahtumasäie saa päivittää näkyviä komponentteja.

Yksinkertaisissa sovelluksissa, joissa ei ole käyttäjän luomia säikeitä, tämä sääntö toimii hyvin. Monimutkaisissa sovelluksissa, joissa *taustasäikeet* (*nonevent-dispatching thread*) käyttävät UI-komponentteja, sääntö on kuitenkin rajoittava tekijä. Esimerkiksi tilanne, jossa tapahtumasäie suorittaa verkkoon tai IO-keskeytyksiin liittyviä operaatioita tai tietokantatoimintoja, ja taustasäikeen pitäisi pystyä päivittämään käyttöliittymää palvelimen informaatiolla. Kun palvelimelle lähetetään pyyntö, jonka perusteella käyt-

töliittymä päivitetään, tapahtumasäie ei voi jäädä odottamaan vastausta, koska tällöin koko käyttöliittymä lukittuu kunnes palvelin vastaa. Jotta näin ei kävisi, jonkun toisen säikeen on odotettava, jotta käyttöliittymää voidaan päivittää. Toisaalta päivitystä ei voi jättää taustasäikeen tehtäväksi, koska näyttö jää epämääräiseen tilaan.

Ratkaisuksi tähän ongelmaan Swing sallii tapahtumasäikeen käyttämisen muiden säikeiden pyyntöjen välittämiseen. Säikeiltä voi jättää operaation suorituspyyntöjä tapahtumasäikeelle käyttämällä `EventQueue.invokeLater()` tai `EventQueue.invokeAndWait()`-kutsuja. Molemmissa välitetään `Runnable` –objekti, jonka `run()` –metodi suorittaa aiotun operaation. Tässä ratkaisun hankaluudeksi tulee `Runnable` –luokkan laajennusluokkien kirjoittaminen jokaista ei-AWT-säikeestä kutsuttavaa metodia varten, sekä näiden luokkien käyttäminen suorien kutsujen sijaan. On myös varmistettava luokan on yhdessä `EventQueue` -luokan kanssa korvattava kaikki ei-AWT-säikeestä tulevat kutsut.

4.1.1 Perinteinen ratkaisu

Liitteessä 3 `Test`-luokan lähdekoodi esittää säieturvallisuuden perinteisen ratkaisun. `Test`-luokka toteuttaa käyttöliittymän *ikkunan* (*frame*). Tällöin yhden säikeen sääntö astuu voimaan.

Tämän jälkeen tehdyt metodin kutsut kääritään anonyymeihin `Runnable` –luokan ilmentymiin. Tämän luokan `run()` –metodi kutsuu suoritettavia metodeja. Anonyymin luokkan sijaan voisi yhtä hyvin käyttää nimettyjä luokkia. Ensimmäinen metodi tulostaa ikkunaan pienen taulukon. Metodi suoritetaan asynkronisesti, joten sille kutsutaan `EventQueue.invokeLater()`-metodia. Toinen metodi tulostaa OK-viestipainikkeen. Tämä metodi suoritetaan asynkronisesti, joten sille kutsutaan `EventQueue.invokeAndWait()`-metodia.

Seuraavaksi välitetään reititysluokan (`Runnable`) ilmentymä. `invokeLater()`-metodia käytetään jos kutsu pitää suorittaa jumiuttamatta kutsujaa. `invokeAnd-`

`Wait()`-metodia käytetään silloin, kun kutsujan pitää jäädä odottamaan operaation suoritusta.

Test-luokan kohtien selitykset:

Kohta 1: Tällä kohtaa käyttöliittymä toteutetaan eli kehys näkyy kuvaruudulla. Taustasäikeitä voi kutsua vielä ennen tätä kohtaa, koska AWT-säie ei käytä komponentteja. Pääsäie on ainoa säie, joka päivittää käyttöliittymäkomponentin tilaa.

Kun käyttöliittymä on toteutettu, AWT-säie lukee ja päivittää sovelluksen tilaa ja koska Swing-luokat eivät tarjoa mitään poissulkevaa käytön suojausta, on pyydettävä tapahtumasäiettä suorittamaan operaatio kutsuvan säikeen puolesta. Näin taataan, että vain tapahtumasäie hallitsee käyttöliittymän komponentteja.

Kohta 2: Pyyntö taulukon kentän arvon asettamisesta voidaan tehdä asynkronisesti, koska ei ole vaatimusta siitä milloin tämä operaatio on suoritettu loppuun. Ensin pyyntö kääritään `Runnable`-luokan toteuttavaan anonyymiin luokkaan, jonka `run()` –metodissa alkuperäinen operaatio jatkuu. Tässä joudutaan antamaan paikallisille muuttujille `final` –määre, koska paikallissa luokissa käytetyille muuttujille Java pakottaa antamaan tämän määreen.

Koska operaatioille ei ole vaadittu synkronista suoritusta, käytetään `EventQueue.invokeLater()` -metodia lähettämään pyyntö operaatioiden suorituksesta asynkronisesti. Nämä pyynnöt menevät tapahtumankäsittelyjonoon, josta tapahtumasäie poimii pyyntöobjektit ja kutsuu `run`()-metodia, joka suorittaa operaatiot.

Kohta 3: `JoptionPane.showMessageDialog()`-metodin kutsu suoritetaan synkronisesti kutsujan kanssa. Kutsuja ei voi edetä seuraavaan operaatioon ennenkuin kutsu on suoritettu, koska kyseessä on toimenpide, joka odottaa ilmoituksen kuittaamista.

Mikäli operaation pitää jäädä odottamaan, käytetään `EventQueue.invokeAndWait()`-metodia pyynnön lähettämiseen. Pyyntö laitetaan tapahtumankäsittelyjonoon ja kutsuja jää odottamaan pyyntöobjektin suorittamista, kun se on

suoritettu kutsuja vapautetaan. Tämän `Test`-luokan suorituksessa käyttäjä joutuu painamaan OK-nappulaa, ennenkuin suoritus jatkuu.

Kohta 4: Kun `getRowCount()`-metodin kutsulta odotetaan paluuarvoa, operaation pitää toimia synkronisesti kutsujan kanssa, koska seuraava operaatio voi riippua paluuarvosta. Koska paikallisissa luokissa käytetyt muuttujat ovat `final`-määreisiä, ei `getRowCount()`-metodista saatua paluuarvoa voida sijoittaa `rowCount`-muuttujaan. Paluuarvo otetaan talteen luomalla uusi `final`-määreinen integer-taulukko, sijoitetaan paluuarvo taulukon elementiksi ja lopulta asetetaan tämä elementti `rowCount`-muuttujan arvoksi. Jos `Test`-luokassa olisi käytetty nimettyjä luokkia, olisi niihin voinut lisätä jäseniä paluuarvoa varten.

Kohta 5: Kutsu `getGridColor`()-metodille on yhtäläinen `getRowCount()`-metodin kanssa, paitsi että paluuarvo saadaan objektina.

Poikkeuksia ei ole käsitelty `Test`-luokassa, jotta esimerkistä ei tulisi pitkä. Liitteessä 3 on `LogUIAspect`-aspekti lähdekoodi, jolla voi kirjoittaa lokin `Test`-luokan toiminnasta.

`LogUIAspect`-aspekti kirjoittaa lokin kaikista metodien kutsuista luokissa, jotka ovat javax –pakkauksessa ja sen alipakkauksissa. Aspektin before advice kirjoittaa metodin sijainnin ja nimen (signaturen) ja `Thread.currentThread()`-metodia käyttävän kutsuvan säikeen nimen.

4.1.2 Säieturvallisuus aspektina

Tässä kappaleessa esitetään AspectJ-kielinen säieturvallisuuden ratkaisu, joka noudattaa yhden säikeen sääntöä. Ratkaisussa sovelletaan Worker-objektin luonti –suunnittelumallia. Käyttöliittymää päivittävät metodit siepataan ja suunnittelumallin mukaisesti luodaan `Runnable`-objekti. Tarkistetaan onko kutsuva säie jo tapahtumajonossa. Jos se on jonossa, annetaan alkuperäisen metodin suorittaa operaatio, jos se ei ole jonossa Worker-objekti laitetaan tapahtumajonoon odottamaan suoritusta.

Tässä aspekti-ratkaisussa toteutetaan Swingin yhdensäikeen sääntö, jolloin säikeiden käsittelyä ei tarvitse toteuttaa `Test`-luokassa, kuten perinteisessä ratkaisussa tehtiin. Aspekti reitittää tapahtumasäikeen kautta ne metodien kutsut, jotka käyttävät tai muuttavat Swing-komponenttien tilaa. Tämän aspektin yhteydessä edellä esitettyä `LogUIActivitesAspect`-aspektia voi käyttää UI-operaatioiden lokin kirjoittamiseen.

Ratkaisussa esitetään abstrakti aspekti, josta implementoidaan konkreettinen aliaspekti, joten ratkaisua voi soveltaa erilaisiin järjestelmiin aliaspektia muuttamalla.

Liitteessä 4 on esitetty abstraktin `SwingThreadSafetyAspect`-aspektin, konkreettisen `DefaultSwingThreadSafetyAspect`-aliaspektin sekä ratkaisua testaavan `Test`-luokan lähdekoodi.

`SwingThreadSafetyAspect`-aspektin abstrakti `uiMethodCalls`-pointcut sieppaa käyttöliittymän metodien kutsut. Pointcut määritellään konkreettisessa `DefaultSwingThreadSafetyAspect`-aliaspektissa.

`SwingThreadSafetyAspect`-aspektin abstrakti `uiSyncMethodsCalls`-pointcut sieppaa synkronista suoritusta tarvitsevat metodit. Pointcut määritellään konkreettisessa `DefaultSwingThreadSafetyAspect` -aliaspektissa.

`SwingThreadSafetyAspect`-aspektin `threadSafeCalls`-pointcut luettelee ne metodien kutsut, joiden ei tarvitse noudattaa yhden säikeen sääntöä. Näitä kutsuja ovat `JComponent.revalidate(), JComponent.repaint()` sekä kuuntelijoiden lisääminen ja poistaminen.

`SwingThreadSafetyAspect`-aspektin `excludedJoinpoints` -pointcut lohkossa valitaan metodien kutsut. Lohkossa poissuljetaan `threadSafeCalls()`-poincut lohkon sieppaamat aspektin join point-kohdat sekä ne jotka suoritetaan tapahtumasäikeessä. Ehto `if(EventQueue.isDispatchhThread())` saa totuusarvon `true,` vain jos tämänhetkinen suoritettava säie on tapahtumasäie.

`SwingThreadSafetyAspect`-aspektin `routedMethods()` -poincut sieppaa ne join point –kohdat, jotka tarvitsevat kutsujen reittämistä tapahtumasäikeen kautta.

`SwingThreadSafetyAspect`-aspektin `around` advice –lohkoissa varmistetaan reititystä tarvitsevien metodien suoritus Worker-objektina synkronisesti. Myös `void`-metodeille, jotka `uiSyncMethodsCalls`-pointcut määrittelee, varmistetaan synkroninen suoritus.

Liitteessä 4 esitetään myös `DefaultSwingThreadSafetyAspect`-aliaspekti, jonka avulla `SwingThreadSafetyAspect`-aspekti kontrolloi metodien reititystä. Aliaspektissa määritellään synkronista reititystä vaativat metodit.

Aliaspektin pointcut-määrittelyjen selitys:

`viewMethodCalls()` –pointcut valitsee kutsut UI-komponenttien metodeille. Se sieppaa kaikki JComponent-luokan ja sen aliluokkien metodit.

`modelMethodCalls()` –pointcut valitsee kaikki UI-malliluokkien ja niiden aliluokkien metodeille.

`uiMethodCalls()` –pointcut valitsee kutsut UI-metodeille yhdistettynä.

`uiSyncMethodCalls()` -pointcut valitsee kaikki kutsut JOptionPane-luokan ja sen aliluokkien metodeille. Ne ovat synkronista suoritusta vaativia UI-metodeja.

4.1.3 Objektin lukitusmalli aspektina

Kilpailevien säikeiden luku- ja kirjoitusoperaatioiden kohdistuessa samaan objektiin voidaan sen eheys varmistaa lukintamenettelyllä. Menettelyn perusajatuksena on, että objektia voi lukea, ellei sen tilaa olla parhaillaan muuttamassa. Perinteisessä luokkatoteuksessa lukitusmekanismi vaatii jokaisen luku- ja kirjoitusmetodin kohdalla erillistä koodausta. Seuraavassa esitetään lukitusmalli, johon kuuluu uudelleenkäytettävä perus-

aspekti sekä sen aliaspekti, jota muokkaamalla mallia voidaan käyttää eri sovellusten luokkien kanssa. Näiden aspektien ja esimerkkiluokkien lähdekoodi on liitteessä 5.

Perusaspekti kapseloi lukitusmekanismin mallin. Aspekti sisältää kaksi pointcut- määritelyä, ensimmäinen sieppaa lukumetodien suoritukset ja toinen kirjoitusmetodien. Nämä suoritukset voidaan synkronoida.

Liitteen 5 `ReadWriteLockSyncronizationAspect`-aspekti käyttää JSR166-luokkakirjastoa, joka on noudettavissa verkosta, katso [Lea05]. Luokkakirjasto on kehitteillä ja täysin valmistuttuaan se tulee olemaan käytössä Javan `java.util.concurrent` –pakkauksena.

`ReadWriteLockSyncronizationAspect`-aspektin selitys kohdittain:

`perthis()`assosioi aspektin instanssit niihin Worker-objekteihin, jotka sopivat luku- tai kirjoitusmetodeihin. Nämä on metodit luetellaan konkreettisessa aspektissa. Uusi aspektin instanssi luodaan jokaiselle objektille, jolle siepattu metodi suoritetaan. Assosiaatio mahdollistaa lukitusobjektin esittelyn kaikille synkronoiduille luokille, objektin tyyppiä ei tarvitse tietää.

Abstrakti `readOperations`() –pointcut määritellään aliaspektissa sieppaamaan kaikki ne metodit, jotka eivät muuta objektin tilaa. Aliaspektissa määritellään myös abstrakti `writeOperations`()-pointcut sieppaamaan metodit, jotka muuttavat objektin tilaa.

Lukitusmallissa `_lock` –muuttuja palvelee syknronoinnin tukena. Koska aspekti on assosioitu tarvittavien join point –kohtien objekteille, `_lock` –muuttuja on assosioitu objektien instansseihin.

`before():readOperations()` ja `after():readOperations`() -advice lohkoissa lukitaan ja vapautetaan lukuoperaatiot.

Samoin `before:writeOperations()` ja `after:writeOperations`() -advice lohkoissa lukitaan ja vapautetaan kirjoitusoperaatiot.

Poikkeusten pehmentäminen konvertoi `acquire`-metodien kutsujen heittämät InterruptedException –poikkeukset, joten siepattujen operaatioiden API:a ei tarvitse muuttaa.

`ReadWriteLockSyncronizationAspect`-aspektia voidaan käyttää aliaspektin kanssa yksittäiselle luokalle tai koko luokkakirjaston lukitusmallina. Esimerkinä `BankingSynchronizationAspect`-aspekti mahdollistaa mallin soveltamisen `Account`-luokkaan, myös näiden lähdekoodi on liitteessä 5.

5 AUTENTIKOINTI JA AUKTORISOINTI

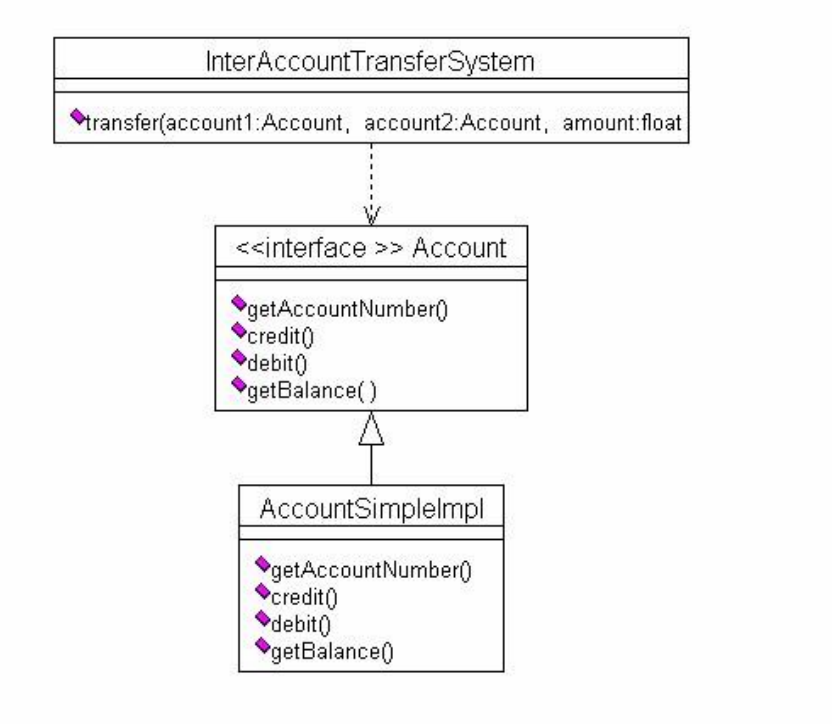
Autentikointi (*authentication*) on oikeuksien tarkistamisprosessi, jossa todennetaan käyttäjän tunnisteet, esimerkiksi käyttäjätunnuksen ja salasanan avulla. *Auktorisointi* (*authorization*) on prosessi, jossa tarkistetaan autentikoidun käyttäjän oikeudet käyttää tarjolla olevia resursseja. Näiden prosessien toteuttamiseksi rakennetut API:t, kuten *Java Authentication and Authorization Service* (*JAAS*), mahdollistavat resurssien käytön valvonnan erottamisen muusta koodista. Näiden ohjelmarajapintojen rinnalle on kehitetty erilaisia standardeja konfiguraation määrittelykieliä, kuten Security Assertion Markup Language (SAML) ja Extensible Access Control Markup Language (XACML). Näiden API:en ja standardien pyrkimyksenä on ollut vähentää monimutkaisuutta ja tarjota nopeita ja toimivia toteutuksia.

Vaikka näitä ohjelmointirajapintoja käytettäisiin, perinteiset ohjelmointitavat vaativat moduulikohtaista autentikoinnin ja auktorisoinnin koodaamista. Esimerkiksi liiketoimintasovelluksen käyttäjien oikeuksien valvonnan toteuttamiseksi pitää lisätä kaikkiin liiketoimintametodeihin kutsut JAAS-metodeille. Sama koskee myös käytön valvonnan logiikkaa, koska liiketoimintalogiikka levittäytyy useisiin moduleihin,.

Seuraavassa kappaleessa esitellään yksinkertainen pankkijärjestelmä. Tähän järjestelmään lisätään myöhemmin autentikointi ja auktorisointi AspectJ-ratkaisuna.

5.1 Pankkijärjestelmä

Pankkijärjestelmän ydintoteutus sisältää seuraavat luokat: pankkitilin rajapintaluokka `Account.java`, josta toteutetaan tililuokka `AccountSimplImpl.java`, poikkeuksia käsittelevä luokka `InsufficientBalanceException.java`, tilinsiirron toteuttava luokka `InterAccountTransferSystem.java`, sekä järjestelmän testaamiseksi `Test.java` –luokka. Katso kuva 6.



Kuva 6. Pankkijärjestelmän ydintoteutus.

`Account`-rajapinta esittelee get-metodin tilinumeron saantiin, tilin hyvitys- ja veloitusmetodin, sekä get-metodin tilinsaldolle. `AccountSimplImpl`-luokka toteuttaa tämän rajapinnan. `InterAccountTransferSystem` –luokka sisältää tilinsiirtometodin. Kaksi viimemainittua luokkaa voivat heittää `InsufficientBalanceException` –poikkeuksen, mikäli tilin saldo ylittyy.

Ohjelman metodien suoritusjärjestyksen seuraamiseksi käytetään `AuthLogging.java` –aspektia, joka listaa tili- ja tilinsiirtoluokan metodien nimet suoritusjärjestyksessä.

`Test`-luokassa luodaan kaksi tiliä, ensimmäistä tiliä hyvitetään kerran ja velotaan kerran, sekä suoritetaan kaksi siirtoa toiselle tilille, jolloin ensimmäisen tilin saldo ylittyy ja ohjelma heittää poikkeuksen.

5.2 JAAS-pohjainen autentikointi perinteisellä tavalla

Järjestelmään voidaan lisätä autentikointitoiminnallisuutta, esimerkiksi *autentikointi tarvittaessa* (*just-in-time*). Silloin autentikointi tapahtuu vasta kun käyttäjä yrittää avata sellaista järjestelmän toimintoa, joka vaatii käyttäjän identiteetin todentamisen. Pelkässä *sisäänkirjautumisen autentikoinnissa* (*upfront login*) kysytään käyttäjänimi ja salasana ohjelman alussa, koska molemmissa autentikoinneissa periaatteessa kyse on samasta asiasta. Seuraavissa kappaleissa kerrotaan (Java Authentication and Authorization Service) JAAS-pohjaisen autentikoinnin keskeisistä asioista, joita ovat:

- Kirjautumisen kontekstin säilyttävän `LoginContext` –objektin luominen.
- Takaisinkutsujen käsittelijät (callback handlers), jotka esittävät kirjautumispyynnön käyttäjälle.
- Kirjautumisen konfiguraatiotiedosto (login configuration file) asettaa luokan, jota käytetään autentikointimodulina. Näin konfiguraatiota voidaan muuttaa ilman lähdekoodin muuttamista.

5.2.1 LoginContext –objektin luonti

Kirjautumisen kontekstin omaava objekti tarvitsee parametreikseen konfiguraation nimen ja takaisinkutsun käsittelijän. Konfiguraation nimi on määritelty konfiguraatiotiedostossa. Objektin luonti noudattaa kieliopillisesti seuraavaa kaava:

```
import javax.security.auth.login.*;
. . .
LoginContext lc =
    new LoginContext(<config file entry name>,
<CallbackHandler to be used for user interaction>);
```

Pankkijärjestelmän `Test`-luokassa se on toteutettu seuraavasti:

```
import javax.security.auth.login.*;
```

```
. . .
LoginContext lc =
        new LoginContext("Sample",
                    new TextCallbackHandler());
        lc.login();
```

5.2.2 Takaisinkutsun käsittelijä

Takaisinkutsun käsittelijä tarjoaa mekanismin autentikointitietojen saamiseksi käyttäjältä. Se kysyy käyttäjältä nimen ja salasanan joko näytöllä keskusteluikkunassa tai ohjelmallisesti. Käyttäjä voi olla ihminen tai toinen järjestelmän osa. Esimerkiksi `TextCallbackHandler`-luokka, joka sisältyy Sunin JRE 1.4 –pakkaukseen, toimittaa autentikointi-informaation kutsuvalle järjestelmälle.

5.2.3 Kirjautumisen konfiguraatiotiedosto

Kirjautumisen konfiguraatiotiedosto asettaa luokan, jota käytetään autentikointi- modulina. Kirjautumisen konfiguraatiotiedoston rakenne ja sisältö kieliopillisesti kuvattuna on seuraavanlainen:

```
<name used by application to refer to this entry> {
   <LoginModule> <flag> <LoginModule options>;
   <optional additional LoginModules, flags and options>;
};
```

Pankkijärjestelmän esimerkin `sample_jaas.config` –tiedosto assosioi tämän konfiguraation `sample`-pakkauksen `SampleLoginModule.java` – luokkaan. Optiolla `debug=true` saa tulostettua viestejä käyttäjälle kirjautumisen onnistuttua.

Pankkijärjestelmän `sample_jaas.config` -konfiguraatiotiedosto:

```
Sample {
 sample.module.SampleLoginModule required debug=true;
};
```

`SampleLoginModule.java` kysyy käyttäjä tunnuksen, joka on "`testUser`" ja salasanan "`testPassword`", jotka annettuaan käyttäjä saa ilmoituksen kirjautumisen onnistumisesta.

Pankkijärjestelmän autentikointia ja kirjautumisen konfiguraatiota varten `sample_jaas.config` –tiedosto sijoitetaan johonkin luokkapolussa olevaan hakemistoon, kuten myös tutoriaalin `sample` -niminen hakemisto, jonka `module`-alihakemisto sisältää `SampleLoginModule.java` –tiedoston.

Tämän raportin pankkijärjestelmä esimerkissä käytetään JAAS-tutorialin tiedostoja. Tutoriaali löytyy Sun-yhtiön verkkosivulta [Sun05]. Tämän autentikointiesimerkin lähdekoodi on raportin liitteessä 6.

5.3 Autentikointi AspectJ-ratkaisuna

AspectJ-ratkaisuna autentikointi voidaan toteuttaa uudelleenkäytettävällä abstraktilla perusaspektilla, jota voi käyttää erilaisten järjestelmän autentikointiin. Perusaspektissa määritellään ilmentymämuuttujaan *autentikoitu subjekti* kirjautumisvaiheessa. Autentikoitu subjekti saa arvokseen `LoginContext`-olion.

Perusaspektille kirjoitetaan aliaspekti, joka laajentaa sen järjestelmäkohtaiseksi. Aliaspektiin määritellään pointcut-lohko, jossa siepataan autentikointia tarvitsevat järjestelmän operaatiot.

Pankkijärjestelmän autentikoinnin mahdollistavat `AbstractAuthAspect` -aspekti ja konkreettinen `BankingAuthAspect`- aliaspekti. Näiden lähdekoodi on liitteessä 7.

`AbstractAuthAspect`-aspektissa määritellään ilmentymämuuttujaan autentikoitu subjekti kirjautumiskontekstin yhteydessä:

```
LoginContext lc =
    new LoginContext("Sample",
                  new TextCallbackHandler());
    lc.login();
    _authenticatedSubject = lc.getSubject();
```

Kirjautuminen tapahtuu kerran ja on voimassa koko prosessin ajan kaikille operaatioille. Autentikoitu subjekti –muuttuja voidaan siirtää myös käyttäväksi muualla, esimerkiksi servletin istunto-objektiin. Järjestelmästä uloskirjautuessa subjekti-muuttujan arvoksi pitää asettaa `null`.

`AbstractAuthAspect`-aspektissa esitellään abstrakti `authOperation()` –poincut, joka määritellään aliaspektissa sieppaamaan autentikointia tarvitsevat operaatiot. Perusaspektin `before` -advice varmistaa, että järjestelmä suorittaa autentikoinnin kerran ohjelman ajon aikana. Jos `_authenticatedSubject` arvo on `null`, autentikointi suoritetaan, jos arvo on `not null`, autentikointia ei suoriteta. Näin järjestelmässä toteutuu just-in-time autentikointi, eli se suoritetaan vain silloin, kun jotain järjestelmän palvelua tarvitaan, eikä siihen ole vielä kirjauduttu.

Konkreettisessa aliaspektissa määritellään pointcut-lohko, jossa siepataan autentikointia tarvitsevat järjestelmän operaatiot. `BankingAuthAspect`-aspektin `execution` –advice voidaan määritellä sieppaamaan myös ohjelman käynnistävän `main()` –metodin, jolloin käyttäjätunnukset kysytään ensimmäiseksi.

5.4 Auktorisointi

Auktorisointiprosesi päättelee onko käyttäjällä oikeudet käyttää hänen haluamiaan järjestelmän toimintoja. Auktorisoinnin esiehtona on että käyttäjä on autentikoitu.

5.4.1 JAAS-pohjainen auktorisointi luokkatoteutuksena

JAAS-pohjaisessa auktorisoinnissa järjestelmä saa autentikoinnin alijärjestelmältä tarkistetun käyttäjän subjektin. Tämä `Subject` –luokan ilmentymä kapseloi käyttäjän tietoja kuten tunnisteen ja valtuudet. Kaikkien auktorisointia tarvitsevien operaatioiden on tarkistettava subjektin valtuudet.

Jokainen auktorisoinnin tarkistamista tarvitseva metodi on kapseloitava `action` -objektiin, joka toteuttaa `PrivilegedAction` tai `PrivilegedExceptionAction` -rajapinnan. Näillä molemmilla on ainoastaan `run()` –metodi, joka suorittaa halutun operaation. Rajapintojen ainoa ero on poikkeuksen käsittely, `PrivilegedAction` ei sisällä poikkeusten julistusta (declaration) lainkaan, mutta `PrivilegedExceptionAction` voi heittää `Exception`-tyyppisen poikkeuksen.

`Action` –objekti suoritetaan autentikoidun subjektin puolesta käyttämällä `Subject` –luokan staattista `doAsPrivileged(Subject, PrivilegedAction, AccessControlAccess)` -metodia. Sen keskimmäinen toimintoparametri voi olla myös `PrivilegedExceptionAction`, jolloin `run()`-metodin heittämä tarkistettu poikkeus kääritään ennen sen heittämistä.

Auktorisoitua toimintoa tarvitsevien metodien on tarkistettava subjektin valtuudet kutsumalla `AccessController.checkPermission()` –metodia ja välitettävä `permission`-objekti, joka sisältää vaaditun valtuuden. Jos käyttäjällä ei ole valtuuksia, metodi heittää tarkastamattoman `AccessControlException`-poikkeuksen.

Auktorisoitavalle järjestelmälle on määriteltävä *käyttöoikeustiedosto* (*policy file*), joka asettaa eri subjekteille valtuudet tiettyihin operaatioihin. `AccessControl-`

ler.checkPermission() –metodi käyttää tätä tiedostoa ja sallii operaatioiden käytön vain mikäli subjektilla niihin valtuudet.

Edellä esitellylle pankkijärjestelmälle on auktorisoinnin toteuttamiseksi kirjoitettu BankingPermission –luokka, jonka konstruktorin String name –muuttuja välittää käyttäjän valtuudet. Muuttujan avulla luetaan käyttöoikeustiedostosta käyttäjän valtuudet. Toisen konstruktorin action-parametria käytetään instantioimaan valtuutuskohde käyttöoikeustiedostosta. BankingPermission –luokan lähdekoodi on liitteessä 8.

Liitteessä 8 on myös esimerkki pankkijärjestelmän käyttöoikeudet määrittelevä security.policy -tiedosto, jossa määritellään testUser -käyttäjälle oikeudet käyttää kaikkia tilinkäyttöön liittyviä operaatioita.

5.4.2 JAAS-pohjainen auktorisointi AspectJ-toteutuksena

JAAS-pohjainen auktorisointi toteutetaan reitittämällä auktorisoidut kutsut sellaisen luokan välityksellä joka implementoi joko PrivilegedExceptionAction- tai PrivilegedAction-poikkeuksen, riippuen siitä heittääkö operaatio tarkistetun poikkeuksen. Worker-objektin luonti -suunnittelumalli nopeuttaa tämän toteuttamista, koska voidaan kirjoittaa aspekti, joka suorittaa operaatiot worker-objektin välityksellä Subject.doAsPrivileged() –metodin kautta.

Auktorisointi voidaan modularisoida yhteen aspektiin kirjoittamalla uudelleen käytettävä abstrakti aspekti, joka lisää auktorisoinnin järjestelmään. Tälle perusaspektille kirjoitetaan järjestelmäkohtainen konkreettinen aliaspekti. Näin saadaan järjestelmä auktorisoitua vähäisellä koodimäärällä. Abstrakti perusaspekti toteuttaa auktorisoinnin ohella myös autentikoinnin. Esimerkkinä tällaisesta perusaspektista on AbstractAuthAspect-aspekti, jonka lähdekoodi on liitteessä 8.

AbstractAuthAspect-aspekti ohjaa kaikki auktorisointia tarvitsevien operaatioiden kutsut PrivilegedExceptionAction-rajapinnan toteuttavan anonyymin luo-

kan kautta. Lisäämällä `proceed()`-metodi worker-objektin suorittavaan `run()`-metodiin, kääritään operaation argumentit, joten ne voivat olla mitä tahansa tyyppiltään ja lukumäärältään.

`AbstractAuthAspect` –aspektissa esitellään abstrakti pointcut `authOperations()`, konkreettisessa aliaspektissa tämä pointcut määrittelee auktorisointia tarvitsevat operaatiot. Nämä operaatiot ovat samoja, jotka tarvitsevat autentikointia. Kun pelkkä sisäänkirjautuminen ohjelmistoon ei riitä resurssien käytön valtuudeksi, operaatiot voidaan eritellä esimerkiksi seuraavasti:

```
pointcut authenticatedOperations()
    :primaryAuthenticatedOperations()
     || authorizedOperations();
```

Abstraktissa perusaspektissa esitellään abstrakti `getPermission()` –metodi, joka hakee tarvittavat valtuudet. Myös tämä metodi määritellään konkreettisessa aliaspektissa. Metodi välittää valtuudet operaatioiden suorittamiseen.

Abstraktin perusaspektin `around` -advice suorittaa siepatut operaatiot autentikoidun subjektin puolesta. Se luo auktorisointia tarvitseville operaatioille worker-objektin ja suorittaa ne `run()`-metodissaan. Perusaspektin `RuntimeException` –poikkeukset kääritään joko `AuthorizationException`- tai `AuthenticationException`-tyyppisiksi poikkeuksiksi.

Konkreettisessa aliaspektissa määritellään järjestelmäkohtainen auktorisointi. Aliaspektiin määritellään vain edellä kerrotulla tavalla `authOperations()`-pointcut ja `getPermission()` –metodi. Metodi välittää valtuudet operaatioon joinpoint-kohdan sieppaaman operaation nimen perusteella. Pankkijärjestelmän esimerkin konkreettin `BankingAuthAspect` –aliaspektin lähdekoodi on liitteessä 8.

6 TRANSAKTIO

Tässä luvussa havainnollistetaan transaktion hallinnan ongelmia. Transaktio koostuu atomisista toimenpiteistä, jotka takaavat että järjestelmä säilyttää ristiriidattoman tilansa ennen toimenpidettä ja sen jälkeen. Atomisuus, eheys, erillisyys ja kestävyys ovat transaktiolta vaadittuja ominaisuuksia.

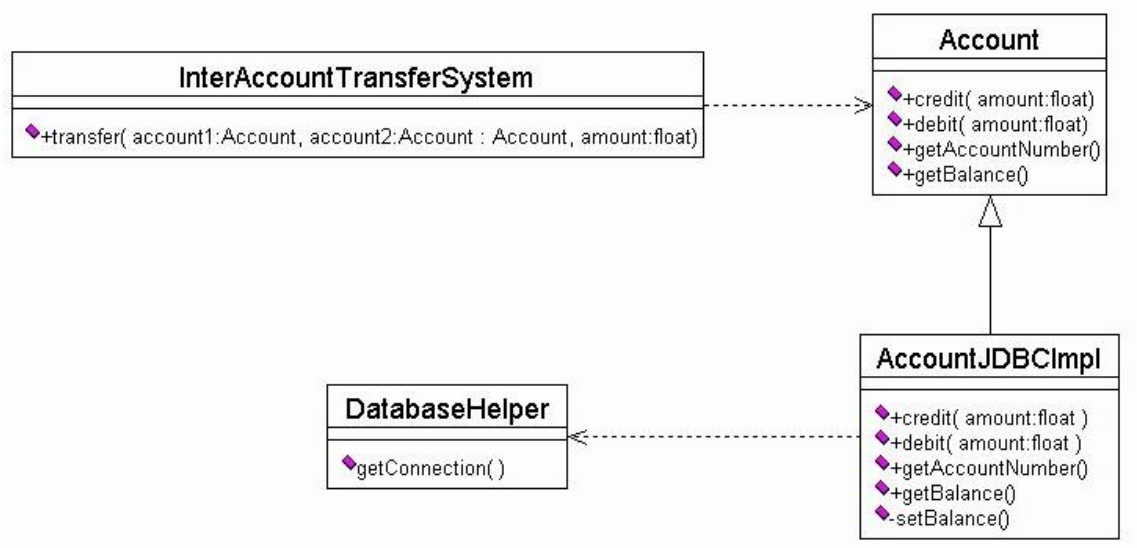
Esimerkkinä toteutetaan edellä kerrotun osittaisen pankkijärjestelmän transaktion hallinta. Ensin kerrotaan peruslähtökohdat ja esitellään perinteinen ratkaisu sekä AspectJ-ratkaisu. Transaktion hallinta on toteutukseltaan järjestelmän poikkileikkaava tehtävä, joten ratkaisun modularisointi selkeyttää sen toteutusta ja ylläpitoa.

6.1 Ydin tehtävien toteutus

Pankkijärjestelmän vaatimuksena on pysyvyys ja tietokannan eheys. Yhteys tietokantaan pitää olla taloudellinen. Käyttäjän pitää pystyä suorittamaan haluamansa operaatio loppuun. Mikäli se ei onnistu, jo tehdyt toimenpiteet perutaan.

Tässä esimerkissä käytetään JDBC- yhteyttä MySQL-tietokantaan. Tietokannan taulujen luominen ja konfigurointi on esitetty liitteessä 1.

Kuva 7 esittää UML-notaatiolla järjestelmän peruslähtökohdat. Kappaleessa 6.2 kerrotaan transaktion perinteisestä ratkaisusta ja kappaleessa 6.3 AspectJ-ratkaisusta.



Kuva 7. Pankkijärjestelmän perusluokat

6.2 Transaktion perinteinen ratkaisu

Tämän pankkijärjestelmän toiminnallinen ydin on toteutettu perinteisillä java-luokilla. Järjestelmässä on tilin ja tilinsiirron toteuttavat luokat, sekä tietokantayhteyden ja transaktion päivityksen hyväksyvä luokka. Esimerkin lähdekoodi on liitteessä 9.

`AccountJDBCImpl`-luokka on `Account`-rajapinnan konkreettinen toteutus. Luokka sisältää rajapinnan metodien toteutuksen lisäksi `private`-määreisen metodin, joka asettaa tilin saldon. Luokan metodit sisältävät transaktioita. Tämän luokan sisällä metodien suorituksessa ei tarvitse huomioida muiden metodien tekemiä päivityksiä tietokantaan.

`InterAccountTransferSystem`-luokan ainoa metodi siirtää rahamäärän tililtä toiselle. Tällöin ohjelman suorituksessa tapahtuu *sisäkkäisiä transaktioita* (*nested transactions*), mikä vaatii metodien suoritusten koordinointia.

Transaktion eheyden vaatimuksen täyttymiseksi sekä `credit()`- että `debit()`-metodien suorituksen täytyy onnistua tai päivitykset tietokantaan pitää perua.

`DatabaseHelper`-luokassa `Connection`-luokan `getConnection()` –metodi palauttaa tietokantayhteyden objektina. Metodissa voidaan määritellä päivitystapa tietokantaan. Jos sen arvoksi on asetettu `connection.setAutoCommit(true)`, tietokannan päivityksiä ei tarvitse hyväksyä ohjelmallisesti, vaan se tapahtuu välittömästi ohjelman suorituksessa. `DatabaseHelper.getConnection()`-metodi luo aina uuden yhteyden jokaiselle tietokantaoperaatiolle. Siksi ei voida tehdä useita päivityksiä yhden transaktion sisällä, koska jo tapahtuneita päivytyksiä ei peruta vaikka operaatiossa yksi päivitys epäonnistuu. Näin ollen järjestelmä voi jäädä virheelliseen tilaan operaation epäonnistuessa. Tämän estämiseksi pitää auto-commit arvoksi asettaa `false`. Lisäksi samaa yhteysobjektia pitää käyttää kaikille saman operaation tietokannan päivityksille ja hyväksyä ne kerralla, jos kaikki ovat onnistuneet tai perua, jos jokin päivitys epäonnisui.

Jos samaa `DatabaseHelper`-luokan luomaa yhteysobjektia käytetään koko operaation ajan. Tämän toteuttamiseksi on kaksi yleistä tapaa; välitetään yhteysobjekti operaatioon osallistuville metodeille argumenttina tai säiekohtaisen varaston käyttäminen.

Yhteysobjektin välittäminen metodeille vaatii melkoisia muutoksia luokkien toteutukseen. Tässä esimerkissä yhteys voidaan luoda `InterAccountTransferSystem`-luokan `transfer()` –metodissa, joka kutsuu `debit()`- ja `credit()`-metodia, ja antaa niille lisäargumentiksi yhteysobjekti. Metodien suoritusta ei hyväksytä erikseen, vasta kun molemmat metodit on suoritettu onnistuneesti, yhteysobjekti kuitataan `Connection` –luokan `commit()`-metodilla ja yhteys suljetaan. Seuraavassa nämä metodin koodiin tehdyt muutokset:

```
public static void transfer(Account from, Account to,
                                    float amount)
    throws InsufficientBalanceException {
    Connection conn = DatabaseHelper.getConnection();
    conn.setAutoCommit(false);
    try {
```

```
            to.credit(amount, conn);
            from.debit(amount, conn);
            conn.commit();
            }
            catch (Exception ex) {
                        conn.rollback();
                        // log exception jne
            }
            finally {
                        conn.close();
        }
    }
```

Näistä muutoksista aiheutuu edelleen muutoksia `credit()` ja `debit()`-metodeihin. Laajemmassa järjestelmässä tällaisten muutosten tekeminen voi olla hyvin työläs.

Toinen - edellistä modularisoidumpi - tapa välittää yhteysobjekti lisäargumenttina on käyttää `ThreadLocal`-luokan tarjoamaa säikeiden varastointia. Tällöin on `DatabaseHelper`-luokaan lisättävä yhteysobjektin käärivä `static final ThreadLocal` -jäsen, jolta `DatabaseHelper.getConnection()`-metodi määritellään pyytämään yhteys. Metodi palauttaa varastosta säiekohtaisen yhteyden, jos sellainen on saatavilla. Jos ei ole, metodi luo uuden yhteyden ja asettaa sen varastoon. Tämä tapa modularisoi yhteyden käyttämisen, mutta ei transaktiota tehtäväkokonaisuutena.

Jos vaaditaan, että sisäkkäisessä transaktiossa vain ylätason operaatio voi suorittaa kuittauksen, on tarpeen määritellä ohjelmallisesti kuittauksen suorittaja. Yksittäistä metodia ei voi määritellä ylätason operaatioksi, koska jokaiseen transaktioon pitäisi määritellä metodien taso erikseen. Eräs ratkaisu on käyttää säikohtaista muuttujaa tai lisäargumenttia, joka ilmaisee kutsun tason tai syvyyden. Kutsun tason mittaria kasvatetaan

metodin suoritukseen mentäessä ja vähennetään suorituksen päättyessä, jolloin nollatasolle palattaessa voidaan suorittaa kuittaus.

Yhteenvetona tästä pankkijärjestelmän osan perinteisestä ratkaisusta voidaan huomata, että sen transaktion hallinnan koodi on hajanaisesti useissa metodeissa ja moduleissa, ja sen muunneltavuus on sidottu liiketoimintalogiikkaan.

6.3 AspectJ-kielinen transaktion ratkaisu

Tässä luvussa muodostetaan edellä esitetystä pankkijärjestelmän osasta AspectJ-kielinen ratkaisu. Järjestelmän perusluokat ovat samat kuin perinteisen järjestelmän ratkaisua esittävässä kuvassa 7, joka esitettiin luvussa 6.1.

Järjestelmän korkean tason vaatimuksena on:

- Ratkaisun on oltava uudelleen käytettävä. Tätä ratkaisua käytetään muissakin JDBC-pohjaisten järjestelmien kehitysprojekteissa.

- Transaktion päivitysoperaatioiden on käytettävä yhtä yhteysobjektia, joka sisältää transaktion tilan. Kun metodin kontrollivirrassa tarvitaan transaktion hallintaa, yhteyden luonnin jälkeen kaikki tämän kontrollivirran päivitykset käyttävät samaa yhteysobjektia. Näin tämä yhteysobjekti muodostaa operaatioiden kontekstin.

- Kaikki transaktion päivitykset hyväksytään, kun ylätason operaatio päättyy onnistuneesti. Jos jokin päivitys epäonnistuu tai jokin liiketoimintametodi heittää poikkeuksen, yhteysobjektin kaikki päivitykset on peruttava (roll back).

Ratkaisun on uudelleen käytettävyys toteutuu luomalla abstrakti aspekti, joka toteuttaa pääpiirteet transaktion logiikasta. Logiikan lisäksi tämä aspekti sisältää abstraktit pointcut –esittelyt transaktion operaatioille ja yhteyden saamiseksi.

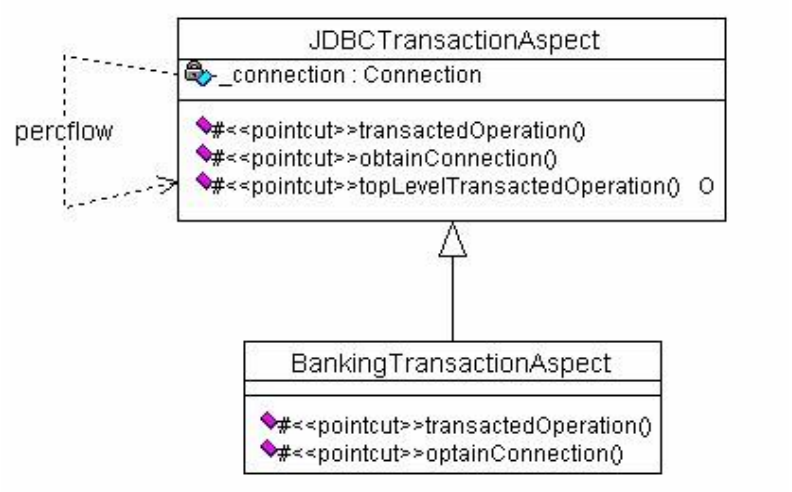
Ylätason operaatioiden sieppaamiseksi määritellään pointcut, joka sieppaa ne ajon operaatiot jotka eivät jo ole toisen transaktion operaation kontrollivirrassa. Ylätason operaa-

tioiden onnistumista valvotaan poikkeuksien hallinnalla. Transaktion operaatioiden suoritus etenee around advice –määrittelyn try/catch –lohkossa. Operaatio on onnistunut mikäli se ei heitä yhtään poikkeusta.

Yhden yhteysobjektin käytön varmistamiseksi transaktion sisällä määritellään advice-ohje transaktion kontrollivirrassa yhteyden luovalle joint point –kohdalle. Kun yhteyttä tarvitaan ensimmäisen kerran siepatussa operaatiossa, yhteys hankitaan ja se varastoidaan aspektin sisällä ilmentymä muuttujaan (instance variable). Kun yhteyttä tarvitaan seuraavan kerran, advice palauttaa varastoidun yhteyden ja välittää siepatun operaation. Aspekti assosioidaan myös ylätason operaatioiden kontrollivirtaan. Koska aspekti varastoi yhteysobjektin jota käytetään kaikkiin päivityksiin, aspektin ilmentymä sisältää transaktion kontekstin niin kauan kuin transaktio kestää.

Transktiota tarvitsevan järjestelmän osan toteutukseen määritellään konkreettinen aliaspekti. Aliaspektissa määritellään konkreettinen pointcut, joka sieppaa perustoteutuksesta transaktion join point –kohdat ja konkreettinen pointcut, joka sieppaa yhteyden muodostamisen join point –kohdat.

Yhteyden luomiseen voidaan käyttää erilaisia tapoja, kuten apuluokkia tai resurssien säiliöitä (resource pooling). Tämän vuoksi perusaspektissa esitellään yhteyksiä varten abstrakti pointcut, joka määritellään konkreettisessa aliaspektissa. Kuva 8 esittää UML-notaationa pankkijärjestelmän transaktion hallinnan aspekteja.



Kuva 8. Pankkijärjestelmän transaktion hallinnan aspektit.

6.3.1 JDBCTransactionAspect-aspektin koodin selitys

Tämän AspectJ-kielisen ratkaisun lähdekoodi on liitteessä 10. `JDBCTransactionAspect` –aspektin selitys kohdittain:

`percflow(topLevelTransactedOperation())` assosioi aspektin ilmentymän transaktion ylätason operaation kontrollivirtaan. Aliaspektin `transactedOperation()`–pointcut määrittelee transaktion operaatiot, joita kutsuttaessa ilmentymä luodaan jos se ei kuulu jo meneillään olevaan transaktioon. Ilmentymä säilyttää transaktion tilan. Tätä ratkaisua voidaan käyttää, jos transaktioon osallistuu vain yksi järjestelmä.

Abstraktit `obtainConnection`-pointcut ja `transactedOperation()`–pointcut määritellään konkreettisessa aliaspektissa. Nämä käsittelevät yhteyden luomisen ja operaatioiden suorituksen.

`topLevelTransactedOperation()`-pointcut määritellään ylätason operaatioksi. Transaktion hallinnan suorittamiseksi sen tukena on abstrakti `transactedOperation()`-pointcut, sekä `Object around` -advice. Aspektin uusi instanssi assosioi-

daan ylätason operaatioon. Kun sellaisen join point –kohdan suoritus alkaa, luodaan konkreettisen aliaspektin uusi instanssi.

`Object around` –advice, joka ohjeistaa `topLevelTransactedOperation()`, vie siepatun operation try/catch -lohkoon. `Try`-lohkossa kutsutaan `proceed()` –metodia operaation suorituksen jatkamiseksi. Jos operaatio heittää suorituksen aikana poikkeuksen, suoritus siirtyy `catch`-lohkoon joka kutsuu yhteysobjektin `rollback()`-metodia, jolloin päivitykset perutaan. `Finally`-lohko sulkee yhteyden.

`if(_connection != null)` käsittelee tapaukset joissa yhteyttä tietokantaan ei luotu, koska liiketoimintalogiikka ei tarvinnut päivityksiä tai kyselyjä tietokannasta. (Ylätason operaatio voi sisältää tällaisia alioperaatioita suorituksensa loppuvaiheessa.)

Tässä transaktion hallinnan ratkaisussa samaa yhteyttä käytetään operaation kaikkiin päivityksiin, jolloin ne voidaan hyväksyä kerralla kutsumalla yhteysobjektin `commit()`-metodia. `Connection around()` –advice tarkistaa, onko `connection` –instanssin arvo `null`. Mikäli se on `null`, ylätason operaation suoritus tarvitsee ensimmäisen kerran yhteyttä. Seuraavaksi advice-lohko hankkii yhteyden ja kytkee sen automaattisen hyväksynnän pois päältä metodissa `setAutoCommit(false)`. Jos operaatio on onnistunut yhteysobjekti kuitataan `commit()`-metodilla. Perättäisille yhteysobjektin kutsuille advice-lohko palauttaa varastoidun yhteysobjektin sen sijaan, että loisi uuden.

`TransactionException` määritellään ajonaikaisena poikkeuksena. Tämän poikkeuksen heittäminen ilmaisee kutsujalle, että transaktio on epäonnistunut ja kääritty poikkeus ilmaisee epäonnistumisen syyn.

Poikkeusten pehmentämisellä vältytään try/catch-lohkojen määrittelyltä `rollback()`- ja `close()`-metodien kutsuissa.

Koska tässä ratkaisussa vain `JDBCTransactionAspect` -aspekti suorittaa kutsut yhteysobjekteille, määritellään koodausta ohjaava `illegalConnectionManagement()`-poincut ja `around()` advice-lohko, jotka käsittelevät aspektin ulkopuolelta

määritellyt virheelliset kutsut. Tällaisia kutsuja voisi tehdä esimerkiksi pankkijärjestelmän määrittelyssä kirjoittamalla uusia transaktioon liittyviä hyväksymiskäytäntöjä kutsuen `Connection.setAutoCommit()`-metodia.

`illegalConnectionManagement()`-poincut sieppaa kaikki yhteyden hallinnan join point –kohdat ja `around()` advice-lohko estää ohjelmoijaa käyttämästä niitä. Advice-lohkon sijaan tai sen ohella voidaan määritellä myös käännöksen aikaisen virheen tulostus `declare error` –lauseella seuraavasti:

```
declare error : illegalConnectionManagement()
        :  “Do not call close(), commit(), rollback(),
        or   setAutoCommit() on Connectin objects from
        here”;
```

6.3.2 BankingTransactionAspect-aspektin koodin selitys

Suurin osa transaktion toiminnallisuudesta on määritelty abstraktissa `JDBCTransactionAspect`-aspektissa. Sitä laajentavaan konkreettiseen `BankingTransactionAspect`-aspektiin tarvitaan vain kaksi pointcut-määrittelyä pankkijärjestelmän transaktion hallinnan toteuttamiseksi.

`transactedOperation()`-pointcut sieppaa transaktion tukea tarvitsevien metodien suorituksen ja `optainConnection()`-pointcut määrittelee kaikki metodien kutsut joita käytetään tietokanta yhteyksien saamiseksi.

6.4 Yksi transaktio – useita alijärjestelmiä

Yhdessä transaktiossa joudutaan usein suorittamaan operaatioita, joihin osallistuu useita alijärjestelmiä. Lisäksi voi olla vaatimuksena että jokaisella alijärjestelmällä on oma aliaspekti, jonka poincut-lohkot määrittelevät transaktion tukea tarvitsevat operaatiot.

Näin alijärjestelmiin tehtävät muutokset aiheuttavat transaktion osalta muutoksia vain aliaspektissa. Tällöin on varmistettava etteivät muutokset aiheuta ristiriitoja transaktion hallinnassa.

Näiden vaatimusten ratkaisemiseksi voidaan käyttää Osallistuja-suunnittelumallia, jonka mukaisesti tehdään jokaiselle transaktion osapuolena olevalle alijärjestelmälle transaktiota tukeva aliaspekti. Nämä sisäaspektit laajentavat abstraktin `JDBCTransactionAspect` -perusaspektin. Ne voidaan toteuttaa itsenäisinä aliaspekteina, tai luokkien sisäaspekteina.

Pankkijärjestelmä esimerkkissä `AccountJDBCImpl`- ja `InterAccountTransferSystem`- luokat ovat toiminnallisesti erillisiä alijärjestelmiä, joten niihin voidaan soveltaa edellä kerrottua Osallistuja-suunnittelumallia. Tässä esimerkissä aliaspektit lisätään alijärjestelmien luokkiin sisäaspekteina. Järjestelmän rakennetta havainnollistaa kuva 9.

6.4.1 TransactionParticipantAspect-aspektin selitys

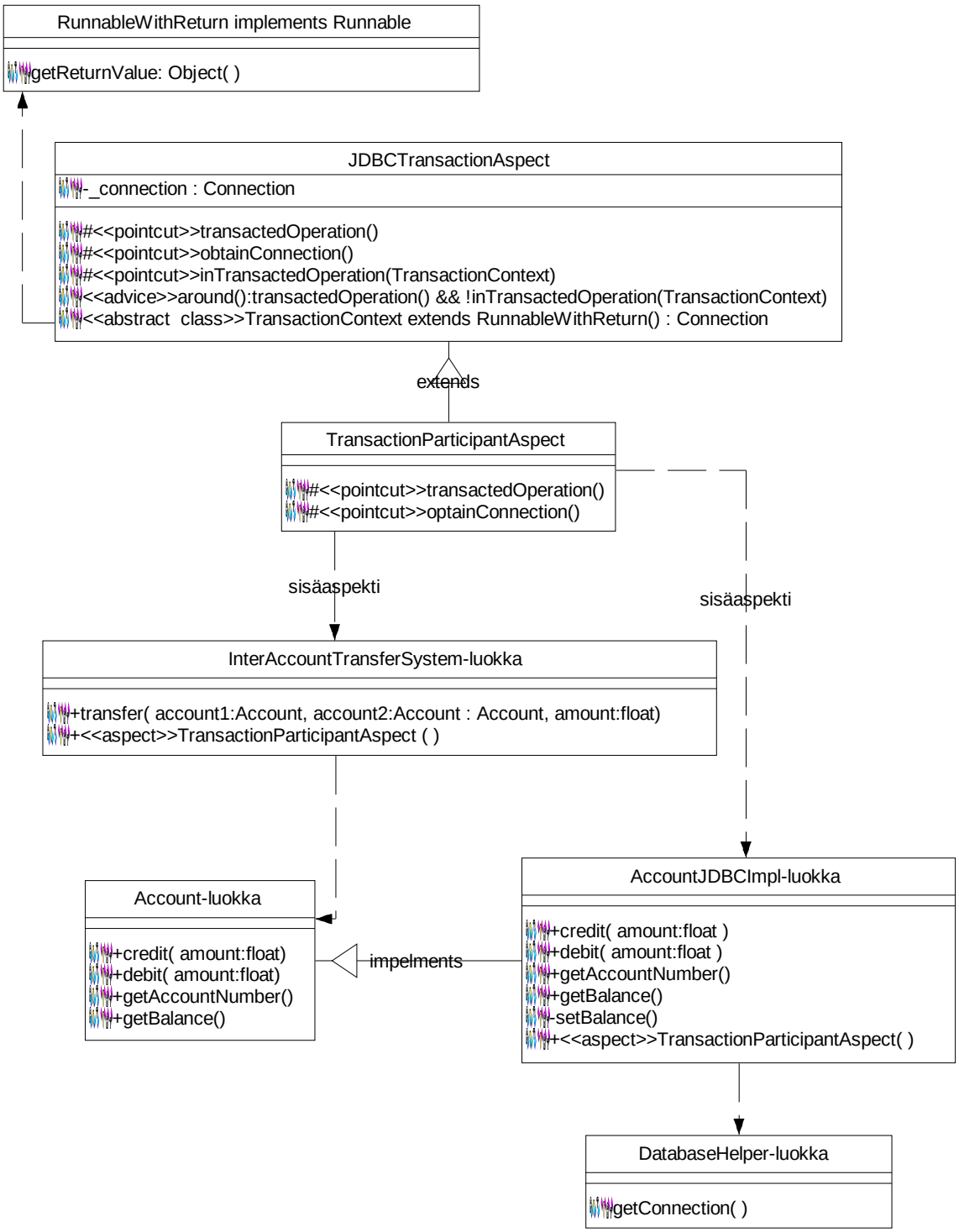
Liitteessä 11 on `AccountJDBCImpl`- ja `InterAccountTransferSystem`-luokkien tämän ratkaisutavan lähdekoodi.

Näiden luokkien `TransactionParticipantAspect`-sisäaspektit laajentavat `JDBCTransactionAspect`-aspektin. Tämän laajennusominaisuuden vuoksi `InterAccountTransferSystem`-luokan sisäaspektissa joudutaan määrittelemään yhteyden hakeva `obtainConnection()`-pointcut, joka ei sisällä yhtään join pointkohtaa, koska tämä luokka ei saa hakea koskaan itse yhteyttä. Luokan sisäaspektien `transactedOperation()`-poincut sieppaa luokansa transaktiota tarvitsevien operaatioiden suoritukset.

Vaikka tämä ratkaisutapa mahdollistaa transaktion hallinnan kapseloinnin `JDBCTransactionAspect` -perusaspektiin, voi syntyä ajonaikaisia ongelmia jos eri osapuolten aliaspektien instanssit eivät käytä samaa yhteysobjektia. Näin ollen vain osa operaatios-

ta saattaa onnistua, esimerkiksi tilinsiirrossa vastaanottajan tilia voidaan hyvittää vaikka maksavan tilin veloitusoperaatio on epäonnistunut.

Jotta transaktion hallinnassa voidaan käyttää useita alijärjestelmien aspekteja, on varmistuttava että transaktion päivityksiin käytetään samaa yhteysobjektia silloinkin, kun eri aspektit sieppaavat transaktion aikana erillisiä operaatioita. Päivityksestä ja huolehditaan transaktion toteuttavassa perusaspektissa, johon tehtävistä muutoksista kerrotaan tarkemmin seuraavassa luvussa.



Kuva 9. Pankkijärjestelmä sisältää kaksi transaktioon osallistuvaa alijärjestelmää, joiden transaktion hallintaa tuetaan sisäaspekteilla.

6.4.2 Usean alijärjestelmän transaktion toteuttava perusaspekti

Edellä luvussa 6.3 esitetyssä versiossa abstrakti `JDBCTransactionAspect` -perusaspekti huolehti kahdesta roolista. Se varastoi yhteysobjektin, johon kaikki transaktion operaatiot pystyvät viittaamaan sekä huolehti transaktion hyväksymisestä tai perumisesta. Tässä luvussa esitellään perusaspekti, jossa nämä roolit on eroteltu. Aspekti määritellään vastuulliseksi pelkästään transaktion hyväksymisestä ja perumisesta, ja käytetään hyväksi erillistä transaktion kontekstin objektia, joka varastoi yhteyden.

Tämän erillisyyden toteuttamiseksi perusaspektissa käytetään oletusarvoista assosiointia `percflow`-assosiaation sijaan, sekä poistetaan `_connection` -muuttuja aspektista. Oletusarvoinen (default) assosiaatio johtaa siihen, että jokaisesta aliaspektista luodaan korkeintaan yksi instanssi ohjelman suorituksen aikana. Koska `_connection` -instanssi poistetaan aspektista se on tilaton aspekti, eivätkä aspektin instanssit tässä ratkaisussa ilmaise transaktion kontekstia. Worker-objektin luonti suunnittelumallia käytetään luomaan automaattisesti Worker-objekti jokaiselle uudelle transaktiolle. Worker-objektin tehtävänä on varastoida yhteysobjekti, näin Worker-objekti palvelee transaktion kontekstina.

Worker-objekti luodaan transaktion hallintaa tarvitseville ylätason operaatioille ja siten käsitellään tätä operaatiota worker-metodina. Worker-objektin `run()`-metodi sisältää worker-metodin joka sisältää transaktion hallinnan logiikan, eli operaation hyväksymisen tai perumisen. Kun suoritetaan ensimmäinen transaktion hallintaa tarvitseva metodi, luodaan yhteysobjekti joka varastoidaan Worker-objektiin. Toteutuksessa käytetään `context`-objektia varmistamaan, ettei luoda ylimääräistä yhteysobjektia kun Worker-objektin `run`()-metodin kontrollivirrassa saadaan käsiteltäväksi jokin uusi join point – kohta.

Aspektiin implementoidaan `TransactionContext`-luokka, joka laajentaa `RunnableWithReturn`-luokan. Jokaisen konkreettisen aliluokan pitää implementoida `run()`-metodi, joka suorittaa worker-metodin ja asettaa `_returnValue`-jäsenen operaation tulokseen. Uusi `context`-objekti luodaan kaikille transaktion aloittaville ope-

raatioille elleivät ne jo ole transaktion kontekstin `run()`-metodin suorituksen kontrollivirrassa. Kukin transaktioon osallistuva aspekti voi luoda transaktion kontekstin, uutta kontekstia ei luoda operaatiolle, joka on jo osa transaktion kontekstia. Koska transaktion hallinta suoritetaan vain `context`-objektin kautta, uutta kontekstia ei luoda sisäkkäisille operaatioille. Liitteessä 11 on tämän usean alijärjestelmän transaktion esimerkin lähdekoodi.

6.4.3 Muutetun JDBCTransactionAspect-aspektin koodin selitys

Aspektin `percflow`-assosiaatio on muutettu oletusarvoiseksi assosiaatioksi. Aspekti sisältää abstraktit `transactedOperation()`- ja `obtainConnection()`-pointcut esittelyt, jotka määritellään konkreettisissa aspekteissa.

`inTransacteOperations()`-pointcut sieppaa `TransactionContext.run()`-metodin kontrollivirran ja taltioi metodiin assosioidun objektin, joka välitetään kutsupinoon. Tämä pointcut tutkii kuuluuko operaatio jo meneillään olevaan transaktioon.

`Object around()` -advice määrittelee `transactedOperations()`ja `inTransacteOperations()`-pointcut määrittelyjen perusteella ne join point –kohdat, jotka eivät vielä ole transaktion kontekstin suorituksessa ja luo tarvittaessa uuden transaktion kontekstin. Lohkon `run()`-metodin määrittely on samanlainen kuin ensimmäisessä versiossa. Tässä toteutuu worker-objektin luonti.

`Connection around()`-advice ohjeistaa yhteyden hankkimista ja tarvitsee suorituksessa olevan transaktion kontekstin objektia, jotta se voi käyttää (access) sen `_connection`-objektia. `inTransacteOperations()`-pointcut sieppaa tämän kontekstin. Tässä on sovellettu madonreikä-suunnittelumallia luomaan suora yhteys transaktion kontekstin objektin `run()`-metodin ja yhteyttä tarvitsevan metodin välille.

Abstrakti `TransactionContext`-luokka laajentaa `RunnableWithReturn`-luokan. Luokan `_connection` -muuttuja säilyttää transaktion yhteysobjektin.

6.5 JTA-pohjainen transaktion hallinta

Tässä raportissa edellä esitellyt järjestelmät perustuivat JDBC-pohjaiseen transaktion tukeen. Nykyaikaisissa tietojärjestelmissä vaatmukset ovat kuitenkin laajempia. Usein tarvitaan hajautettua transaktion hallintaa ja mukana on useita resursseja sekä tietokantoja. Operaatioissa joudutaan päivittämään myös muiden järjestelmänosien tai ERP (Enterprise Recourse Planning) –järjestelmien tietokantoja. Näissä tapauksissa transaktio käsittelee useita yhteysobjekteja.

Transaktio voi ulottua myös monenlaisiin resursseihin. Esimerkiksi samassa transaktiossa voidaan päivittää tietokantaa ja lähettää viestejä jonoon käyttäen JMS (Java Message Service) –palvelua. Tällöin sekä tietokannan päivityksen että viestijonon lähetyksen pitää onnistua, jotta järjestelmä säilyttää ehyen (consistent) tilansa. Nämä tapaukset vaativat (Transaction Processing) TP-monitorin käyttöä, jolla koordinoidaan resurssien transaktion onnistumista. Tällöin resurssien (kuten tietokantojen ja JMS viestityksen väliohjelmistojen) pitää pystyä yhteistyöhön TP-monitorin kanssa. Monitoroinnissa ohjelmallisten rajapintojen käyttö, kuten JTA (Java Transction API), mahdollistavat TP-monitorin soveltavan käytön erilaisissa ympäristöissä.

Käytettäessä puhtaasti olio-suuntaunutta ohjelmointitekniikkaa, JTA ei mahdollista transaktion tehtävien eriyttämistä ydintehtävistä. Tämä johtuu siitä, että toteutuksessa transaktion luonti, hyväksyminen ja peruminen leviävät kaikkiin moduleihin, joissa on transaktion hallintaa vaativia operaatioita. JTA tarjoaa läpinäkyvyyttä käytettävään TP-monitorointiin, mutta vaatii sen API:n kutsumista useista ydinmoduuleista.

AspectJ-kielen käyttö yhdessä JTA:n kanssa mahdollistaa transaktion hallinnan modularisoinnin poikkileikkaukselliseksi tehtäväksi. Liitteessä 12 on abstrakti aspekti `JTATransactionAspect`, joka kapseloi kaikki JTA-pohjaiset transaktiot. Tuekseen se tarvitsee yhden tai useampia konkreettisia aliaspekteja, joissa määritellään `transactedOperation`()-pointcut sieppaamaan transaktion tukea tarvitsevat operaatiot.

Tämä perusaspekti poikkeaa `JDBCTransactionAspect`-aspektista vain seuraavilta osin:

`JTATransactionAspect`-aspektissa käytetään transaktion hallintaan `UserTransaction` –objektia, joka tarjoaa API:n transaktion hyväksymiseen ja perumiseen. Transaktio alussa kutsutaan `UserTransaction.begin()`, hyväksyttäessä `UserTransaction.commit()`, ja peruttaessa `UserTransaction.rollback()`.

Tässä aspektissa ei tarvita pointcut- ja advice-lohkoa takaamaan saman yhteysobjektin käyttöä. JTA:ta käytettäessä ei yksittäiseen päivitykseen tarvitse käyttää vain yhtä resurssia, kuten tietokantayhteyttä.

Tätä aspektia käytettäessä on käännösympäristössä hyvä käyttää valvovia menettelytapojen sääntöjä takaamaan, ettei aspektia käytetä muille kuin JTA:han sopiville resursseille.

7 YHTEENVETO

Aspektisuuntautunut ohjelmointitekniikka mahdollistaa ohjelmiston toiminnallisuuden modularisoinnin aspektien avulla. Aspekti laajentaa oliosuuntautuneen ohjelmoinnin luokkien käyttötapaa toteutuksessa. Toteutusta voidaan tiivistää muutamaan aspektiin sen sijaan, että koodaus leviäisi laajalle ohjelmiston rakenteeseen. Modularisointi selkeyttää ohjelmiston rakennetta ja parantaa sen uudelleen käytettävyyttä. Perinteiset suunnittelumallit ovat silti edelleen käytettävissä.

Tämän raportin esimerkeissä esitetyt aspektit ovat rakenteeltaan uudelleen käytettäviä ratkaisuja. Niissä toteutetaan käyttäjän kirjautuminen järjestelmään ja auktorisointi, jossa käyttäjälle annetaan valtuuksia käyttää tiettyjä resursseja ja järjestelmän operaatioita, joihin tarvitaan transaktion tukea. Esimerkkien ratkaisut täyttävät modulaarisuuden vaatimuksen, koska kukin ohjelmiston poikkileikkauksellinen tehtävä tai huolehdittava asia on toteutettu selkeästi erillisessä ohjelman osassa, joka koostuu muutamasta aspektista. Ratkaisut eivät vaadi suuria muutoksia sovellettaessa niitä samankaltaisiin järjestelmiin.

LÄHTEET

[ElF01] Elrad T., Filman r., Bader A., *The Aspect Oriented Programming*, Chicago,2001,(ACM 2005)

[HiH04] Hilsdale E., Hugin J., *Advice Weaving in AspectJ*, Lancaster UK, 2004 (ACM, 13.1.2005)

[Ken99] Kendall E. A., *Role model designs and implementations with aspect-oriented programming*, Denver US, October 1999, ACM 17.1.2005

[LaC03] Lafferty D., Cahill V., *Language-independent aspect-oriented programming*, Trinity College Dublin, October 2003 (ACM, 13.1.2005)

[Lad03] Laddad R., *AspectJ in Action Practical Aspect-Oriented Programming*, Manning Publications Co, USA, 2003.

[Sun05] *JAAS Authentication Tutorial*, viitattu 12.2.2005 , Saatavilla http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html

[Xer03] Xerox Corporation, Palo Alto Research Center, *The AspectJ Programming Guide*, viitattu 20.2.2005, saatavilla: http://dev.eclipse.org/viewcvs/ indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html.

OHJELMISTOT JA LÄHDEKOODIT:

[ECL05] *AspectJ Compiler* (*aspectj-1.2.1.jar*) -ohjelmisto, viitattu 15.1.2005 saatavilla: http://eclipse.org/aspectj/,

[ECL05a] *Eclipse Platform SDK 3.0.1* ja *Eclipse AspectJ Development Tools*, viitattu 18.1.2005, saatavilla: http://eclipse.org/downloads/index.php

[MYS05] *MySQL 4.1-ohjelma*, viitattu 28.2.2005 Saatavilla: http://dev.mysql.com/downloads/

[Lea05] Lea Doug, *JSR166-luokkakirjasto,* viitattu 10.3.2005 saatavilla: http://gee.cs.oswego.edu/dl/concurrency-interest/index.html.

[Lea05a] Lea Doug, *Concurrent-luokkakirjaston kotisivu, viitattu 10.3.2005, saatavilla:* http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/ intro.html

[Lea03b] Lea Doug, *AspectJ in Action -kirjan esimerkkien lähdekoodi*, saatavilla: http://www.manning.com/catalog/view.php?book=laddad&item=source

[Lea03c] Lea Doug, JSR 166 –ohjelmisto ja dokumentaatio, viitattu 10.3.2005, saatavilla: http://gee.cs.oswego.edu/dl/concurrency-interest/index.html

[Lea03d] Lea Doug, *backport-util-concurrent luokkakirjasto,* viitattu 10.3.2005, saatavilla: http://www.mathcs.emory.edu/dcl/util/backport-util-concurrent/

Muita resursseja:

Suunnittelumallien toteutuksia:

Implementations of GoF Design Patterns in Java and AspectJ, saatavilla: http://www.cs.ubc.ca/~jan/AODPs/