Metropolia Ammattikorkeakoulu
Tietotekniikan koulutusohjelma

Mark Virta

Aspektipohjainen .NET-ohjelmistokehitys

Insinöörityö 19.5.2009

Ohjaaja: ohjelmistokehittäjä Tommi Kemppi
Ohjaava opettaja: yliopettaja Kari Aaltonen

Metropolia Ammattikorkeakoulu Insinöörityön tiivistelmä

Tekijä Otsikko Sivumäärä Aika	Mark Virta Aspektipohjainen .NET-ohjelmistokehitys 51 sivua 19.5.2009
Koulutusohjelma	tietotekniikka
Tutkinto	insinööri (AMK)
Ohjaaja Ohjaava opettaja	ohjelmistokehittäjä Tommi Kemppi yliopettaja Kari Aaltonen

Tässä insinöörityössä käsitellään tehokasta ja erittäin yleistä koodia tuottavaa ohjelmointitapaa, aspektiohjelmointia, sekä verrataan sitä perinteisempiin, oliopohjaiseen ja proseduraaliseen ohjelmointiin. Työssä esitellään myös kehitystyökalut aspektipohjaisen sovelluksen luomiseksi .NET-alustalle sekä luodaan Eficia Oy:lle liikelahjakomponentti.

Aspektiparadigmaa on sovellettu jo 1980-luvulta lähtien ohjelmoinnissa, mutta silti levinneisyys ohjelmistonkehityksessä on ollut pientä verrattuna oliopohjaiseen ja proseduraaliseen ohjelmointiin. Nykyaikainen sovelluskehitys suosii korkean tason abstraktiota koodin uudelleenkäytettävyyden lisäämiseksi, joten aspektipohjainen paradigma vastaa teoriassa hyvin tarpeeseen. Tässä työssä esitellään aspektiohjelmoinnin edut ja haitat sekä syvennytään sen määrittelyyn, toteutukseen käytännössä sekä vaiheisiin. Projektissa käytetyt työkalut ja teknologiat esitellään, mm. Spring .Net-ohjelmistokehys, NHibernate, MicroSoft Visual Studio sekä MicroSoft SQL.

Työssä ohjelmoitiin osaksi laajempaa kokonaisuutta liikelahjakomponentti käyttäen aspektiohjelmointia ja olio-ohjelmointia. Komponentilla korvattiin vanha ja hankalasti ylläpidettävä liikelahjarekisteri. Projekti piti sisällään määrittelyn yhdessä asiakkaan kanssa, jonka pohjalta suunniteltiin, toteutettiin ja testattiin tietokannat, ohjelmistoluokat ja käyttöliittymät sekä kartoitustiedostot. Valmis komponentti luovutettiin asiakkaalle.

Hakusanat	ohjelmointi, aspektiohjelmointi, ohjelmistokehitys, Windows ohjelmointi, C#, .Net, Spring

Helsinki Metropolia University of Applied Sciences Abstract

Author Title Number of Pages Date	Mark Virta Aspect Oriented .NET Software Development 51 19.5.2009
Degree Programme	Information Technology
Degree	Bachelor of Engineering
Instructor Supervisor	Tommi Kemppi, Software Developer Kari Aaltonen, Senior Lecturer

This thesis discusses aspect oriented programming (AOP), an efficient and very generic code generation methodology. It is compared to more traditional object-oriented and procedural programming. Also, development tools needed for developing an aspect-oriented program for MicroSoft Windows platform are introduced and progression of development of a software component for Eficia Ltd. is reported.

The aspect-oriented paradigm has been part of software development from the 1990s, but utilization has been low compared to procedural and object-oriented programming. Modern software development favours high level abstraction in code in order to gain optimal reusability, which means that AOP is a good answer to demands. This document describes benefits and drawbacks of AOP and studies the specifications, and sequences of AOP. Also, the actual implementation is discussed. All the tools used for development are introduced, e.g. Spring .Net Framework, NHibernate, MicroSoft Visual Studio and MicroSoft SQL.

This thesis covers development of a software component as a part of a larger enterprise resource planning software entity, completed using aspect-oriented programming and object-oriented programming. The project included specification together with the customer, which acted as a basis for designing, programming and testing of databases, software classes, interfaces and mapping files. The completed component delivered to customer.

Keywords	programming, aspect-oriented programming, software development, Windows programming, C#, .Net, Spring

Sisällys

Lyhenteet, käsitteet ja termit

AD	*Active Directory,* käyttäjätietokanta ja hakemistopalvelu, joka sisältää tietoa käyttäjistä, tietokoneista ja windows-verkon resursseista
AJAX	*Asynchronous JavaScript And XML,* joukko sovelluskehityksen tekniikoita, joiden avulla verkkosovelluksista voidaan tehdä vuorovaikutteisempia.
AOP	*Aspect-Oriented Programming,* aspektiohjelmointi, aspektipohjainen ohjelmointi
ASP	*Active Server Pages*, MicroSoftin kehittämä, dynaamisten www-sivujen luomiseen tarkoitettu palvelinpuolen ohjelmointimenetelmä
ASP.NET	Verkkosovellusten kehittämiseen luotu ohjelmistokehys ja osa .Net- ohjelmistokehystä
ASPX	ASP.NET ympäristöön kehitetty dynaamisten www-sivujen luomiseen tarkoitettu palvelinpuolen ohjelmointimenetelmä
C#	*C Sharp,* MicroSoftin .Net-konseptia varten kehittämä ohjelmointikieli. Se tukee funktionaalista, imperatiivista, geneeristä, komponentti- ja olio-ohjelmointia.
Eficia Core	Eficia Oy:n päätuote, toiminnanohjausjärjestelmä
JavaScript	Netscape Communications Corporationin kehittämä pääasiassa Web-ympäristössä käytettävä komentosarjakieli, joka lisää www-sivujen dynaamisuutta
LDAP	*Lightweight Directory Access Protocol,* hakemistopalvelujen käyttöön tarkoitettu verkkoprotokolla
Modulariteetti	Konsepti, joka kuvaa tyypillisesti systeemin komponenttien eroteltavuuden ja uudelleenkäytön tason
MSI	*MicroSoft Installer,* Windowsin käyttämä ohjelmistojen asennuspakettien asennuspalvelu

.Net Framework	MicroSoftin kehittämä ohjelmistokehys, joka sisältää laajan kirjaston ja virtuaalikoneen joka hallitsee ajettavaa ohjelmaa
NHibernate	objekti-relaatio kartoitusohjelmisto SQL-tietokantayhteyksien hallintaan .Net-alustalle
OOP	*Object-Oriented Programming,* olio-ohjelmointi, oliopohjainen ohjelmointi
RAD kontrollit	*RAD Controls,* Telerikin kehittämä valikoima työkaluja dynaamisten verkkolomakkeiden tuottamiseen
SOC	*Separation of Concerns,* Mielenkiinnon kohteiden, asioiden tai ominaisuuksien eriyttäminen, vaatimusten jäsentäminen
Spring .Net Framework	Vapaan lähdekoodin ohjelmistokehys, joka tukee aspektipohjaista ajattelua ja laajentaa .Net-kehyksen toiminnallisuuksia
SQL	*Structured Query Language,* Tietokantakieli, joka on suunniteltu tiedon säilöntään ja hallintaan relaatiotietokantasysteemeissä
VPN	*Virtual Private Network,* virtuaalinen lähiverkkoteknologia, joka salaa liikenteen pisteiden välillä
XML	*Extended Markup Language,* merkintäkieli tai standardi, jolla tiedon merkitys on kuvattavissa tiedon sekaan. Kieltä käytetään sekä formaattina tiedonvälitykseen järjestelmien välillä että formaattina dokumenttien tallentamiseen

1 Johdanto

Ohjelmistokehitys on joutunut vastaamaan erilaisiin, haastaviin markkinoiden vaateisiin kehittymällä nopeasti alusta asti. Ensimmäisistä, yksinkertaisista laitteisto- ja aikakriittisistä ohjelmistoista ja hyvin rajoittuneista kehitys-työkaluista on siirrytty yhä enenevässä määrin kohti massiivisia, graafisia sovelluksia. Tyypillisesti näiltä ohjelmistoilta vaaditaan myös korkeata integraatiota toisiin järjestelmiin ja niitä rakennetaan laajoilla kehitysympäristöillä.

Myös koko ohjelmistosuunnittelun ajatusmalli on pitänyt uudistaa paremmin palvelemaan moderneja vaatimuksia. 1970-luvulla kehitetty olioparadigma mullisti ohjelmistojen kehityksen ja niiden mahdollisuudet suorittaa yhä laajempia tehtäviä monipuolisessa ympäristössä. Jatkuvasti muuttuvat vaatimukset myös vaikeuttavat ohjelmistojen ylläpitoa niiden monimutkaistuessa jatkuvasti [1, s. 4]. Tässä työssä käsiteltävä aspektiohjelmointi ja aspektipohjainen ohjelmistokehitys vievät olio-ohjelmoinnissa tärkeässä roolissa olevan abstraktion edelleen pidemmälle käsittelemään yhä laajempia kokonaisuuksia.

Aspektiohjelmoinnin tavoitteena on luoda mahdollisimman modulaarista ja uudelleenkäytettävää koodia parantamalla tiettyjä epäkohtia sovelluksen kehityksessä, mihin olio-ohjelmointi on liian rajoittunutta. Aspektiohjelmoinnin hyöty kasvaa sitä enemmän mitä suuremmista sovelluksista on kyse, mikä tekee siitä laajoissa ja haastavissa ympäristöissä kustannustehokkaan ja nopeasti päivitettävän ratkaisun.

Eficia Oy on pieni, moderniin .Net-ohjelmointiin keskittynyt ohjelmistoyritys, jonka päätuote on toiminnanohjausjärjestelmä Eficia Core. Yritys suunnittelee ja tuottaa omia ohjelmistoja, komponentteja ja käyttöliittymiä sekä suorittaa Windows-järjestelmien välisiä integraatioprojekteja ja vuokraa konsulttipalveluita

ohjelmistokehitysyrityksille. Eficia huolehtii myös omien tuotteidensa ylläpidosta, tuesta ja myynnistä suoraan yritysmyyntinä.

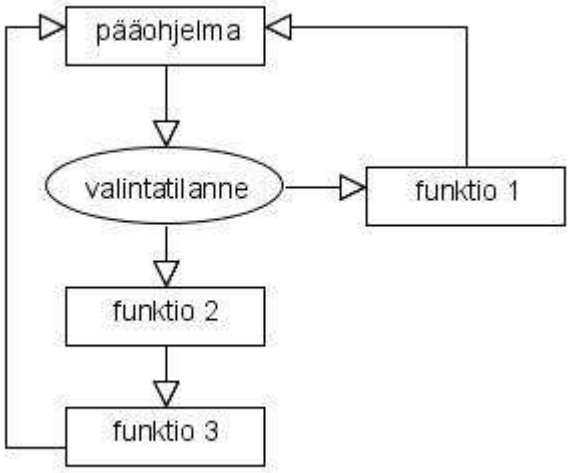
Aspektipohjaista ajattelua soveltaen suunniteltu ja toteutettu liikelahjakomponentti Eficia Coreen esitellään tarkemmin luvuissa 5 ja 6. Komponentti korvasi vanhan ja hankalasti päivitettävän liikelahjojen hallintarekisterin. Koko ydinohjelmisto johon komponentti kehitettiin, on ohjelmoitu aspekti- ja olioparadigmoja soveltaen, ja se on tuotantokäytössä usealla asiakkaalla. Liikelahjakomponentilla hallinnoidaan nykyaikaisella selainkäyttöliittymällä liikelahjarekisteriä, joka sijaitsee MicroSoftin SQL-kannassa. Komponentti on integroitu käyttäjätietokantaan, joka takaa muun muassa käyttäjätunnistuksen, kustannuspaikat ja osoitetiedot. Komponentissa on myös helppokäyttöiset raportointiominaisuudet suoraan taulukkolaskentasovellukseen.

2 Ohjelmointimallien kehitys

2.1 Proseduraalinen ohjelmointi

Ohjelmistonkehityksen varhaisessa vaiheessa ohjelmistot kehitettiin konekielellä ja ne olivat tyypillisesti erittäin laitesidonnaisia ja nykyaikaisiin ohjelmistoihin verrattuna yksinkertaisia. Tällainen ohjelmistonkehitys vaati ohjelmoijalta laajaa tuntemusta laitteistosta, johon ohjelmistoa kehitettiin sekä konekielisten käskyjen osaamista. Laajojen ohjelmistojen kehitys ns. ”tietokoneen kielellä” kävi kuitenkin pian haastavaksi, mikä johti korkean tason kielien kehittymiseen. Nämä kielet ovat helpommin ihmisen luettavissa, ja erillinen kääntäjä huolehtii koodin käännöstä konekielisiksi käskyiksi. Tämä helpotti laajempien ohjelmistojen kehittämistä vähentämällä työtä.

Hietasen mukaan [1, s. 4] perusajatus proseduraalisessa ohjelmoinnissa on ohjelmiston toiminnallisuuden pilkkominen pienempiin osiin. Näitä osia, funktioita, metodeja, rutiineja tai alirutiineja, voidaan toistaa halutussa järjestyksessä. Niitä voidaan kutsua milloin tahansa ohjelman suorituksen aikana. Tyypillisesti yksi rutiini suorittaa yhtä tehtävää, ja sitä voi kutsua mikä tahansa toinen rutiini tai se itse. Näillä hallituilla, peräkkäisillä rutiinien kutsuilla ohjelmointiin saadaan enemmän tehokkuutta verrattuna suoraan hierarkkiseen koodiin tai yksinkertaiseen paikasta toiseen hyppimiseen koodin seassa, jolloin myös koodin hallinnointi ja päivitys on helpompaa.



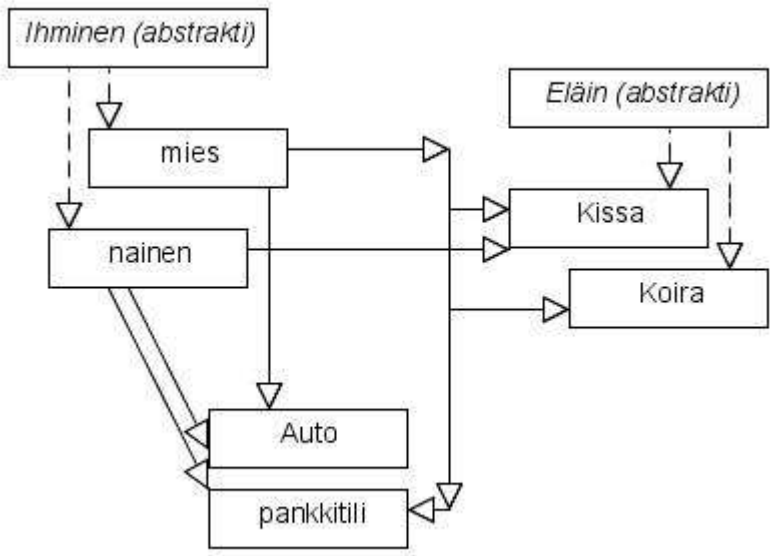
Kuva 1. Proseduraalisessa ohjelmoinnissa eri rutiineja voidaan kutsua halutussa järjestyksessä milloin tahansa

Jokainen tällainen alirutiini voi myös kutsua toisia alirutiineja milloin tahansa kuten kuvassa 1 esitetään, ja tyypillisesti korkean tason proseduraalisissa kielissä, kuten C:ssä tai alkuperäisessä Pascalissa, alirutiinista palataan suorituksen jälkeen alirutiinikutsua seuraavaan ohjelmointilausekkeeseen. Alirutiinit pystyvät välittämään tietoa toisille alirutiineille tyypillisesti palautusarvona ja parametreina. Tällaisella ohjelmointityylillä päästään laitteiston kannalta tehokkaaseen koodiin, mutta koodin ylläpito, hahmottaminen ja päivitys ovat edelleen ohjelmoijan kannalta haasteellisia.

2.2 Olio-ohjelmointi

Olio-ohjelmoinnin juuret ulottuvat vuoteen 1967, jolloin julkaistiin Simula [1, s. 6]. Käytännössä olio-ohjelmoinnista tuli hallitseva paradigma 1990-luvulla sen syrjäyttäessä vanhemmat ajatusmallit. Nykypäivän Windows-maailmassa lähes kaikki sovellukset on ohjelmoitu oliokielillä. Suosituimpiin oliokieliin lukeutuvat java, C#, C++, php ja visual basic.

Olio-ohjelmoinnissa pyritään mallintamaan reaalimaailmaa havainnollisemmin kuin proseduraalisessa ohjelmoinnissa ja helpottamaan laajojen sovellusten tuottamista sekä koodin uudelleenkäytettävyyttä. Peruskomponenttina olio-ohjelmoinnissa toimii olio, joka voi olla koodissa eri abstraktiotasoilla ja olla käytännössä mitä tahansa [1, s. 7]. Kuvasta 2 käy ilmi, miten niinkin abstrakti asia kuin pankkitili voi olla olio, samoin kuin olioiden periytyminen joltain korkeammalta abstraktiotasolta. Kuvassa esitetään myös olioiden mahdollisuudet toisien olioiden omistamiseen tai kommunikointiin niiden kanssa kuten reaalimaailmassa.

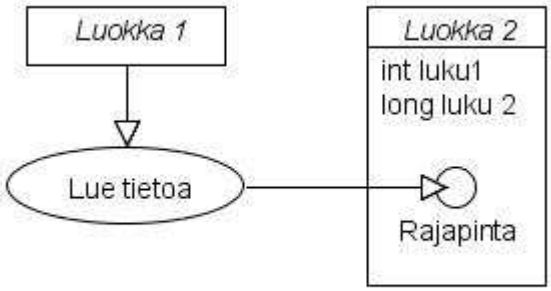


Kuva 2. Olio-ohjelmoinnissa oliot voivat olla eri abstraktiotasoilla, periytyä ylemmältä abstraktiotasolta ja kommunikoida keskenään

Jokainen olio ilmentää luokkaa. Luokassa määritellään olion palvelut, attribuutit, jotka ovat usein tietoja, jotka kuvailevat luokan ominaisuuksia sekä metodit jotka hoitavat olion toiminnallisuuden. Olio-ohjelmoinnissa on myös muita etuja perinteisempiin ajatusmalleihin verrattuna, jotka tehostavat ohjelmistokehitystä: kapselointi, periytyminen, monimuotoisuus ja mallit tai yleiset tietotyypit ovat olio-ohjelmoinnin kulmakiviä ja mahdollistavat huomattavasti uudelleenkäytettävämmän ja havainnollisemman koodin kuin proseduraalisella tavalla toteutetut ratkaisut. Näitä käyttämällä saadaan aikaiseksi

uudelleenkäytettäviä luokkakirjastoja, ja kirjastojen ollessa tarpeeksi laajoja perinteinen koodinkirjoitus vähenee lähes minimiin ja sen sijaan voidaan käyttää valmiita rakennuskomponentteja, joita yhdistelemällä saadaan aikaiseksi sovellus.

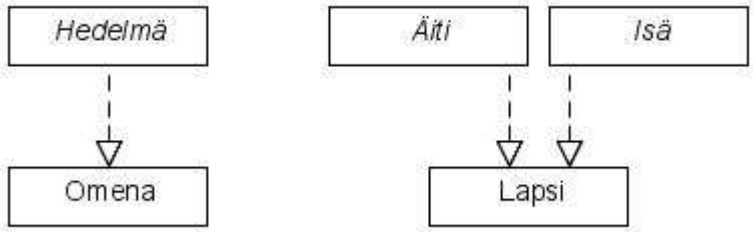
Kapselointi parantaa ohjelman ylläpidettävyyttä. Tavoitteena on erottaa komponentin ulkoiset rajapinnat sisäisestä tiedonkäsittelystä. Kun luokan ominaisuuksia käytetään muualla ohjelmistossa, sitä kutsuvat komponentit käyttävät ainoastaan määriteltyä rajapintaa, eivät luokan tietoja itsessään kuten kuvassa 3. Näin ollen rajapinta hoitaa olion tiedonpiilotuksen [1, s. 10]. Näin ohjelmoituihin sovelluksiin ei tarvitse päivitettäessä tehdä muutoksia ympäristöön, vaan luokan sisäistä toiminnallisuutta pystytään muuttamaan koskematta rajapintoihin.



Kuva 3. Kapselointi eristää luokan sisäisen toiminnallisuuden ulkopuolisista luokkakutsuista. Kaikki luokkaa kutsuvat oliot käyttävät sen ominaisuuksia rajapinnan kautta.

Periytymisessä on kyse varsin samasta asiasta kuin genetiikassa; Kantaluokat sisältävät ominaisuuksia ja palveluita, jotka aliluokka voi periä, hyvin samaan tapaan kuin ihmisten jälkeläiset perivät tiettyjä ominaisuuksia vanhemmiltaan. Luokat ja oliot ovat siis hierarkkisessa järjestyksessä, eli kantaluokan periytyviä ominaisuuksia ja palveluita ei tarvitse kirjoittaa jokaiselle aliluokalle erikseen, vaan voidaan käyttää jo olemassa olevia, perittyjä ominaisuuksia [1, s. 10]. Periytyminen voidaan joissain kielissä toteuttaa joko yksiperiytymisenä, jolloin aliluokalla on vain yksi kantaluokka, tai moniperiytymisenä, jolloin kantaluokkia

on useampi kuin yksi. Molemmista periytymistavoista on esimerkki kuvassa 4. Uudemmissa oliokielissä moniperintä on pääosin korvattu rajapinnoilla.



Kuva 4. Luokat voivat periä ominaisuuksia joko yhdeltä tai usealta kantaluokalta joko moniperinnän tai rajapintojen kautta.

Monimuotoisuudella tarkoitetaan sitä, että kutsuessaan oliota ohjelmoijan ei tarvitse vertailla luokkien tyyppejä välittäessään viestiä sille, järjestelmä huolehtii siitä. Hietasen [1, s. 10] mukaan ”monimuotoisuus mahdollistaa ohjelmoinnin periytymishierarkian ylimmällä abstraktiotasolla välittämättä olion todellisesta määrittelyluokasta.” Jokaisella oliolla voi olla oma toteutuksensa välitetystä viestistä, mutta jos tällaista ei löydy, olio käyttää automaattisesti kantaluokasta periytyvää toteutusta. Esimerkkinä voidaan käyttää vaikka muodostinta, jolla määritellään, miten olio alustetaan. Kaikilla olioilla on oletusmuodostin, joka luo tyhjän olion ilman alkuarvoja tai määrittelyjä. Sen lisäksi voidaan kirjoittaa yksi tai useampi muodostin, jotka alustavat luotavan olion halutulla tavalla.

Mallit, yleiset tietotyypit tai parametrisoidut tietotyypit ovat aliohjelmia tai luokkia, jotka sisältävät toisia aliohjelmia, jotka eivät ole sidottuja mihinkään tiettyyn tietotyyppiin. Nämä yleiset tyypit mahdollistavat saman ohjelmalogiikan käyttämisen eri tietotyyppien yhteydessä kirjoittamatta ja muokkaamatta lähdekoodia uudelleen [1, s. 10]. Yleisen luokan käyttöönottovaiheessa ohjelmoija ilmoittaa parametreilla lopullisen tietotyypin, joka hyödyntää malliluokan koodia. Esimerkkinä toimivat säiliöt, kuten listat ja jonot. Nämä voivat olla yleisesti kirjoitettuja malleja, jotka tyypitetään halutulle tietotyypille.

Kaikki nämä olio-ohjelmoinnin ominaisuudet mahdollistavat huomattavasti tietoturvallisemman ja tehokkaamman ohjelmistokehityksen. Kunnollisella suunnittelulla ja abstraktiota soveltaen voidaan ohjelmoida erittäin yleisiä ja tehokkaita komponentteja, jolloin koodin uudelleenkäytettävyys paranee ja ylläpitotyö vähenee. Tämä vaatii kuitenkin järeämmät työkalut kuin perinteisempi ohjelmointi. Oliokehityksessä .Net-kehitysympäristö on usein laajahko paketti, joka sisältää runsaasti yhden tai useampien kielien luokkakirjastoja, graafisen ja älykkään käyttöliittymän, kattavan ohjeistuksen, edistyksellisen virheenkäsittely- ja testiympäristön sekä hyvät laajennusvalmiudet kolmannen osapuolen tuotteille.

Oliokehityksessä on myös omat huonot puolensa. Ohjelman toiminnallisuus pilkotaan pieniksi osasiksi, jotka kootaan takaisin yhdeksi ohjelmaksi. Näin ollen muodostuu komponenttien välisiä solmukohtia, jotka eivät ole modulaarisia. Tästä hyvän kuvan antaa java-kielen poikkeuskäsittely [2, s. 8], joka ei ole modulaarista vaan ohjelmoijat kirjoittavat usein uuden, räätälöidyn virheenkäsittelyn jokaiselle tarvitsevalle komponentille. Virheenkäsittelyn koodi ei ole helposti uudelleenkäytettävää, sillä sen suorittaminen yleisellä tasolla tuottaa helposti epäluotettavia ja tehottomia ohjelmistoja.

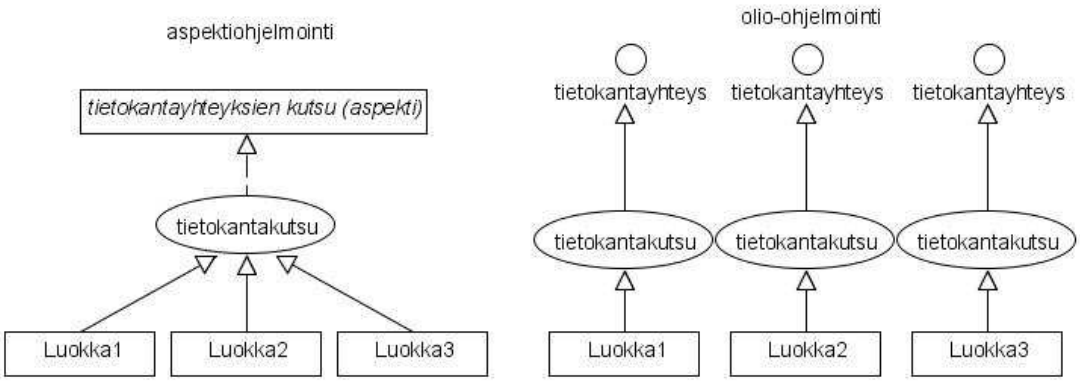
2.3 Aspektiohjelmointi

Aspektipohjainen ohjelmointi on Xeroxin Palo Altossa sijaitsevassa tutkimuskeskuksessa kehitetty ohjelmointimenetelmä, jonka avulla läpileikkaavat ominaisuudet voidaan kerätä yhteen paikkaan, jolloin niiden käsittely on helpompaa ja ohjelmakoodista saadaan siistimpää ja tehokkaampaa. Läpileikkaavat ominaisuudet ovat toimintoja, joita käytetään useissa luokissa tai proseduureissa, kuten esimerkiksi poikkeusten käsittely. Aspektiohjelmointia käytetään muiden ohjelmointiparadigmojen rinnalla. Ohjelma toteutetaan ensin läpileikkaavia näkökulmia lukuun ottamatta

perinteisellä ohjelmointikielellä, minkä jälkeen läpileikkaavat ominaisuudet koodataan aspekteihin aspektikielellä [3, s. 1].
Jotta aspektiohjelmoinnista saadaan kaikki hyöty irti, tarvitaan kokonaisvaltainen lähestymistavan muutos ohjelmistojärjestelmän kehittämiseen perinteikkäämpiin paradigmoihin verrattuna. Tähän kuuluvat näkökulmat vaatimusten analysointiin ja suunnitteluun, toteutukseen ja testaukseen. Aspektipohjaisessa ohjelmistokehityksessä koko ohjelmistoa läpileikkaavat ominaisuudet kuvataan paitsi suunnittelussa myös ohjelmakoodin tasolla muusta luokkarakenteesta ja toisistaan erillisinä aspekteina. Näin pyritään muun muassa estämään ohjelmakoodin sekoittuminen mikä johtaa koodin pirstaloitumiseen [2, s. 5].

Aspektipohjaisella ohjelmistokehityksellä saavutetaan parempi modulariteetti koko järjestelmälle, niin funktionaalisille kuin ei-funktionaalisillekin vaatimuksille. Myös ohjelmiston rakenne on ainakin teoriassa helpommin ymmärrettävissä ja koodi selkeämpää, mikä parantaa ylläpidettävyyttä ja koodin päivityksen tehokkuutta merkittävästi. Erityisesti ohjelmistojen muunneltavuuden ja uudelleenkäytön avuksi on kehitetty aspektipohjaisia menetelmiä.

Aspektia voidaan ajatella näkökulmana, joka mahdollistaa useiden jaotteluperusteiden esittämisen. Ohjelmallisesti tämä tarkoittaa sitä, että ohjelmiston läpileikkaavia ominaisuuksia kapseloidaan uusiksi ositusyksiköiksi, aspekteiksi [3, s. 1], jotka punotaan perusohjelmaan joko staattisesti, eli käännöksen aikana, tai dynaamisesti, eli ohjelmaa suoritettaessa. Looginen seuraus on ohjelmiston jakaminen kahteen osaan: perusohjelmaan joka ei ole tietoinen aspekteista ja joka sisältää ohjelman toiminnalliset osat, sekä aspektiohjelmaan, joka sisältää ei-toiminnalliset osat ja kuvaukset niiden liittämisestä perusohjelmaan. Näin ollen muutokset voidaan tehdä vain yhteen paikkaan ilman koko ohjelmiston päivittämistä.



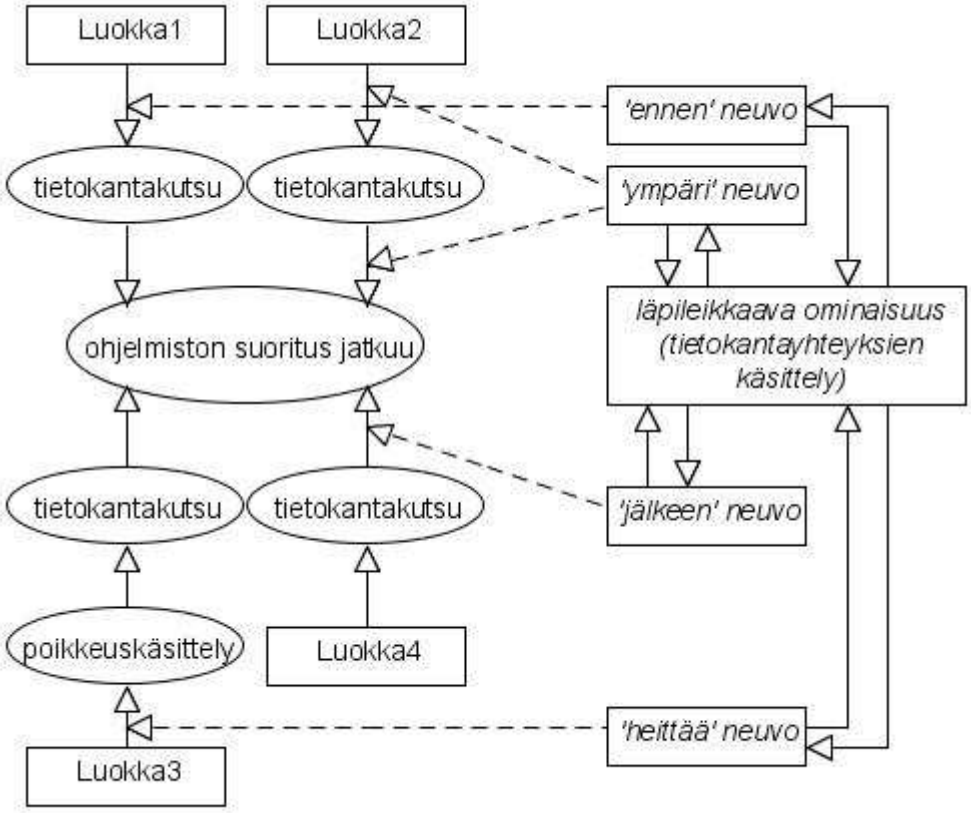
Kuva 5. Läpileikkaavien ominaisuuksien kapselointi aspekteiksi tehostaa ohjelmistokehitystä ja sovelluksen ylläpidettävyyttä verrattuna perinteiseen malliin.

Aspektiohjelmoinnin terminologiassa läpileikkaavat ominaisuudet tarkoittavat niitä ominaisuuksia, joita useammat kuin yksi luokka tarvitsee kuten kuvassa 5 tietokantayhteydet. Vaikka suurin osa olio-ohjelmoinnin luokista suorittaa uniikkia tehtävää, ne myös jakavat sekundaarisia tarpeita muiden luokkien kanssa, eikä niillä voida toteuttaa kaikkia ominaisuuksia modulaarisesti [3, s. 6]. Sekundaariset tarpeet ovat läpileikkaavia ominaisuuksia, ja vaikka eri luokkien toteutukset ja toiminnallisuudet vaihtelevat, pysyvät nämä tarpeet samanlaisina. Edellä mainittu virheenkäsittely on tästä hyvä esimerkki; useat luokat tarvitsevat virheen tai poikkeuksen käsittelyä jossain välissä ajon aikana.

Liitoskohta on se kohta koodissa, milloin käytetään läpileikkaavia ominaisuuksia. Liitoskohta on tyypillisesti metodikutsu, esimerkiksi metodikutsu virheenkäsittelyyn, ja kun tällainen kutsu ilmaistaan, toteutetaan yhdeksi aspektiksi kapseloitu toiminto, eli esimerkkitapauksessa virheen tai poikkeuksen käsittely. Jokainen aspektikieli määrittää oman mallinsa [2, s. 16] sekä toteutuksen liitoskohdille. Niitä on kaikissa niissä luokissa, jotka käyttävät läpileikkaavia ominaisuuksia.

Koska aspekteilla on yleinen pääsy kehitettävään järjestelmään, ne sisältävät myös *neuvon* tilanteesta, milloin läpileikkaavaa ominaisuutta tulee käyttää [4, s. 4]. Käytännössä neuvot ovat ylimääräistä koodia, jotka kertovat ohjelmistolle

milloin ja minkälaisia läpileikkaavia ominaisuuksia missäkin liitoskohdassa tulisi käyttää. Neuvoja on erilaisia; jos neuvo on *ympäri*, liitoskohta ympäröidään eri ominaisuuksilla. Räätälöity koodin käyttäytyminen suoritetaan ennen ja jälkeen metodikutsua, jolloin ohjelmisto voi joko jatkaa liitoskohtaan tai ohittaa sen. Jos neuvon tyyppi on *ennen*, räätälöity koodi suoritetaan ennen metodikutsua, eikä mahdollisuutta liitoskohdan ohittamiseen ole. *Jälkeen*-neuvossa räätälöity koodi suoritetaan metodikutsun jälkeen normaalisti, *heittää*-neuvo taas suoritetaan, jos metodikutsu tarvitsee poikkeuskäsittelyä. Kuva 6 havainnollistaa erityyppisten neuvojen suoritustapaa.

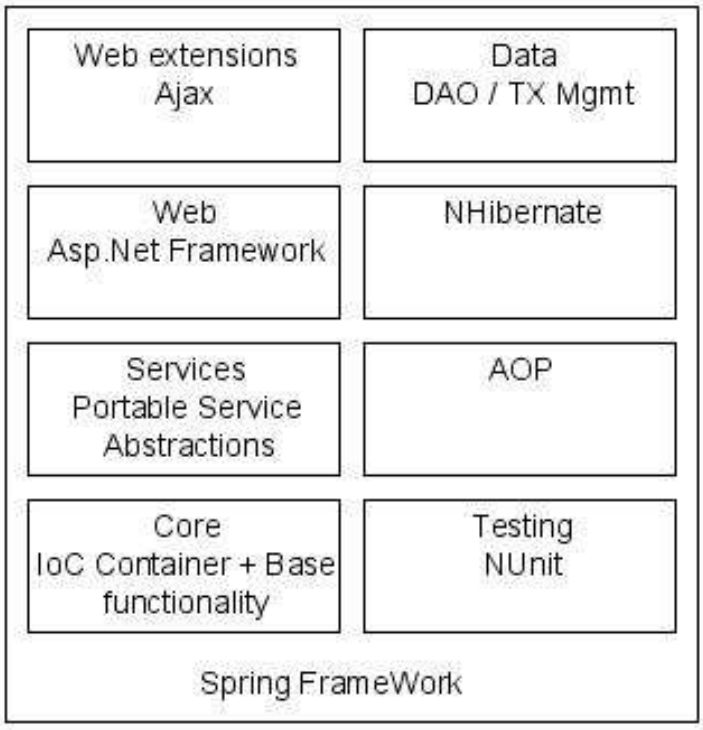


Kuva 6. Neuvot voivat olla erityyppisiä, ja räätälöidyn koodin suoritushetki riippuu neuvon tyypistä.

3 Taustajärjestelmät ja teknologiat

3.1 Spring .Net-ohjelmistokehys

Spring .NET-ohjelmistokehys on SpringSourcen kehittämä laaja, kokonaisvaltainen ratkaisu kyseisessä ohjelmistokehyksessä ajettavien sovellusten kehittämiseen. Se tukee täysin aspektipohjaista ohjelmistokehitystä, ja sen toiminnot ja logiikka pohjautuvat Spring-ohjelmistokehyksen Java-versioon, joka kehitettiin vuonna 2003 ja joka on ollut laajassa käytössä yrityssovellusten kehittämisessä vuodesta 2006. Spring on jaettu eri komponentteihin, joista jokainen hoitaa omia, niille määrättyjä toiminnallisuuksia [5, s. 16] samalla tehostaen ja laajentaen .NET-ympäristöä ja kiertäen sen rajoitteita. Kaikki kehyksen komponentit on esitelty kuvassa 7.



Kuva 7. Spring .NET-ohjelmistokehyksen moduulit

Core-komponentti hoitaa riippuvuusinjektiot koodiin. Normaalisti jos objekti tarvitsee pääsyn johonkin tiettyyn palveluun, joudutaan siihen ohjelmoimaan pääsy kyseiseen palveluun. Riippuvuusinjektion avulla objektilla on vain ominaisuus, joka voi pitää sisällään viittauksen kyseiseen palveluun. Kun komponentti luodaan, ulkoinen mekanismi injektoi automaattisesti ominaisuuden paikalle viittauksen kyseisen palvelun toteutukseen.

Spring AOP:n avulla rakennetaan nimensä mukaisesti aspektipohjaisia sovelluksia. Se hoitaa keskitetysti kaikki yleiset läpileikkaavat toiminnallisuudet ohjelmistossa ja tarjoaa kehittäjän käyttöön aspektikirjaston, joka sisältää laajan valikoiman erilaisia yleisesti käytettyjä aspekteja, kuten virheenkäsittely-, transaktio-, kirjaus- ja suorituskyvyn tarkkailuaspektit. Tämä komponentti on toteutettu dynaamista punomista käyttäen ja kokonaan C#-kielellä, joten erillistä kääntämisprosessia ei tarvita.

Datamoduuli hoitaa tiedon tietokantaan kirjoittamiseen liittyvät operaatiot. Sen avulla saavutetaan tehokkaampi ja toimintavarmempi ADO.NET-kirjaston hyödyntäminen muun muassa mallien, transaktioiden, keskitettyjen yhteyksien ja komentojen kautta. Yksi tärkeimmistä ominaisuuksista on monien ADO.NET-kutsujen kapselointi yhdeksi transaktioksi, joka tehostaa ja yksinkertaistaa tietokantoihin kirjoittamiseen tai niistä lukemiseen liittyvien toimintojen ohjelmoimista. Yksi tämän moduulin alimoduuleista, Spring.Data.NHibernate, hoitaa NHibernaten ja Spring .Net-ohjelmistokehyksen integraation ja mahdollistaa ADO.NET- ja NHibernate-operaatioiden käytön yhdellä transaktiolla.

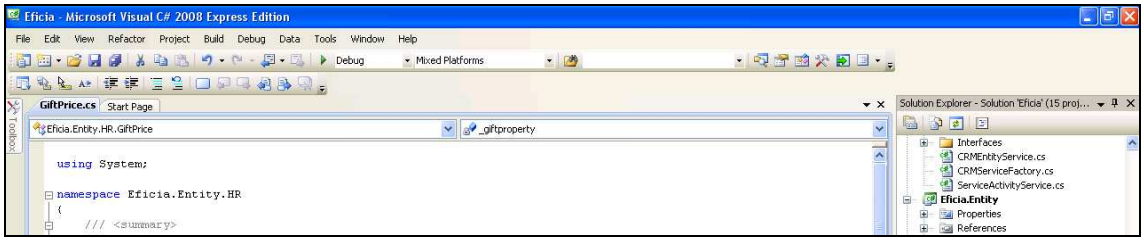
Web-komponentilla saavutetaan korkeampi abstraktiotaso ASP.NET-verkkosovelluksia kirjoitettaessa. Se tuo helpotusta moniin yleisiin ongelmakohtiin kuten tiedon sitomiseen ja vahvistamiseen. Web-komponentti tukee kahdensuuntaista sidontaa ja riippuvuusinjektiota kaikkiin ASP.NET-artefakteihin. Myös käyttöliittymän lokalisointi on helpompaa komponentin avulla.

Edellä esiteltyjen moduulien lisäksi Spring .NET-ohjelmistokehys tarjoaa lukuisan määrän muita .NET-kehitystä tehostavia komponentteja, ja laajaa integraatiota sekä mallikirjastoja tarjoten se nopeuttaa ja yksinkertaistaa ohjelmistojen kehitystä, mahdollistaen näin modulaarisemman ja helpommin hallittavan koodauksen. Ohjelmoija saa päättää, hyödyntääkö kaikkia ominaisuuksia vai vain osaa niistä, mikä parantaa kehityksen joustavuutta edelleen. Aspektiohjelmoinnin hyödyntämisessä Spring on yksi markkinoiden johtavista ratkaisuista.

3.2 MicroSoft Visual Studio kehitysympäristö

MicroSoftin kehittämä Visual Studio on käytännössä standardi kehitysympäristö .Net-ohjelmoinnissa. Sen ensimmäinen versio julkaistiin vuonna 1997, ja silloin tuettuja kieliä olivat C++, J++, FoxPro ja visual basic. Vuonna 2002 julkaistiin .Net-sovelluskehys versio 1.0, jolloin Visual Studio rakennettiin tukemaan kyseiselle alustalle suunnattua sovelluskehitystä. Tämän jälkeen uuden .Net-kehyksen version ja Visual Studion uuden version julkistamiset ovat tapahtuneet käsi kädessä. Visual Studio 2008 julkaistiin vuonna 2007, ja se vaatii toimiakseen samanaikaisesti julkistetun .Net-kehyksen version 3.5.

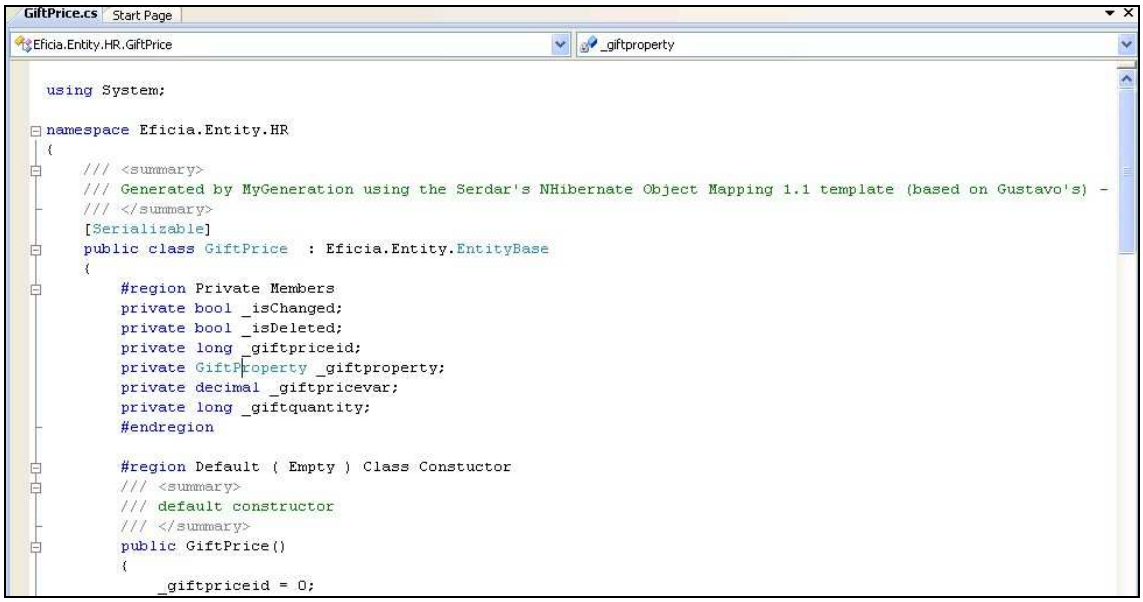
Periaatteessa ohjelman tuottamiseen tarvitaan vain tekstieditori ja kääntäjä [6, s. 13], mutta näin tuotettu koodi on hidasta ja vaivalloista kirjoittaa. Visual Studio on täysin integroitu kehitysympäristö (IDE, integrated development environment). Se sisältää sekä graafisen käyttöliittymän että konsolin ohjelmiston rakentamiseen sekä koodieditorin, kääntäjän ja kattavat työkalut poikkeusten käsittelyyn. Ohjelmistolla pystytään rakentamaan muun muassa konsoliohjelmia, graafisia .Net-ohjelmistoja hallinnoidussa ympäristössä tai sovelluksia alkuperäisessä ympäristössä, sekä verkko-sovelluksia ja -palveluita. Ohjelmisto on myös helposti laajennettavissa, ja siihen löytyy useita kolmannen osapuolen tuotteita täydentämään jo alkujaankin laajaa luokkakirjastoa ja suurta määrää työkaluja. Kuvassa 8 näytetään osa Visual Studion käyttöliittymästä.



Kuva 8. Visual Studio express 2008-kehitysympäristön ikkunat ja työkalurivit.

Ohjelmisto tukee ilman laajennuksia MicroSoft Visual C++-, MicroSoft Visual C#- ja MicroSoft Visual Basic kieliä. Sen arkkitehtuuri on suunniteltu siten, että toiminnallisuutta pystytään laajentamaan VS-paketeilla; kun paketti on asennettu, se tarjoaa IDE:n käyttöön palvelun kyseisen paketin sisältämään toimintoon. IDE tarjoaa ohjelmoijalle käyttöön kolme peruspalvelua: projektien ja ratkaisujen luonnin ja hallinnoinnin, käyttöliittymän sekä VS-pakettien tuomien palveluiden hallinnan.

Koodieditori tarjoaa kaikki normaalit, nykyaikaiset apuvälineet tehokkaaseen ohjelmointiin, kuten kuvasta 9 voidaan nähdä. Visuaaliset tehosteet, kielen syntaksin mukainen automaattinen asettelu ja automaattiset täydennystoiminnot nopeuttavat koodin tuottamista ja vähentävät kieliopin vastaisia virheitä. Williamsin mielestä [6, s. 451] Visual Studio on kuin prosessori joka tietää mitä ohjelmoija haluaa sanoa seuraavaksi. VS-paketeilla editori saadaan laajennettua ymmärtämään lukuisten eri kielien syntaksia. Tausta-ajotoiminnon avulla kirjoitettavaa ohjelmaa voidaan ajaa taustalla samanaikaisesti, kun siihen kirjoitetaan editorilla lisää koodia. Näin saadaan tietoa käännöksen aikana tulevista virheistä sekä kielioppivirheistä ennen itse kääntämistä.



Kuva 9. Visual Studion koodieditori ymmärtää liitännäisten avulla useiden eri kielten syntakseja. Kuvassa editori viritetty C#-kielen kieliopin mukaiseksi.

Koodieditorin lisäksi Visual Studiosta löytyy lukuisia valmiita suunnittelukomponentteja VS-paketteina. Suunnittelijat ja erilaiset mallit löytyvät muun muassa Windows-lomakkeille, verkkolomakkeille, luokille ja tietokantayhteyksille. Nämä ovat graafisia, käyttäjää opastavia, osittain tai kokonaan automatisoituja työkaluja, jotka tuottavat valmista koodia kääntäjän käyttöön. Tällainen laajasti automatisoitu, moderni kehitysympäristö siirtää ajan käytön painopisteen itse koodin tuottamisesta pikemminkin työkalujen käyttöön ja laajojen kokonaisuuksien hahmottamiseen ja suunnitteluun.

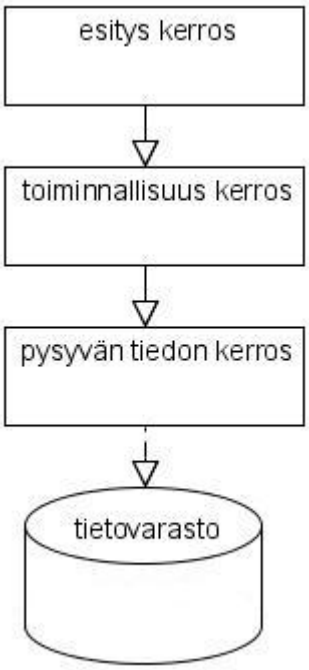
Poikkeuskäsittelyyn löytyy Visual Studiossa laajat, tehtävää helpottavat työkalut, vaikka tällaista valmista ominaisuutta ei ohjelmistosta oikeastaan löydy [6, s. 599], ja ohjelmoijat kirjoittavat tyypillisesti omat käsittelyt tarvittaville poikkeuksille. Valmiit työkalut tukevat konekielisten komentojen, hallitun .Net-koodin ja natiivin koodin virheidenhavainnoinnin, ilmoittamisen ja korjaamisen käyttäjän ohjelmoimalla tavalla. Muistista pystytään ottamaan vedos, jolla voidaan jäljittää esimerkiksi muistin ylivuotoja, ja ohjelmaa pystytään ajamaan joko komento tai lohko kerrallaan. Virheilmoituksista löytyy laajalti tietoa joko verkkokirjastosta tai ohjelmiston mukana asennetusta paikallisesta katalogista,

ja usein tietolähteet tarjoavat myös erilaisia malliratkaisuja kyseiseen virhetilanteeseen.

3.3 NHibernate ja tietokantayhteydet

NHibernate on avoimen lähdekoodin objektienkartoitusohjelma .Net-ympäristöön. Avoin lähdekoodi tarkoittaa vapaan jakelun ohjelmistoa, jonka lähdekoodi on saatavilla. NHibernate tarjoaa huomattavasti tehokkaamman ADO.Net SQL-tietokantayhteyksien hallinnoinnin kartoitustiedostojen avulla. Sovelluksen läpinäkyvyyttä lisäten kaikki rutiininomaiset tietokantakutsut voidaan tuottaa automaattisesti ja tietokanta-taulut sekä -solut voidaan kirjoittaa kartoitustiedostoon, jolloin muutokset tarvitsee tehdä vain yhteen paikkaan [7, s. 1]. Kyseistä toiminnallisuutta voitaisiin kuvata tietokantayhteyksien hoitamisen aspektiksi; toiminnallisuus on selkeästi erillään itse suorittavasta koodista XML-tiedostossa, ja tietokantayhteyksien ollessa liitoskohtia, nämä kartoitustiedostot toimivat läpileikkaavina ominaisuuksina.

Normaalisti olio-ohjelmaa kirjoitettaessa jokaiselle oliolle joka sisältää tallennettavaa tietoa, luodaan omat tietokantayhteydet sisältävät luokat ja käyttöliittymät. Nämä sisältävät tarvittavat tietokanta -kutsut ja–kyselyt ja niihin on kirjoitettu tiedon osoite tietokannassa (taulu ja solu). Näitä luokkia ja käyttöliittymiä on laajassa sovelluksessa useita kymmeniä, jolloin niiden hallinnointi ja päivitys on työlästä. NHibernatea käytettäessä ohjelmoija voi kirjoittaa kartoitustiedostoon kaiken tarvittavan muuttuvan tiedon SQL-tietokantayhteyksistä, jolloin ohjelmisto tuottaa tarvittavat luokat määrittelyineen niiden pohjalta. Tällaista mallia voidaan kuvata kerroksittaiseksi arkkitehtuuriksi jossa NHibernate tuottaa alimman tason pysyvän tiedon kerroksen. Kerroksittainen malli on havainnollistettu kuvassa 10.



Kuva 10. Kerroksittaisessa mallissa kerrokset ovat tietoisia vain suoraan niiden alla olevasta kerroksesta.

Kyseisessä mallissa pyritään eristämään ohjelman eri toiminnallisuudet ja ongelmat eri kerroksille [8, s. 8]. Jokainen näistä kerroksista kommunikoi vain alemman tason kerroksen kanssa ja on tietämätön muista kerroksista. Näin yhteen kerrokseen voidaan tehdä helpommin muutoksia häiritsemättä muiden kerrosten toimintaa. Yleisesti kerrokset ajatellaan seuraavanlaisesti; Ylinnä on esityskerros, joka sisältää käyttöliittymän ohjelmistoon. Tämän alta löytyy toiminnallisuuskerros, joka sisältää kaiken älyn ja tiedonkäsittelyn kuten laskentaoperaatiot. NHibernate luo toiminnallisuuskerroksen alle yhden kerroksen lisää tietokantayhteyksien väliin, pysyvän tiedon kerroksen. Tämä kerros hoitaa kaiken liikennöinnin ohjelmiston ja tietovaraston välillä.

Kartoitustiedostot, jotka toimivat osana pysyvän tiedon kerrosta, ovat puhtaasti XML-tiedostoja, joten niitä voi muokata ja luoda esimerkiksi MicroSoftin Visual Studiolla sekä lukuisilla muilla koodieditoreilla. XML-muotoiset tiedostot ovat myös luettavissa sellaisenaan kaikissa internet-selaimissa. Kartoitustiedostoon

kirjoitetaan halutun tiedon tyyppi sekä sitä vastaava tietokantataulun solun tyyppi ja osoite. Tiedon tyyppi voi olla esimerkiksi verkkokaavakkeen tietyn tekstikentän arvo, joka luetaan muuttujaan. Kuvassa 11 on esimerkki tällaisesta tiedostosta.

```
<?xml version="1.0" encoding="utf-8" ?>
- <hibernate-mapping xmlns="urn:nhibernate-mapping-2.2">
  - <class name="Eficia.Entity.HR.GiftPrice,Eficia.Entity" table="GiftPrice" lazy="true">
    - <id name="_GiftPriceid" column="GiftPriceID" type="Int64" unsaved-value="0">
        <generator class="assigned" />
      </id>
      <many-to-one name="_GiftProperty" column="GiftProperty" class="Eficia.Entity.HR.GiftProperty,Eficia.Entity" />
      <property column="GiftPrice" type="Decimal" name="_GiftPriceVar" />
      <property column="GiftQuantity" type="Int64" name="_GiftQuantity" />
      <property column="CreatedBy" type="String" name="_CreatedBy" length="130" />
      <property column="CreatedOn" type="DateTime" name="_CreatedOn" />
      <property column="UpdatedOn" type="DateTime" name="_UpdatedOn" />
      <property column="UpdatedBy" type="String" name="_UpdatedBy" length="130" />
    </class>
  </hibernate-mapping>
```

Kuva 11. Esimerkki NHibernaten kartoitustiedostosta.

4 Aspektiohjelmointi

Aspektiparadigma perustuu Elradin [9, s. 2] mukaan ajatukseen, jossa tietokonejärjestelmien kehitys on tehokkaampaa, kun järjestelmästä eriytetään erillisiksi kokonaisuuksiksi järjestelmän eri vaatimukset tai mielenkiinnonkohteet ja kuvaukset niiden välisistä suhteista. Tämän jälkeen annetaan Aspektiympäristön mekanismien punoa tai koostaa nämä kohteet muuhun ohjelmistokoodiin, jolloin saadaan halutunlainen sovellus. Mielenkiinnonkohteet voivat vaihdella korkean tason käsitteistä, kuten koko ohjelmiston tietoturvasta tai toimintavarmuudesta, matalan tason operaatioihin, kuten välimuistin hallintaan.

Laddad sanoo [10, s. 90] aspektiohjelmoinnin olevan vallankumouksellinen harppaus kohti parempaa toteutuksen ymmärrettävyyttä ja uusien vaatimusten täyttämistä, sekä muutosten toteuttamista olemassa oleviin komponentteihin. Tällainen järjestelmällinen lähestymistapa tarjoaa vaatimusten ja suunnittelutavoitteiden suoran kuvauksen itse toteutukseen. Jos läpileikkaavat ominaisuudet on määritelty ja suunniteltu oikein ja toteutettu tehokkaasti, epäsuorana vaikutuksena kehitysresursseja vapautuu itse päätoiminnallisuuksien hiomiseen ja niiden laadun parantamiseen. Kaikki nämä ominaisuudet yhdessä parantavat tuotettavien ohjelmistojen laatua ja kehitystyön tehokkuutta.

Kaikki eivät ole yhtä mieltä aspektiohjelmoinnin hyödyllisyydestä, muun muassa Steimann [11, s. 1] esittää paradigman suosion perusteeksi modulariteetin ja ohjelman rakenteen parantamiseen, vaikka tosiasiassa se lisää ohjelmistojen monimutkaisuutta ja itsenäistä kehitystä. Koska tähän ei löydy Steimannin mielestä ratkaisua, hän pitää aspektiohjelmoinnin suosiota paradoksaalisena.

Koska aspektiohjelmointi on vielä nuori paradigma, ei sen hyödyllisyyttä isoissa sovelluksissa pystytä täysin vielä mittaamaan [4, s. 6]. Se ei ole teknologiana vielä kypsä, ja avoimina kysymyksinä säilyvät esimerkiksi, miten se parantaa

ohjelmistojen ylläpidettävyyttä ja päivitystä pitkällä aikavälillä, tai miten tehokkuuden parantumista mitataan käytännössä.

4.1 Vaiheet

Aspektiohjelmoinnissa pyritään erittelemään ohjelmiston ominaisuudet kahteen eri luokkaan: ydinominaisuuksiin ja poikkileikkaaviin ominaisuuksiin [4, s. 3]. Ydinominaisuudet ovat sovelluksen toimintaan liittyviä komponentteja ja niitä kutsutaan yleisesti myös liiketoiminnallisiksi ominaisuuksiksi. poikkileikkaavat ominaisuudet ovat järjestelmän toiminnallisuuksia, jotka liittyvät useampiin komponentteihin. Ne ovat toissijaisia komponentteja, joiden toiminta-alue voi koskettaa useita ydinominaisuuksia. Tällaisia komponentteja ovat esimerkiksi virheen- tai poikkeuskäsittelyn hoitavat komponentit.

Sorsan mielestä [3, s. 13] poikkileikkaavien ominaisuuksien erottelun kyvyttömyys suunnittelu- ja toteutusvaiheessa johtavat vaikeasti ylläpidettävään ja ymmärrettävään ohjelmistoon. Tämän seurauksena ohjelmiston rinnakkainen kehitystyö ja laajennettavuus kärsivät. Ratkaisuksi Jacobsonin ja Ngin [12, s. 3] mukaan riittää ohjelmistokehityksen apuväline, joka on kykeneväinen tällaisten ominaisuuksien erittelyyn ohjelmakooditasolla, sekä yksinkertainen ja tehokas paketointimekanismi, jotta kunkin vaatimuksen suunnittelu ja toteutus luo lopputuloksena halutunlaisen järjestelmän.

Kun erittely on määritelty suunnitteluvaiheessa, toteutetaan kapselointi aspekteiksi koodausvaiheessa erillisellä aspektikielellä. Ydinohjelmisto on tyypillisesti toteutettu oliokielellä, ja sen sisältämät toiminnallisuudet on ohjelmoitu eri tiedostoihin kuin poikkileikkaavat ominaisuudet. Näin saavutetaan ohjelmoijan kannalta ainakin teoreettisesti olennaista selkeyttä ohjelman ymmärrettävyyteen, vaikka Daigle ja Akhlagh [13, s. 1] esittävät poikkileikkaavan olevan terminä harhaanjohtavan ja aiheuttavan sekaannusta

ohjelmoijien keskuudessa, sillä ihmiselle luonnollisemman ajatusmallin mukaan komponentit mielletään helpommin olioiksi.

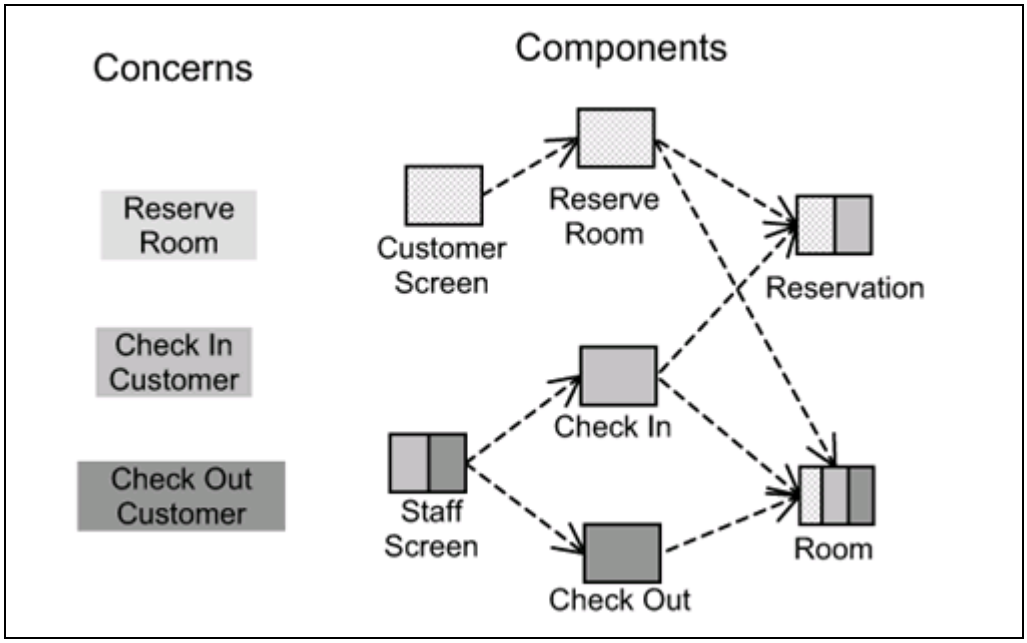
Ohjelmistokehityksessä koodauksen jälkeen on normaalisti vuorossa ohjelmiston testaus ja virheenkorjaus. Tietäväisen mielestä [14, s. 44] aspektiohjelmoinnilla on testaukseen kaksiosainen vaikutus: aspektit voivat auttaa testauksessa ja rutiineja pystytään sen avulla tehostamaan, mutta toisaalta aspektit itsessään pitää pystyä testaamaan. Koska aspektiohjelmointi on vielä uusi teknologia, ei siitä ole pystytty löytämään kokemuksen kautta kaikkia ongelmakohtia. Vianselvityksen täytyy olla yksilöllistä jokaiselle aspektille ja sen ominaisuuksille, mikä johtaa haastavaan testaukseen. Perinteisestä virheenkorjauksesta aspektipohjaisen ohjelmiston korjaus eroaa ainakin liitoskohdissa: myös kaikki ydinohjelmistoon punotut aspektit pitää pystyä testaamaan ja löytämään niistä mahdolliset virheet.

4.2 Määritelmä

Aspektiohjelmoinnilla voidaan Tietäväisen mukaan [14, s. 1] tarkoittaa kahta erilaajuista termiä. Suppeampi merkitys pitää sisällään lähinnä ohjelmointia jollain aspektipohjaisella kielellä, ja laajempi merkitys voidaan käsittää sateenvarjotermiksi, joka pitää sisällään käsitteet mukautuva ohjelmointi, koostesuotimet ja moniulotteinen ohjelmointi. Daigle ja Akhlaghi [13, s. 1] esittävät, että AOP on helpoin määritellä joko toteutuksen tai aspektikielien kautta. Aspektikielet mahdollistavat ominaisuuksia ja konsepteja mitä proseduraalisissa tai olio-kielissä ei ole mahdollista toteuttaa.

Alexander ja Bieman [15, s. 3] toteavat, että perusajatuksen mukaan aspektiohjelmoinnissa kaikkia asioita ja ominaisuuksia tulisi kohdella modulaarisina yksikköinä piittaamatta ne toteuttavan kielen rajoituksista. Heidän mukaansa pääasiallinen mekanismi, jolla määritellään läpileikkaavia ominaisuuksia, on aspekti.

Yhteisiä piirteitä kaikille eri käsityksille aspektiohjelmoinnista ovat liitoskohdat koodissa ja niissä ajettavat läpileikkaavat ominaisuudet. Kaikki ohjelmointikielet tukevat kapselointia ja käsitteiden ryhmittelyä jollain tasolla, mutta aspektikielet kapseloivat läpileikkaavat ominaisuudet omiksi toiminnallisuuksiksi, joita ei perinteikkäämmillä ohjelmointikielillä pystytä saavuttamaan yhtä toimintavarmasti tai ollenkaan. Sorsan [3, s. 14] mielestä tähän on syynä poikkileikkaavien ominaisuuksien ohjelmakoodin pirstaloituminen ja sekoittuminen ympäri ydinohjelmistoa, jolloin sen hallinta, ymmärrettävyys ja muokattavuus kärsivät. Näistä ongelmista on esitetty esimerkki kuvassa 12.



Kuva 12. Hotellin hallintajärjestelmässä erilliset komponentit aiheuttavat pirstaloitumista ja sekoittumista. [12]

Paradigma voidaan määritellä myös tavoitteiden kannalta; Elrad ja Al [9, s. 31] kiteyttävät AOP:n ohjelmointitekniikaksi, jonka tehtävänä on, niin kuin kaikkien muidenkin ohjelmointitekniikoiden, ottaa huomioon ohjelmoijan mahdollisuudet kertoa tietokoneelle koodilla halutunlaiset operaatiot ja toteuttaa toimiva järjestelmä. Tästä syystä aspektipohjaisten järjestelmien täytyy pystyä poikkileikkaavien ominaisuuksien esityksen ja niiden toteuttamisen lisäksi takaamaan mekanismien toimivuus ja käsitteellinen yksinkertaisuus.

4.3 Käsitteet

4.3.1 Vaatimusten jäsentely

Ohjelmistokehityksen alkuajoista lähtien, eri mielenkiinnonkohteet tai eri toiminnallisuudet on pyritty eriyttämään toisistaan. Tällä eriyttämisellä, tai vaatimusten jäsentelyllä, (SOC, *Separation of concerns*) pyritään ennen kaikkea parantamaan kaikkien osien modulaarisuutta ja laatua [16, s. 1]. Sillä saavutetaan myös parempi ohjelmakoodin hallinta ja päivitys sekä yksinkertaistetaan ohjelmistorakennetta. Korkean tason kielillä eriyttäminen saattoi tarkoittaa eri toiminnallisuuksien pilkkomista funktioihin, olio-ohjelmoinnissa jokin luokka hoitaa sille varattua tehtävää ja aspektiohjelmoinnissa eriytetään sekundaariset, poikkileikkaavat ominaisuudet omaksi mielenkiinnon kohteeksi.

Käsitteenä SOC juontaa juurensa vuoteen 1982, milloin Dijkstran [17, s. 61] mukaan sitä ei ollut vielä täydellisesti mahdollista toteuttaa, mutta se tarjosi ainoan tehokkaan tavan jäsentää ominaisuudet ja vaatimukset ihmisten ajattelussa. Hänen mukaansa muita aspekteja tai mielenkiinnonkohteita ei tullut sivuuttaa, mutta ajateltaessa asiaa aspektin näkökulmasta, muut mielenkiinnonkohteet olivat merkityksettömiä. Näin kapseloitu ajattelumalli on osoittautunut myöhemmin ohjelmistoteollisuuden ja -paradigmojen kulmakiveksi, ja sitä on pyritty tehostamaan ja laajentamaan jatkuvasti.

4.3.2 Liitoskohta ja liitoskohtamäärittely

Liitoskohta on se kohta koodissa, milloin ohjelmiston poikkileikkaavaa ominaisuutta käytetään. Mikä tahansa kohta ohjelmarakenteessa ei ole

soveltuva liitoskohdaksi, vaan kuten Bradley ja Alexander [18, s. 13] toteavat, liitoskohdat ovat hyvin määriteltyjä kohtia ohjelman suorituksessa. Näitä ovat esimerkiksi metodin tai funktion alku- tai loppukohta, objektin luominen tai poikkeuskäsittelyn ajaminen.

Aspektikielissä tärkein osa on liitoskohtamalli tai määrittely (join-point model). Tapanaisen [3, s. 14] mukaan tämä malli määrittelee kielen sallimat liitoskohdat, mitä tietoja nämä kohdat paljastavat, miten liitoskohtamääritykset voivat poimia liitoskohtia sekä miten neuvot vaikuttavat liitoskohtiin. Aspektikieli määrittää nämä sallitut liitoskohdat peruskielen kieliopista ja semantiikasta, kuten myös sen, mitä tietoja liitoskohdasta paljastetaan aspektille.

Liitoskohtamallin sisäistäminen on merkitsevin konsepti aspektipohjaisen ajattelutavan omaksumisessa. Ohjelmoijien tärkein tehtävä on liitoskohtien määrittely sovellusta suunniteltaessa, ja aspektipohjaisen kielen päätehtävä on toteuttaa sopiva kuvaus liitoskohdille [19, s. 1].

4.3.3 Neuvo

Neuvo sisältää ohjelmakoodin, joka määrittelee miten liitoskohdassa toimitaan. Yhdessä neuvot ja liitoskohtamääritys kapseloivat läpileikkaavan toiminnallisuuden [14, s. 27]. Neuvot koostuvat kolmesta osasta: esittelystä, liitoskohtamäärityksen kuvauksesta ja ohjelmakoodia sisältävästä rungosta. Esittelyssä kuvataan, milloin koodi suoritetaan, esimerkiksi ennen liitoskohtien suoritusta. Kuvassa 6 on esitelty kaikki erityyppiset neuvot. Neuvot voidaan määritellä siten, että niiden sisältämä koodi ajetaan liitoskohdan ohjelmakoodin sijasta. Tällaista ominaisuutta kutsutaan käärimiseksi (wrapping), ja jokainen aspektikieli määrittää omat sääntönsä kyseiselle ominaisuudelle. [2, s. 19]

4.3.4 Punominen

Punomisella tarkoitetaan mekanismia, jolla liitetään läpileikkaava ominaisuus liitoskohtaan; ne eivät vaikuta ydinohjelmaan itsenäisesti. Tapanaisen [3, s. 12] mukaan liittäminen voidaan toteuttaa eri tavoilla, mutta lopputuloksena neuvon vaikutuksen tulee esiintyä kaikissa liitoskohtamäärityksen osoittamissa liitoskohdissa. Punoja on väline tai mekanismi aspektikielessä, joka hoitaa punomisen.

Staattinen punominen tarkoittaa läpileikkaavan koodin sitomista pääohjelmistoon ohjelmakoodia käännettäessä suoritettavaksi ohjelmaksi, dynaaminen punominen sitä, että koodin punonta suoritetaan ajon aikana. Tässä työssä projektivaiheessa käytetty Spring .Net-ohjelmistokehys käyttää dynaamista punomista. Saton [20, s. 9] mielestä siinä ei ole kyse mitenkään vaativasta konseptista, vaan AOP:n käytännöstä, joka on hyvin samankaltainen olio-ohjelmoinnin dynaamisen metodien sitomisen eli polymorfismin kanssa.

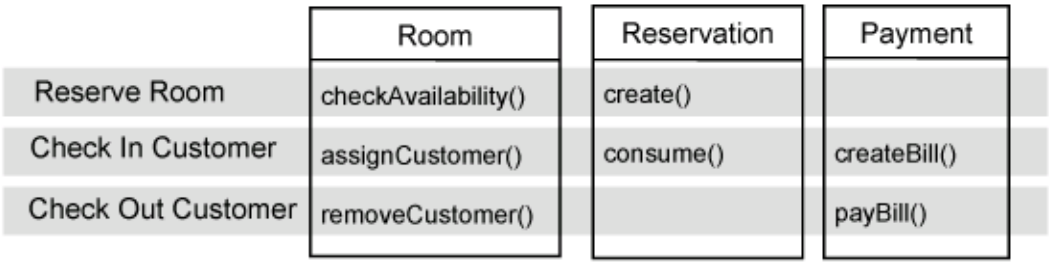
Dynaamisen punomisen eduksi voidaan lukea aspektien poistaminen tai ohjelmasta poispunonta ajonaikaisesti, jos tarvetta tämän kaltaisiin operaatioihin ilmenee [18, s. 15]. Ongelmakohtina voidaan pitää punomisen testausrutiinien ja niistä saatujen luotettavien tulosten vähäisyyttä [15, s. 6].

4.3.5 Aspekti

Aspektit voidaan mieltää käsitteellisiksi kokonaisuuksiksi, jotka koskettavat ohjelmistoa globaalisti. Ne eivät ole riippuvaisia abstraktiotasosta, eikä niiden toteutus ole aina yksiselitteistä. Tämä tekee niistä hankalasti hahmotettavia ohjelmoijalle. Harman ja Black [21, s. 3] toteavat Dooyeweerdin teoksen ”Theory of Aspects” muodostavan filosofisen lähtökohdan aspektiohjelmoinnille ja aspektiperusteiselle ohjelmistokehitykselle.

Koska aspektit eivät ole systeemistä aina helposti löydettävissä, on olennaista, että ne pyritään määrittelemään jo suunnitteluvaiheessa. Jos tätä ei kuitenkaan ole tehty, voidaan järjestelmästä löytää aspekteja erityisesti tähän tarkoitukseen valmistetuilla työkaluilla. Tällaista menettelytapaa kutsutaan louhimiseksi [14, s. 38], ja se voidaan kohdentaa itse ydinohjelmistoon, suunnitelmaan, algoritmeihin, järjestelmän määritykseen ja niin edelleen.

Käsitettä voidaan havainnollistaa Jacobsonin ja Ng:n [12, 2.2] esimerkillä hotellin hallintaohjelmasta; Kuva 13 ilmaisee heidän näkemyksensä luokista, jotka edustavat eri mielenkiinnonkohteita.



Kuva 13. Hotellin hallintajärjestelmän eri mielenkiinnonkohteita ohjelmistoluokkina [12].

Kuvassa huoneen varaaminen ja asiakkaan kirjautuminen sisään tai ulos ovat aspekteja, jotka koskettavat kaikkia kolmea luokkaa, joita transaktioihin tarvitaan.

4.4 Hyödyt ja haitat

Ainakin teoriassa aspektiohjelmointi tarjoaa huomattavia parannuksia ohjelmistokomponenttien uudelleenkäytettävyyteen, parantaen samalla niiden laatua. Kyseisten ominaisuuksien toteuttaminen on ollut ohjelmistoteollisuuden haaste alusta alkaen, ja aika näyttää, saavutetaanko käytännössä aspektiohjelmoinnin teoreettinen lupaus erittäin modulaarisista ohjelmistoista. Bradleyn [18, s. 5] mielestä aspektiohjelmoinnin tarjoamilla tekniikoilla voidaan

helposti ja tehokkaasti saavuttaa tämä korkeampi modulariteetti liittämällä läpileikkaavat ominaisuudet ydinohjelmistoon paradigman määrittelemällä tavalla.

Koska paradigma on vielä nuori, kaikkia epäkohtia ei ole saatu ratkaistua tai määriteltyä. Steimannin [11, s. 493] mielestä myös koko ideologia, missä koetetaan modularisoida mahdottomat kohteet, on heikko. Hänen mielestään AOP:n tulisi tämän sijasta keskittyä tukemaan muita ohjelmistomalleja omilla vahvuuksillaan, sillä aspektiohjelmoinnin omat mekanismit rikkovat modulaarisuuden ja ovat näin ollen paradoksaalisia. Myöskään olio-ohjelmointiin verrattuna heikko määrittely ei saa kiitosta Daiglelta ja Akhlaghilta . [13, s. 1]

AOP:n ja siihen liittyvien työkalujen yleistyessä ja kehittyessä se voi olla vastaus yhteen ohjelmistokehityksen haasteellisimmista ongelmista; Tehokkaan ja uudelleenkäytettävän koodin kehitykseen ja toteuttamiseen. Näin uskoo myös Elrad [9, s. 32], joka odottaa tulevaisuuden tuovan tullessaan lisää aspektipohjaisia sovelluksia, ja mainitsee paradigman suosion esimerkiksi siihen keskittyvän konferenssin, joka järjestetään vastaavan, olioparadigmaan keskittyvän, konferenssin kanssa vuosittain.

5 Liikelahjakomponentti

5.1 Alkuperäinen liikelahjarekisteri

Liikelahjakomponenttiprojekti käynnistyi Efician asiakkaan määritellessä tarpeen uudelle komponentille, joka korvaisi käytössä olevan vanhentuneen liikelahjarekisterin [22]. Vanha järjestelmä oli erillisratkaisu, joka oli erittäin vaikeasti hallittavissa ja päivitettävissä ja toteutettu epästandardein ratkaisuin. Pääosin rekisteri oli ohjelmoitu ASP-konseptia käyttäen ja verkkolomakkeisiin upotettua javascriptia hyödyntäen. Sovellus ei sisältänyt tietokantaa, vaan tiedot tallennettiin yksinkertaisesti tiedostoihin. Käyttäjinä liikelahjarekisterissä toimivat asiakkaan muutaman osaston sihteerit, jotka vastasivat yrityksen liikelahjojen hallinnasta.

Liikelahjarekisteri oli toiminnallisuuksiltaan heikko: hakutoiminnot olivat epäluotettavat ja suppeat, visuaalinen ilme oli erittäin pelkistetty ja epälooginen, käyttöliittymä hankala ja toiminnallisuudet sekä raportointiominaisuudet riittämättömät. Esimerkiksi tuoteartikkelien lisääminen tai hintojen muuttaminen oli erittäin hankalaa, eikä kustannusseurantaa pystytty toteuttamaan osastokohtaisesti raporttien perusteella. Ison suuryrityksen ollessa kyseessä myös muihin järjestelmiin integroituja käyttäjätietoja olisi kaivattu; henkilöstömuutokset aiheuttivat joka kerta rekisterin päivittämistä manuaalisesti.

Ohjelmakoodin tasolla sovellus ei myöskään vakuuttanut. Koodaus oli erittäin epästandardimaisesti suoritettu eikä koodia ollut juuri dokumentoitu. Laajennusmahdollisuudet olivat sellaisenaan heikot, sillä alun perin rekisteri oli tarkoitettu vaatimattomampaan ja pienimuotoisempaan käyttöön. Rekisterin tarkastamisen ja arvioinnin jälkeen projektiryhmässä tultiin siihen lopputulokseen, että päivitysprojekti ei ole kannattava, vaan Eficia Coreen ohjelmoitava komponentti toteuttaisi asiakkaan määrittelemät vaatimukset paremmin. [22]

5.2 Uusi komponentti

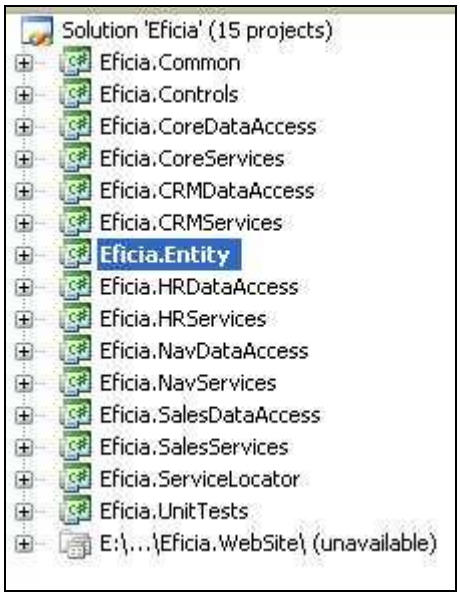
Tarpeen ja vaatimusten ollessa selvillä päätettiin toteuttaa uusi komponentti, joka liitettäisiin osaksi asiakkaalla tuotantokäytössä olevaa Eficia Corea. Projektiryhmään kuuluivat minun lisäkseni Eficialta ohjelmistokehittäjä Tommi Kemppi ja ohjelmistoarkkitehti Timo Saikkonen sekä asiakkaan edustaja. Ryhmä oli yksimielisesti uuden komponentin kannalla, koska vanha oli heikosti päivitettävissä.

Asiakas määritteli uuden komponentin päätoiminnoiksi kattavat, osastokohtaiset tai yksilöidyt raportit, laajat ja helppokäyttöiset hakutoiminnot, yksinkertaisen liikelahja-artikkelien muokkaamiseen, lisäämisen ja poistamisen sekä ”anna liikelahja”-transaktion. Lisäksi komponentin visuaalinen ilme ja käyttöliittymä suunniteltiin yhdessä Efician edustajien kanssa. Tavoitteena oli kustannusseurannan mainittava parantuminen ja varastosaldojen laadukas ylläpito ilman jatkuvaa inventointitarvetta kuten vanhan järjestelmän kanssa [22].

Komponentin suunnittelutyöhön vaikuttivat vaatimusten lisäksi ohjelmointiympäristö ja valitut teknologiat. Core-ohjelmiston myötä tietokannaksi valikoitui MicroSoftin SQL-palvelin, ohjelmointikieliksi C#, javascript ja XML sekä käyttöliittymäksi internet-selain, jossa ajetaan ASPX-verkkolomakkeita. Saikkosen [23] mukaan Coren soveltaessa osaan toiminnoistaan aspektiparadigmaa liikelahjakomponentin käyttöön tulivat automaattisesti muun muassa tietokantayhteyksienkäsittely-, poikkeustenkäsittely- ja käyttäjätunnistusaspektit. Suunnitteluvaiheessa olennaisinta oli Coren vahvojen liitäntämahdollisuuksien ja laajennettavuuden tunteminen, sillä vahvasti virtualisoitu ohjelmisto tarjosi erittäin uudelleenkäytettäviä ratkaisuja eri komponenttien liittämiseksi.

5.3 Ohjelmiston rakenne

Eficia Core on jaettu 15:een eri nimiavaruuteen toiminnallisuuksien mukaan. Jokainen nimiavaruus sisältää useita nimiavaruuksia, esimerkiksi Eficia.Website-nimiavaruus pitää sisällään kaikki ASPX-tyyppiset verkkolomakkeet, Eficia.Entity kaikki entiteettiluokat jne. Kuvassa 14 on listaus Eficia Coren eri nimiavaruuksista. Jokainen, tietyn liiketoiminnallisuuden sisältävä moduuli, on Coressa entiteetti, esimerkiksi liikelahjakomponenttia varten luotiin entiteetit GiftHistory, GiftPrice ja GiftProperty.



Kuva 14. Eficia Coren nimiavaruudet noudattelevat ohjelman eri toiminnallisuuksia ja konsepteja.

Sovellus noudattelee luvussa 4.3 esitettyä kerroksittaista mallia. Ylin kerros sisältää käyttöliittymän eli ASPX-verkkolomakkeet, jotka toimivat pääosin vain tiedon esittäjänä ja mahdollisuutena syöttää arvoja järjestelmään. Joitain kevyitä, esimerkiksi tyyppitarkistuksia suorittavia, javascript-metodeita löytyy tästä kerroksesta, mutta yleisesti ohjelman loogiset operaatiot ja toiminta on toteutettu esityskerroksen alla sijaitsevassa toiminnallisuuskerroksessa. Toiminnallisuuskerroksen alla sijaitsee pysyvän tiedon kerros, eli NHibernaten ja Spring .Net-ohjelmistokehyksen avulla luotu, tietovarastojen, niiden

yhteyksien ja niihin suoritettavien transaktioiden käsittelyn hoitava kerros. Alimpana kerroksena toimii SQL-pohjainen tietovarasto.
Aspekti-ideologian mukaisesti Coressa tarjotaan moduulien käyttöön useita yleis-palveluita. Nämä palvelut ovat erillisinä nimiavaruuksinaan ohjelmistossa, jolloin niiden toiminnallisuus on yksinkertaisempi ymmärtää eivätkä niiden sisältämät läpileikkaavat ominaisuudet pirstaloidu ja sekoitu muuhun koodiin. Tärkeimpinä palveluina voidaan pitää tietokantayhteyksien ja -transaktioiden käsittelyä, poikkeuskäsittelyä, lokien keräämistä sekä raportointiominaisuuksia. Esimerkki tietokantatransaktiosta on esitetty kuvassa 14. Kaikkia näitä palveluita pystyy käyttämään mikä tahansa entiteettiluokka, jos luodaan vaadittavat kartoitustiedostot ja tyyppimuunnoksen suorittavat luokat ja rajapinnat, joissa tyypitetään yleiset palvelut halutun luokan mukaisiksi.

```
using System.Data;
using Spring.Data.Core;
using System.Data.SqlClient;
using System.Data.Common;
using NHibernate;
using NHibernate.Expression;
using System.Collections;

namespace Eficia.HRDataAccess
{
    class GiftDataAccess : HibernateDaoSupport, IGiftDataAccess
    {
        private AdoTemplate adoTemplate;
        #region IGiftDataAccess Members

        public void SaveOrUpdate(GiftHistory gifthistory)
        {
            HibernateTemplate.SaveOrUpdate(gifthistory);
        }
        public void SaveOrUpdate(GiftPrice giftprice)
        {
            HibernateTemplate.SaveOrUpdate(giftprice);
        }
        public void SaveOrUpdate(GiftProperty giftproperty)
        {
            HibernateTemplate.SaveOrUpdate(giftproperty);
        }
```

Kuva 14. Esimerkki tietokantayhteyksien yleisen palvelun käytöstä kun sitä käyttävät liikelahjakomponentin luokat

Liikelahjakomponentin rakenne seuraa Coren linjaa. Esityskerroksessa sijaitsevat lahjojen hallintaan tarkoitetut lomakkeet ja niiden alla toiminnallisuusluokat (code-behind). Tietovaraston ja toiminnallisuuskerroksen välissä pysyvän tiedon kerroksessa ovat rajapinnat ja tietokantapalvelut. Tietokantayhteyksien käyttöä varten komponentin luokkamäärittelyt tuli saattaa pysyvän tiedon kerroksen tietoon käyttöliittymillä ja tyyppimuunnoksilla.

5.4 Sidosohjelmistot

Osa Coren palveluista on hyvin käyttäjäsidonnaisia, ja modernin sovelluksen mukaisesti eri operaatiot vaativat eri käyttäjätasoja ja -luvituksia. Tätä varten Coressa on oma moduuli, ADSynchronizationService, joka hoitaa yrityksen käyttäjätietokantaan, LDAP-protokollalla toteutettuun *Active Directoryyn,* kohdistuvat operaatiot. Osa kyseisen moduulin lähdekoodista on esitetty kuvassa 16. MicroSoftin Active Directory on käytännössä standardi ratkaisu jokaisessa yrityksessä päivittäiseen IT-infrastruktuurin hallintaan ja ylläpitoon. Se takaa laajat ominaisuudet ja tarkasti kuvatut ja vakioidut rajapinnat tiedon hakemiseen ja viemiseen, joten yhteensopivuus muiden järjestelmien kanssa on hyvää tasoa.

```
#region"ConnectionFunctions"

private DirectoryEntry openAD()
{
    //= "LDAP://" + adconnect + "," + aduser + "," + adpw;
    DirectoryEntry entry = new DirectoryEntry("LDAP://" + _addomain, _aduser + "@" + _addomain, _adpassword);
    return entry;
}
//Tällä haetaan AD:Sta yksittäinen property
private string GetProperty(SearchResult searchResult, string PropertyName)
{
    if (searchResult.Properties.Contains(PropertyName))
    {
        return searchResult.Properties[PropertyName][0].ToString();
    }
    else
    {
        return string.Empty;
    }
}
```

Kuva 16. Eficia Coren käyttäjätietojen hallintaan keskitetty moduuli hakee tietoa Active Directorysta.

Active Directoryn lisäksi Coren toiminnallisuus on suoraan sidoksissa internet-selaimeen: koko käyttöliittymä on toteutettu ASPX-teknologialla, joka pohjautuu selaimessa ajettavaan koodiin. Eficia takaa täyden yhteensopivuuden yritysympäristössä lähes vakioaseman saavuttaneelle Internet Explorerille sekä toiselle suositulle selaimelle, Mozilla Firefoxille.

6 Projektin toteutus ja testaus

6.1 Kehitysympäristö

Osana projektia koko kehitysympäristö rakennettiin normaaliin, Efician pc-koneeseen, jossa käyttöjärjestelmänä toimi MicroSoft Windows XP Professional. Koneeseen asennettiin ajoympäristöksi uusin .Net-ohjelmistokehys. Kone, jolla ohjelmointi suoritettiin, sijaitsi samassa Windows-lähiverkossa Efician kehitys- ja tietokantapalvelimien kanssa. Ohjelmistojen asennukset suoritettiin DVD-levyillä tai verkkoasemilla sijaitsevilla MSI-paketeilla, ja ohjelmistot säädettiin joko ohjattujen toimintojen tai asetustiedostojen avulla. Virtuaalikoneita ei projektissa käytetty. Asiakkaan tuotantoympäristön tietokanta- ja ohjelmistopalvelimiin yhteydet hoidettiin VPN-tunnelointiratkaisulla.

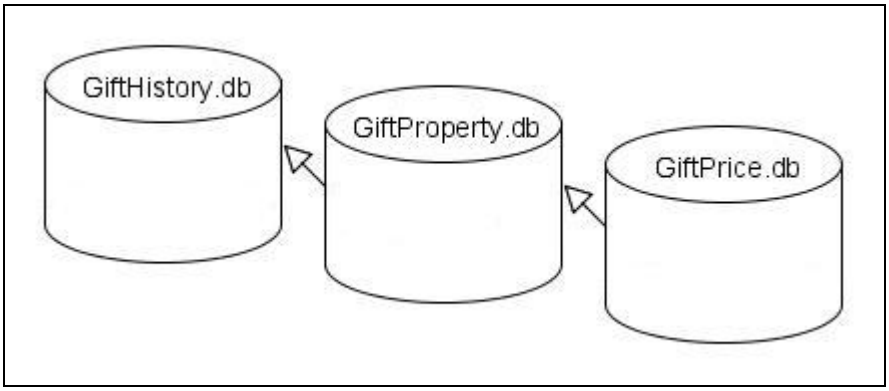
Kehitysympäristönä projektissa toimi MicroSoftin Visual Studio 2008, johon oli asennettu lisäosiksi Spring .Net-ohjelmistokehyksen moduulit, NHibernaten moduulit sekä Telerikin RAD-kontrollit. Entiteettiluokat luotiin MyGeneration koodigeneraattorilla, joka tukee NHibernate-teknologiaa. Tietokantana toimi MS SQL 2005 palvelin, jota hallinnoitiin MS:n management studio ohjelmistolla. Versiohallinta hoidettiin vapaan lähdekoodin tortoiseSVN-versionhallintasovelluksella, testaaminen internet-selaimilla ja yhteydet asiakkaan Windows-verkkoon Cisco Systemsin valmistamalla vpn-ohjelmistolla. Projektin sisäinen resursointi ja raportointi hoidettiin Eficia Coren viikkoraporttimoduulilla.

6.2 Tietokanta

Eficia Coren toiminta nojaa SQL-yhteensopivaan relaatiotietokantaan, joten tuotanto- ja testi-ympäristössä ratkaisuksi valittiin MS SQL 2005 palvelin.

Muutokset tietokantaan ja kyselyihin tehtiin aluksi testiympäristöön, josta ne siirrettiin versiopäivitysten yhteydessä VPN-tunnelin läpi asiakkaan tuotantopalvelimelle, jolloin muutoksia myös testattiin asiakasympäristössä.

Liikelahjakomponentin tietokantataulujen suunnitteluun vaikutti asiakkaan määrittelemät toiminnallisuudet. Koska asiakas halusi laajat haku- ja raportointi-ominaisuudet, päätettiin luoda (liikelahjakomponentin) päätaulu GiftHistory, joka sisältää transaktioiden historiatiedot sekä relaation tauluun GiftProperty. GiftProperty sisältää yleiset tiedot artikkelista, kuten artikkelin nimen, määrän ja toimittajan, sekä relaation tietokantaan GiftPrice, joka sisältää kyseisen artikkelin hintatiedot. Kaikki komponentin tietokantataulut ja niiden riippuvuussuhteet, on kuvattu kuvassa 17. Tähän ratkaisuun päädyttiin asiakkaan määriteltyä, että eri tuoteartikkelien lisääminen eri hinnalla tai toimittajalla varustettuna pitää onnistua. Kyselyitä tietokantaan ei tarvinnut luoda, piti vain määritellä taulut ja arvot, joita MyGeneration käytti koodin luomiseen.



Kuva 17. Liikelahjakomponentin tietokantataulut

6.3 Entiteettiluokat ja tietokantapalvelut

MyGeneration koosti entiteettiluokat luotujen tietokantataulujen perusteella suoraan NHibernaten käyttöön. Ohjelmisto säädetään opastetuilla toiminnoilla käyttämään haluttua tietokantaa ja kansioita, joihin luotavat tiedostot viedään, minkä jälkeen luodut luokat pitää vain lisätä projektiin Visual Studiossa. Luokkia olivat siis GiftHistory, GiftProperty ja GiftPrice. Luokkiin tuotetaan valmiiksi oikein tyypitetyt muuttujat, oletusmuodostin, julkiset luokan ominaisuudet, joissa tehdään merkkijonon pituuden tarkistus ja poikkeuskäsittely, jos se eroaa määritellystä, sekä NHibernaten tarvitsemat virtuaaliset aksessorit (accessors).

Koska tietokantayhteyksienhallinta on sovelluksessa poikkileikkaava ominaisuus ja aspekti, sen toiminnot ovat saatavilla alkuperäisesti vain yleiselle tietotyypille. Tämän takia kyseisestä aspektia varten piti ohjelmoida rajapinnat ja liikelahjapalvelu, giftservice, jotta poikkileikkaavat ominaisuudet käsittelisivät myös määriteltyjä luokkia. Palvelu takaa ohjelmiston käyttöön haku-, tallennus- ja poistopalvelut määritellyille tietotyypeille, ja rajapinta iGiftService ilmentää GiftService palvelua. Spring .Net-ohjelmistokehys huolehtii pysyvän tiedon kerroksessa tarvittavista koodi-injektioista transaktioita suoritettaessa tai tietokantayhteyksiä avattaessa. Kuvassa 18 on esitelty osa GiftService palvelun lähdekoodista ja kuvassa 19 havainnollistetaan miten tyyppimuunnos haluttuun tietotyyppiin tehdään lähdekoodissa.

```
public class GiftService : IGiftService
{
    private IGiftDataAccess giftDAO;

    #region IGiftService Members
    [Transaction]
    public GiftHistory FindByGiftHistoryID(long entryID)
    {
        return giftDAO.FindByGiftHistoryID(entryID);
    }
    public GiftPrice FindByGiftPriceID(long entryID)...
    public GiftProperty FindByGiftPropertyID(long entryID)...
```

Kuva 18. Esimerkki GiftService luokan hakupalvelun toteutuksesta.

```
class GiftDataAccess : HibernateDaoSupport, IGiftDataAccess
{
    private AdoTemplate adoTemplate;
    #region IGiftDataAccess Members

    public void SaveOrUpdate(GiftHistory gifthistory)
    {
        HibernateTemplate.SaveOrUpdate(gifthistory);
    }
```

Kuva 19. Tietokantayhteyksiä varten niitä hallinnoivalle ominaisuudelle tuli tehdä tyyppimuunnos haluttuihin luokkiin

6.4 Toiminnallisuudet sisältävät luokat

Kerroksittaisessa mallissa toiminnallisuuskerroksessa sijaitsee ohjelmiston looginen älykkyys ja operaatiot, niin myös Eficia Coressa. Liikelahjakomponentin toiminnallisuus on jaettu kahteen pääyksikköön: transaktioon ”lahjan antaminen” sekä artikkelien tarkasteluun ja lisäämiseen. Näitä toiminnallisuuksia varten luotiin kaksi verkkolomaketta ja näin ollen myös kaksi luokkaa, jotka sisältävät toiminnallisuuden, jota lomakkeilla esitetään.

Gift.aspx.cs ja GiveGift.aspx.cs luokissa luodaan uudet entiteettioliot sekä toteutetaan liikelahjojen selailua, poistamista, muutoksia ja lisäyksiä varten tarvittavat toiminnot ja sidotaan määritellyt verkkolomakkeiden kentät tai objektit oikeisiin entitettiluokkien muuttujiin. Näissä luokissa myös määritellään toiminnallisuus verkkolomakkeiden visuaalisille kontrolleille, kuten esimerkiksi nappien painalluksille, ja tehdään alustustoiminnot, kun verkkolomake latautuu käyttäjälle.

6.5 Käyttöliittymä

Ylimmässä kerroksessa, esityskerroksessa, sijaitsee ohjelmiston käyttöliittymä eli ASPX-verkkolomakkeet. Koko Efician verkkosivustoa koskevat määrittelyt, muotoiluasetukset ja perusasettelu on myös kapseloitu yhdeksi, tosin vain verkkosivustoa koskevaksi, aspektiksi tai sapluunaksi. Sivusto on rakennettu siten, että nämä ominaisuudet ovat kaikkien lomakkeiden käytössä ohjelmoijan niin halutessaan: lomakkeille tulee vain antaa tiedot käytettävistä moduuleista. Kuvassa 20 on havainnollistettu yleisten määrittelyjen vaikutuksia verkkolomakkeen ulkoasuun.

Kuva 20. Esimerkki liikelahjakomponentin käyttöliittymästä.

Normaalien Asp.Net-komponentin tarjoamien esitys-mallien ja -tapojen lisäksi Coren visuaalista ilmettä on lisätty Telerikin RAD-kontrolleilla. Tämä kokoelma työkaluja mahdollistaa monipuolisten ja monimutkaisten AJAX-komentojen sitomisen tiiviiksi, toiminnallisuuden mukaan jaotelluiksi kokonaisuuksiksi, kuten työkaluriveiksi tai näyttöpaneeleiksi. Näin saavutetaan näyttävämpiä

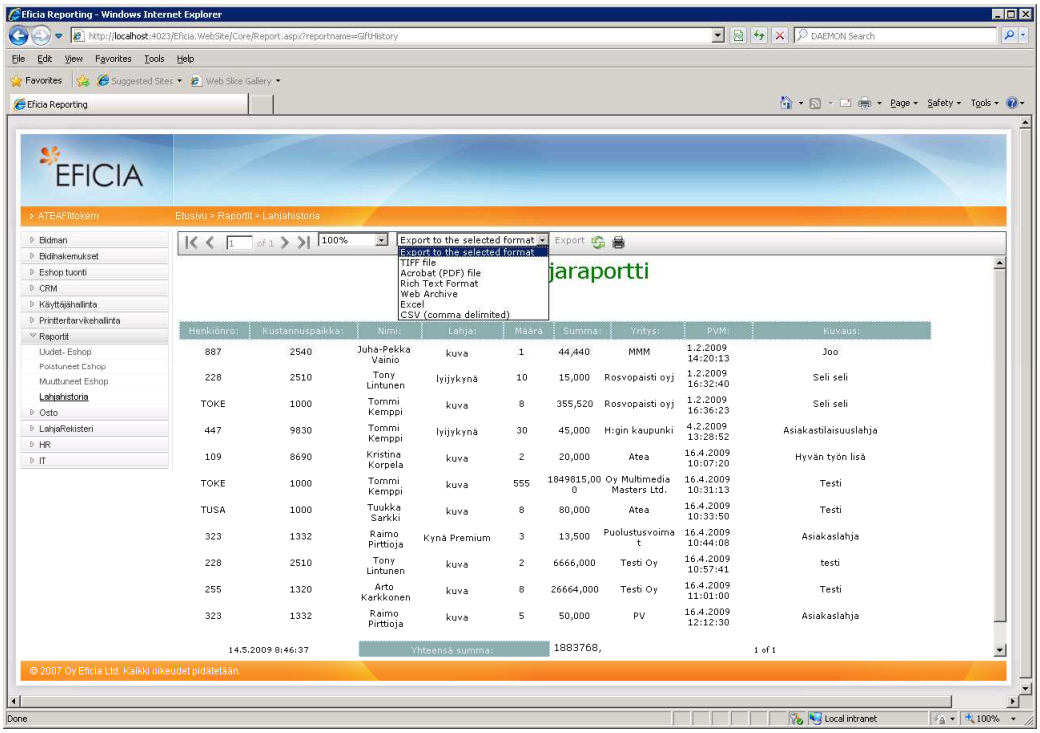
verkkolomakkeita tehokkaammin ja helpommin. Työn kannalta olennaisimmat komponentit olivat RADQueryGrid , RADAjaxPanel, RADTabStrip ja RadToolBarButton. Näillä kontrolleilla toteutettiin hakuruudukko, jonka soluihin haettiin tietoa tietokannasta, välilehdet lomakkeille sekä yksilöidyt napit eri toiminnallisuuksia varten. Kuvassa 21 on esimerkki Rad kontrollien käytöstä ohjelmakoodissa.

```
<%@ Register Assembly="RadGrid.Net2" Namespace="Telerik.WebControls" TagPrefix="radG" %>
<%@ Register Assembly="RadToolbar.Net2" Namespace="Telerik.WebControls" TagPrefix="radTlb" %>
<%@ Register Assembly="Spring.Web" Namespace="Spring.Web.UI.Controls" TagPrefix="spring" %>
<%@ Register Assembly="Eficia.Controls" Namespace="Eficia.Controls" TagPrefix="Eficia" %>
<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit" TagPrefix="ajaxToolkit" %>

<asp:Content ID="Content1" ContentPlaceHolderID="cphApplicationContent" runat="Server">
    <telerik:RadAjaxPanel ID="rapGift" runat="server" LoadingPanelID="alpLoading" Width="646px">
        <radTS:RadTabStrip ID="rtsGift" runat="server" SelectedIndex="0" MultiPageID="rmpGift"
            Width="100%" OnTabClick="rtsGift_TabClick">
            <Tabs>
                <radTS:Tab ID="Tab1" runat="server" Text="Historia">
                </radTS:Tab>
```

Kuva 21. Esimerkki RAD kontrollien käytöstä ohjelmakoodissa

Telerikin Reporting-moduulilla saatiin luotua erilaisia, näyttäviä raportteja helposti ohjattujen toimintojen avulla. Graafisesti suunniteltavat raportit, joihin pudotettiin ominaisuuksia, tietokenttiä ja asemointitietoja hiirellä, toivat merkittävän parannuksen verrattuna ASPX-verkkolomakkeen koodaamiseen käsin. Eficia Coreen koodattu tiedon taulukkolaskentasovellukseen vieminen oli verkkolomakkeiden käytössä palveluna, jolloin kyseiselle toiminnolle piti vain tuottaa oma nappi, jota painettaessa Reporting-moduulien tuottamat raportit muunnettiin taulukkolaskentamuotoon ja tallennettiin haluttuun kansioon. Kuvassa 22 on liikelahjakomponentin raportointinäkymä.



Kuva 22. Liikelahjakomponentin raportointinäkymä.

6.6 Testaus ja virheenkorjaus

Virheen- tai poikkeustenkäsittely on tyypillinen aspekti, ja sitä käsiteltiin sellaisena myös Eficia Coressa. Kaikki tietokantatransaktioiden, verkkosivujen ja toiminnallisuuksien poikkeuskäsittely oli ohjelmoitu muusta ohjelmistosta erikseen yleiseksi ominaisuudeksi. Näin ollen poikkeuskäsittelyn koodia ei tarvinnut kirjoittaa ollenkaan projektia varten, vaan valmiit mekanismit olivat automaattisesti myös uusien luokkien käytössä. Virheenkorjaus kohdistui lähinnä kartoitus- ja asennustiedostoihin, joilla ohjattiin NHibernaten ja Spring .Net-ohjelmistokehyksen toimintaa, sekä verkkolomakkeiden tiedon sitomiseen entiteettiluokkien muuttujiin.

Ohjelmiston testaaminen suoritettiin internet-selaimilla, joissa ajettiin testipalvelimella olevia verkkolomakkeita, sekä Visual Studion omalla suoritusympäristöllä. Koska käytännössä vain ylemmät kerrokset, esitys- ja

toiminnallisuuskerrokset, sisälsivät manuaalista ohjelmointia, alempien kerrosten toiminnallisuuksien ollessa lähes automatisoituja, nopeutui testaaminenkin huomattavasti. Verkkolomakkeen pystyi rakentamaan välittömästi luokka- ja rajapintamäärittelyjen jälkeen. Toiminnallisuuksien ohjelmoiminen kävi helposti kaikkien tiedon säilömiseen ja hakemiseen liittyvien operaatioiden ollessa kunnossa. Testaaminen suoritettiin ensin yksilöidysti eri ominaisuuksille ja sen jälkeen moduulitestauksena koko liikelahjakomponentille.

7 Yhteenveto

Aspektiparadigma on vielä nuori, eikä aspektiohjelmointi ohjelmointityylinä ole vielä yhtä tunnettu kuin olio-ohjelmointi, mutta paradigman mukaan toteutetussa ohjelmistossa päästään läpileikkaavilla ominaisuuksilla tehokkaampaan koodaukseen, joka tuottaa toimintavarmempaa ja yleisempää koodia. Se vastaa ominaisuuksillaan yhteen ohjelmistokehityksen suurimmista ongelmista helpottaessaan ohjelmistojen päivittämistä ja ylläpitoa. Vielä toistaiseksi paradigman käyttö on ollut rajoittunut muutamiin, kaikissa ohjelmistoissa esiintyviin ominaisuuksiin, kuten poikkeuskäsittelyyn tai tiedon hakemiseen, poistamiseen ja tallentamiseen, mutta mahdollisuuksia käytön laajentamiseen on tulevaisuudessa runsaasti.

Toistaiseksi aspektiohjelmoinnissa on myös omat haittapuolensa. Paradigma on ohjelmoijan näkökulmasta vaikeampi hahmottaa, ja toimii näin ollen modulaarisuutta vastaan. Virheentarkistuksessa tulee ottaa huomioon joukko uusia asioita, kuten kartoitustiedostot ja poikkileikkaavien ominaisuuksien toteutus kielensisäisesti. Kriitikoiden mukaan aspektiohjelmoinnin ei tulisi pyrkiä modularisoimaan mahdotonta. Myöskään vielä ei pystytä osoittamaan aspektiohjelmoinnin hyödyllisyyttä käytännössä pitkällä aikavälillä, eikä kehitystyön tehokkuuden mittaamiseksi ole standardeja mittareja, joilla saataisiin vertailukelpoista tietoa hyödyistä.

Projektin toteutuksessa tyypilliset aspektiohjelmoinnin vaikeudet ja hyödyt korostuivat; osa ohjelmistokehityksestä oli pitkälti automatisoitu, mikä nopeutti kehitystä. Vastapainoksi tälle alussa ohjelmarakennetta ja sen toimivuutta oli vaikeampi ymmärtää. Laajennettavuus Core-ohjelmistossa oli raskaan virtualisoinnin ja AOP:n myötä erinomaista luokkaa. Kyseisen ohjelmiston päivittämisessä ja laajentamisessa pitkällä aikavälillä aspektiohjelmoinnin hyödyt kertautuisivat. Kun paradigma on ymmärretty riittävän syvästi, uusien komponenttien kehittäminen muuttuu todella nopeaksi ja lähes sarjatuotantomaiseksi.

Jos nuoren paradigman alkuvaikeuksista päästään eroon ja työkaluja, jotka tukevat AOP:n kehitystä, parannetaan, kyseinen ohjelmointityyli tulee lunastamaan lupaukset, joyka se antaa tehokkaammasta koodauksesta. Se on jo todistanut tehonsa käytännössä, joten tulevaisuudessa se tullee nousemaan olio-ohjelmoinnin rinnalle yleisesti käytettynä sovellusohjelmointityylinä.

Lähteet

1 Hietanen, Päivi. C++ ja olio-ohjelmointi. Jyväskylä: Docendo Finland Oy, 2004.

2 Sorsa, Miia. Aspektiympäristöt AspectJ ja Spring AOP. Pro Gradu, Tietojenkäsittelytiede, Joensuun yliopisto, 2008.

3 Tapanainen, Timo. Aspektiohjelmointi ja hauraat liitoskohtamääritykset, Pro Gradu, Tietojenkäsittelytieteiden laitos, Helsingin yliopisto, 2007.

4 Khatchadourian, Raffi. Aspects of AOP: An exploration of the Aspect-Oriented Paradigm. Technical report, Computer Science and engineering, The Ohio State University, 2009.

5 Pollack, Mark ja muut. Spring .NET Reference Documentation 1.2. (WWW-dokumentti) http://www.springframework.net/doc-latest/reference/pdf/spring-net-reference.pdf. Luettu 15.3.2009.

6 Williams, Vanessa L. Visual studio 2005 all in on desk reference for dummies. Wiley Publishing, 2007.

7 Gehtland, Justin. NHibernate. (WWW-dokumentti) http://www.theserverside.net/tt/articles/showarticle.tss?id=NHibernate. Luettu 29.3.2009.

8 Kuaté Pierre Henri, Harris Tobin ja muut. Nhibernate in Action. Manning Publications, 2008.

9 Elrad, T., Filman, R., Bader, A. Aspect-Oriented Programming. Teoksessa Communications of The ACM Vol. 44, No. 10, 2001.

10 Laddad, Ramnivas. Aspect-Oriented programming will improve quality. Teoksessa IEEE Software, 20(6), 2003.

11 Steimann, Friedrich. The paradoxical Success of Aspect-Oriented Programming. ACM SIGPLAN notices: proceedings of the 2006 OOPSLA Conference, ACM, New York, 481-497, 2006.

12 Jacobson Ivar, Ng Pan-Wei. Aspect-Oriented Software Development with Use Cases. Addison-Wesley Professional, 2004.

13 Daigle John., Akhlaghi, Arash. Aspect Orientation as a Software Engineering Problem. Demeterpaper, Computer Science, Georgia State University, 2006.

14 Tietäväinen, Olli. Aspektipohjaisuus ohjelmistokehityksessä. Pro Gradu, Tietotekniikan laitos, Jyväskylän Yliopisto, 2006.

15 Alexander, Roger. Andrews, Anneliese A. ja Bieman, James. Towards the Systematic Testing of Aspect-Oriented Programs. Technical Report CS-04-105, Department of Computer Science, Colorado State University, 2004.

16 Suzuki J., Yamamoto Y. Extending UML with Aspects: Aspect Support in the Design Phase. Proceedings of the 3 rd AOP Workshop held in conjunction with ECOOP '99, 1999.

17 Dijkstra, Edsger W. Selected Writings of Computing, A Personal perspective. Springer Verlag, 1982.

18 Bradley, Jeremy ja Alexander, Roger. An examination of Aspect-Oriented programming in industry. Technical Report CS-03-108, Department of Computer Science, Colorado State University, 2003.

19 Cazzola, Walter., Jézéquel, Jean-Marc. ja Rashid, Awais. Semantic Join Point Models: Motivations, Notions and Requirements. (WWW-dokumentti) http://www.irisa.fr/triskell/publis/2006/Cazzola06a.pdf. Luettu 24.4.2009.

20 Sato, Y. A Study of Dynamic Weaving for Aspect-Oriented Programming. (WWW-dokumentti) http://www.csg.is.titech.ac.jp/paper/yoshiki-phd2006.pdf, 2005. Luettu 10.2.2009.

21 Harman, Mark ja Black, Sue. Aspect oriented software development towards philosophical basis, Technical Report TR-06-01, department of Computer Science, King’s College London, 2006.

22 Kemppi, Tommi. Ohjelmistokehittäjä, Eficia Oy, Efician asiakkaan edustaja., Helsinki. projektipalaveri 20.8.2008.

23 Saikkonen, Timo. Ohjelmistoarkkitehti, Eficia Oy, Helsinki. Eficia Core esittely luento 8.8.2008.