TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Informaatika aluste õppetool

Aspektorienteeritud programmeerimise kasutamine AspectJ näitel

Bakalaurusetöö

Üliõpilane: Jaroslav Artjuh

Üliõpilaskood: 112526IAPB

Juhendaja: lektor Ants Torim

Tallinn
2014

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

---------- ----------

(*kuupäev*) (*allkiri*)

Annotatsioon

Minu töö eesmärk on uurida aspektorienteeritud programmeerimise meetodit. Mille poolest aspektide kasutamine erineb tavalise Objektorienteeritud programmerimisest ja kuidas aspekte on võimalik kasutada. Milliseid eeliseid annab aspektorienteeritud metoodika kasutamine ja kui palju aega võib kuuluda uue meetodi valdusse võtmisele.

Töös on käsitletud aspektide kasutamise alasid ning olukordi millal on tarvis aspektidega tegeleda. Uurin mida nõuab aspektide kasutamine ja millised on kasutamise tagajärjed. Töös on näidatud millises olukorras on praegu aspektorienteeritud programmeerimine ja miks üldse peab oma tähelepanu aspektidele pöörama.

Oma töös teen ma selgeks aspektide integreerimise põhimõtteid ja toon praktilisi näiteid aspektorienteeritud programmeerimise kasutamisest ning teen seda AspektJ abil. Seon teooria ja praktikat kokku, et paremini seda teemat selgitada.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 33 leheküljel, 5 peatükki, 23 joonist, 2 tabelit.

Abstract

The main aim of this research is to investigate into aspect-oriented programming; the difference between using aspect and object-oriented programming; the ways and advantages of using aspect-oriented programming; time required for studying aspect-oriented programming.

The research covers areas of using aspects, the situations when they should be used; and the consequences of using them. This research also shows the current state of aspect oriented programming and why it is worth to be studied.

In this research I will explain basic principles of aspects integration and will give examples of practical use of aspect-oriented programming using syntaxes in AspectJ. I will combine both theory and practical aspects to cover this topic.

The thesis is in Estonian and contains 33 pages of text, 5 chapters, 23 figures, 2 tables.

Lühendite ja mõistete sõnastik

Objektorienteeritud programmeerimine (OOP)	***Object-oriented programming*** See on programmeerimise paradigma, mis kasutab "objekte" – andmestruktuure, mis koosnevad andmeväljadest ning meetoditest [1].
Ristlõikuv funktsionaalsus	***Cross-cutting concerns*** See on funktsionaalsus, mis läbib erinevaid programmi klasside piire ja ei täida põhifunktsionaalsust [2].
Aspektorienteeritud programmeerimine (AOP)	***Aspect-oriented programming*** Aspektorienteeritud programmeerimine (AOP) on programmeerimis paradigma, mille põhi idee on funktsionaalsuse jagamine, mis aitab programmi paremini mooduliteks jaotada [3].
Aspekt	***Aspect*** See on programmi osa, mis ei ole seotud põhifunktsionaalsusega, tema ülesanne on modulaarsuse tagamine ja ristlõikuva funktsionaalsuse inkapsuleerimine ning põhifunktsionaalsuse omavaheliste seoste tekkitamine [4].
Joinpoint	***Joinpoint*** See on sündmus, jooksvas programmis, mis annab võimaluse Advice koodi käivitada [3].
Pointcut	***Pointcut*** See on koht programmi koodi, mis annab teada Advice-le, kus ja millal ta peab oma funktsionaalsust realiseerima [5].
Advice	***Advice*** See on aspekti funktsionaaluse kood, mis täidab oma ülesannet õigel ajal ja kohal, mis nätab talle pointcat [3].
AspectJ	***AspectJ*** See on AOP realisatsioni laiendus Java programmeerimis keele jaoks [6].

Jooniste nimekiri

Tabelite nimekiri

Sisukord

Sissejuhatus

Programmeerimise piirkond areneb ja muutub iga päev. Ilmuvad uued keeled, tehnoloogiad ja programmeerimise meetodid. Kõik arendajad püüavad teha tarkvara arendamise protsessi efektiivsemalt ja universaalsemalt ning tagada, et tarkvara oleks stabiilsem ja kvaliteetsem. Aspektorienteeritud programmeerimine on see metoodika, mis tulevikus võib arendajate põhiprintsiipe muuta.

Töö annab ülevaade aspektide kasutamisest. Töös uuritakse milleks on üldse vaja seda metoodikat kasutada ja kuidas see aitab erinevaid ülesandeid lahendada. See töö oleks kasulik arendajatele, kes tegelevad erinevate süsteemide arendamisega ja nende projekteerimisega ning programmeerijatele, kes tahavad oma koodi õieti ja loogiliselt struktureerida.

Praegu on selline olukord, et paljud arendajad ei kasuta seda meetodit, sest see ei ole veel nii hästi levinud ja populaarseks muutunud. Oma töös ma tahan anda informatsiooni ja praktilisi näiteid aspektide kohta, uurida aspektide kasutamise alad ja nende omadusi. Tahan anda arendajatele ülevaadet aspektidest ja pöörata nende tähelepanu aspektide kasutamisele.

Alguses kirjutan ma objektorienteeritud programmeerimise (OOP) kasutamisest ja põhimõtetest ning edasi püstitan probleemi, mis on seotud OOP ja ristlõikuva funktsionaalsusega. Edasi ma kirjutan aspektorienteeritud programmeerimisest (AOP) ja sellest, kuidas see metoodika aitab seda probleemi lahendada. Järgmiseks tulevad AOP põhimõtted ja teooria. Pärast ma tutvustan AspectJ programmeerimiskeelt Java keele jaoks ja näitan väga lihtsustatud programmi, mida demonstreerib AOP funktsionaalsust ja AspectJ süntaksi. Järgmiseks tulevad AspectJ spetsiifilised omadused ja laiendatud süntaksi näited, mis aitavad teooriat lihtsamini aru saada. Kirjutan ka kuidas ma AspectJ kasutama hakkasin ja mida ma sellega tegin. Kirjutan millist funktsionaalust ma selle abil realiseerisin ja kuidas see reaalses elus aitab.

1. Objektorienteeritud programmeerimine

Objektorienteeritud programmeerimine (OOP) on mõistlik programmi koodi struktureerimine, mis teeb kõik kirjutatud programmi lihtsamaks ja arusaadavamaks. OOP tegeleb objektidega ning see võimaldab kõik süsteemi osad jagada objektideks, et iga objekt lahendas oma ülesannet süsteemis [7]. See tähendab, et [7]:

- kõik programmi osad on objektid
- programm ise on objektide grupp, mis suhtlevad omavahel sõnumite abil ja annavad käske ehk tööülesandeid teine teisele
- iga objektil on oma mälu, kus ta hoiab teisi objekte
- igal objektil on oma tüüp
- kõik samatüüpilised objektid võivad saada ühesuguseid sõnumeid.

OOP kasutusel võivad olla selliseid võtted nagu andmete abstraktsioon, kapseldamine, modulaarsus, polümorfism ning pärimine [1]. See kõik aitab mõistlikult programmi osadeks jagada ja tagada kõrget modulaarsust. Omakorda moodulite muutmine, lisamine või ümberkirjutamine annab programmile uut funktsionaalsust.

OOP kasutatakse tarkvara arendamise protsessis väga tihti, sest see toob kaasa palju kasu, edu ja efektiivsust [5]. Arendajad püüavad struktureerida oma koodi nii, et seda saaks väga lihtsalt ja tihti korduvalt kasutada. Modulaarsus annab neile seda võimalust ja tõstab kirjutatud koodi taaskasutamist Aga on ka olukordi, millal seda teha on väga raske.

1.1 Ristlõikuv funktsionaalsus ja OOP

OOP kasutades saab väga hästi modulariseerida programmi põhifunktsionaalsust (core concerns), näiteks panga süsteemis on see: kliendihaldus, kontode haldamine ja laenude organiseerimine [8]. Teine niinimetatud ristlõikuv funktsionaalsus (cross-cutting concerns) on näiteks: logimine, ressursside ühiskasutus, caching, tööjõukuse jälgimine, kokkulangevuse kontroll ja transaktioonide haldus, pole võimalik modulariseedida kasutades OOP [8]. See ristlõikuv funktsionaalsus toob kaasa koodi üleliigsust ja samasuguse koodi korduvust erinevates moodulites.

Tavaliselt OOP kasutades tehakse iga ristlõikuva funktsionaalsuse jaoks eraldi moodulit ja kasutatakse seda, kui see funktsionaalsus on tarvis. Seda võib näha joonisel number 1.

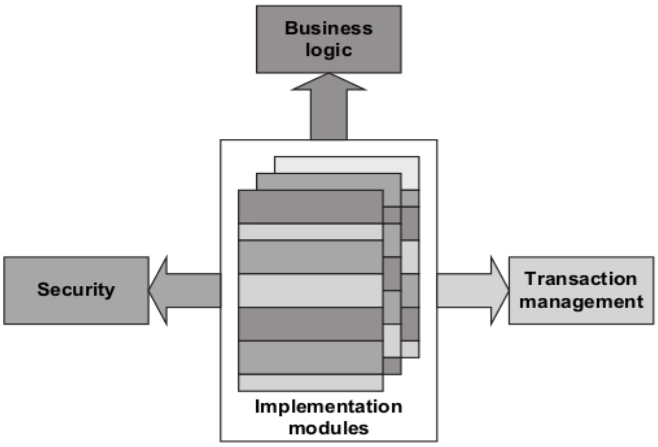


Figure 1.1 Viewing a system as a composition of multiple concerns. Each implementation module deals with some element from each of the concerns the system needs to address.

Joonis 1: Ristlõikuva funktsionaalsuse realisatsioon kasutades OOP [8]

Skeemil on näha, et need moodulid kutsuvad iga kord teisi mooduleid ja tekkib koodi üleliigsus ning mooduli struktuur muutub ebaselgemaks ja keerulisemaks. Järgmisel joonisel number 2 on näidatud kuidas pangandussüsteem rakendab turvalisuse kasutades tavapärast tehnikat. Isegi kui kasutada läbimõeldud turvalisuse moodulit, kus on abstraktne API, eraldi tehtud kontode moodul, ATM moodul ja andmebaasi moodul. Ikka on vaja kirjutada koodi, kus pöördutakse turvalisuse mooduli poole [8].

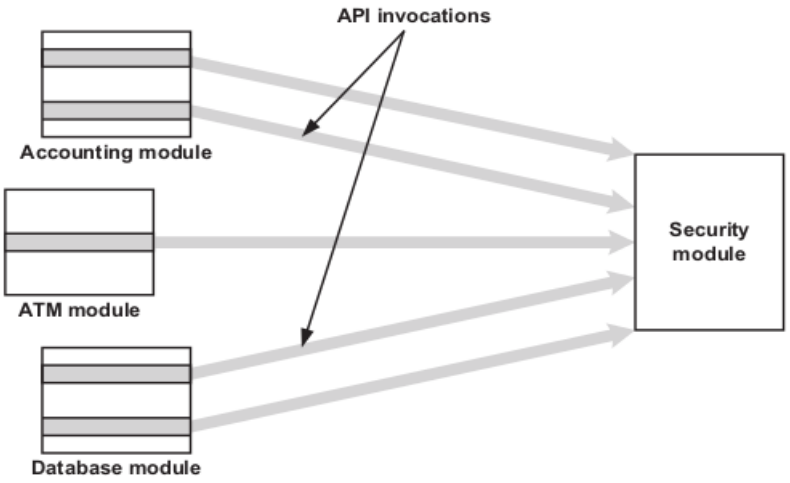


Figure 1.4 Implementation of a security concern using conventional techniques. The security module provides the API for authentication and authorization. But the client modules—accounting, ATM, and database—must embed the code to invoke the API to, say, check permission.

Joonis 2: Pangandussüsteemi turvalisuse mooduli realiseerimine [8]

2. Aspektorienteeritud programmeerimine

Aspektorienteeritud programmeerimine (AOP) on programmeerimis paradigma, mille põhi idee on funktsionaalsuse jagamine, mis aitab programmi paremini mooduliteks jaotada [3]. AOP metoodika on mõeldud välja OOP laiendamiseks ja edasi arendamiseks, aga mitte mingil juhul ei saa AOP asendada tavalist OOP metoodikat. AOP aitab ristlõikuva funktsionaalsuse probleeme lahendada ja neid lihtsamaks teha.

AOP eraldab põhifunktsionaalsust ja ristlõikuva funktsionaalsust. Mis teeb koodi loetavamaks ja annab arendajatele võimaluse põhifunktsionaalsusega efektiivsem tegeleda. AOP ise hoolitseb selle ees, et kõik kirjutatud aspektid õigel ajal tööle panna ja täita õiges kohas nõutud funktsionaalsust. Seda võib näha joonisel number 3, mis on võrreldes joonisega number 2, kasutab juba AOP tehnoloogiat.

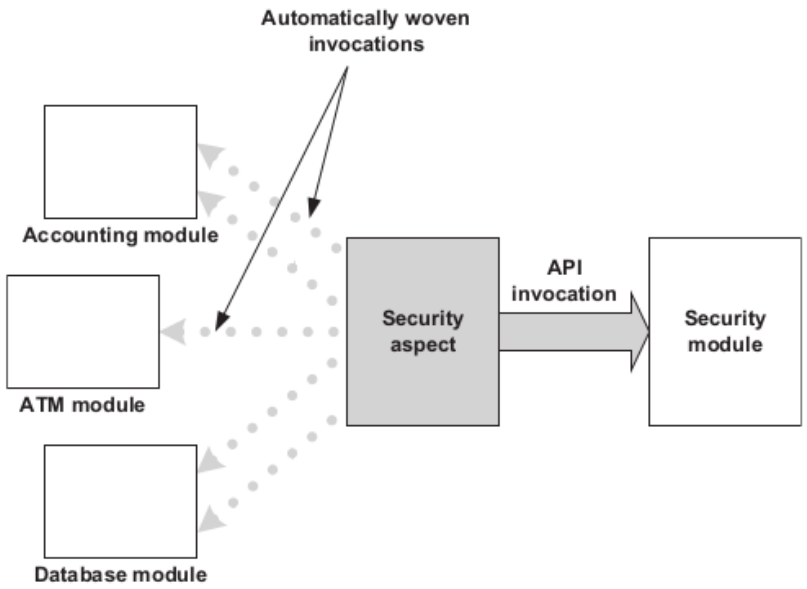


Figure 1.5 Implementing a security concern using AOP techniques: The security aspect defines the interception points needing security and invokes the security API upon the execution of those points. The client modules no longer contain any security-related code.

Joonis 3: Pangandussüsteemi turvalisuse mooduli realiseerimine kasutades AOP [8]

2.1 AOP põhimõtted

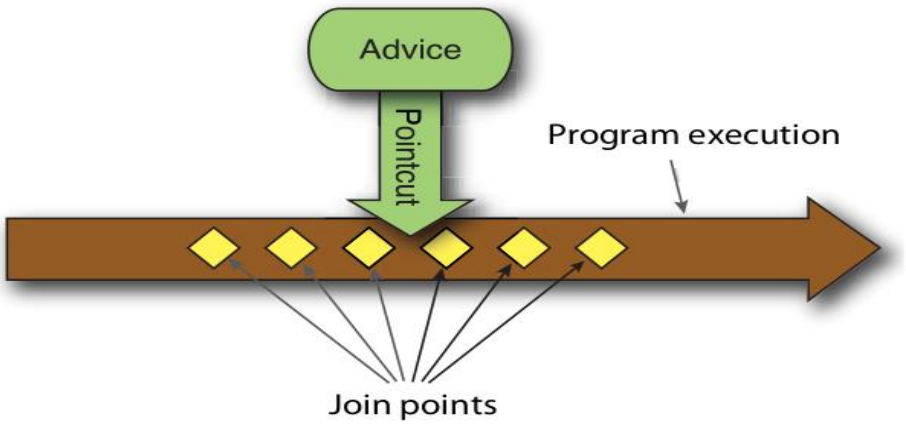
Aspect – programmi osa, mis ei ole seotud põhifunktsionaalsusega, tema ülesanne on modulaarsuse tagamine ja ristlõikuva funktsionaalsuse inkapsuleerimine ning põhifunktsionaalsuse omavaheliste seoste tekkimine [4]. Aspeti võib nimetada ka klassiks, aga tema ise ei täida mingit funktsionaalsus, ta on mõeldud tavaliste klasside abivahediks ja nende omavahelise seadistamiseks. Aspektiga on seotud sellised põhimõtted nagu pointcut, joinpoint ja advice.

Joinpoint – sündmus, jooksvas programmis, mis annab võimaluse Advice koodi käivitada. Need sündmused võivad olla näiteks [5]:

- meetodi väljakutse
- meetodi täitmine (execution)
- vigade (exception) käsitlemine
- muutuja kinni püüdmine
- klassi või objekti initsialiseerimine

Pointcut – koht programmi koodi, mis annab teada Advice-le, kus ja millal ta peab oma funktsionaalsust realiseerima [5]. See koht asub tavaliselt joinpoint-ide vahel.

Advice – aspekti funktsionaalsuse kood, mis täidab oma ülesannet õigel ajal ja kohal, mis nätab talle pointcat [3]. See kood võib käivituda enne, pärast või üldse joinpointi asemel.



Joonis 4: AOP põhimõtete graafiline esitlus [9]

3. AspectJ

AspectJ on AOP realisatsioni laiendus Java programmeerimis keele jaoks [6]. Seda arendas Xerox PARC (Palo Alto Research Center Incorporated) ja pärast andis seda projekti eclipse.org ühiskonnale ning praegu on see Eclipse Foundation avatud lähtekoodiga projekt, mida võib uurida [6].

AspectJ on originaalne ja kõige parem AOP implementatsioon („AspectJ is the original and still the best implementation of AOP“) [8]. See tähendab seda, et Aspectj võimaldab kasutada kõik AOP funktsionaalsust natiivse meetodiga. Kasutades AspectJ võib realiseerida kõike olemasolevaid aspektide funktsionaalsust ja arendada neid, lähtekoodi täiendades. Need võimalused annavad palju kasu arendajatele, kes tahavad AOP-ga tegeleda ja seda edasi arendada.

AspectJ ei ole aga ainuke AOP realisatsioon, on ka teised näiteks Spring AOP, AspectWerkz, AspectC++, PHPaspect, AspectLua ja Lightweight Python AOP. See annab meile teada, et AOP paradigma areneb, leiab oma huvilisi ja tulevikus võib populaarseks muutuda.

3.1 Lihtne näide

Selles punktis ma tahan näidata AspectJ süntaksi ja kirjutada lihtsa aspekti. Alguses ma kirjutasin väga lihtsa programmi, mis väljastab konsooli peale ühte teksti tüüpi rida - „Teen midagi“.

```
 3  public class Programm {
 4
 5      public static void main(String[] args) {
 6
 7          System.out.println("Teen midagi...");
 8
 9      }
10  }
```

Joonis 5: Lihtne programm Java keeles

Pärast ma kirjutasin selle programmi juurde logimise funktsionaalsust, mis väljastab konsoolile, enne programmi täitmist, et „Programm loodi“ ja pärast täitmist, et „Programm lõpetas oma tegevust“. Reas number 3 toimub **aspekti** deklareerimine. Pärast tuleb **pointcut,** mida ma nimetasin „logging“ ja **joinpoint** on programmi main() funktsioon. Järgmiseks ma kirjutasin 2 **advice** plokki. Esimene kutsutakse enne (before) main() funktsiooni täitmist ja teine pärast (after) seda täitmist.

```
 3 public aspect Logs {
 4
 5     pointcut loging():
 6         execution(public void main(..));
 7
 8     before(): loging() {
 9         System.out.println("Programm loodi");
10     }
11
12     after(): loging() {
13         System.out.println("Programm lõpetas oma tegevust");
14     }
15 }
```

Joonis 6: Lihtne logimine kasutades AspectJ

```
Programm loodi
Teen midagi...
Programm lõpetas oma tegevust
```

Joonis 7: Lihtne programmi konsooli tulemus

Programmi kirjutamiseks ma kasutasin Eclipse IDE ja AJDT (AspectJ Development Tools). Piltidel number 5 ja 6 võib näha, et koodi ridade kõrval tekkisid nooled, mis aitavad aspekti tegevust paremini jälgida ja aru saada.

Kui selle lihtsa programmi kirjutamiseks ma ei hakka aspekti kasutama, siis mul on vaja logimise jaoks uue klassi defineerida ja kirjutada tema funktsionaalsust. Käivitava programmi koodi pean ma ka juurde kirjutama, et logimis operatsiooni realiseerida. Alguses on vaja seda klassi tekitada, siis alles kutsuda tema meetodeid programmi alguses ja lõpus. Võime näha, et tuleb välja midagi sellist:

```
3  public class Programm {
4
5      public static void main(String[] args) {
6
7          LogService logService = new LogService();
8
9          logService.programStart();
10         System.out.println("Teen midagi...");
11         logService.endOfProgram();
12
13     }
14 }
```

Joonis 8: Lihtne programm Java keeles ilma aspektideta

```
3  public class LogService {
4
5      public void programStart() {
6          System.out.println("Programm loodi");
7      }
8
9      public void endOfProgram() {
10         System.out.println("Programm lõpetas oma tegevust");
11     }
12
13 }
```

Joonis 9: Lihtne programmi logimise klass

Siin on lihte märgata, et programmi põhikoodi ilmus ristlõikuv funktsionaalusus, mis pole programmi põhitegevusega seotud.

3.2 Joinpoint tüübid AspectJ süntaksi näitel

Siin on mõned joinpoint näited, mis on võimalik programmeerimisel kasutada.

- Mingi meetodi täitmine: **execution(void Point.setX(int)).** Siin käivitub aspekt alles siis, kui programm on jõudnud meetodi sisse ja parameetrite väärtused on edastatud.
- Mingi meetodi väljakutsumine: **call(void Point.setX(int)).** Sellel juhul programm ei ole jõudnud veel meetodit täita ja parameetrite väärtused ei veel määratud.
- Vigade käsitlemine: **handler(NumberFormatException).** Siin reageerib mingi aspekt ainult siis, kui programmi käigus tekkib mingi viga, siin näiteks NumberFormatException.

- Käivitav kood on mingi klassi oma: **within(Point).** Saab näiteks kontrollida kõik neid funktsioone, mis on Point klassis kirjutatud.
- Mingi klass käivatas mingit meetodit: **withincode(public MyClass.printPoint()).** Aspekt hakkab tegutsema ainult siis, kui meetod printPoint() oli väljakutsutud klassist MyClass.

3.3 Joinpoint ja metasümbolite kasutamine

AspectJ süntaksiga on väga lihtne märkida õige koht programmis ja deklareerida pointcut-e, sest selle jaoks on tehtud väga paindlik ja mugav deklareerimise viis. Saab kirtutada nii täpne objekti omadus ja nimi kui ka märkida metasümbolite abil relatiivsetl seda objekti või objekti gruppi, kus on vaja kasutada advice koodi. Joinpoint kohad võib tekitada erinevatel meetoditel ja programmi kohtadel, vajalik on ainult neid kohad õieti märkida.

Siin on mõned süntaksi näted koos selgitustega [8]:

- **execution(* *(..))** Aspeti hakatakse kasutama kõikide funktsioonide täitmisel. Esimene sümbol ***** tähendab, et pole oluline, mis tüüpi on funktsioon, kas public, privete, protected või mis objekti tüüpi see tagastab, näiteks int, String, boolean või see on üldse void funktsioon. Teine ***** määrab täitava funktsiooni nimi, sellel näitel on kõik funktsioonid märgitud. Sulgudes olevad täppid (**..**) tähendavad, et parameetrid ka pole olulised.
- **execution(* set(..))** See joinpoint näitab kõik programmi **set** meetodite peale.
- **execution(* setX(double))** Kõik **setX** meetodit, mis parametriteks tahavad **double** tüüpi väljakutse ajal saada.
- **execution(* Point.setY(int))** Advice koodi saab täita ainult siis, kui **Point** klass kutsub välja meetodi **setY** ja tema parameetriks on **int**.
- **execution(* .new(double, int))** Siin **.new** tähendab seda, et tegemist on konstruktoriga, mis nõuab double ja int parameetreid.
- **execution(int get*())** Sellisel juhul joinpoint-iks on kõik programmi meetodid, mis algavad **get** nimega (näiteks getY()), neil pole parameetreid ja tagastavad **int** tüüpi väärtust.

- **execution((@Transactional) * *(..))** Kõik meetodid, mis on märgitud **@Transactional** anatatsiooniga on aspekti jaoks kättesaadavad.

3.4 Pointcut ja joinpoint omavahel ühendamine

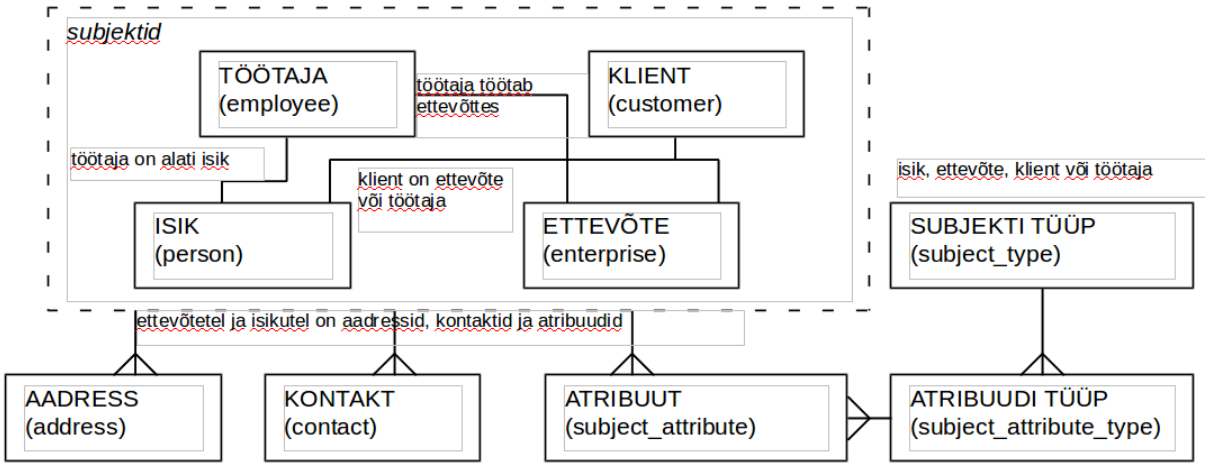
Õige pointcut-i saamiseks on võimalik ka joinpoint-e ja pointcut-e omavahel ühendada. AspectJ süntaksi abil on võimalik grupeerida ja liita joinpoint-id omavahel. Selle jaoks peab kasutama tavalist Boole'i algebra operatsioone: && (and), || (or), xor, ! (not) ja teised.

Näiteks:

- **target(Point) && call(double *(..))**
- **call(public *(..)) && (within(Point) || within(Triangle))**
- **within(Point) && execution(* .new(double, int))**
- **!this(Triangle) && call(int get*())**

4. AspectJ praktiline kasutamine subjektide haldamise süsteemis

Selles osas ma kirjutan oma kogemustest AspectJ kasutamisel ja toon näited minu projekti tööst, mida ma kirjutasin ainel **IDU0200** - „Veebipõhiste rakenduste arhitektuur, disain ja tehnoloogia“. Minu teemaks on „Subjektid“ ja selles töös mul oli vaja realiseerida subjektide haldamist: nende andmete, atribuutide, aadressite, ja kontaktide sisestamine ning muutmine [10]. Subjektide mõiste hulga kuuluvad isik, töötaja, klient ja ettevõte. Nad on seotud omavahel erinevate suhetega ja omavad kontakte, aadresse ja atribuute, mis on abstraktsed ja kuuluvad iga subjekti kohta.

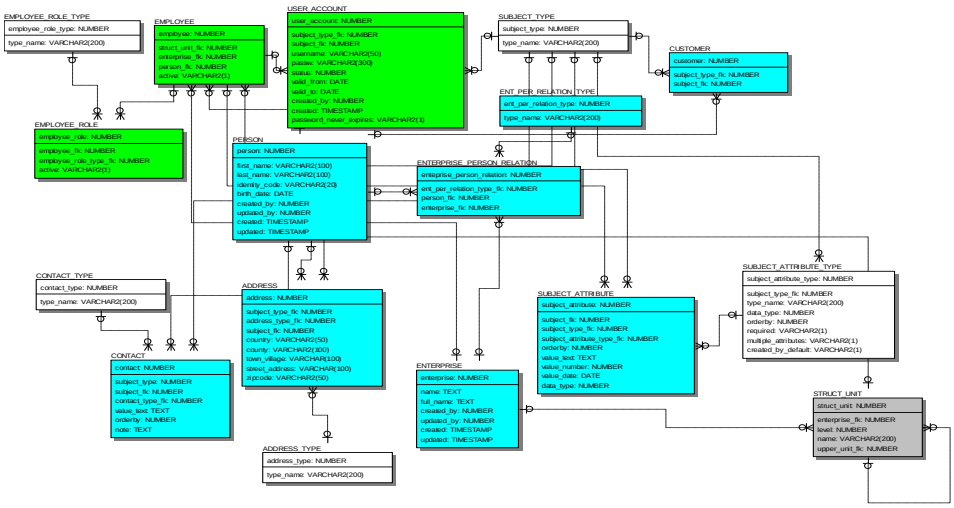


Joonis 10: Subjektide mõisteskeem [11]

See aine annab võimaluse ise valida erinevaid tehnoloogiad ja meetodeid selle ülesanne realiseerimiseks, sest aine kursusel tutvustatakse erinevaid programmeerimiskeeli, raamistikke ja teeke. Õpetatakse kuidas on mõistlik oma rakendust struktureerida ja teha õiget arhitektuuri. Võimalik on ka valida mingi oma tehnikat, programmeerimiskeelt või raamistiku (peale PHP-d) ja teha seda ülesannet.

Mina otsustasin valida Java Platform Enterprise Edition (**Java EE**), Java Persistence API (**JPA**) + **Hibernate**, JavaServer Pages (**JSP**), **PostgreSQL** ja **AspectJ**. Minu lähtekoodi kirjutatud subjektide süsteemi jaoks ma panin avalikkuse Github-i repositooriumisse, et seda oleks mugavam uurida https://github.com/jarik888/Subject-system-with-AspectJ [12].

Projekti jaoks andmebaasi skeem ja postgreSQL-i create ning insert laused olid juba ette antud. Skeemi peal võib näha neid värvutud tabeleid, kus läbi rakenduse peaks olema võimalik andmeid lisada, muuta, kustutada [11].



Joonis 11: Subjektide andmebaasi skeem [11]

4.1 Logimise aspekt

Alguses ma realiseerisin logimise funktsionaalsust, kirjutades logimise aspekti. See aspekt realiseerib kõikide rakenduses tekitavate vigade logimist. Aspekt on tehtud nii, et see saab iga vea tekkimise korral püüda seda ja kirjutada log faili. Logimiseks ma kasutasin **log4j** teeki, mis kirjutab eraldi logid Apache serverisse, mis pole Tomcat serveriga seotud, kus asub kirjutatud rakendus.

```
3  import org.apache.log4j.Logger;
4
5  public aspect Logs {
6
7      static Logger logger = Logger.getLogger(Logs.class);
8
9      before(Exception e): handler(Exception+) && args(e) {
10         logger.error(thisJoinPoint.getSourceLocation().toString() + " Error: " + e.toString());
11     }
12 }
```

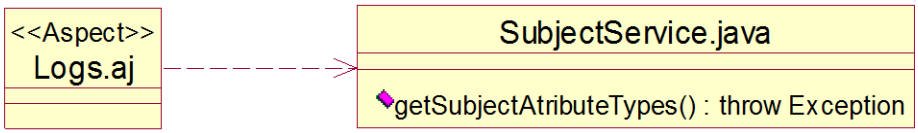
Joonis 12: Logimise aspekt [12]

Reas number 3 on kokku kompaktselt kirjutatud pointcut ja joinpoint. Selles reas enne (before) viga tekkimist (Exception e), tehakse seda viga töötlemine (handler), **Exception+** (pluss märgiga) tähendab seda, et töödeldakse kõik Exception tüüpi ja tema alamklassi vigu. Reas number 4 toimub selle vea log faili kirjutamine: **thisJoinPoint** võimaldab saada täpse joinpoint-i asukoha, **getSourceLocation()** väljastab klassi nimi ja koodis oleva read numbrit, **e.toString()** annab teada mis tüübi veaga on tegemist. Log failist võetud rida: *„SubjectService.java:79 Error: org.apache.jasper.JasperException: javax.servlet.ServletException: java.lang.NullPointerException - 20 May 2014 13:54:46“*

```
70      public List<Subject_attribute_type> getSubjectAtributeTypes(long subject_type_fk) {
71          List<Subject_attribute_type> sat = null;
72          try {
73              GenericDAO personDAO = new GenericDAO(manager, Subject_attribute_type.class);
74              String query = "SELECT sat FROM Subject_attribute_type sat "
75                          + "WHERE sat.subject_type_fk = :subject_type_fk";
76              String param_name = "subject_type_fk";
77              Long param_value = subject_type_fk;
78              sat = (List<Subject_attribute_type>) personDAO.findByQuery(query, param_name, param_value);
79          } catch (Exception e) {}
80          return sat;
81      }
```

Joonis 13: SubjectService klassi meetod [12]

Võib näha, et nüüd ei pea ma enam iga kord defineerida logimise klassi ja igas catch plokkis mingi viga log faili kirjutada. Ma võin lihtsalt seda plokki tühjaks jätta või ainult mingi põhifunktsionaalsus juurde kirjutada. Logimise eest hoolitseb juba üks aspket kõikide catch plokkide jäoks.



Joonis 14: Logimise aspekt klassi diagrammi kujul

4.2 Parameetrite ja atribuutide tagastatavate väärtuste kontrollimis aspekt

See aspekt kirjutab ümber ehk teisendab standartse Java EE funktsioone, mis on **ServletRequest** klassi omad: **getAttribute(String arg0)** ja **getParameter(String arg0)**. Need funksioonid tagastavad request massiivist erivevaid parameetreid ja atribuute tekstilise nime järgi. Juhul kui otsitav atribuut või parameeter on olemas, siis tagastatakse seda Object-i või String tüüpi väärtust, aga kui seda ei leita, siis tagastatakse null. Minu aspekt kontrollib

seda, et tagastamise objektiks oleks mitte null väärtus. See on tehtud selleks, et vältida NullPointerException-e ja vähendada null väärtuste kontrolli kontrollerite koodis.

```
3  import javax.servlet.ServletRequest;
4
5  public aspect GetParamAndAttr {
6
7      pointcut getParamInControllers(ServletRequest req, String s):
8          target(req) && args(s) && within(frontend.controllers.*) &&
9          call(String javax.servlet.ServletRequest.getParameter(String));
10
11     String around(ServletRequest req, String s): getParamInControllers(req, s) {
12         String result = req.getParameter(s);
13         if (result == null) {
14             return "";
15         }
16         return result;
17     }
18
19     //all get() calls in this aspect
20     pointcut getCallsInAspect(): this(GetParamAndAttr) && call(* get*(String));
21
22     pointcut getAttrEverywhere(ServletRequest req, String s):
23         target(req) && args(s) && call(* *.getAttribute(..));
24
25     Object around(ServletRequest req, String s): getAttrEverywhere(req, s) && !getCallsInAspect() {
26         Object result = req.getAttribute(s); // avoid loop -> !getCallInAspect()
27         if (result == null) {
28             return "";
29         }
30         return result;
31     }
32 }
```

Joonis 15: Parameetrite ja atribuutide aspekt [12]

Selles aspektis ma teen kolm pointcut-i. Esimene on ridadel number 7 kuni 9. Ta näitab kõikidele **getParameter(String arg0)** meetoditele, mida kutsuvad kontrollerid, mis asuvad **frontend.controllers** pakettis. Teine pointcut on reas number 20, ta on seotud kõikide meetoditega, mille nimed algavad **get** tähtedest ja nende paramtriks on **String** väärtus ning need meetodid kuuluvad **GetParamAndAttr.aj** aspekti juurde. See pointcat on tehtud selleks, et vältida lõpmatu tsükli tekkimist, kui ma hakkan **getAttribute(String arg0)** funktsiooni (rida number 26) selles aspektis kutsuma. Kolmas pointcut on ridadel number 22 ja 23. See näitab kõikidele **getAttribute(String arg0)** meetoditele, mida kutsutakse iga klassis sees.

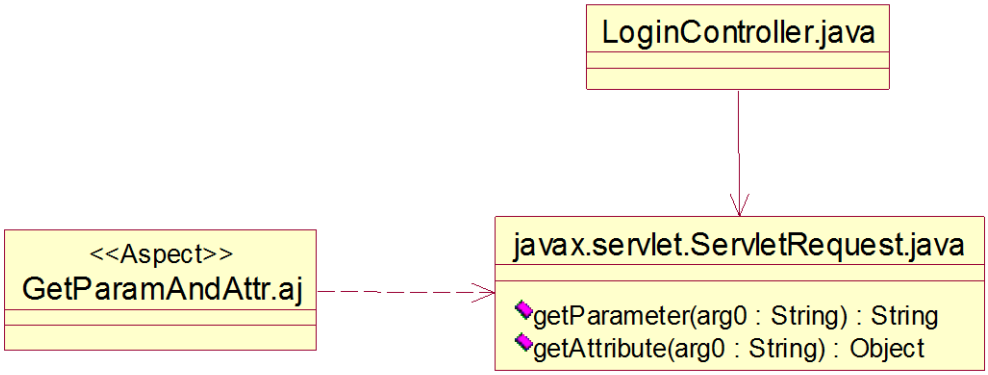
Advice koodi ridades võib märgata, et toimud **getAttribute(String arg0)** ja **getParameter(String arg0)** meetodite täitmine ja kui tagastatav väärtus pole null, siis tagastatakse seda väärtus, aga kui ilmub null, siis teisendatakse seda tühja stringi väärtusega

(„“). See võimaldab väärtuse saamise ajal kontrolleri koodi vähendada, sest ei pea iga kord null väärtust kontrollida.

```
21    private void navigate(HttpServletRequest req, HttpServletResponse res) {
22
23        String page = "/login.jsp";
24  //    String action = req.getParameter("action") == null ? "" : req.getParameter("action");
25        String action = req.getParameter("action");
26
27        if (action.equals("login")) {
28            String username = req.getParameter("username");
29            Long user_id = loginS.login(username, req.getParameter("password"));
30            if (user_id != null) {
31                mainS.session(req); //createNewSession
32                mainS.getSesson().set("username", username);
33                mainS.getSesson().set("user_id", String.valueOf(user_id));
34                page = "/main.jsp";
35            } else {
36                req.setAttribute("LoginError", "Login or password error!");
37                req.setAttribute("username", username);
38            }
39        } else if (action.equals("logout")) {
40            mainS.getSesson().invalidateCurrent();
41        }
42        new NavigationController().showPage(req, res, page);
43    }
```

Joonis 16: Parameetrite ja atribuutide aspekti kasutamise näide [12]

Siin võib näha, et aspekti täidetakse ridadel number 25, 28 ja 29. Reas number 24 on kommenteeritud kood, mida ma pean kindlasti kasutama, kui ma ei kasuta eeltoodud aspekti, et ei tekkiks NullPointerException. See viga võib ilmuda siis, kui ma hakkan reas number 27 või 39 **req.getParameter(String arg0)** võetud väärtuse ja tavalist teksti väärtust võrrelda. Reas number 29 ma pean meetodile **login(String arg0, String arg1)** argumentideks edastama ka kaks tekstilist väärtust ja seal võib ka tekkida NullPointerException, kui ma ei kasuta seda aspekti.



Joonis 17: Parameetrite ja atribuutide aspekt klassi diagrammi kujul

4.3 Transaktsioonide aspekt

Transaktsioonid moodustavad ka ristlõikuva funktsionaalsust programmis ja tekkitavad koodi üleliigsust. Selle aspektiga ma jagan transaktsiooni loogikat põhifunktsionaalsest ja kirjutan transaktsiooni osa aspektiga. See aspekt täidetakse transaktsiooni meetodite ümber ehk tehakse transaktsiooni funktsionaalsust väljaspoolt ja püüakse neid meetodeid täita. Reas number 18 alustatakse transaktsiooni ja reas number 20 püüakse meetodit täita, **proceed** – viitab sellele meetodile, mis on aspekti pointcuti-ga seotud. Kui see täitmine õnnestub, siis lõpus kinnitatakse see transaktsioon reas number 25. Kui tekkib mingi viga, siis enne transaktsiooni katkestamist reas number 22, käivitub ka logimise aspekt, mida ma olen juba enne kirjutanud.

```
3  import frontend.service.SubjectService;
4
5  import javax.persistence.EntityTransaction;
6  import javax.servlet.http.HttpServletRequest;
7
8  public aspect Transactions {
9
10     pointcut dbTrans(SubjectService sService, HttpServletRequest req):
11         target(sService) && args(req) &&
12         execution(* add*ToDB(HttpServletRequest));
13
14     Long around(SubjectService sService, HttpServletRequest req): dbTrans(sService, req) {
15
16         EntityTransaction tx = sService.getTransactionsManager();
17
18         tx.begin();
19         try {
20             return proceed(sService, req);
21         } catch (Exception e) {
22             tx.rollback();
23             return null;
24         } finally {
25             tx.commit();
26         }
27     }
28 }
```

Joonis 18: Transaktsiooni aspekt [12]

Võib näha, et seda aspekti saab kasutada mitme meetodite jaoks, näiteks mull on see tehtud **addPersonToDB** ja **addEnterpriseToDB** meetodite jaoks. Niimoodi näeb välja üks neist (joonis number 19). Kommenteeritud on need read mida olid enne aspekti kirjutamist kasutusel (109–111 ja 136–139) . Oluline on ka see, et try-catch plokk läks transaktsiooni aspekti sisse.

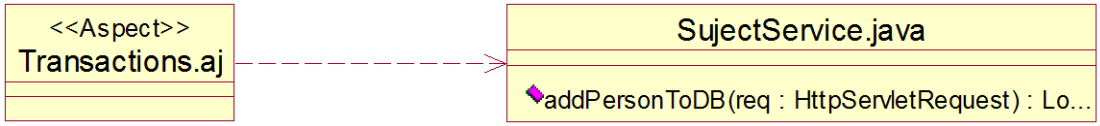
```
107     public Long addPersonToDB(HttpServletRequest req) throws ParseException {
108         Long newPersonId = null;
109 //      EntityTransaction tx = manager.getTransaction();
110 //      tx.begin();
111 //      try {
112             GenericDAO personDAO = new GenericDAO(manager, Person.class);
113             Person p = new Person();
114             p.setFirstName(req.getParameter("first_name"));
115             p.setLastName(req.getParameter("last_name"));
116             p.setIdentityCode(req.getParameter("identity_code"));
117             p.setBirthDate(parseDate(req.getParameter("birthdate")));
118             p.setCreatedBy(Long.parseLong((String) req.getAttribute("user_id")));
119             p.setUpdatedBy(Long.parseLong((String) req.getAttribute("user_id")));
120             p.setCreated(new Date());
121             p.setUpdated(new Date());
122             personDAO.persist(p);
123             newPersonId = p.getPerson();
124
125             long personOrCustomer = 1;
126             if (req.getParameter("customer_checkbox") != null) {
127                 personOrCustomer = 4;
128                 addSubjectAttributes(newPersonId, 4, req); //Customer attributes
129                 addCustomer(newPersonId, 4); //create customer
130             }
131             addSubjectAttributes(newPersonId, 1, req); //Person attributes
132
133             addSubjectAddress(newPersonId, personOrCustomer, req);
134             addSubjectContact(newPersonId, personOrCustomer, req);
135
136 //      } catch (Exception e) {
137 //          tx.rollback();
138 //      }
139 //      tx.commit();
140         return newPersonId;
141     }
```

Joonis 19: Transaktsiooni aspekti kasutamise näide [12]

Aga siin peab ka meeles pidama, et transaktsiooni haldamise operatsioon saab tekkida ainult **SubjectService.java** klassi sees, seepärast peab seda klassi all toodud funktsiooniga täiendada. See annab võimaluse transaktsiooni aspektile transaktsioone läbi viia.

```
37     public EntityTransaction getTransactionsManager() {
38         return manager.getTransaction();
39     }
```

Joonis 20: Transaktsiooni haldamise get meetod [12]



Joonis 21: Transaktsiooni aspekt klassi diagrammi kujul

4.4 Autoriseerimise aspekt

See aspekt on seotud turvalisuse mooduliga ja sisselogimise ehk autentimise funktsionaalsusega. See aspekt kontrollib, kas kliendi sessioon juba eksisteerib või ei. Kui see eksisteerib, siis kasutaja saab süsteemi kasutada, aga kui kasutaja pole sisse loginud või sessioon on aegunud, aspekt annab juhtimist **LoginController**-ile. Logimise kontroller omakorda näitab kasutajale logimis vormi ette ja ei anna süsteemi kasutamise õigust, kui logimine ei õnnestu.

```
3  import javax.servlet.http.HttpServletRequest;
4  import javax.servlet.http.HttpServletResponse;
5
6  import frontend.controllers.LoginController;
7  import frontend.service.MainService;
8
9  public aspect Authentication {
10
11     private MainService mainS;
12
13     //capture MainService constructor - Dependency injection
14     after(MainService ms) : execution(MainService.new()) && this(ms) {
15         mainS = ms;
16     }
17
18     pointcut checkUserSession(HttpServletRequest req, HttpServletResponse res):
19         within(frontend.controllers.FrontController) &&
20         execution(* doGet(HttpServletRequest,HttpServletResponse)) && args(req, res);
21
22     void around(HttpServletRequest req, HttpServletResponse res) : checkUserSession(req, res) {
23         mainS.session(req);
24         if (mainS.getSesson().loggedIn()) {
25             proceed(req, res);
26         } else {
27             new LoginController(mainS, req, res);
28         }
29     }
30 }
```

Joonis 22: Autentimise aspekti [12]

Sessiooni kontrollimiseks mul on vaja saada juurdepääs **MainService** klassile. Selles aspektis ma realiseerin seda juba teisiti, võrreldes näiteks eelmises punktis toodut transaktsiooni haldamissüsteemist. Realisatsiooni ridades number 14 ja 15 ma teen nii nimetatud

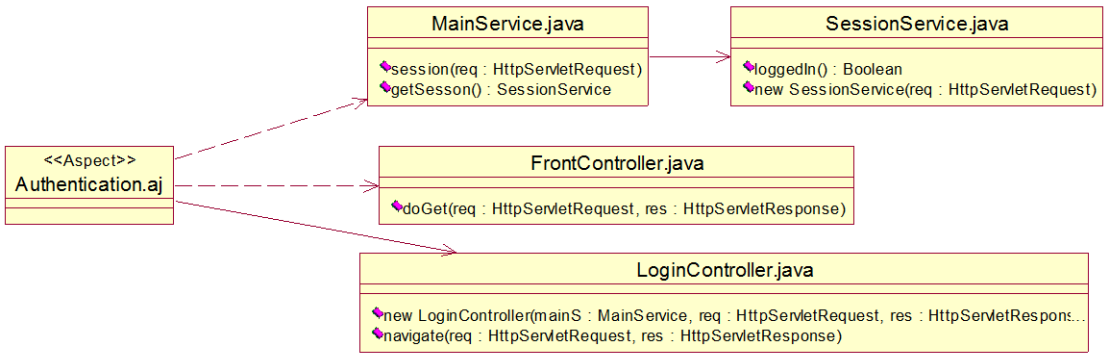
dependenci injection funktsionaalsust, aspekti abil. See on võimalik tänu sellele, et aspektide abil on võimalik iga klassi tekkimist programmis jälgida ja sellele juurdepääsu saada. See võimalus ei sunni mind **FrontController**-i koodi get meetodiga täiendada.

Aspekti põhi osa on ridades number 23 – 28. Seal toimub sessiooni kontroll ja juurdepääsu süsteemile (**proceed(req, res)** – **FrontController**-i doGet() meetodile viit) või logimis protseduurile (**new LoginController(mainS, req, res)**).

```
29      protected void doGet(HttpServletRequest req, HttpServletResponse res)
30              throws ServletException, IOException {
31
32  //      ms.session(req);
33  //      if (ms.getSesson().loggedIn()) {
34              System.out.println(req.getSession(false).getId());
35              String service = req.getParameter("service");
36              if (service.equals("login")) {
37                  new LoginController(ms, req, res);
38              } else if (service.equals("subject")) {
39                  new SubjectController(ms, req, res);
40              } else {
41                  req.getRequestDispatcher("/main.jsp").forward(req,res);
42              }
43
44  //      } else {
45  //          new LoginController(ms, req, res);
46  //      }
47      }
```

Joonis 23: Autentimise aspekti kasutamise näide [12]

Kui sessioon on korras, siis lubab see aspekt edasi rakendust juhtida **FrontController**-ile ja käivitub pildil 23 toodut kood. Kommenteeritud read näitavad seda funktsionaalust, mida realiseerib see aspekt. Võib näha, et aspekti funktsionaalsus vähendab FrontController-i koodi ja eraldab ristlõikuva funktsionaalsusega turvalisuse moodulit.



Joonis 23: Autentimise aspekti klassi diagrammi kujul

5. Enda summaarne kogemus ja uurimiste olulisemad tulemused

Enda kogemuse pealt tahan öelda, et AOP metoodika on väga efektiivne, mitmekülgne ja lai. See võimaldab palju asju teistpidi teha ja struktureerida. AspectJ on väga paindlik ja loogilise struktuuriga keel, seda oli minu jaoks lihte õppida ja aru saada. Aga kõike neid teadmisi ja AOP põhiprintsiipide valdamist sain ma ainult teooria ja praktika uurimise lõpus.

Enne seda pole ma selle teemaga enne kokku puutunud ja üldse aspektorienteeritud programmeerimisest kuulnud. Kui ma esimene kord leidsin sellest infot, siis kohe hakkasin selle vastu huvi tundma. Uurimiste alguses oli aga väga palju arusaamatuid kohti ja kõik tundus keeruliseks. Kõike printsiipide uurimine alguses võtsid palju aega. Edaspidi läks kõik juba paremini, kui hakkasin praktilist osa tegema. Kõik võttis aega umbes kaks kuud. Tegelesin sellega igapäevaselt.

Saan öelda, et AspectJ kasutamine nõuab hea Java keele teadmist ja OOP tundmist ning kasutamist. Need teadmised olid mul juba enne, seepärast aitas see mulle väga selle töö uurimisel. Töö käigus sain teada, et alguses AspectJ kasutamine tundub üleliigseks, keeruliseks ja vähem arusaadav, aga pärast muutub see lihtsaks ja läbipaistvaks. Selgus see, et AOP on tarvis kasutada ainult keeruliste ja suurte süsteemide projekteerimisel, lihtsate projektide puhul saab ilma AOP kasutamist hästi hakkama. Aga kui väikest süsteemi tahetakse perspektiivis muuta või laiendada, siis AOP peab kasutama kohe algusest peale.

Praegu on selline olukord, et AOP kasutatakse harva, raamatuid on mitte nii palju ja selle metoodika spetsialistide arv pole ka nii suur [5]. Info otsimiseks on tarvis kasutada erinevaid allikaid Internetist ja foorumeid. Minu uurimiste käigus tekkisid, ka sellised küsimused, mille peale polnud võimalik vastust raamatutest leida, siis abiks olid erinevaid foorumid, kus on võimalik küsimusi spetsialistidele esitada. See oli minu jaoks väga kasulik ja õpetlik.

Tahan tuua veel mõned statistika näiteid, mis selgitavad, et suured ettevõtted ja firmad kasutavad AOP ja AspectJ oma süsteemides.

Project	Domain	Size/complexity	Type	AO technologies used	AO features used	Crosscutting concerns modularized
Soarian (Siemens)	Healthcare	300 modules, 400 developers, 500 concurrent users per server	Industrial	AspectJ, fastAOP	Basic	Architecture validation, caching, auditing, performance monitoring
IBM WebSphere	Enterprise systems	WebSphere Product Center—3,700 classes, 43 aspects; Enterprise Service Bus—1,300 classes, 6 aspects	Industrial	AspectJ	Basic	Tracing, logging, first failure data capture
MOTOwi4	Telecom	12 components, 100 developers, 200,000 sessions per server	Industrial	WEAVR	Basic	Tracing, timeout
Toll system (Siemens, Lancaster University, EMNantes)	e-transport	90 classes, 20 aspects	Industry-academia	AspectJ, AWED, EA-Miner	Advanced	Charging, error handling, persistence, logging, monitoring, compatibility, security
Wit-Case (Katholieke Universiteit Leuven, SSEL-VUB, Alcatel-Lucent)	Telecom	8 Prolog modules, 1,200 lines of code, 1 technical architect, 2 developers	Industry-academia	Padus	Advanced	Fixed billing, duration billing, platform customizations, logging
Health-Watcher (Recife, Brazil)	Healthcare	130 modules, 3 developers, 1.5 million people served	Controlled experiment	AspectJ, CaesarJ	Advanced	Concurrency, distribution, exception handling, persistence, design patterns
AJHotDraw	Graphics	279 classes, 31 aspects, 1 AO developer, 25 OO developers	Controlled experiment	AspectJ	Basic	Persistence, design policies, contract enforcement, Undo command
MobileMedia	Imaging	3 developers, 51 classes, 36 aspects; a software product line for cell phones	Controlled experiment	AspectJ	Advanced	Exception handling, alternative and optional features

Tabel 1: AOP ja AspectJ kasutamine [5]

Firmad, kes kasutavad seda tehnoloogiat stimuleerivad ka selle edasiarendamist. Ühest küljest see toob kasu, aga teisest küljest võib tekitada palju probleeme [5]:

- Firmad vajavad kvalifitseeritud eksperte, kes saaksid sellega tegeleda.
- See võib olla riskantne mitte nii hästi levinud tehnoloogia kasutusele võtta ja stabiilset tarkvara produkti saada.
- Kui kasutada tarkvara, mis ei ole veel lõpuni arendatud, siis palju aega võib kuluda analüüsiks, kas valitud metoodika saab püstitatud ülesannet täielikult lahendada või ei.

Järgmiseks tahan AOP tehnoloogia arendamisel kasutusele võtmise plussid ja miinuseid tuua.

Poolt argumendid	Vastu argumendid
Tõstab modulaarsust	Vajab aega, et seda õppida ja kasutusele võtta
Võimaldab koodi ridu vähendada	Vähe spetsialiste, kes valdavad seda metoodikat
Eraldab rislõikuva funktsionaaluses põhifunktsionaalussesk	
Teeb süsteemi paindlikuks	

Tabel 2: AOP tehnoloogia kasutamine: plussid ja miinused

Enne AOP tehnoloogiat kasutamisele võtmist on mõistlik enda jaoks kolm küsimust esitada, et selgitada, kas seda on tarvis teha või ei [5]:

- Kas AOP integreerimine saab konkreetset probleemi lahendada või lihtsustada?
- Kas kõik plussid kaaluvad üle kõik kulud, mis on seotud metoodika kasutamisega?
- Milliseid tööriistaid ja abivahendeid on tarvis kasutada, et teatuid vigu vältida.

Selle peatükki info minu arvates oleks väga kasulik nendele, kes tahavad AOP tehnoloogiat uurida ja kasutusele võtta. Enda poolt tahan veel lisada, et AOP tehnoloogiat on ikka tarvis õppida, ka juhul kui töötavas süsteemis see kasutuses pole, sest tulevikkus võib see erinevate probleemide lahendamisel aidata.

Kokkuvõte

Objektorienteeritud programmeerimist kasutatakse täna kõikjal, sest see tõestas oma efektiivsust ja väärtuslikust. Aga kui OOP võimalustest ei piisa, et süsteemi lihtsustada või paremini modulariseerida, siis on väga mõistlik oma tähelepanu AOP poole pöörduda.

Minu töö näitab, et AOP on väga võimas tööriist, mis aitab ristlõikuva funktsionaalsust mooduliteks jagada. Aspektide abil saab ka uut funktsionaalsust juurde kirjutada, põhikoodi muutmata. Võimalik on ka standartseid funktsioone muuta ja ümber kirjutada. Kõik see lihtsustab süsteemi modifitseerimist ja edasiarendamist.

AspectJ on väga hea AOP realisatsioon Java keele jaoks. See omab loogilist struktuuri ja paindliku süntaksi, mis võimaldab väikese koodi ridade arvuga suurt funktsionaalsust saavutada. Seda võib näha töös toodud näidetes, kus ma uurisin AspetJ süntaksi ja tõin praktilisi näiteid oma projekti kirjutamisel ning analüüsisin ja selgitasin kirjutatud aspektide funktsionaalsust. Töö selgitab seda, et ristlõikuva funktsionaalsust saab väga hästi AOP abil mooduliteks jagada.

Lõpuks tahan öelda, et AOP võib palju kasu tuua, kui osata seda õigesti kasutada. Ise kavatsen ma kasutusele võetud teemaga ka edaspidi tegelemist jätkata.

Summary

Object-oriented programming is widely used as it is very efficient and can be used to solve different tasks. Sometimes it is impractical to use OOP due to some limitations, then aspect-oriented programming can be used instead. In my research I have tried to prove that AOP is very powerful tool that is able to do modularizing as well as crosscutting-concerns. It is possible to add new functions to program without changing the actual code, it can be easily done by using aspects. Also it is possible to edit and rewrite default functions which would help to modify and maintain system easily.

AspectJ is a very good implementation of AOP for the Java programming language. AspectJ has a logical structure as well as a flexible syntaxes that allows a user to write small amount of code but at the same time to get lots of functions, it is called write less do more. I have a few examples of AspectJ syntaxes from my coursework that I have included in my research. I have analysed them as well as explained the functionality of written aspects. In my research I have also introduced a section that explains that crosscutting-concerns can be easily divided into modules by using AOP method.

In conclusion I would like to state that AOP is very useful and effective tool that helps to solve lots of different problems. I really like AOP and plan to keep using it in future.

Kasutatud kirjandus

1. Object-oriented programming [WWW]
   http://en.wikipedia.org/wiki/Object-oriented_programming

2. Cross-cutting concern [WWW]
   http://en.wikipedia.org/wiki/Cross-cutting_concern

3. Aspect-oriented programming [WWW]
   http://en.wikipedia.org/wiki/Aspect-oriented_programming

4. Aspect [WWW]
   http://en.wikipedia.org/wiki/Aspect_%28computer_science%29

5. Anwendung von Aspektorientierter Programmierung Autoren: Florian Strieg, Seminarausarbeitung Fachtext (Verantwortlicher: Prof. Frank Dopatka) [WWW]
   https://mediaarchiv.reutlingen-university.de/?media&id=323

6. AspectJ [WWW]
   http://en.wikipedia.org/wiki/AspectJ

7. Thinking in Java, 4th edition - Bruce Eckel

8. AspectJ in Action, Second Edition Ramnivas By Laddad, Foreword by Rod Johnson 2009

9. AOP põhimõtete graafiline esitlus [WWW]
   http://s24.postimg.org/jf9edhoat/Screenshot.png

10. IDU0200_2014 - IDU0200-Veebipõhiste rakenduste arhitektuur, disain ja tehnoloogia (2014 kevad, R.Liivrand) [WWW]
    http://maurus.ttu.ee/aine_index.php?aine=325

11. Praktikaülesande „SUBJEKTID" kirjeldus: andmebaasi struktuurid ja rakenduse funktsionaalsuse kirjeldus [WWW]
    http://maurus.ttu.ee/ained/IDU0200_2014/doc/35/SUBJEKTID_juhend.pdf

12. Minu lähtekood kirjutatud subjektide süsteemi jaoks [WWW]
    https://github.com/jarik888/Subject-system-with-AspectJ