Tehnike aspektno orijentiranog programiranja

Maričić, Mihovil

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mihovil Maričić

Tehnike aspektno orijentiranog programiranja

DIPLOMSKI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mihovil Maričić

Matični broj: 46431/17–R

Studij: Informacijsko i programsko inženjerstvo

TEHNIKE ASPEKTNO ORIJENTIRANOG PROGRAMIRANJA

DIPLOMSKI RAD

Mentorica:

Prof. dr. sc. Danijel Radošević

Varaždin, rujan 2019.

Mihovil Maričić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autorica potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Objasniti tehnike aspektno orijentiranog programiranja, prikazati nastanak aspekata kroz povijest programiranja, usporediti različite okvire (eng. Framework) za kreiranje aspekata poput *Spring-a, AspectJ-a, AspectC++-a* i slično. Usporediti rad objektno orijentiranog programiranja s aspektno orijentiranim te prikazati po čemu se razlikuju od proceduralnog. Slikovito pokazati i opisati korištenje ključnih elemenata za kreiranje aspekata u programskom kodu putem anotacija i putem *XML-a*. Usporedba aspekata s takozvanim *interceptorima* uz objašnjenje različitosti i sličnosti. Konačno, objasniti za što se koriste aspekti i prikazati različite načine njihove implementacije te prkaz svega naučenog uz izradu aplikacije u praktičnom dijelu rada koristeći se *Spring* okvirom i *Intellij* razvojnim okruženjem.

Ključne riječi: Java; Aspect; Spring; AOP; OOP; AspectJ; AspectC++; Annotations

Sadržaj

1. Uvod

Tema ovog diplomskog rada su tehnike aspektno orijentiranog programiranja. U današnje vrijeme programeri imaju jako puno alata i različitih programskih jezika na raspolaganju u svrhu razvijanja nekog proizvoda ili aplikacije. Najčešći pristup je koristeći koncepte objektno orijentiranog programiranja. Ta, objektno orijentirana paradigma, je pristup programiranju na visokoj razini koja je nastala iz nižih jezika poput C-a. Razlog nastajanja bio je taj što su programeri i arhitekti aplikacija shvatili da mogu na bolji način koristiti resurse računala i ubrzati proces razvoja putem korištenja već napisanih komponenti. Drugim riječima, nastala je kao odgovor na nedostatke nižih programskih jezika. Isto tako nastala je aspektno orijentirana paradigma kao odgovor na nedostatke objektno orijentiranog programiranja. Koncept aspektno orijentirano programiranje prvi put se spomenulo u članku napisanom 1997. godine u razvojnom centru koji se zove „Xerox Palo Alto Research Center“. Napisao ga je Gregor Kiczales i članak se zvao „Aspect-Oriented Programming“. U tom članku objasnio je zašto su aspekti potrebni i kako će se riješiti spomenuti nedostaci. Sve to proći ćemo u nastavku ovoga rada.

Veliki broj programera nije ni čula za riječ aspekt jer nije tema koja se uči na fakultetu zato što je u našem svijetu, svijetu programera, relativno novi pojam koji se ne koristi često ili programer niti ne zna da ga koristi zbog dobro pripremljenih okvira za rad koji obave većinu posla u pozadini ili kako bi se reklo, ispod haube

U radu će se primjeri prvenstveno raditi u Java-i, ali neće biti zanemareni ostali programski jezici koji podržavaju korištenje aspekata u svom radu. Naravno, svaki korak bit će analiziran i objašnjen na razumljiv način i s dobrim, praktičnim i zanimljivim primjerima.

U zadnjem dijelu rada bit će napravljena aplikacija koja će objediniti sve što će se proći u radu te tako omogućiti bolje razumijevanje i svrhu korištenja takvog načina rada. Vidjet ćemo da aspekti nisu potrebni kako bi se izradila aplikacija, međutim to je isto kao kada kažemo da ni objektno orijentirani pristup nije potreban jer se sve može napisati u nižim jezicima. Aspekti su tu da olakšaju i ubrzaju rad programerima te ujedno da kod čine razumljivijim i preglednijim.

Iako rad sadržava osnovne korake za kreiranje aspekata, predzadnje u objektno orijentiranom programiranju i poznavanje *Java*-e znatno će olakšati razumijevanje gradiva koje sadrži programski kod.

2. Programske paradigme kroz povijest

U ovom poglavlju proći ćemo kroz povijest programske paradigme počevši s imperativnim programiranjem koji predstavlja temelj na kojem je sve započelo. Programska paradigma je stil programiranja, odnosno način razmišljanja pri kreiranju nekog programskog proizvoda ili aplikacije. Svaki programski jezik prati svoju metodologiju razvoja koja koristi alate i tehnike za izvršavanje svoga zadatka. Spomenuta metodologija zapravo predstavlja programsku paradigmu. (Rani, Introduction of Programming Paradigms, n.d.) Aspekti predstavljaju naj noviji način razmišljanja i pisanja koda tijekom rješavanja nekog zadatka, a više o njima pričat ćemo u nastavku.

2.1. Imperativna programska paradigma

Imperativna programska paradigma je, kako sam već spomenuo, najstariji oblik pisanja programskog koda. Temelji se na *Von Neumann*-ovoj arhitekturi što znači da mijenja stanja programa putem svojih linija koda (eng. Assignment statements). Programski jezici koji koriste ovaj pristup su C, Fortran i Basic. Primjer takvog koda svima nam je poznat, a izgleda ovako:

```
int marks[5] = { 12, 32, 45, 13, 19 } int sum = 0;
float average = 0.0;
for (int i = 0; i < 5; i++) {
    sum = sum + marks[i];
}
average = sum / 5;
```

Ova paradigma dijeli se na tri kategorije koje uključuju proceduralno programiranje, objektno orijentirano programiranje (eng. Object Oriented Programming, kraće OOP) i paralelno procesiranje. Također postoji i deklarativna programska paradigma, međutim kako to nisu teme ovoga rada zadržat ćemo se još na proceduralnoj i objektno orijentiranoj koje su bitne za razumijevanje teme u kasnijem dijelu rada. (Rani, Introduction of Programming Paradigms, n.d.)

2.2. Proceduralna programska paradigma

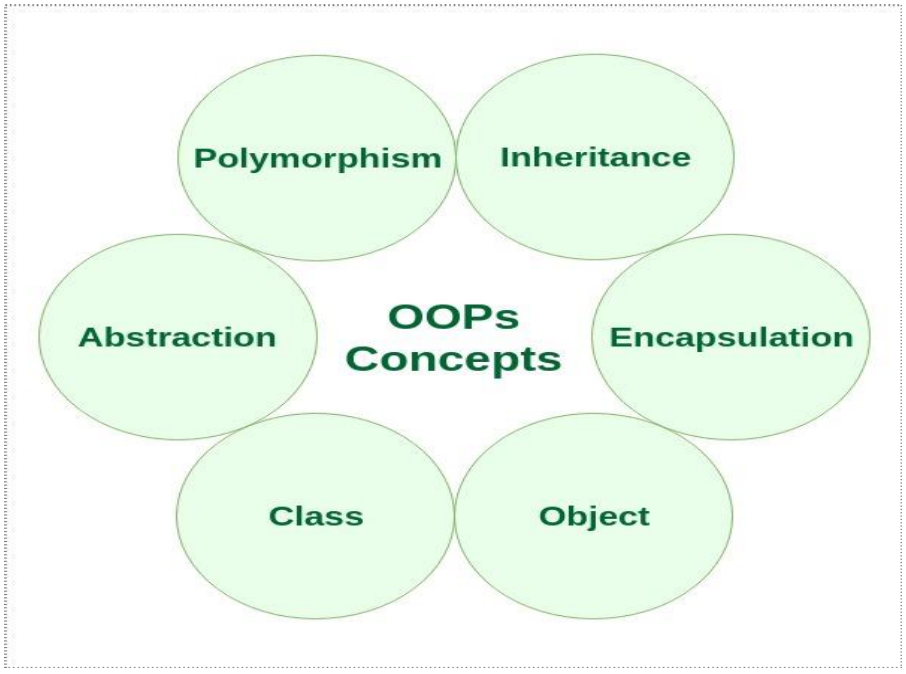
Proceduralni pristup ne razlikuje se previše od prethodno objašnjenog imperativnog pristupa. Jedina razlika je ta što proceduralni pristup ima mogućnost ponovnog korištenja koda kako bi se olakšao i ubrzao proces programiranja. Programski jezici koji mogu koristiti ovaj pristup su: C, C++, Java, ColdFusion i Pascal. Primjer proceduralnog programiranja bi bilo

uključivanje svima poznate biblioteke *iostream*, i korištenja prostornog imena *std* u C++-u. (Rani, Introduction of Programming Paradigms, n.d.)

2.3. Objektno orijentirano programiranje

Ovaj način programiranja danas je najraširenij u svijetu jer omogućuje pisanje kvalitetnog koda koji se može ponovno koristiti i time olakšava izradu aplikacije. Program koji se razvija ovim pristupom sastoji se od kolekcije klasa i objekata koji međusobno komuniciraju u svrhu odrađivanja nekog cilja. Temelj ovog pristupa je klasa objekt od koje sve druge klase nasljeđuju svoje ponašanje. Cilj OOP je uvođenje pojmova iz stvarnog svijeta u programiranje poput nasljeđivanja (eng. Inheritance), polimorfizma (eng. Polymorphism) i skrivanja, odnosno enkapsulacije (eng. Encapsulation).

Sljedeća slika prikazuje osnovne koncepte objektno orijentiranog programiranja:



Slika 1.Koncepti objektno orijentiranog programiranja (Baeldung, Introduction to Pointcut, 2017)

Svi jezici više razine poput Java-e i C#-a koriste gore prikazane koncepte u svome radu, a neke ćemo proći u sljedećim poglavljima. (Indika, 2011)

2.3.1. Objekt i klasa

U objektno orijentiranom programiranja klasa predstavlja predložak nekog entiteta iz stvarnog svijeta, a objekt je instanca te klase. Objekt ne može postojati bez klase koju inicijalizira, i isto tako klasa bez inicijalizacije ne može se koristiti u radu osim u nekim specifičnim uvjetima koje ćemo također proći kad budemo razgovarali o apstrakcijama. Sljedeći kod prikazuje primjer klasa i objekata u Java programskom jeziku. (Gibbs, 2019) (Rani, OOPs | Object Oriented Design, n.d.)

```
public class Auto {
    private String boja;
    private int maxBrzina;
    private String vrsta;
    private String model;

    public Auto(String boja, int maxBrzina, String vrsta, String model) {
        this.boja = boja;
        this.maxBrzina = maxBrzina;
        this.vrsta = vrsta;
        this.model = model;
    }

    public void vozi(){
        System.out.println("vozim auto");
    }
}
```

U ovom kodu vidimo primjer klase auto sa svojim vrijednostima, konstruktorom (eng. Constructor) i metodu vozi. Kako smo već rekli ova klasa sama po sebi nema nikakve koristi u kodu dok se ne inicijalizira, a to radimo na sljedeći način:

```
Auto auto = new Auto("zelena", 250, "karavan", "626");
mazda.vozi();
```

Vidimo kreiranje objekta pod nazivom auto gdje smo pomoću konstruktora ubacili sve željene vrijednosti i dobili mogućnost korištenja metoda te klase. Ovakav način kreiranja objekta unutar neke druge klase naziva se umetanje zavisnosti (eng. Dependency injection), a kasnije ćemo također vidjeti kako *Spring* okvir rješava taj problem. (Indika, 2011) (Gibbs, 2019)

2.3.2. Prednosti objektno orijentiranog programiranja

Glavna prednost ovog pristupa programiranju je mogućnost ponovnog korištenja napisanog koda zbog svoje mogućnosti da složene problema razdvoji u više modula kojima je lakše upravljati i održavati. Također omogućuje kontrolirani protok podataka te ih, koristeći enkapsulaciju, može zaštititi od vanjskih funkcija. (Gibbs, 2019)

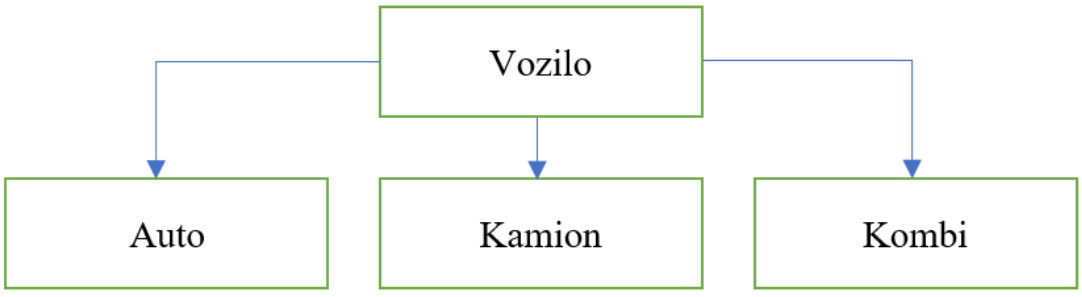
Još neke od prednosti su te da OOP modelira svoje podatke koristeći entiteta stvarnog svijeta na dobar način kao što je prikazano u prijašnjem poglavlju. Omogućuje i lakše testiranje programa kojeg razvijamo te lakši pronalazak grešaka u radu putem (eng. Debugging). Sve u svemu OOP je trenutno najjači i najbrži pristup programiranju ali još uvijek nije savršen. Nedostatke ćemo proći u sljedećem poglavlju i objasniti način na koji se rješavaju ti nedostaci. (Indika, 2011) (Rani, OOPs | Object Oriented Design, n.d.)

2.3.3. Nedostaci objektno orijentiranog programiranja

Znanje koje je potrebno za razvijanje složene aplikacije je visoko i zahtjeva dobro poznavanje arhitekture cijelog sustava kao i njenu domenu. Programer treba biti u mogućnosti razmišljati na višoj, apstraktnijoj razini kako bi bio u mogućnosti napisati kvalitetan kod i tako iskoristiti sve mogućnosti koje OOP nudi. Primjer nekih slučajeva u kojima treba unaprijed razmišljati kako bi se izbjeglo mijenjanje koda (eng. Refactoring) i ponovno pisanje istog možemo vidjeti na sljedećem problemu. (Rani, OOPs | Object Oriented Design, n.d.)

2.3.4. Code Scattering

Zamislite da radite sustav koji upravlja vozilima. Jedna od klasa zasigurno bi bila slična našoj klasi auto, međutim što kad bi trebali ubaciti dodatne klase poput kamiona ili kombija? Tada bi trebali ponovno koristiti neke od varijabli i metoda za opis takvog vozila i tako pišemo kod koji će u konačnici raditi, ali neće biti napisan na dobar način. Takav problem nazive se *Code Scattering* i rješava se korištenjem koncepta nasljeđivanja kao rješenja u kojem bi imali klasu vozilo koja bi sadržavala zajedničke vrijednosti svih ostalih klasa i tu bi klasu naslijedile ostale klase poput na primjer auta i kamiona. Tada bi dijagram izgledao ovako:



Slika 2. Dijagram klasa vozila

Na slici vidimo tako zvanu super klasu Vozilo i njene pod klase Auto, Kamion i Kombi. Naravno u stvarnosti ovaj primjer bi bio puno kompliciraniji međutim za prikaz koncepta nasljeđivanja bit će dovoljan. Klasa vozilo se u ovom slučaju ne može instancirati, odnosno ne

možemo napraviti objekt klase vozilo. Prije smo spomenuli da postoje klase koje se ne bi trebale moći instancirati i ovo je jedan od takvih primjera. U takvom slučaju klasu definiramo kao apstraktnu s ključnom riječi *abstract* i tada je ona dostupna samo ostalim klasama za nasljeđivanje odnosno proširenje svojih mogućnosti. (Yang, n.d.) (Laukkanen, 2008)

2.3.4.1. Primjer nasljeđivanja kao rješenja

Iznad smo ukratko pokazali kako nasljeđivanje predstavlja jedan od načina kako bi izbjegli *Code Scrattering*, a sada ćemo to vidjeti na jednom praktičnom primjeru.

Prvo moramo dodati klasu Vozilo koja izgleda ovako:

```
Public abstract class Vozilo {
    private String boja;
    private int maxBrzina;
    private String vrsta;
    private String model;

    public Vozilo(String boja, int maxBrzina, String vrsta, String model) {
        this.boja = boja;
        this.maxBrzina = maxBrzina;
        this.vrsta = vrsta;
        this.model = model;
    }

    public abstract void vozi();
}
```

Ovo je primjer super klase koja sadrži zajedničke vrijednosti svih vozila, a nakon toga možemo dodati pod klase Auto, Kombi, Kamion i slično. Na primjer sada bi klasa Kamion izgledala sljedeće:

```
Public class Kamion extends Vozilo {
    private int maxNosivostTereta;

    public Kamion(String boja, int maxBrzina, String vrsta, String model,
int maxNosivostTereta) {
        super(boja, maxBrzina, vrsta, model);
        this.maxNosivostTerete = maxNosivostTereta
    }

    @Override
    public void vozi(){
      System.out.println("Vozim kamion");
    }
}
```

Tu vidimo da klasa Kamion nasljeđuje klasu Vozilo s ključnom riječju *extends* što znači da ta klasa ima sadrži iste vrijednosti kao klasa Vozilo te su dodane nove vrijednosti koje su specifične samo za tu klasu. Objekt te klase bi stvorili na sljedeći način:

```
Vozilo vozilo = new Kamion("zelena", 250, "karavan", "626", 3500);
```

Na sličan način bi stvarali i ostale klase poput Auta i Kombija te bi svaka od njih bi naslijedila klasu Vozilo. Tako bi se riješili problema ponavljajućih vrijednosti i izbjegli *Code Scattering*.

2.3.5. Code Tangling

U ovom dijelu doznat ćemo jedan od najvećih problema OOP, a to je zapletanje koda (eng. Code Tangling). Ideja svakog programera bi bila da metoda radi točno ono za što je definirana. Na primjer kada bi imali metodu koja prima dva broja kao parametar i vraća zbroj ta dva broja primjer je dobro definirane metode. Takav kod izgledao bi ovako:

```
private int zbrojDvaBroja(int prviBroj, int drugiBroj){
    return prviBroj + drugiBroj;
}
```

Sada kad bi u taj postojeći kod htjeli dodati zapis (eng. Logging) o izvođenju te metode, on bi izgledao ovako:

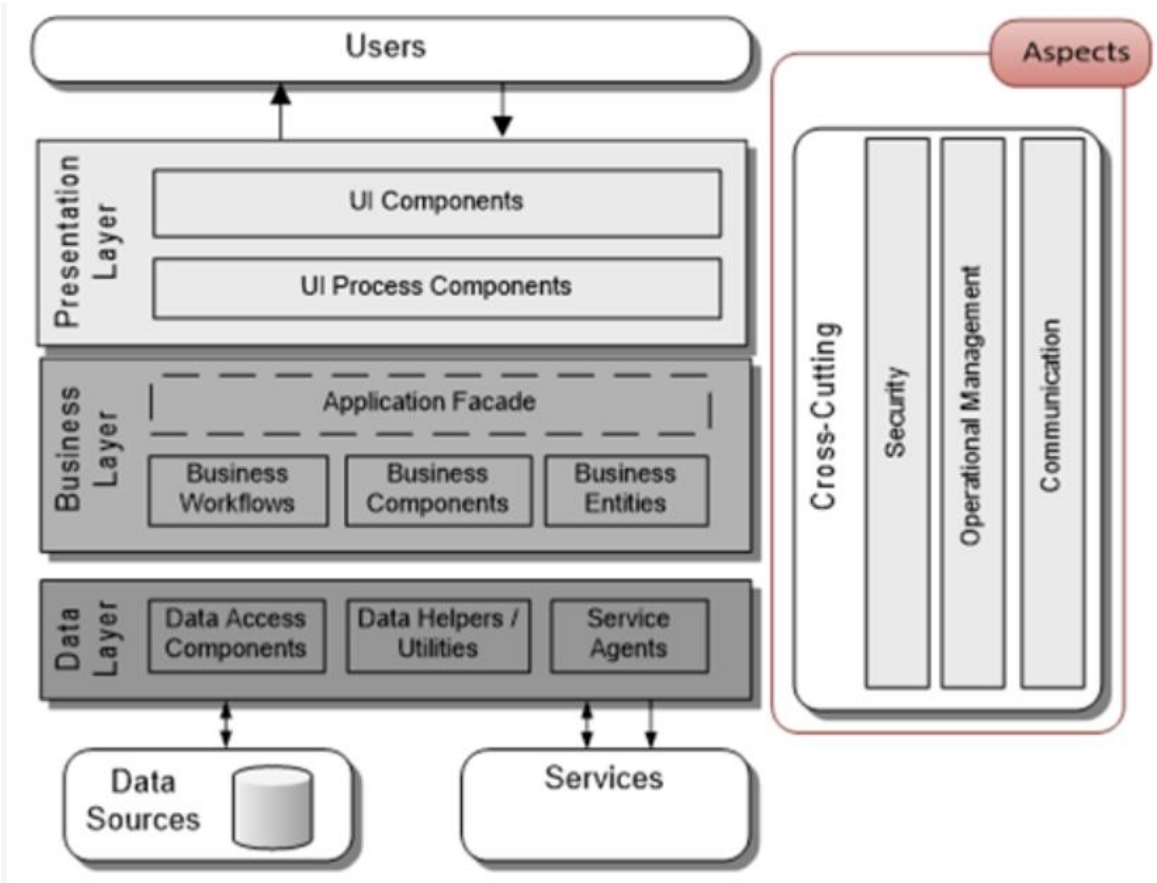
```
private int zbrojDvaBroja(int prviBroj, int drugiBroj){
    Logger.getLogger(Demo.class).entering("Demo", "zbrojDvaBroja");
    return prviBroj + drugiBroj;
}
```

U tom slučaju dolazimo do umetanja odgovornosti u metodu za zbroj dva broja jer joj dodajemo još jednu odgovornost, a to je da zapiše u log svoje izvršavanje. Zamislite kada bi taj log trebalo dodati u sve metode svih klasa. Taj problem naziva se *Code Tangling* i predstavlja jedan od problema OOP. Kada bi se još odlučili za promjenu log-a, ili kada bi htjeli dodati još neku informaciju tada bi trebali to mijenjati u svim metodama naše aplikacije. Tu opet dolazi do prije spomenutog *Code Scattering*-a. (Yang, n.d.) (Laukkanen, 2008)

Zadnja tri poglavlja opisuju problem presjecajućih svojstava (eng. Cross Cutting Concerns) koja se mogu riješiti uporabom aspekata, što je i tema ovoga rada.

3. Cross cutting concerns

Kako smo do sada vidjeli OOP omogućuje programerima brže i kvalitetnije izrađivanje aplikacije koristeći svoje koncepte i mogućnost ponovno korištenja koda. Na žalost to nekad nije dovoljno jer kada je riječ aplikaciji koja mora izvršavati logiku prilikom poziva svojih metoda OOP ne nudi kvalitetno rješenje. Takve situacije najčešće se pojavljuju prilikom transakcija kod rada s bazom podataka, kod provjere autorizacije prijavljenih korisnika ili kod upravljanja pogreškama (eng. Error Handling). *Cross cutting concerns* (kraće CCC) je dio svake aplikacije koja se sastoji od više slojeva i može dodatno zakomplicirati posao programera ako dođe do promjene u pojedinim slojevima aplikacije. Za takve posebne slučajeve razvijena je aspektno orijentirana paradigma koja omogućuje umetanje sadržaja na jednom mjestu u aplikaciji koja se može koristiti u svim slojevima kako možemo vidjeti na sljedećoj slici. (Vaidya, 2019) (Laukkanen, 2008)



Slika 3. Aplikacija podijeljena na slojeve (Cshandler, 2015)

Na slici možemo vidjeti kako svaka složenija aplikacija funkcionira. Sastoji se od prezentacijskog sloja, poslovnog sloja i podatkovnog sloja. Prezentacijski sloj je ono što korisnik vidi na svom računalu i putem kojega zapravo upravlja aplikacijom. Poslovni sloj je onaj dio aplikacije u kojem se nalazi sva poslovna logika aplikacije. Tamo se nalazi kod koji u pozadini izvršava korisnikove zadatke i koji je direktno odgovoran za pravilan rad aplikacije. Treći, podatkovni, sloj direktno je odgovoran za komuniciranje s bazom podataka. Ta tri sloja su dovoljna za uspješan rad svake aplikacije, međutim kako smo do sad raspravljali postoje dijelovi koji se protežu kroz sva tri sloja i koji se mogu realizirati korištenjem OOP, ali ne na kvalitetan način. Takvi scenariji se rješavaju korištenjem aspekata. Prije nego detaljno krenemo s objašnjavanjem aspekata prvo ćemo ih usporediti s OOP i kako bi bolje objasnili njihovu ulogu, usporedit ćemo ih a takozvanim *interceptor*-ima. (Cshandler, 2015)

3.1. Interceptori

Kod izrade web aplikacije svi su se susreli sa pojmovima poput zahtjeva (eng. Request) i odgovora (eng. Response). Ideja je da korisnik putem svoga web preglednika (eng. Browser) napravi zahtjev na *backend* dio aplikacije, odnosno servera, nakon čega čeka odgovor. Jedan primjer bi bio da se korisnik nalazi na stranici za prijavu u mail servis fakulteta Organizacije i informatike. Nakon unosa korisničkog imena i lozinke, pritiskom na gumb za prijavu šalje zahtjev te ako se taj zahtjev odobri prikazuje mi se odgovor servera sa sadržajem svojih poruka. *Interceptori* se u mogu koristiti za upravljanje tim zahtjevima i odgovorima u smislu da mogu izvršiti određeni kod prije nego li se neki zahtjev ili odgovor u potpunosti obavi. Dobar primjer bi bio provjera korisnikove autentičnosti prilikom slanja maila, ili spremanje zapisa o korisnikovim akcijama. *Interceptori* su dakle jedan od načina za rješavanje CCC prilikom upravljanja zahtjevima i odgovorima kod web aplikacije. Aspekt radi istu stvar, ali nije ograničen na zahtjev i odgovor. Može se koristiti prije ili poslije poziva bilo koje metode ili prilikom definiranje neke varijable. Neki okviri poput *AspecJ*-a omogućuju i provjeru prenesenih parametara, ali detaljnije o tome u sljedećim poglavljima. (Zhou, 2019) (Debnath, 2014) (ProgrammingAndPolitics, 2007)

3.2. Što je aspekt?

U ovom poglavlju odgovorit ćemo na pitanje što je zapravo aspekt. Kako bi mogli razumjeti taj pojam trebamo se vratiti na poglavlje o cross cutting concerns jer na najjednostavniji način svaki aspekt predstavlja CCC, odnosno brigu (eng. Concern) koju je teško razumjeti u zatvorenom sustavu koji se proteže kroz cijelu aplikaciju. Ta briga predstavlja prije spomenute primjere sigurnosti, zapisa i sličnih stvari. Upravo zbog takvih problema u

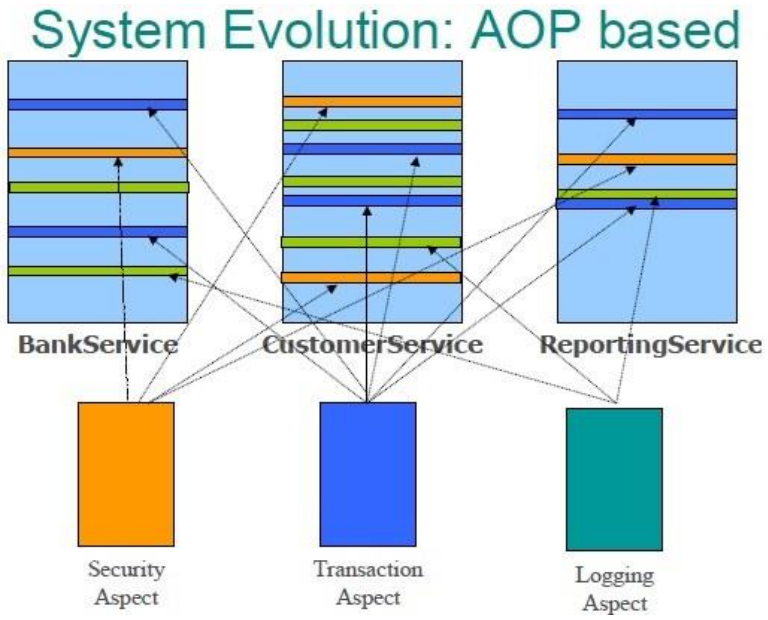
programiranju razvijena je aspektno orijentirana paradigma koja je omogućuje u jedan modul aplikacije ubaciti rješenje koje će koristiti cijeli sustav. (Yang, n.d.) (Laukkanen, 2008)

Prema Gregoru Kiczalesu, osobi koja je zajedno sa svojim suradnicima objavila prvi rad o aspektno orijentiranom programiranju, aspekt je funkcionalnost u aplikaciji koja se dodaje u sustav ako se ne može jasno enkapsulirati u opću proceduru unutar programa. Aspekti za razliku od komponenti ne teže dekompoziciji sustava, već teže funkcionalnosti koje utječu na performansu i semantiku komponenti unutar sustava. (Laukkanen, 2008)

Kasnije ćemo još detaljnije proći kroz temu aspekta i aspektno orijentiranog programiranja, a za sad je bitno razumjeti, kako sam već naveo, da aspekt predstavlja modul koji rješava probleme CCC-a.

3.2.1. AOP vs OOP

Sada ćemo proći kroz glavne razlike i sličnosti OOP i AOP. Oba pojma predstavljaju programsku paradigmu koja je temeljni dio programiranja, ali su zamišljene na drugačiji način. Objektno orijentirano programiranje fokusira se na svoje temeljne koncepte objekta i klasa kako bi riješio neki problem koristeći entiteta iz stvarnoga svijeta. To ga čini poprilično jednostavnijim za razumijevanje od aspektno orijentiranog programiranja. AOP se fokusira na razdvajanje komponenti programa koji se protežu kroz cijelu aplikaciju u svoje module. Primjer kako aplikacija izgleda korištenjem aspekata prikazana je na slici ispod.



Slika 4. Primjer aplikacije s aspektima (Dinesh, 2017)

Vidimo prikaz aplikacije koja koristi tri različita servisa sa svoj rad. Svaki od tih servisa u sebi sadrži više različitih modula. Narančastom bojom je prikazan aspekt koji se obavlja posao sigurnosti aplikacije poput autorizacije i autentifikacije. Sigurnost koriste sva tri servisa na više mjesta i to je primjer prije spomenutog *code* tangling-a i *code scattering*-a. Isto vrijedi za ostale aspekte poput transakcija i spremanja zapisa. Kada bi promatrali prvi dio slike bez aspekata dobili bi rješenje u OOP, odnosno razbacan kod kroz sve servise više puta. Čim ubacimo aspekte možemo lokalizirati te probleme na jednom mjestu i tako omogućiti takozvani *Speratation of Concerns* (kraće SoC). To znači da bi svaka klasa i svaka metoda radila samo ono za što je zamišljena i ništa više. Aspekti su tu da, ako je potrebno, prošire mogućnosti tih klasa i metoda. (Yang, n.d.) (Indika, 2011) (Dinesh, 2017)

Ovo je dobar primjer u kojem vidimo kako OOP i AOP nisu paradigme koje bi se trebale uspoređivati u smislu koja je bolja, već koristiti oba dvije kako bi se razvio kvalitetan i razumljiv kod kojeg je kasnije lakše refaktorirati i modificirati.

4. Tehnike aspektno orijentiranog programiranja

U ovome poglavlju prvo ćemo proći kroz elemente aspektno orijentiranog programiranja. Objasnit ćemo na koji način se pišu aspekti u različitim okvirima za rad i pokazati primjere koda kako bi lakše razumjeli njihovu bit. Prije smo spomenuli za što se aspekti koriste a sada ćemo korak po korak proći kroz primjere pisanja aspekta pomoću njegovih koncepata. U primjerima ispod koristit će se čisti *Spring AOP* za pisanje aspekata.

Slika 5. AOP koncepti (Vaidya, 2019)

4.1. Join point

Spojna točka (eng. Join point) je mjesto u kodu koje predstavlja kandidata za uvođenje aspekta. Te točke moraju biti jednoznačno definirane, a najčešće točke spajanja su poziv konstruktora ili metoda. Naravno različiti okviri za rad podržavaju i različite točke spajanja pa tako možemo točku spajanja definirati i izvršavanjem neke metode tako da upravljamo bačenom iznimkom (eng. Exception). Neki okviri također omogućuju i točku spajanja prilikom definiranja pojedinih varijabli unutar klase što predstavlja jako snažan mehanizam koji se može koristiti kao na primjer provjera vrijednosti koja je dodijeljena pojedinoj varijabli. To može biti neko ograničenje poput provjere da je vrijednost veća od nule i slično. Primjer spojnih točki

možemo vidjeti na sljedećem kodu. (Vaidya, 2019) (Laukkanen, 2008) (I. Bergman, C. Goransson, 2004)

```
public class Klasa {
  public int polje;

  public Klasa() {
    polje = 0;
  }
  public String metoda(int i) {
    return Integer.toString (polje + i);
  }
}
```

U primjeru imamo klasu i sada akcije nad tom klasom predstavljaju točke spajanja.

```
Klasa klasa = new Klasa(); // poziv konstruktora
int a = klasa.polje; // dodjela vrijednosti varijabli

String s = klasa.metoda (10); // poziv metode
```

4.2. Pointcut

Točka spajanja (eng, Pointcut) predstavlja izraz u kojem navodimo nad kojim će se od prethodno spomenutih spojnih točaka izvršiti određena akcija. Ovisno o okviru putem kojeg koristimo aspekte točke spajanja možemo definirati putem takozvanih anotacija ili unutar XML datoteke. Kada pričamo o *Java* programskom jeziku anotacije predstavljaju meta podatke koje kompajler čita i ovisno o anotaciji mijenja ponašanje koda. Najpoznatije anotacije za koje svi znamo, i ugrađene su (eng. Built in) u *Java*-i su na primjer *@Override* ili *@Depricated*. Primjer definiranja točke spajanja u XML-u možemo vidjeti u sljedećem kodu. (Vaidya, 2019) (Laukkanen, 2008) (I. Bergman, C. Goransson, 2004)

```
<aop>
  <aspect class="NekiAspect" scope="PER_VM"/>

  <bind pointcut="execution(public String NekaKlasa->nekaMetoda(..))">
    <advice name="nekiSavjet" aspect="NekiAspect"/>
  </bind>
</aop>
```

Ova datoteka nam govori da će se savjet (eng. Advice) pod nazivom nekiSavjet vezati na metodu nekaMetoda unutar klase NekaKlasa. Više o savjetima govorit ćemo u sljedećem poglavlju. Isti primjer korištenjem anotacija možemo vidjeti u nastavku. (Vaidya, 2019)

```
@Aspect (scope = Scope.PER_VM)
public class NekiAspect {
  @Bind (pointcut="execution("* NekaKlasa -> nekaMetoda ())")
  public Object trace (Invocation invocation) throws Throwable {
```

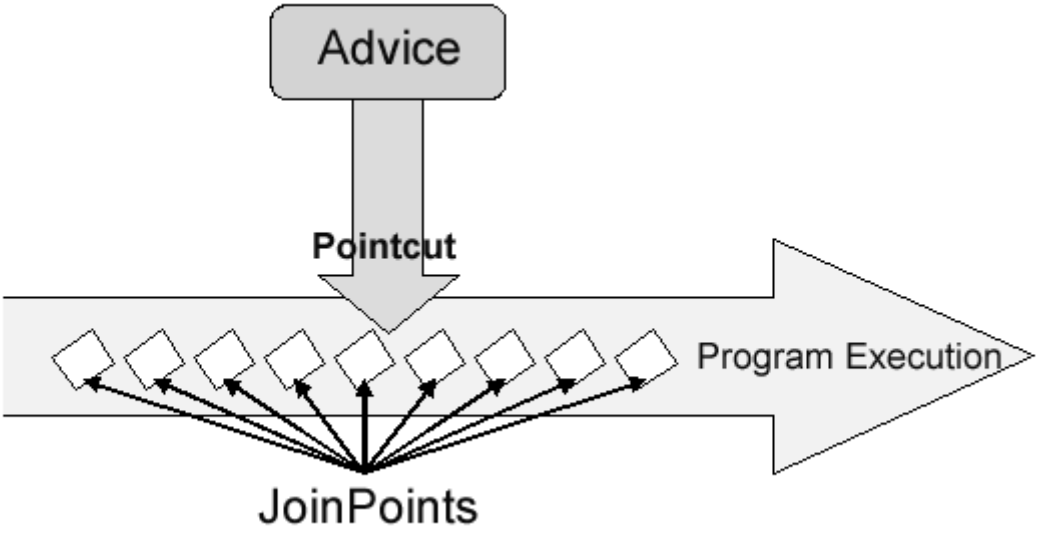
```
    try {
      System.out.println("Ulazim u točku spajanja");
      return invocation.invokeNext ();
    } finally {
      System.out.println("Izlazim iz točke spajanja");
    }
  }
}
```

Vidimo da su anotacije zapravo i jednostavnije za korištenje jer se baziraju na *Java* kodu umjesto na XML datoteku.

Također izraze točke spajanja možemo i kombinirati koristeći sljedeće logičke operatore: && što predstavlja logički izraz i (eng. And), || za izraz ili (eng. Or) i ! što predstavlja ne (eng. Not). Na primjer u gore navedenom kodu možemo nakon nekaMetoda dodati && `nekaDrugaKlasa -> nekaDrugaMetoda`.

4.3. Advice

Savjet (eng, Advice) predstavlja akciju odnosno programsku logiku koja se treba izvršiti nakon okidanja definirane točke spajanja. Postoji više savjeta koje AOP nudi programerima, a to su savjeti prije (eng. Before), nakon (eng. After) i njihova kombinacija što se onda naziva okolo (eng. Around). Također podržavaju i savjete prilikom generiranja iznimke (eng. After Throwing) te savjet nakon što metoda vrati neku vrijednosti (eng. After returning) Po svojoj strukturi savjeti izgledaju kao najobičniji *Java* kod prilikom pisanja neke metode kako ćemo vidjeti u sljedećim poglavljima. (Vaidya, 2019) (Laukkanen, 2008) (I. Bergman, C. Goransson, 2004)



Slika 6. Korištenje aspekata (dohvaćeno 15.08.2019. sa: https://stackoverflow.com/questions/15447397/spring-aop-whats-the-difference-between-joinpoint-and-pointcut)

Na slici iznad možemo vidjeti kako to izgleda u praksi. Za vrijeme izvršavanja programa postoji više spojnih točaka u koje možemo implementirati odnosno koristiti aspekte. Točka spajanja tada predstavlja mjesto koje smo odredili kao dobrog kandidata za njegovo uvođenje i tada pomoću savjeta dodajemo logiku koja će se izvršiti nad tom točkom.

4.3.1. Before

Primjer korištenja savjeta prije možemo vidjeti na sljedećem kodu.

```
public class PrijeMetode implements MethodBeforeAdvice
{
	@Override
	public void before(Method method, Object[] args, Object target)
		throws Throwable {
		  System.out.println("Prije metode!");
	}
}
```

Ovo je najjednostavniji dio nakon čega ćemo u XML datoteci definirati nad kojom klasom se ovaj kod treba izvršiti. Uzmimo za primjer da imamo klasu pod nazivom KlasaZaTestiranje sa svojim vrijednostima i metodama, tada bi ju definirali na sljedeći način u XML-u. (Mkyong, 2010)

```
<bean id="KlasaZaTestiranjeProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
		<property name="target" ref="klasaZaTestiranje" />
		<property name="interceptorNames">
```

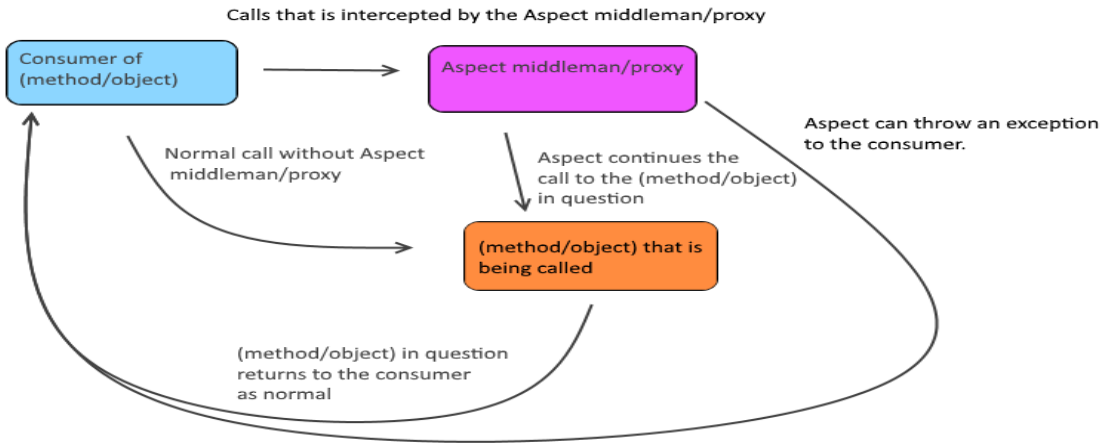
```
            <list>
                  <value> PrijeMetode </value>
            </list>
      </property>
</bean>
```

Sada je definirano da se before metoda klase PrijeMetode izvrši svaki put kada pokrenemo neku metodu klase KlasaZaTestiranje. Na primjer (Mkyong, 2010):

```
KlasaZaTestiranje kzt = new KlasaZaTestiranje();
kzt.ispisiVrijednostiPolja();
```

Gore napisan kod bi dao sljedeći rezultat u konzoli. (Widinghoff, 2015)

```
Prije metode!
Rezultat poziva ispisiVrijednostiPolja();
```



Slika 7. Dijagram savjeta prije (Widinghoff, 2015)

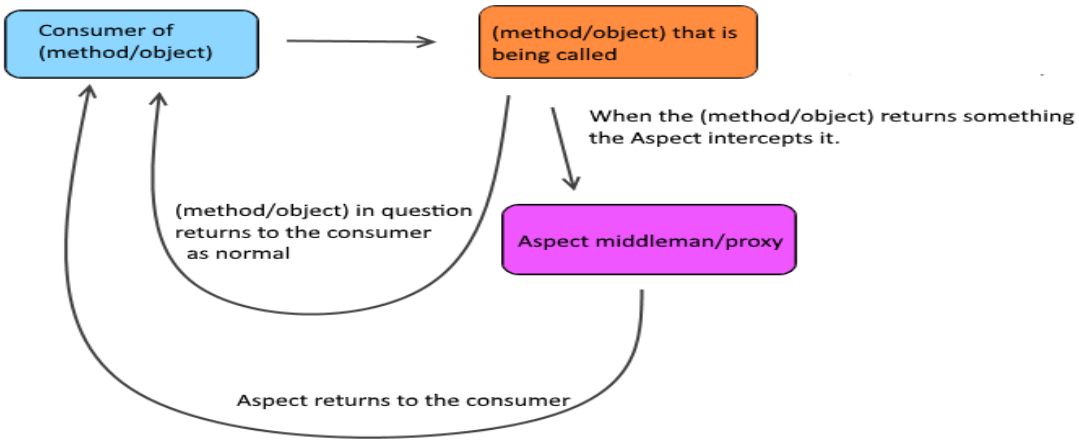
4.3.2. After

Kod korištenja savjeta nakon, većina koda i konfiguracija u XML-u bile bi iste, međutim klasa bi morali implementirati *AfterAdvice interface* i onda nadjačati (eng. Override) metodu after. Kada bi onda koristili gore navedeni primjer dobili bi sljedeći rezultat.

```
Rezultat poziva ispisiVrijednostiPolja();
Nakon metode!
```

Ovakva vrsta savjeta izvršit će se bez obzira da li metoda koju smo definirali baci iznimku ili ne. Slično ovome postoji i *After Returning Advice* koji se izvrši samo ako je poziv metode uspješan, odnosno ako metoda ne baci iznimku. Isto tako postoji i *After Throwing*

Advice. Takva vrsta savjeta izvršava se samo u slučaju ako metoda baci iznimku. Razlike u kodu za postizanje takvih vrsta savjeta su iste kao i razlike kod savjeta prije i poslije. Točnije, treba promijeniti malo strukturu XML datoteke te promijeniti neke ključne riječi. (Mkyong, 2010)



Slika 8. Dijagram savjeta nakon (Widinghoff, 2015)

4.3.3. Around

Jedan od najjačih i najkorištenijih savjeta u aspektno orijentiranom programiranju je savjet okolo. Njegove mogućnosti kombiniraju sve gore navedeno s dodatnim mogućnostima. Savjet okolo ima mogućnost, ako je potrebno i definirano, spriječiti izvršavanje metode. Primjer svega navedenog možemo vidjeti na kodu ispod. (Widinghoff, 2015) (Mkyong, 2010)

```
public class OkoloMetode implements MethodInterceptor {

	@Override
	public Object invoke(MethodInvocation methodInvocation) throws
Throwable {

		System.out.println("naziv metode : "
				+ methodInvocation.getMethod().getName());


		// Ovaj dio je isti kao kod savjeta prije
		System.out.println("Prije metode!");

		try {
			// izvršavanje metode
			Object result = methodInvocation.proceed();

			// Isto kao savjet nakon
			System.out.println("Nakon metode!");

			return result;
```

```
                // Isto kao savjet nakon vraćanja
                System.out.println("Nakon vraćanja rezultata!");

            } catch (Exception e) {
                // Ovako bi se ponašao savjet nakon bacanja iznimke
                System.out.println("Iznimka!");
                throw e;
            }
        }
}
```

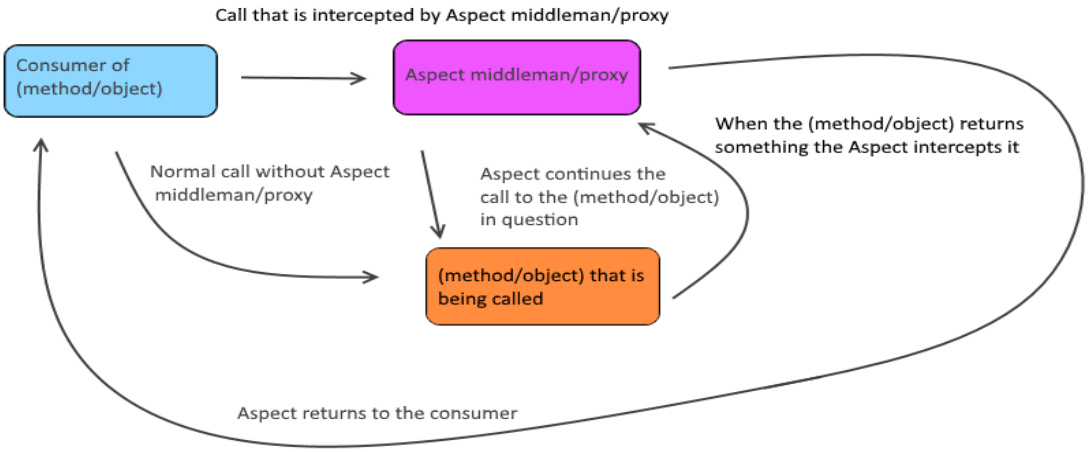
Vidimo da smo u ovom kodu morali implementirati *MethodInterceptor* sučelje te nadjačati metodu *invoke*. Unutra metode kod je sličan ostalim savjetima, ali kao što sam spomenuo ovdje imamo više mogućnosti za upravljanje ponašanja metode oko koje se ovaj savjet koristi.

XML konfiguracija bi izgledala više-manje isto kao kod prije objašnjenog savjeta prije gdje smo definirali točku spajanja. Kada bi se ovaj kod izvršio nad istom takvom klasom, odnosno metodom, dobili bi sljedeći ispis u konzoli.

```
Naziv metode: ispisiVrijednostiPolja
Prije metode!
Rezultat poziva ispisiVrijednostiPolja();
Nakon metode!
Nakon vraćanja rezultata!
```

Ako bi se dogodilo da metoda baci iznimku rezultat bi bio malo drugačiji. Ne bi postojao ispis *Nakon vraćanja rezultata!* već bi prije kraja izvršavanja pisalo *Iznimka!*. Bacanje iznimke možemo postići na jednostavan način da unutar metode koristimo ključnu riječ *throw* i unutra definiramo vrstu iznimke koju želimo baciti tijekom izvršavanja. Korištena je sljedeća linija koda: *throw new RuntimeException();*. Rezultat ispisa je tada sljedeći. (Widinghoff, 2015)

```
Naziv metode: ispisiVrijednostiPolja
Prije metode!
Rezultat poziva ispisiVrijednostiPolja();
Iznimka!

Nakon metode!
```

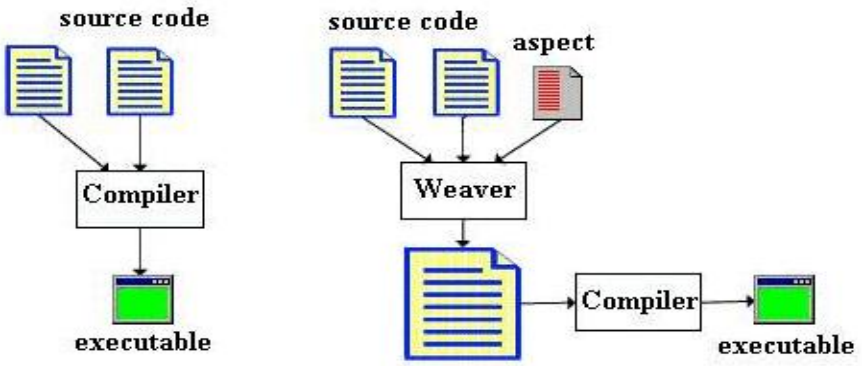


Slika 9. Dijagram savjeta okolo (Widinghoff, 2015)

4.4. Weaving

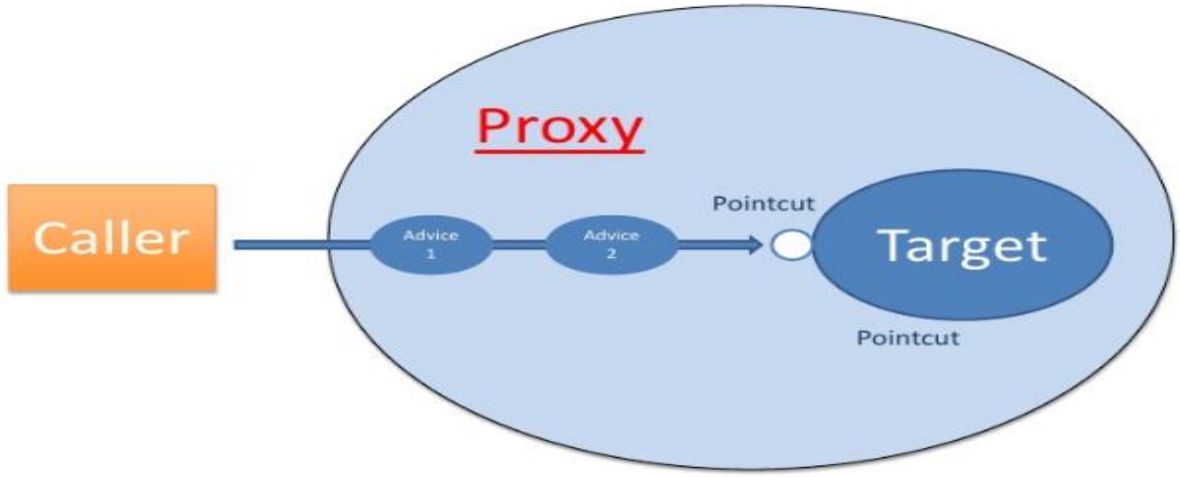
Kada pričamo o aspektno orijentiranom programiranju pričamo o paradigmi koja je, u odnosno na OOP, novija u svijetu programiranja. Mnogo jezika poput *Java*-e napisano je puno prije nego li je koncept AOP prvi put spomenut pa je zanimljivo na koji način nam to jezici omogućuju pisanje aspekata u nečemu što je nastalo toliko prije njih. Upravo iz tog razloga nastao je takozvani tkalac (eng. *Weaver)* koji omogućuje sastavljanje (eng. Compile) napisanog koda koji sadržava aspekte. Ako ne bi postojao takav koncept svaki programski jezik koji sadrži aspekte trebao bi se raditi iznova kako bi omogućio njihovo korištenje. On zapravo predstavlja proširenje kompajlera za prihvaćanje novih tehnologija. (Laukkanen, 2008)

Na slikama ispod možemo vidjeti kako to izgleda. Lijeva slika predstavlja pretvaranje izvornog koda (eng. Source Code) u izvršnu datoteku (eng. Executable) preko kompajlera dok na slici desno možemo vidjeti kako tkalac uzima sav kod koji sadrži i aspekte te ga pretvara u izvorni kod koji kompajler može razumjeti kako bi generirao izvršnu datoteku. (Laukkanen, 2008)



Slika 10. Aspekt weaver (Yang, n.d.)

Slika prikazuje pretvaranje koda tijekom izvršavanja (eng. Runtime), međutim to nije jedini način korištenja *weavinga*. Postoje takozvani statični (eng. Static) i dinamički (eng. Dynamic) tkalac. Njihova razlika je poprilično jednostavna. Statički znači da se prilikom kompajliranja generira sav izvorni kod i najpoznatiji primjer takvog korištenja je *AspectJ* okvir. Takav pristup ima svoje mane jer kada treba otkloniti neispravnosti (eng. Debug) tkalac generira izvorni kod zajedno s aspektima. Tada se pokretanjem izvršne datoteke ne zna pripada li određeni dio koda aspektu ili najobičnijoj komponenti. Kod dinamičkog je situacija obrnuta zato što se kod pretvara tijekom izvršavanja aplikacije. *Spring* okvir predstavlja najpoznatiji pristup takvom pretvaranju, međutim i on ima svoje mane. Iako ovaj način povećava fleksibilnost aplikacije i omogućuje lakše uklanjanje grešaka, dosta usporava aplikaciju. Razlog usporavanju je konstantno provjeravanje i pretvaranje aspekata u novi kod korištenjem takozvanih *proxy* klasa koje sadrže istu funkcionalnost ali s ubačenom logikom aspekata. Primjer možemo vidjeti na slici ispod. (Yang, n.d.)



Slika 11. Dinamički weaving (dohvaćeno 20.08.2019. sa: https://www.slideshare.net/rohitsghatol/aspect-oriented-prog-with-aspectj-spring-aop)

Kako bi sve funkcioniralo kako treba aspektni tkalci moraju biti u mogućnosti pročitati i generirati dobar izvorni kod nad svakom točkom spajanja unutar aplikacije jer se u suprotnom program neće moći izvršiti.

5. Različiti okviri za AOP

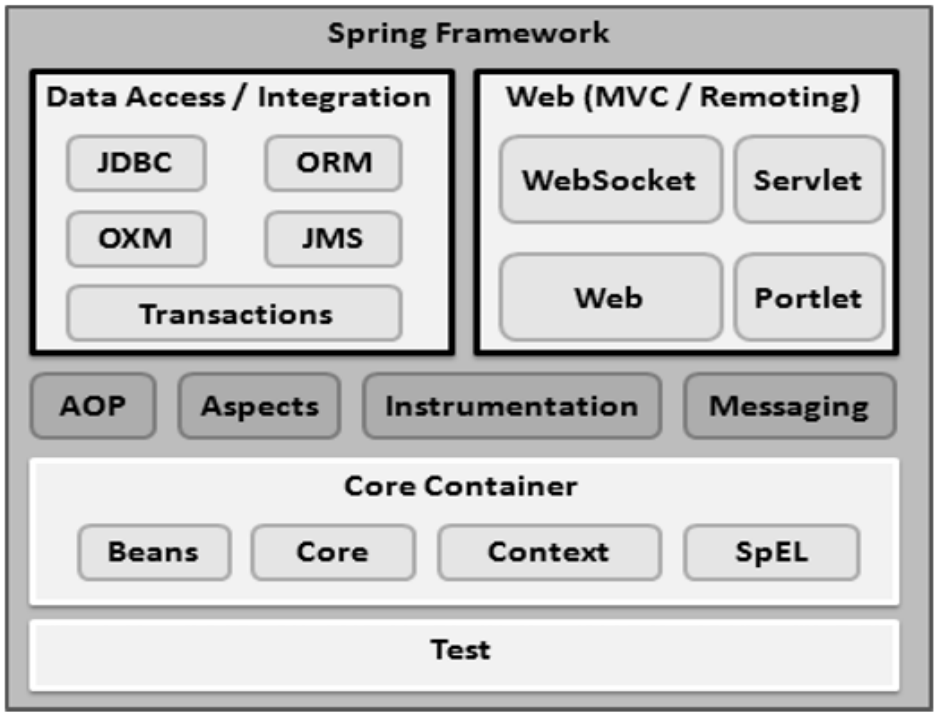
U ovome poglavlju pričat ćemo o primjeni različitih okvira za implementaciju aspektno orijentiranog programiranja. Već smo spomenuli da se AOP koristi kao dodatak na OOP u smislu proširivanja njegovih mogućnosti. Korištenjem samih aspekata nije moguće izraditi funkcionalnu aplikaciju jer se on koristi kako bi riješio već spomenute *cross cutting concers*-e unutar aplikacije koja koristi OOP. Najpoznatiji OOP jezici su *Java*, *C#, C++* i slični što ih čini i najpoznatijim jezicima za upotrebu aspekata u svome radu. U *Java* programskom jeziku najpopularniji okviri za korištenje aspekata su ujedno i najbolji, a to su *Spring* i *AspecJ*. Kada pričamo u *C++* najpopularniji okvir je *AspectC++* dok za *C#* koji koristi *.NET* postoje okviri poput *PostSharp*-a i *NConcern*-a čija je implementacija malo izazovnija od ostalih. Najčešći pristup korištenja AOP u *C#* je ipak pomoću *RealProxy* klase. Manje popularniji objektno orijentirani jezici poput PHP-a također imaju svoj okvir za korištenje aspekata pod nazivom Flow framework, no više o svemu tome u nastavku. (Tereshchenko, 2017)

5.1. Spring framewoork

Spring predstavlja najpopularniji okvir za razvijanje *Java* aplikacije. To je *open source* okvir kojeg prema njihovoj službenoj stranici koriste milioni programera diljem svijeta kako bi na jednostavan način mogli razviti visoko kvalitetan kod kojeg je lako održavati i ponovno koristiti. Nastao je u lipnju 2003. godine, a napisao ga je Rod Johnson. Nama najvažniji dio ovoga okvira je njegova snaga u korištenju aspekata, međutim nećemo zanemariti njegove ostale mogućnosti, odnosno prednosti njegova korištenja. Neke od tih prednosti su sljedeće:

- Omogućuje razvijanje aplikacije korištenjem *plain old java object* (kraće POJO) klasa. POJO klasa je klasa bez posebne programske logike.
- Modularno je organiziran što uvelike olakšava rad i snalaženje u njegovim paketima.
- Koristi dobre postojeće tehnologije poput *ORM* okvira, *JEE* i slično.
- Rješava problem umetanja odgovornosti (eng. Dependency Injection, kraće DI).
- Aspektno orijentirano programiranje

DI predstavlja način prenošenja parametara putem konstruktora ili *settera* kako bi se izbjeglo čvrsto povezivanje objekata tako da se unutar jedne klase ne koristi ključna riječ *new* za kreiranje nekog drugog objekta. To znači da jedna klasa ne mora ovisiti o drugoj za rad. (Pankaj, n.d.) (Tutorialspoint, Spring Framework - Overview, n.d.) (Tutorialspoint, AOP with Spring Framework)



Slika 12. Arhitektura Spring okvira

Na slici iznad vidimo arhitekturu Spring okvira koja se sastoji od više modula. Ti moduli posloženi su u takozvane kontejnere (eng. Container) ili slojeve (eng. Layer). Iako nam je za razvoj aplikacije dostupno oko dvadesetak modula, nećemo ih uvijek sve koristiti. Takozvani *Core Container* sastoji se od sljedećih modula:

- *Core* modul koji pruža osnovne funkcionalnosti Spring okvira poput prije spomenutog umetanja ovisnosni.
- *Bean*[1] modul koji omogućuje olakšanu implementaciju *Factory* uzorka dizajna pomoću *BeanFactory*-a.

[1] Objekt koji se može serijalizirati i koji ima javan konstruktor bez argumenata i sadrži getter, setter metode.

- *Context* modula koji se nadovezuje na Core i Bean module i služi kao posrednik (eng. Mediator) pristupanju i konfiguraciji pojedinih objekata.
- *SpEL* modula koji omogućuje manipuliranje objektima i upita nad bazom tokom izvođenja aplikacije

Web sloj sastoji se od Web modula za integraciju osnovnih funkcionalnosti web aplikacije, Web-MVC modula koji sadrži *Spring*-ov *Model-View-Controller* (kraće MVC) za razvoj Web aplikacija, Web-Socket modula koji nudi podršku za dvosmjernu komunikaciju između klijenta (eng. Client) i servera (eng. Server), te Web-Portlet modula koji omogućuje korištenje MVC-a u *portlet* [2]okruženju.

Također se sastoji od raznoraznih ili *Miscellaneous* modula u koje spadaju nama bitni AOP modul i aspekt modul kako možemo vidjeti i na slici.

5.1.1. Implementacija aspekata

Već smo više puta spomenuli da postoji više načina implementacije aspekata unutar aplikacije i u ovome pod poglavlju ćemo ih prikazati. Ovaj okvir nudi već neke pripremljene aspekte koje možemo koristiti ali u ovome radu fokusirat ćemo se na rad s našim prilagođenim (eng. Custom) aspektima. *Spring* nudi dva pristupa implementacije naših aspekata. Jedan je putem XML sheme, a drugi je putem anotacija. Prikazat ćemo kako se kreiraju i implementiraju aspekti dok ćemo funkcionalne primjere prikazati u poglavlju praktičnog rada. (Pankaj, n.d.) (Baeldung, Introduction to Pointcut, 2017)

5.1.2. Putem XML sheme

Prvo što je potrebno napraviti za korištenje aspekata na ovaj način je ubaciti (eng. Import) springAOP shemu na sljedeći način.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop = "http://www.springframework.org/schema/aop"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

   <!-- bean definition & AOP specific configuration -->

</beans>
```

[2] Generiranje dinamičkog sadržaja za prikaz na ekranu nakon zahtjeva prema serveru od strane klijenta.

Prilikom deklaracije samog aspekta koriste se *<aop:aspect>* element u XML-u, kako možemo vidjeti u kodu ispod.

```
<aop:config>
   <aop:aspect id = "mojAspect" ref = "nekiBean">
      ...
   </aop:aspect>
</aop:config>

<bean id = "nekiBean" class = "...">
   ...
</bean>
```

U kodu možemo vidjeti da se navedeni aspekt referencira na pojedini *bean* s ključnom riječi *ref* nakon kojeg slijedi naziv *bean*-a kojeg želimo referencirati. U ovome slučaju to je klasa pod nazivom *nekiBean*. Prije smo također spominjali pojam ubacivanja ovisnosti i to se rješava tako da se definira i *bean* sa svojim id-om kako bi ga *Spring* znao instancirati bez kreiranja ovisnosti. Svakom *bean*-u zadani (eng. Default) *scope* u kojemu se kreira je singleton[3].

Kod definiranja točke spajanja unutar postojećeg aspekta dodamo XML element koji ga definira i tada kod izgleda ovako:

```
<aop:config>
   <aop:aspect id = "mojAspect" ref = "nekiBean">
      <aop:pointcut id = "nekiServis"
         expression = "execution(*hr.foi.myapp.service.*.*(..))"/>
         ...
   </aop:aspect>
</aop:config>
```

U ovome slučaju ovaj aspekt će se aktivirati nad svakom metodom u klasama unutar aplikacije koje se nalaze u paketu hr.foi.myapp.service. Naravno to možemo promijeniti tako da umjesto takvog izraza napišemo hr.foi.myapp.service.Auto.getVrsta(…). Tada će se aspekt aktivirati samo unutar klase Auto nad metodom *getVrsta()*.

Nakon definiranja točke spajanja sljedeći korak bio bi definirane savjeta. Kako smo rekli, postoji više vrsta savjeta i oni se definiraju tako da u postojeći aspekt ubacimo određeni XML element koji bi izgledao ovako (TutorialPoint, XML Schema Based AOP with Spring, n.d.):

```
<aop:{before, after, after-returning, after-throwing, around} pointcut-ref =
"nekiServis" method = "izvrši" >
```

[3] Objekt koji se instancira samo jednom i koristi unutar rada aplikacije.

5.1.3. Putem anotacija

Do sada smo doznali da je *Spring* okvir za rad jako opširan i aspekti mu nisu jedina mogućnost. Za svoj rad pomoću anotacija čak i *Spring* koristi pomoć *AspectJ*-a kako bi realizirao aspekte na najbrži i najjednostavniji način.

Anotacija @AspectJ je zapravo stil pisanja prilikom deklaracije aspekata kao *Java* klasa koje spadaju pod mogućnostima Jave 5 i dalje. Iako se koristi pomoć *AspectJ* okvira, izvođenje se bazira na čistom Spring-u i nema veze s AspectJ-om. Više o sličnostima i razlikama bit će prikazano u sljedećim poglavljima.

Prva stvar koju je potrebno napraviti kako bi omogućili pisanje aspekata putem anotacije je u konfiguracijskom datoteci ubaciti sljedeći XML element:

```
<aop:aspectj-autoproxy/>
```

Nakon toga također je potrebno ubaciti neke od biblioteka (eng. Libraries) koje se mogu jednostavno pronaći na internetu, a zovu se:

- aspectjrt.jar
- aspectjweaver.jar
- aspectj.jar
- aopalliance.jar

Kada se završi s konfiguracijskim dijelom možemo početi s definiranjem aspekata. Anotacija @Aspect predstavlja da je klasa zapravo aspekt. Primjer takve klase možemo vidjeti u kodu ispod.

```
@Aspect
public class AspektModul {
}
```

Vidimo da je ovaj način puno pregledniji i zanimljiviji način pisanja od gore prikazanog XML-a. Na sličan način se radi i točka spajanja, samo umjesto anotacije @Aspect koristimo anotaciju @Pointcut iznad metode nad kojom ga želimo postaviti.

```
@Pointcut("execution(* hr.foi.myapp.service.*.*(..))")
private void nekiServis() {}
```

Isto tako i kada pišemo savjete samo moramo pripaziti na tip savjeta kojeg želimo implementirati. Sintaksa je sljedeća:

`@{Before, After, AfterReturning, AfterThrowing, Around}` nakon čega slijedi metoda koju želimo anotirati. (Baeldung, Introduction to Pointcut, 2017) (Spring, n.d.) (JBossDeveloper, n.d.)

5.2. AspectJ framework

AspectJ službeno je objavljen 2001. godine kao ekstenzija *Java* programskom jeziku za aspektno orijentirano programiranje. Popularnost mu je tokom godina znatno rasla jer je predstavljen kao Java jezik s proširenim mogućnostima. Ta proširenja su zapravo nove ključne riječi (eng. Key Word) kojima su se označavali i implementirali aspekti. Sada je jedan on najpoznatijih i najpopularnijih okvira za AOP u Java-i i predstavlja temelj ostalim AOP okvirima. Jako dobro se i prilagodio *Spring* okviru što je programerima uvelike olakšavalo pisanje i razumijevanje koda.

Sastoji se od dva dijela. Prvo dio je specifikacija jezika (eng. Language specification) koja definira gramatiku i semantiku te drugog dijela, a drugi dio je implementacija jezika (eng. Language implementation) koji se bavi pretvaranjem koda kako bi ga kompajler mogao razumjeti, drugim riječima bavi se *weaving*-om. U *AspectJ*-u postoje tri vrste *weaving*-a:

- Source weaving
- Binary weaving
- Load-time weaving

Kod prvog načina tkalac radi na sličan način kao kompajler i to je način koji je smo već prošli i objasnili. Ukratko, procesuira izvorni kod kako bi dobio bitovni kod (eng. Byte Code) koji odgovara *Java* specifikaciji i razumljiv je svakoj virtualnoj mašini (eng. Virtual Machine).

Binary weaving funkcionira tako da su svi napisani aspekti koje tkalac treba obraditi napisani u bitovnom kodu. Upravo zbog toga se taj kod onda mora kompajlirati s drugim kompajlerom, najčešće *Java* ili *AspectJ*.

Zadnji način weaving-a radi tako da prima napisane klase i aspekte u obliku binarnih (eng. Binary) podataka. Za razliku od ostalih on transformira klase prilikom njihovog učitavanja u virtualnu mašinu. (Baeldung, Intro to AspectJ, 2019) (Marketing, n.d.)

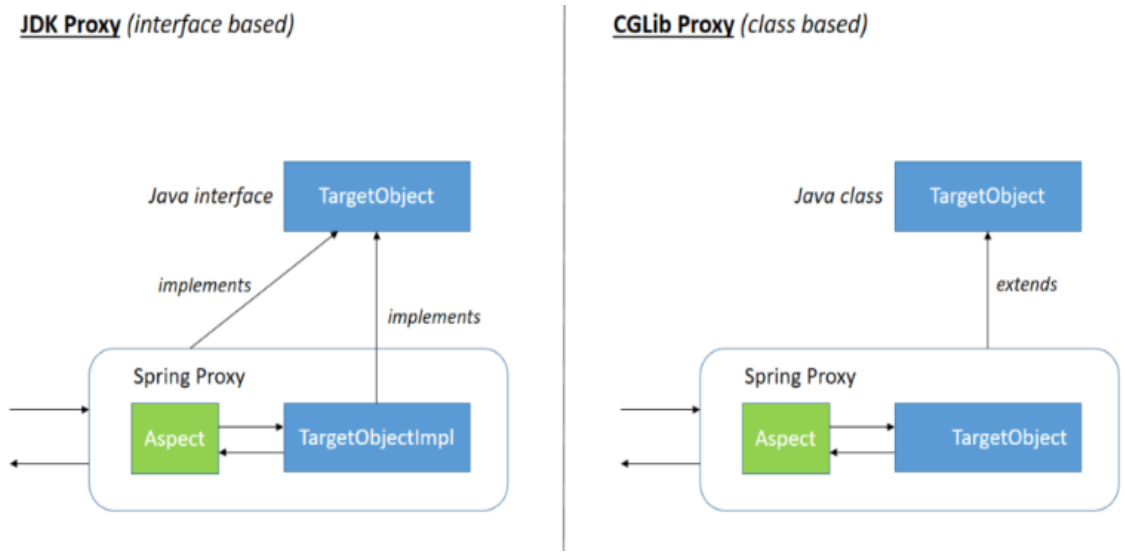
Implementacija kod ovog okvira može se pisati na isti način koji je objašnjen kod *Spring* okvira jer kako smo već rekli, napisan je da odgovara *Spring*-u i da mu proširi mogućnosti. Te mogućnosti moći ćemo vidjeti u praktičnom dijelu ovoga rada.

5.3. Razlike Spring AOP i AspectJ

U ovom odlomku usporedit ćemo dva najpoznatija okvira za pisanje aspekata u Java programskom jeziku, a to su *Spring* i *AspectJ*.

Prva bitna razlika ova dva okvira su ciljevi koje njihovom uporabom želimo ostvariti. Kod Spring-a AOP je napisano kako bi proširilo elemente OOP i tako olakšalo programerima pisanje koda. Također može se primijeniti samo kod onih *bean*-ova kojima upravlja *Spring* kontejner (eng. Container). Za razliku od njega, AspectJ je mnogo opširniji jer pruža više mogućnosti, što ga čini i malo kompliciranijim za rad. On nije ograničen na *Spring* kontejnere, odnosno može se koristiti duž cijele domene (eng. Domain) aplikacije.

Druga razlika je *weaving*. Već smo spomenuli da se AspectJ bazira na tri vrste *weaving*-a, dok se Spring koristi samo *weaving*-om tijekom izvršavanja korištenjem JDK Proxy-a ili CGLib Proxy-a.



Slika 13. Spring runtime weaving (Baeldung, Comparing Spring AOP and AspectJ, 2017)

Na slici vidimo dva načina transformacije, odnosno *woving*-a kod *Spring* okvira. Vidimo da je njihova razlika samo u tome da li objekt implementira sučelje (eng. Interface) ili nasljeđuje klasu. Kako smo već spomenuli da je *weaving* tijekom izvršavanja dosta sporiji od ostalih načina, možemo zaključiti da je *AspectJ* podosta brži kod transformacije koda od *Spring*-a.

Na sljedećoj tablici možemo vidjeti glavne razlike koje smo objasnili u ovome radu.

Tablica 1. Razlike Spring AOP i AspectJ

Spring AOP	AspectJ
Implementiran u Javi	Implementiran kao ekstenzija Java programskog jezika
Dovoljan jedan kompajler	Treba AspectJ kompajler
Omogućuje samo weaving tokom izvršavanja	Omogućuje tri različita načina weavinga
Podupire samo weaving na razini metoda	Može weavati polja, metoda, konstruktore itd.
Korištenje samo na beanovima kojima upravlja Spring kontejner	Podupire implementaciju nad cijelom domenom projekta
Dosta sporiji	Brža i bolja performansa
Lakši za učenje	Dosta kompliciraniji

Treba opet spomenuti da su ova dva okvira kompatibilna te da ih je prilikom pisanja aplikacije najbolje koristiti zajedno kako bi iskoristili puni potencijal aspektno orijentiranog programiranja. (Baeldung, Comparing Spring AOP and AspectJ, 2017)

5.4. AspectC++ framework

AspectC++ napisan je kako bi olakšao pisanje aspekata u C++ programskom jeziku. Kako je C++ jezik niže razine trebalo je napraviti takav okvir koji može zadržati njegovu brzinu prilikom izvođenja i da istovremeno programerima olakša i ubrza pisanje koda. (Spinczyk, 2005)

5.4.1. Implementacija aspekata

Kreiranje aspekta radi se s ključnom riječi a*spect* umjesto tipične *class* na sljedeći način.

```
aspect nekaKlasa{}
```

Isto kao i kod klase možemo definirati konstruktore, metode, polja i slične elemente.

Savjeti se definiraju na ključnom riječi *advice* i podržava tri vrste savjeta: prije, poslije i okolo. Prilikom definiranje savjeta treba definirati i točku spajanja na koju se savjet odnosi. Primjer jednog savjeta napisanog u AspectC++ okviru možemo vidjeti u kodu ispod.

```
advice execution("% nekiPaket::NekaKlasa::nekaMetoda(...)") : after() {
printf("nakon metode!" );
}
```

Prikazan je savjet s ključnom riječi *execution* nakon kojeg slijedi točka spajanja što označava da se napisani savjet izvršava nakon implementacije funkcija napisanih unutar navedene točke spajanja. Uz *execution*, postoje i ostale funkcije poput *call* koja označava

pozivanje metode navedene točkom spajanja. Također postoje construction i destruction što označava kreiranje ili uništavanje metoda ili klasa navedenih u točki spajanja. Navedeni izrazi koriste se za metode i klase međutim postoje i izrazi kao *cflow* koji izdvaja spojne točke koje su navedene u točki spajanja, ili *base* i *derived* koji se odnose na bazne i izvedene klase. Uz to postoji i *within* koje označava područje primjene te *that* i *target* koje se odnose za kontekst. (Spinczyk, 2005)

6. Praktični primjer

U ovom dijelu rada prikazat ćemo aplikaciju koja je razvijena kako bi mogli na praktičnom primjeru pokazati što smo do sada naučili. Aplikacije se zove *BookRegistry* i kao što sam naziv govori radi se o jednostavnoj aplikaciji preko koje korisnik upravlja knjigama iz baze podataka. Razvijena ja u *Intellij* okviru korištenjem *Java*-e kao programski jezik za server te JavaScript i HTML za korisničko sučelje (eng. User Interface). Kao što je već spomenuto korišten je *Spring* okvir za realizaciju pozadinskog (eng. Backend) dijela aplikacije i *Angular* okvir za realizaciju sučelja (eng. Frontend). Točnije korišten je *Spring Boot* što je zapravo pojednostavljena verzija *Spring*-a s kojom se brže može razviti aplikacija jer je sva konfiguracija unaprijed pripremljena. Aplikacija koristi četiri različita aspekta napisana uz pomoć *AspectJ* okvira unutar *Spring*-a te jedan aspekt koji je već unaprijed pripremljen, odnosno podržan od strane okvira. Kroz kod i sučelje objasnit ćemo na koji način ti aspekti rade te kako su napravljeni. Kada se aplikacija pokrene prvo što se dogodi je da se u bazu podataka unesu knjige koje se sastoje od naslova, autora, i količine. To možemo vidjeti na slici ispod.

```
@EventListener(ApplicationReadyEvent.class)
public void addBooks() {
    List<Book> bookSet = new ArrayList<>();
    bookSet.add(new Book( title: "Harry Potter and the Sorcerer's Stone",  quantity: 5,  author: "J. K. Rowling"));
    bookSet.add(new Book( title: "The Hobbit",  quantity: 5,  author: "J. R. R. Tolkien"));
    bookSet.add(new Book( title: "1984",  quantity: 5,  author: " George Orwell"));
    bookSet.add(new Book( title: "Pride and Prejudice",  quantity: 5,  author: "Jane Austen"));
    bookSet.add(new Book( title: "To Kill a Mockingbird",  quantity: 5,  author: "Harper Lee"));
    bookSet.add(new Book( title: "The Da Vinci Code",  quantity: 5,  author: "Dan Brown"));
    bookSet.add(new Book( title: "The Catcher in the Rye",  quantity: 5,  author: "J. D. Salinger"));
    bookSet.add(new Book( title: "The Great Gatsby",  quantity: 5,  author: "F. Scott Fitzgerald"));
    bookSet.add(new Book( title: "Twilight",  quantity: 5,  author: "Stephenie Meyer"));
    bookSet.add(new Book( title: "The Hunger Games",  quantity: 5,  author: "Suzanne Collins"));
    bookSet.add(new Book( title: "Jane Eyre",  quantity: 5,  author: "Charlotte Brontë"));
    bookSet.add(new Book( title: "Animal Farm",  quantity: 5,  author: " George Orwell"));
    bookSet.add(new Book( title: "The Kite Runner",  quantity: 5,  author: "Khaled Hosseini"));
    bookSet.add(new Book( title: "Brave New World",  quantity: 5,  author: "Aldous Huxley"));
    bookSet.add(new Book( title: "The Lord of the Rings",  quantity: 5,  author: "J. R. R. Tolkien"));
    bookSet.add(new Book( title: "Fahrenheit 451",  quantity: 5,  author: "Ray Bradbury"));
    bookSet.add(new Book( title: "New Moon",  quantity: 5,  author: "Stephenie Meyer"));
    bookSet.add(new Book( title: "Angels & Demons",  quantity: 5,  author: "Dan Brown"));
    bookSet.add(new Book( title: "Wuthering Heights",  quantity: 5,  author: "Emily Brontë"));
    bookSet.add(new Book( title: "The Odyssey",  quantity: 5,  author: "Homer"));
    bookSet.add(new Book( title: "Life of Pi",  quantity: 5,  author: "Yann Martel"));
    bookSet.add(new Book( title: "Catching Fire",  quantity: 5,  author: "Suzanne Collins"));
    bookSet.add(new Book( title: "Slaughterhouse-Five",  quantity: 5,  author: "Kurt Vonnegut"));
    bookSet.add(new Book( title: "The Time Traveler's Wife",  quantity: 5,  author: "Audrey Niffenegger"));

    bookRepository.saveAll(bookSet);

    userRepository.save( new User( username: "admin", passwordEncoder().encode( charSequence: "admin"),  admin: true));
}
```

Slika 15. Unos knjiga u bazu podataka

Također možemo vidjeti da se kreira i jedan korisnik koji je postavljen kao *admin* što znači da samo on može dodavati nove knjige u bazu podataka.

Objekti koji se koriste kao *model* za knjige i korisnike prikazani su u sljedećim slikama.

```java
@Entity
public class Book {

    @Id
    @Column
    @GeneratedValue(strategy = AUTO)
    private Long id;

    @Column
    private String title;

    @Column
    private Integer quantity;

    @Column
    private String author;

    @ManyToMany(mappedBy = "books")
    private List<User> users;

    public Book(String title, Integer quantity, String author) {
        this.title = title;
        this.author = author;
        this.quantity = Objects.requireNonNullElse(quantity, defaultObj: 1);
    }
```

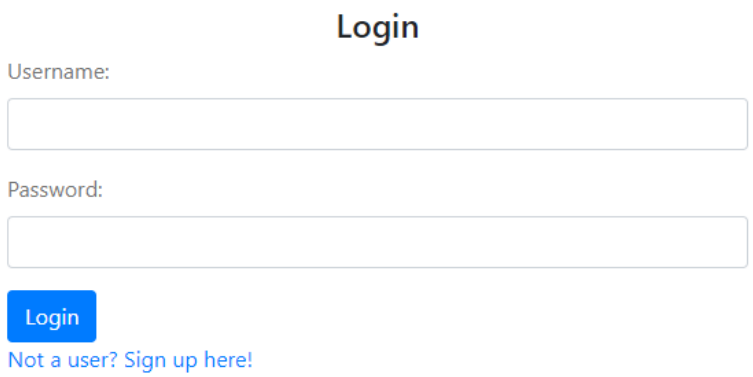
Slika 16. Model klase Book

```java
@Entity
public class User {

    @Id
    @Column
    @GeneratedValue(strategy = AUTO)
    private Long id;

    @Column(unique = true)
    private String username;

    @Column
    private String password;

    @Column
    private boolean admin;

    @ManyToMany
    @JoinTable(
            name = "has_book",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "book_id"))
    private List<Book> books;


    public User(String username, String password, boolean admin) {
        this.username = username;
        this.password = password;
        this.admin = admin;
    }
```

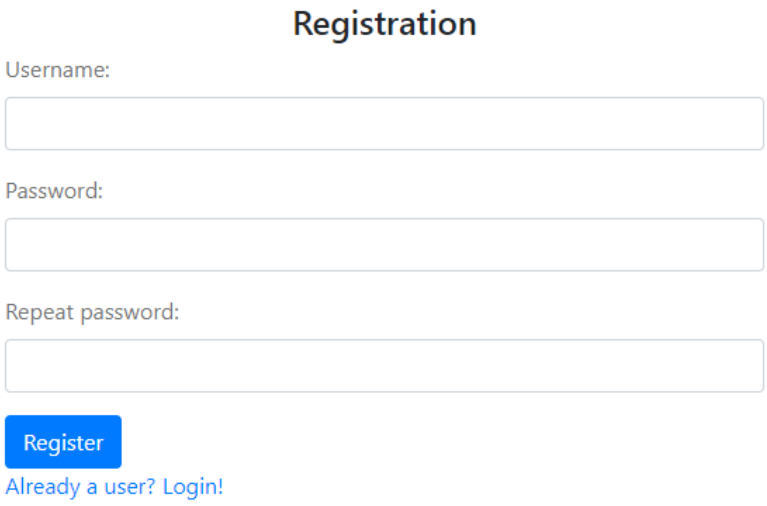
Slika 17. Model klase User

Na slikama možemo vidjeti klase koje su anotirane s ključnom riječju ***@Entity*** koja označava da se ta klasa također zapiše u bazu podataka.

Početni zaslon kada se otvori aplikacija je zaslon za prijavu (eng. Login) koji je uz registraciju (eng. Registration) nezaštićen. To znači da mu mogu pristupiti i neprijavljeni korisnici. Vidjet ćemo u nastavku kako je pomoću aspekata implementirana zaštita takozvanog *Rest API*-a.



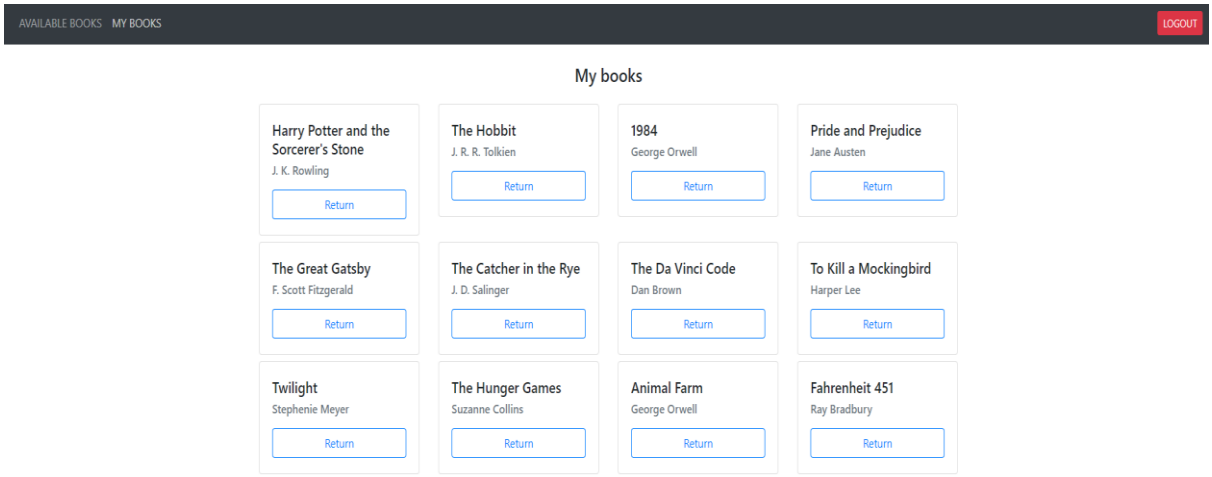
Slika 18. Sučelje za prijavu



Slika 19. Sučelje za registraciju

Iznad su prikazana sučelja koja su napisana u prije spomenutom *Angular* okviru. Oni rade tako da kada korisnik popuni polja podacima i klikne na gumb pošalje se zahtjev aplikaciji koja radi u pozadini i taj zahtjev obradi. Nakon uspješne prijave otvara se sljedeći zaslon na

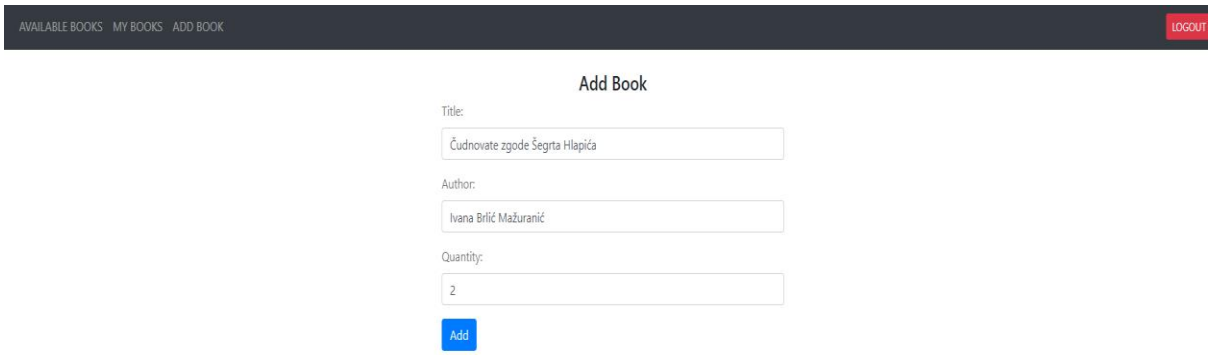
kojem korisnik može vidjeti knjige koje trenutno posjeduje, ali i knjige koje su slobodne za posudbu. Ta dva ekrana poprilično su slična pa ćemo na sljedećoj slici prikazati jedan od njih.



Slika 20. Prikaz korisnikovih knjiga

Nakon prijave prikazuje se gore prikazan ekran i navigacijska traka preko koje korisnik može provjeravati slobodne i svoje knjige. Također vidimo i gumb za odjavu (eng. Logout). Pritiskom na gumb *Return* se vraća i više nije vidljiva na ovom zaslonu.

Ako se prijavi korisnik koji je ujedno i *admin* dobijemo novu poveznicu (eng. Link) u navigacijskoj traci sa sljedećim zaslonom u kojem se mogu dodavati nove knjige.



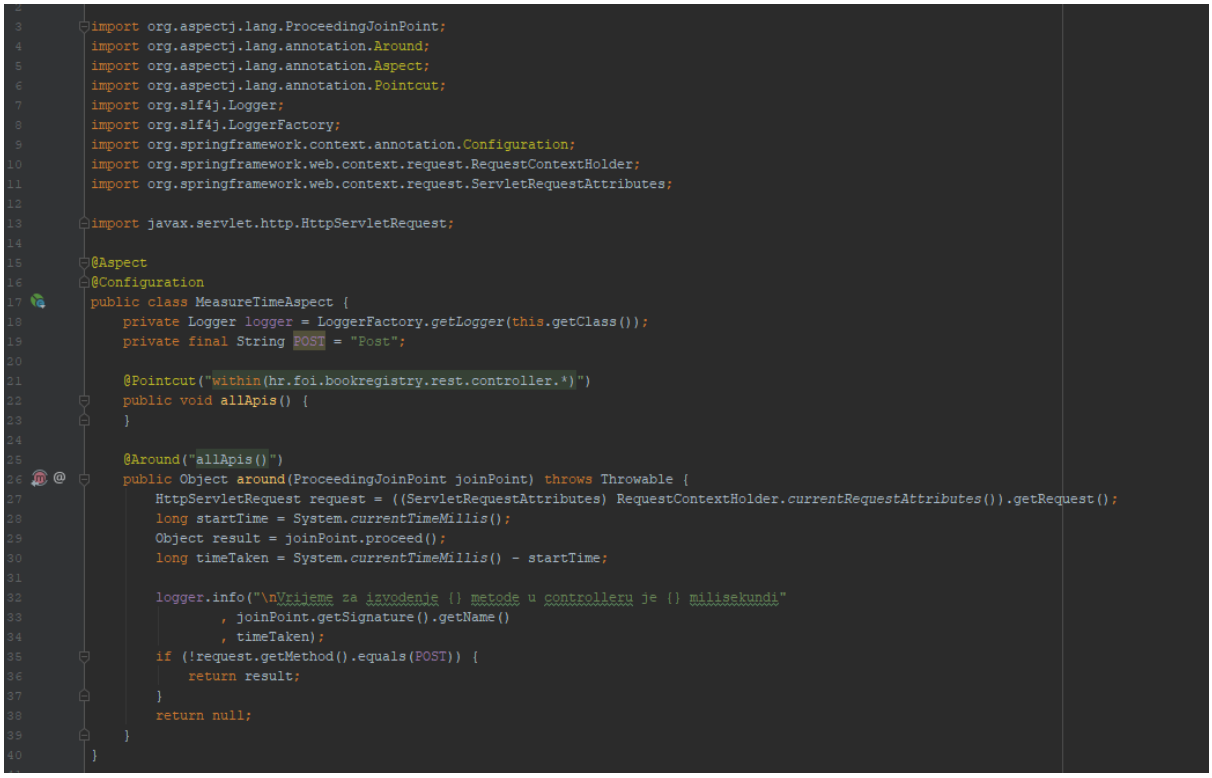
Slika 21. Dodavanje knjige

6.1. Aspekti

Kako je već spomenuto aplikacija koristi pet različitih aspekata za različite funkcionalnosti. Već nam je poznato da se najčešći aspekti koriste u svrhu zapisa, transakcija, i sigurnosti pa u nastavku možemo vidjeti kako su oni rađeni.

6.1.1. Mjerenje performansi

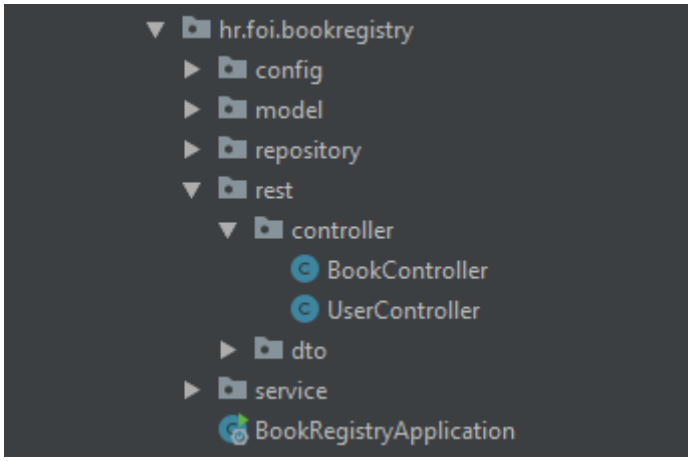
Ovaj aspekt je napravljen kako bi mogli pratiti vrijeme izvršavanje svakog poziva metoda prilikom spajanja klijenta na server, odnosno prilikom svakog *Rest* poziva.



```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.context.request.RequestContextHolder;
import org.springframework.web.context.request.ServletRequestAttributes;

import javax.servlet.http.HttpServletRequest;

@Aspect
@Configuration
public class MeasureTimeAspect {
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    private final String POST = "Post";

    @Pointcut("within(hr.foi.bookregistry.rest.controller.*)")
    public void allApis() {
    }

    @Around("allApis()")
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
        HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolder.currentRequestAttributes()).getRequest();
        long startTime = System.currentTimeMillis();
        Object result = joinPoint.proceed();
        long timeTaken = System.currentTimeMillis() - startTime;

        logger.info("\nVrijeme za izvodenje {} metode u controlleru je {} milisekundi"
                , joinPoint.getSignature().getName()
                , timeTaken);
        if (!request.getMethod().equals(POST)) {
            return result;
        }
        return null;
    }
}
```

Slika 22. Aspekt za mjerenje performansi

Vidimo najobičniju klasu koja je anotirana na ključnom riječju *@Aspect*. Točka spajanja definirana je s ključnom riječju *@Pointcut* nakon koje možemo vidjeti na kojim mjestima želimo da se ovaj aspekt aktivira. U ovom slučaju to su sve metode unutar paketa u kojem se nalaze servisi za primanje i obradu zahtjeva. Na slici također vidimo da se koristi i savjet okolo kako bi mogli započeti mjerenje prije poziva metode te izračunati razliku nakon poziva. Metoda *proceed()* od klase *ProceedingJoinPoint* omogućuje nam pravi početak izvršavanja metode u navedenim klasama. Zamislite sada da smo ovaj kod imali u svakoj od metoda unutar specificiranih klasa. Na ovaj način podijelili smo odgovornosti, a metode i dalje rade samo ono za što su napisane. Namjerno su ostavljeni i paketi koji su ubačeni kako bi vidjeli da se koristi *AspectJ* prilikom definiranja aspekata.



Slika 23. Prikaz Controller paketa

Na slici vidimo tu putanju koja je definirana u točki spajanja. Spojne točke u ovom primjeru su sve metode unutar klasa koje se nalaze u tom paketu.

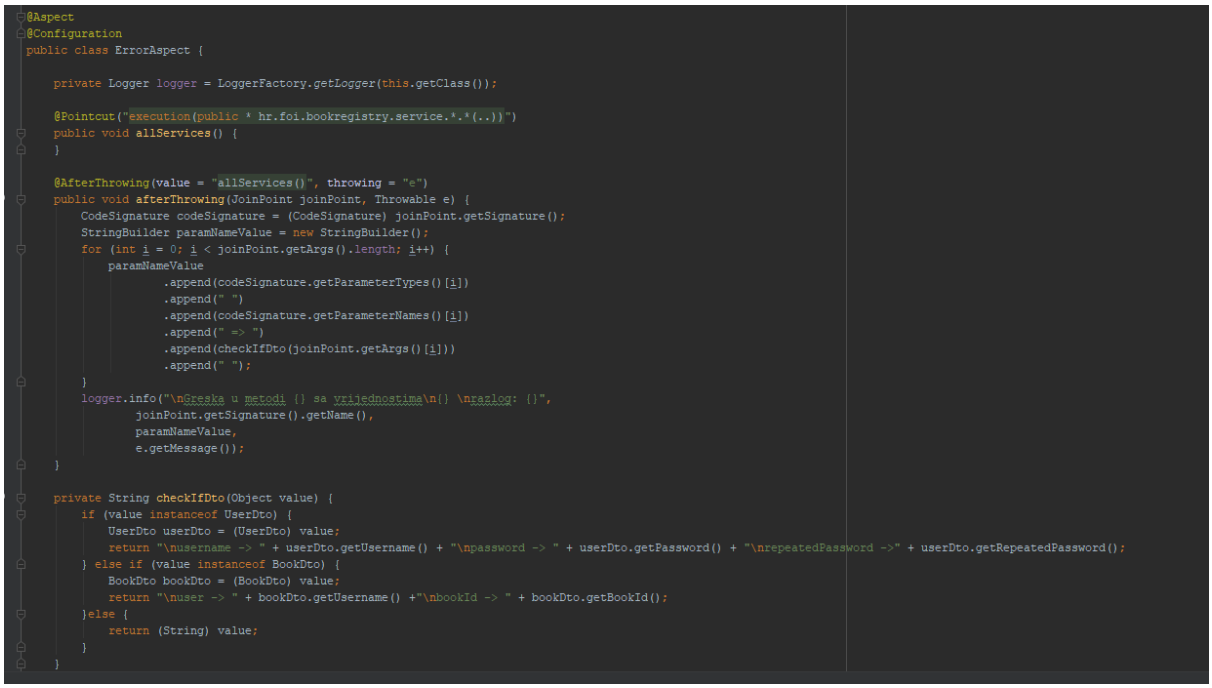
Svaki put kada se aktivira neki poziv prema servisu taj aspekt će se aktivirati. Primjer je pritisak gumba na bilo kojem od zaslona koje smo vidjeli prije. Točnije aspekt će se aktivirati samo ako nije bačena iznimka prilikom izvršavanja tih metoda. Sljedeća slika prikazuje konzoli ispis nakon uspješne prijave.

```
2019-09-01 15:58:56.325  INFO 5176 --- [nio-8080-exec-2] eAspect$$EnhancerBySpringCGLIB$$f565b5ce :
Vrijeme za izvodenje login metode u controlleru je 125 milisekundi
```

Slika 24.Konzoli ispis nakon okidanja aspekta

6.1.2. Hvatanje iznimaka

Sljedeći aspekt napisan je kako bi mogli hvatati iznimke unutar aplikacije.

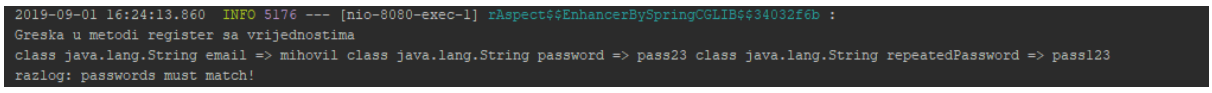


```
@Aspect
@Configuration
public class ErrorAspect {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Pointcut("execution(public * hr.foi.bookregistry.service.*.*(..))")
    public void allServices() {
    }

    @AfterThrowing(value = "allServices()", throwing = "e")
    public void afterThrowing(JoinPoint joinPoint, Throwable e) {
        CodeSignature codeSignature = (CodeSignature) joinPoint.getSignature();
        StringBuilder paramNameValue = new StringBuilder();
        for (int i = 0; i < joinPoint.getArgs().length; i++) {
            paramNameValue
                    .append(codeSignature.getParameterTypes()[i])
                    .append(" ")
                    .append(codeSignature.getParameterNames()[i])
                    .append(" => ")
                    .append(checkIfDto(joinPoint.getArgs()[i]))
                    .append(" ");
        }
        logger.info("\nGreska u metodi {} sa vrijednostima\n{} \nrazlog: {}",
                joinPoint.getSignature().getName(),
                paramNameValue,
                e.getMessage());
    }

    private String checkIfDto(Object value) {
        if (value instanceof UserDto) {
            UserDto userDto = (UserDto) value;
            return "\nusername -> " + userDto.getUsername() + "\npassword -> " + userDto.getPassword() + "\nrepeatedPassword ->" + userDto.getRepeatedPassword();
        } else if (value instanceof BookDto) {
            BookDto bookDto = (BookDto) value;
            return "\nuser -> " + bookDto.getUsername() +"\nbookId -> " + bookDto.getBookId();
        }else {
            return (String) value;
        }
    }
```

Slika 25. Hvatanje iznimaka

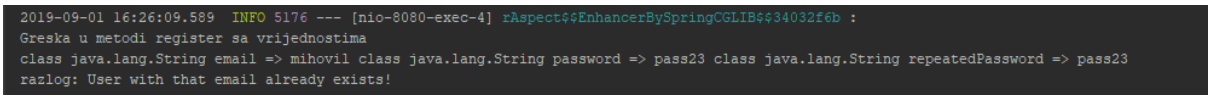
Na ovoj primjeru vidimo da je definirana drugačija točka spajanja. Ovdje smo naveli da želimo da se ovaj aspekt aktivira nakon bačene iznimke u bilo kojoj od klasa unutar navedenog paketa čije su metode definirane kao javne (eng. Public). Također smo specificirali da metoda može imati bilo koji povratni tip pomoću znaka zvjezdice „*“ te da može imati bilo koji broj parametara koje prima. Znamo da se aspekt aktivira samo kod bačenih iznimaka zbog savjeta definiranog s anotacijom *@AfterThrowing*. Savjet je napisan tako da se prikaže naziv metode u kojoj je bačena iznimka zajedno sa vrijednostima koje je metode primila prilikom izvršavanja. Na slici također vidimo još jednu privatnu metodu koja provjerava da li primljena vrijednost objekt, ako je, onda ispiše njegove podatke.

Na sljedećim slikama možemo vidjeti ispis kada se aktivira aspekt.



```
2019-09-01 16:24:13.860  INFO 5176 --- [nio-8080-exec-1] rAspect$$EnhancerBySpringCGLIB$$34032f6b :
Greska u metodi register sa vrijednostima
class java.lang.String email => mihovil class java.lang.String password => pass23 class java.lang.String repeatedPassword => pass123
razlog: passwords must match!
```

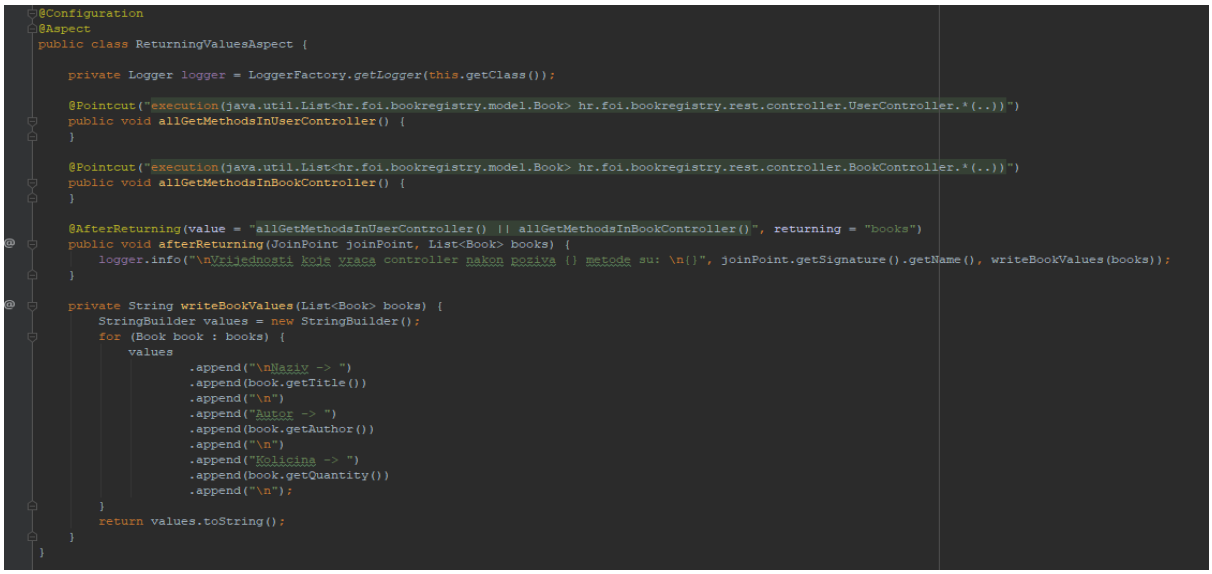
Slika 26. Različite lozinke prilikom registracije



Slika 27. Korisnik već postoji prilikom registracije

6.1.3. Povratne vrijednosti

U ovom poglavlju vidjet ćemo aspekt koji se okida nakon što definirana metoda vrati rezultat, koji je u našem slučaju lista knjiga. Takvu vrstu akcije primjenjujemo anotacijom *@AfterReturning*.



Slika 28. Aspekt za ispis vraćenih vrijednosti

Na slici vidimo definirane dvije točke spajanja. Jedna se odnosi na sve metode unutar klase *UserController* koje imaju definirani povratni tip listu knjiga. Druga je skoro identična, a razliku čini samo specificirana klasa, u ovom slučaju *BookController.* Vidimo kako smo kombinirali te dvije pomoću znaka za ili „||“. Naravno mogli smo te klase definirati u jednoj točki spajanja međutim ovako vidimo još jednu od mogućnosti koju smo prije spominjali.

Primjer ispisa vraćenih vrijednosti vidljiv je na slikama ispod.

```
2019-09-01 17:35:44.659  INFO 1444 --- [nio-8080-exec-5] sAspect$$EnhancerBySpringCGLIB$$ab2491c9 :
Vrijednosti koje vraca controller nakon poziva getUserBooks metode su:

Naziv -> Harry Potter and the Sorcerer's Stone
Autor -> J. K. Rowling
Kolicina -> 4

Naziv -> The Hobbit
Autor -> J. R. R. Tolkien
Kolicina -> 4

Naziv -> 1984
Autor ->  George Orwell
Kolicina -> 4

Naziv -> Pride and Prejudice
Autor -> Jane Austen
Kolicina -> 4

Naziv -> The Great Gatsby
Autor -> F. Scott Fitzgerald
Kolicina -> 4

Naziv -> The Catcher in the Rye
Autor -> J. D. Salinger
Kolicina -> 4
```

Slika 29. Prikaz vraćenih vrijednosti u konzoli

6.1.4. Transakcije

Ako izvršavamo više akcije nad bazom podataka unutar jedne metode može se dogoditi da se jedna akcija izvrši uspješno dok druga ne. Najčešće tada želimo da se ni prva akcija nije dogodila. Primjer takvog događaja bi bio dohvaćanje korisnika iz baze kojem moramo dodijeliti novu knjigu. Trebalo bi nakon dodavanja knjige korisniku u bazu spremiti promjene nad tablicom korisnika, ali i tablicom knjiga jer se treba modificirati jedan stupac. Ako bi uspjeli spremiti nove podatke u tablicu korisnik a ne bi uspjeli promijeniti stupac u tablici knjige došlo bi do nepoželjnog spremanja podataka u bazu. Tada trebamo izvršiti takozvani *Rollback*. On se može aktivirati kada metodu anotiramo s ključnom riječju *@Transactional*. Za njega ne trebamo definirati novi aspekt jer je već podržan od strane *Spring* okvira. Naravno mogli bi napraviti novi aspekt koji radi istu stvar, međutim uvijek je bolje koristiti unaprijed pripremljene akcije koje su razvili stručnjaci kako bi olakšali posao programerima.

Primjer metode koja koristi transakcije možemo vidjeti ispod.

```
@Transactional
public void returnBook(Long bookId, String email) {
    User user = getUserByUsername(email);
    Book book = user.getBooks() List<Book>
            .stream() Stream<Book>
            .filter(userBook -> userBook.getId().equals(bookId)) Stream<Book>
            .findAny() Optional<Book>
            .orElseThrow(() -> new RuntimeException("User does not own this book"));
    user.getBooks().remove(book);
    userRepository.save(user);
    book.setQuantity(book.getQuantity() + 1);
    bookRepository.save(book);
}
```

Slika 30. Metoda s transaction anotacijom

Kada bi gledali što se događa ispod haube te anotacije dobili nešto slično sljedećem kodu.

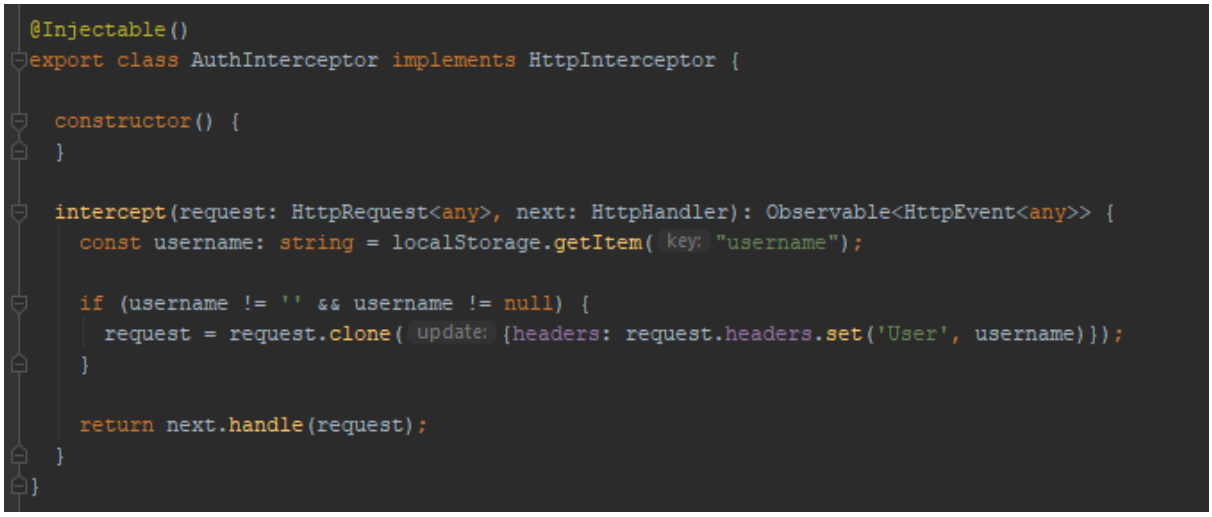
```
UserTransaction utx = entityManager.getTransaction();
try {
    utx.begin();
    poslovnaLogika();
    utx.commit();
} catch(Exception ex) {
    utx.rollback();
    throw ex;
}
```

Već vidimo da bi savjet okolo bio idealan kada bi ručno kreirali aspekt za transakcije jer bi metodu poslovnaLogika() zamjenili s već viđenom metodom *procced()* od klase *ProceedingJoinPoint*. Ovaj aspekt napravljen je putem ručno rađenih anotacija koji točku spajanja gledaju kao sve metode koju su anotirane s ključnom riječju *@Transactional*. U sljedećem poglavlju vidjet ćemo kako se to radi u praksi.

Sigurnost

Sada ćemo prikazati kako je implementirana sigurnost aplikacije putem aspekata. Pod sigurnosću mislimo na zaštitu takozvanih API krajnjih točaka (eng. Endpoint). Ideja ja takva da su sve krajnje točke osim onih za prijavu i registraciju zaštićene, odnosno da im ne može pristupiti nitko osim korisnika koji postoje u bazi. Admin ima dodatnu provjeru kojom se zaštičuje i krajnja točka koja omogućuje dodavanje nove knjige u bazu podataka. Spring okvir naravno već pruža način zaštite aplikacije korištenjem dodatnih biblioteka koje su vezane za sigurnost, ali za potrebe prikaza aspekata implementirani su ručno. Zasloni koje smo vidjeli u prijašnjim poglavljima također su zaštićeni korištenjem *JavaScripta* i lokalne pohrane (eng.

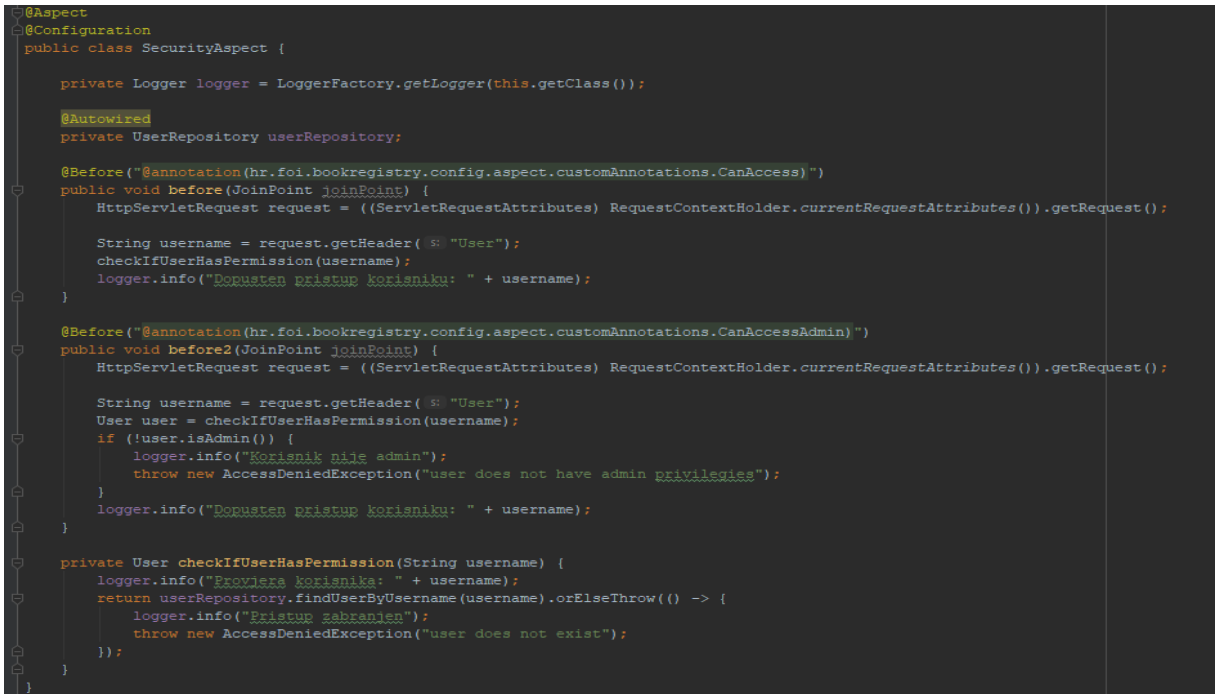
Local Storage). Točnije kada se korisnik uspješno prijavi u aplikaciju, u lokalnu pohranu spremi se vrijednost koja se šalje u zaglavlje (eng. Header) svakog sljedećeg zahtjeva kako je prikazano na sljedećoj slici.



Slika 31. Interceptor u Angularu

Gore vidimo presretač kojeg smo objašnjavali pri početku ovoga rada. Njegova zadaća u ovom slučaju je hvatanje svih zahtjeva koji su upućeni prema serveru. Nakon što se zahtjev uhvati, on se modificira tako da se u njega doda nova vrijednost koju će primiti server.

Aspekt je implementiran na sljedeći način.



Slika 32. Aspekt za zaštitu krajnjih točaka

Prva metoda odnosi se na sve krajnje točke koje nisu prijava i registracija, dok se druga odnosi na samo jednu krajnju točku koja je dozvoljena *admin* korisnicima.

Ovdje nije kreirana posebna metoda koja je definirana s anotacijom *@Pointcut* kako bi demonstrirali da se ista logika može ubaciti odmah nakon definiranja savjeta koji će se koristiti. Za ovaj slučaj vidimo da je ključna riječ točke spajanja *@annotation* jer smo ručno kreirali anotaciju koju smo definirali ispred onih metoda koje želimo zaštititi. Sljedeća slika prikazuje kako se ručno kreira anotacija te na koji način je primjenjena u kodu.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface CanAccess {
}

    @CanAccess
    @RequestMapping(value = "/borrow-book")
    @PostMapping
    public void borrowBook(@RequestBody BookDto bookDto) {
        userService.borrowBook(bookDto.getBookId(), bookDto.getUsername());
    }
```

Slika 33. Ručno rađene anotacije

Pomoću anotacije *@Target* definiramo koje spojne točke će biti cilj našega aspekta. U ovom slućaju radi se o metodama. *@Retention* nam omogućuje specificiranje načina na koji se odvija definirana anotacija.

Na sljedećoj slici prikazat ćemo kako se odvija ispis u konzoli nakon okidanja gore navedenog aspekta.

```
2019-09-03 18:29:50.636  INFO 1144 --- [nio-8080-exec-2] yAspect$$EnhancerBySpringCGLIB$$de937e3f : Provjera korisnika: admin
2019-09-03 18:29:50.640  INFO 1144 --- [nio-8080-exec-2] yAspect$$EnhancerBySpringCGLIB$$de937e3f : Dopusten pristup korisniku: admin
```

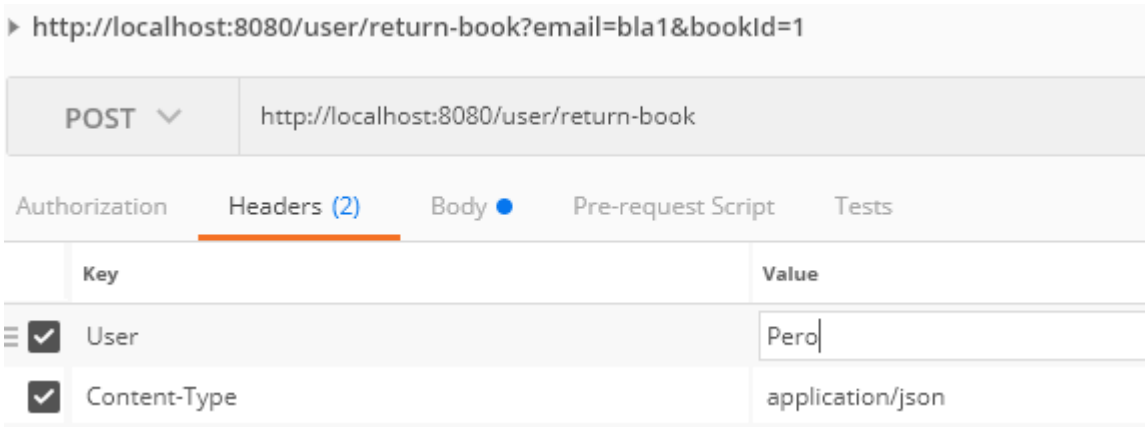
Slika 34. Konzolni prikaz aspekta za sigurnost

Vidimo da se prijavljeni korisnik admin i nakon provjere baze podataka aplikacija je dozvolila pristup krajnjoj točki kako bi se izvršio zahtjev.

Neuspješan zahtjev se ne može testirati putem aplikacije jer će se uvijek u zaglavlju zahtjeva slati validni podaci. Iz tog razloga koristili smo aplikaciju *Postman*[4] kako bi simulirali neuspješan pokušaj izvršenja zahtjeva.

[4] Alat za slanje Rest API poziva na definirane krajnje točke.

U *Postmanu* primjer takvog pokušaja prikazan je u nastavku.



Slika 35. Zahtjev u aplikaciji Postman

Korisnik Pero ne postoji u bazi stoga je odgovor koji se dobio nakon zahtjeva sljedeći:


```
{
    "timestamp": "2019-09-03T16:38:25.658+0000",
    "status": 403,
    "error": "Forbidden",
    "message": "Access Denied",
    "path": "/user/return-book"
}
```

7. Zaključak

U ovome radu najviše smo pričali o tehnikama aspektno orijentiranog programiranja. Vidjeli smo da postoje razni načini implementacije u različitim okvirima i programskim jezicima. Primijetili smo da pisanje aspekata može, ali i ne mora biti složeno jer sve ovisi o iskustvu programera i poznavanju tehnika i alata za njihovo primjenjivanje. Iako nije složeno ne znači da ih je lagano primijeniti u smislu pisanja čitkog koda jer treba dobro poznavati domenu aplikacije koja ih koristi. To znači prepoznati najbolja mjesta, odnosno presijecajuća svojstva, za rješavanje problema poput zapletanja ili raspršivanja koda.

Dosta vremena smo iskoristili i za usporedbu aspektno orijentiranog programiranja s objektnim, gdje možemo slobodno ponoviti da se oni ne bi nikada trebali uspoređivati u smislu „koji je bolji?“, jer iz svega možemo zaključiti da je AOP nastao kao dodatak OOP kako bi mogao proširiti njegove mogućnosti, odnosno smanjiti njegovo nedostatke. Također smo naučili da AOP nije nužan uvjet da bi aplikacija radila, ali zato svojom modularnošću uvelike pridonosi lakšem čitanju i razumijevanju koda.

Iako postoje različiti okviri za pisanje aspekata, svi se baziraju na koncepte poput savjeta, spojnih točaka, točaka spajanja i slično. Razumijevanje tih koncepata odlika su poznavanja tehnika aspektno orijentiranog programiranja koje bi trebalo naučiti prije nego li se netko odluči za njihovu primjenu u svojoj aplikaciji.

Programski primjer razvijen za potrebe ovoga rada koristio je *Spring* okvir uz pomoć *AspectJ* biblioteke, te je upravo ta kombinacija po meni najbolja kako bi se napisao kvalitetan i razumljiv kod. Internet je pun primjera i savjeta kako i gdje se mogu aspekti primijeniti što uvelike olakšava posao programerima koji se žele okušati u njihovoj primjeni. Trenutno ne postoje tehnike rada koje na ovako kvalitetan način rješavaju probleme presijecajućih svojstava pa mislim da je poznavanje te tehnike potrebno kako bi se mogao napisati kod koji je ne samo lakši za održavanje, već kojeg je lakše ponovno koristiti i tako ubrzati razvoj velikih aplikacija.

Smatram da će tehnike aspektno orijentiranog programiranja u budućnosti biti puno poznatije novim programerima i da će biti jedna od osnova koje će se učiti uz objektno orijentirano programiranje.

Popis literature

Baeldung. (October 2017). *Comparing Spring AOP and AspectJ.* Dohvaćeno iz Baeldung: https://www.baeldung.com/spring-aop-vs-aspectj

Baeldung. (July 2017). *Introduction to Pointcut.* Dohvaćeno iz Baeldung: https://www.baeldung.com/spring-aop-pointcut-tutorial

Baeldung. (August 2019). *Intro to AspectJ.* Dohvaćeno iz Baeldung: https://www.baeldung.com/aspectj

Cshandler. (October 2015). *Managing Cross Cutting Concerns – Logging.* Dohvaćeno iz A developers blog: https://www.cshandler.com/2015/10/managing-cross-cutting-concerns-logging.html#.XSsKhugzaM8

Debnath, M. (December 2014). *Understanding Interceptors for Java EE.* Dohvaćeno iz developer.com: https://www.developer.com/java/understanding-interceptors-for-java-ee.html

Dinesh, R. (March 2017). *Spring AOP Tutorial Examples Aspect Framework.* Dohvaćeno iz Dinesh on Java: https://www.dineshonjava.com/spring-aop-tutorial-with-example-aspect-advice-pointcut-joinpoint/

Gibbs, M. (2019). *Aspect-Oriented Programming vs. Object-Oriented Programming.* Dohvaćeno iz Study.com: https://study.com/academy/lesson/aspect-oriented-programming-vs-object-oriented-programming.html

I. Bergman, C. Goransson. (June 2004). *An Introduction to Aspect-Oriented Programming.* Dohvaćeno iz Computer Science – Datavetenskap: https://www.cs.kau.se/cs/education/courses/davcdiss/Exjobb_2004/C2004-06.pdf

Indika. (July 2011). *Difference Between AOP and OOP.* Dohvaćeno iz Difference Between.com: https://www.differencebetween.com/difference-between-aop-and-vs-oop/

JBossDeveloper. (n.d.). *Aspect Oriented Programming (AOP) Support.* Dohvaćeno iz JBossDeveloper: https://docs.jboss.org/jbossas/jboss4guide/r4/html/aop.chapt.html

Laukkanen, J. (FEbruary 2008). *Aspect-Oriented Programming.* Dohvaćeno iz Computer Science: https://www.cs.helsinki.fi/u/paakki/laukkanen.pdf

Lopes, C. V. (December 2002). *Aspect-Oriented Programming:An Historical Perspective.* Dohvaćeno iz CiteSeerX: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=6B37F8165E587F52CDCAB965CDB00CC5?doi=10.1.1.58.7760&rep=rep1&type=pdf

Marketing, M. (n.d.). *AspectJ in Action.* Dohvaćeno iz http://assets.devx.com/download/AspectJinAction_CH2.pdf

Mkyong. (March 2010). *Spring AOP Example – Advice.* Dohvaćeno iz Mkyong.com: https://www.mkyong.com/spring/spring-aop-examples-advice/

Pankaj. (n.d.). *Spring AOP Example Tutorial.* Dohvaćeno iz Journal Dev: https://www.journaldev.com/2583/spring-aop-example-tutorial-aspect-advice-pointcut-joinpoint-annotations

ProgrammingAndPolitics. (May 2007). *Interceptors vs. AOP.* Dohvaćeno iz Programming and politics: http://fupeg.blogspot.com/2007/05/interceptors-vs-aop.html

Rani, B. (n.d.). *Introduction of Programming Paradigms.* Dohvaćeno iz GeeksforGeeks: https://www.geeksforgeeks.org/introduction-of-programming-paradigms/

Rani, B. (n.d.). *OOPs | Object Oriented Design.* Dohvaćeno iz GeeksforGeeks: https://www.geeksforgeeks.org/oops-object-oriented-design/

Spinczyk, O. (2005). *Aspect-Oriented Programming with C++.* Dohvaćeno iz Modularity: http://modularity.info/conference/2005/archive/AspectCpp-talk.pdf

Spring. (n.d.). *Aspect Oriented Programming with Spring.* Dohvaćeno iz Spring: https://docs.spring.io/spring/docs/4.3.15.RELEASE/spring-framework-reference/html/aop.html

Tereshchenko, V. (December 2017). *Aspect Oriented Programming in C# using DispatchProxy.* Dohvaćeno iz DZone: https://dzone.com/articles/aspect-oriented-programming-in-c-using-dispatchpro

TutorialPoint. (n.d.). *Spring Framework - Overview.* Dohvaćeno iz tutorialspoint: https://www.tutorialspoint.com/spring/

Vaidya, N. (May 2019). *Spring AOP Tutorial.* Dohvaćeno iz Edureka: https://www.edureka.co/blog/spring-aop-tutorial/

Widinghoff, S. (September 2015). *The basics of AOP.* Dohvaćeno iz blog.: https://blog.jayway.com/2015/09/07/the-basics-of-aop/

Yang, G. K. (n.d.). *A Brief Introduction to Aspect-Oriented Programming.* Dohvaćeno iz College of Engineering | Computer Science and Engineering

: http://www.cse.msu.edu/~cse870/Lectures/2015/12-AOP-suppl-new-notes.pdf

Zhou, A. (April 2019). *What is interceptor in Java and what is it used for?* Dohvaćeno iz Quora: https://www.quora.com/What-is-interceptor-in-Java-and-what-is-it-used-for

Popis slika

Popis tablica