Univerzitet u Beogradu

Matematički fakultet

Master teza

Aspektno Orijentisano Programiranje

(metodologija i implementacije)

Ivan Cikić

Mentor: Dušan Tošić

Beograd
2008

Sadržaj

1. Uvod

Softverski sistemi svakim danom postaju sve kompleksniji i svi indikatori pokazuju da će se takav trend nastaviti u budućnosti. Programeri se susreću sa sve složenijim i obimnijim zahtevima, samim tim je sve teže napraviti dobar dizajn, tj. dizajn koji će uzeti u obzir kako sadašnje zahteve tako i potencijalne buduće pravce razvoja sistema.
Ključ u borbi sa kompleksnošću softvera jeste modularizacija. Dekompozicijom problema na manje celine - potprobleme, i rešavajući svaku celinu pojedinačno, lakše se dolazi do zadovoljavajućeg dizajna i implementacije celog sistema. Svaki od prepoznatih potproblema u sistemu predstavlja jednu funkcionalnost sistema, odnosno jednu dužnost (*concern*) koju sistem mora da ispuni da bi obavljao svoju funkciju. Prema tome, softverski sistem možemo definisati kao skup dužnosti koje mora da ispuni. Sve dužnosti koje sistem mora da zadovolji možemo podeliti u dve kategorije:

- Centralna (*core*) dužnost koja opisuje glavnu funkciju jednog modula i karakteristična je samo za taj modul.
- Isprepletane (*crosscutting*) dužnosti koje opisuju sporedne funkcije sistema prisutne u mnogobrojnim modulima.

Na primer, u bankarskom sistemu, osnovna dužnost sistema jeste da vodi računa o klijenima i računima, izračunava kamate, obavlja transakcije izmedju banaka, itd. Svaku od ovih dužnosti nazivamo centralnim dužnostima koje sistem mora da zadovolji. Medjutim, pored ovih, glavnih dužnosti, svaka aplikacija sa poslovnom primenom mora zadovoljiti još mnogo drugih, “sporednih” dužnosti, koje se prepliću sa svim centralnim modulima: identifikacija i autorizacija korisnika, logovanje akcija, keširanje podataka, nadgledanje peformansi sistema, upravljanje transakcijama nad bazom podataka i mnogo drugih. Provera identiteta korisnika je neophodna u svakom modulu koji mora da ispuni određene sigurnosne zahteve, upravljanje transakcijama je potrebno na svakom mestu u sistemu koji komunicira sa bazom podataka itd. Takve dužnosti sistema nazivamo isprepletane.

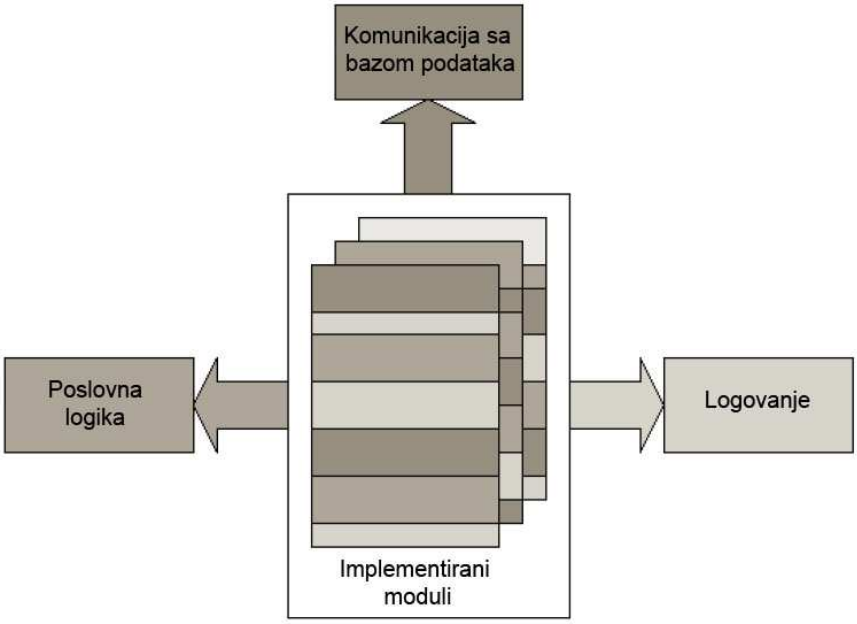
Objektno orjentisano programiranje (OOP), najrasprostranjenija metodologija danas, uvela je u programiranje objektnu apstrakciju i modularizaciju. Zbog toga, OOP je odličan pristup u implementaciji centralnih modula sistema, odnosno zadovoljavanju centralnih dužnosti sistema. Snaga OOP jeste u dekompoziciji sistema na manje objekte i modelovanju ponašanja svakog objekta. Medjutim, OOP nije najbolji način za realizaciju isprepletanih dužnosti jer dovodi do rasipanja koda neophodnog za implementaciju takve funkcionalnosti u mnogo drugih modula. Na primer, u implementaciji centralnog modula često se može videti kôd koji se bavi upravljanjem transakcijama, hvatanjem grešaka ili logovanjem. Ukratko, tipičan OOP dovodi do nepoželjne, čvrste veze (*tight coupling*) izmedju centralnih i isprepletanih dužnosti sistema. Uvodjenje nove isprepletane dužnosti koju sistem mora da zadovolji ili modifikovanje postojeće može dovesti do modifikacija u relevantnom centralnom modulu. Ovde na scenu stupa **Aspektno Orjentisano Programiranje (AOP).**

AOP je metodologija koja omogućava razdvajanje isprepletanih dužnosti sistema uvodeći novu jedinicu apstrakcije – **aspekt**. Koristeci AOP, realizacija svake isprepletane dužnosti sistema je enkapsulirana u aspektu i ne dolazi do sjedinjavanja sa centralnim modulima. Fokus svakog aspetka je specifična isprepletana dužnost, čime se omogućava implementacija centralnih modula koji nisu više opterećeni ispunjavanjem isprepletanih dužnosti i slobodni su da evoluiraju nezavisno od njih. Aspektni tkalac (*weaver*), entitet sličan kompajleru, je zadužen za kompoziciju finalnog sistema, kombinujući centralne i isprepletane module, zadovoljavajući sve dužnosti sistema. Taj process se zove tkanje (*weaving*). Rezultat procesa jeste da AOP uvodi modularizaciju u isprepletene dužnosti i dovodi do arhitekture sistema koja je laka za dizajn, implementaciju i održavanje.

2. AOP metodologija

2.1 Simptomi isprepletanih dužnosti

U uvodu smo uveli pojam centralnih (glavnih) i isprepletanih (sporednih) dužnosti koje sistem mora da zadovolji i realizuje. Slika 2.1 prikazuje realizaciju i interakciju ovih dužnosti u jednom modulu sistema. Ovaj graficki prikaz pokazuje sistem dobijen korišćenjem danas široko prihvaćenih tehnika za implementaciju softvera, sistem u kome su raznovrsne dužnosti sistema medjusobno zamršene.



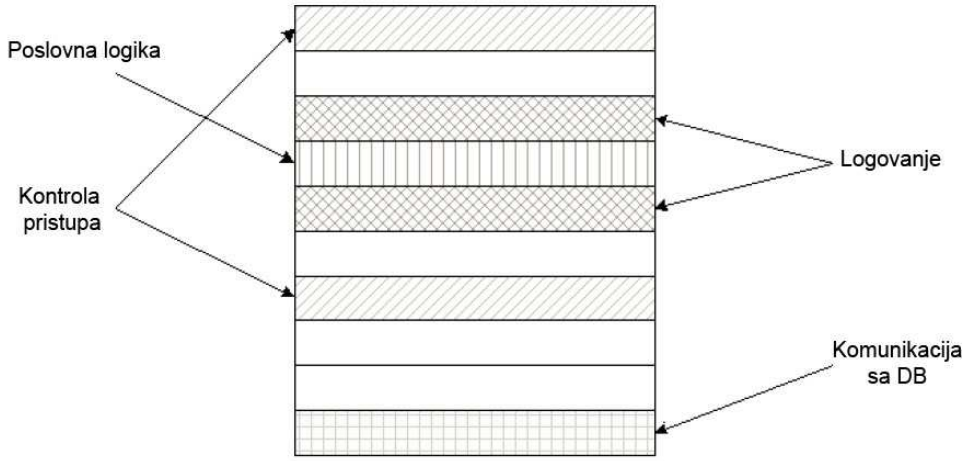
Slika 2.1: Sistem kao kompozicija više dužnosti. Svaki implementacioni modul sadrži kôd koji realizuje više od jedne dužnosti sistema.

Čist objektno orjentisani pristup implementaciji isprepletanih dužnosti sistema dovodi do pojave dva klasična neželjena simptoma: preplitanje koda (*code tangling*) i rasipanje koda (*code scaterring*).

Preplitanje i rasipanje koda mogu se pojaviti takodje i kao posledica lošeg dizajna i/ili implementacije (npr. kopiranje istog koda na više mesta). Takvi problemi se, naravno, mogu razrešiti u okvirima OOP. Medjutim, upotrebom OOP za realizaciju isprepletanih dužnosti sistema, problem preplitanja i rasipanja koda je uvek prisutan.

2.1.1 Preplitanje koda

Preplitanje koda se javlja u slučajevima kada jedan modul ispunjava više dužnosti sistema istovremeno. Programer pri implementaciji jednog modula mora da vodi računa o dužnostima sistema kao sto su poslovna logika, sinhronizacija, logovanje, optimizacija, upravljanje greškama, kontrola pristupa, itd. Rezultat takve implementacije je preplitanje koda prikazano šemom 2.2:



Slika 2.2: Preplitanje koda nastalo kao posledica simultane implementacije više dužnosti sistema u jednom modulu.

Ilustrujmo ovaj princip kroz deo koda. Primer prikazuje implementaciju modula koji enkapsulira poslovnu logiku u klasičnom OOP stilu.

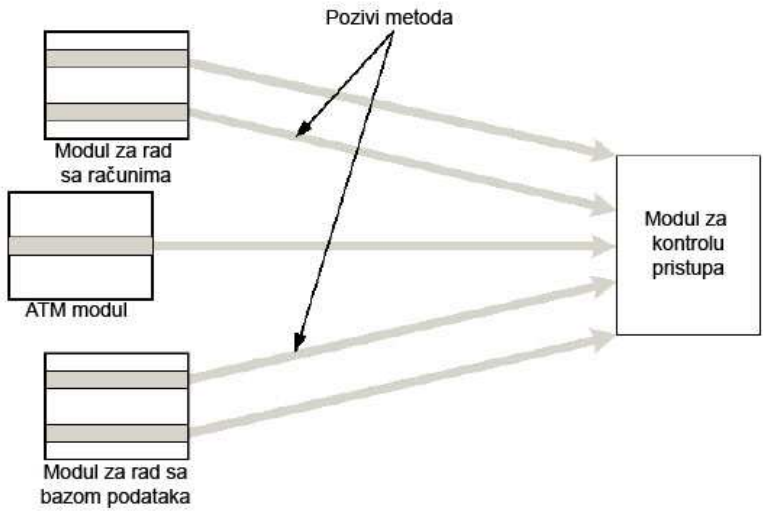
```
00 public class KonkretniServis extends ApstraktniServis {
01 ... definicija članova klase
02 ... definicija logera
03 ... status ažuriranja keša
04 ... objekat potreban za kontrolu konkurentnosti
05 ... preklopljene metode
06 public void servisOperacija1(<parametri operacije>) {
07 ... provera indentiteta – autorizacija
08 ... zaključaj objekat koji kontroliše pristup klasi (“thread-safety”)
09 ... ažuriranje keša
10 ... početak transakcije
11 ... loguj početak operacije
12 ... Izvrši centralnu operaciju
13 ... loguj kraj operacije
14 ... komituj transakciju
15 ... otključaj objekat za kontrolu konkurentnosti
}
```

Primer 2.1

Iako je svaki problem specifičan, ovaj primer dobro prikazuje uobičajni problem sa kojim se suočavaju programeri. I pored toga postoji konceptualna razdvojenost izmedju različitih dužnosti sistema tokom izrade dizajna, tokom implementacije dolazi do medjusobnog preplitanja više dužnosti u jednom modulu. Ovakva implementacija ne ispunjava neke od osnovnih principa dobrog objektno orijentisanog dizajna, izmedju ostalih i “*Single Responsibility Principle*” - za promenu klase nikada ne sme postojati više od jednog razloga[10], kao i “*Open/Closed Principle*” - softverski entiteti (klase, moduli, funkcije, itd) treba da budu otvoreni za nasledjivanje i zatvoreni za promenu[11].

2.1.2 Rasipanje koda

Rasipanje koda nastaje kada jednu funkciju sistema implementiramo u više modula. S obzirom da su po definiciji ispripletane dužnosti sistema prisutne u više modula, njihove implementacije su takodje prisutne u svakom od tih modula. Slika 2.3 ilustruje kako bi u bankarskom sistemu bila implementirana kontrola pristupa koristećenjem konvencionalne tehnike.



Slika 2.3: Implementacija kontrole pristupa korišćenjem konvencionalne tehinke. U modulu za kontrolu pristupa implementrana je logika za autorizaciju u javnim metodama. Svaki klijentski modul mora da sadrži kôd koji poziva te metode.

I u slučaju dobro dizajniranog sigurnosnog modula koji nudi apstraktan API i sakriva detalje implementacije, svaki "klijentski" modul (u ovom primeru modul za rad sa računima, ATM modul i modul za rad sa bazom podataka) mora da sadrži kôd koji poziva sigurnosni API. Taj kôd je rasut po svim modulima koji koriste sigurnosni modul i posledica toga je neženjena povezanost svih modula kojima treba sigurnosna provera i samog sigurnosnog modula (primer 2.1, linija 07).

2.2 Posledice nemodularnosti

Preplitanje i rasipanje koda utiču negativno na dizajn i razvoj softvera na više načina: otežano praćenje koda, niska produktivnost, niži nivo iskorišćenosti koda, loš kvalitet i otežana evolucija.

- Otežano praćenje koda - Simultana implementacija više dužnosti u modulu, dovodi do nejasne veze izmedju dužnosti sistema i njene implementacije, što otežava praćenje veze izmedju zahteva i implementacije. Na primer, u slučaju da želimo da nadjemo implementaciju dužnosti za autorizaciju korisnika, potencijalno bi morali da ispitamo sve module u sistemu.
- Niska produktivnost - Simultana implementacija više dužnosti u modulu prebacuje fokus sa glavne na više sporednih dužnosti. Taj gubitak fokusa dovodi do smanjene produktivnosti.
- Niži nivo iskorišćenosti koda - U slučaju kada je u modulu implementirano više dužnosti sistema, veća je verovatnoća da drugi sistemi kojima je potrebna slična funkcionalnost neće moći da iskoriste već postojeći modul. Zamislimo implementaciju jednog servisa koji zahteva sigurnosnu proveru pre nego omogući korisniku pristup. Nekom drugom projektu može trebati isti servis ali sa drugačijom proverom sigurnosti, dok treći projekat nema potrebe za sigurnosnom proverom. U ovom slučaju implementacija servisa iz prvog projekta ne može se ponovo iskoristiti iako je logika samog servisa identična u sva tri slučaja.
- Loš kvalitet - Preplitanje koda otežava pregledanje koda i uočavanja potencijalnih grešaka. Na primer, da bi se izvršilo kvalitetno ispitivanje kvaliteta koda nad takvom implementacijom modula, neophodno je učešće eksperata u svakoj od oblasti koje taj modul pokriva.
- Otežana evolucija - Usled nejasnih zahteva i ograničenih resorsa često dolazi do dizajna koji reflektuje samo trenutne zahteve. Takav sistem je teško održavati i razvijati u skladu sa budućim zahtevima. S obzirom da rešenje nije modularno, buduće prepravke podrazumevaju modifikaciju mnogobrojnih modula. Veliki broj promena u implementaciji može dovesti do nekonzistentnosti i pojavljivanja novih grešaka.

Svi ovi problemi su doveli do potrage za boljim pristupom pri dizajnu i implementaciji softvera, a kao jedno od rešenja pojavio se i AOP.

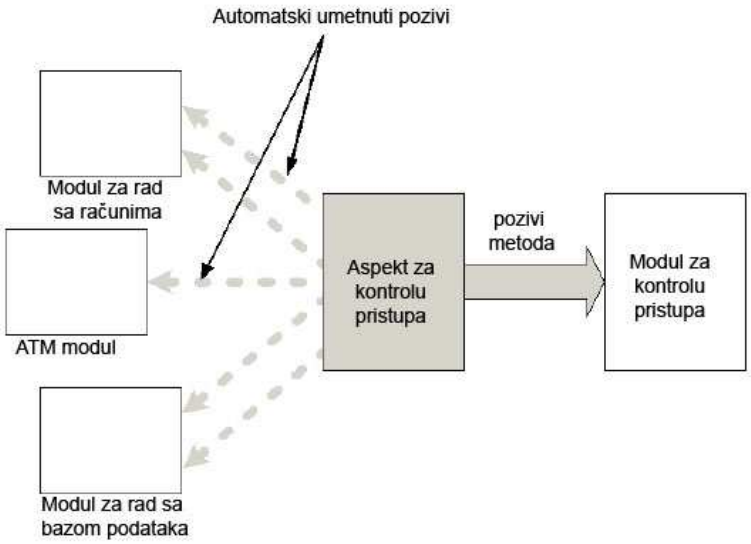
2.3 Modularnost uz pomoć AOP-a

U OOP, centralni moduli mogu biti povezani “slabom” vezom koristeći interfejse (*loose coupling*), ali ne postoji način da se takva veza uspostavi sa isprepletanim funkcionalnostima[5]. Razlog za to je što se implementacija svih dužnosti sistema sastoji iz dva dela, serverski deo (koji nudi niz servisa) i klijentski deo (koji koristi te servise). OOP može pomoći u modularizaciji serverskog dela pomoću klasa i interfejsa. Medjutim, kada je dužnost sistema isprepletane prirode, klijentski deo koda koji se sastoji od poziva servisa na serveru, rasut je svuda u sistemu.
Kao primer ovoga, posmatrajmo tipičnu implementaciju isprepletane dužnosti pomoću OOP: modul za autorizaciju koji definiše servis kroz jedan interfejs (primer 2.2, linija 03). Upotreba interfejsa oslabljuje vezu izmedju klijenata koji koriste servis i implementacija samog interfejsa. Klijenti koji koriste interfejs (*AccountServiceOOP* u primeru 2.2) nisu svesni implementacije koju zaista referenciraju. Bilo kakva promena u implementaciji servisa neće zahtevati promene u implementaciji klijenta. Čak i zamena trenutno korišćenog modula za autorizaciju sa drugim je samo stvar instanciranja nove implementacije modula u klijentu, a to se može izvršiti bez ili sa malo promena u klijentskom modulu. Ovaj aranžman ipak zahteva da klijent sadrži ugnježdene pozive servisa za autorizaciju. Takvi pozivi su nephodni u svakom modulu koji koristi autorizaciju (primer 2.2, linije 06, 15, 23).

```
00 public class AccountServiceOOP implements IAccountService {
01     IUserDao userDao;
02     IAccountDao accountDao;
03     IAuthorizationService authorizationService;
04
05     public List<Account> getAccountsForUser(User user) {
06         if (authorizationService.isAccessAllowed(user)) {
07             return accountDao.getAccountsForUser(user);
08         } else {
09             throw new SecurityException();
10         }
11     }
12
13     }
14     public void removeAccount(User user, Account account) {
15         if (authorizationService.isAccessAllowed(user, account)) {
16             user.getAccounts().remove(account);
17             userDao.saveOrUpdate(user);
18         } else {
19             throw new SecurityException();
20         }
21     }
22     public void saveOrUpdate(User user, Account account) {
23         if (authorizationService.isAccessAllowed(user, account)) {
24             user.getAccounts().add(account);
25             userDao.saveOrUpdate(user);
26         } else {
27             throw new SecurityException();
28         }
29     }
30 }
```

Primer 2.2. Primer implementacije modula za administraciju računa u bankarskom sistemu korišćenjem OOP metodologije. U svakoj metodi, svake klase gde je potrebna kontrola pristupa, potreban je poziv metode koja ne ispunjava centralnu dužnost modula

Koristeći AOP, ni jedan modul neće sadržati pozive za autorizaciju. Slika 2.4 prikazuje implementaciju iste sigurnosne provere prikazanu i na slici 2.3, samo koristeci AOP. Realizacija kontrole pristupa u sistemu (implementacija servisa i poziv servisa) sada se nalazi izolovana u sigurnosnom modulu i sigurnosnom aspektu.



Slika 2.4: Implementacija kontrole pristupa korištešćenjem AOP tehnike. Aspekt za kontrolu pristupa definiše tačke preseka u kojima treba izvršiti proveru indentiteta i autorizaciju i poziva javne metode definisane na modulu za kontrolu pristupa prilikom izvršavanja tih tačaka preseka. Klijentski moduli više ne sadrže kôd za autorizaciju.

Zahtevi za sigurnosnom proverom u svim modulima su sada realizovani u samo jednom modulu – aspektu za kontrolu pristupa (*AuthorizationAspect* u primeru 2.3). Sa ovakvom modularizacijom, svaka promena u logici sigurnosne provere će se reflektovati samo na aspekt za kontrolu pristupa, potpuno izolujući klijentske module (*AccountServiceAOP*). Fundamentalna razlika koju AOP donosi jeste obostrana nezavisnost pojedinačnih dužnosti sistema i nakon same implementacije, ne samo u toku dizajna. Implementacija se lako povezuje sa odgovarajućom dužnošću sistema koju realizuje, što kao rezultat daje sistem koji je jednostavnije razumeti, implementairati i menjati.

```
00 public class AccountServiceAOP implements IAccountService {
01     IUserDao userDao;
02     IAccountDao accountDao;
03 
04     public List<Account> getAccountsForUser(User user) {
05         return accountDao.getAccountsForUser(user);
06     }
07     public void removeAccount(User user, Account account) {
08         user.getAccounts().remove(account);
09         userDao.saveOrUpdate(user);
10     }
11     public void saveOrUpdate(User user, Account account) {
12             user.getAccounts().add(account);
13             userDao.saveOrUpdate(user);
14       }
15 }
16 
17 public aspect AuthorizationAspect {
18    IAuthorizationService authorizationService;
19    pointcut secureAccess() :
20             execution(* org.matf.icikic.banking.service.AccountServiceAOP.*(..));
21    before(User user, Account account) :
22               secureAccess() && args(user, account) {
24        if (!authorizationService.isAccessAllowed(user, account)) {
25            throw new SecurityException();
26        }
27    }
28 }
```

Primer 2.3. Implementacija modula za administraciju računa u bankarskom sistemu koja je u potpunosti izolovana od kontrole pristupa i AspectJ implementacija aspekta koji će obaviti sigurnosnu proveru **pre** svake metode u *AccountServiceAOP* klasi.

Do sada je bilo reči samo o AOP metodologiji. Vreme je da zađemo malo dublje i predjemo na AOP implementaciju. Kao i svaka druga metodologija u programiranju, AOP se sastoji iz dva dela[5]:

- Specifikacija jezika - opisuje jezičke konstrukcije i definiše sintaksu.
- Implementacija jezika - zadužena za verifikaciju i prevodjenje koda u izvršivu formu. Uglavnom realizovano pomoću neke vrste kompajlera.

2.4 AOP specifikacija jezika

Svaka implementacija AOP-a mora sadržati specifikaciju jezika kojim će se implementairati individualne dužnosti sistema i jezik kojim će se izražavati pravila za kompoziciju (preplitanje) implementiranih dužnosti u zajednički sistem[5].

- Implementacija individualnih dužnosti sistema
  Kao i u svakoj drugoj metodologiji, implementacija individualnih dužnosti sistema se izoluje u posebne module koje odlikuju trenutno stanje (podaci) i definisano ponašanje. Na primer, modul koji implementira keširanje podataka u sistemu će sadržati kolekciju keširanih objekata, proveravati vreme validnosti tih objekata, itd. Za implementaciju i centralnih i ispreplетanih funkcija sistema, uglavnom se koriste standardni jezici kao sto su C++, Java i C#.
- Specifikacija pravila preplitanja
  Pravila preplitanja odredjuju način na koji implementirane pojedinačne dužnosti treba integrisati da bi se formirao konačni sistem. Na primer, nakon implementiranog modula za autorizaciju potrebno je uvesti taj modul u postojeći sistem. U ovom slučaju pravila preplitanja bi se sastojala od informacija koje operacije u sistemu zahtevaju kontrolisani pristup (primer 2.3, linija 20) i pod kojim uslovima korisnicima treba dopustiti pristup (u primeru 2.3 logika enkapsulirana u *IAuthorizationService* modulu). Prava moć AOP-a jeste u jednostavnom predstavljanju pravila preplitanja.
  S obzirom da sada možemo sve sporedne (ispreplетane) dužnosti sistema izdvojiti u posebne module, glavna klasa brine samo o svojoj centralnoj dužnosti.

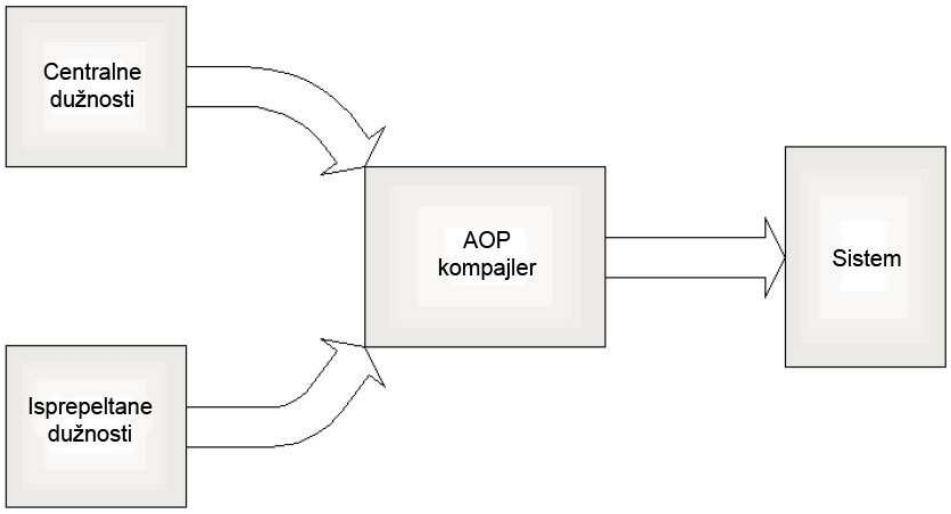
```
public class KonkretniServis extends ApstraktniServis {
... definicija članova klase
... preklopljene metode
public void servisOperacija1(<parametri operacije>) {
... izvrši centralnu operaciju
}
...
```

Primer 2.4

U poredjenju sa kodom prikazanim u primeru 2.1, ova klasa ne sadrži kôd kojim se realizuju sporedne dužnosti, samo kôd za glavnu dužnost klase. Izabrana AOP implementacija će spajanjem klasa i aspekata kreirati isprepletan (kombinovan) modul za izvršavanje. Jezik koji se koristi za specifikaciju pravila preplitanja može biti ekstenzija postojećeg jezika (AspectJ je AOP implementacija koja koristi Java jezik i definise ekstenziju Jave za definisanje pravila), ali može i deskriptivno definisati pravila preplitanja (pomoću XML-a, Java anotacija, .NET atributa itd).

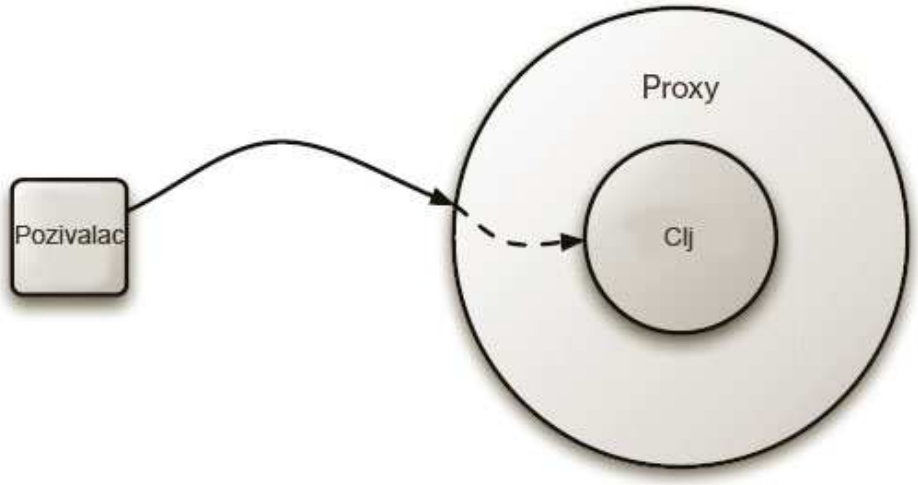
2.5 AOP implementacija

U AOP implementaciji izvršavaju se dve operacije: kombinuje se individulane dužnosti sistema korišćenjem pravila preplitanja, a zatim se dobjeni rezultati konvertuje u izvršiv kôd. Za izvršavanje ovih operacija zadužen je AOP procesor zvani aspektni tkalac (*weaver*).
Aspektni tkalac može biti implementiran na više načina. Na primer, u slučaju AOP implementacija baziranih na Javi, izvorni kôd biva transformiran u ispreplетan izvorni kôd a zatim ga Java kompjler konvertuje u bajtkod. U ovom pristupu postoji više nedostataka, najviše u tome da se dobijeni izvršivi kôd teško može debagovati koristeći početni izvorni kôd. Drugi pristup jeste da se izvorni kôd prvo kompajlira u bajtkod korišćenjem klasičnog Java kompajlera, a zatim se koristeći aspektni kompajer izvršava bajtkod manipulacija, tj. umeću se aspekti (videti poglavlje 3.5.1).



Slika 2.5: AOP implementacija zasnovana na aspektnom kompajleru.

Može se i dodatno manipulisati načinom umetanja aspekata u klase, u slučaju da želimo pomeriti proces preplitanja od kompaliranja ka izvršavanju. U Java programskom jeziku za takve potrebe koristiti se specijalni učitavač klasa (*class loader*), koji prvo učitava bajtkod aspekata, a zatim ih automatski umeće u klase prilikom učitavanju klasa.
Opisani pristupi su različiti u načinu i trenutku umetanja aspekata, ali se svi zasnivaju na konceptu aspektnog kompajlera. Postoji drugi opšte prihvaćeni način implementacije AOP, zasnovan na korišćenju dinamički kreiranih proxy objekata. U ovakvom pristupu svaki objekat u koji treba umetnuti aspekte je obuhvaćen proxy objektom. Na primer, Spring framework implementira AOP kreirajući proxy objekte u trenutku izvršavanja sistema, koji presreću pozive definisanih metoda i zatim preusmeravaju te pozive ka ciljnim metodama.

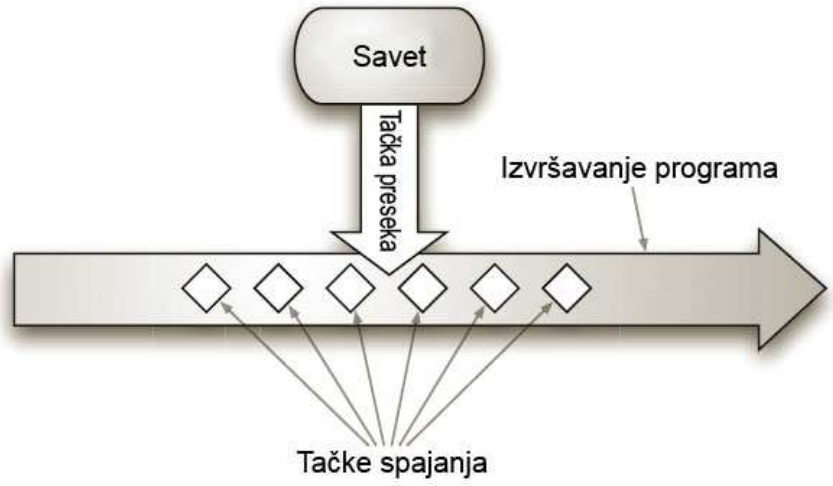


Slika 2.6: Proxy objekat koji okružuje ciljni objekat, presreće pozive metoda i izvršava dodatnu logiku pre pozivanja metode na ciljnom objektu.

2.6 AOP terminologija

Kao i većina tehnologija, AOP je formirao sopstveni žargon. Pojmovi uvek prisutni u diskusiji o AOP-u jesu tačke spajanja, tačka preseka i savet[1].

1. **Tačke spajanja** (j*oin points*) su jedinstveno prepoznatljive tačke u toku izvršavanja sistema. Primer takvih tačka su izvršavanje metode, instanciranje objekta, ili bacanje izuzetaka. Tačke spajanja su prisutne u svakom sistemu, nisu vezane za AOP.
2. **Tačka preseka** (*pointcut*) je konstrukcija kojom se odredjuju tačke spajanja koje zadovoljavaju odredjeni kriterijum. Na primer, za aspekt za logovanje interesantni su pozivi svih javnih metoda u sistemu.
3. **Savet** (*advice*) je konstrukcija koja menja ponašanje programa. Kada tačka preseka izabere odgovarajuće tačke spajanja na scenu dolazi savet aspekta, koji sadrži dodatnu logiku koja se se umeće i modifikuje ponasanje modula. Savet umeće dodatno ponašanje pre, posle ili okružuje selektovanu tačku spajanja. Savet je forma **dinamičkog preplitanja** (*dynamic crosscutting*) jer utiče na ponašanje programa u toku njegovog izvršavanja. Na primer, u implementaciji logovanja, savet je zadužen da zabeleži ulazak u svaku javnu metodu.
4. Pored dinamičkog, postoji i **statičko preplitanje** koje omogućava promenu strukture samog sistema. Upotrebom “medju-tipovne deklaracije” (*inter-type declaration*), moguće je dodati u definiciju klase novu promentljivu.
5. **Aspekt** je konstrukcija u kojoj se izražavaju sve prethodno pomenute konstrukcije. Cilj AOP-a je da postoji modul koji enkapsulira svu isprepletanu logiku, a aspekt je mesto gde se definise ta logika. Aspekt sadrži tačke preseka, savete i konstrukcije statičkog preplitanja.



Slika 2.7: Funkcionalnost aspekta (savet) se umeće u toku izvršavanja programa u jednoj ili više tačaka.

Svaka AOP implementacija mora da sadrži model tačaka preseka (čine ga tačke spajanja i tačke preseka), jer je to centralna konstrukcija oko koje se sve gradi. Medjutim, ne mora svaka implementacija da podržava sve ostale delove generičkog modela. Kao sto je već pomenuto Spring AOP akcenat stavlja na kombinovanje isprepletanih i centralnih dužnosti sistema u toku izvršavanja programa i ne podržava statičko preplitanje.

2.7 Istorija AOP-a

Godinama unazad, teoretičari se slažu da je najbolji način za kreiranje robusnog sistema, a opet lakog za razumevanje i održavanje, identifikacija i razdvajanje dužnosti koje sistem mora da ispuni. Ova tema je bolje poznata kao “***separation of concerns***” ili ukratko *SoC*[14] (termin koji

[1] Detaljnije o svakom terminu u poglavljima 3.1, 3.2, 3.3, 3.4.

je u nauku o programiranju uveo poznati naučnik E.W.Dijkstra[15]). U radu izdatom 1972. David Parnas je dao predlog za najbolji način postizanja SoC: modularizacija - proces kreiranja modula koji sakrivaju svoje odluke medjusobno. OOP je pružilo moćan način za razdvajanje centralnih dužnosti sistema. Medjutim, nije se pokazalo tako dobro kod isprepletanih dužnosti. Nekoliko metodologija se pojavilo kao rešenje za modularizaciju isprepletanih dužnosti: meta-programiranje, reflektivno prigramiranje, subjekt orjentisano, adaptivno programiranje, aspektno orjentisano, itd. Velika zastupljenost AOP-a u razvoju novih aplikacija sa primenom u svakodnevnom životu, pokazuje da je AOP izraslo u najpopularnije rešenje.
Veliki deo istraživanja koje je dovelo do pojave AOP-a sprovedeno je u naučnim institucijama. *Cristina Lopes* i *Gregor Kiczales* iz *Paolo Alto Research Center* su jedni od prvih koji su radili na razvoju AOP-a. Gregor[5] je takodje i započeo rad na AspectJ, prvoj implementaciji AOP-a.

2.8 Prednosti i mane AOP-a

Uvodjenje AOP-a dovodi do novog nivoa apstrakcije u programiranju. Apstrakcija smanjuje kompleksnost posmatranog problema deleći ga u izolovane celine – module. S obzirom da svaki modul predstavlja mali deo celog sistema, kompleksnost logike sadržane u modulu je smanjena na nivo koji programer može lakše da shvati. Ali apstrakcija uvedena AOP-om dolazi uz odredjenu cenu:

- Uvodjenje nove apstrakcije (aspekata) u sistem je posao koji zahteva visok nivo znanja i iskustva. Programeri moraju da primene odredjene dekompozicione tehnike da razdvoje centralne i isprepletane dužnosti sistema na pravi način.
- Apstrakcija po svojoj prirodi krije detalje implementacije. U software-skim sistemima veći nivo apstrakcije uvek znači manje informacija o samoj implementaciji. Drugim rečima samo gledanje koda ne može se preslikati na ono sto ce se dešavati u trenutku izvršavanja (dobar primer toga jesu polimorfne funkcije u OOP, ne može se znati koja funkcija će se izvršiti u trenutku kompajliranja). AOP uvodi još veću kompleksnost u praćenju toka izvršavanja programa (ne može se znati koja akcija će se izvršiti u odredjenom trenutku, gledajući samo kôd koji implementira centralnu dužnost modula).

Pomenute mane su ipak mala cena koju treba platiti zarad prednosti koje donosi AOP:

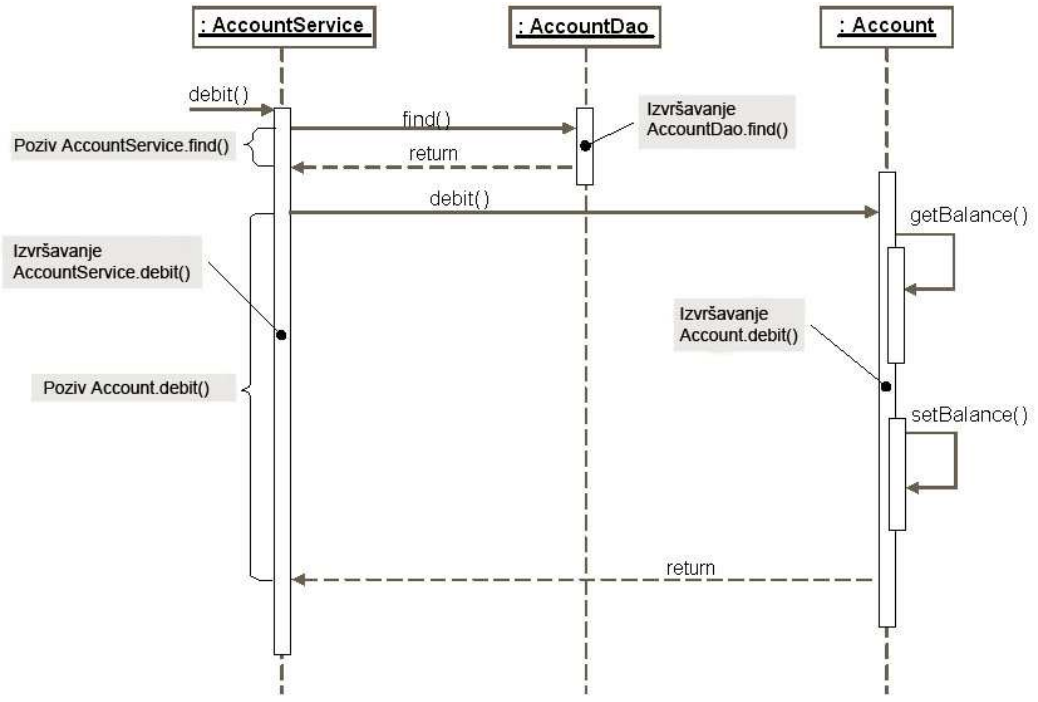
- Precizno definisane obaveze svakog modula - modul više nije odgovoran za ispunjavanje isprepletanih duznosti sistema.
- Viši nivo modularizacije - AOP pruža mehanizam za izolaciju svih dužnosti sistema (čak i isprepletanih) uz minimalnu medjusobnu vezu. Rezultat je sistem koji sadrži manje dupliranja koda i koji je lakši za razumevanje i održavanje.
- Lakša evolucija sistema - uvodjenje nove funkcionalnosti u sistem se svodi na uvodjenje novog aspekta i ne zahteva menjanje centralnih modula. Obrnuto, aspektni kompajler će postojeće aspekte umetati u svaki novo-uvedeni centralni modul.
- Bolja iskorišćenost koda - bolja podela dužnosti i labava povezanost dovodi do bolje iskorišćenosti koda. AOP implementira svaku dužnost u posebnom aspektu i svaki modul je vezan za sistem slabijom vezom nego koristeći konvencionalne tehinke. Precizinije rečeno, veza izmedju različitih modula je prisutna samo u specifikaciji pravila za kombinovanje isprepletanih dužnosti sistema.

3. AspectJ

AspectJ[5] je aspektno-orjentisana ekstenzija programskog jezika Java. Sastoji se od specifikacije i implementacije programskog jezika. Specifikacija definiše sintaksu i semantiku jezika kojim se piše kôd. Implementacija AspectJ-a uključuje aspektni tkalac koji može biti u obliku kompajlera i linkera. Tkalac proizvodi bajtkod koji je u skladu sa Java specifikacijom omogućavajući svakoj Java virtuelnoj mašini (JVM) da izvršava taj kôd.
AspectJ je nastao i razvijao se u početku kao poseban jezik sa novim rezervisanim rečima i posebnim kompajlerom za prevodjenje tog jezika na bajtkod (tradicionalna sintaksa). Medjutim, sa pojavom Jave 5 i uvodjenjem anotacija kao sredstvo za izražavanje meta-podataka, razvila se alternativna sintaksa za izražavanje aspektno orjentisanih konstrukcija (@AspectJ sintaksa). Dizajn modela tačaka preseka AOP sistema omogućava programerima da pišu robusne i lako održive sisteme ograničavajući pristup samo nekim tačkama spajanja. Jezik korišćen u AspectJ-u za izražavanje tačaka preseka je sofisticiran i omogućava selektovanje tačaka spajanja na osnovu strukturnih informacija kao što su tip objekta, ime objekta, argumenti, prisutne anotacije, ali uzimajuću u obzir i kontrolni tok programa (*control flow*).

3.1 Tačke spajanja *(Join Points)*

Tačka spajanja je svaki trenutak u izvršavanju programa koji se može jedistveno indentifikovati. Poziv metode ili pristup promenljivoj su primeri tačke spajanja. Na slici 3.1 prikazan je sekvencijalni UML dijagram bankarske transakcije i obeležene su tačke spajanja gde postoji mogućnost uvodjenja isprepletanog ponašanja.



Slika 3.1: Tačke spajanja u izvršavnju programa (poziv i izvršavanje metode su najčešće korišćene tačke spajanja).

Možemo videti više tačaka spajanja u trenutku poziva *debit()* metode na *AccountService* objektu. Prva tačka spajanja jeste poziv *debit()* metode, a odmah zatim i izvršavanje iste metode. U toku izvršavanja nailazimo na nove tačke spajanja u vidu poziva i izvršavanja *find()* metode na *AccountDao* objektu i *debit()* metode na *Account* objektu. Poziv i izvršavanje metoda nisu jedine tačke spajanja, dodeljivanje vrednosti promenljivoj (npr. u toku izvršavanja *setBalance()* metode na *Account* objektu) je takodje tačka spajanja.

Kategorizacija tačaka spajanja u AspectJ[5]:

- Na metodama
  Najčešće korišćene tačke spajanja. AspectJ podržava dve vrste tačaka spajanja na metodama: **poziv** i **izvršavanje** metode. Izvršavanje metode obuhvata izvršavanje koda unutar tela funkcije. Drugim rečima to znači da se kôd može umetati pre, posle ili i pre i posle tela metode. Poziv metode se dešava na mestima gde se metoda poziva, tj. unutar nekog druge programske konstrukcije iz koje se poziva metoda. U većini slučajeva razlika izmedju tačke spajanja pri izvršavanju i pozivu metode je zanemarljiva. Najbitinija razlika se ogleda u mestu gde tkalac umeće savet. U slučaju korišćenja izvršavanja metode tkalac umeće savet unutar tela metode, dok prilikom poziva metode tkalac umeće savet svuda gde se poziva ta metoda. Iz ovoga sledi da kad god je moguće treba potencirati umetanje koda prilikom izvršavanja metoda.

- Na konstruktorima
  Slično kao u slučaju tačaka spajanja na metodama samo što predstavljaju izvršenje i poziv konstruktora.

- Prilikom pristupa promenljivima
  AspectJ podržava umetanje isprepletanog koda prilikom **čitanja** i **pisanja** klasne ili promenljive koja pripada instaci klase (ne podržava pristup lokalnim promenljivima).

- Prilikom obradjivanja izuzetaka
  AspectJ nudi mogućnost procesiranja izuzetaka tako što podržava tačke spajanja koje ustvari predstavaljaju *catch* blok u *try/catch* konstrukciji.

- Prilikom inicijalizacije objekata
  U inicijalizaciju ubrajamo više podkategoraija: tačke spajanja u trenutku **inicijalizacije klasa** (učitavanje klasa u Java virtuelnu mašinu (JVM), inicijalizacija statičkih promenljivih, izvršavanje statičkih blokova), tačke spajanja u trenutku **inicijalizacije objekata** (obuhvataju trenutak izmedju kraja izvršavanja konstruktora nadklase do kraja izvršavanja prvog pozvanog konstruktora), tačke spajanja u trenutku **preinicijalizacije objekata** (obuhvata prostor izmedju zvanja konstruktora i poziva konstruktora nadklase).

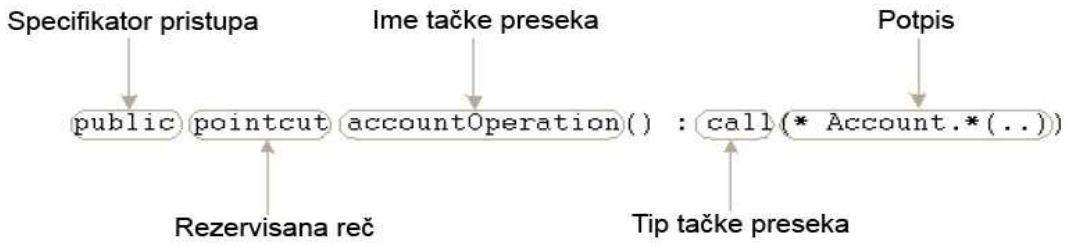
- Prilikom izvršavanja saveta
  Kategorija koja obuhvata trenutak izvršavanja bilo kog već prisutnog saveta u sistemu.
  Omogućava pisanje saveta kojima se mogu pratiti ponašanja saveta u celokupnom sistemu.

3.2 Tačke preseka *(Pointcuts)*

Tačka preseka je programska konstrukcija koja vrši selekciju tačaka spajanja i čuva celokupni kontekst odabrane tačke spajanja[5]. Klasa, interfejs, metod i promenljive su jedinstveno odredjeni svojim potpisom. Tačka preseka koristi ove potpise da bi opisala selektovanu tačku spajanja. Često sam potpis nije dovoljan za jedinstveno indentifikovanje tačke spajanja. AspectJ nudi tačke preseka koje koriste informacije o toku izvršavanja programa, kao što su celokupan kontrolni tok programa (stek poziva metoda) koji je doveo do tačke spajanja. Kontekst tačke spajanja su informacije o toku izvršavanja programa asocirane uz svaku tačku spajanja. Na primer, u trenutku poziva metode kontekst podrazumeva objekat koji poziva metodu, objekat nad kojim se poziva metoda (*this*), argumente, i anotacije nakačene na pozvanu metodu.

3.2.1 AspectJ jezik za izražavanje tačaka preseka

AspectJ koristi isti jezik za izražavanje tačaka preseka u tradicionalnoj i @AspectJ sintaksi. Tačka preseka se može deklarisati unutar aspekta, klase ili interfejsa.
Primer definicije tačke preseka koja selektuje sve metode unutar *Account* klase dat je na slici 3.2:



Slika 3.2: Definicija tačke preseka. Sastoji se od rezervisane *pointcut* reči, imena i kriterijuma za selekciju tačaka spajanja.

Tačka preseka se definiše korišćenjem rezervisane reči *pointcut*. Ime tačke preseka se dodeljuje da bi definisana tačka mogla da se koristi kasnije u definiciji saveta aspekta. Deo nakon dvotačke je izraz koji selektuje tačke spajanja koristeći tip tačke preseka i potpis programskih elemenata.

```
before() : accountOperation() {
        … telo saveta
}
```

AspectJ podržava Java logičke operatore (!, ||, &&) u svom jeziku za izražavanje tačaka preseka čime omogućava konstrukciju kompleksnih pravila za odabir tačaka spajanja kombinujući više jednostavnih tačaka preseka.
Centralna konstrukcija u definiciji tačke preseka je potpis programskog elementa. S obzirom da ispreplетane dužnosti sistema, po samoj definiciji, prožimaju više modula, jezik za izražavanje tačaka preseka mora pružiti ekonomičan način za izražavanje selekcionih kriterijuma. AspectJ koristi džoker elemente (*wildcards*) za selekciju elementa koji dele zajedničke karateristike:

- `*` jedan ili više karaktera izuzimajući tačku
- `..` jedan ili vise karatera uključujući tačku
- `+` bilo koji podtip datog tipa

U zavisnosti od vrste potpisa programskog elementa ovi znaci dobijaju uža značenja.

3.2.1.1 Sintaksa potpisa programskih elemenata

1. Šabloni za izražavanje potpisa tipa

U AspectJ termin tip obuhvata klase, interfejse, primitivne tipove i aspekte. U ovim potpisima specijalni znaci dobijaju uže značenje:

- `*` deo imena tipa ili paketa
- `..` svi direktni ili indirektni podpaketi
- `+` obeležava podtipove

Primer 3.1:

`*Account` - bilo koji tip čije se ime završava sa `Account`
`java.*.Date` - tip `Date` u bilo kom direktnom java podpaketu
`java..*` - bilo koji tip u java paketu ili bilo kom direktnom ili indirektnom podpaketu
`@Entity Account` - tip `Account` sa `Entity` anotacijom
`*<Account>` - bilo koji tip čiji je parameter `Account` tip
`@Named*Query User+` - tip `User` uključujući njegove podtipove koji su obeleženi anotacijom čije ime počinje sa `Named` i završava sa `Query`
`@(Secured || Sensitive)*` - bilo koji tip koji ima ili `@Secured` ili `@Sensitive` anotaciju

2. Šabloni za izražavanje potpisa metoda i konstruktora
Potpis metode, odnosno konstruktora je odredjen imenom, tipom koji vraćaju[1] (samo za metode), tipom na kojem je deklarisana, tipovima argumenta, definisanim nivoom pristupa i prisutnim anotacijama.
Primer 3.2:

`public void Account.set*(*)` - bilo koji javni metod definisan na `Account` tipu, čije ime počinje sa set, ne vraća rezultat i prima samo jedan argument bilo kog tipa.
`* File.*(..)` - svaki metod definisan na `File` tipu, koji prima 0 ili više argumenta i vraća rezultat bilo kog tipa.
`@Transactional * *(..)` - bilo koji metod označen sa `@Transactional` anotacijom.
`!public * Account.*(..)` - bilo koji ne-javni metod na `Account` tipu.
`* *(..) throws SQLException` - svaki metod koji je deklarisan da baca izuzetak tipa `SQLException`.
`* (@ThreadSafe *).*(..)` - svaki metod definisan na tipu koji je označen `@ThreadSafe` anotacijom.

3. Šabloni za izražavanje potpisa promenljive
Slično kao i u prethodnim slučajevima, potpis promenljive se koristi za selekciju tačaka spajanja koje odgovaraju čitanju ili pisanju promenljivih.
Primer 3.3:

`private float Account.balance` – privatna promenljiva `balance` tipa `float` definisana na `Account` tipu.
`* User.*` - bilo koja promenljiva definisana na `User` tipu
`private !static !final @Id * *.*` - privatna, ne-statička i ne-finalna promenljiva definisana na bilo kom tipu, koja je označena `@Id` anotacijom.

3.2.1.2 Podela tačaka preseka

Postoje dva načina na koji se tačka preseka uparuje sa tačkom spajanja u AspectJ[5]:

- direktne tačke preseka – koje se direktno uparuju sa kategorijama tačaka spajanja koje smo spominjali u 3.1.
- indirektne tačke preseka – koje se ne mogu direktno upariti sa određenim tačkama spajanja iz 3.1, već selektuju tačke spajanja koristeći dodatne infomacije o tački spajanja, kao sto su tipovi objekata u trenutku izvršavanja programa, programski tok (*control flow*) i leksički domen.

3.2.1.2.1 Direktne tačke preseka

Direktne tačke preseka prate odredjenu sintaksu da bi selektovali bilo koju vrstu podržanih tačaka spajanja u AspectJ-u.

Kategorija tačaka spajanja	Sintaksa tačke preseka
Izvršavanje metode	`execution(Potpis metode)`
Poziv metode	`call(Potpis metode)`
Izvršavanje konstruktora	`execution(Potpis konstruktora)`
Poziv konstruktora	`call(Potpis konstruktora)`
Inicijalizacija klase	`Staticinitialization(Potpis tipa)`
Čitanje promenljive	`get(Potpis tipa)`
Pisanje promentljive	`set(Poptis tipa)`
Procesiranje izuzetaka	`handler(Potpis tipa)`
Inicijalizacija objekta	`initialization(Potpis konstruktora)`
Pre-inijalizacija objekta	`preinitialization(Potpis konstruktora)`
Izvršavanje saveta	`adviceexecution()`

[1] Za razliku od potpisa metoda u Java jeziku, u potpis metode izraženom pomoću AspectJ jezika za definisanje tačaka preseka i povratni tip pripada potpisu.

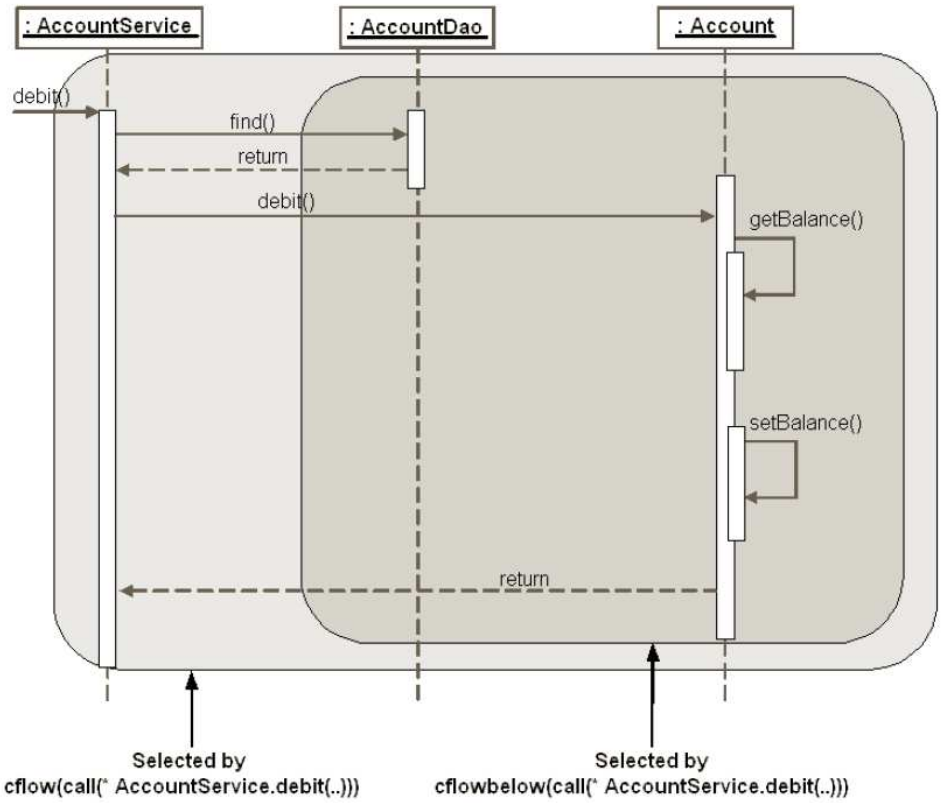
3.2.1.2.2 Indirektne tačke preseka

Indirektne tačke preseka odredjuju tačke spajanja po nekom dodatnom kriterijumu, ne samo potpisu tačke spajanja. AspectJ nudi indirektne tačke preseka koje se zasnivaju na kontrolnom toku programa, leksičkoj strukturi, objektu nad kojim se trenutno izvršava program, argumentima, anotacijama i uslovnim izrazima.

1. Tačke preseka na bazi kontrolnog toka programa

Ove tačke preseka selektuju tačke spajanja na osnovu programskog toka u drugim tačkama spajanja odabranim od strane druge tačke preseka (programski tok tačke spajanja se može posmatrati kao tok programskih instrukcija koje nastaju kao rezultat pozivanja te tačke spajanja. Npr. *Account.debit()* metoda poziva *Account.getBalance()*, pa kazemo da se poziv *Account.getBalance()* desio unutar *Account.debit()* programskog toka).Postoje dve vrste tačaka preseka u AspectJ i oba zahtevaju drugu tačku preseka kao argument.

- *cflow(<Tačka preseka>)* selektuje sve tačke spajanja u kontrolnom toku specifirane tačke preseka uključujući i tačke spajanja koje odgovaraju datoj tačke preseka. *cflowbelow(<Tačka preseka>)* selektuje iste tačke spajanja kao i *cflow()*, izuzimajući tačke spajanja koje odgovaraju datoj tački preseka.



Slika 3.3: Tačke preseka na bazi kontrolnog toka programa (*cflow, cflowbelow*) i tačke spajanja koje selektuju na sekvencijalnom UML dijagramu.

2. Tačke preseka zasnovane na leksičkoj strukturi

Leksičke tačke preseka odabiraju tačke spajanja unutar leksičkog domena date klase, aspekta, metode. Postoje dve ovakve konstrukcije u AspectJ-u:

- *within(<Potpis tipa>)* selektuje bilo koju tačku spajanja unutar tela date klase ili aspekta
- *withincode(<Potpis metode>)* selektuje bilo koju tačku spajanja untar leksičke strukture metode ili konstruktora.

3. Tačke preseka zasnovane na tipu objekata pri izvršavanju programa

- *this(<Tip>)* selektuje tačke spajanja koje imaju asocirani *this* objekat datog tipa (tačke spajanja gde *this instaceof Tip* izraz vraća *true*).
- *target(<Tip>)* slično kao i *this()*, ali ispituje se objekat nad kojim se obavlja operacija, ciljni objekat (npr. objekat nad kojim se poziva metoda).

4. Tačke preseka na bazi argumenta

- *args(<Tip>)* poredi tip argumenta u toku izvršavanja sa datim tipom u slučaju metode ili konstruktora, sa bačenim izuzetkom u slučaju procesiranja izuzetaka, u slučaju pristupa promenljivoj radi pisanja poredi sa tipom nove vrednosti.

5. Tačke preseka zasnovane na uslovnim izrazima

Selektuju tačke spajanja na osnovu nekih provera u tački spajanja. Oblika su *if(Logički izraz)*.

3.3 Savet (*Advice*)

Savet je konstrukcija slična metodi pomoću koje AspectJ realizuje isprepletane dužnosti sistema. Savet definiše akciju koja se izvršava u tačkama spajanja određenim pomoću tačke preseka. Detalji definicije saveta se razlikuju od klasifikacije (sekcija 3.3.1), ali se sastoji od tri osnovna dela:

- specifikacije da li će se izvršavati pre, posle ili će okruživati izvršavanje tačke spajanja.
- definiše koje tačke spajanja će biti savetovane.
- telo saveta sadrži kod koji će se izvršavati prilikom izvršavanja savetovane tačke spajanja.

3.3.1 Klasifikacija saveta

AspectJ podržava tri vrste saveta:

- "pre" savet (*before advice*)– izvršava se pre izvršavanja tačke spajanja.
  Primer 3.4:

```
//Savet vrši autorizaciju pre izvršenja bilo koje metode sa
//@Secure anotacijom
before() : execution(@Secure * *(..)) {
      // autorizacija
}
```

- "posle" savet (*after advice*) – izvršava se nakon izvršavanja tačke spajanja. Postoje tri varijacije u zavisnosti od rezultata izvršavanja:
  - "uvek posle" (*after finally*) – izvršava se nakon izvršavanja tačke spajanja nezavisno od rezultata akcije.
    Primer 3.5:

```
//Savet će se izvršavati nakon svake metode u Account
//klasi, nezavisno od rezultata metode
after() : call(* Account.*(..)) {
      // loguj povratnu vrednost
}
```

  - "posle uspešnog povratka" (after returning) – izvršava se nakon uspešnog izvršavanja akcije, tj. bez pojavljivanja greske.
    Primer 3.6:

```
after() returning: call(* Account.*(..)) {
      // loguj uspesno izvrsenje
}
```

    AspectJ nudi mogućnost upotrebe povratne vrednosti savetovanog metoda

Primer 3.7:

```
after() returning(Account account) :
            call(AccountDao.find(..)) {
      // operacija nad Account objektom
}
```

Ovaj savet se neće primeniti nad svim *AccountDao.find()* metodama već samo na onim koje vraćaju *Account* objekat.

- "posle greške" (*after throwing*) – izvršava se nakon neuspelog izvršavanja akcije, tj. bacanja izuzetka.
  Primer 3.8:

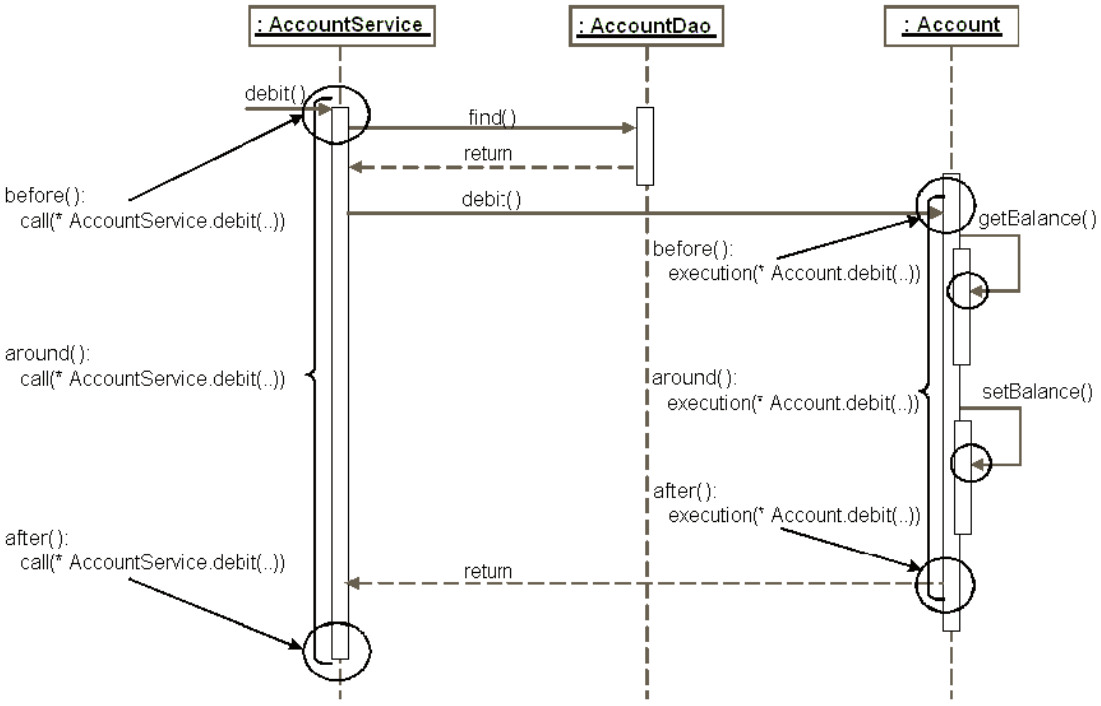
```
after() throwing : call(* Account.*(..)) {
      // operacija nakon pojave greške
}
```

Kao i u prethodnom slučaju postoji mogućnost korišćenja bačenog izuzetka.
Primer 3.9:

```
after() throwing(Throwable ex) : execution(* *(..)) {
      // loguj izuzetak
      loger.error(ex);
}
```

- "okružujući" savet (*around advice*) – okružuje izvršavanje tačke spajanja. Ovaj savet je specifičan jer ima mogućnost da izvrši originalnu akciju sa ili bez menjanja konteksta nula ili više puta.
  Tipična upotreba ovog saveta je da se obavi neka dodatna logika pre izvršavanja same tačke spajanja, da se prespoji (preskoči) izvršavanje tačke spajanja i primeni neka alternativna logka, ili obuhvati izvršavanje tačke spajanja sa *try/catch* blokom, transakcijom itd… Unutar okružujućeg saveta mora se pozvati specijalna rezervisana reč *proceed()* da bi se savetovana tačka spajanja izvršila.
  Primer 3.10:

```
void around(Account account, float amount)
      : call (void Account.debit(float) throws BalanceExceptin)
      && target(account)
      && args(amount) {
      try {
            proceed(account, amount);
      }catch(BalanceException ex) {
            // loguj grešku
      }
}
```



Slika 3.4: Tačke u programskom toku koje aspekti mogu "savetovati". Svaki kružić na sekvencijalnom dijagramu predstavlja mogućnost umetanja "pre" i/ili "posle" saveta, a tok izmedju dva odgovarajuća kružića predstavlja mogućnost za „okružujući" savet.

3.4 Statičko preplitanje

Tačke preseka i saveti zajedno čine **dinamička pravila** za umetanje koda. Statička pravila za umetanje koda se izražavaju u vidu medju-tipovne deklaracije (*inter-type declaration*) i upozorenjima/greškama prilikom kompajliranja (*compile-time warnings/errors*).

3.4.1 Među-tipovne deklaracije

Među-tipovna deklaracija je isprepletana konstrukcija koja menja statičku strukturu klase, interfejsa ili aspekta. Omogućava dodavanje nove promenljive u definiciju klase, implementaciju dodatnog interfejsa u definiciji klase ili dodavanje nove anotacije.
Primer 3.11:

```
private long AccessTracked.lastAccessTime
declare parents: CacheStatistics implements Observer
declare @method: public * Account.*(..):
@Secure(permission="adminOperation")
declare @field: private  * (@Entity *).id:
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

3.4.2 Upozorenja/greške prilikom kompajliranja

Omogućava definisanje dodatnih upozorenja i grešaka koje će kompajler prijaviti u slučaju ispunjenja postavljenih kriterijuma.

Primer 3.12:

```
declare error : callToUnsafeCode(): "This will result in crash";
declare warning : callToDaoLayer(): "Please ensure all calls to
data access layer are performed via service layer";
```

3.5 AspectJ tkalac (Weaver)

Pošto se bajtkod proizveden pomoću AspectJ kompajlera mora izvršavati na JVM mora se pridržavati specifikacije Java bajt koda. To znači da se AOP konstrukcije moraju preslikati u Java konstrukte. Sledi pojednostavljen pregled transformacije AspectJ koda u čist Java kôd:

- Aspekti se preslikavaju u klase (uglavnom singlton, ali postoje i drugačije asocijacije[1]), svaki element i metod aspekta postaje i element klase.
- Tačke preseka se preslikavaju u metode (bez tela). Mogu imati pridružene pomoćne metode koje olakšavaju odabir tačaka spajanja u toku izvršavanja koda.
- Savet se mapira u jednu ili više metoda. Tkalac umeće pozive ovih metoda na mestu koje odgovara pridruženoj tački preseka.

3.5.1 Mehanizimi umetanja koda

Da bi aspekti zadovoljili isprepletane dužnosti sistema, neophodno je da tkalac “uplete” klase i aspekte. AspectJ podržava tri načina preplitanja koda[5]:

- Umetanje izvornog koda
  Ulaz u tkalac su izvorni kôd klasa i aspekata. Tkalac u ovom slučaju ima ulogu kompajlera, procesira izvorni kôd i proizvodi isprepleten bajtkod.
- Umetanje binarnog koda
  U ovom slučaju ulaz u tkalac su klase i aspekti u bajtkod formi, tj. pre ulaska u tkalac klase i aspekti su odvojeno kompajlirane. Može se reći da tkalac ima ulogu *linkera* u ovom modu, koristeći kompajliran kôd proizvodi novi, isprepletan, binarni kôd.
- Umetanje prilikom učitavanja
  Kao u slučaju binarnog umetanja, ulaz su kompajlirane klase i aspekti (u bajtkodu). Aspekte umeće specijalni agent u trenutku učitavanja klasa (*load-time agent*), koji može imati više oblika: posaban učitavač klasa (*classloader*), predprocesor klasa u aplikacionom serveru ili samoj virtuelnom mašini.

[1] AspectJ podržava 4 vrste asocijacija aspekata: singlton(osnovni), po objektu, po programskom toku i po tipu.

4. Objektno Orjentisani Dizajn (OOD) i AOP

Projektni obrasci opisuju probleme koji se često sreću u razvoju softvera i nude fleksibilna rešenja tih problema koja se mogu primeniti u mnogim situacijama. Svaki obrazac definišu **problem** (opisuje kada se primenjuje obrazac), **rešenje** (opisuje elemente koji čine dizajn, njihove medjusobne odnose i obaveze) i **posledice** (rezultati primene obrasca i ustupke učinjene pri tome)[3]. Kada su indentfikovani projektni obrasci smatrani su remek delom objektno orijentisanih jezika. Medjutim, sa pojavom radova [7,8] koji su pokazivali da jezik koji se koristi za implementaciju obrasca utiče na njegovu implementaciju, počeli su da se istražuju efekti aspektno orjentisanih tehnika na implementaciju objektno orjenitsanih obrazaca.
Projektni obrasci dodeljuju **uloge** (*role*) svojim učesnicima, primer toga su *Subject* i *Observer* u *Observer* ili *Component*, *Leaf* i *Composite* u *Composite* obrascu [3]. Uloge definišu funkcionalnosti učesnika u kontekstu obrasca. Istraživanjem je pokazan značajan napredak u modularnosti implementacija većine projektnih obrazaca korišćenjem AOP tehinka. Najveći napredak je postignut u slučajevima gde je ispripletena struktura izmedju uloga i klasa koje učestvuju u obrascu.

4.1 Isprepletana struktura projektnih obrazaca

Isprepletanost u struturi projektnog obrasca je uzrokovana različitim vrstama uloga koje učestvuju u obrascu i njihovom interakcijom sa klasama prisutnim u obrascu.
Uloge mogu biti **definišuće**[4], što znači da učesnici u obrascu nemaju funkcionalnost izvan one koja ga definiše u obrascu, tj. da uloge kompletno odredjuju učesnike u obrascu. Na primer, objekti koji imaju *Facade* ulogu [3] pružaju unifroman pristup nekom podsistemu i uobičajno je da nemaju nikakvu drugu ulogu. Druga vrsta uloga su **nadogradjene** (*superimposed*)[4], koje su dodeljene klasama koje imaju funkcionalnosti i obaveze van domena obrasca. Na primer, u *Observer* obrascu klase koje imaju ulogu subjekata (*Subject)* i posmatrača (*Observer)* rade više od pukog ispunjavanja obaveze koje im nameće obrazac.
U objektno orijentisanom programiranju definišuće uloge su realizivane uglavnom nasledjivanjem apstraktne bazne klase, čime se postiže različito, ali srodno ponašanje; nadogradjene uloge se realizuju interfejsima koji definišu ponašanje i zaduženja klase.
U slučaju definišućih uloga, svaka jedinica apstrakcije (klasa) predstavlja jedinstvenu funkcionalnost sistema, koja odgovara ulozi u obrascu i klasa se prirodno uklapa u svoju poziciju u sistemu kao deo obrasca. Kod nadogradjenih uloga, učesnici imaju zaduženja izvan konteksta projektnog obrasca i umetanjem klase sa takvom ulogom u kontekst obrasca dobijamo modul koji u najboljem slučaju implementira dve dužnosti sistema – originalnu i specifičnu za obrazac u kom se koristi[4]. U ovakvim slučajevima modularnost je kompromitovana i upotrebom AOP jezika moguće je izdvojiti dužnosti specifične za obrazac u poseban modul – aspekt.

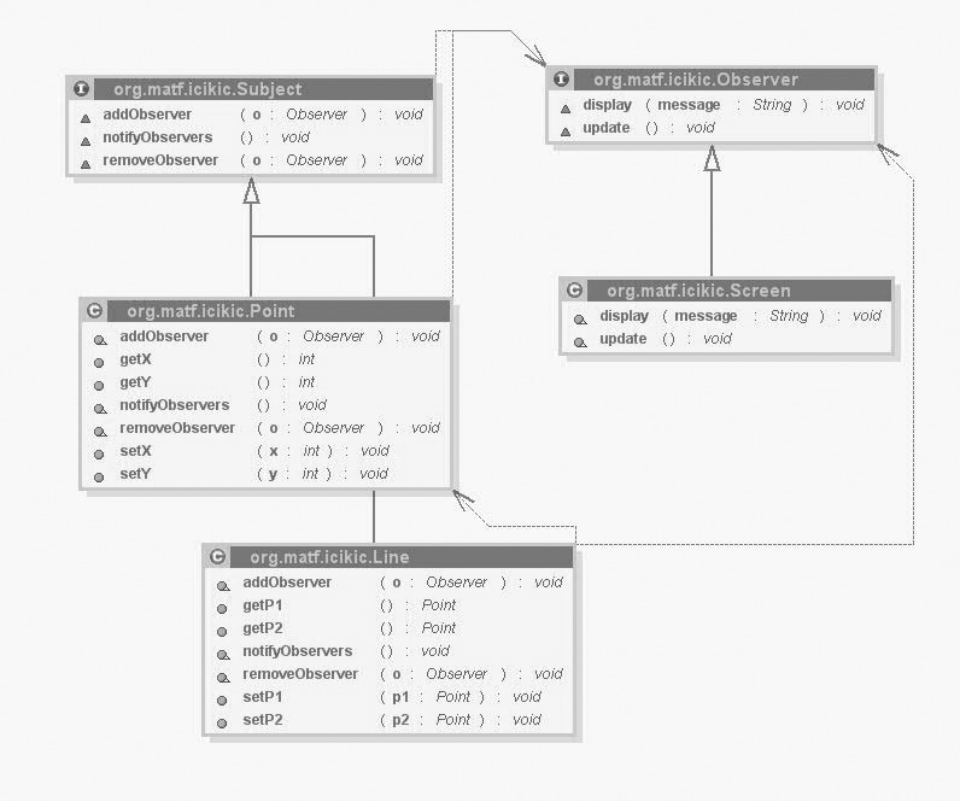
4.2 Izazovi u implementaciji projektnih obrazaca

Tri najvažnija izazova u realizaciji projektih obrazaca su vezana za implementaciju, dokumentaciju i kompoziciju[4].
Implementacija projektnog obrasca vuče više nepoželjnih efekata. Pošto obrazac utiče na strukturu sistema, a njegova implementacija zavisi od strukture sistema[2], implementacija obrasca često je prilagodjena objektu koji ga koristi. Ovo uzrokuje gubitak modularnosti i teškoće u razlikovanju obrasca, objekta u kom se primenjuje, i celog objektnog modela[6]. Dodavanje ili sklanjanje projektnog obrasca u/iz sistema često je invazivan posao. Posledica ovoga je da iako je obrasac ponovno upotrebljiv, sama implementacija nije.
Zbog invanzivne prirode implementacije projektnih obrazaca i preplitanja i rasipanja koda koje se pojavljuje dokumentovanje obrazaca postaje teško izvodljivo.
Kompozicija obrazaca podrazumeva upotrebu više obrazaca u istim klasama, što proizvodi veliki broj čvrsto vezanih klasa. Ovo je važan problem jer neki projektni obrasci eksplicitno koriste druge u svojim rešenjima.

4.3 Primer korišćenja AspectJ-a u implementaciji projektnih obrazaca: *Observer* obrazac

Namena *Observer* obrasca je da definiše “jedan ka više” vezu izmedju objekata tako da kada jedan objekat promeni stanje, svi zavisni objekti budu obavešteni i ažurirani automatski[3].

Objektno orjentisana implementacija *Observer* obrasca obično podrazumeva definisanje dodatne promenljive u svim subjektima u kojima se čuva lista *Observer*-a zainteresovanih za taj subjekt. Kada subjekt želi izvestiti *Observer*-e o promeni stanja, poziva *notify()* metod, koji poziva *update()* metod na svakom *Observer*-u u listi.



Slika 4.1: Dijagram klasa jednostavne implementacije *Observer* obrasca. Tačke, odnosno linija imaju ulogu subjekta koje obaveštavaju o svojoj promeni ekran (koji ima ulogu observera) radi ponovnog iscrtavanja.

U sistemu pokazanom na slici, *Observer* obrazac je zadužen da svaka promena na subjektima rezultira ažuriranjem ekrana. Kao što se vidi kôd koji implementira obrazac je resejan po svim *Subject* klasama. Svi učesnici (*Point* i *Line*) moraju da budu svesni svoje uloge u obrascu, tj. sadrže kôd specifičan za obrazac.
Neki delovi strukture *Observer* obrasca su zajednički za sve instance obrasca u sistemu, dok su neki delovi specifični za svaku instancu[3]. Delovi isti za sve instance su:

- Postojanje subjekat i posmatrač uloga (neke klase su subjekti, neke posmatrači)
- Održavanje veze od subjekta ka posmatraču
- Opšta logika ažuriranja (promene na subjektu iniciraju ažuriranje posmatrača)

Delovi specifični za svaku instacu obrasca:

- Koje klase mogu biti subjekti a koje posmatrači
- Skup promena subjekta koji iniciraju promenu posmatrača
- Specifičan način ažuriranja različitih vrsta posmatrača

U nastavku će biti prikazan AspectJ kôd na kome se može primetiti razdvojenost ovih delova strukture. Apstraktni aspekt enkapsulira generalizovane delove strukture obrasca, dok konkretna implementacija aspekta brine o specifičnim delovima strukture svake instance obrasca.

Primer 4.1

```
01 public abstract aspect ObserverProtocol {
02    protected interface Subject { }
03    protected interface Observer { }
04
05    private WeakHashMap<Subject, List<Observer>> _perSubjectObservers;
06
07    protected List<Observer> getObservers(Subject s) {
08        if (_perSubjectObservers == null) {
09          _perSubjectObservers = new WeakHashMap<Subject, List<Observer>>();
10      }
11      List<Observer> observers = _perSubjectObservers.get(s);
12      if (observers == null) {
13          observers = new LinkedList<Observer>();
14          _perSubjectObservers.put(s, observers);
15      }
16      return observers;
17    }
18    public void addObserver(Subject s,Observer o){
19        getObservers(s).add(o);
20    }
21    public void removeObserver(Subject s,Observer o){
22      getObservers(s).remove(o);
23    }
24
25    abstract protected pointcut subjectChange(Subject s);
26
27    abstract protected void updateObserver(Subject s, Observer o);
28
29    after(Subject s): subjectChange(s) {
30        for (Observer o : getObservers(s)) {
31            updateObserver(s, o);
32      }
33    }
```

4.3.1. Subjekat i Posmatrač uloge

Uloge su realizovane preko zaštićenih interfejsova, nazvanih *Subject* i *Observer* (primer 4.1, linija 2-3). Interfejsi su ovde definisani pre svega da bi se omogućila njihova upotreba u kontekstu ovog obrasca, a konkretne implementacije *ObserverProtocol* aspekta će dodeliti uloge odgovarajućim klasama. Definisani su kao zaštićeni jer će biti korišćeni samo u *ObserverProtocol* i njegovim konkretnim implementacijama. Ovi interfejsi su prazni (ne definišu nijednu metodu) jer obrazac ne definiše metode za uloge subjekta i posmatrača. Metode koje bi uobičajeno definisali na subjektu i posmatraču su definisane u aspektu.

4.3.2. Subjekt-Posmatrač veza

Implementacija ove veze u AspectJ kodu je lokalizovan u *ObserverProtocol* aspektu. Realizovan je pomoću heš mape (konkretno *WeakHashMap*) u kojoj je mapirana lista posmatrača za svaki subjekt. Svaka instanca obrasca je predstavljena konkretnim ObserverProtocol podaspektom, pa će svaka instanca imati svoje subjekt-posmatrač mapiranje. Promene u mapiranju se realizuju pomoću *addObserver* i *removeObserver* metoda (primer 4.1, linija 19-23). Da bi objekat O postao posmatrač subjekta S, klijent poziva ove metode na podaspektima. Na primer:

```
CoordinateObserving.aspectOf().addObserver(S, O);
```

Privatni metod *getObservers()* se koristi samo interno da kreira odgovarajuću strukturu podataka (primer 4.1, linija 7-17). U ovakvoj implementaciji *Observer* obrasca struktura podataka koja realizuje mapiranje izmedju subjekta i observera je centralizovano u apstraktnom aspektu. Svaka instanca obrasca ima svoju individualnu strukturu u kojoj čuva veze izmedju subjekta i observera. Alternativa je decentralizovani pristup u kome svaki učesnik u obrascu čuva svoj observere.

4.3.3 Opšta logika ažuriranja

U ponovno upotrebljivom, apstraktnom aspektu, implementiran je samo osnovni koncept logike ažuriranja, tj. subjekti mogu da se promene na takav način da iniciraju ažuriranje svih njihovih posmatrača. Ova implementacija ne definiše šta čini promenu, niti na koji način se posmatrači ažuriraju. Opšta logika ažuriranja u apstraktnom aspektu se sastoji iz tri dela[4]:

- Promene od interesa sistemu se mogu posmatrati kao skup tačaka u izvršavanju programa u kojima subjekat treba da ažurira svoje posmatrače. Posmatrajući ovu definiciju u kontekstu AOP-u prirodno je da se takve tačke odredjuju tačkama preseka. U apstraktnom aspektu, znamo samo da postoje promene koje nas zanimaju, ali ne znamo koje su to promene. Zbog toga je definisana apstraktna tačka preseka *subjectChange* koju treba konkretizovati u specifičnoj instanci pod-aspekta (primer 4.1, linija 25).
- U ponovno upotrebljivom delu implementacije takodje znamo da će posmatrači biti ažurirani ali ne možemo predvideti na koji način će to biti uradjeno. Zbog toga je definisana apstraktna metoda *updateObserver()* koja ce biti konkretizovana za svaku instancu obrasca (primer 4.1, linija 27).
- Na kraju, ponovno upotrebljiv aspekt implementira logiku ažuriranja koristeći već pomenute apstraktne elemente. Ova logika je sadržana u “posle” savetu (primer 4.1, linija 29-33).

4.3.4 Konkretni aspekti u Observer obrascu

Svaki konkretni aspekt definiše jednu vrstu subjekt-posmatrač relacije, tj. jednu instancu obrasca. Podaspekti definišu tri stvari[4]:

- Klase koje imaju uloge subjekta, odnosno posmatrača. Ovo je realizovano upotrebom AspectJ *declare parent* konstrukcije koja dodaje interfejs u definiciju klase, u nameri da dodeli ulogu toj klasi.
- Definišu konceptualne operacije, koje iniciraju ažuriranje observera. Realizovano konkretizacijom *subjectChange* tačke preseka.
- Definišu način na koji se ažuriraju posmatrači. Realizovano konkretizacijom *updateObserver()* metode.

Primer 4.2

```
00 public aspect CoordinateObserver extends ObserverProtocol {
01     declare parents: Point implements Subject;
02     declare parents: Line implements Subject;
03     declare parents: Screen implements Observer;
04
05     protected pointcut subjectChange(Subject s):
06         (call(void Point.setX(int))
07          || call(void Point.setY(int))
08          || call(void Line.setP1(Point))
09          || call(void Line.setP2(Point)) ) && target(s);
10
11     protected void updateObserver(Subject s, Observer o) {
12        ((Screen)o).display("Coordinate change.");
13     }
14 }
```

4.4 Poboljšanja u implementaciji projektnih obrazaca koristeći AOP

Za veliki broj projektnih obrazaca, AOP implementacija se manifestovala sa nekoliko srodnih poboljšanja u modularnosti: loklizacija, ponovna upotrebljivost koda, inverzija zavisnosti, transparentna kompozicija i jednostavno uključivanje obrasca [4].
Poboljšanja na konkretnom primeru (*Observer* obrazac):

- Lokalizacija koda – sav kôd neophodan za implementaciju *Observer* obrasca je izolovan u aspektima. Klase koje učestvuju u obrascu su u potpunosti nesvesne konteksta obrasca i kao posledicu ne postoji veza izmedju učesnika. Potencijalna promena u obrascu je lokalizovana na jednom mestu.
- Ponovna iskorišćenost koda – Centralni deo implementacije obrasca je izvučen u apstraktni aspekt koji je predvidjen za ponovno korišćenje. Za svaku instancu obrasca potrebno je definisati konkretan podaspekt.

- Transparentna kompozicija – Obzirom da implementacije učesnika u obrascu nisu vezane za obrazac, ako subjekat ili posmatrač stupi u nove (višebrojne) subjekat-posmatrač odnose njihov kod se ne komplikuje i instance obrasca se ne mogu izgubiti u kompoziciji.
- Jednostavno uključivanje – Subjekat i pasmatrač nisu svesni svoje uloge u instanci obrasca, tako da je moguće isključiti, odnosno uključiti obrazac, u sistem, bez dodatnih promena nad učesnicima.

5. Zaključak

Iz svega rečenog u prethodnim poglavljima, izvodi se zaključak da Aspektno Orijentisano Programiranje, pravilnom upotrebom, može dovesti do veće modularnosti u implementaciji softverskih sistema. Dužnost svakog modula je precizno definisana, međusobna veza je minimalna, a evolucija sistema je olakšana, svaka modifikacija ili uvođenje potpuno nove funkcionalnosti u sistem je lokalizovano unutar jednog aspekta.
Međutim, ne treba izgubiti iz vida da sa novim nivoom apstrakcije, dolazi i viši novi kompleksnosti sistema, za čije je razmevanje potrebno visok nivo obuke i iskustva. Preterana upotreba aspekata u sistemu može dovesti do toka programa koji je teško ili nemoguće pratiti. Novi alati za editovanje aspektnih jezika, debageri kojima se može zaustaviti izvršavanje programa u aspektu, pomažu u tome, ali pravilna upotreba aspekata je ipak najbolja prevencija nerazumljivih sistema.

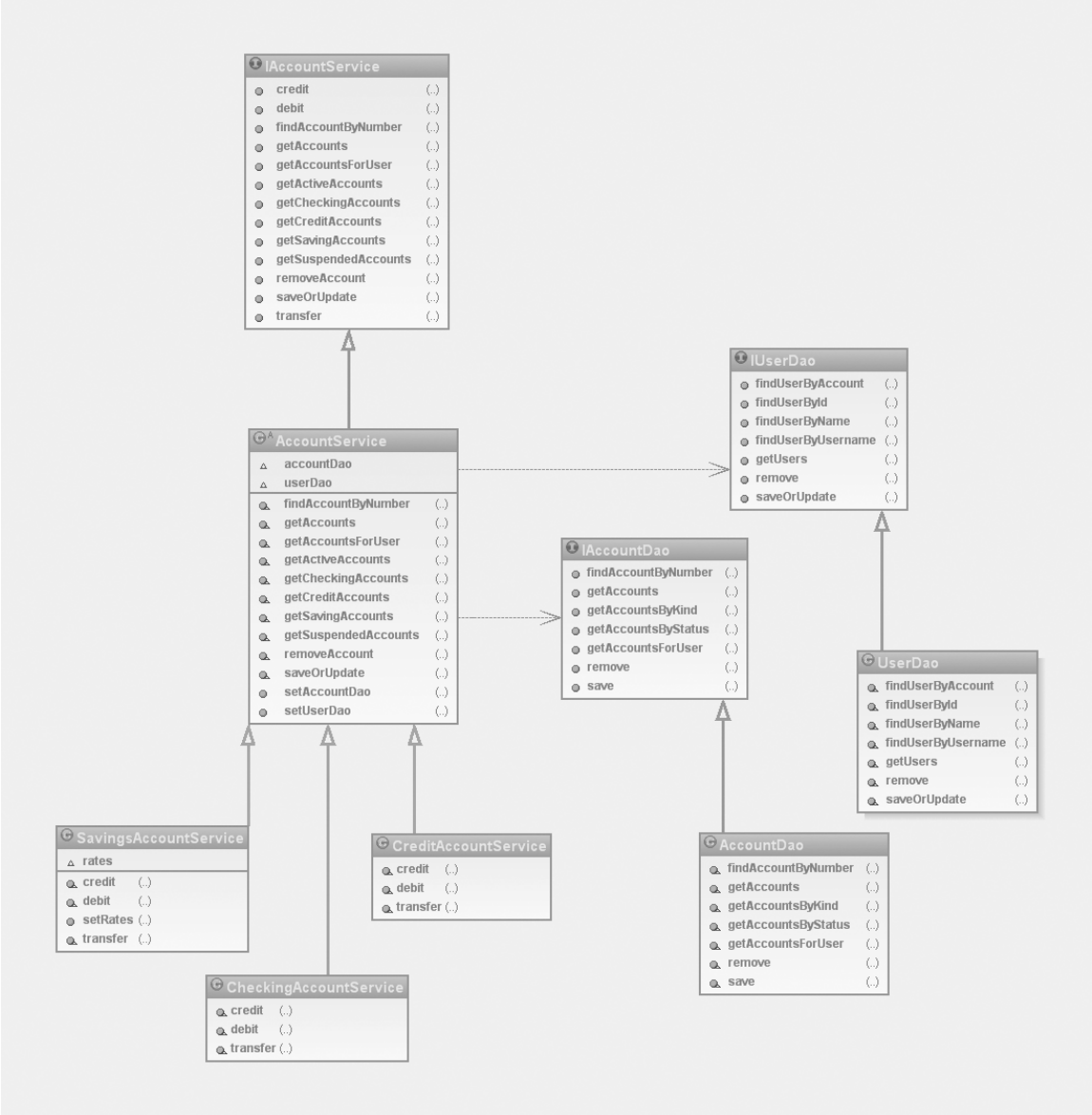
Dodatak - Primer implementacije bankarskog sistema koristeći AOP

Nakon detaljnog opisa AOP metodologije, možemo prikazati upotrebu ovog pristupa u implementaciji aplikacije iz svakodnevnog života – bankarskog sistema. Usluge koje jedna banka mora da pruži svojim klijentima, ali i radnicima zaposlenim u banci, verovatno su svima poznate, ali da nabrojimo najbitnije: povlačenje sredstava sa računa, stavljanje sredstava na račun, prebacivanje sredstava sa jednog na drugi račun, administracija korisnika (kreiranje, brisanje, pregled korisnika), administracija računa (kreranje, brisanje, suspendovanje, pregled računa). Naravno, u uplementaciji svake od ovih navedenih operacija, mora se voditi računa o isprepletanim dužnostima sistema koje su od ključnog značaja za svaki bankarski sistem: transakcijama nad bazom podataka, sigurnosnom proverom (autorizacija korisnika), validacijom ulaznih podataka i logovanjem svake obavljene operacije u sistemu.

D.1. Pregled rešenja implementacije i korišćenih tehnologija

Naš bankarski sistem implementiran kao primer korišćenja AOP-a je pojednostavljeno rešenje sistema koji bi koristile banke u realnom životu, gde je akcenat stavljen na implementaciju isprepletanih dužnosti sistema koristeći AspectJ, ali rešenje koje zadovoljava i osnovne, centralne dužnosti koje takav sistem mora da zadovolji. U datom primeru, korišćeni su trenutno aktuelni principi *software*-skog inženjerstva: implementacija korišćenjem interfejsa, *dependency injection*, objektno-relaciono mapiranje i troslojna apstrakcija poslovne logike – domenski, DAO (Data Access Object) i servisni sloj. Domenski objekti predstavljaju mapirane tabele (preciznije rečeno redove u tabeli) relacione baze podataka, DAO objekti definišu operacije koje se mogu izvršavati nad domenskim objektima i servis objekti definišu operacije koje su dostupne korisniku sistema i koriste DAO objekte da bi realizovali te operacije. Kao i u mnogim aplikacijama napisanim korišćenjem Java programskog jezika i srodnih tehnologija, korišćen je set tehnologija koji poštuje pomenute principe: *Spring framework* zarad bolje konfigurabilnosti i modularnosti aplikacije, JPA (*Java Persistence API*) standard za objektno-relaciono mapiranje, MySQL relaciona baza za perzistenciju podataka, *Maven2* za upravljanje projektima (kompajliranje, buildovanje, pakovanje u jar) i AspectJ za implementaciju svih “sporednih” dužnosti sistema. Na slikama D.1.1 i D.1.2 prikazani su dijagrami klasa koje interaktuju u implementaciji dva servisa prisutna u priloženom primeru bankarskog sistema (servis za upravljanje klijentima banke (*UserService*) i servis za upravljanje računima banke (*AccountService*)).

Slika D.1.1 Dijagram klasa koje učestvuju u implementaciji servisa za upravljanje klijentima banke– *IUserService* je interfejs koji definiše operacije koje korisnik aplikacije može izvršiti nad klijentom banke (*User*), *UserService* je konkretna implementacija tog interfejsa, *IUserDao* definše operacije nad domneskim objektom i *UserDao* je JPA/Hibernate implementacija tog interfejsa.



Slika D.1.2 Dijagram klasa koje su učestvuju u implementaciji servisa za upravljanje računima banke – *IAccountService* je interfejs koji definiše operacije koje korisnik aplikacije može izvršiti nad računima banke (*Account*), *AccountService* je abstraktna implementacija tog interfejsa koja sadrži implemnetacije operacija koje su zajedničke za sve tipove računa, dok *CheckingAccountService*, *SavingsAccountService* i *CreditAccountService* sadrže implementacije operacija koje su specifične za svaku vrstu računa (tekući, štedni i kreditni). *IAccountDao* definše operacije nad domenskim objektom i *AccountDao* je JPA/Hibernate implementacija tog interfejsa.

D.2. Implementacija kontrole pristupa (sigurnosne logike)

```
00 @Aspect
01 public class SecurityAspect {
02    IAuthorizationService authorizationService;
03    UserSecurityContext userSecurityContext;
04    IUserService userService;
05
06    @Before("execution(* *(..)) &&
07             @annotation(org.matf.icikic.banking.security.Secure)")
08    public void secure(JoinPoint jp) {
09        String role = null;
10        Method method = getMethod(jp);
11        Annotation[] annotations = method.getAnnotations();
12        for (int k = 0; k < annotations.length; k++) {
13            Annotation annotation = annotations[k];
14            if (annotation instanceof Secure) {
15                role=((Secure)annotation).value();
16            }
17        }
18        User target = null;
19        for (int i=0; i<jp.getArgs().length; i++) {
20           Object param = jp.getArgs()[i];
21            if (param instanceof User) {
22                target = (User) param;
23            }
24        }
25        if (target == null) {
26            for (int i=0; i<jp.getArgs().length; i++) {
27                Object param = jp.getArgs()[i];
28                if (param instanceof Account) {
29                    target = userService.findUserByAccount((Account)param);
30                }
31            }
32        }
33        userSecurityContext.setTargetUser(target);
34        authorizationService.authorize(userSecurityContext, role);
35    }
36
37    protected Method getMethod(JoinPoint jp) {
38        Method invoked = null;
39        try {
40            MethodSignature met = (MethodSignature) jp.getSignature();
41            invoked = jp.getSourceLocation().getWithinType().getMethod(
42                      met.getMethod().getName(),
43                      met.getMethod().getParameterTypes());
44        } catch (NoSuchMethodException e) {
45            throw new RuntimeException(e);
46        }
47        return invoked;
48    }
49 }
```

Primer D.2.1. Aspekt koji implementira kontrolu pristupa u bankarskom sistemu

U datom primeru implementacije bankarskog sistema, kontrola pristupa je enkapsulirana u jednom aspektu (*SecurityAspect*). Predstavljeni aspekt, kao i drugi aspekti iz ovog primera, napisani su korišćenjem @AspectJ sintakse. Dva osnovna razloga su uticala na takvu odluku, prvi i važniji jeste da bi i alternativna sintaksa AspectJ-a (svi primeri do sada prikazani su korišćenjem klasične AspectJ sintakse) bila predstavljena. Drugi razlog je mogućnost rada sa AspectJ korišćenjem običanog Java editor koda (ne postoje nove rezervisane reči, sve AspectJ pojedinosti jezika se izražavaju pomoću Java anotacija). Jedna od "negativnih" strana upotrebe *@AspectJ* sintakse je nemogućnost prosleđivanja konteksta tačke preseka (argumenti metode, this objekat, itd) u savet aspekta, već se za takve potrebe mora koristiti refleksija (primer D.2.1, linija 37-48).

Kao što je već pomenuto u ranijem tekstu, svaki aspekt određuju tačke preseka i savet koji definiše.

Aspekt iz primera D.2.1. selektuje tačke izvršavanja svih metoda prisutnih u bankarskom sistemu koje su označene sa *@Secure* anotacijom (primer D.2.1, linija 06-07) i umeće definisan savet pre izvršavanja takve metode (*@Before* anotacija definiše "pre" savet u @AspectJ sintaksi).

Sigurnosna logika implementirana u ovom primeru se zasniva na postojanju privilegija koje korisnik mora posedovati da bi izvršio zahtevanu operaciju. Potreban nivo privilegija je definisan unutar *@Secure* anotacije (primer D.2.2, linija 05, 12). Unutar saveta se definisan nivo privilegija izvlači pomoću refleksije (primer D.2.1, linija 10-17), a zatim se takođe korišćenjem refleksije izvlači i *User* nad kojim se izvršava operacija (prosleđen ili kao argument metode (primer D.2.1, linija 18-24), ili kao vlasnik računa koji je prosleđen kao argument (primer D.2.1, linija 25-32)). Sama autorizacija korisnika se delegira na autorizacioni servis, definisan kroz *IAuthorizationService* interfejs.

```
01 public abstract class AccountService implements IAccountService {
02     IAccountDao accountDao;
03     IUserDao userDao;
04
05     @Secure("ROLE_ADMIN")
06     public void removeAccount(User user, Account account) {
07         user.getAccounts().remove(account);
08         userDao.saveOrUpdate(user);
09     }
10     ...
11
12     @Secure("ROLE_USER")
13     public List<Account> getAccountsForUser(User user) {
14         return accountDao.getAccountsForUser(user);
15     }
```

Primer D.2.2. Operacije koje zahtevaju sigurnosnu proveru pre izvršavanja. Implementacija operacije ne sadrži proveru sigurnosnih pravila aplikacije, ali će poštovanje tih pravila nametnuti *SecurityAspect* umetanjem definisanog saveta pre izvršavanja same operacije.

```
00 public class AuthorizationService implements IAuthorizationService {
01   public void authorize(UserSecurityContext context, String neededRole) {
02      IUserDetails details = context.getCurrentUserDetails();
03      User currUser = userDao.findUserByUsername(details.getUsername());
04      if (null == currUser) {
05          throw new SecurityException("Username '" + details.getUsername() + "'
06                                      not found.");
07      }
08      if(!currUser.getCredentials().getPassword().equals(details.getPassword())) {
09          throw new SecurityException("Bad password provided for username '" +
10                                      details.getUsername() + "'.");
11      }
12      if (neededRole.equals("ROLE_ADMIN")) {
13          if (!containsRoleByName(currUser, neededRole)) {
14              throw new SecurityException("Access denied. User [" + currUser +
15                                          "] has insufficient privileges.");
16          }
17      } else if (neededRole.equals("ROLE_USER")) {
18      User targetedUser = context.getTargetUser();
19      final boolean usersMatch =
targetedUser.getCredentials().getUsername().equals(currUser.getCredentials().getUser
name());
20      final boolean containsAdminRole = containsRoleByName(currUser, ROLE_ADMIN");
21      final boolean containsUserRole = containsRoleByName(currUser, "ROLE_USER");
22      if (!containsAdminRole && !(usersMatch && containsUserRole)) {
23          throw new SecurityException("Access denied. User [" + currUser +
24                                      "] is not matched to target user.");
25      }
26 }
```

Primer D.2.3. Implementacija *IAuthorizationService* interfejsa, servis koji enkapsulira pravila aplikacije za davanje, odnosno zabrane pristupa trenutno ulogovanog korisnika sistema odredjenoj operaciji sistema.

D.3. Validacija ulaznih podataka

```
00 @Aspect
01 public class ValidationAspect implements ApplicationContextAware {
02    ApplicationContext applicationContext;
03
04    @Before("execution(* *(..)) &&
05             @annotation(org.matf.icikic.banking.validation.Validation)")
06    public void validate(JoinPoint jp) {
07        ValidationErrors errors = new ValidationErrors();
08        Annotation[][] annotations = getMethod(jp).getParameterAnnotations();
09        for (int i = 0; i < annotations.length; i++) {
10            Annotation[] annos = annotations[i];
11            for (int j = 0; j < annos.length; j++) {
12                Annotation annotation = annos[j];
13                if (annotation instanceof ValidatorConfig) {
14                    String validatorName =
15                           (ValidatorConfig)annotation).validatorName();
16                    IValidator validator =
17                           (IValidator)applicationContext.getBean(validatorName);
18                    boolean valid =  validator.validate(jp.getArgs()[i], errors);
19                }
20           }
21        }
22        if (!errors.isEmpty()) {
23            throw new ValidationException(errors);
24        }
25    }
26 }
```

Primer D.3.1 Aspekt koji implementira validaciju podataka u bankarskom sistemu

Kao i u slučaju kontrole pristupa, validacija podataka je enkapsulirana u jednom aspektu koji selektuje tačke izvršavanja svih metoda u sistemu označenih sa *@Validation* anotacijom (primer D.3.1, linija 04-05). Unutar saveta koji se umeće pre izvršavanja validirane metode, refleksijom se izvlače svi parametri metode i validacija se izvršava nad onim koji su označeni *@ValidationConfig* anotacijom (primer D.3.1, linija 08-21). U *@ValidationConfig* anotaciji definisano je ime validatora koji treba primeniti na dati argument i na osnovu kojeg se sam validator instancira iz Spring-ovog aplikacionog konteksta. Robustnost celokupnog mehanizama validacije primenjen u primeru bankarskog sistema, zasniva se na implementaciji generičkih validatora (*RequiredValidator* i *ConditionValidator*), Spring konfiguraciji i evaluaciji OGNL (*Object-Graph Navigation Language*) izraza, ali to prevazilazi okvire ovog poglavlja[1]. U našem slučaju bitno je pomenuti prisutnost validatora koji proverava da dati parametar postoji (nije *null*) i validator računa koji proverava da je račun aktivan i odredjenog tipa (tekući, štedni, kreditni).

```
00 public class UserService implements IUserService {
01    @Validation
02     public User saveOrUpdate(@ValidatorConfig(validatorName="requiredUser")
03                              User user) {
04         return userDao.saveOrUpdate(user);
05     }
06 }
07
08 public class CheckingAccountService extends AccountService {
09     @Validation
10     public void credit(@ValidatorConfig(validatorName="activeCheckingAccount")
11                        Account account, BigDecimal amount) {
12         BigDecimal balance = account.getBalance();
13         if (balance.compareTo(amount) >= 0) {
14             account.setBalance(balance.subtract(amount));
15             accountDao.save(account);
16         } else {
17             throw new InsufficientBalanceException();
18         }
19     }
20 }
```

[1] Za bolje razumevanje implementiranog mehanizma validacije, pogledati izvorni kod bankarskog sistema

Primer D.3.2. Označene metode i argumenti nad kojima treba primeniti validaciju. U prvom primeru se definiše validator koji proverava prisutnost argumenta (*user != null*), u drugom definisani validator proverava da li je račun aktivan i da li je tip računa tekući.

D.4. Upravljanje transakcijama u radu sa bazom podataka

Prilikom pravljenja ovog primera bankarskog sistema, namera je bila da se predstavi način na koji AspectJ pomaže u implementaciji isprepletanih dužnosti sistema, ali i da taj sistem ima određenu vrednost u stvarnom životu, tj. da se ispoštuju neki opšte prihvaćeni principi u izradi poslovne aplikacije. Zbog toga, upravljanje transakcijama u našem primeru nije implementirano ručno, već uz pomoć ugrađene podrške koju Spring framework nudi za upravljanje transakcijama. Spring nudi više različitih načina za upravljanje transakcijama u zavisnosti od potrebnog nivoa granularnosti i raspoloživih tehnologija; ali svi pristupi se oslanjaju na AOP metodologiju. Za nas najzanimljiviji pristup i verovatno već prepoznatljiv jeste postojanje *@Transactional* anotacije kojom se obeležavaju metode koje treba izvršiti unutar transakcije. Iako ovakav pristup deluje da je identičan već opisanim u slučaju kontrole pristupa i validacije, Spring nudi mnogo robusnije rešenje, komercijalno prihvatljivo, čija se robusnost izražava između ostalog i u mogućnosti biranja načina na koji se *@Transactional* anotacije "obrađuju". U osnovnom modu koriste se *proxy* objekti kojim se okružuju metode/klase (videti poglavlju 2.5, slika 2.6), ali je moguće specificirati AspectJ mod u kome Spring nudi *AnnotationTransactionAspect* koji umeće kod za upravljanje transakcijama (savet) oko označenih metoda. Vredno je pomenuti da se Spring kao jedno od najčešće upotrebljivanih rešenja u razvoju software korišćenjem Java tehnologija, u velikoj meri oslanja na AOP (upravljanje transakcijama, web servisima, konfiguracija objekata, itd), što se može posmatrati kao priznanje elegantosti koju AOP metodologija unosi u rešavanju isprepletanih dužnosti sistema.

```
00 public class CreditAccountService extends AccountService {
01
02    @Transactional
03    public void transfer(Account from, Account to, BigDecimal amount) {
04        credit(from, amount);
05        debit(to, amount);
06    }
```

Primer D.4.1. Metode koje zahtevaju izvršavanje unutar transakcije. Sama metoda ne sadrži kod za upravljanje transakcijama, ali će Spring AOP umetanjem definisanog AspectJ saveta, odnosno obmotavanjem metode u proxy objekat obezbediti izvršavanje unutar transakcije.

D.5. Logovanje izvršenih operacija

```
00 @Aspect
01 public class LoggingAspect extends BaseBankingAspect {
02     @Pointcut("execution(* org.matf.icikic.banking.service..*(..))")
03     protected void allServiceMethods() {
04     }
05
06     @Pointcut("execution(* org.matf.icikic.banking.dao..*(..))")
07     protected void allDaoMethods() {
08     }
09
10     @Pointcut("allServiceMethods() || allDaoMethods()")
11     protected void allServiceAndDaoMethods() {
12     }
13
14     @Around("allServiceAndDaoMethods()")
15     public Object log(ProceedingJoinPoint pjp) throws Throwable {
16         Logger logger = LoggerFactory.getLogger(getTypeName(pjp));
17         try {
18             boolean isDebugEnabled = logger.isDebugEnabled();
19             if (isDebugEnabled) {
20                logger.debug(getOperationName(pjp)
21                             + "( --> " + getArgs(pjp) + ")");
22            }
23            long start = System.nanoTime();
24            Object ret = pjp.proceed();
25            long end = System.nanoTime();
26            if (isDebugEnabled) {
27                double execTime = (end - start) / 1000;
28                logger.debug(getOperationName(pjp) + "( <-- " +
29                  formatString(ret) + ") , execution time: " + execTime+ " ms");
30            }
31            return ret;
32        } catch (Throwable e) {
33            if (logger.isErrorEnabled()) {
34                logger.error(getOperationName(pjp) + "( <-- ) : "
35                             + e.getLocalizedMessage(), e);
36            }
37            throw e;
38        }
39    }
40 }
```

Primer D.5.1. Aspekt za logovanje i prćenje performansi sistema

Aspekt za logovanje se razlikuje od prehodno prikazanih u par tačaka. On definiše imenovane tačke preseka (*allServiceMethods* i *allDaoMethods*) i primenjuje “okružujući” savet oko svake metode u svakoj klasi u *org.matf.icikic.banking.dao* i *org.matf.icikic.banking.service* paketu. Pre ulaska u svaku operaciju, ime operacije i prosleđeni argumenti se loguju. Nakon završetka operacije, loguju se rezultat i vreme izvršavanja, odnosno greška u slučaju da operacija nije uspela.

Reference

Knjige i članci

[1] Bauer, C., King, . *Java Persistence with Hibernate*, Manning Publications, 2005.
[2] Florijn, G., Meijers, M., Winsen, P. van. *Tool support for object-oriented patterns*. Objavljeno u *Proceedings of ECOOP*, 1997.
[3] Gamma, E., Helm, R., Johnson R., Vlissides J. *Design Patterns-Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[4] Hannemann, J., Kiczales, G. *Design Pattern Implementation in Java and AspectJ*. Objavljeno u *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, strana 161-73, 2002.
[5] Laddad, R. *AspectJ in Action*, drugo izdanje. Manning Publications, 2008.
[6] Mapelsden, D., Hosking, J. and Grundy, J. *Design Pattern Modelling and Instantiation using DPML*. Objavljeno u *Proceeding of TOOLS Pacific*, 2002.
[7] Norvig, P. *Design Patterns in Dynamic Programming*. Objavljeno u *Object World*, Boston MA, 1996.
[8] Sullivan, G. T. *Advanced Programming Language Features for Executable Design Patterns*. MIT Artificial Intelligence Laboratory, 2002.
[9] Walls, C., Breidenbach, R.. *Spring in Action*, drugo izdanje. Manning Publications, 2008.

Internet resorsi

[10] Martin, Robert C. *The Open Closed Principle*, 2002. Dostupno na http://www.objectmentor.com/resources/articles/ocp.pdf.
[11] Martin, Robert C. *The Single Responsibility Principle, 2*002. Dostupno na http://www.objectmentor.com/resources/articles/ocp.pdf.
[11] Aspektno orijentisano programiranja http://aosd.net
[12] AspectJ http://www.eclipse.org/aspectj/
[13] Spring framework http://www.springframework.org/
[14] Separation of Concerns http://ctrl-shift-b.blogspot.com/2008/01/art-of-separation-of-concerns.html
[15] On the role of scientific thought (E.W.Dijsktra) http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html