Univerzitet u Beogradu

Matematički fakultet

Milan Panić

Aspektno orijentisani pristup programiranju

master rad

Beograd

2016.

Mentor: **dr Vladimir Filipović**
Matematički fakultet,
Univerzitet u Beogradu

Članovi komisije: **dr Vladimir Filipović**
Matematički fakultet,
Univerzitet u Beogradu

dr Saša Malkov
Matematički fakultet,
Univerzitet u Beogradu

dr Aleksandar Kartelj
Matematički fakultet,
Univerzitet u Beogradu

Datum odbrane: ______________________________

Sadržaj:

Uvod 1
  Razvoj programiranja 1
  Objektno-orijentisana paradigma 2
  Objektno orijentisana paradigma i nedostaci 3
Aspektno orijentisano programiranje 5
  Ključne odlike aspektno orijentisanog sistema 5
  Istorija aspektno orijentisanog programiranja i postojeće implementacije 5
AspectJ 8
  Tradicionalna AspectJ sintaksa 8
  @AspectJ sintaksa 10
  Tačke spajanja 12
  Jezik za definisanje tačaka reza 16
  Gramatika jezika za definisanje tačaka reza 16
  Tkač (Weaver) 19
    Java virtuelna mašina 19
    Faza prevođenja 22
    Faza tkanja 22
  Analiza implementacije aspektnog sistema AspectJ-a 23
Spring AOP 26
  Spring okvir 26
  Inverzija kontrole - Princip na kome radi jezgro Spring okvira 27
  Spring aplikacija 31
  Aspektno orijentisano programiranje sa zastupnicima 34
    Projektni uzorak *zastupnik* 34
Alternative aspekto orijentisanom programiranju 46
  Projektni uzorci 46

Automatsko generisanje kôda 46
Zaključak 47
Reference 50
Ostala literatura 52

Uvod

Razvoj programiranja

Programiranje je od svog nastanka namenjeno pisanju programa kojima se rešavaju problemi. Sa unapređenjem mogućnosti hardvera računara koje se, grubo gledano, povećavaju u okvirima Murovog zakona, rastu i želje da se taj moćan hardver upotrebi. Problemi koje ljudi žele da reše postaju sve složeniji. Programiranje, da bi odgovorilo takvim zahtevima, kroz istoriju traži način da pisanje programa učini što efikasnijim. Ono što se u računaru mora predstaviti kao program na mašinskom jeziku, programeri su kroz istoriju zamenili pisanjem programa za asemblere i prevodioce i na taj način učinili programiranje sve složenijih problema mogućim za rešavanje.

Neki od nastalih jezika od samog svog nastanka bili su namenjeni za rešavanje konkretnog problema (ili jako malog broja konkretnih problema) dok su drugi nastali sa ciljem da budu jezici upotrebljavani za rešavanje najrazličitijih problema. Ovi drugi poznatiji su i pod nazivom jezici opšte namene i u ovom kontekstu su zanimljivi za proučavanje i dalja priča odnosi se samo na njih.

Programski jezici opšte namene, kao i prirodni, opstaju samo ako ih neko koristi. Proučavanjem programskih jezika koji su ostali u aktivnoj upotrebi, uočeno je da mnogi od njih imaju zajednička svojstva, pa su tako grupisani u zajedničke grupe, odnosno programske paradigme. Najdominantnije u današnje vreme su proceduralna, funkcionalna i objektno-orijentisana paradigma. Takođe, valja napomenuti da postoje i druge paradigme, kao i potparadigme osnovnih, a i da postoje jezici koji po svojoj strukturi spadaju u više paradigmi. Tipičan primer višeparadigmatskog programskog jezika je C++ koji pripada i proceduralnoj i objektno-orijentisanoj paradigmi.

Gore pomenute paradigme dobile su naziv po strukturama u svojoj semantici koje omogućavaju programeru da se izbori sa složenošću problema koji rešava. Jako složeni problemi teže da se podele na manje složene celine, odnosno module, koji bi samim tim bili lakši za rešavanje [1]. Ovaj proces naziva se funkcionalna dekompozicija. Jedinice modularnosti u objektno-orijentisanoj paradigmi predstavljaju objekti, u proceduralnoj procedure, a u funkcionalnoj funkcije.

Među njima je bez sumnje najrasprostranjenija objektno-orijenisana paradigma, dok druge dve ponajviše opstaju u aktivnoj upotrebi u jezicima koji podržavaju programiranje u više paradigmi ili kao jezici koji su podrška centralnom objektno-orijentisanom programskom jeziku na projektu. Samostalna upotreba ovih programskih jezika na projektima ograničena je uglavnom na akademsku zajednicu.

Pored izbora samog programskog jezika u kome će se raditi projekat, jako je bitan i izbor razvojnog okruženja (engl. Integrated Development Environment) i drugih alata koji ubrzavaju sam proces programiranja, dostupnost gotovih biblioteka napisanih da daju gotovu osnovu za specifične vrste problema, brzina savladavanja programskog jezika ukoliko jezik treba tek učiti itd.

Ono što dodatno ubrzava programiranje jeste i primena projektnih uzoraka (engl. design patterns) koji predstavljaju apstraktne obrasce za rešavanje nekih problema koji su se već pokazali dobrim u praksi.

Objektno-orijentisana paradigma

Objektno-orijentisana paradigma je bazirana na konceptu objekta koji se opisuje svojstvima i svojim ponašanjem. Svojstva objekata jesu strukture podataka koje se čuvaju u promenljivama i nazivaju se atributi, dok se ponašanje opisuje metodama, tj. procedurama za obradu podataka. Ceo program posmatra se kao skup objekata, a sam programski tok kao interakcija među njima. Objektno orijentisani jezici najčešće obezbeđuju i neki način organizacije kôda u relativno nezavisne module, najčešće su to paketi ili imenski prostori (engl. namespace).

Objektno-orijentisani jezici najčešće su klasnog tipa. To podrazumeva da svi objekti koji imaju ista svojstva i ponašanje pripadaju jednoj klasi, koja ih opisuje, dok se konkretni objekti nazivaju primercima (ili instancama) klase. Ovo je naročito korisno jer je za rešavanje nekog problema najčešće potrebno više sličnih objekata, pa i objektno-orijentisani jezici koji nisu klasnog tipa obezbeđuju način za kreiranje sličnih objekata - najčešće preko prototipa.

Principi koji su najzaslužniji za mogućnost pisanja kratkog i dobro dizajniranog kôda i koji obezbeđuju prednjačenje objektno-orijentisane paradigme u odnosu na druge jesu: enkapsuliranje, kompozicija i nasleđivanje. Ovi principi tipični su za jezike klasnog tipa, dok jezici koji nisu klasnog tipa sličan efekat postižu alternativnim putem.

- Enkapsuliranje ili učaurivanje podrazumeva mogućnost ograničavanja vidljivosti i pristupa nekom delu kôda ili strukturama podataka. Ovo je jako korisna apstrakcija koji obezbeđuje smanjenje složenosti polaznog problema tako što sakriva sve suvišne informacije, ograničiva mogućnost korišćenja kôda na samo onaj koji objedinjuje sve skrivene delove, a i zabranjuje nenamernu i zlonamernu modifikaciju podataka.
- Kompozicija predstavlja mogućnost da atributi objekta budu drugi objekti.
- Nasleđivanje predstavlja apstrakciju koja daje mogućnost hijerarhijske organizacije klasa, čime se pojednostavljuje dizajn programa, a takođe smanjuje količina dupliranog kôda. Nasleđivanje je različito implementirano unutar različitih jezika. Neki jezici daju mogućnost višestrukog nasleđivanja, što po pravilu dovodi do nelinearnosti unutar hijerarhije, dok drugi daju mogućnost samo jednostrukog nasleđivanja, ali uz još neke strukture koje obezbeđuju mogućnost upotrebljivosti već napisanog kôda, najčešće preko interfejsa ili crta (engl. trait).

Svi navedeni principi, kao i još mnogi koji su specifični za podgrupu ili pojedinačne jezike iz ove grupe, namenjeni su da olakšaju dizajn programskog sistema koji treba napraviti, kao i da smanje programsko vreme potrebno za njegovu implementaciju.

Prilikom dizajniranja i implementacije nekog programskog sistema uvek se uočavaju najbitnije funkcionalnosti koje taj sistem treba da poseduje i one se nazivaju ključnim funkcionalnostima. Na primer, ako treba da se napravi program koji

simulira rad automobila, najbitnije je simulirati brzinu automobila, kretanje, izgled itd. pa su to ključne funkcionalnosti. Pri izgradnji simulatora, količina detalja koja se uzima u obzir se razlikuje u zavisnosti od toga koliko simulator treba da bude verodostojan: od odgovora na pitanje da li je simulator potreban za simulaciju u video igri ili za simulaciju rada vazdušnih jastuka unutar automobilske kompanije će zavisiti nivo detaljnosti.

Pored ključnih funcionalnosti, postoje i sporedne funkcionalnosti sistema, koje predstavljaju željena ponašanja sistema koja nisu direktno vezana za željeni konkretan program već više predstavljaju željena ponašanja svih programa. Mnoge od sporednih funkcionalnosti treba da se prostiru u više različitih modula ili kroz više hijerarhijski nezavisnih klasa. Na primer, svi programi treba da budu obezbeđeni od neovlašćenog upada hakera, treba proveriti koja je brzina izvršavanja nekog dela programa, ukoliko program radi sa relacionom bazom podataka neophodan je rad sa transakcijama kako ne bi došlo do nekonzistentnosti podataka itd.

Prilikom pravljenja dizajna sistema logično je da se najviše pažnje poklanja ključnim funkcionalnostima sistema. Osnovne odlike dobrog dizajna programskog sistema podrazumevaju laku nadogradnju i izmenu postojećih funkcionalnosti u sistemu.

Objektno-orijentisani jezici stekli su popularnost prvenstveno zato što je moguće elegantno implementiranje i dizajniranje ključnih funkcionalnosti sistema. Na primer, da bi se povećala mogućnost ponovne upotrebe već napisanog kôda i smanjila potreba za lančanim izmenama u objektno-orijentisanom jeziku treba poštovati neke osnovne principe među kojima su i jedinstvena odgovornost klase (engl. single responsibility principle), i otvorenost za proširenje, a zatvorenost za promenu (engl. open/closed principle) i nepridržavanje ovih principa sigurno dovodi do zamršenja odnosno rasejanja kôda [2].

Objektno orijentisana paradigma i nedostaci

Jedna od osnovnih mana tradicionalnih objektno-orijentisanih jezika jeste to što je praktično nemoguće implementirati funkcionalnosti koje treba da važe globalno ili u više modula (engl. crosscutting functionality) bez zamršenja ili rasejanja kôda.

Tipičan nedostatak objektno-orijentisane paradigme biće prikazan u analizi mogućih implementacija sigurnosti unutar Java aplikacije.

Programski jezik Java je klasnog tipa u kome postoji jednostruko nasleđivanje dok se dodatne funkcionalnosti i ponovno iskorišćenje već napisanog kôda klasa osim nasleđivanjem obezbeđuje interfejsima i njihovom implementacijom.

Sigurnost aplikacije je takve prirode da treba da važi u svakom delu aplikacije, tačnije sigurnost je jaka upravo onoliko koliko je jak njen najslabiji deo. Ovom problemu može da se pristupi na različite načine, ali je svakako bitan deo dizajna svakog informacionog sistema.

Naivan pokušaj implentacije sigurnosti je da se unutar svake klase koja obezbeđuje neku ključnu funkcionalnost proveri da li onaj ko je započeo funkcionalnosti ima prava da izvrši traženi kôd. Ovo podrazumeva prosleđivanje podataka o korisniku kroz sve metode aplikacije, što podrazumeva mnogo dupliranja kôda.

Drugi, sofisticiraniji pristup, koji se najčešće i koristi, jeste postojanje servisa u vidu interfejsa koji obezbeđuje metode za proveru prava pristupa nekom delu kôda.

Interfejsi u programskom jeziku Java obezbeđuju slabu povezanost komponenata zato što daju mogućnost da se implementacija neke komponente zameni bez potrebe da se komunikacija između različitih komponenata promeni.

Ovakva apstrakcija sigurnosti je najbolja moguća unutar postojeće objektno-orijentisane paradigme. Očigledno je da ovakva apstrakcija ne omogućava potpunu razdvojenost modula koji implementira sigurnost zato što svi moduli koji treba da budu sigurni moraju da znaju interfejs i da eksplicitno u kôdu pozivaju interfejs modula koji obezbeđuje proveru prava pristupa.

Aspektno orijentisano programiranje

Aspektno orijentisano programiranje (AOP) rešava pomenuti problem uvođenjem nove jedinice modularnosti - aspekta. Aspekt objedinjuje funkcionalnost koja bi inače bila zajednička za više modula.

Aspekt po svojoj prirodi nije povezan sa ostalim modulima programa. Nakon definisanja višemodularne funkcionalnosti unutar aspekta, mora se izvršiti poseban proces tkanja (engl. weaving) definisane funkcionalnosti aspekta u ostale module sistema.

AOP nije jedini način bolje organizacije višemodularnih funkcionalnosti u objektno orijentisanom programiranju. Pored aspektno orijentisanog sistema postoje i druge tehnike za rešavanje istiih problema i da bi se od njih razlikovalo biće predstavljena jasna definicija aspektno orijentisanog sistema.

Ključne odlike aspektno orijentisanog sistema

Aspektno orijentisan sistem (AOS) identifikuje i klasifikuje specifične tačke u izvršenju programa koje naziva tačkama spajanja (engl. join points). Svaki računarski program ima karakteristične tačke u svom izvršenju. Tačke spajanja u objektno orijentisanom programu mogu biti: početak i kraj izvršenja metoda, poziv konstruktora, izbacivanje izuzetka, početak for petlje, dodela vrednosti proizvoljnoj promenljivoj itd. AOS mora da obezbedi način da identifikuje podskupove svih tačaka spajanja koje može da identifikuje - to su tzv tačke reza (engl. pointcut) i konstrukciju kojom može da doda nove funkcionalnosti ili da potpuno promeni izvršenje dela programa koji je povezan tim tačkama spajanja - to je tzv. savet (engl. advice).

AOS najčešće jasno definiše i aspekt, konstrukciju koja objedinjuje navedene funkcionalnosti unutar jednog modula. Dodatno, AOS može da poseduje i konstrukcije kojima se menja statička struktura programa.

Istorija aspektno orijentisanog programiranja i postojeće implementacije

Ideja aspektno orijentisanog programiranja nastala je u razvojnom timu istraživačkog centra PARC kompanije Xerox na čijem čelu je bio Gregor Kiczales. Prvi naučni rad na temu aspektno orijentisanog programiranja objavljen je 1997. godine [3]. Istorija i dalji razvoj AOP-a direktno je vezana za AspectJ implementaciju, koja je razvijana u ovom istraživačkom centru. AspectJ omogućava aspektna svojstva programskom jeziku Java i do danas je najpotpunija implementacija aspektno orijentisanih ideja.

AspectJ je razvijan u okviru kompanije Xerox do 2001. godine kada je preveden projekat u Eclipse open-source zajednice, u čijim se okvirima razvija do danas. [4]

AspectJ je nastao kao poseban programski jezik koji je nadogradnja programskog jezika Java aspektnim konstrukcijama. Iako je zamišljen kao sveobuhvatni projekat koji će obezbediti i razvojno okruženje koje omogućava lakše

korišćenje aspekata, nije bilo dovoljno brzog razvoja koji bi pratio porast mogućnosti razvojnih okruženja za programski jezik Java.

AspectJ jeste nudio lepa rešenja za organizaciju multimodularnih funkcionalnosti, ali su dug period učenja i nedostatak gotovih alata koji bi olakšavali rad odbijali programere od njegovog masovnog korišćenja.

Aspektno orijentisano programiranje je patentirano 2002. godine kao izum Gregora i njegovih petoro kolega [5]. U prvim godinama nakon patenta interesovanje za razvoj aspektno orijentisanog programiranja je bilo veliko, najviše među programerima na Java platformi.

Bilo je i programera koji su prihvatili koncept koji nudi aspektno orijentisano programiranje, ali ne i rešenje koje AspectJ nudi, pa su započeli alternativne projekte, najviše za java platformu.

Najozbiljniji projekat među njima bio je AspektWertz. AspektWertz je implementirao aspektne funkcionalnosti unutar xml (engl. Extensible Markup Language) datoteka. Tkanje aspekata u ostale module sistema AspectWertz projekat obavljao je prilikom učitavanja klasa u virtuelnu mašinu, što je otklonilo potrebu za posebnim programskim jezikom.

Programeri su neko vreme činili odvojene napore u ovim open-source projektima, ali vremenom je uočeno da bi njihovim spajanjem došlo do potpunijeg rešenja za višemodularne funkcionalnosti i dogovoreno je da se projekti spoje unutar AspectJ projekta, i AspectJ verzije 5 uključio je zajedno sa mnogim revolucionarnim izmenama i sam Aspect Wertz projekat. [6]

Takođe, i sam programski jezik Java je unapređivan. Programski jezik ponudio je mogućnost definisanja i procesiranja anotacija unutar svog kôda. Anotacije su vrsta metaobjekata koji koji može da bude dostupan u različitim fazama izvršavanja programa. AspectJ je prihvatio ovu promenu kao alternativu za poseban programski jezik, ponudio je i alternativnu sintaksu koja je omogućila pisanje aspekata uz pomoć anotacija poznatiju kao @AspectJ. Ona nudi podskup funkcionalnosti celog AspectJ projekta i standardno bolje radi sa razvojnim okruženjima od tradicionalne AspectJ sintakse.

Spring projekat, najzastupljeniji open-source okvir (engl. framework) za rad poslovnih aplikacija na Java platformi takođe je zamišljen i implementiran sa podrškom za aspektno orijentisano programiranje. [7] Programski model za aspekte koji je Spring nudio u početku je predstavljao implementaciju biblioteke AOP alianse, koja je pokrenula inicijativu za kreiranje AOP svojstava unutar samog Java programskog jezika. Ovaj programski model bio je prilično neelegantan, pa se njegova primena svodila uglavnom na programere koji su pisali sam Spring okvir. Programeri samog Spring okvira uvideli su prednosti jasne sintakse koju @AspectJ nudi i implementirali mogući podskup ove sintakse unutar projekta.

Široko rasprostranjeni server za rad sa poslovnim aplikacijama JBoss [8] takođe nudi aspektno orijentisana rešenja jako slična @AspectJ sintaksi.

Java platforma prednjači u implementacijama aspektno orijentisanih principa. Najpotpunije implementacije van Java platforme trenutno su AspectC++ [9], rešenje za programski jezik C++ inspirisano AspectJ implementacijom i Spring.Net projekat insipiran Spring okvirom.

Upravo Spring i AspectJ predstavljaju referentne tačke za analizu mogućnosti aspektno orijentisanog programiranja. Sintaksno, aspektno orijentisani sistem koji nude je jako sličan, ali sa fundamentalno različitim pristupom implementaciji aspektno orijentisanih principa, a samim tim i mogućnostima koje nude programerima.

AspectJ

Tradicionalna AspectJ sintaksa

Pod tradicionalnom sintaksom AspectJ-a podrazumeva se proširenje programskog jezika Java dodatnim jezičkim konstrukcijama. Ukoliko se koristi tradicionalna sintaksa potrebno je instalirati AspectJ prevodilac ili koristiti neko razvojno okruženje koje će to omogućiti. Prevodilac nosi ime ajc [10] (engl. AspectJ compiler) i predstavlja proširenje javac prevodioca mogućnostima prevoda dodatnih aspektnih konstrukcija u bajt kôd. Aspekti pisani tradicionalnom sintaksom najčešće imaju ekstenziju .aj, iako ovo nije neophodno, da bi se razlikovali od java klasa.

Aspekt se definiše u strukturi koja po svojoj sintaksi podseća na klasu. Unutar nje definišu se ostale međumodularne konstrukcije.

Osnovni način na koji AspectJ omogućava definisanje međumodularnih funkcionalnosti jeste preko definisanja presečnih tačaka i saveta oko tih tačaka:

Tačka reza predstavlja skup mesta u kôdu na kojima treba umetnuti zajednički kôd za međumodularnu funkcionalnost. Ona može predstavljati, na primer, sve metode neke klase ili sve implementacije metoda nekog interfejsa itd. Jezik tačaka reza predstavlja srž AspectJ-a i naknadno će biti detaljno objašnjen pošto je zajednički i za tradicionalnu i za novu @AspectJ sintaksu.

Savet oko tačke reza je zapravo implementacija međumodularne funkcionalnosti. AspectJ definiše pet tipova saveta preko kojih može da se unese višemodularni kôd. Ovi saveti se definišu po mestu smeštanja aspektnog kôda u odnosu na tačku reza i to su:

- before, aspektni kôd se izvršava neposredno pre izvršenja tačke reza
- after, aspektni kôd se izvršava neposredno nakon izvršenja tačke reza
- after returning, aspektni kôd se izvršava neposredno nakon izvršenja tačke reza, ali samo ukoliko nije došlo do izuzetka
- after throwing, aspektni kôd se izvršava neposredno nakon izvršenja tačke reza, ali samo ukoliko je došlo do izuzetka
- around, aspektni kôd u potpunosti obavija tačku reza i daje mogućnost izvršenja aspektnog kôda i pre i posle tačke reza, daje mogućnost čak i da se tačka reza ne izvrši

Savet podseća na metod, ima telo koje predstavlja implementaciju aspektnog ponašanja koje ima isti programski tok kao i tok Java metoda, može imati parametre ukoliko ih ima presečna tačka, ključnu reč *this* kojom se može pristupiti instanci aspekta. Around savet ima i povratnu vrednost.

Ipak, saveti nisu metodi, nemaju ime, ne pozivaju se nikada direktno već se AspectJ brine o tome gde treba da budu pozvani. Saveti poseduju i neke specijalne promenljive (thisJoinPoint, thisJointPointStaticPart, EnclosingJoinPointStaticPart) koje nose dodatne informacije o samoj tački spajanja.

Sažetak jezičkog proširenja tradicionalne sintakse:

```
/** Definicija aspekta */
public aspect SimpleAspect {
	/** Definicija tačke reza */
	pointcut PointcutName() : execution(* SomeClass.*(..));
	/**
	 * Definicija saveta - jedne od najvažnijih konstrukcija koje dodaje ili
	 * menja ponašanje u tačkama reza. Mogući tipovi saveta su:
	 */
	before() : PointcutName() {
		/** Telo saveta se izvršava pre tačke reza. */
	}
	after() returning: PointcutName(){
		/**
		 * Telo saveta se izvršava nakon izvršenja koda koji je definisan tačkom
		 * reza, ali samo ukoliko se nije dogodio izuzetak.
		 */
	}
	after() throwing: PointcutName(){
		/**
		 * Telo saveta se izvršava nakon izvršenja koda koji je definisan tačkom
		 * reza, ali samo ukoliko se dogodio izuzetak.
		 */
	}
	after(): PointcutName() {
		/**
		 * Telo saveta se izvršava nakon izvršenja koda koji je definisan tačkom
		 * reza. Objedinjuje after returning i after throwing savet.
		 */
	}
	ReturnType around() : PointcutName(){
		/**
		 * Savet koji potpuno okružuje tačku reza i daje mogućnost izvršenja
		 * proizvoljnog koda oko tačke reza. Savet around() ima i povratnu
		 * vrednost koja predstavlja povratnu vrednost koda tačke reza koji
		 * je učauren. Kod tačke reza se, ukoliko se to želi, izvršava
		 * pozivanjem specijalne funkcije proceed().
		 */
		return proceed();
	}

	/** Eksplicitno definisanje redosleda izvršenja aspekata: */
	declare precedence: Aspect1, Aspect2;

	/** Definicija konstrukcija koje menjaju statičku strukturu programa: */
	/** 1.Dodavanje definicije interfejsa klasi */
	declare parents: PointcutName implements SomeInterface;
	/** 2.Dodavanje atributa klasi */
	private long SomeClass.additionalAttribute;
	/** 3.Dodavanje metoda klasi */
	public ReturnType SomeClass.additionalMethod() {
		return null;
	}
}
```

Pored ovog uobičajnog načina definisanja međumodularnih funkcionalnosti, AspectJ daje mogućnost menjanja i statičke strukture Java programa.

Klasama se na ovaj način, van samih klasa, definisati novi interfejsi koje implementira, te dodavati atributi i metode sa pravom pristupa koji je javni ili privatni. Na primer, transakcioni interfejs koji ima podrazumevanu implementaciju i sve klase implementiraju dati interfejs. Mane ovakvog pristupa ipak ostaju: programska struktura ostaje svesna višemodularne funkcionalnosti, daje se mogućnost eksplicitnog pozivanja transakcionih metoda, programer mora da vodi računa o imenima entiteta definisanih unutar i van klase, ne sme doći do kolizije itd.

Čini se da je ovaj dodatak tradicionalnoj sintaksi prvenstveno pokušaj da AspectJ popravi neke dobro poznate nedostatke samog programskog jezika Java. Ovim pristupom se može definisati podrazumevana implementacija interfejsa, samim tim dobiti nešto što podseća na osobine (engl. traits) u programskim jezicima PHP, C++ itd. [11]

Dodatne ključne reči van programskog jezika Java rezervisane su samo u odgovarajućem kontekstu. Ovo podrazumeva da će, na primer, reč „aspect" biti prepoznata kao ključna reč samo u kontekstu gde aspekt može biti definisan, u ostalim će biti samo ime nekog drugog entiteta jezika.

Tradicionalna sintaksa pored pomenutih funkcionalnosti, daje mogućnost definisanja i apstraktnih aspekata, aspekata unutar klasa, upozorenja i grešaka do kojih će doći prilikom prevođenja programa ukoliko je neko definisano pravilo narušeno, automatskog prevođenja proverenih (engl. checked) izuzetaka u neproverene izuzetke.

@AspectJ sintaksa

@AspectJ sintaksa umnogome podseća na tradicionalnu sintaksu. Ona je zasnovana na Java anotacijama koje zamenjuju rezervisane ključne reči u jeziku. Anotacije su bolje prihvaćene od novog programskog jezika pošto one predstavljaju deo Java platforme.

@AspectJ anotacije prirodno zamenjuju mnoge aspektne konstrukcije tradicionalne sintakse. Tako imamo @Aspect, anotaciju na klasi koja je u potpunosti analogna istoimenoj ključnoj reči u tradicionalnoj sintaksi, anotacije na metodama @Before, @AfterReturning, @AfterThrowing, @After i @Around koji unutar metoda nad kojim su definisani očekuju implementaciju višemodularne funkcionalnosti. Anotacija @Pointcut ima jednu neobičnost, ona se definiše nad metodama koje imaju prazno telo. Menjanje statičke strukture programa je ograničeno samo na definisanje novih interfejsa koje neka klasa implementira i to kroz dve anotacije @DeclareMixin i @DeclareParents.

Jedna od ključnih prednosti @AspectJ sintakse je činjenica da je za prevođenje aspekata koji su njom definisani dovoljan običan Java prevodilac, pošto on već poseduje mogućnost prevođenja anotacija. Ono što je minimalno potrebno da za prevođenje jeste dodavanje neke od biblioteka koja sadrži definicije anotacija unutar projekta, aspectjrt ili aspectjweaver.

Sažetak @AspectJ sintakse:

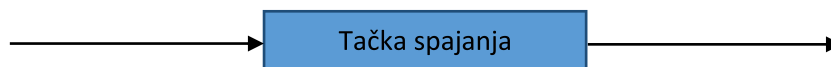
```
/** Definicija aspekta */
@Aspect
/** Definisanje redosleda izvršenja aspekata */
@DeclarePrecedence("Aspect1,Aspect2")
public class SimpleAnnotationAspect {

	/** Definicija tačke reza */
	@Pointcut("execution(* milan.panic.master.common.SomeClass.*(..))")
	public void PointcutName() {
	}

	/**
	 * Definicija saveta - jedne od najvažnijih konstrukcija koje dodaje ili
	 * menja ponašanje u tačkama reza. Mogući tipovi saveta su:
	 */
	@Before("PointcutName()")
	public void beforePoincut() {
		/** Telo saveta se izvršava pre tačke reza. */
	}
	@AfterReturning("PointcutName()")
	public void afterReturning() {
		/**
		 * Telo saveta se izvršava nakon izvršenja koda koji je definisan tačkom
		 * reza, ali samo ukoliko se nije dogodio izuzetak.
		 */
	}
	@AfterThrowing("PointcutName()")
	public void afterThrowing() {
		/**
		 * Telo saveta se izvršava nakon izvršenja koda koji je definisan tačkom
		 * reza, ali samo ukoliko se dogodio izuzetak.
		 */
	}
	@After("PointcutName()")
	public void afterPointcut(JoinPoint joinPoint) {
		/**
		 * Telo saveta se izvršava nakon izvršenja koda koji je definisan tačkom
		 * reza. Objedinjuje after returning i after throwing savet.
		 */
	}
	@Around("PointcutName()")
	public ReturnType aroundPointcut(
			ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
		/**
		 * Savet koji potpuno okružuje tačku reza i daje mogućnost izvršenja
		 * proizvoljnog koda oko tačke reza. Savet around() ima i povratnu
		 * vrednost koja predstavlja povratnu vrednost koda tačke reza koji
		 * je učauren. Kod tačke preseka se, ukoliko se to želi, izvršava
		 * pozivanjem specijalne funkcije proceed().
		 */
		return (ReturnType) proceedingJoinPoint.proceed();
	}

	/** Definicija konstrukcija koje menjaju statičku strukturu programa: */
	/** 1.Dodavanje definicije interfejsa klasi. */
	@DeclareParents(value = "milan.panic.master.common.SomeClass",
			defaultImpl = DefaultInterfaceImpl.class)
```

```
public SomeInterface implementedInterface;
@DeclareMixin("milan.panic.master.common.SomeClass")
public SomeInterface declareMixin() {
       return new DefaultInterfaceImpl();
}
/** 2.Dodavanje atributa klasi nije podržano. */
/** 3.Dodavanje metoda klasi nije podržano. */
```

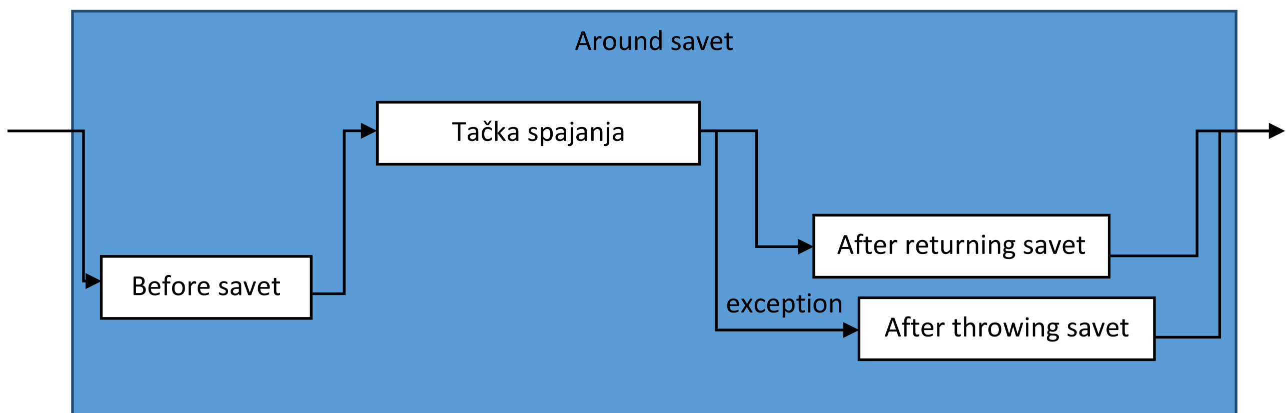
Tačke spajanja

Model tačaka spajanja je centralni pojam unutar aspektno orijentisanog programiranja. Tačka spajanja predstavlja karakteristični imenovani deo u izvršenju programa. AspectJ definiše aspektno ponašanje u tačkama spajanja tako što definiše dodatnu logiku koja se izvršava pre i/ili posle izvršenja tačke spajanja.

- Normalan programski tok:



- Programski tok sa definisanim aspektom:



AspectJ programeru dostupne su sledeće tačke spajanja:

- Izvršenje metoda ili konstruktora

AspectJ ovom tačkom spajanja okružuje sam kôd gde se izvršava metod. Praktično, ovo znači da će AspectJ programeru, od metapodataka koje može da koristi u definisanju saveta, biti dostupna sama klasa u kojoj je metoda definisana, ali ne i ona klasa i metod koji je poziva. Ova tačka spajanja ima najširu primenu. Ukoliko je potrebno definisati neko aspektno ponašanje koje univerzalno važi za metod definisaće se aspektno ponašanje oko ove tačke spajanja.

```
        // Aspektni kod definiše se oko samog tela metoda
        public ReturnType exampleMethod() {
                // Before savet izvršava se pre izvršenja tela metoda
                return new ReturnType();
                // After savet izvršava se posle izvršenja tela metoda
        }
        // Aspektni kod definiše se oko samog tela konstruktora
        public ExampleClass() {
                // Before savet izvršava se pre tela konstruktora
                // Telo konstruktora
                // After savet izvršava se posle tela konstruktora
        }
```

Primer primene: Svi pozivi ka bazi unutar jedne metode treba da se izvrše u istoj transakciji. Napisaće se aspektni kôd oko izvršenja metode, gde će se pre početka izvršenja definisati transakcija, a nakon izvršenja tela metode ta transakcija izvršiti.

- <u>Poziv metode ili konstruktora</u>

Ova tačka spajanja omogućava da se aspektno ponašanje definiše prilikom poziva samog metoda. Pozivalac metoda je mesto gde će aspektni kôd biti umetnut a ne sam metod koji se poziva. Ovo može biti korisno ukoliko aspektno ponašanje ne važi univerzalno za dati metod, već samo za karakteristične delove sistema.

```
public class Caller1 {
        // Aspektni kod definiše se oko poziva metoda
        public void callMethod1(ExampleClass exampleClass) {
                // Before savet izvršava se pre poziva metoda
                exampleClass.exampleMethod();
                // After savet izvršava se poziva metoda
        }
}

public class Caller2 {
        public void callMethod1(ExampleClass exampleClass) {
                // Potpuno nova tačka spajanja koja može, ali ne mora biti, obuhvaćena
                // aspektom
                exampleClass.exampleMethod();
        }
}
```

Primer primene: Ukoliko u programskom sistemu postoji komponenta koja je široko korišćena a čiji su međurezultati potrebni da bi se shvatilo zašto neki delovi sistema greše, idealno je izolovati samo pozive komponente iz tih delova sistema i te međurezultate upisati u programski dnevnik. U suprotnom može se desiti da dođe do zagađenja programskog dnevnika nepotrebnim informacijama usled upisa međurezultata iz celog sistema gde može biti mnogo teže ispitivati neželjeno ponašanje sistema.

- <u>Pristup čitanju vrednosti iz atributa i upisu vrednosti u atribut klase ili objekta</u>

Sami atributi objekata i klasa predstavljaju srž podataka koji se nalaze u sistemu. AspectJ omogućava definisanje aspektnog ponašanja u trenucima kada se ovi podaci čitaju i upisuju.

Primer primene: Ova tačka spajanja može naći primenu u različitim zapisivanjima u programski dnevnik gde je potrebna evidencija o eksplicitnom pristupu vrednim resursima sistema.

- Inicijalizacija klase

Ova tačka spajanja daje programeru mogućnost definisanja aspektnog ponašanja u trenutku kada virtuelna mašina (classloader) učitava klasu. To podrazumeva aspektno ponašanje oko statičkog bloka definisanog unutar klase ili definisanja dodatnog ponašanja čak iako statički blok nije definisan.

```
public class ClassInicialization {
	static {
		// Inicijalizacija klase daje pristup statičkom bloku, čak i ukoliko
		// nije eksplicitno definisan, pa može izvršiti i dodatnu
		// inicijalizaciju klase
	}

	// Definisanje aspektnog ponašanja oko statičkih metoda identično je kao i
	// kod nestatičkih metoda, tj. definiše se oko izvršenja i poziva metoda.
	public static void exampleMetod() {

	}
}
```

Primer primene: Prilikom upisivanja redosleda učitavana klasa od strane Java virtuelne mašine u programski dnevnik.

- Inicijalizacija objekta

Ova tačka spajanja predstavlja kôd unutar konstruktora nakon pozvanog nadkonstruktora (*super*). Ukoliko nadkonstruktor nije eksplicitno definisan (klasa nasledjuje samo klasu Object) ova tačka spajanja je identična izvršenju konstruktora.

```
public class ObjectInitialization extends ExampleClass {
	public ObjectInitialization() {
		super();
		// Aspektno ponašanje koje se definiše pre inicijalizacije objekta
		// Telo konstruktora
		// Aspektno ponašanje koje se definiše posle inicijalizacije objekta
	}

}
```

Primer primene: Potrebna je dodatna inicijalizacija unutar konstruktora.

- Preinicijalizacija objekta

Ova tačka spajanja se jako retko koristi i predstavlja kôd od poziva konstruktora do poziva njegovog nadkonstruktora, odnosno trenutak prosleđivanja argumenata nadkonstruktoru.

```
public class ObjectPreinitialization extends ExampleClass {
	public ObjectPreinitialization() {
		// Aspektno ponašanje koje se definiše pre poziva nadkonstruktora
		super();
		// Aspektno ponašanje koje se definiše posle poziva nadkonstruktora

		// Telo konstruktora
	}
}
```

- <u>Procesiranje izuzetka</u>

Ova tačka spajanja predstavlja kôd za procesiranje izuzetka.

```
public void exceptionMethod() {
		try {
			// ...
		} catch (ExampleException e) {
			// Aspektno ponašanje pre procesiranja izuzetka

			// Procesiranje izuzetka

			// Aspektno ponašanje posle procesiranja izuzetka
		}
	}
```

Primena: Najčešće ukoliko je potreban generički način za procesiranje izuzetaka.

Nisu svi delovi programa u izvršenju tačke spajanja. Pored ovih tačaka spajanja u Java programima mogu da se definišu i druge tačke spajanja: početak petlje, provera uslova pri grananju, naredbe dodele itd... Funkcionalno je moguće da i druge tačke spajanja budu dostupne programeru za umetanje aspektnog ponašanja.

Na primer, jedan od poslova koji je aspektne prirode jeste i praćenje toka izvršenja programa i njegovo zapisivanje u programski dnevnik. Programski dnevnik može biti detaljan onoliko koliko je to želja programera. Nekome bi se možda učinilo primamljivim da ima mogućnost presretanja početaka naredbi grananja i zapisivanja koji je uslov bio zadovoljen. Praktično, ovo bi značilo da je moguće bilo gde u metodi neke klase umetnuti proizvoljan kôd za izvršenje.

AspectJ ovo ne omogućava. Svaka dodatna tačka spajanja narušila bi objektnu prirodu programskog jezika Java.

Izvršenje aspektnog svojstva programa omogućeno je samo izvan osnovnog toka programa. To podrazumeva mogućnost da se ili ceo kôd neke metode izvrši ili nijedan njegov deo. Ovo doprinosi tome da ključne funkcionalnosti sistema ostanu netaknute unutar klasnih metoda i da se ne raseju unutar aspektnog dela kôda.

- <u>Izvršenje aspektnog saveta</u>

Aspektno ponašanje nije ništa drugo do programski kôd koji se definiše na drugom mestu u programu. AspectJ omogućava jos jednu tačku spajanja koja je karakteristična za same aspekte. Ova tačka spajanja predstavlja mogućnost pristupa

samom aspektnom kôdu i definisanje aspektnog ponašanja oko njega. Primenu može naći ukoliko je potrebno rekurzivno izvršenje aspektnog kôda.

Jezik za definisanje tačaka reza

Ono što AspectJ čini najpopularnijim rešenjem za aspektne probleme jeste upravo kompaktan i intuitivan način za definisanje delova programa gde treba umetnuti aspektni kôd a to se čini jezikom za definisanje tačaka reza.

Skup tačaka spajanja u kojima se definiše neko aspektno ponašanje naziva se tačka reza.

Tačka reza može biti anonimna ili imenovana. Anonimna tačka reza, kao i anonimna klasa u programskom jeziku Java, definiše se na mestu korišćenja. Ukoliko je potrebno koristiti tačku reza više puta koristi se imenovana tačka reza. Takođe, dobra je praksa praviti složenije tačke reza od prostijih, da bi se smanjila zabuna prilikom definisanja. Definisanje imena tačke reza:

Tradicionalni AspectJ:

```
pointcut ImeTačkeReza() : definicija tačke reza
```

@AspectJ:

```
@Pointcut("definicija tačke reza")
public void ImeTačkeReza()
```

Gramatika jezika za definisanje tačaka reza

Jezik za definisanje tačaka reza može se definisati gramatikom sa sledećim pravilima izvođenja:

Logička pravila:

- TačkaReza → ImeTačkeReza

  Objašnjenje: Ukoliko je tačka reza već definisana moguće je koristiti njeno ime umesto eksplicitne definicije tačke reza

- TačkaReza → TačkaReza && TačkaReza

  Objašnjenje: Tačka reza dobijena korišćenjem operatora && jeste presek skupova tačaka spajanja koje predstavljaju tačke rezova na kojima je operator primenjen.

- TačkaReza → TačkaReza || TačkaReza

  Objašnjenje: Tačka reza dobijena korišćenjem operatora || jeste unija skupova tačaka spajanja koje predstavljaju tačke rezova na kojima je operator primenjen.

- TačkaReza → ! TačkaReza

  Objašnjenje: Tačka reza dobijena korišćenjem operatora ! jeste komplement skupa tačaka spajanja koje predstavlja tačka reza na kojoj je operator primenjen u skupu svih tačaka spajanja.

Pravila zasnovana na potpisu programskih elemenata:

- TačkaReza → call/execution(ObrazacPotpisaMetoda)

- TačkaReza → call/execution(ObrazacPotpisaKonstruktora)

  Objašnjenje: Tačka reza dobijena korišćenjem operatora call/execution predstavlja skup svih poziva/izvršenja metoda definisanim zadatim obrascem.

- TačkaReza → get/set(ObrazacPotpisaAtributa)

  Objašnjenje: Tačka reza dobijena korišćenjem operatora get/set predstavlja skup svih čitanja/upisivanja vrednosti u atribut/iz atributa definisanih zadatim obrascem.

- TačkaReza → staticinitialization(ObrazacPotpisaTipova)

  Objašnjenje: Tačka reza dobijena korišćenjem operatora staticinitialization predstavlja skup svih inicijalizacija klasa definisanih zadatim obrascem.

- TačkaReza → initialization/preinitialization(ObrazacPotpisaKonstruktora)

  Objašnjenje: Tačka reza dobijena korišćenjem operatora initialization /preinitialization predstavlja skup svih inicijalizacija/preinicijalizacija objekata definisanih zadatim obrascem.

- TačkaReza → handler(ObrazacPotpisaTipova)

  Objašnjenje: Tačka reza dobijena korišćenjem operatora handler predstavlja skup svih procesiranja izuzetaka tipova definisanih obrascem.

- TačkaReza → adviceexecution()

  Objašnjenje: Tačka reza dobijena korišćenjem operatora adviceexecution predstavlja skup svih izvršenja saveta u programu.

- ObrazacPotpisaTipova → ImeEntiteta

  Objašnjenje: Ime entiteta predstavlja kvalifikovano ime klase, interfejsa, anotacije ili aspekta uz mogućnost korišćenja karaktera sa specijalnim značenjem: '' za bilo koji karakter osim tačke, '..' za bilo koji skup karaktera uključujući i tačke (primenu nalazi ukoliko se žele svi tipovi nezavisno od paketa) i '+' kao sufiks za podtipove tipa definisanog obrascem.*

- ObrazacPotpisaAtributa → ModifikatoriPristupa ObrazacPotpisaTipova.ImeAtributa

- ObrazacPotpisaMetoda → ModifikatoriPristupa ObrazacPotpisaTipova.ImeMetoda(ObrazacPotpisaParametara)

- ObrazacPotpisaKonstruktora → ModifikatoriPristupa ObrazacPotpisaTipova .new(ObrazacPotpisaParametara)

  Objašnjenje: Prilikom selekcije konstruktora umesto imena metode koristi se rezervisana reč programskog jezika Java "new".

- ObrazacPotpisaParametara → *Konsultovati zvaničnu Java gramatiku* [12]

- Modifikatori pristupa → *Konsultovati zvaničnu Java gramatiku.* [12]

Takođe, treba napomenuti da osim definisane gramatike AspectJ jezik za definisanje tačaka reza obuhvata i označavanja tačaka reza zasnovane na anotacijama i generičkim tipovima (uvedenim sa verzijom 5 Jave i AspectJ-a), koja

su u skladu sa definisanjem osnovnih tipova tačaka reza i samim tim gramatička pravila neće biti definisana.

Pravila zasnovana na leksičkoj strukturi programa:

- TačkaReza → within(ObrazacPotpisaTipova)

  Objašnjenje: Tačka reza dobijena korišćenjem operatora within predstavlja sve tačke spajanja unutar tela aspekta ili klasa definisanih obrascem i unutar bilo kojih ugnježdenih klasa.

- TačkaReza → withincode(ObrazacPotpisaMetoda)
- TačkaReza → withincode(ObrazacPotpisaKonstruktora)

  Objašnjenje: Tačka reza dobijena korišćenjem operatora withincode predstavlja sve tačke spajanja unutar tela metoda/konstruktora definisanih obrascem.

Pravila zasnovana na programskom toku:

Programski tok tačke spajanja podrazumeva sve tačke spajanja koje će se direktno izvršiti dok je tačka spajanja na steku, tj. ova tačka spajanja direktno ili indirektno poziva kôd za izvršenje tih tačaka spajanja. Jasno je da programski tok nije moguće statički odrediti jer se može razlikovati od izvršenja do izvršenja, tako da je potrebna provera za vreme izvršenja programa.

- TačkaReza→ cflow/cflowbellow(TačkaReza)

  Objašnjenje: Tačka reza dobijena korišćenjem operatora cflow/cflowbellow predstavlja sve tačke spajanja koje se nalaze u programskom toku definisane tačke reza uključujući/ne uključujući tačku reza koja ih definiše.

  Najšešće se upotrebljava da bi se obezbedilo nerekurzivno izvršenje aspekata: Ako se aspekt definiše na nekoj tački reza onda TačkaReza && !cflowbellow(TačkaReza) sigurno nije rekurzivno.

- TačkaReza → args(ObrazacPotpisaParametara)

  Objašnjenje: Tačka reza dobijena korišćenjem operatora args predstavlja sve tačke metode i konstruktore kojima su prosleđeni parametri sa zadatim obrascem. Parametri koji se proveravaju predstavljaju prosleđene parametre prilikom izvršenja programa, a ne prilikom definisanja metode, što daje mogućnost konkretizacije aspektnog ponašanja za pojedine podtipove parametara. Jako često kombinuje sa operatorom execution/call da bi se konkretizovalo ponašanje i smanjila potreba pa prevelikim proveravanjem tipova za vreme izvršenja programa.

- TačkaReza → this(ObrazacPotpisaTipova)

  Objašnjenje: Tačka reza dobijena korišćenjem operatora this predstavlja sve tačke spajanja kod kojih specijalna promenljiva "this" jeste jedan od tipova definisanih obrascem. Najčešće se koristi da bi se obezbedio dodatni kontekst u tački reza, tj. omogućio pristup promenljivoj this prilikom definisanja aspektnog ponašanja.

- TačkaReza → target(ObrazacPotpisaTipova)

Objašnjenje: Tačka reza dobijena korišćenjem operatora target predstavlja sve tačke spajanja kod kojih je objekat u kome je definisan kôd tačke spajanja jedan od tipova definisanih obrascem. Najčešće se koristi u kombinaciji sa call operatorom pošto je tada objekat target u stvari objekat u kome je definisana metoda da bi se obezbedio dodatni kontekst u tački reza.

Uslovne tačke reza

- TačkaReza → if(Bulovski izraz)

  Objašnjenje: Selektuje one tačke spajanja kod kojih je bulovski izraz tačan. Sve promenljive u bulovskom izrazu moraju biti određene u vreme izvršenja programa, tj. ili je definisan kao izraz sa statičkim promenljivima aspekta u kome se definiše tačka reza ili drugi delovi tačke reza čiji je if deo moraju obezbediti da su sve promenljive unutar bulovskog izraza vidljive u trenutku izvršenja.

Tkač (Weaver)

Funkcionalnost aspektnog sistema omogućava se povezivanjem aspektnog kôda sa ostalim modulima sistema. AspectJ programi izvršavaju se na Java platformi tj. Java virtuelnoj mašini.

Java virtuelna mašina

Java virtuelna mašina (JVM) predstavlja apstraktnu računarsku mašinu i ključnu komponentu u uspehu Java programskog jezika. Java programski jezik jeste portabilan programski jezik, što znači da je dostupan na mnogim platformama. Programski jezik Java prevodi se Java prevodiocem i dobijeni prevedeni programski kôd izvršava se na JVM.

Kao i svaka druga računarska mašina JVM poseduje svoj mašinski jezik. Jezik JVM naziva se *bajt* kôd smešten u *.class* format i on je platformski nezavistan i definisan specifikacijom. Bajt kôd se može shvatiti kao univerzalan asemblerski jezik za Java virtuelnu mašinu.

Java programski jezik i prva implementacija JVM nastali su u kompaniji *Sun Microsystems* koja je samim tim objavljivala i njihove standarde. Nakon preuzimanja Sun kompanije od strane Oracle korporacije, Oracle je na sebe preuzeo i objavljivanje pomenutih specifikacija i trenutno je definisana verzija 8 [13], koja definiše arhitetkturu JVM, standardni format bajt kôda, proces inicijalizacije mašine, učitavanja i izvršenja programa kao i 149 instrukcija koje Java virtuelna mašina treba da bude u stanju da izvrši da bi zadovoljila standard.

Arhitektura Java virtuelne mašine:

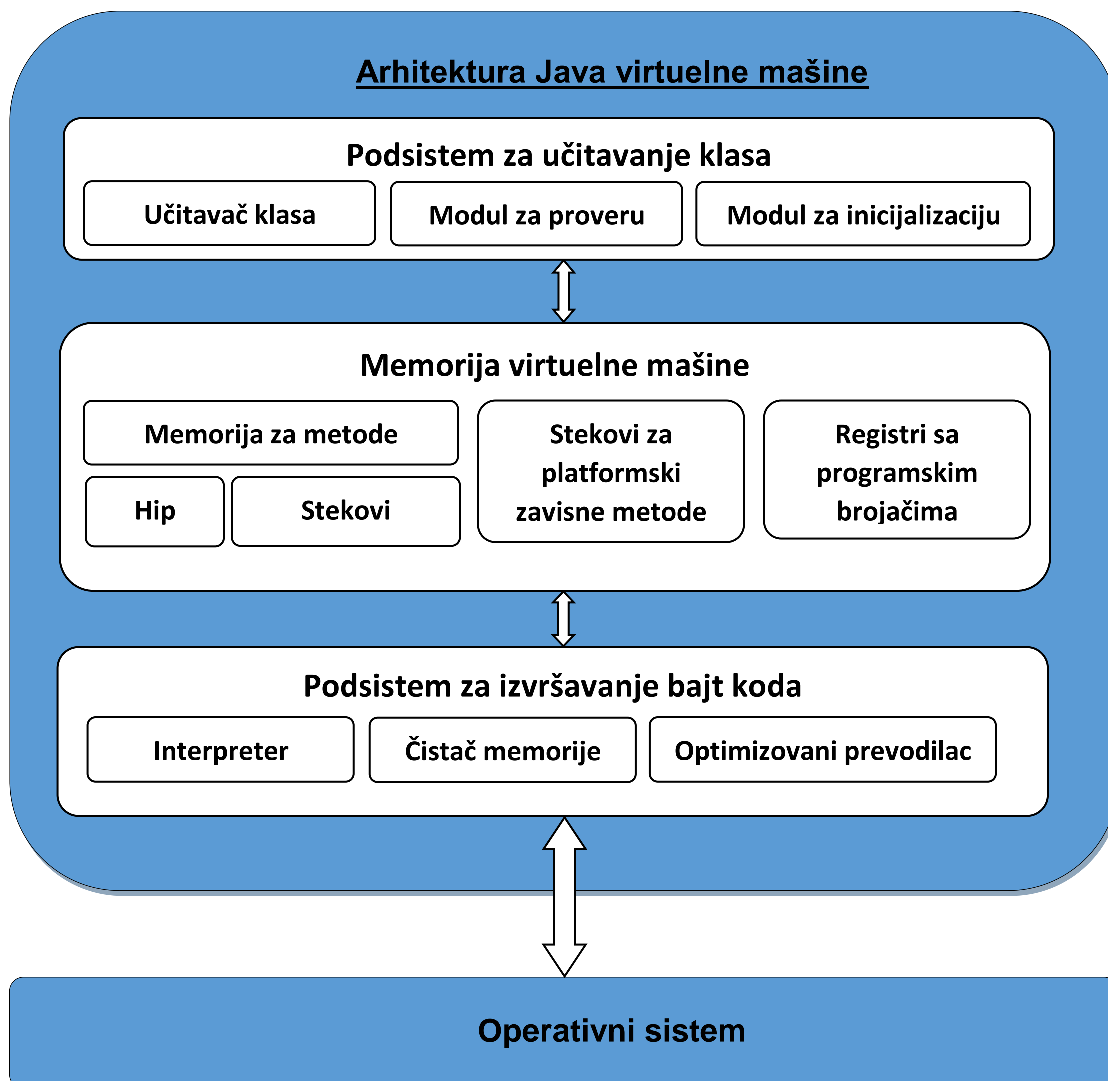
Java virtuelna mašina sastoji se iz tri glavna podsistema: podsistema za učitavanje klasa, memorije virtuelne mašine i podsistema za izvršavanje bajt kôda (engl. execution engine).

- Podsistem za učitavanje klasa zadužen je za učitavanje bajt kôda u memoriju virtuelne mašine. Podsistem za učitavanje klasa sastoji se od učitavača

klasa (engl. classloader), zatim modula koji proverava bajt kôd i modula koji zauzima memoriju za statičke promenljive i postavlja ih na odgovarajuće vrednosti.

- Memorija virtuelne mašine podrazumeva različite tipove memorijskih sektora: sektor za definisanje metoda, hip, lokalni stek za svaku nit, registre za programske brojače i stekove za izvršenje platformski zavisnih metoda.

- Podsistem za izvršavanje bajt kôda učitava bajt kôd iz memorije, vodi računa o programskom brojaču za svaku nit i izvršava redom instrukcije definisane bajt kôdom. Instrukcije Java virtuelne mašine mogu se svrstati u sledeće kategorije: čitanje i upis u memoriju (primer: istore, upisivanje celobrojne vrednosti u lokalnu promenljivu), aritmetičke i logičke instrukcije (primer: iadd, sabiranje celobrojnih vrednosti), konverzija primitivnih tipova (primer: i2l, konverzija int vrednosti u long), kreiranje objekata i manipulacija objektma (new, kreiranje novog objekta na hipu), kontrolne operacije na steku (primer: goto; ifeq), pozivanje metoda i povratak iz metoda (primer: invokestatic, pozivanje statičke metode; areturn, povratak iz metode). Najvažniji delovi ovog podsistema su interpreter, čistač memorije i optimizovani prevodilac za česte instrukcije. Ovaj podsistem izvršava instrukcije tako što preko interfejsa za komunikaciju sa platformski zavisnim metodama, JNI (engl. Java Native Interface), komunicira sa operativnim sistemom koji izvršava mašinski kôd.



Kompanija Oracle pored definisanja standarda obezbeđuje i implementaciju Java prevodioca i virtuelne mašine za većinu značajnijih operativnih sistema. Pošto je specifikacija javna, moguće je izvršenje bajt kôda na bilo kojoj platformi ako se na toj platformi implementira Java virtuelna mašina u skladu sa standardom. Ovo je dovelo da pored mašine koju Oracle nudi, postoji mnogo implementiranih alternativa koje su otvorenog kôda za popularne platforme, kao i implementacija JVM za specifične platforme koje Oracle ne podržava, što i jeste uzrok velike popularnosti programskog jezika Java.

Iako je JVM nastala sa idejom da se na njoj izvršava bajt kôd prevedenog Java programskog jezika, bajt kôd koji se izvršava na JVM potpuno je nezavistan od Java programskog jezika. Java prevodiocem se Java programski jezik prevodi u instrukcije Java virtuelne mašine prilično direktno:

```
/**Primer dobijenog bajt koda prilikom prevođenja*/
/**
 * Za prevođenje iz bajt koda korišćen je javap disasembler,
 * standardni deo Oracle Java JDK distribucije.
 */

/** Compiled from "Example.java"
public class milan.panic.master.bytecode.Example {
  public int method(int);
    Code:
       0: new           #3   // kreiranje objekta na hipu
       3: invokespecial #8   // Poziv konstruktora
       6: iload_1            // Učitavanje vrednosti iz lokalne promenljive
       7: iconst_2           // Učitavanje konstante
       8: iadd               // Sabiranje
       9: ireturn            // Povratak iz metoda
} */
package milan.panic.master.bytecode;
public class Example {
	public int method(int a){
		new Object();
		return a+2;
	}
}
```

Sam Java prevodilac zadužen da prevedeni bajt kôd bude validan i predstavlja upravo program definisan u programskom jeziku Java. Konceptualno, Java virtuelna mašina pored programa koje je moguće prevesti Java prevodiocima, od samog svog nastanka u stanju je da izvršava i druge validne programe u bajt kôdu koji se Java prevodiocem ne mogu dobiti.

Ovo je nakon nekon vremena od nastanka Jave, naročito usled nekih nedostataka koje poseduje Java programski jezik, rezultiralo time da počne da se pravi jasnija razlika između Jave programskog jezika i Java virtuelne mašine. Tada počinje da se razmišlja o implementiranju drugih programskih jezika koji se mogu izvršavati na JVM i Java virtuelna mašina u ovom kontekstu postaje platforma za izvršenje raznih programskih jezika (Java platforma).

Do danas nastao je niz programskih jezika koji se izvršavaju na JVM: Closure (funkcionalni lisp dijalekat) [14], Scala (jezik za funkcionalnim i objektnim karakteristikama, jako popularan u današnje vreme) [15], Apache Groovy (skript

jezik) [16] kao i implementacije postojećih programskih jezika za java platformu od kojih su u današnje vreme najkorišćeniji JRuby (implementacija prevodioca za Ruby programski jezik) i Jython (implementacija Python prevodioca). Mnogi od njih pored portabilnosti koju podržava Java platforma iskoristili su i ogromnu količinu već napisanog kôda koji je već razvijen za Java programski jezik tako što su omogućili integraciju već postojećih biblioteka u novonapisane jezike.

AspectJ program takođe treba prevesti u validan bajt kôd pri čemu će u dobijenom programu biti implementirane međumodularne funkcionalnosti. Pravljenje aspektnog programa od AspectJ kôda konceptualno podrazumeva dve faze: prevođenje i tkanje (engl. weaving).

Faza prevođenja

Prevođenje predstavlja prevođenje aspekata i java klasa u bajt kôd.

Prevođenje @AspectJ aspekata moguće je i bilo kojim drugim Java prevodiocem, zato što oni predstavljaju validan Java kôd, tako da je u potpunosti jasno da se oni prevode identično bilo kom Java kôdu. Ovo podrazumeva da se aspekti prevode u klase sa istim imenom, saveti i tačke reza prevode se u metode sa anotacijama, dok se metapodaci koji opisuju tačke reza ne modifikuju.

Tradiconalni AspectJ aspekti ne predstavljaju validan Java kôd i upotreba posebnog prevodioca, *ajc* [10], je neophodna. Rezultat prevođenja aspekata ajc prevodiocem analogan je prevođenju @AspectJ aspekata: Aspekti se direktno mapiraju u klase, saveti u metode. Jedina razlika u odnosu na prevođenje @AspectJ aspekata jeste što se prilikom generisanja aspekata ajc prevodiocem generišu generička imena aspektnih konstrukcija koja će pomoći u njihovom razlikovanju od tradicionalnog Java kôda u fazi tkanja.

Faza tkanja

Kao što je predstavljeno, prevedeni aspekt ne predstavlja ništa drugo do Java klasu, a poznato je da Java klasa ne može da omogući modularizaciju međumodularne funkcionalnosti. Jasno je da sve što se opisuje aspektnim kôdom može da se opiše i bez njega zato što je aspektni kôd prisutan zbog potreba bolje organizacije kôda.

Prevođenje aspekata samim tim nije dovoljno da aspektni sistem postane funkcionalan, potrebno je naknadno povezati prevedeni aspekt i ostale delove sistema, tj. utkati funkcionalnost aspetka u sistem.

Tkanje aspektnog kôda konceptualno je jako slično linkovanju u programskim jezicima koji se prevode na mašinski jezik. AspectJ tkač uzima bajt kôd u kome se nalaze definisani aspekti (klase) i Java kôd (takođe klase) i proizvodi skup klasa u bajt kôdu koji imaju utkane aspekte u sebi (pozive ili ceo kôd odgovarajućih metoda aspekta smešta na mesta definisana tačkama reza). Moć aspektno orijentisanog programiranja nije u tome što radi stvari koje je nemoguće odraditi ručno, već upravo u tome što automatizuje implementaciju međumodularne funkcionalnosti u zavisnosti od jezika tačaka reza.

Zbog same arhitekture Java virtuelne mašine, postoji više mesta gde se može obezbediti tkanje saveta. U zavisnosti od toga u kom će se trenutku desiti tkanje

aspekata AspectJ omogućava pravljenje aspektnog programa na tri načina, pri čemu se garantuje da je kôd u izvršenju identičan bez obzira na to koji način tkanja se primenjuje [17]:

- Tkanje se dešava u trenutku prevođenja - source code weaving

Tkanje u trenutku prevođenja omogućava formiranje gotovog aspektnog sistema u trenutku prevođenja aspekata, tako što spaja proces prevođenja i tkanja u jedan proces. Ovo je tradicionalan način i identičan je prevođenju Java programa, pri čemu je potrebno u potpunosti zameniti Java prevodilac AspectJ prevodiocem.

- Tkanje se dešava nakon prevođenja - byte code weaving

Tkanje nakon prevođenja, podrazumeva da su svi delovi sistema (aspekti i java kôd) već prevedeni. Ovaj proces će uzeti postojeći bajt kôd i njega utkati. Najčešće se koristi ukoliko već postoje prevedene biblioteke kojima se želi dodati aspektno ponašanje, pri čemu izvorni kôd nije dostupan.

- Tkanje se dešava prilikom učitavanja klasa u Java virtuelnu mašinu - load time weaving

Prethodna dva načina su konceptualno identična i analogno se mogu uopštiti na većinu programskih jezika koji se prevode. Zbog same arhitekture Java virtuelne mašine, koja podrazumeva postavljanje sopstvenih učitavača klasa (koji se mogu implementirati tako da automatski modifikuju originalan bajt kôd), moguć je još jedan način tkanja aspekata. Tkanje prilikom učitavanja uklanja potrebu za eksplicitnim tkanjem aspekata, jer odlaže tkanje aspekata do samog učitavanja klasa u virtuelnu mašinu gde specijalno napravljen učitavač klasa ima ulogu tkača aspekata. Prilikom pokretanja Java virtuelne mašine potrebno je obezbediti postojanje AspectJ biblioteke *acpectjweaver.jar* u kojoj postoji specijalna implementacija Java agenta koji će obezbediti izvršenje aspektno orijentisanog kôda.

Striktno govoreći, jasno je da tradicionalni AspectJ nije Java programski jezik već da predstavlja samo jezik Java platforme zato se ne prevodi Java prevodiocem, već ima sopstveni prevodilac. @AspectJ sa druge strane ne može biti jasno definisan, jer je moguće prevesti ga Java prevodiocem ali potrebna je manipulacija bajt kôdom (dodatnim tkanjem van java prevodioca ili prilikom učitavanja u virtuelnu mašinu) da aspektno orijentisani sistem unutar Java aplikacije postane funkcionalan.

Analiza implementacije aspektnog sistema AspectJ-a

Radi potpune demistifikacije implementacije aspektno orijentisanog sistema koji nudi AspectJ biće demonstriran konačan sistem koji se dobija prevođenjem AspectJ programa, tako što će biti jedan jednostavan aspektno orijentisani program biti analiziran. Radi uprošćenja analize konačan bajt kôd je preveden nazad u Java kôd korišćenjem Java dekompajlera (korišćen je Java dekompajler otvorenog kôda JD [18]).

```
/**Originalan aspekt*/
package milan.panic.master.weaving;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class ExampleAspect {
	@Before("execution(* Program.method(..))")
	public void beforeExecution() {
		System.out.println("beforeExecution");
	}
	@Before("call(* Program.method(..))")
	public void beforeCall() {
		System.out.println("beforeCall");
	}
}
```

```
/**Prevedeni aspekt*/
package milan.panic.master.weaving;
import org.aspectj.lang.NoAspectBoundException;
/**
 * Aspekt se prevodi u klasu
 */
public class ExampleAspect {
	private static Throwable ajc$initFailureCause;
	public static ExampleAspect ajc$perSingletonInstance;
	/**
	 * Metod koji se koristi za dohvatanje instance aspekta
	 */
	public static ExampleAspect aspectOf() {
		if (ajc$perSingletonInstance == null) {
			throw new NoAspectBoundException(
					"milan.panic.master.weaving.ExampleAspect",
					ajc$initFailureCause);
		}
		return ajc$perSingletonInstance;
	}
	/**
	 * Metod koji provera da li aspekt postoji
	 */
	public static boolean hasAspect() {
		return ajc$perSingletonInstance != null;
	}
	/**
	 * Metod za kreiranje instance aspekta
	 */
	private static void ajc$postClinit() {
		ajc$perSingletonInstance = new ExampleAspect();
	}
	/**
	 * Statički blok u kom se kreira instanca aspekta
	 */
	static {
		try {
			ajc$postClinit();
		} catch (Throwable localThrowable) {
			ajc$initFailureCause = localThrowable;
		}
	}
```

```
        /**
         * Metodi klase koja je prevedena od aspekta su identični savetima
         */
        public void beforeExecution() {
                System.out.println("beforeExecution");
        }
        public void beforeCall() {
                System.out.println("beforeCall");
        }
}
```

```
/**Originalan program*/
package milan.panic.master.weaving;

public class Program {
        public static void main(String[] args) {
                new Program().method();
        }
        public static void method() {
                System.out.println("method");
        }
}
```

```
/**Program u kome su utkani saveti */
package milan.panic.master.weaving;

public class Program {
        public static void main(String[] args) {
                new Program();
                /**
                 * Aspektni savet utkan je pre poziva metoda, kao što je
                 * definisano jezikom
                 */
                ExampleAspect.aspectOf().beforeCall();
                /**
                 * Regularan poziv metoda
                 */
                method();
        }
        public static void method() {
                /**
                 * Aspektni savet utkan je pre izvršenja metoda(na početku
                 * izvršenja metoda) kao što je definisano jezikom
                 */
                ExampleAspect.aspectOf().beforeExecution();
                System.out.println("method");
        }
}
```

Iz priloženog, jasno se zaključuje da konačan sistem nije ništa drugo do bilo koji drugi Java program ukoliko bi se ručno upisivala međumodularna funkcionalnost unutar mesta definisana jezikom tačaka reza.

Spring AOP

Spring okvir

Spring okvir predstavlja skup biblioteka za Java programski jezik uz pomoć kojih je moguća dobra organizacija velikih poslovnih aplikacija.

Prvu verziju Spring okvira implementirao je 2002. godine Rob Džonson kao dodatak svojoj knjizi „Eksperti, jedan na jedan - J2EE dizan i implementacija", u kojoj je sažeo najbolju programersku praksu za programiranje na Java platformi. Nakon dugogodišnjeg rada na Java platformi, smatrao je da iako J2EE standard dobro zamišljen mnogi projekti propadaju zbog nepostojanja dobrog prenosa znanja eksperata koji rade na uspešnim projektima. 2003. godine Spring je postao dostupan sa Apache 2.0 licencom [19] i tada počinje njegov vrtoglavi razvoj i popularnost pod uticajem „open source" zajednice. Spring je osmišljen da bude dopuna Java standarda za poslovne aplikacije dobrom praksom i uveliko je kompatibilan sa njim, ali danas se na Spring više gleda kao na alternativu preobimnom i prekompleksnom J2EE stadnardu, naročito delu koji se tiče rada sa Java zrnima (engl. Enterprise Java Beans).

Spring se može koristiti u bilo kojoj Java aplikaciji. Kao projekat, Spring okvir sastoji se od više modula koji su u velikoj meri nezavisni, tako da svaka aplikacija može da koristi samo deo Spring okvira koji joj je potreban. Danas, sam Spring okvir obuhvata preko 20 zvaničnih modula, kao i ogroman broj proširenja koja omogućavaju intergraciju sa različitim tehnologijama.

Najveću popularnost i primenu Spring je našao kao okruženje za razvoj veb aplikacija. Može se reći da za Java platformu on predstavlja nezvanični standard i potpuno rešenje za njihov razvoj, jer se lako integriše sa Java servletima a same aplikacije razvijaju se na neverovatno standardan i udoban način. Udobnost u programiranju koju nudi Spring pre svega se ogleda u mogućnostima korišćenja već predefinisane hijerarhijske organizacije aplikacije uz pomoć modula Spring MVC, deskriptivnom opisivanju komunikacije sa bazama podataka preko skupa modula Spring Data pri čemu u većini slučajeva nije potrebno opisivanje konkretnog kôda za dohvatanje podataka, relativno jednostavnom načinu osiguravanja aplikacija preko modula Spring Security itd.

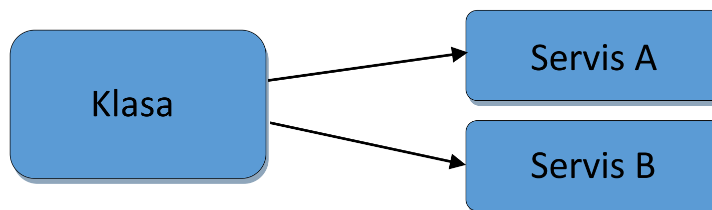
Programeri Spring okvira prilično nezadovoljni postojećim načinima organizovanja aplikacija na Java platformi uvideli su šansu u pravljenju okvira koji omogućava jednostavan način za modlarizaciju poslovnih aplikacija. Spring omogućava uniforman način za modularizaciju aplikacija pri čemu je proširenje aplikacije jako jednostavno. Modularizacija u Spring okviru se omogućava razdvajanjem ključnih i međumodularnih funkcionalnosti.

Svaka Spring aplikacija mora da ubaci minimalno jezgro koje omogućava rad ostalim modulima Springa. Jezgro Springa omogućava razdvajanje ključnih funkcionalnosti aplikacije. Zasnovano je na principu inverzije kontrole i sadrži aplikativni sloj koji omogućava programiranje tzv. spring zrna i rad sa njima.

Inverzija kontrole - Princip na kome radi jezgro Spring okvira

Inverzija kontrole predstavlja način dizajna aplikacija koji omogućava nezavisnost ključnih funkcionalnosti između modula. Možda čudno zvuči da je ključne funkcionalnosti u različitim modulima teško držati nezavisnim, ipak praksa pokazuje da intuitivan način za razvoj aplikacija dovodi do međuzavisnosti modula, pri čemu održavanje aplikacije postaje sve teže i teže kako aplikacija raste.

Konceptualan primer ključnih zavisnosti ogleda se u sledećem [20]: Klasa (komponenta) zavisi od dva servisa (servis odnosno komponenta predstavljaju klase, samo su zbog konceptualne razlike u primeru razlikovani) čiji su konkretni tipovi poznati u vreme kompilacije. Nameće se pitanje, kako dizajnirati aplikaciju tako da konkretni tipovi servisa budu dostupni klasi u vreme izvršenja a da aplikacija bude što fleksibilnija?



Pokušaj direktne implementacije (definisati konkretne tipove servisa unutar klase) ove situacije nameće sledeće probleme:

- Ukoliko treba zameniti ili ažurirati servise mora se menjati sam kôd klase.
- Konkretni tipovi servisa moraju biti dostupni u vreme kompilacije klase.
- Klasu je jako teško testirati izolovano zato što poseduju direktne reference na servise od kojih zavisi. Ovo znači da je zavisnosti nemoguće zameniti njihovom imitacijom što je čest slučaj jer je tada lako testirati funkcionalnost samo tražene klase.
- Sve klase koje koriste ove servise imaju kôd koji će se ponavljati za kreiranje, lociranje i/ili korišćenje ovih zavisnosti.

Inverzija kontrole predstavlja način dizajna aplikacija gde se konkretni tipovi servisa od kojih klase zavise ostavljaju nekom eksternom modulu na odlučivanje, pri čemu će klase neće biti svesne kokretnih tipova servisa u vreme kompilacije. Inverzija kontrole predstavlja princip koji može biti implementiran u mnogim paradigmama i jezicima. Sam termin inverzija kontrole popularizovan je od strane Martina Fowlera [21] i Roberta C. Martina, velikih pobornika kvalitetnog dizajna aplikacija.

U programskom jeziku Java inverzija kontrole može se implementirati na više načina od kojih se izdvajaju i najviše su korišćeni:

- <u>Servis lokator projektni uzorak</u>

Servis lokator podrazumeva da postoji komponenta koja zna kako da dođe do kokretne implementacije svih servisa koje će aplikacija potencijalno koristiti.

Komponente koje koriste servise pozivaju lokator servisa koji iz registra svih servisa poziva odgovarajuću implementaciju.

Ovaj pristup prirodno stavlja ogroman teret na lokator servisa koji mora biti svestan svih servisa i omogućiti pravilan rad ostalih delova sistema. Takođe mnogi delovi sistema postaju direktno zavisni od lokatora servisa čiji tip takođe mora biti dostupan u vreme izvršenja, a to opet treba na neki način obezbediti (što liči na početni problem, samo je u pitanju jedna komponenta).

- Umetanje zavisnosti

Umetanje zavisnosti inverziju kontrole realizuje tako što konkretne implementacije servisa prosleđuje prilikom kreiranja ili inicijalizacije objekata koji zavise od servisa. Ovaj pristup sličniji je tradicionalnoj implementaciji aplikacija nego servis lokator uzorka, jer za razliku od njega, svi objekti koji koriste servise imaju reference na servise unutar svojih klasa.

U zavisnosti od toga kako se umeću zavisnosti u aplikaciju prepoznaju se konstruktorski, seterski i intefejsni tip umetanja zavisnosti.

Konstruktorski tip zavisnosti umeće prilikom kreiranja objekta, u konstruktoru kome se kao parametri proseđuju implementacije servisa. Ovaj način je najpragmatičniji i najsigurniji jer obezbeđuje to da je korisnik servisa uvek u validnom stanju, jer u suprotnom korisnik servisa neće ni biti kreiran.

```
package milan.panic.master.dependency.injection;

public class ConstructorInjectionClient {

	private Service service;

	/**
	 * Konstruktorski princip umetanja zavisnosti: Objekat se kreira
	 * konstruktorom čiji su parametri svi interfejsi servisa od kojih zavisi, a
	 * prilikom pozivanja konstruktora proseđuju se konkretni tipovi servisa.
	 */
	public ConstructorInjectionClient(Service service) {
		this.service = service;
	}

}
```

Seterski tip podrazumeva postojanje pojedinačnog metoda za postavljanje svakog servisa, preko koga se objektu prosleđuju konkretne implementacije servisa. Seterski pristup postavljanju zavisnosti daje mogućnost veće fleksibilnosti sistema, pošto se daje mogućnost promene implementacije servisa tokom postojanja objekta. Zauzvrat povećava se mogućnost greške usled nepostojanja implementacije servisa za vreme izvršenja programa i postojanje velikog broja seterskih metoda.

```java
package milan.panic.master.dependency.injection;

public class SetterInjectionClient {

	private Service service;

	/**
	 * Seterski princip umetanja zavisnosti. Zavisnost se umeće
	 * metodom, mora se obezbediti postojanje svih implementacija
	 * servisa u vreme njihovog korišćenja. Da bi se ovo obezbedilo
	 * najčešće se pozivaju setuju svi servisi u vreme inicijalizacije
	 * objekta, ali ovaj princip daje mogućnost da se naknadno
	 * implementacija servisa promeni.
	 */
	public void setService(Service service) {
		this.service = service;
	}

}
```

Interfejsni tip umetanje zavisnosti obezbeđuje implementacijom metoda interfejsa definisanih za postavljanje zavisnosti. Naizgled, ovaj način se ne razlikuje od seterskog pristupa za umetanje zavisnosti, ali glavna razlika je u tome što komponente koje postavljaju zavisnosti mogu biti potpuno nesvesne tipova kojima će postavljati implementacije servisa, jer im mogu pristupiti preko interfejsa, što je i glavna prednost ovakvog pristupa.

```java
package milan.panic.master.dependency.injection;

/** Interfejs preko koga se umece servis */
public interface ServiceSetter {
	public void setService(Service service);
}
```

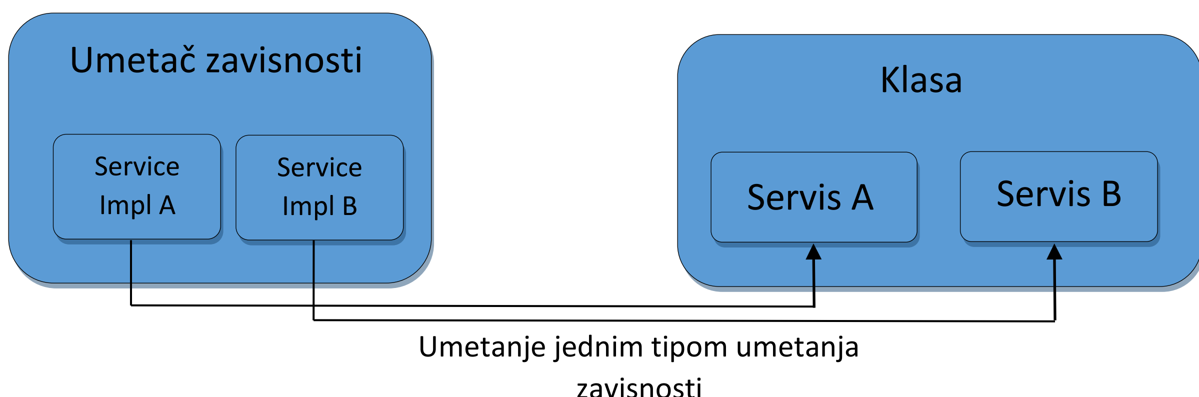
```java
package milan.panic.master.dependency.injection;

public class InterfaceInjectionClient implements ServiceSetter {

	private Service service;

	/** Umetanje servisa preko interfejsa. */
	@Override
	public void setService(Service service) {
		this.service = service;
	}

}
```

Može se reći uz pomoć umetanja zavisnosti objekti postaju konfigurabilni, jer jedino što oni podrazumevaju jeste izvršenje sopstvenog kôda, dok apstrahuju objekte od kojih zavise.

Ovako definisane zavisnosti moguće je umetati direktno, što funkcioniše intuitivno za male programe.

```java
package milan.panic.master.dependency.injection;

/**
 * Aplikacije koje koriste princip umetanja zavisnosti konceptualno
 * treba da poseduju mesto(idealno samo jedno) u kodu, modul tzv.
 * umetač zavisnosti, najčešće programski okvir(eng. framework) oko
 * aplikacije, gde se konkretne implementacije servisa umeću u kod i
 * pokreću ostale module programa.
 */
public class Injector {

	/**
	 * Trivijalna implementacija umetača zavisnosti unutar main metode
	 * programa.
	 */
	public static void main(String[] args) {

		Service service = new ServiceImpl();

		ConstructorInjectionClient constructorInjectionClient =
				new ConstructorInjectionClient(service);

		SetterInjectionClient setterInjectionClient =
				new SetterInjectionClient();
		setterInjectionClient.setService(service);

		/**
		 * Glavna prednost umetanja zavisnosti preko interfejsa jeste
		 * što umetač ne mora da vodi računa o tipu objekta kome umeće
		 * zavisnost, već dohvata objekat ako je instanca interfjesa.
		 * Mana je to što objekti potencijalno mogu da zavise od mnogo
		 * servisa što može dovesti do zabune i prljanja koda.
		 */
		ServiceSetter serviceSetter = new InterfaceInjectionClient();
		serviceSetter.setService(service);
	}
}
```

Ipak, ovaj ad hoc pristup sigurno bi doveo do zabune u većim programima, pa se najčešće pristupa nekom obliku automatizacije umetanja zavisnosti, pa konceptualno, programi koji koriste umetanje zavisnosti imaju još jednu komponentu, takozvani umetač zavisnosti.



Umetanje jednim tipom umetanja zavisnosti

Može se odmah uvideti da implementacija umetača zavisnosti za velike aplikacije neće predstavljati trivijalan zadatak. Za aplikacije u kojima postoji ogroman broj objekata i zavisnosti biće potrebno čuvati reference na sve njih, ukoliko je potrebno kreirati objekte koji nedostaju, u zavisnosti od tipova umetati reference na ogroman broj mesta, obezbediti pravilno uništenje objekata (ukoliko objekat ne treba više da postoji mora se obezbediti da niko više ne referencira na njega, što je teško ukoliko druge komponente zavise od njega i imaju njegovu referencu) i sve u svemu obezbediti pravilno funkcionisanje celog sistema za vreme pokretanja, izvršenja i završetka programa. Navedeni razlozi doveli su do toga da postoje projekti koji će programerima obezbediti okruženje koje obezbeđuje umetanje zavisnosti koje je do neke mere apstrahovalo sve probleme koje bi programer imao prilikom razvoja aplikacije.

Jezgro Spring programskog okvira predstavlja upravo jednu implementaciju umetača zavisnosti. Svoju snagu Spring crpi upravo iz činjenice što je ovo jezgro u potpunosti automatizovalo i apstrahovalo proces umetanja zavisnosti. Spring umeće objekte u zavisnosti od definisane konfiguracije, tako da programer može da se posveti relativno direktnoj implementaciji svoje aplikacije, pri čemu nema potrebe da vodi računa o svim ranije pomenutim problemima, jer će zadržati mogućnost jednostavne promene dizajna i konfiguracije modula u svom programu.

Spring aplikacija

Spring aplikacija predstavlja Java aplikaciju kojoj je negde u programskom kôdu (najčešće prilikom startovanja, u *main* metodu) definisano učitavanje Spring jezgra. Kao što je navedeno Spring jezgro predstavlja implementaciju umetača zavisnosti i ono će povezati sve komponente sistema u sazdati upotrebljiv sistem.

Sam umetač zavisnosti u Spring terminima naziva se Spring kontejner dok se svi objekti za čije je kreiranje, umetanje i uništavanje on zadužen nazivaju Spring zrnima (engl. Spring beans). Spring kontejner radi tako što skenira definisanu konfiguraciju koja se nalazi u jednom od dva oblika metapodataka, xml datotekama ili Java anotacijama i u zavisnosti od toga automatski pravi sistem u kome su sve zavisnosti umetnute.

```
package milan.panic.master.spring;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import milan.panic.master.dependency.injection.ConstructorInjectionClient;
import milan.panic.master.dependency.injection.Service;
import milan.panic.master.dependency.injection.SetterInjectionClient;

/** Spring automatsko umetanje zavisnosti */
public class SpringProgram {
	/**
	 * ApplicationContext je centralna klasa unutar Spring projekta i
	 * predstavlja kontainer koji sadrži sve kreirane, povezane Spring
	 * objekte dobijene od predefinisanih metapodataka korišćenjem
	 * umetanja zavisnosti. Valja napomenuti da će Spring sve objekte
```

```java
         * kreirati kao singltone ukoliko nije drugačije navedeno u
         * konfiguraciji.
         */
        static ApplicationContext springContainer;

        public static void xmlDependencyInjection() {
                /**
                 * Učitavanje metapodataka i kreiranje Spring kontejnera preko
                 * konfiguracije definisane unutar xml fajla.
                 */
                springContainer = new ClassPathXmlApplicationContext(
                                "dependancy-injection.xml");
        }
        public static void annotationDependencyInjection() {
                /**
                 * Učitavanje metapodataka i kreiranje Spring kontejnera
                 * konfiguracije definisane unutar klase koja sadrži
                 * specijalne konfiguracione anotacije definisane unutar Spring
                 * projekta.
                 */
                springContainer = new AnnotationConfigApplicationContext(
                                SpringConfiguration.class);

        }

        public static void main(String[] args) {
                if (args.length == 1
                                && args[0].equalsIgnoreCase("xml")) {
                        xmlDependencyInjection();
                } else {
                        annotationDependencyInjection();
                }

                /**
                 * Objekti su automatski kreirani, umetnute su sve definisane
                 * zavisnosti pri čemu su provereno postojanje svih
                 * zavisnosti, kompatibilnost tipova itd. Odavde na dalje,
                 * moguće je koristiti objekte iz Spring kontainera. Jedan od
                 * načina jeste korišćenja je dohvatanje Spring zrna
                 * eksplicitno na sledeći način:
                 */
                Service springObjectService = (Service) springContainer
                                .getBean("service");
                ConstructorInjectionClient springObjectConstructorClient =
                                (ConstructorInjectionClient)
                                springContainer.getBean("constructorClient");
                SetterInjectionClient springObjectSetterClient =
                                (SetterInjectionClient)
                                springContainer.getBean("setterClient");
        }

}
```

```xml
<!-- dependency-injection.xml -->

<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
```

```
	<!--  Objekti se u automatski kreiraju od strane Spring kontejnera
	      pri skeniranju ove datoteke. Svakom Spring zrnu definiše se ime,
	      kao i klasa koja se koristi za instanciranje objekta, nakon toga moguće
	      je bilo gde u Spring kontaineru koristiti definisano ime kao referencu
	      na Spring zrno. -->
	<bean id="service"
		class="milan.panic.master.dependency.injection.ServiceImpl" />

	<!-- Spring umetanje zavisnosti -->
	<!-- Spring podržava dva moguća tipa umetanja zavisnosti: -->
	<!-- 1. Konstruktorski tip kome se definiše parametar constructor-arg. -->
	<bean id="constructorClient"
	class="milan.panic.master.dependency.injection.ConstructorInjectionClient">
		<constructor-arg ref="service" />
	</bean>
	<!-- 2. Seterski tip kome se definiše parametar property. -->
	<bean id="setterClient"
		class="milan.panic.master.dependency.injection.SetterInjectionClient">
		<property name="service" ref="service" />
	</bean>
</beans>
```

```
package milan.panic.master.spring;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import milan.panic.master.dependency.injection.ConstructorInjectionClient;
import milan.panic.master.dependency.injection.Service;
import milan.panic.master.dependency.injection.ServiceImpl;
import milan.panic.master.dependency.injection.SetterInjectionClient;
/**
 * Spring učitava metapodatke definisane unutar klase sa
 * predefinisanim Spring anotacijama i od toga pravi gotov sistem. Ova
 * konfiguracije identična je konfiguraciji definisanoj unutar xml
 * datoteke.
 */
@Configuration
public class SpringConfiguration {
	/**
	 * Opciono, moguće je eksplicitno definisati ime Spring zrna
	 * preko @Bean(name = "service"), u suprotnom ime metoda koji
	 * kreira objekat postaće ime Spring zrna.
	 */
	@Bean
	public Service service() {
		return new ServiceImpl();
	}
	/** Konstruktorski tip umetanja zavisnosti */
	@Bean
	public ConstructorInjectionClient constructorClient() {
		return new ConstructorInjectionClient(service());
	}
	/** Seterski tip umetanja zavisnosti */
	@Bean
	public SetterInjectionClient setterClient() {
		SetterInjectionClient setterClient = new SetterInjectionClient();
		setterClient.setService(service());
		return setterClient;
	}
}
```

Spring obezbeđuje proveru postojanja svih zavisnosti i njihovog tipa prilikom učitavanja kontejnera, tako da će izvršenje programa biti zaustavljeno odmah prilikom učitavanja i neće doći do neželjenih sporednih efekata izvršenja loše definisanog programa. Poseduje i mnoge revolucionarne ideje kada je u pitanju umetanje zavisnosti kao što je uključivanje automatskog skeniranja projekta za klasama odnosno atributima anotiranim specijalnim anotacijama @Component i @Autowired, @Inject itd. koji će biti automatski kreirani od strane kontejnera, bez potrebe za eksplicitnom definicijom spajanja unutar konfiguracije (vise informacija [7]), tako da se konfiguracija potrebna za funkcionisanje aplikacije svodi na minimum.

Aspektno orijentisano programiranje sa zastupnicima

Spring je od početka zamišljen i razvijan sa podrškom za aspektno orijentisano programiranje kao načinom da se obezbedi modularizacija međumodularnih funkcionalnosti, tako da postoji proširenje Spring jezgra (projekti Spring AOP i Spring Aspect) koje daje podršku za pisanje aspekata. Spring ima svoju implementaciju aspektno orijentisanog sistema.

Konceptualna razlika u implementacji u odnosu na AspectJ jeste upotreba projektnog uzorka zastupnik kojom se izbegava potreba za eksplicitnim tkanjem aspekata van programskog jezika Java.

Projektni uzorak *zastupnik*

Projektni uzorak *zastupnik* predstavlja pristup u dizajnu aplikacija koji omogućava zamenu jedne komponente sistema drugom komponentom, takozvanim zastupnikom (engl. proxy), pri čemu se način komunikacije zastupnika sa okolinom ne razlikuje od komunikacije prvobitne komponente [22]. Ovaj projektni uzorak omogućava dodavanje ili menjanje programske logike prvobitne komponente pri čemu komunikacija unutar originalnog sistema ne menja.

U programskom jeziku Java komunikacija između komponenti unifikuje se interfejsima pošto su nezavisni od implementacije, tako da ima smisla arhitekturu sistema praviti tako da se zahteva da interfejsi između komponenata ostaju nepromenjeni, pa je konceptualno najbolje implementirati zastupnike oko interfejsa a ne objekata.

Primer naivne implementacije projektnog uzorka *zastupnik*:

```
package milan.panic.master.proxy;
/**
 * Interfejs koji će imati zastupnika.
 */
public interface Interface {
	public Object method();
}
```

```
package milan.panic.master.proxy;
/** Trivijalna implementacija zastupnika jednog interfejsa. */
public class ManualProxy implements Interface {
	private Interface interfaceInstance;
	public ManualProxy(Interface interfaceInstance) {
		this.interfaceInstance = interfaceInstance;
	}
```

```
        /**
         * Zastupnik zadržava istu komunikaciju kao i originalni objekat.
         */
        @Override
        public Object method() {
                /**
                 * Implementacija metode koja se zastupa je proizvoljna, može
                 * da uključi poziv originalnog metoda objekta koji se
                 * zastupa, a i ne mora, što dovodi do toga da se u domenu
                 * aspektno orijentisanog programiranja metod zastupnika
                 * ponaša kao around savet unutar aspektno orijentisanog
                 * programiranja:
                 */
                /** Dodatna logika pre poziva originalnog metoda */
                /** Opcioni poziv originalnog metoda */
                Object returnValue = interfaceInstance.method();
                /** Dodatna logika posle poziva originalnog metoda */

                /**
                 * Povratna vrednost mora biti istog tipa, komunikacija ostaje
                 * ista.
                 */
                return returnValue;
        }

}
```

Uz pomoć podrške koju Java programski jezik obezbeđuje za refleksiju moguće je uopštiti implementaciju zastupnika tako da zastupanje bude nezavisno od metoda koji se zastupa:

```
package milan.panic.master.proxy;
import java.lang.reflect.Method;
/**
 * Zastupnik će koristiti ovaj interfejs za sve metode koje bude zastupao.
 */
public interface ProxyHandler {
        public Object invoke(Object proxy, Method method,
                        Object[] args) throws Throwable;
}
```

```
package milan.panic.master.proxy;

import java.lang.reflect.Method;

public class ProxyHandlerImpl implements ProxyHandler {
        private Object obj;
        public ProxyHandlerImpl(Object obj) {
                this.obj = obj;
        }

        @Override
        public Object invoke(Object proxy, Method method,
                        Object[] args) throws Throwable {
                System.out.println("AdvancedProxy: Additional logic");
                return method.invoke(obj, args);
        }
}
```

```java
package milan.panic.master.proxy;

import java.lang.reflect.Method;
/**
 * Napredna implementacija zastupnika moze da uz pomoć refleksije
 * uključi univerzalnog rukovaoca metodima koji su definisani u
 * interfejsu koji se zastupa. Ipak, uz pomoć refleksije nije moguće
 * dinamički dodeliti interfejs nekom objektu, tako da definicija
 * interfejsa i eksplicitna implementacija svih metoda zastupanog
 * interfejsa ostaje unutar ovako napravljenog zastupnika.
 */
public class AdvancedProxy implements Interface {

	Interface interfaceInstance;
	ProxyHandler proxyHandler;

	public AdvancedProxy(Interface interfaceInstance,
			ProxyHandler proxyHandler) {
		this.interfaceInstance = interfaceInstance;
		this.proxyHandler = proxyHandler;
	}

	/**
	 * Univerzalniji način zastupanja originalnog metoda. Telo svakog
	 * metoda koji se zastupa u zastupaocu biće identično: metod i
	 * parametri metoda se prosleđuju interfejsu ProxyHandler koji će
	 * se izvršiti umesto njega i ukoliko je potrebno pozvati
	 * originalni metod zastupanog objekta.
	 */
	@Override
	public Object method() {
		try {
			String methodName = new Object() {
			}.getClass().getEnclosingMethod().getName();
			Method method = interfaceInstance.getClass()
					.getMethod(methodName, null);
			return proxyHandler.invoke(interfaceInstance, method,
					null);
		} catch (SecurityException e) {
			// Obrada sigurnosnog izuzetka.
		} catch (Throwable e) {
			// Obrada throwable.
		}
		return null;
	}
}
```

Ipak, usled ograničenja samog Java programskog jezika zbog koga nije moguće u vreme izvršenja programa menjati statičku strukturu programa i dinamički dodeljivati interfejse objektima, koje je pre svega nametnuto zbog sigurnosti aplikacija, nemoguće je otkloniti implementacije metoda koje se zastupaju iz zastupnika i dalje unaprediti ovaj uzorak korišćenjem samog programskog jezika Java.

Široka primena zastupnika, pritisak privrede i Java zajednice kao i moguće posledice uvek neželjenog zaobilaženja standarda kao i manipulacije van samog programskog jezika do kojih dolazi ukoliko nema zvanične podrške nekoj tehnologiji doveli su do razmatranja uvođenja bolje podrške za rad sa zastupnicima u samom

programskom jeziku. To je urodilo plodom i Java 1.3 standard omogućio je mehanizam za univerzalno kreiranje zastupnika nad interfejsima. Uvedena je klasa *Proxy* kao deo standardne Java biblioteke za refleksiju koja kreira zastupnika nad skupom interfejsa pri čemu zastupniku dodeljuje instancu *InvocationHandler* (referenca) interfejsa čiji metod *invoke* presreće metode interfejsa i daje mogućnost dodavanja ili menjanja njihovih funkcionalnosti (slično kao i *AdvancedProxy* primer).

```
package milan.panic.master.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
/**
 * Opšti način za pravljenje zastupnika zasniva se na istom principu
 * kao i prikazan AdvancedProxy mehanizam: Metod invoke interfejsa
 * InvocationHandler spona je u izvršenju metoda zastupnika i
 * originalnog objekta. On kontroliše izvršenje svih metoda zastupnika.
 */
public class JdkInvocationHandlerImpl
             implements InvocationHandler {

      Object proxiedObject;

      public JdkInvocationHandlerImpl(Object object) {
             this.proxiedObject = object;
      }

      @Override
      public Object invoke(Object classLoader, Method method,
                   Object[] methodArgs) throws Throwable {
             System.out.println(
                          "JDK Dynamic Proxy invoke method: Additional logic");
             return method.invoke(proxiedObject, methodArgs);
      }
}
```

```
package milan.panic.master.proxy;
/**
 * Objekat koji implementira interfejs, zastupnik će promeniti
 * ponašanje interfejsne metode koja je ovde implementirana.
 */
public class InterfaceImpl implements Interface {
      @Override
      public Object method() {
             return null;
      }
}
```

```
package milan.panic.master.proxy;
import java.lang.reflect.Proxy;
/**
 * Kreiranje, odnosno korišćenje zastupnika.
 */
public class ProxyProgram {
      public static void main(String[] args) {
             Interface object = new InterfaceImpl();

             Interface manualProxy = new ManualProxy(object);
             manualProxy.method();
```

```
            AdvancedProxy advancedProxy = new AdvancedProxy(object,
                          new ProxyHandlerImpl(object));
            advancedProxy.method();

            /**
             * Univerzalno rešenje podržano od strane Java programskog
             * jezika za pravljenje zastupnika: Zastupnik se pravi nad
             * skupom interfejsa pri čemu se prosleđuje implementacija
             * presretača svih metoda definisanih interfejsa (slično kao i
             * unutar AdvanceProxy implementacije) dok eksplicitna
             * implementacija metoda zastupnika nije potrebna usled
             * implementirane podrške unutar samog programskog
             * jezika Java.
             */
            Interface jdkDynamicProxy = (Interface) Proxy
                          .newProxyInstance(
                                        /** Učitavač klasa */
                                        object.getClass().getClassLoader(),
                                        /** Skup interfejsa */
                                        object.getClass().getInterfaces(),
                                        /** Videti implementaciju klase */
                                        new JdkInvocationHandlerImpl(object));
            jdkDynamicProxy.method();
      }
}
```

Projektni uzorak zastupnik omogućava dodavanje funkcionalnosti metodima bez direktnog odražavanja na zavisnost među komponentama, ali i bez znanja objekta koji je zastupan što se može između ostalog iskoristiti i za rešavanje problema modularizacije međumodularnih funkcionalnosti.

Ukoliko se međumodularna funkcionalnost definiše unutar metode *InvocationHandler* objekta i prilikom kreiranja zastupnika se obuhvate oni delovi sistema za koje treba da vazi ta međumodularna funkcionalnost to će zastupnika učiniti modulom za međumodularnu funkcionalnost. Posmatrano iz ugla definisanog aspektno orijentisanog sistema, može se reći da implementacija opšteg Java zastupnika definisanog klasom *Proxy* ima ulogu tkača u aspektno orijentisanom sistemu (pri čemu je tkanje izvršeno implicitno, unutar definicije same standardne biblioteke za refleksiju), pri čemu omogućava pristup tačkama spajanja *IzvršenjeMetoda,* skup metoda interfejsa nad kojim je definisan čini jednu tačku reza, a definisani *invoke* metod u stvari predstavlja *around* savet.

Direktno korišćenje zastupnika za definisanje međumodularnih funkcionalnosti postoje sledeći konceptualni problemi:

- Svi zastupnici moraju biti eksplicitno kreirani prilikom definisanja međumodularne funkcionalnosti.
- InvocationHandler ne vodi računa o tipovima objekata, što može dovesti do neželjenih efekata ukoliko su pogrešni objekti obuhvaćeni zastupnikom.
- Ne postoji jezik za definisanje tačaka reza niti ijedan drugi jednostavan način da se definiše koji delovi sistema sadrže međumodularnu funkcionalnost. To u mnogome otežava obuhvatanje složenijih tačaka reza, samim tim postoji mogućnost veštačkog kreiranja naizgled nepotrebnih interfejsa u sistemu samo da bi delovi sistema bili obuhvaćeni zastupnikom.

Svi ovi nedostaci direktnog korišćenja zastupnika čine projektni uzorak zastupnik kao takav ipak nepodesnim za korišćenje kao zamenu za aspektno orijentisani sistem pošto je uz pomoć njega implementiranje bilo koje malo složenije međumodularne funkcionalnosti praktično nemoguće.

Imajući u vidu upravo projektni uzorak zastupnik i podršku samog Java programskog jezika za njega, ali i druge teorijski moguće implementacije aspektno orijentisanih sistema, u vreme nastanka Spring okvira postojala je struja mišljenja da pristup kojim se kreće AspectJ u vezi sa aspektno orijentisanim principom u programiranju možda nije najbolja. Razvijaoci Spring okvira smatrali su da bi rešenje trebalo potražiti upravo u zastupnicima. Oni su zajedno sa još nekoliko drugih programera osnovali su takozvanu *AOP alijansu* [23] [24] i napisali minimalnu biblioteku u kojoj se nalaze definicije interfejsa koje bi svaka implementacija aspektno orijentisanog programiranja u Javi trebalo da poseduje.

Ova biblioteka konceptualno je unutar sebe definisala tačku spajanja i savet, dok su nedefinisani ostali tačka reza kao i jezik za definisanje tačaka reza. Biblioteka AOP alijanse, najverovatnije zbog ovog konceptulnog nedostatka, nije doživela široku prihvaćenost i danas je potpuno zanemarena. U Spring implementaciji, oslanjanje na biblioteku AOP alijanse rezultiralo je time da se u prvoj verziji Spring okvira najbolje reši samo problem automatskog kreiranja zastupnika, dok su ostali problemi sa ovom implementacijom ostali nerešeni:

Razvijeni okvir za umetanje zavisnosti odličan je početak za automatizaciju kreiranja bilo kog objekta pa i zastupnika koji može služiti za AOP. Uvedena je klasa *ProxyFactoryBean* koja služi za automatizaciju pravljenja zastupnika preko jezgra Spring okvira.

```
<!-- spring-old-aop.xml -->
<!-- Aspektno orijentisano programiranje u nastanku Spring okvira. -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">

       <!-- Objekat koji se zastupa -->
       <bean id="service" class="milan.panic.master.spring.aop.ServiceImpl" />

       <!-- Savet koji je definisan kao Spring objekat a koji predstavlja
              implementaciju interfejsa MethodInterceptor koji je definisan
              unutar biblioteke AOP alijanse -->
       <bean id="aroundMethodBean"
              class="milan.panic.master.spring.aop.AroundMethod" />

       <!-- Spring zrno koje kreira zastupnika i enkapsulira svu
              ostalu logiku unutar sebe. -->
       <bean id="serviceProxy"
              class="org.springframework.aop.framework.ProxyFactoryBean">
              <!--Ručno definisanje oko kog interfejsa zastupnika treba kreirati.
                     Nije neophodno eksplicitno defnisanje: ukoliko ovaj parametar
                     nije definisan, automatski će se ustanoviti kojim interfejsima
                     klasa pripada. Ako klasa ne implementira interfejse kreira se
```

```
                CGLIB proxy. -->
            <property name="proxyInterfaces"
                value="milan.panic.master.spring.aop.Service" />
            <!--Spring bjekat oko koga se kreira zastupnik -->
            <property name="target" ref="service" />
            <!--Lista saveta koji se proseđuju zastupniku. Redosled u listi
                određuje i redosled izvršenja saveta ukoliko ih ima više. -->
            <property name="interceptorNames">
                <list>
                    <value>aroundMethodBean</value>
                </list>
            </property>
        </bean>

</beans>
```

```
package milan.panic.master.spring.aop;

/**
 * Definicija MethodInterceptor, MethodInvocation interfejsa AOP
 * alijanse u potpunosti je analogna InvocationHandler metodu proksi
 * mehanizma, odnosno Method interfejsu mehanizma refleksije podržanom
 * od strane Java programskog jezika.
 */
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

/**
 * Implementacija ovog interfejsa je u stvari around savet unutar AOP
 * definicija. Ovaj savet je najopštiji i vidi se da u mnogome podseća
 * na invoke metod koji koristi zastupnik. Ponašanje ostalih
 * saveta(before, after, after throwing, after returning) moguće je
 * lako definisati od ovog najopštijeg.
 */
public class AroundMethod implements MethodInterceptor {
        @Override
        public Object invoke(MethodInvocation methodInvocation)
                throws Throwable {
            try {
                Object result = methodInvocation.proceed();
                return result;
            } catch (IllegalArgumentException e) {
                throw e;
            }
        }
}
```

Mehanizam koji se krije iza ove klase [25] enkapsulira poprilično logiku razvijenu za kreiranje zastupnika koji su specijalizovani za probleme aspektno orijentisanog programiranja.

Inicijalna implementacija ovako kreiranih Spring zastupnika odlično je podržavala definisanje većeg broja saveta za jednog zastupnika u vidu niza saveta koji se prosleđuje kao parametar prilikom definicije ovog Spring zrna. Redosled izvršenja saveta u jednom zastupniku je jasno definisan redosledom saveta u nizu. Saveti se implementiraju kao Spring zrna i oni predstavljaju implementacije interfejsa *MethodInterceptor* iz biblioteke AOP alijase. Može se napraviti direktna paralela

između ovog interfejsa i interfejsa *InvocationHandler* koji se prosleđuje prilikom kreiranja zastupnika u Javi.

Spring AOP mehanizam sposoban je za kreiranje zastupnika nad interfejsima, što podržava sam jezik Java, ali i nad klasama, što se ostvaruje eksternom CGLIB bibliotekom [26].CGLIB biblioteka predstavlja biblioteku koja je u stanju da generiše bajt kôd direktno, kao i da manipuliše i promeni postojeći bajt kôd, što daje mogućnost zaobilažena samog Java programskog jezika i daje dodatne mogućnosti koje u samom programskom jeziku nisu podržane.

Opšte je mišljenje da ne treba koristiti CGLIB zastupnike unutar Spring biblioteke, osim ako to nije apsolutno neophodno, zato što oni ograničavaju sam Java programski jezik i usled toga može doći do grešaka u izvršenju bajt kôda na virtuelnoj mašini ukoliko se dođe do konstrukcije u Java programskom jeziku koja nije podržana. Naime, CGLIB bilblioteka zahteva da klasa oko čije instance se kreira zastupnik ne sme biti označena kao *final*, takođe nije moguće zastupati *final* metode, o čemu sam Spring programer mora voditi računa u svojoj aplikaciji ukoliko koristi ovu vrstu zastupnika. U suptrotnom, događa se izuzetak u samom izvršenju virtuelne mašine i ona se gasi!

Inače, preovlađujuće je mišljenje da manipulacija samim bajt kôdom, bez implementiranog i nadasve dobro istestiranog višeg programskog jezika nije poželjna na Java platformi jer se time ugrožava jedno od osnovnih principa sa kojim je platforma nastala, sigurnost.

U ovoj, inicialnoj implementaciji Spring okvira implementaciji ostalo je na samom programeru da obezbedi obuhvatanje odgovarajućih tačaka spajanja prilikom pisanja programa (unutar samog zastupnika), nikakav jezik za obuhvatanje tih tačaka nije obezbeđen, tako da se može reći da je sama poenta uvođenja aspektno orijentisanog sistema u Spring okviru verzija 1.x izostala. Ipak, postavljena je odlična baza za aspektno orijentisano programiranje koja je kasnije unapređena i danas predstavlja potpunu implementaciju aspektno orijentisanog sistema.

Spring je u narednim verzijama uveo koncept tačke reza i implementirao jezik za njihovo definisanje. Jezik za definisanje tačaka reza pozajmljen je iz AspectJ projekta pošto se pokazao prilično intiutivnim. Spring je, zbog potpune kompatibilnosti preuzeo i u svoj projekat uveo komponentu za parsiranje jezika tačaka reza iz projekta AspectJ. Spring implementacija AOP-a, zbog proksi mezanizma kojim je implementirana, nema pristup svim vrstama tačaka spajanja koje je definisao AspectJ već samo tački spajanja *Izvršenje metoda* pa samo implementira podskup jezika koji predstavlja tačke reza sačinjene od nje.

Ispod haube Spring okvira implementacija jezika za definisanje tačaka reza implementirana tako što je stvoren mehanizam za automatsko stvaranje zastupnika oko Spring zrna u zavisnosti od tačaka reza definisanih aspektnom. Sve što programer treba da uradi ukoliko želi da omogući aspektno orijentisano programiranje u svom Spring projektu jeste da u konfiguraciji omogući ovo automatsko kreiranje zastupnika *(@EnableAspectJAutoProxy* u Java konfiguraciji odnosno *<aop:aspectj-autoproxy/>* u xml konfiguraciji), definiše aspekt i ovaj Spring mehanizam će odraditi ostatak.

Spring prelazno rešenje za definisanje tačaka reza:

```
<!-- spring-intermidiate-aop.xml -->
<!-- Spring prelazno AOP rešenje pre konačno prihvaćenog
       AspectJ jezika za definisanje tačaka reza. -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">

       <bean id="service"
              class="milan.panic.master.spring.aop.ServiceImpl" />

       <!-- Spring uvodi sve vrste saveta. -->
       <bean id="aroundMethodBean"
              class="milan.panic.master.spring.aop.AroundMethod" />
       <bean id="beforeMethodBean"
              class="milan.panic.master.spring.aop.BeforeMethod" />
       <bean id="afterMethodBean"
              class="milan.panic.master.spring.aop.AfterMethod" />
       <bean id="afterThrowingMethodBean"
              class="milan.panic.master.spring.aop.ThrowException" />

       <!--  Spring prelazno rešenje za definisanje tačaka reza
              jeste koncept definisanje tačaka reza  u takozvanim nadgledačima,
              konceptu koji je spring razvio. Nadgledač se može dodeliti
              zastupniku objekta i on mu govori koje metode može da presretne.
              Ovo je značajno jer automatizuje zastupanje samo pojedinih
              metoda, ali ovaj jezik ipak nije dovoljno ekspresivan, niti
              efikasan. -->
       <bean id="advisor"
              class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
              <property name="advice">
                     <ref bean="aroundMethodBean" />
              </property>
              <property name="patterns">
                     <list>
                            <value>.*serviceMethod</value>
                     </list>
              </property>
       </bean>

       <!-- Zastupnik -->
       <bean id="serviceProxy"
              class="org.springframework.aop.framework.ProxyFactoryBean">
              <property name="proxyInterfaces"
                     value="milan.panic.master.spring.aop.Service" />
              <property name="target" ref="service" />
              <property name="interceptorNames">
                     <list>
                            <value>aroundMethodBean</value>
                            <value>beforeMethodBean</value>
                            <value>afterMethodBean</value>
                            <value>advisor</value>
                     </list>
              </property>
       </bean>

</beans>
```

Inicijalna implementacija Spring AOP-a nije imala ni koncept aspekta kao modula u kome se definiše međumodularna funkcionalnost. Aspekt je kasnije uveden i definisan kao Spring zrno. Samo definisanje aspekata u Springu omogućeno je na dva načina koja su konceptualno identična.

Prvi način je konceptualno dosledan tradicionalnom Spring pristupu, gde je dodata posebna xml konfiguracija koja se odnosi na AOP programiranje, dok drugi predstavlja @AspectJ sintaksu.

```
/** Xml definicija aspekta. */
package milan.panic.master.spring.aop;

/**Definicija tačke spajanja iz AspectJ projekta. */
import org.aspectj.lang.JoinPoint;

/**
 * Aspekt se definiše kao najobičniji objekat(registruje se kao Spring
 * zrno). On služi samo za definisanje metoda koji će služiti kao
 * saveti, kasnije, u xml konfiguraciji se registruje pravi aspekt,
 * tačka reza i ovaj objekat koristi kao referenca prilikom
 * definisanja saveta. Videti spring-new-aop.xml.
 */
public class XmlAspect {
	public void beforeMethod(JoinPoint joinPoint) {
		System.out.println("Before advice-Xml defined aspect.");
	}
}
```

```
/**Java definicija aspekta.*/
package milan.panic.master.spring.aop;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
/**
 * Aspekt definicija je identična @AspectJ-u
 */
@Aspect
public class JavaAspect {
	@Pointcut("execution(* milan.panic.master..Service*.*(..))")
	public void pointcut() {
	}

	@Before("pointcut()")
	public void beforeMethod(JoinPoint joinPoint) {
		System.out.println("Before advice-Java defined aspect.");
	}
}
```

```
<!-- spring-new-aop.xml -->
<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-4.3.xsd
	http://www.springframework.org/schema/aop
	http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
```

```
    <bean id="service"
          class="milan.panic.master.spring.aop.ServiceImpl" />

    <!--Omogućavanje automatskog stvaranje zastupnika oko Spring
          zrna. Analogno se može definisati Java konfiguracija
          anotacijom @EnableAspectJAutoProxy. -->
    <aop:aspectj-autoproxy />

    <bean id="xmlAspect"
          class="milan.panic.master.spring.aop.XmlAspect" />

    <aop:config>
          <!-- Xml definisanje aspekta - eksplicitno definisanje
                procesa tkanja za logiku koja je definisana u jednom Spring
                zrnu. -->
          <aop:aspect id="aspectName" ref="xmlAspect">
                <!-- Definicija tačke reza - AspectJ jezik -->
                <aop:pointcut id="pointcut"
                      expression="execution(* *..Service*.*(..))" />
                <!-- Definicija saveta: koji se metod iz definicije Spring
                      objekta treba primeniti i na koju tačku reza -->
                <aop:before method="beforeMethod"
                      pointcut-ref="pointcut" />
          </aop:aspect>
    </aop:config>

    <!-- Java definisanje aspekta - identično @AspectJ definicji
          aspekta -->
    <bean id="javaAspect"
          class="milan.panic.master.spring.aop.JavaAspect" />
    <!--Napomena:Ukoliko je definisana konfiguracija context-component-scan
          u xml konfiguraciji odnosno, @ComponentScan u Java konfiguraciji
          koja omogućava automatsko skeniranje projekta u potrazi
          za Spring objektima a klasa u kojoj se definiše Java aspekt
          je anotirana sa @Component nema potrebe za eksplicitnim
          definisanjem Spring zrna, tako da se potpuno automatizuje
          kreiranje aspekta. -->
</beans>
```

Spring aspektno orijentisano programiranje predstavlja prilično moćan alat za rešavanje međumodularnih funkcionalnosti u potpunosti implementiran u Java programskom jeziku (ukoliko se koriste samo JDK zastunici). Ipak, zbog svoje implementacije zasnovane na zastupnicima ono ima svoja ograničenja:

- Zastupnici se definišu nad interfejsima i omogućavaju dodavanje aspektnog ponašanje samo izvršenja *public* metoda, druge tačke spajanja nisu dostupne.
- Spring programiranje automatizuje kreiranje zastupnika samo nad Spring zrnima tako da ukoliko neki objekat konceptualno ne bi morao biti napravljen mehanizmom umetanja zavisnosti Spring okvira, ukoliko se želi aspektno ponašanje oko njega on mora postati Spring zrno.
- Spring AOP neće omogućiti izvršenje aspektnog saveta dva puta ukoliko se iz tela jednog *public* metoda poziva direktno drugi *public* metod istog objekta. Ovo je nuspojava samog zastupanja: u trenutku kada se poziva drugi metod

referenca na *this* objekat jeste pravi objekat a ne zastupnik, tako da se ne može obezbediti dodatno aspektno ponašanje.

Ova ograničenja treba razumeti i imati ih uvek u vidu prilikom pisanja aspekata u Spring okviru jer u suprotnom njihovo nerazumevanje može proizvesti velike probleme.

Alternative aspekto orijentisanom programiranju

Više puta u radu napomenuto je da problem koji aspektno orijentisano programiranje rešava nije nov već da je fundamentalni problem objektno orijentisane paradigme i načina na koji se problemi apstrahuju unutar nje.

Tradicionalno, ovaj problem se ne rešava aspektno orijentisanim programiranjem. Tehnike koje se pored aspektno orijentisanog programiranja mogu iskoristiti za pokušaj modularizacije međumodularnih funkcionalnosti predstavljaju:

Projektni uzorci

Projektni uzorci u programiranju predstavljaju standardne načine za rešavanje čestih problema na koje se nailazi prilikom osmišljavanja dizajna delova ili celih aplikacija. Među opšte prihvaćenim projektnim uzorcima pojedini se mogu iskoristiti i da se uz pomoć njih delimično ili potpuno umetnu međumodularne funkcionalnosti i smanji rasejanje kôda.

- Uzorak posmatrač [27] podrazumeva postojanje dodatnih objekata u sistemu, takozvanih posmatrača, koji su zaduženi za nadgledanje objekata od interesa i osluškivanje promena u njima. Sami objekti zaduženi su za obaveštavanje svojih posmatrača o promenama koje se dešavaju. Ovo se može iskoristiti tako što će u posmatraču biti implementirana aspektna logika. Loša stvar jeste to što će i dalje u kôdu samih objekata biti kôd o obaveštavanju posmatrača.
- Projektni uzorak zastupnik detaljno je objašnjen u ranijim poglavljima pošto predstavlja fundamentalni deo implementacije apsektno orijentisanog programiranja u Spring okviru. Valja napomenuti da se sve češće uočava kao poseban i projektni uzorak presretač, koji je deo napredne implementacije proksi mehanizma ne obuhvata (interfejs u definiciji Java zastupnika *InvocationHandler* predstavlja uzorak presretač), a koji se definiše kao projektni uzorak u kome se implementira objekat koji je u stanju da presretne metode i promeni im ponašanje.

Automatsko generisanje kôda

Tehnika automatskog generisanja kôda podrazumeva da deo kôda koji će biti napisan u konačnom programu bude generisan od strane mašine. Automatsko generisanje kôda funkcioniše tako što se pravi generator kôda koji je u stanju da automatski generiše programski kôd uz navođenje programera.

Tipičan primer automatskog generisanja kôda jeste korišćenje generatora kôda umetnutih u razvojna okruženja. Ovi generatori u stanju su da generišu tipične konstrukcije koje se nalaze u programskom jeziku automatski i ovo višestruko povećava produktivnost programera.

Generisanje kôda može se iskoristiti kao tehnika za pisanje kôda koji će biti umetnut na mnogim mestima automatski i ovo će rešiti problem međumodularne funkcionalnosti.

Zaključak

Alternativni pristupi koji se tradicionalno upotrebljavaju za implementaciju međumodularnih funkcionalnosti podrazumevaju ulaganje popriličnog truda od strane samih programera ukoliko žele da dobijeni kôd unutar sistema koji stvaraju bude dobro moduliran. Mehanizam koji treba da se konstruiše da bi se napravio izolovani modul međumodularne funckionalnosti je ogroman bilo da se odabere pristup zasnovan na kombinaciji projektnih uzoraka ili na generisanju kôda.

Programere najčešće prvenstveno interesuje, usled nedostatka vremena i drugih faktora koji utiču na realizaciju projekta, da sistem koji se programira funkcioniše i ponaša se u skladu sa definisanim zahtevima, ne i da bude lak za izmenu, tako da u većini slučajeva rezultujući sistem ipak ima isprepletene međumodularne funkcionalnosti u vidu repliciranja kôda. Ozbiljniji projekti, sa predefinisanom arhitekturom i unapred poznatim načinom realizacije, podrazumevaju da će sistem biti dobro moduliran, da bi se mogao lako menjati u budućnosti i najčešće se definiše i implementira sopstveni mehanizam unutar projekta koji će to podržati.

Imajući analizu prikazanih implementacija aspektno orijentisanih principa u vidu, jasno je da AspectJ ne predstavlja ništa drugo do mehanizam generisanja kôda (ukoliko se generisanje kôda posmatra na nivou bajt kôda), a Spring AOP mehanizam zasnovan na projektnim uzorcima. Iste funkcionalnosti koje svaki ozbiljniji projekat želi da ostvari pomenute implementacije pružaju uz mogućnost univerzalne (bilo koja međumodularna funkcionalnost može biti definisana) i potpuno automatizovane (automatsko tkanje u projekat) upotrebe. Uz to ova rešenja su jako dobro testirana da rade ono za šta su namenjena, pa samim tim su zasigurno neuporedivo rešenja u odnosu na svaki pokušaj implementacije za potrebe izolovanog projekta (ili par projekata).

Inicijalni radovi na temu aspektno orijentisanog programiranja su u vreme nastajanja bili u žiži programerske javnosti. Podržani prvim implementacija aspektno orijentisanih principa postavili su odličan temelj za dalji razvoj aspekata. Inicijalne implementacije odlično su prihvaćene, mnogi uspešni projekti implementirani su uz pomoć ovih tehnologija, ali poslednjih godina, valja primetiti, smanjio se entuzijazam za daljim razvojem aspekata kao novog nivoa apstrakcije u programiranju.

Pored toga što su najverovatnije postavljena nerealna očekivanja od same programerske javnosti da će aspektno orijentisano programiranje rešiti sve probleme koji u programiranju postoje, razlozi za manjak entuzijazma mogu se potražiti i u sledećem:

Sam trud koji se treba uložiti u savladavanje AspectJ rešenja prilično je visok uprkos tome što je uložen priličan trud u njegovo smanjenje uvođenjem @AspectJ jezika i pravljenjem podrške za pojedina razvojna okruženja.

- Sam koncept aspekta, kao i svaka druga apstrakcija koja smanjuje programski kôd, nosi sa sobom prilično otežanje razumevanja programskog toka programa.

- Rad sa aspektima zahteva potpuno drugačiji način razmišljanja jer nudi drugačiji pristup rešavanju problema (na primer, sa podrškom aspektno orijentisanog programiranja moguće je za potrebe projekta napraviti projektni uzorak *posmatrač* u nekoliko linija kôda, što inače iziskuje mnogo više truda), drugačiji proces testiranja projekta i sve u svemu drugačiji skup sposobnosti koje programer treba da stekne da bi bio produktivan.
- Trenutne implementacije aspektno orijentisanih principa, zahtevaju popriličan oprez u radu sa njima:
  - Lako se može dogoditi da izraz definisan jezikom tačaka reza obuhvati i nešto što nije željeno (ovo je naročito karakteristično za AspectJ projekat, čije tačke reza obuhvataju čitav projekat, pa između ostalog i biblioteke koje nisu pisane u okviru samog projekta - engl. third party libraries)
  - Implementacije saveta omogućavaju pristup tačkama spajanja uz pomoć refleksije (ukoliko programer nije oprezan, neminovan je problem pri radu sa refleksijom).

Imajući navedeno u vidu mnogi programeri koji su i bili zainteresovani za aspektno orijentisano programiranje, zbog postojećih AOP implementacija odlučili su da aspektno orijentisano programiranje donosi više zla nego koristi i napustili ovu ideju.

Spring AOP rešilo je, ili bar ublažilo, neke probleme koje je prva AspectJ implementacija posedovala, ali ne predstavlja univerzalno rešenje već AOP rešenje unutar Spring projekata, što umnogome ograničava njegovu širu primenu.

Implementacije aspektno orijentisanih prinicipa moguće su i te kako i van Java platforme osmišljavanjem kvalitetnog jezika za definisanje tačaka reza i odabirom jednog od dva pristupa implementaciji koja su u radu analizirana. Ipak, postoje i jezici u kojima je teško ostvarivo aspektno orijentisano programiranje, a čak i ako je ostvarivo, AOP konceptualno narušava prirodu programskog jezika, pa u njima aspektno orijentisano programiranje ne predstavlja kvalitetno rešenje međumodularnih funkcionalnosti.

Takvi su, na primer, jezici funkcionalne paradigme. Iako je bilo pokušaja da se uvede aspektno orijentisano programiranje u njih, za razliku od jezika objektno orijentisane paradigme, to nije rezultiralo plodom. Zbog same prirode funkcionalnih jezika, međumodularne funkcionalnosti u skoro svim slučajevima obuhvataju samo jednu funkciju, pa jezik za definisanje tačaka reza nema mnogo smisla. Takođe, funkcionalni jezici su jezici bez stanja i omiljeni prvenstveno zbog toga što je matematička dokazivost ispravnosti programa prilično direktna za programe napisane „čistim” funkcionalnim jezicima (jezicima koji pripadaju samo funkcionalnoj paradigmi), dok aspektno orijentisano programiranje ove fundamentalne principe narušava. [28]

Mnogi programski okviri nude rešenja za implementaciju čestih i najpotrebnijih međumodularnih funkcionalnosti koje se pojavljuju na projektima, pa se usled toga potreba za aspektno orijentisanim programiranjem kao generičkim pristupom implementaciji međumodularnih zavisnosti izbegava. Programski okviri interne

implementacije delimičnih AOP funkcionalnosti zasnivaju najčešće na projektnim uzorcima ili kombinaciji projektnih uzoraka i generisanja kôda.

Jezici sa dinamičkom strukturom i podrškom za metaprogramiranje (mogućnost programa da menja sopstveni programski kod u vreme izvršenja) stavljaju na raspolaganje programerima daleko moćniji mehanizam od aspektno orijentisanog programiranja, kojim se između ostalog mogu implementirati i međumodularne funkcionalnosti. Metaprogramiranje je zbog svoje prirode oduvek smatrano nesigurnim i njegova upotreba strogo kontrolisana i zbog toga u većini programskih jezika ne postoji u svom izvornom obliku. Sa jedne strane na metaprogramiranje bi se moglo gledati kao na alternativu aspektno orijentisanom programiranju, ali usled nedefinisanosti jasne strukture programa u jezicima koji podržavaju metaprogramiranje, težnje da se smanji upotreba metaprogramiranja i definiše jasna struktura programa u ovim jezicima, kao i težnje da se uvede AOP unutar jezika koji ga podržavaju, na metaprogramiranje se pre može gledati kao na osnovni gradivni element aspektno orijentisanog programiranja nego na apstrakciju koja ga zamenjuje.

U današnje vreme, aspektno orijentisano programiranje živi aktivno samo u Spring okviru i eventualnoj minimalnoj upotrebi AspectJ projekta uz laku integraciju koju Spring okvir omogućava ukoliko je potrebniji moćniji aspektni mehanizam.

Ostaje nada da će aspektno orijentisano programiranje kao princip u budućnosti ponovo zaživeti i da neće ostati rezervisan samo za Java platformu, a i da će doživeti bolje i potpunije implementacije, pošto poseduje neverovatne mogućnosti i neuporedivo štedi programersko vreme.

Reference

[1] D. L. Parnas, „On the criteria to be used in decomposing systems into modules,“ *Communications of the ACM,* t. 15, br. 12, pp. 1053-1058 , 12. decembar 1972.

[2] „The Principles of OOD,“ [Internet]. Dostupno na: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda / V. L. Cristina, „Aspect-Oriented Programming,“ u *European Conference on Object-Oriented Programming (ECOOP)*, Jyväskylä, Finland, 1997.

[4] Eclipse Foundation, „AspectJ Documentation,“ [Internet]. Dostupno na: https://eclipse.org/aspectj/docs.php. [Poslednji pristup 2015].

[5] G. J. Kiczales, J. O. Lamping, C. V. Lopes, J. J. Hugunin, E. A. Hilsdale / C. Boyapati, „Aspect-oriented programming“. SAD Patent US6467086 B1, 20. jul 2002.

[6] „AspectJ and AspectWerkz to Join Forces,“ 2006. [Internet]. Dostupno na: http://www.eclipse.org/aspectj/aj5announce.html. [Poslednji pristup 2015].

[7] R. e. a. Johnson, „Spring Framework Reference Documentation,“ 2015. [Internet]. Dostupno na: http://docs.spring.io/spring/docs/current/spring-framework-reference/pdf/spring-framework-reference.pdf. [Poslednji pristup 2016].

[8] „JBoss AOP,“ [Internet]. Dostupno na: https://access.redhat.com/documentation/en-US/JBoss_Enterprise_Application_Platform/5/html/Administration_And_Configuration_Guide/jboss_aop.html. [Poslednji pristup 2015].

[9] „AspectC++ Documentatiion,“ [Internet]. Dostupno na: http://www.aspectc.org/Documentation.php. [Poslednji pristup 2015].

[10] „AspectJ Compiler Reference,“ [Internet]. Dostupno na: http://www.eclipse.org/aspectj/doc/released/devguide/ajc-ref.html. [Poslednji pristup 2016].

[11] S. Denier, „Traits Programming with AspectJ,“ *Revue des sciences et technologies de l'information,* 2005.

[12] „Java Language Specification - Syntax,“ [Internet]. Dostupno na: https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html. [Poslednji pristup 2016].

[13] F. Y. G. B. A. B. Tim Lindholm, „Java Virtual Machine Specification - Java SE 8 Edition,“ 2015. [Internet]. Dostupno na: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf. [Poslednji pristup 2016].

[14] R. Hickey, „The Clojure Programming Language,“ [Internet]. Dostupno na: https://clojure.org/. [Poslednji pristup 2016].

[15] „Scala Language Specification,“ [Internet]. Dostupno na: http://www.scala-lang.org/files/archive/spec/2.11/. [Poslednji pristup 2016].

[16] „Groovy programming language," [Internet]. Dostupno na: http://www.groovy-lang.org/. [Poslednji pristup 2016].

[17] D. Zhdanov, „Weaving with AspectJ," 15. oktobar 2009. [Internet]. Dostupno na: http://denis-zhdanov.blogspot.rs/2009/08/weaving-with-aspectj.html.

[18] „Java Decompiler," [Internet]. Dostupno na: https://github.com/java-decompiler. [Poslednji pristup 2016].

[19] „Spring Framework History: 2002 – Present," [Internet]. Dostupno na: http://springtutorials.com/spring-framework-history/. [Poslednji pristup 2016].

[20] „Inversion of Control," [Internet]. Dostupno na: https://msdn.microsoft.com/en-us/library/ff921087.aspx. [Poslednji pristup 2016].

[21] M. Fowler, „Inversion of Control Containers and the Dependency Injection pattern," 23. Januar 2004. [Internet]. [Poslednji pristup 2016].

[22] J. V. R. J. R. H. Erich Gamma, Design Patterns: Elements of Reusable Object-Oriented Software - Proxy pattern, Addison-Wesley Professional, 1994, pp. 233-245.

[23] R. Pawlak, „The AOP Alliance: Why did we get in?," 12. jul 2003. [Internet]. Dostupno na: http://aopalliance.sourceforge.net/white_paper/white_paper.pdf.

[24] J. B. C. J. B. L. R. Ö. R. P. A. P. T. B. S. Y. C. Beust, „AOP Alliance (Java/J2EE AOP standards)," [Internet]. Dostupno na: http://aopalliance.sourceforge.net/.

[25] „Spring Core Technologies - Spring AOP APIs," [Internet]. Dostupno na: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop-api.html.

[26] „CGLIB library open source code," [Internet]. Dostupno na: https://github.com/cglib/cglib. [Poslednji pristup 2016].

[27] J. V. R. J. R. H. Erich Gamma, Design Patterns: Elements of Reusable Object-Oriented Software - Observer pattern, Addison-Wesley Professional, 1994, pp. 293-304.

[28] B. C. d. S. O. Meng Wang, What Does Aspect-Oriented Programming Mean for Functional Programmers?, Oxford University Computing Laboratory.

Ostala literatura

I. Sommerville, Software engineering, deveto izdanje, Addison-Wesley, 2010.

R. Laddad, AspectJ in Action, drugo izdanje, Manning, 2009.

G. Kiczales, „Aspect Oriented Programming: Radical Research in Modularity," Google TechTalks, 2006.

J. Hunt, „Aspect oriented programming with Java," 26. oktobar 2006. [Internet]. Dostupno na: http://www.theregister.co.uk/2006/10/26/aspects_java_aop/.

Guirec, „Aspect Oriented Programming: learn step by step and roll your own implementation! - CodeProject," 11 septembar 2013. [Internet]. Dostupno na: http://www.codeproject.com/Articles/479302/Aspect-Oriented-Programming-learn-step-by-step-and. [Poslednji pristup 2015].

D. Gotseva / M. Pavlov, „Aspect-oriented programming with AspectJ," *IJCSI International Journal of Computer Science Issues,* t. 9, br. 5, pp. 212-208.

D. Belcham, „Introduction to Aspect Oriented Programming," .NET Conf UY 2014, 2014.

Oracle, „Java SE Specifications," [Internet]. Dostupno na: https://docs.oracle.com/javase/specs/. [Poslednji pristup 2015].

Oracle, „Interceptors Requirements JSR 318," 27. februar 2009. [Internet]. Dostupno na: http://download.oracle.com/otn-pub/jcp/ejb-3.1-pfd-oth-JSpec/ejb_interceptors-3_1-pfd-spec.pdf. [Poslednji pristup 2016].

„Aspect-oriented software development," avgust 2012. [Internet]. Dostupno na: http://isa.unomaha.edu/wp-content/uploads/2012/08/Aspect-oriented-software-development.pdf.

D. M. Vitas, Prevodioci i interpretatori (Uvod u teoriju i metode kompilacije programskih jezika), Beograd: Matematički fakultet, 2006.

M. Volkmann, „Aspect-Oriented Programming (AOP)," 14. avgust 2003. [Internet]. Dostupno na: http://java.ociweb.com/javasig/knowledgebase/2003-08/AOP.pdf.