Uloga aspektno orijentirane paradigme (AOP) u razvoju aplikacija

Kozarac, Matej

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
ELEKTROTEHNIČKI FAKULTET

Preddiplomski studij računarstvo

ULOGA ASPEKTNO ORIJENTIRANE PARADIGME (AOP) U RAZVOJU APLIKACIJA

Završni rad

Matej Kozarac

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P: Obrazac za ocjenu završnog rada na sveučilišnom prijediplomskom studiju

Ocjena završnog rada na sveučilišnom prijediplomskom studiju

Ime i prezime pristupnika:	Matej Kozarac
Studij, smjer:	Sveučilišni prijediplomski studij Računarstvo
Mat. br. pristupnika, god.	R4453, 17.10.2019.
JMBAG:	0036499968
Mentor:	doc. dr. sc. Tomislav Galba
Sumentor:	izv. prof. dr. sc. Alfonzo Baumgartner
Sumentor iz tvrtke:	
Naslov završnog rada:	Uloga aspektno orijentirane paradigme (AOP) u razvoju aplikacija
Znanstvena grana završnog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak završnog rada:	Opisati aspektno orijentiranu paradigmu. Na jednostavnoj aplikaciji (proizvoljan jezik) prikazati upotrebu i opisati prednosti.

Datum prijedloga ocjene završnog rada od strane mentora:	19.09.2024.
Prijedlog ocjene završnog rada od strane mentora:	Izvrstan (5)
Datum potvrde ocjene završnog rada od strane Odbora:	11.10.2024.
Ocjena završnog rada nakon obrane:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije završnog rada čime je pristupnik završio sveučilišni prijediplomski studij:	24.10.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 24.10.2024.

Ime i prezime Pristupnika:	Matej Kozarac
Studij:	Sveučilišni prijediplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	R4453, 17.10.2019.
Turnitin podudaranje [%]:	9

Ovom izjavom izjavljujem da je rad pod nazivom**: Uloga aspektno orijentirane paradigme (AOP) u razvoju aplikacija**

izrađen pod vodstvom mentora doc. dr. sc. Tomislav Galba

i sumentora izv. prof. dr. sc. Alfonzo Baumgartner

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

Sadržaj

1. Uvod

Tema završnog rada odnosi se na istraživanje i proučavanje uloge aspektno orijentirane paradigme (engl. *Aspect-oriented programming* (AOP)) u razvoju aplikacija, osnovnih koncepata, prednosti i nedostataka njezine primjene te praktičnih primjera. AOP predstavlja značajnu programsku paradigmu koja pruža bolji način upravljanja aspektima softverskih rješenja i poslovnih aplikacija te nastoji riješiti izazove koji su vezani uz presijecanje funkcionalnosti (engl. *Cross-cutting concerns*). Izazovi vezani uz presijecanje funkcionalnosti odnose se na aspekte ili probleme koji se uglavnom protežu kroz cijelu aplikaciju ili kroz više njezinih dijelova te nude mogućnost izdvajanja od ostatka aplikacije kako bi se ostvarila modularnost programskog *koda*, omogućilo lakše održavanje i bolja čitljivost.

Rad je podijeljen u šest poglavlja i potpoglavlja. U prvom, uvodnom poglavlju, naglasak je na predstavljanju teme rada, motivaciji za izbor teme i sažetom opisu poglavlja. Drugo poglavlje nastoji pobliže objasniti pojam aspektno orijentiranog programiranja, njegovu ulogu, najvažnije koncepte te prednosti korištenja. U trećem poglavlju nastoje se opisati najpopularniji razvojni okviri za rad s AOP paradigmom. Četvrto poglavlje donosi usporedbu objektno orijentirane paradigme s aspektno orijentiranom paradigmom kako bi se utvrdile najbitnije razlike te identificirale primjene u korištenju. U petom poglavlju predstavlja se praktični primjer u programskom jeziku C# i korištenje PostSharp razvojnog okvira kojim se nastoji prikazati implementacija programske paradigme na pravom primjeru te načine na koje se mogu riješiti stvarni problemi. U posljednjem, šestom poglavlju iznose se najbitniji zaključci teme.

AOP predstavlja relativno novi pristup programiranju koji uvodi nove načine razmišljanja o radu i organizaciji programskog *koda*. Motivacija istraživanja o ovoj temi leži u želji za razumijevanjem potencijala AOP paradigme i njezinom doprinosu u razvoju softverskih sustava.

2. Aspektno orijentirana paradigma

Aspektno orijentirana paradigma je način pisanja programskog *koda* kojim se povećava modularnost odvajanjem engl. *Cross-cutting* problema, odnosno problema povezanih s presijecanjem funkcionalnosti iz poslovne logike glavnog programa [1]. Može se definirati kao odvajanje *koda* na različite module, također poznato kao modularizacija, gdje je aspekt ključna jedinica modularnosti [2]. Aspektno orijentirano programiranje, kao što ime sugerira, koristi aspekte u programiranju. Može se reći da je aspekt "potprogram" koji je povezan s određenim svojstvom programa, a ako se svojstvo mijenja tada se efekt provlači i odražava kroz cijeli program [1].

Aspektno orijentirana paradigma povećava modularnost odvajanjem problema u *kodu* koji se ponavljaju u više navrata na različitim mjestima, takozvani aspekti povezani s presijecanjem funkcionalnosti (engl. *Cross-cutting concerns*) [3]. To su aspekti programa koji utječu na nekoliko modula bez mogućnosti enkapsulacije unutar samih modula. Oni jednostavno ne mogu biti odstranjeni od ostatka sustava i/ili potencijalno mogu biti uzrok dupliciranju *koda* ili povećavanju ovisnosti između sustava [4].

Ovakav način razvoja softvera omogućuje da svo ponašanje i funkcionalnosti softvera koji nisu u središnjoj, poslovnoj logici softvera budu dodane bez zatrpavanja *koda*. AOP uključuje programske metode i alate koji podržavaju modularizaciju problema na razini izvornog *koda*, dok se aspektno orijentirani razvoj softvera odnosi na cijelu inženjersku disciplinu [3].

Razvoj složenih aplikacija zahtijeva visok nivo znanja i temeljito razumijevanje arhitekture cijelog sustava, kao i njegove domene. Programeri trebaju razmišljati na višoj, apstraktnijoj razini kako bi mogli pisati kvalitetan *kod* i maksimalno iskoristiti sve mogućnosti koje pruža objektno orijentirano programiranje.

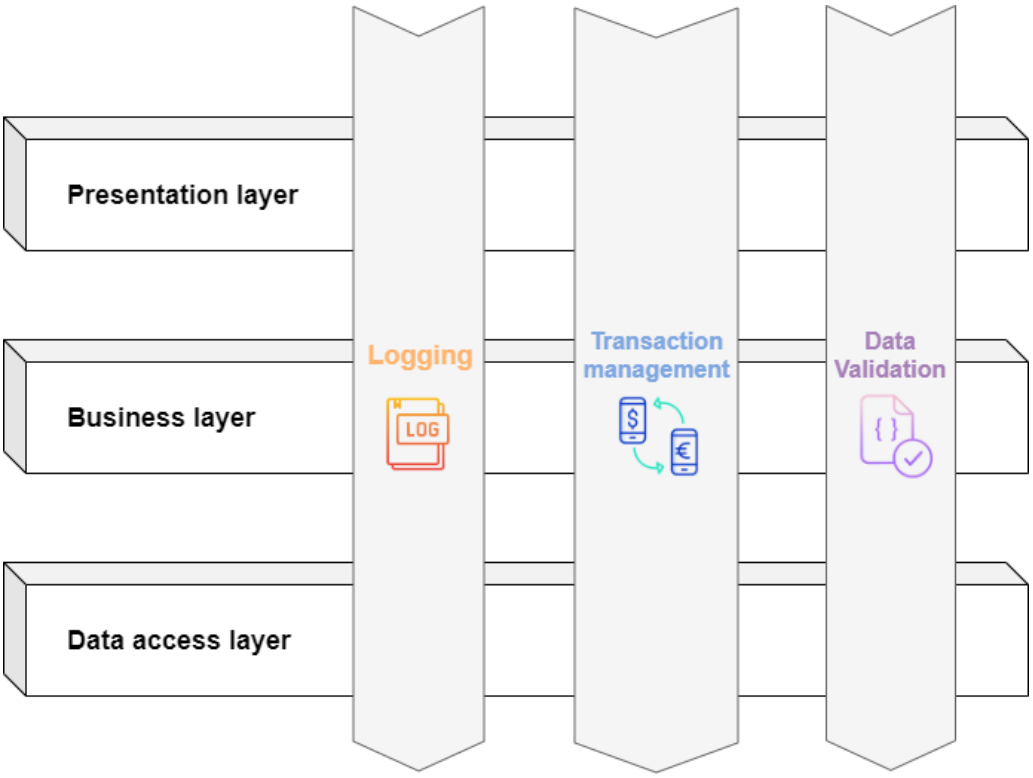
Aspektno orijentirano programiranje podrazumijeva rastavljanje programske logike na različite dijelove (tzv. probleme područja kohezije funkcionalnosti). Gotovo sve paradigme programiranja podržavaju određenu razinu grupiranja i enkapsuliranja problema u zasebne, neovisne entitete pružanjem apstrakcija (npr. funkcija, procedura, modula, klasa, metoda) koje se mogu koristiti za implementaciju, apstrahiranje i sastavljanje tih problema [3].

Neki problemi "presijecaju" višestruke apstrakcije u programu i prkose raznim oblicima implementacije. Logiranje je primjer sveobuhvatnog problema jer strategija logiranja nužno utječe na svaki zabilježeni dio sustava i samim time presijeca sve zabilježene klase i metode [5].

Sve implementacije aspektno orijentirane paradigme imaju neke ekspresije presijecanja koje sažimaju svaki problem na jednom mjestu. Razlika između implementacija leži u snazi, sigurnosti i upotrebljivosti ponuđenih konstrukcija. AspectJ ima niz takvih ekspresija i enkapsulira ih u posebnu klasu, aspekt [6]. Na primjer, aspekt može promijeniti ponašanje osnovnog *koda* (ne-aspektni dio programa) primjenom savjeta (dodatno ponašanje) na različitim točkama spajanja (točkama u programu) navedenim u kvantifikaciji ili upitu koji se zove točka reza (engl. *Pointcut*) koja otkriva podudara li se određena točka spajanja [7]. Aspekt također može izvršiti binarno kompatibilne strukturne promjene u drugim klasama, poput dodavanja članova ili roditelja.

2.1. Izazovi presijecajuće dužnosti

Izazovi presijecajuće dužnosti (engl. *Cross-cutting concerns*) ili kraće CCC, odnose se na određene aspekte softvera koji imaju tendenciju utjecati na više modula ili komponenti sustava, prelazeći preko uobičajenih modularnih granica [5]. Ovi problemi ili izazovi često se nazivaju izazovima presijecajuće dužnosti jer se presijecaju s osnovnom funkcionalnošću sustava i može ih biti izazovno modularizirati korištenjem tradicionalnih tehnika objektno orijentiranog programiranja. Na slici 2.1 može se vidjeti tri glavna sloja aplikacije: prezentacijski sloj (engl. *Presentation layer*), poslovni sloj (engl. *Business layer*) i sloj za pristup podacima (engl. *Data access layer*). Također, na slici 2.1 može se vidjeti kako se logiranje (engl. *Logging*), upravljanje transakcijama (engl. *Transaction management*) i validacija podataka (engl. *Data validation*) primjenjuju u svakom od tri navedena sloja aplikacije, čime može reći da ih presijecaju. Ovo je samo jedan od primjera presijecajućih dužnosti [5].



Slika 2.1 Primjer presijecajuće dužnosti

Kako bismo detaljnije objasnili ovaj koncept, raščlanit ćemo ga na nekoliko ključnih točaka:

- Modularnost u dizajnu softvera : u softverskom inženjerstvu modularnost je temeljni princip. Uključuje podjelu sustava na manje, samostalne module ili komponente, od kojih je svaka odgovorna za određeni aspekt funkcionalnosti sustava. Cilj modularnosti je poboljšati mogućnost održavanja softvera, mogućnost ponovne upotrebe i razumljivost enkapsulacijom povezane funkcionalnosti unutar pojedinačnih modula.
- Problemi presijecajućih dužnosti : aspekti softverskog sustava koji se ne mogu jasno sažeti unutar jednog modula ili komponente. Ovi problemi imaju tendenciju "prijeći" modularne granice, utječući na više modula ili čak na cijeli sustav. Primjeri ovog tipa problema su logiranje, sigurnost, rukovanje pogreškama, praćenje performansi i upravljanje transakcijama.
- Izazovi presijecajućih dužnosti: presijecajuće dužnosti mogu dovesti do zatrpavanja i zapetljanja *koda*. Zatrpavanje *koda* događa se kada se isti *kod* implementira u više modula što otežava održavanje i ažuriranje. Zapetljanje *koda* događa se kada se *kod* koji se odnosi na temeljnu funkcionalnost modula pomiješa s *kodom* koji se odnosi na CCC,

što uvelike otežava praćenje ovisnosti i održavanje urednog *koda*. Suočavanje s presijecajućim dužnostima korištenjem tradicionalnih tehnika objektno orijentiranog programiranja može rezultirati dupliciranim *kodom*, povećanom složenošću i smanjenom modularnošću. Istraživanja u aspektno orijentiranom programiranju fokusirala su se na različite aspekte kao što su dizajn jezika, mehanizmi izrade i alati za aspektno orijentiran razvoj [5].

Studije su pokazale da AOP može dovesti do poboljšanja metrike kvalitete softvera kao što su modularnost, mogućnost održavanja i razumljivost.

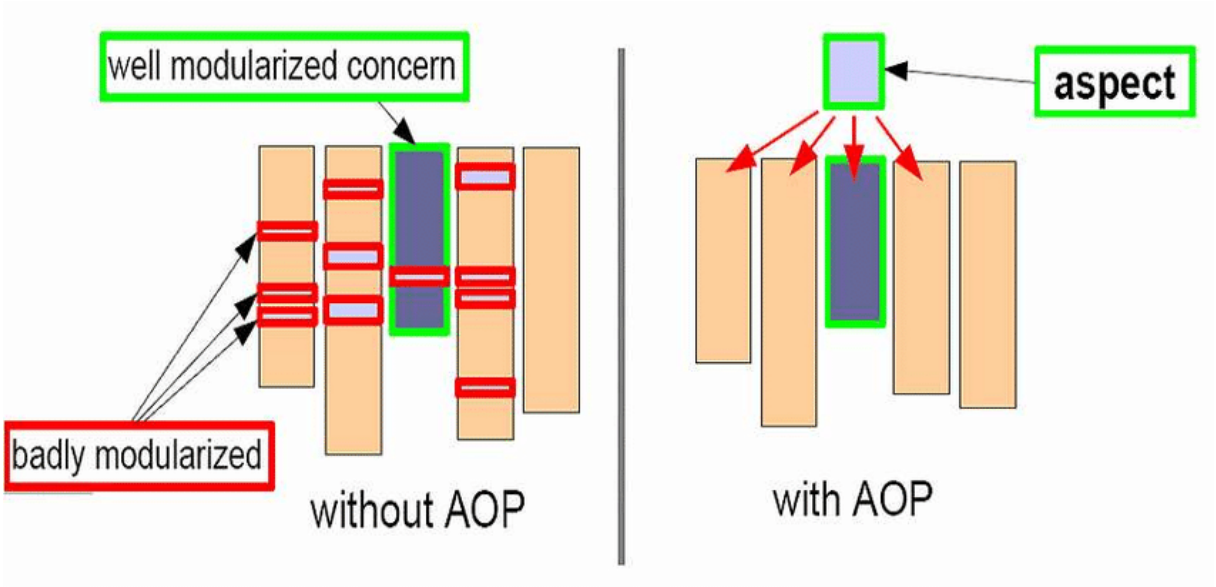
2.2. Prednosti AOP-a

U ovom poglavlju navedene su i opisane najznačajnije prednosti aspektno orijentirane paradigme (AOP):

Modularizacija presijecajućih dužnosti

AOP omogućuje modularizaciju presijecajućih dužnosti, kao što su logiranje, sigurnost, rukovanje pogreškama i upravljanje transakcijama. Odvajanje prijašnje navedenog čini *kod* organiziranijim i lakšim za razumijevanje uklanjanjem nereda presijecajućeg *koda* iz osnovne logike. Programeri se mogu usredotočiti na primarnu funkcionalnost bez da ih ometaju nepovezani problemi [8].

Jedan primjer razlike modularizacije presijecajućih dužnosti kada se ne koristi i kada se koristi aspektno orijentirano programiranje može se vidjeti na slici 2.2. Na lijevoj strani slike 2.2 (bez AOP-a) prikazan je problem s loše modulariziranim dužnostima gdje se presijecajuća dužnost (crveni pravokutnici) mora duplicirati i ručno implementirati u različitim dijelovima sustava. To dovodi do loše modularnosti i složenijeg održavanja *koda*. S desne strane slike 2.2 prikazana je bolja modularnost uz pomoć AOP-a gdje se presijecajuća dužnost (aspekt) centralizira i može se primijeniti na više dijelova sustava iz jednog mjesta. Ovaj pristup omogućuje bolju organizaciju *koda*, povećanu modularnost i lakše održavanje softverskih rješenja.



Slika 2.2 Odvajanje dužnosti u AOP-u

Poboljšana mogućnost ponovne upotrebe koda i provedbe pravila

Aspekti u AOP-u mogu se ponovno koristiti u različitim dijelovima aplikacije. Ova mogućnost ponovne upotrebe smanjuje dupliciranje *koda*, povećava dosljednost i pojednostavljuje održavanje. Mogućnost ponovne upotrebe *koda* omogućava dosljednu provedbu politika i pravila u cijeloj aplikaciji (primjer: provedba sigurnosne politike ili zahtjeva za logiranje bez raspršivanja *koda*) [9].

Poboljšana mogućnost održavanja te smanjena pretrpanost koda

Izoliranjem problema presijecajućih dužnosti AOP poboljšava mogućnost održavanja *koda*. Promjene ili ažuriranja ovih pitanja mogu se napraviti na jednom središnjem mjestu, smanjujući rizik od unošenja grešaka i pojednostavljujući testiranje. Također, bez AOP-a presijecajuće dužnosti često dovode do prenapuhanosti *koda*, gdje se osnovna funkcionalnost miješa s *kodom* specifičnim za problem. AOP pomaže u uklanjanju nereda iz *koda*, a *kod* čini sažetijim i fokusiranijim [10].

Odvajanje odgovornosti

AOP promiče načelo odvajanja odgovornosti dopuštajući programerima da jasno odvoje različite aspekte sustava. Ovo odvajanje čini *kod* organiziranijim i lakšim za održavanje [10].

Skalabilnost i kompatibilnost

Kako sustav raste, dodavanje novih presijecajućih dužnosti ili modificiranje postojećih može se učiniti lakšim i bez opsežnih promjena *koda* u osnovnoj funkcionalnosti. AOP principi mogu se primijeniti na različite programske jezike i platforme, čineći ga prilagodljivim širokom rasponu scenarija razvoja softvera [10].

Testiranje i otklanjanje pogrešaka

Odvajanjem problema presijecajućih dužnosti AOP pojednostavljuje testiranje i otklanjanje pogrešaka. Programeri se mogu usredotočiti na testiranje temeljne logike bez brige o složenostima koje donosi *kod* koji sadrži presijecajuće dužnosti [10].

Suradnja i prilagodljivost

Različiti timovi mogu neovisno raditi na aspektima, smanjujući sukobe i napore paralelnog razvoja. Mogu se dodavati ili mijenjati aspekti bez opsežnog mijenjanja postojećeg *koda*, što je osobito korisno u razvoju softvera [10].

2.3. Koncepti i terminologija AOP-a

Osnovni koncepti AOP-a mogu se vidjeti na slici 2.3, a u daljnjem tekstu biti će navedeni i detaljnije opisani.

Aspekt (engl. *Aspect*) - modul ili jedinica *koda* koja sažima presijecajuće dužnosti, kao što su logiranje, sigurnost ili rukovanje pogreškama. Aspekti su odvojeni od glavne logike aplikacije.

Točka spajanja (engl. *Join point*) - točka tijekom izvođenja programa, kao što je izvođenje metode ili rukovanje iznimkom [2]. Simbolizira točku unutar aplikacije gdje se može

jednostavno uključiti aspekt u samu aplikaciju [3]. To može biti metoda koja se poziva, iznimka ili modificiranje polja [1].

Savjet (engl. *Advice*) - radnja koju poduzima aspekt na određenoj točki spajanja. U osnovi, to su metode koje se izvršavaju kada određena točka spajanja naiđe na odgovarajuću točku reza u aplikaciji. Postoji 5 vrsta savjeta, a to su :

- Prije savjeta (engl. *Before Advice*)- pokreće savjet prije izvršenja metode i također se izvršava prije točke spajanja.
- Nakon vraćanja savjeta (engl. *After Returning Advice*) - pokreće savjet nakon izvršenja metode samo ako se metoda uspješno završi i izvrši nakon što se točka spajanja normalno završi.
- Nakon izbacivanja savjeta (engl. *After Throwing Advice*) - izvršava savjet nakon izvođenja metode samo ako metoda rezultira izbacivanjem iznimke.
- Nakon savjeta (engl. *After (finally) Advice*) - izvršava se nakon točke spajanja bez obzira na izlaz iz točke spajanja, bilo normalno ili pri iznimnom povratku. To znači da će pokrenuti savjet nakon izvršenja metode bez obzira na njezin ishod.
- Okolo savjeta (engl. *Around Advice*) - izvršava se prije i nakon pozivanja preporučene metode. To je najmoćnija vrsta savjeta, a okolo savjeta može se izvesti prilagođeno ponašanje prije i nakon pozivanja metode. Također je odgovoran za odabir hoće li nastaviti do točke spajanja ili prečacem za izvršenje preporučene metode vraćanjem vlastite povratne vrijednosti ili bacanjem iznimke [11].

Točka reza (engl. *Pointcut*) - izraz koji se uspoređuje s točkama spajanja kako bi se odredilo treba li savjet izvršiti ili ne, odnosno skup jedne ili više točaka spajanja na koje se savjet može primijeniti. Točke reza definiraju kriterije za odabir točaka spajanja na temelju potpisa metoda, klasa, naziva paketa i drugih čimbenika [11].

Ciljani objekt (engl. *Target object*) - objekt na koji se primjenjuje jedan ili više aspekata. Također se naziva i preporučeni objekt. U puno slučajevao to je objekt na kojem se poziva metoda ili se izvodi operacija [11].

Tkanje (engl. *Weaving*) - proces integracije ili primjene aspekta u *kod*. Proces koji je odgovoran za povezivanje aspekta s drugim vrstama aplikacija ili objektima za stvaranje preporučenog objekta. Služi u svrhu povezivanja aspekta s drugim vrstama aplikacija ili objektima kako bi se stvorio preporučeni objekt. Može se dogoditi:

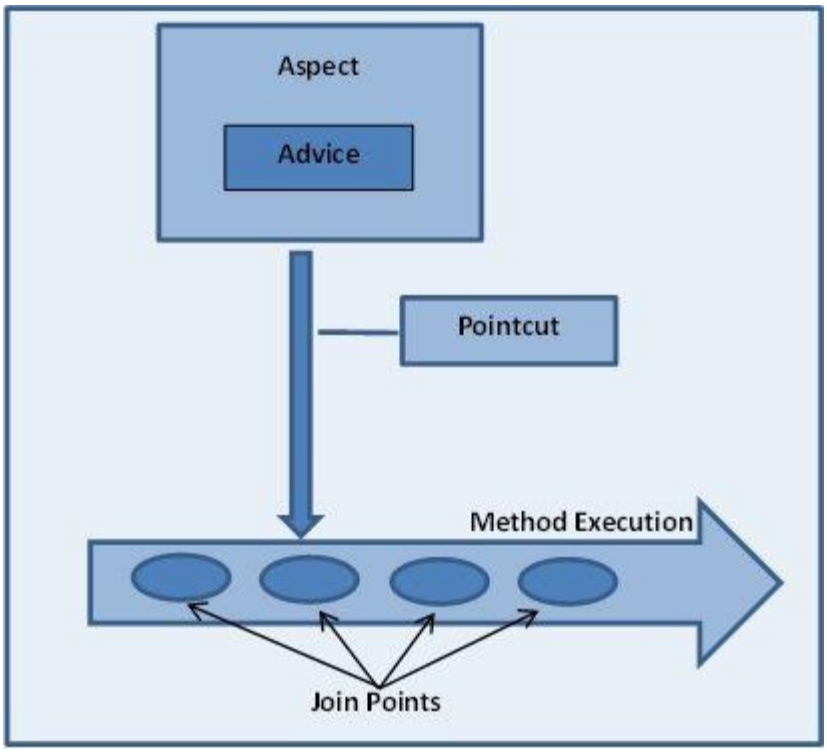
- tijekom kompajliranja: aspekti su utkani u *kod* tijekom kompajliranja

- tijekom učitavanja: aspekti su utkani u *kod* kada se učitavaju klase
- tijekom izvođenja: aspekti su utkani u *kod* tijekom izvođenja.

Zastupnik (engl. *Proxy*) - objekt kreiran nakon primjenjivanja savjeta-a na ciljani objekt, kako bi se implementirali aspekti ugovora [11].

Slika 2.3 Osnovni koncepti AOP-a

Na slici 2.4 prikazano je kako koncepti AOP arhitekture djeluju tijekom životnog ciklusa aplikacije u Spring okviru. Na slici 2.4 može se vidjeti izvršavanje metode koja sadrži različite točke spajanja. Točke spajanja predstavljaju specifične trenutke u tijeku izvršavanja metode na koje se može primijeniti dodatna logika. Aspekt je modul koji sadrži tu dodatnu logiku, dok je savjet konkretna instrukcija koja se izvršava u određenim trenucima metode. Točka reza definira na koje točke spajanja u tijeku izvršavanja savjet treba biti primijenjen. Sve ovo omogućava dodavanje funkcionalnosti bez promjene osnovne logike programa.



Slika 2.4 Primjer arhitekture aspektno orijentirane aplikacije u Spring okviru

3. Dominantni okviri u AOP-u

AOP uključuje metode programiranja i okvire na kojima je podržana i implementirana modularizacija *koda*. U tekstu koji slijedi navedeni su neki od najpoznatijih okvira u AOP-u.

3.1 AspectJ

AspectJ je proširenje za programiranje u Java programskom jeziku stvoreno u istraživačkom centru PARC (*Palo Alto Research Center*). Koristi sintaksu sličnu Java programskom jeziku i uključuje integracije razvojnog okruženja za prikaz strukture presijecajućih dužnosti [4]. Ima vlastiti kompajler i tkalac (engl. *Weaver*), što omogućuje korištenje svih značajki aspektno orijentiranog programiranja koje AspectJ nudi u aplikacijama napisanim u Java programskom jeziku. Postao je široko korišten okvir za AOP, naglašavajući jednostavnost i upotrebljivost za krajnje korisnike. AspectJ uključuje integracije razvojnog okruženja za prikaz strukture koja se prožima od svog prvog javnog izdanja 2001. godine [4].

AspectJ se može implementirati na puno načina, uključujući izvorno tkanje (engl. *Weaving*) ili tkanje *bajt-koda* (engl. *Byte-code weaving*) te izravno u virtualnom stroju (engl. *Virtual machine*). U svim slučajevima AspectJ program postaje pravi Java program koji se izvodi u Java virtualnom stroju. Klase na koje utječu aspekti su binarno kompatibilne s nepromijenjenim klasama (kako bi ostale kompatibilne s klasama koje su kompajlirane s nepromijenjenim izvornicima (engl. *Unmodified source*)). Podrška za više implementacija omogućuje jeziku da raste kako se mijenja tehnologija, a kompatibilnost s Javom osigurava dostupnost platforme [12].

Okvir AspectJ omogućuje čistu modularizaciju presijecajućih dužnosti, kao što su rukovanje pogreškama, sinkronizacijom, ponašanjem ovisno o kontekstu, optimizacijom performansi, praćenjem, logiranjem i protokolima s više objekata [4].

Ključ uspjeha okvira su inženjerske i jezične odluke koje jezik čine upotrebljivim, a aplikacije napisane u jeziku lakšima za isporučiti. Izvorna implementacija AspectJ u Xeroxu koristila je tkanje izvora (engl. *Source weaving*), što je zahtijevalo pristup izvornom *kodu* [12]. Kada je Xerox doprinio *kodu* za Eclipse, AspectJ je ponovno implementiran korištenjem Eclipse Java kompajlera i alata za *bajt-kod* tkanje temeljenog na BCEL-u (engl. *Byte Code Engineering Library*), kako bi programeri mogli pisati aspekte *koda* u binarnom (.class) obliku [12]. U to vrijeme jezik AspectJ bio je ograničen na podršku model po klasi (engl. *Per-class model*) koji je neophodan za inkrementalno kompajliranje i tkanje tijekom učitavanja. To je učinilo IDE integracije jednako responzivnima, a programerima je omogućilo implementaciju aspekata bez mijenjanja procesa izgradnje [12]. To je dovelo do povećanja usvajanja okvira, čime su ga programeri počeli više koristiti, jer je AspectJ postao upotrebljiv za nestrpljive programere i implementacije na razini poduzeća. Od tada je Eclipse time povećao performanse i ispravnost, nadogradio jezik AspectJ da podrži jezične značajke Jave 5, kao što su tipovi (engl. *Generics*) i komentari (engl. *Annotations*) [12].

3.2. Spring

Aspektno orijentirano programiranje jedan je od ključnih elemenata Spring okvira, a Spring AOP dizajniran je za rad s problemima presijecajućih dužnosti unutar tog okvira [13]. Springov vlastiti AOP okvir modularizira presijecajuće dužnosti, a prednost ovog okvira je pružanje osnovnih AOP značajki bez prekomjerne složenosti u dizajnu, implementaciji ili konfiguraciji [13].

Spring AOP okvir u potpunosti iskorištava prednosti Spring spremnika (engl. *Container*). Okvir se konfigurira za vrijeme izvođenja, što uklanja potrebu za korakom kompajliranja ili tkanjem tijekom učitavanja [13]. U usporedbi s okvirom AspectJ, Spring AOP je manje moćan, ali i manje kompliciran. Verzija Spring 1.2 uključuje podršku za konfiguriranje aspekata AspectJ u spremniku. Spring 2.0 verzija dodala je više integracije s AspectJ okvirom. Na primjer, jezik točke reza se ponovno koristi i može se kombinirati s aspektima temeljenim na Spring AOP-u. Nadalje, dodan je i Spring Aspects koji koristi AspectJ da ponudi uobičajene značajke Springa kao što su deklarativno upravljanje transakcijama i injektiranje ovisnosti (engl. *Dependency injection*) putem AspectJ-a u vrijeme kompajliranja ili tkanja u vrijeme učitavanja [13]. Spring okvir interno koristi Spring AOP za upravljanje transakcijama, sigurnosti, daljinskim pristupom i JMX-om [13].

3.3 Usporedba Spring AOP-a, AspectJ-a i PostSharp-a

Spring AOP, AspectJ i PostSharp okviri za aspektno orijentirano programiranje imaju svoje prednosti i mane te imaju neke ključne razlike u pogledu implementacije, integracije i fleksibilnosti. Izbor između Spring AOP-a,AspectJ AOP-a i PostSharp-a ovisi o specifičnim zahtjevima i ograničenjima projekta. Spring AOP je lakši i pogodniji za jednostavnije slučajeve upotrebe, dok AspectJ AOP pruža naprednije značajke i prednosti u performansama, ali dolazi sa strmijom krivuljom učenja i dodatnim postavkama. PostSharp AOP, s druge strane, nudi visoku razinu izražajnosti i fleksibilnosti za .NET aplikacije uz jednostavno postavljanje, ali zahtijeva posebne komponente za .NET razvoj. U daljnjem tekstu će biti opisane razlike između Spring AOP-a,AspectJ AOP-a i PostSharp-a [14].

U načinu implementacije navedeni okviri se razliku na sljedeći način:

- Spring AOP koristi AOP temeljen na zastupnicima (engl. *proxy*), pri čemu koristi dinamičke zastupnike ili CGLIB za stvaranje zastupnika za određene objekte.
- AspectJ AOP implementira tkanje tijekom kompajliranja ili tkanje tijekom učitavanja, integrirajući aspekte izravno u *bajt-kod*.
- PostSharp AOP koristi tkanje tijekom kompajliranja, gdje se aspekti ugrađuju u .NET *bajt-kod* tijekom izgradnje aplikacije.

U načinu integracije navedeni okviri se razlikuju na sljedeći način:

- Spring AOP je usko integriran sa Spring okvirom, što omogućuje besprijekornu integraciju sa Spring beanovima i komponentama.
- AspectJ AOP može se koristiti neovisno o Springu ili integrirati s njim, a nudi više kontrole nad integracijom.
- PostSharp AOP integrira se s .NET okvirom i .NET Core-om omogućujući prilagodbu za .NET aplikacije.

S gledišta performansi navedeni okviri se razlikuju na sljedeći način:

- Spring AOP može biti nešto sporiji zbog korištenja zastupnika za vrijeme izvođenja ili stvaranja CGLIB zastupnika, pa je prikladan za osnovne AOP potrebe.
- AspectJ AOP nudi bolje performanse zbog tkanja tijekom prevođenja ili učitavanja, što ga čini pogodnim za aplikacije osjetljive na performanse.
- PostSharp AOP također pruža visoku brzinu, budući da se aspekti integriraju u vrijeme kompajliranja za .NET aplikacije.

U količini i vrsti podržanih točaka spajanja navedeni okviri se razlikuju na sljedeći način:

- Spring AOP podržava ograničeni skup točaka spajanja, uključujući izvršenje metode, poziv metode i pristup polju, ali samo unutar Spring bean-a.
- AspectJ AOP podržava širi raspon točaka spajanja, uključujući izvođenje metoda, izvođenje konstruktora, pristup polju i druge.
- PostSharp AOP podržava razne točke spajanja unutar .NET-a, uključujući pozive metoda, izvršenje metoda i pristup svojstvima.

Kada se uspoređuju mogućnosti navedeni okviri se razlikuju na sljedeći način:

- Spring AOP nudi ograničenu fleksibilnost u izražavanju, jer se oslanja na komentare ili XML konfiguracije te je jednostavniji za korištenje u osnovnim scenarijima.
- AspectJ AOP pruža veću izražajnost i fleksibilnost, omogućujući detaljnije definiranje aspekata.
- PostSharp AOP također omogućuje visoku razinu izražajnosti i fleksibilnosti u .NET aplikacijama.

Kada se uspoređuju načini konfiguracije navedeni okviri se razlikuju na sljedeći način:

- Spring AOP konfigurira se koristeći komentare, XML ili konfiguraciju temeljenu na Javi unutar Spring ApplicationContext-a.
- AspectJ AOP može se konfigurirati pomoću komentara, XML-a ili zasebne AspectJ konfiguracijske datoteke, pružajući napredne opcije konfiguracije.
- PostSharp AOP konfigurira se korištenjem atributa, prilagođenih atributa ili XML-a unutar .NET aplikacija.

Kada se uspoređuju načini dodavanja raznih ovisnosti navedeni okviri se razlikuju na sljedeći način:

- Spring AOP koristi Springove mehanizme za umetanje ovisnosti u svoje aspekte.
- AspectJ AOP upravlja ovisnostima unutar aspekata putem Java konstruktora ili metoda.
- PostSharp AOP omogućuje umetanje ovisnosti u aspekte koristeći razne tehnike koje nudi PostSharp.

Kada se uspoređuje u koje vrijeme tkanja navedeni okviri dodaju aspekte oni se razlikuju na sljedeći način:

- Spring AOP provodi tkanje za vrijeme izvođenja, što znači da se aspekti dinamički primjenjuju prilikom poziva metode.
- AspectJ AOP omogućuje tkanje tijekom kompajliranja ili učitavanja, što znači da su aspekti integrirani u *bajt-kod* prije izvršenja.
- PostSharp AOP primjenjuje tkanje tijekom kompajliranja, pri čemu se aspekti ugrađuju u .NET assembly-je tijekom izgradnje.

Kada se uspoređuje kompleksnost postavljanja u programski *kod* navedeni okviri se razlikuju na sljedeći način:

- Spring AOP je jednostavniji za postavljanje i korištenje u jednostavnim scenarijima unutar Spring aplikacija.
- AspectJ AOP zahtijeva dodatno postavljanje i potencijalno zaseban korak kompajliranja za definirane aspekte.
- PostSharp AOP je općenito jednostavan za postavljanje u .NET aplikacijama, osobito ako programer već ima iskustva s .NET razvojem.

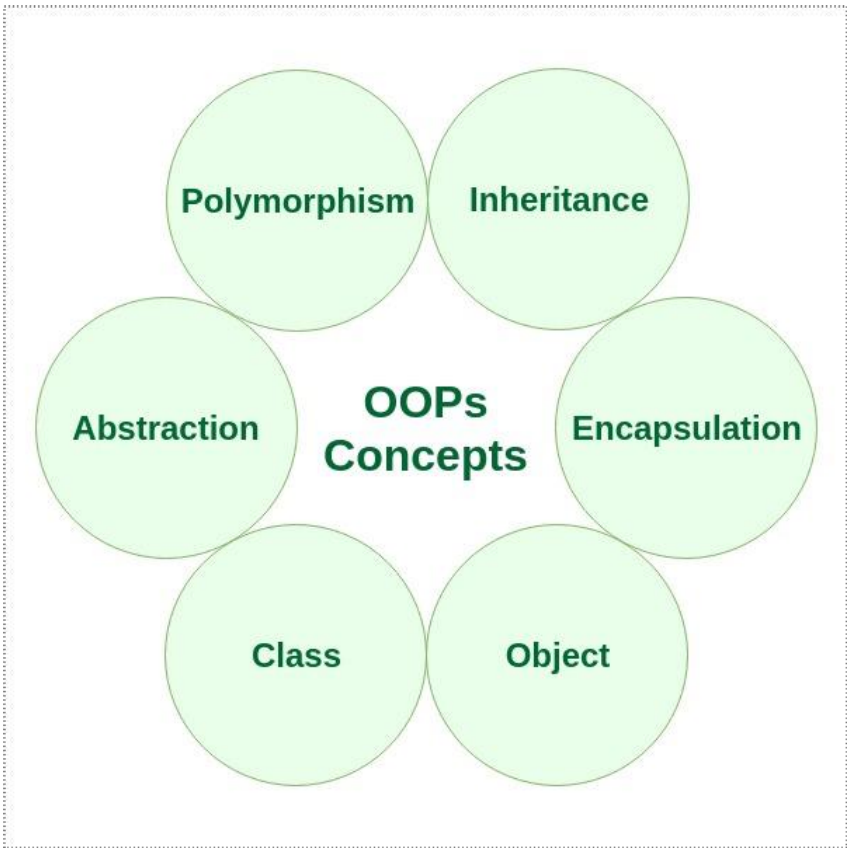
Kada se uspoređuje u kojem slučaju je pojedini okvir najbolje koristiti oni se razlikuju na sljedeći način:

- Spring AOP je široko prihvaćen u aplikacijama temeljenim na Springu te je idealan za aplikacije s umjerenim AOP potrebama.
- AspectJ AOP je popularan u scenarijima gdje su napredne AOP značajke i visoka izvedba ključni, ali može zahtijevati veću stručnost.
- PostSharp AOP je popularan u .NET ekosustavu, posebno za aplikacije na razini poduzeća, pružajući AOP mogućnosti za C# i VB.NET.

4. Usporedba AOP i OOP paradigmi

U ovom poglavlju naglasak je na usporedbi objektno orijentiranog programiranja (OOP) kao programske paradigme temeljene na konceptima objekata i klasa te aspektno orijentiranog programiranja (AOP) koje predstavlja programsku paradigmu usmjerenu na razdvajanje programskog *koda* i povećanje modularnosti.

Objektno orijentirano programiranje predstavilo je novi pristup u programiranju koji rješava pitanja iskoristivosti dijelova *koda*, održavanja te zaštite objekata primjenom enkapsulacije [18]. Takav pristup programiranju omogućio je lakši način rješavanja stvarnih problema izjednačujući entitete s objektima kao srži OOP paradigme [19]. Važni koncepti OOP paradigme vežu se uz pojmove objekata, klasa, enkapsulacije, apstrakcije, polimorfizma i nasljeđivanja, kao što je prikazano na slici 4.1.



Slika 4.1 Koncepti OOP paradigme

Uz brojne prednosti objektno orijentiranog programiranja, važno je navesti i nedostatke ovog pristupa koji se vežu uz narušavanje performansi, budući da se programski *kod* razlaže na puno dijelova, probleme s nasljeđivanjem klasa, otežan način rada s paralelizacijom te mogućeg prekomjernog iskorištavanja memorije zbog velikog broja objekata. Nadalje, predstavlja koncept koji nije primjenjiv za sve probleme [18].

Aspektno orijentirano programiranje predstavlja programsku paradigmu koja nastoji razlomiti program u područja funkcionalnosti koje se nazivaju izazovi (engl. *Concerns*). Cilj takvog pristupa je povećati modularnost i ponovnu iskoristivost dijelova programskog *koda* [23]. Na sličan način, OOP primjenjuje klase kako bi se grupirale slične aktivnosti u okviru objekata. Međutim, postoje funkcionalnosti poput bilježenja izvršenih događaja (eng. *Logging*), rada s bazama podataka, provođenja sigurnosnih mjera te praćenja performansi koje najčešće zahvaćaju cijelu aplikaciju [22]. Primjenom AOP paradigme, takve funkcionalnosti bi se implementirale tako da se odvoje u zasebne izazove na jednoj lokaciji bez dupliciranja programskog *koda* [21], [24].

Usporedba navedenih paradigmi prikazana je u tablici 4.1.

Tablica 4.1 Usporedba OOP i AOP paradigmi

	Objektno orijentirano programiranje	**Aspektno orijentirano programiranje**
Definicija	OOP je programska paradigma koja nalaže uporabu objekata za predstavljanje entiteta, njihovih obilježja (atributa) i ponašanja (metoda).	AOP je programska paradigma koja nastoji modulirati izazove presijecajućih odgovornosti na način da se odvoje od osnovne programske logike.
Osnovna obilježja	Objekti sa atributima i metodama kao instance klasa.	Aspekti koji predstavljaju izazove presijecajućih odgovornosti.
Organizacija programskog *koda*	Unutar klasa i objekata.	Modularizacija izazova.
Duplikacija programskog *koda*	Postoji mogućnost duplikacije programskog *koda* ukoliko dođe do preklapanja funkcionalnosti pojedinih klasa.	AOP smanjuje mogućnost duplikacije programskog *koda* zahvaljujući uporabi aspekata kroz cijelu aplikaciju te njihovom ponovnom korištenju.
Mogućnosti izmjene	Postoji mogućnost izmjena većeg broja klasa ukoliko to promjene zahtijevaju te ukoliko veći broj klasa sudjeluje u ostvarenju pojedinih funkcionalnosti.	Izmjene i održavanje programskog *koda* su lakši procesi unutar AOP paradigme zahvaljujući bilježenju promjena na jednom mjestu.

Čitljivost programskog *koda*	Programski *kod* može postati manje čitljiv kada se jasno ne razdvoji programska logika od pojedinih funkcionalnosti.	Poboljšana čitljivost programskog *koda* postignuta je izoliranjem izazova presijecajućih odgovornosti unutar zasebnih aspekata.
Primjeri	Objektno-orijentirani programski jezici poput: Python, Java, C#, C++	Razvojni okviri poput: Spring AOP (Java), PostSharp (.NET), AspectJ
Primjene	Prilikom rješavanja problema unutar kojih se poslovni entiteti mogu prikazati objektima koji enkapsuliraju njihova obilježja i ponašanja.	Prilikom rada s izazovima presijecajućih odgovornosti koji se primjenjuju kroz cijelu aplikaciju i zahtijevaju odvajanje od poslovne logike, lakše održavanje i čitljivost te jedinstveno mjesto izmjena.

Cilj ovog poglavlja bio je objasniti dvije važne programske paradigme, njihove uloge, prednosti i nedostatke. Važno je naglasiti kako AOP paradigma ne predstavlja konkurenciju OOP paradigmi, već je nastala na temelju njezinih koncepata, proširuje njezine mogućnosti i rješava probleme. Poput OOP paradigme i AOP paradigma je podložna nedostacima poput kompleksnosti implementacije, otežanog testiranja i debugiranja te potencijalnim problemima kompatibilnosti i integracije s postojećim bibliotekama i servisima [25]. Zahvaljujući novom principu rada - izdvajanju izazova presijecajućih odgovornosti od poslovne logike, nastoji omogućiti lakše održavanje, proširenje funkcionalnosti, bolju čitljivost te jasno mjesto izmjena bez dupliciranja programskog *koda* [22].

5. Praktični primjer

Za potrebe praktičnog primjera uloge aspektno orijentirane paradigme korišteno je razvojno okruženje dugoročne podrške (engl. *Long term IDE*) Visual Studio. Sav programski *kod* prikazan na slikama u ovom poglavlju moguće je naći u prilogu P1 koji sadrži kompletan projekt *InventoryManagementSystem*. Navedeno razvojno okruženje odabrano je zbog puno prednosti. Neke od važnijih prednosti su: velika podrška za različite jezike kao što su C#, C++, JavaScript, Python i drugi, zatim okruženje je izdano i podržano od tvrtke Microsoft koja ima dobru reputaciju. Visual Studio pruža sve što vam je potrebno za razvoj aplikacije unutar jednog okruženja, što uključuje uređivač *koda*, alate za debugiranje, izgradnju i razna druga pomagala i što je također važno, besplatno je što ga čini dobrim izborom za ovaj završni rad. Uz Visual Studio odabran je C# programski jezik koji uz redovna ažuriranja i podršku zajednice programera dobiva i redovna ažuriranja i unapređenja koja distribuira tvrtka Microsoft. Za potrebe završnog rada sa službene stranice Microsoft tvrtke skinut je Visual Studio verzije Community 2022 [26].

Kako bi se najbolje prikazala uloga aspektno orijentirane paradigme korišten je primjer iz stvarnoga života. Može se zamisliti dućan koji već nekoliko godina odlično posluje i čiji je vlasnik odlučio proširiti asortiman robe. Vlasnik je ubrzo nakon proširenja uvidio problem skladišta u kojem više ne zna što mu se i koliko nalazi u skladištu. Vlasnik je odlučio angažirati vanjsku tvrtku da mu izradi sustav za upravljanje zalihama robe.

Za izradu navedene aplikacije korišten je Blazor Server predložak (engl. *Template*) unutar Visual Studia koji omogućuje izradu *web* aplikacije s bogatim i dinamičkim korisničkim sučeljem. Što se tiče .NET okvira (eng. *Framework*), odabran je .NET 6.0 zbog dugoročne podrške koju pruža tvrtka Microsoft. Blazor Server je odabran s razlogom što omogućuje izvršavanje C# *koda* unutar internet preglednika, čime programeru nisu potrebna znanja JavaScript programskog jezika koji se uglavnom koristi kod razvoja *web* aplikacija. Programer nije ograničen s korištenjem samo C# jezika već je moguće koristiti i JavaScript jezik prema potrebi [27].

Nakon kreiranja projekta bilo je potrebno implementirati klasu koja predstavlja robu, odnosno artikl. Izgled klase artikla, nazvane Item može se vidjeti na slici 5.1. Zbog standardizacije *koda,* priložene slike *koda* sadrže englesku terminologiju. U pravoj aplikaciji klasa Item bi bila puno opširnija, ali za potrebe ovog primjera ovdje je pojednostavljena. Podatke o artiklima potrebno je imati negdje skladištene te ih je moguće uređivati i dohvatiti,

zbog čega je kreirana klasa InventoryService. Navedena klasa je dodana kao *singleton*, čime se osigurava da postoji samo jedna instanca klase koja se može koristiti tijekom cijelog životnog ciklusa aplikacije. Izgled klase InventoryService može se vidjeti na slici 5.2.

```
namespace InventoryManagementSystem
{
    16 references
    public class Item
    {
        10 references
        public int Id { get; set; }
        10 references
        public string Name { get; set; }
        10 references
        public int QuantityInStock { get; set; }
        10 references
        public Double Price { get; set; }
        10 references
        public string Supplier { get; set; }
    }
}
```

Slika 5.1 Izgled klase koja reprezentira artikl

```
using InventoryManagementSystem.Shared;

namespace InventoryManagementSystem
{
    4 references
    public class InventoryService
    {
        private List<Item> Items;

        0 references
        public InventoryService()
        {
            Items = new List<Item>
            {
                new Item { Id = 1, Name = "Computer Mouse", QuantityInStock = 50, Price = 12.99, Supplier = "Mouse Supplier Inc." },
                new Item { Id = 2, Name = "Mechanical Keyboard", QuantityInStock = 30, Price = 79.99, Supplier = "Keyboard World" },
                new Item { Id = 3, Name = "Central Processing Unit (CPU)", QuantityInStock = 20, Price = 199.99, Supplier = "Tech CPU Corp." },
                new Item { Id = 4, Name = "Laptop", QuantityInStock = 15, Price = 799.99, Supplier = "Laptop Central" },
                new Item { Id = 5, Name = "External Hard Drive (1TB)", QuantityInStock = 40, Price = 59.99, Supplier = "Storage Solutions Ltd." },
                new Item { Id = 6, Name = "Graphics Card", QuantityInStock = 25, Price = 299.99, Supplier = "Graphics Producers Inc." },
                new Item { Id = 7, Name = "Wireless Router", QuantityInStock = 35, Price = 49.99, Supplier = "NetConnect Technologies" },
                new Item { Id = 8, Name = "Monitor (24-inch)", QuantityInStock = 20, Price = 129.99, Supplier = "Display Systems Co." },
                new Item { Id = 9, Name = "Gaming Headset", QuantityInStock = 30, Price = 59.99, Supplier = "GameGear Audio" },
                new Item { Id = 10, Name = "Inkjet Printer", QuantityInStock = 18, Price = 89.99, Supplier = "PrintMaster Inc." },
            };
        }

        1 reference
        public List<Item> getItems()
        {
            return Items;
        }
    }
}
```

Slika 5.2 Izgled klase koja reprezentira skladište

U .NET ekosustavu postoji nekoliko popularnih AOP okvira koji olakšavaju implementaciju i upravljanje aspektima u aplikacijama. Neki od poznatijih i popularnijih su: PostSharp, Castle Windsor, AspectDNG i drugi. Od navedenih okvira najviše se ističe PostSharp okvir koji je direktno namijenjen implementiranju koncepata aspektno orijentirane paradigme. Neki od problema u programiranju koje PostSharp adresira su: duplikacija *koda*,

složenost *koda*, praćenje performansi i sigurnosne provjere. PostSharp rješava navedene probleme raznim aspektima i mehanizmima te omogućuje programerima rješavanje navedenih problema, čineći razvoj *koda* čišćim, efikasnijim i održivijim. Koliko je okvir prihvaćen i primijenjen u raznim projektima, govori i činjenica o velikom broju preuzimanja samog NuGet paketa (preko 10,6 milijuna preuzimanja na NuGet.org platformi). Zbog navedenih razloga za ovaj praktični primjer odabran je PostSharp za implementaciju aspekata AOP-a [28].

PostSharp koristi aspektne klase za implementaciju AOP-a, čija se detaljnija dokumentacija kao što je opis konstruktora, svojstava i metoda opisuje na PostSharpovoj službenoj stranici. Aspektna klasa u PostSharpu je obična .NET klasa koja se koristi za definiranje ponašanja koje želimo primijeniti na određene dijelove *koda*. Atributi se koriste za označavanje određenih elemenata *koda* koji će biti podložni aspektima. Jedan od primjera korištenja je aspekt koji će pratiti izvršavanje metode označene specifičnim atributom. Atributi su ključni za PostSharp jer omogućavaju programerima da precizno odrede gdje i kako žele primijeniti aspekte u *kodu*. Atributi se primjenjuju na metode, klase, svojstva i druge elemente *koda* kako bi se definiralo ponašanje aspekta [28].

Neki od najvažnijih aspektnih klasa u PostSharpu su:

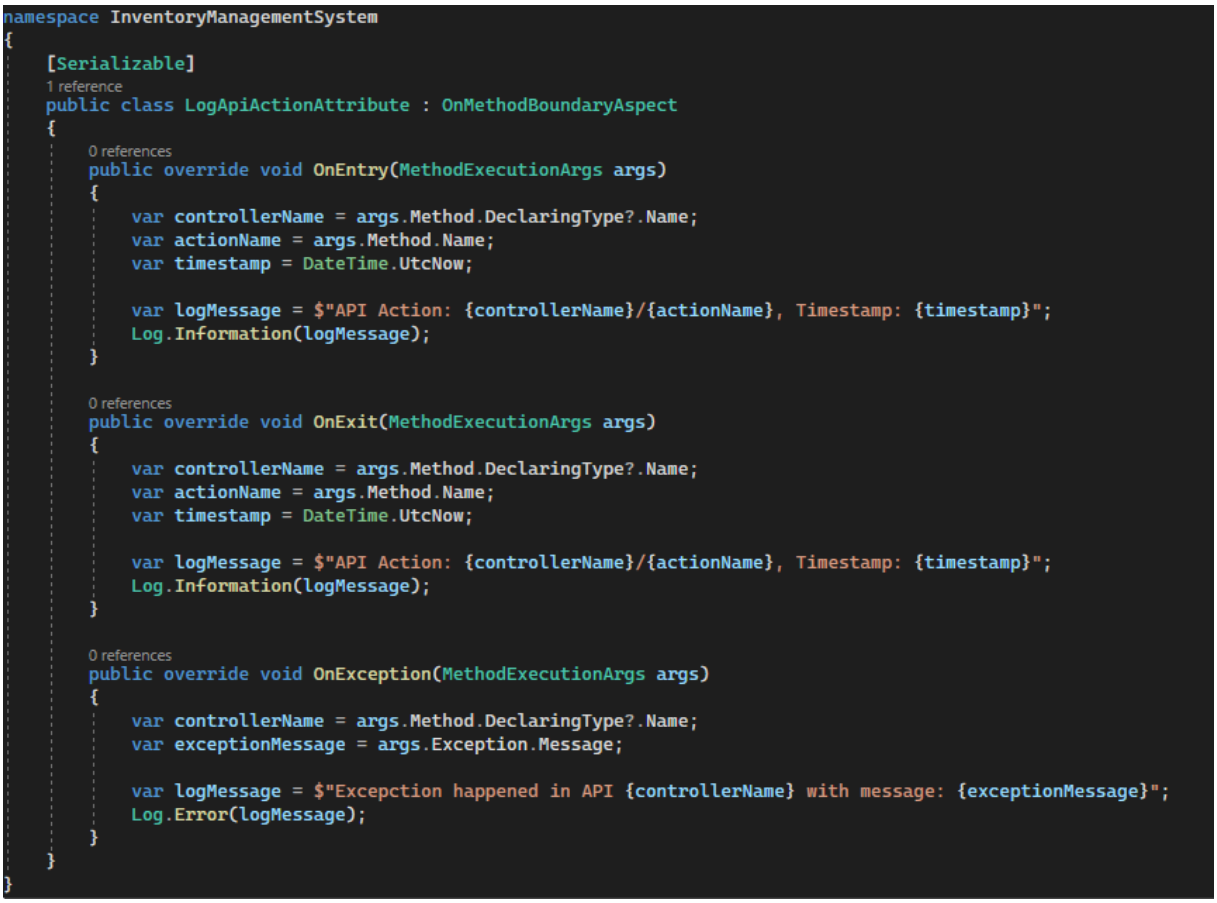
- Aspect: Ovo je bazna klasa koja omogućava stvaranje vlastitih aspektnih klasa. Programeri mogu naslijediti ovu klasu kako bi stvorili prilagođene aspekte za svoje specifične potrebe.
- OnMethodBoundaryAspect: Ova klasa omogućava definiranje aspekata koji će se izvršiti prije, nakon ili oko izvršavanja metode. Koristi se za implementaciju funkcionalnosti poput logiranja, mjerenja vremena izvršavanja i transakcija.
- MethodInterceptionAspect: Koristi se za izmjenu ponašanja metode tijekom njezina izvršavanja. Omogućava programerima da "zamotaju" metodu i prilagode njezino ponašanje prije i nakon izvršavanja.
- LocationInterceptionAspect: Ova klasa se koristi za dodatnu kontrolu i validaciju pristupa svojstvima i poljima. Omogućava programerima da definiraju posebno ponašanje kada se čitanje ili pisanje vrijednosti svojstva ili polja dogodi.
- MulticastAttribute: Ova klasa omogućava primjenu aspekata na više elemenata *koda* odjednom. To je korisno za primjenu istog aspekta na više metoda ili klasa.

5.1. OnMethodBoundaryAspect

U današnje vrijeme logiranje je neizostavni dio svake aplikacije, operativnog sustava ili bilo kojeg uređaja u mreži. *Log* je tekstualna datoteka koja se koristi za bilježenje automatski generirane i vremenski označene dokumentacije događaja, ponašanja i uvjeta relevantnih za određeni sustav. Izrada logova, odnosno dnevnika, ključna je zbog nekoliko razloga. Dnevnici bilježe informacije i događaje cijele aplikacije u njezinom radu na jednom određenom mjestu, pa su ključni za utvrđivanje uzroka problema. S druge strane, analiza dnevnika ključna je za sigurnost. Analiza dnevnika omogućila je razvoj alata za upravljanje događajima i sigurnosnim informacijama, kao što su SIEM sustavi (Sustavi za upravljanje informacijama o sigurnosti). Kod aplikacija koje su podijeljene u više modula, gdje svaka generira svoj dnevnik, ključno je imati dobar sustav za spajanje i filtriranje zapisa kako bi se našao uzrok problema u slučaju da do njega dođe. S obzirom na važnost logiranja unutar aplikacije, ono je odabrano kao prvi aspekt za implementaciju u aplikaciji [25].

U prvom koraku kod implementacije aspekta logiranja bilo je potrebno dodati sam PostSharp NuGet paket. S obzirom na to da u službenoj dokumentaciji PostSharp-a postoji podosta podržanih .NET biblioteka za logiranje. Istraživanjem prijašnje spomenutih biblioteka odabrana je Serilog biblioteka zbog velike popularnosti (skoro 900 milijuna skidanja na NuGet.org platformi) i broja preporuka na raznim forumima i člancima. Neki od ključnih razloga za odabir Serilog biblioteke su: jednostavna konfiguracija, bogatstvo odredišta za pohranu logova (konzola, datoteka, Elasticsearch itd.), strukturalni logovi, proširivost, aktivna zajednica, brzina i performanse. Serilog je unutar projekta bilo potrebno instalirati putem NuGet packet managera, konfigurirati kroz Program.cs klasu i testirati kako bi se moglo nastaviti na idući korak [25].

Idući korak je kreiranje klase koja nasljeđuje PostSharp-ovu klasu OnMethodBoundaryAspect, koja nam omogućuje da u našoj klasi umetnemo dio *koda* koji se može izvršavati u raznim točkama nekog drugog *koda,* kao što su prije *koda*, nakon *koda* ili kada se dogodi iznimka (engl. *Exception*). Događaj iznimke u *kodu* jako je značajan jer on označava da se tijekom izvršavanja *koda* dogodila greška koja je prekinula normalan protok izvršavanja *koda*. Na slici 5.3. je vidljiv izgled klase LogApiActionAttribute koja nasljeđuje klasu OnMethodBoundaryAspect.



```
namespace InventoryManagementSystem
{
    [Serializable]
    1 reference
    public class LogApiActionAttribute : OnMethodBoundaryAspect
    {
        0 references
        public override void OnEntry(MethodExecutionArgs args)
        {
            var controllerName = args.Method.DeclaringType?.Name;
            var actionName = args.Method.Name;
            var timestamp = DateTime.UtcNow;

            var logMessage = $"API Action: {controllerName}/{actionName}, Timestamp: {timestamp}";
            Log.Information(logMessage);
        }

        0 references
        public override void OnExit(MethodExecutionArgs args)
        {
            var controllerName = args.Method.DeclaringType?.Name;
            var actionName = args.Method.Name;
            var timestamp = DateTime.UtcNow;

            var logMessage = $"API Action: {controllerName}/{actionName}, Timestamp: {timestamp}";
            Log.Information(logMessage);
        }

        0 references
        public override void OnException(MethodExecutionArgs args)
        {
            var controllerName = args.Method.DeclaringType?.Name;
            var exceptionMessage = args.Exception.Message;

            var logMessage = $"Excepction happened in API {controllerName} with message: {exceptionMessage}";
            Log.Error(logMessage);
        }
    }
}
```

Slika 5.3 Izgled klase LogApiActionAttribute

Unutar klase LogApiActionAttribute prvo je potrebno uočiti atribut Serializable koji indicira da tip klase može biti serijaliziran koji je ključan da bi klasa i njene metode radile. Nadalje, unutar klase mogu se vidjeti tri različite metode. Prva metoda nasljeđuje OnEntry metodu i nadopunjuje ju logikom koja će se izvršiti prije neke druge logike, dok će se logika metode OnExit izvršiti nakon. Zanimljiva je metoda OnException koja će se izvršiti samo ako se u nekom dijelu *koda* dogodi greška, odnosno iznimka. Ključna je ovdje i uloga klase MethodExecutionArgs, čiju instancu dobivamo kroz parametar metode. Instanca navedene klase nosi važne podatke kao što su podaci o metodi uz koji se poziv metode OnEntry veže ili pak opis iznimke ako je potrebno doći do njega. Sama logika metoda je jednostavna jer sprema u varijable važne podatke koji se trebaju logirati.

S obzirom na način na koji je klasa LogApiActionAttribute implementirana može se koristiti kao atribut.. Sam njezin naziv i implementacija govore da se treba koristiti za logiranje nakon akcija u API klasama. Na slici 5.4 je vidljiv *kod* klase kontrolera u kojem se koristi prethodno navedeni atribut klase LogApiActionAttribute. Implementiranim se postiglo da se

zapis (engl. *Log*) ispisuje prije i poslije pozivanja Get metode unutar API-ja i u slučaju ako dođe do iznimke.

```
namespace InventoryManagementSystem.Controller
{
    [ApiController]
    [Route("[controller]")]
    1 reference
    public class InventoryController : ControllerBase
    {
        private InventoryService _inventoryService;

        0 references
        public InventoryController(InventoryService inventoryService)
        {
            _inventoryService = inventoryService;
        }

        [HttpGet]
        [LogApiAction]
        0 references
        public IEnumerable<Item> Get()
        {
            return _inventoryService.getItems().ToArray();
        }
    }
}
```

Slika 5.4 Izgled API kontrolera za skladište

Izgled ispisa zapisa u konzoli može se vidjeti na slici 5.5 gdje se zapis ispisuje i prije i nakon poziva API metode. Neke od korisnih primjena ovog aspekta uključuju situacije kada je API zaštićen autorizacijom i autentifikacijom, gdje bi se za svaki poziv u logovima bilježilo tko, kada i što je radio. Time bi se unutar aplikacije moglo adresirati pitanje sigurnosti. Zatim mogućnost ispisa prije i nakon izvršenja neke akcije, kao što je na primjer na bazi, bila bi korisna razvojnom timu aplikacije tijekom razvoja i održavanja same aplikacije jer bi se brzo vidjeli rezultati akcija i ako je došlo do pogreške koja nije rezultirala iznimkom (logička greška postoji u aplikaciji).

```
[23:07:45 INF] API Action: InventoryController/Get, User: , Timestamp: 18.9.2023. 21:07:36
[23:07:45 INF] API Action: InventoryController/Get, User: , Timestamp: 18.9.2023. 21:07:45
```

Slika 5.5 Izgled konzole u kojoj se nalaze ispisi logova

Treba još navesti da je kod prvog pokretanja aplikacije nakon dodavanja PostSharp NuGet paketa bilo potrebno ispuniti kratku anketu koju je postavio PostSharp tim i prihvatiti uvjete korištenja nakon čega se aplikacija pokrenula i krenula s radom. Cilj napravljenog primjera bio je pokazati kako koristeći OnMethodBoundaryAspect aspektnu klasu PostSharpa napraviti svoju klasu koja u nekom dijelu aplikacije, kao što su to u ovom slučaju bile API klase, rješava aspekte logiranja i obrade pogrešaka u *kodu*.

5.2. MethodInterceptionAspect

Jedna od aspektnih klasa PostSharpa koje treba spomenuti zasigurno je MethodInterceptionAspect klasa koja, kao što je i prije spomenuto, se koristi za izmjenu ponašanja metode tijekom njezina izvršavanja. Jedna od zanimljivijih primjena navedene klase bila bi aspekt keširanja. Keširanje je tehnika skladištenja i ponovnog korištenja podataka kako bi se ubrzao pristup tim podacima. Umjesto da se podaci svaki put iznova dohvaćaju iz skladišta podataka, mogu se privremeno pohraniti u brži i lakše dostupan spremnik podataka, poznat kao engl. *cache*. Keširani podaci se često koriste i ažuriraju kako bi se osiguralo da uvijek odražavaju najnovije informacije. Keširanjem podataka znatno se smanjuje potreba za pristupanjem skladištu podataka, kao što su baze podataka ili udaljeni servisi. Jedan od glavnih razloga za korištenje keširanja je poboljšanje performansi aplikacije [32].

Na primjeru klase CachingAttribute, koja je vidljiva na slici 5.6, može se vidjeti primjer korištenja aspektne klase MethodInterceptionAspect. Klasa se sastoji od metode OnInvoke koja koristi .NET klasu MemoryCache koja omogućuje keširanje podataka direktno u radnu memoriju računala na kojem se aplikacija izvršava. Za prikupljanje, spremanje i organiziranje podataka, MemoryCache klasa koristi ključeve kojima se jednostavno može dohvatiti željeni podatak. Metoda OnInvoke ima zadatak prvo provjeriti je li podatak keširan te, ako nije, podatak će biti *keširan* u radnu memoriju kako bi ga se brzo moglo dohvatiti. Također, kod dodavanja podataka u *keš* moguće je postaviti vrijeme koliko dugo će se podatak nalaziti u kešu.

```
namespace InventoryManagementSystem
{
    1 reference
    public class CachingAttribute : MethodInterceptionAspect
    {
        private readonly string cacheKey;
        private readonly int cacheDurationMinutes;

        0 references
        public CachingAttribute(string cacheKey, int cacheDurationMinutes)
        {
            this.cacheKey = cacheKey;
            this.cacheDurationMinutes = cacheDurationMinutes;
        }

        0 references
        public override void OnInvoke(MethodInterceptionArgs args)
        {
            ObjectCache cache = new System.Runtime.Caching.MemoryCache("InventoryCache");

            var cachedResult = cache.Get(cacheKey);

            if (cachedResult != null)
            {
                args.ReturnValue = cachedResult;
                Log.Information($"Cache hit for key: {cacheKey}");
            }
            else
            {
                args.Proceed();

                cache.Add(cacheKey, args.ReturnValue, new CacheItemPolicy
                {
                    AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(cacheDurationMinutes)
                });
                Log.Information($"Cache miss for key: {cacheKey}. Caching result.");
            }
        }
    }
}
```

Slika 5.6 Izgled klase CachingAttribute

Na slici 5.7 moguće je vidjeti kako se koristi klasa CachingAttribute nad nekom drugom klasom pomoću atributa. Važno je unutar atributa uočiti broj 10 koji označava koliko će dugo podaci biti keširani.

```
[CachingAttribute("GetInventoryItems", 10)]
1 reference
public List<Item> getItems()
{
    return Items;
}
```

Slika 5.7 Korištenje atributa CachingAttribute

Kroz stvarne primjere korištenja aspektnih klasa pokazalo se kako kroz aspektne klase, PostSharp omogućava programerima da precizno definiraju i primijene aspekte u svojoj aplikaciji, poboljšavajući modularnost i održivost *koda* te olakšavajući implementaciju AOP-a u .NET projektima. Korištenje PostSharp aspekata donosi brojne prednosti, uključujući bolju modularnost, mogućnost ponovnog korištenja *koda*, čitljivost *koda* i smanjenje ponavljanja. Implementacija AOP-a pomoću PostSharp-a može biti ključna za razvoj skalabilnih, održivih i aplikacija visokih performansi.

6. Zaključak

Aspektno orijentirana paradigma u programiranju povećava modularnost odvajanjem presijecajućih dužnosti iz poslovne logike. Aspekt je ključna jedinica modularnosti koja omogućuje dodavanje funkcionalnosti bez kompliciranja osnovnog *koda*. Aspektno orijentirano programiranje (AOP) pruža brojne prednosti u razvoju softvera. AOP omogućuje modularizaciju presijecajućih dužnosti kao što su logiranje i sigurnost, čime se olakšava organizacija *koda* i usredotočenje na primarnu funkcionalnost. Potiče ponovnu upotrebu *koda* i dosljednu primjenu pravila kroz aplikaciju, čime se smanjuje dupliciranje i pojednostavljuje održavanje. Izdvajanjem problema presijecanja dužnosti, olakšava se održavanje, testiranje i uklanjanje pogrešaka.

Ova paradigma također promiče odvajanje odgovornosti, skalabilnost, kompatibilnost, suradnju timova i prilagodljivost, čineći je korisnim alatom za razvoj sofisticiranih softverskih aplikacija.

U koncepte AOP-a spadaju aspekti kao moduli *koda* koji služe za odvajanje presijecajućih dužnosti, točke spajanja kao točke izvođenja, savjeti kao radnje koje aspekti izvršavaju na tim točkama, točke reza koje definiraju kriterije za odabir tih točaka, ciljani objekt na koji se primjenjuju aspekti, tkanje koje označava proces integracije aspekata u *kod* tijekom kompajliranja, učitavanja ili izvođenja te zastupnik koji se kreira nakon primjene aspekata na ciljani objekt. Razumijevanje ovih termina pomaže programerima bolje organizirati i održavati softverske aplikacije koristeći AOP paradigmu.

Postoji nekoliko najpoznatijih i najkorištenijih okvira u aspektno orijentiranom programiranju. AspectJ, koji pruža napredne značajke i performanse i ima bogatu jezičnu podršku, Spring AOP koji je integriran u Spring okvir te nudi jednostavniju upotrebu i konfiguraciju za osnovne AOP potrebe i PostSharp koji je specifičan za .NET i omogućava visoku fleksibilnost i izražajnost za AOP u .NET aplikacijama. Navedeni okviri su slični ,međutim imaju svoje prednosti i mane, kao i najbolje slučajeve upotrebe, pa se tako odabir između ovih okvira svodi na potrebe u projektu i preference programera. U radu je priložena i tablica u kojoj su uspoređeni okviri s obzirom na 10 bitnih karakteristika koje pomažu pri odabiru okvira.

Usporedba između objektno orijentiranog programiranja i aspektno orijentiranog programiranja pokazuje dvije značajne programske paradigme. OOP se temelji na konceptima

objekata, klasa, enkapsulacije i organizacije *koda*, koje su uvelike doprinijele razvoju programskog inženjerstva, ali isto tako mogu dovesti do narušavanja performansi, problema s nasljeđivanjem i kompleksnosti paralelizma. Zbog toga postoji AOP koji se fokusira na modularnost, odvajanje presijecajućih odgovornosti od glavne poslovne logike, poboljšavajući održavanje i čitljivost. Odabir između njih ovisi o potrebama projekta, pri čemu AOP proširuje OOP koncepte za bolje upravljanje složenim aplikacijama.

U praktičnom primjeru je demonstrirano kako se AOP paradigma primjenjuje uz pomoć PostSharp okvira u razvojnom okruženju Visual Studio. Primjer je odrađen u Visual Studio-u, .NET 6.0 framework-u i C# programskom jeziku. Pomoću aspekata riješeni su problemi logiranja i keširanja u aplikaciji, poboljšavajući modularnost i održivost *koda*. PostSharp i AOP olakšavaju rješavanje složenih problema na modularan način. Ovaj primjer ilustrira kako AOP pridonosi razvoju skalabilnih, održivih i brzih aplikacija u Visual Studio okruženju.

Literatura

[1] Neha Vaidya: Spring AOP Tutorial – AOP for Beginners with Examples, https://www.edureka.co/blog/spring-aop-tutorial/

[2] SpringSource: Aspect Oriented Programming with Spring, https://docs.spring.io/spring-framework/docs/2.5.5/reference/aop.html

[3] Simplilearn: Spring AOP Tutorial, https://www.simplilearn.com/tutorials/spring-tutorial/spring-aop-aspect-oriented-programming

[4] AspectJ Team: The AspectJTM Programming Guide, https://eclipse.dev/aspectj/doc/released/progguide/index.html

[5] Quan Vo: An Introduction to Aspect-Oriented Programming, https://saigontechnology.com/blog/an-introduction-to-aspect-oriented-programming

[6] Baeldung: Intro to AspectJ, https://www.baeldung.com/aspectj

[7] Baeldung: Intro to Spring AOP, https://www.baeldung.com/spring-aop

[8] javaTpoint: Spring Boot AOP, https://www.javatpoint.com/spring-boot-aop

[9] Bora Kartal, Ece Guran Schmidt : An evaluation of aspect oriented programming for embedded real-time systems, https://www.researchgate.net/figure/Use-of-Aspect-Oriented-Programming-24_fig8_4321119

[10] Benefits of aop (aspect oriented programming), JAVA Programming, https://www.expertsmind.com/questions/benefits-of-aop-(aspect-oriented-programming)-3017856.aspx

[11] Pankaj: Spring AOP Example Tutorial - Aspect, Advice, Pointcut, JoinPoint, Annotations, XML Configuration, https://www.digitalocean.com/community/tutorials/spring-aop-example-tutorial-aspect-advice-pointcut-joinpoint-annotations

[12] Eclipse: ntroduction to AspectJ, https://eclipse.dev/aspectj/doc/released/progguide/starting-aspectj.html

[13] Spring: Aspect Oriented Programming with Spring, https://docs.spring.io/spring-framework/reference/core/aop.html

[14] GeeksForGeeks: Spring Boot – Difference Between AOP and AspectJ, https://www.geeksforgeeks.org/spring-boot-difference-between-aop-and-aspectj/

[15] HowToDoInJava: Spring AOP Tutorial, https://howtodoinjava.com/spring-aop-tutorial/

[16] Abdulcelil Cercenazi: AOP with Spring (Boot), https://reflectoring.io/aop-spring/

[17] tutorialsPoint: AOP with Spring Framework, https://www.tutorialspoint.com/spring/aop_with_spring.htm

[18] Satyam Tripathi: Procedural Programming and Object Oriented Programming in C++, https://www.scaler.com/topics/cpp/procedural-programming/

[19] Alexander S. Gillis: Object-oriented programming (OOP), https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

[20] RishabhPrabhu: Object Oriented Programming (OOPs) Concept in Java, https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/

[21] GeeksForGeeks: Spring Boot – Difference Between AOP and OOP, https://www.geeksforgeeks.org/spring-boot-difference-between-aop-and-oop/

[22]Hamza Belmellouki: Introduction to AOP and Spring AOP, https://dev.to/hamzajvm/introduction-to-aop-and-spring-aop-3ff7

[23] Nam Ha Minh: Understanding Spring AOP, https://www.codejava.net/frameworks/spring/understanding-spring-aop

[24] DifferenceBetween: Difference Between AOP and OOP, https://www.differencebetween.com/difference-between-aop-and-vs-oop/

[25] javabyKiran: Disadvantages of AOP, https://www.jbktutorials.com/spring-aop/disadvantages-of-spring-aop.php#gsc.tab=0

[26] Službena stranica Visual Studia, https://visualstudio.microsoft.com/

[27] Build beautiful web apps with Blazor, https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor

[28] Službena stranica PostSharp, https://www.postsharp.net/

[29] Alexander La rosa: Log Monitoring: not the ugly sister, https://web.archive.org/web/20180214153657/https://blog.pandorafms.org/log-monitoring/

[30] Službena stranica Seriloga, https://serilog.net/

[31] Joydip Kanjilal: How to program using PostSharp in C#, https://www.infoworld.com/article/3048204/how-to-program-using-postsharp-in-c.html

[32] Anurag Patel: Why “Caching” is important?, https://medium.com/geekfarmer/why-caching-is-important-853d4fe16ca8

Sažetak

Cilj ovoga rada je opisati pojam aspektno orijentirane paradigme, specifične probleme koje koncept adresira i primjerom pokazati ulogu iste u razvoju aplikacije. U radu je opisano što je AOP i koje su njegove prednosti, odnosno njegova poboljšanja i načela. Obrađeni su dominantni okviri AOP-a na način da su opisani AspectJ, Spring i PostSharp razvojni okviri te uspoređeni na temelju 10 kategorija kao što su implementacija, integracija, performanse itd. Opisan je pojam aspekta kao ključne jedinice modularnosti koja omogućuje dodavanje funkcionalnosti bez kompliciranja osnovnog *koda*. Opisane su OOP i AOP paradigme, njihove uloge, prednosti, nedostaci te njihova povezanost. U završnom dijelu rada na praktičnom primjeru prikazano je kako uz PostSharp razvojni okvir adresirati razne aspekte AOP paradigme koristeći aspektne klase i atribute u C# Web aplikaciji.

Ključne riječi: aspekti, Aspektno orijentirana paradigma, C# Web aplikacija, izazovi presijecajuće dužnosti, okviri aspektno orijentiranog programiranja, PostSharp

Abstract

The goal of this paper is to describe the concept of an aspect-oriented paradigm, the specific problems that the concept addresses and to demonstrate its role in the development of applications with an example. The paper describes what AOP is and what its advantages are, i.e. its improvements and principles. The dominant frameworks of AOP are covered in such a way that AspectJ, Spring and PostSharp development frameworks are described and compared based on 10 categories such as implementation, integration, performance, etc. The concept of an aspect is described as a key unit of modularity that allows adding functionality without complicating the basic code. The OOP and AOP paradigms, their roles, advantages, disadvantages and their connection are described. In the final part of the work, a practical example shows how to address various aspects of the AOP paradigm using the aspect class and attributes in a C# Web application with the PostSharp development framework.

Key woards: aspects, Aspect-oriented paradigm, cross-cutting concern, C# Web application, frameworks for aspect-oriented programming, PostSharp

Prilozi

Prilog P1 - svi programski *kodovi* korišteni u 5. poglavlju nalaze se na *github* računu Mateja Kozarca. *Github* profil se nalazi na linku: https://github.com/matejkozarac, a repozitorij web aplikacije se nalazi na linku: https://github.com/matejkozarac/ZavrsniRad-InventoryManagementSystem.