

**SVEUČILIŠTE U SPLITU
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I
BRODOGRADNJE**

DIPLOMSKI RAD

**IZRADA WEB APLIKACIJE DOMAIN
DRIVEN DESIGN PRISTUPOM**

Ivan Brković

Split, rujan 2017.



SVEUČILIŠTE U SPLITU
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I
BRODOGRADNJE



Diplomski studij: **Računarstvo**

Oznaka programa: 250

Akadska godina: 2016./2017.

Ime i prezime: **IVAN BRKOVIĆ**

Broj indeksa: 694-2015

ZADATAK DIPLOMSKOG RADA

Naslov: **IZRADA WEB APLIKACIJE DOMAIN DRIVEN DESIGN PRISTUPOM**

Zadatak: Izučiti Domain Driven Design pristup izradi aplikacija. Analizirati prednosti ovog pristupa u odnosu na klasične pristupe i mogućnosti primjene kod izrade poslovnih aplikacija. Izučiti koncepte izrade web aplikacija pomoću Symfony frameworka otvorenog koda. Dizajnirati, realizirati i testirati web aplikaciju za izdavanje računa prateći Domain Driven Design.

Prijava rada: 1.3.2017.

Rok za predaju rada: 22.9.2017.

Rad predan:

Predsjednik
Odbora za diplomski rad:

Mentor:

prof. dr. sc. Sven Gotovac

doc.dr.sc.Ljiljana Šerić

IZJAVA

Ovom izjavom potvrđujem da sam diplomski rad s naslovom IZRADA WEB APLIKACIJE DOMAIN DRIVEN DESIGN PRISTUPOM pod mentorstvom doc. dr. sc. LJILJANA ŠERIĆ pisao samostalno, primijenivši znanja i vještine stečene tijekom studiranja na Fakultetu elektrotehnike, strojarstva i brodogradnje, kao i metodologiju znanstveno-istraživačkog rada, te uz korištenje literature koja je navedena u radu. Spoznaje, stavove, zaključke, teorije i zakonitosti drugih autora koje sam izravno ili parafrazirajući naveo/la u diplomskom radu citirao/la sam i povezao/la s korištenim bibliografskim jedinicama.

Student/ica

Ivan Brković

SADRŽAJ:

1	UVOD	1
2	DOMAIN DRIVEN DESIGN.....	3
2.1	Zajednički jezik i model	7
2.2	Model koda	7
2.3	Kontekst domene i strateški principi	11
2.3.1	Povezivanje ograničenih konteksta	12
3	TAKTIČKI PRINCIPI	15
3.1	Entitet.....	15
3.2	Vrijednosni objekt	16
3.3	Servis	17
3.4	Modul.....	18
3.5	Agregat	18
3.6	Factory	19
3.7	Repositorij	20
4	ARHITEKTURE APLIKACIJA	21
4.1	MVC	22
4.2	Slojevita arhitektura.....	23
5	WEB APLIKACIJA ZA IZDAVANJE RAČUNA	27
5.1	Uspostava zajedničkog jezika.....	28
5.2	Specifikacija zahtjeva	32
5.2.1	Opseg aplikacije	32
5.2.2	Pregled sustava	32
5.2.3	Funkcionalni zahtjevi	33
5.3	Dizajn domene	35
5.4	Tehnologije implementacije	38
5.4.1	PHP.....	38
5.4.2	Symfony radno okruženje	39
5.4.3	Doctrine ORM.....	42
5.4.4	MySQL.....	45
5.5	Preslikavanje domene u model koda	45
5.6	Korištenje slojevite arhitekture.....	52

5.6.1	Aplikacijski sloj.....	52
5.6.2	Infrastrukturni sloj.....	53
5.6.3	Symfony u ulozi prezentacijskog sloja.....	54
5.7	Testiranje aplikacije.....	57
ZAKLJUČAK.....		61
LITERATURA		63
POPIS OZNAKA I KARTICA		64
SAŽETAK.....		65
SUMMARY.....		66

1 UVOD

U posljednje vrijeme sve više i više dolazi do razvoja softvera te je softverska industrija vodeći pokretač mnogih modernih država. Mnoge države ulažu velike novce u softversku industriju kako bi ubrzali i modernizirali mnoge procese unutar svojih granica. Pa tako sve više i više dolazi do razvoja pametnih gradova, okoliša, zdravstva i mnogih drugih područja koji će ljudima znatno olakšati živote. Smisao softvera je upravo u tome da ljudima što više olakšava živote, a ne suprotno. Tu najveću odgovornost imaju softverski stručnjaci i da li će završni proizvod biti uspješan ili ne ovisi o njihovom znanju i umijeću. Kako se tehnologija razvija javljaju se potrebe za još većim brojem softverskih stručnjaka, pa stoga i ne čudi njihova velika potražnja pogotovo u razvijenijim državama.

Prema definiciji softver je računalni program i pripadajuća dokumentacija koja se odnosi na različite specifikacije, modele dizajna te mnoge druge dokumente. Inženjerska disciplina koja se bavi svim aspektima razvoja softvera (teorijom, metodama i alatima) je programsko inženjerstvo te se primjenom navedene discipline može izraditi što kvalitetniji softver. Svaki softver je definiran nizom procesa koje dovode do konačnog puštanja u rad. Kvalitetan softver bi trebao imati sljedeće softverske procese: specifikacija, dizajn, implementacija, validacija i održavanje. Softverski procesi mogu biti planirani, ili agilni koji su danas svakako najpopularniji. Uvođenje agilnih procesa donio je pozitivne pomake u brzini razvoja softvera budući da nema generiranja velike količine dokumentacije za koje se prije gubilo mnogo vremena. Domain-Driven Design (skraćeno DDD) princip koji je tema ovoga rada također koristi agilne metodologije razvoja softvera. Softver je alat koji nam omogućava da se lakše nosimo s različitim problemima koje susrećemo u životu, odnosno da nam omogući presliku stvarnih procesa što je moguće bolje. Oni se izrađuju kako bi automatizirali procese koji postoje u stvarnom svijetu, odnosno kako bi riješili poslovne probleme. Nemoguće je napraviti kompleksan softver bez širokog pogleda na problem. Kodiranje je jako važan korak pri razvoju softvera, ali ne mogu se preskočiti svi prethodni koraci i prijeći u programiranje. Taj pristup nema smisla, jer bez pravog poznavanja domene problema, bez pravog modela odnosno preslike stvarnih procesa nema ni kvalitetnog softvera. Može se reći da je dizajn softvera neka vrsta umjetnosti te ga je nemoguće izraditi koristeći uvijek ista pravila i obrasce. Svaki je softver jedinstven sa svojim ograničenjima i problemima, te će ga svaki programer izraditi na svoj jedinstveni način. Programerima i arhitektima softvera su dostupni alati i

principi koje kombinacijom te pravilnim izborom dovode do kvalitetnog završnog proizvoda. Bitno je odvojiti dva koncepta pri razvoju softvera, a to su dizajn softvera i dizajn koda. Softver dizajn je nešto poput dizajniranja arhitekture kuće (šira slika), dok je kod dizajn implementacija detalja primjerice gdje ćemo smjestiti nešto na zidu. Softver dizajn je najvažniji jer pogreške u ovoj fazi su najskuplje u kasnijim fazama.

Softverska industrija je od početka izrade softvera koristila niz principa razvoja sa svojim prednostima i nedostacima. Ovaj rad će govoriti o jednoj metodi koja se prvi put pojavila 2004. godine a to je DDD.

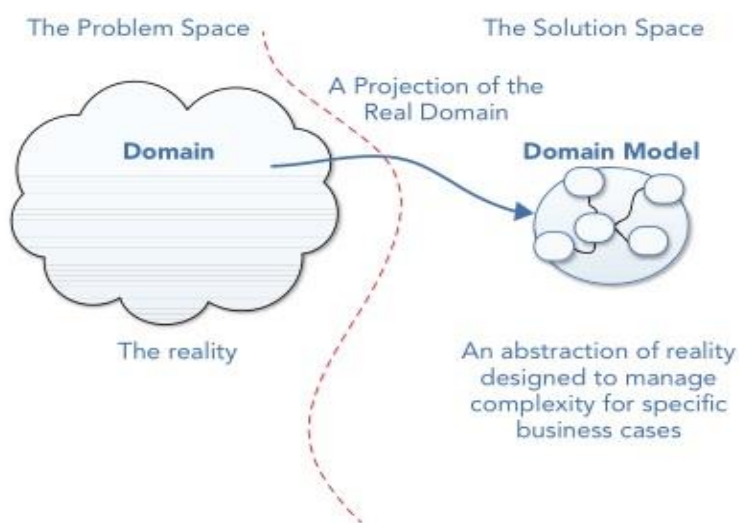
Rad je podijeljen u više poglavlja. Na početku će biti objašnjen detaljno DDD te pristup analiziranja problema od razgovora s domenskim stručnjakom i uspostave zajedničkog jezika do apstrakcije modela. Također će se reći nešto o tome za kakve projekte je on u stvarnosti namijenjen, te kako se nositi s kompleksnim domenama. Nakon toga će biti navedeni svi najvažniji taktički uzorci, odnosno građevni elementi DDD-a koji nam olakšavaju presliku stvarnosti u apstrakciju te pružaju učinkovitije i kvalitetnije programiranje. DDD je pristup koji je neovisan o arhitekturi, bitno je samo da je domena aplikacije izolirana. Biti će navedena slojevita arhitektura te njezine karakteristike kao osnova izrade kompleksnih aplikacija. Zadnje poglavlje je rezervirano za primjenu principa DDD-a za izradu web aplikacije za izdavanje računa. Tu će se pokušati kroz primjenu principa pokazati potpuni tijek razvoja aplikacije.

2 DOMAIN DRIVEN DESIGN

DDD je filozofija i pristup razvoja softvera definirana od softverskog dizajnera Eric Evansa. Prema njemu [1] DDD je način razmišljanja i skup prioriteta koje omogućavaju efikasnu izradu softverskih projekata koji se trebaju nositi s kompleksnim domenama. Naziv principa DDD određuju dvije važne stvari: domena i dizajn.

Prema Evans-u domena predstavlja problematično područje na koje korisnik primjenjuje program. Mnogo je problematičnih područja u stvarnosti s kojima se svi sigurno susreću. Smisao programa je da olakša rad u takvim problematičnim područjima (domenama), odnosno da automatizira što više procese unutar domene. Usvojenim znanjem problematične domene, koja ima svoje procese i pravila, omogućuje se dizajn domenskog modela.

Za model se može reći da je to selektivno pojednostavljeni i strukturirani oblik znanja koji predstavlja zajednički jezik kojeg dijele svi članovi tima i domenski stručnjaci. To je strogo organizirana apstrakcija prikupljenog znanja. Dizajn modela može biti u obliku čistih rečenica, dijagrama, slika ili bilo čega što nam daje korisne informacije. Za model je važno da rješava probleme u kontekstu aplikacije u kojoj je napravljen odnosno da rješava poslovne slučajeve korištenja (engl. *use cases*). Pri izvlačenju informacija na slici 2.1. iz stvarnog problematičnog domenskog prostora u apstrakciju domenskog modela bitno je ignorirati nepotrebne i nevažne informacije za implementaciju.



Slika 2.1. Preslikavanje domene u domenski model [3]

DDD pristup se prije svega bazira na dobroj međusobnoj komunikaciji između programera (ili tima programera) i domenskog stručnjaka koji je stručnjak određene domene. Pri tome je jako bitno držati stalni kontakt s poslovanjem kako bi domenski model bio u skladu s poslovanjem. Tako primjerice pri razvoju bankovnog softvera programeri imaju jako malo znanja o poslovanju banaka što im onemogućava daljnji razvoj. Programer može steći znanja kroz razgovor s različitim ljudima te kroz literaturu, ali svejedno to znanje nije dovoljno da bi se krenulo u razvoj softvera. Pri razvoju softvera prvo na što treba obratiti pozornost koje su potrebe klijenta koji koristi aplikaciju. Osoba koja ima najviše znanja o domeni je zasigurno bankar koji je stručnjak bankovne domene. On je stručnjak u svome području te poznaje sve poslovne procese, te sve zamke koje domena krije. Tako bankar postaje prva osoba s kojom treba ostvariti prvi kontakt te kroz prvi razgovor steći globalnu sliku domene. Takvih razgovora će biti više te će se vrlo brzo uspostaviti zajednički jezik (engl. *ubiquitous language*). Bitno je da je to znanje negdje zapisano jer nije dobro da se nalazi samo u glavi programera jer se brzo zaboravi. Ovako model dijele svi programeri u timu, softver arhitekti i domenski stručnjaci. To je centralno mjesto se sve promjene reflektiraju. Primjer komunikacije i uspostave zajedničkog jezika može se vidjeti najbolje u poglavlju 5. gdje je kroz razgovor programera i domenskog stručnjaka točno opisan način na koji se uspostavlja zajednički jezik u kontekstu implementirane web aplikacije u prilogu ovoga diplomskog rada. DDD princip osigurava da poslovanje i softver „govore“ isti jezik, odnosno da svatko u timu zna što se događa s poslovanjem.

Što se tiče problematičnih domena, one mogu biti manje i više kompleksne. Velike kompanije imaju unutar sebe jako puno poslovnih procesa kao što su nabava, skladištenje, proizvodnja, knjigovodstvo i računovodstvo te svi ti procesi generiraju jako puno dokumenata koji su nositelji podataka unutar procesa. Kod takvih velikih domena loše je napraviti jedan model koji će opisivati cijelu organizaciju. DDD princip je sve oko smanjivanja kompleksnosti, a jedan monolitan model bi sigurno povećao razinu kompleksnosti. Umjesto toga je potrebno problematičnu domenu podijeliti, odnosno napraviti proces podjele na manje modele koji će se nositi s problematičnom domenom. Na taj način možemo se usredotočiti na domenu koja je od presudnog značaja (engl. *core domain*). To je poddomena koju identificiraju vrijednosti kompanije na kojima se temelji, odnosno koja je od presudnog značaja za poslovanje kompanije. Za razvoj takve domene treba trošiti najviše resursa i staviti najbolje programere koje kompanija ima. Primjer podjele domene turističke agencije prikazan je na slici 2.2.

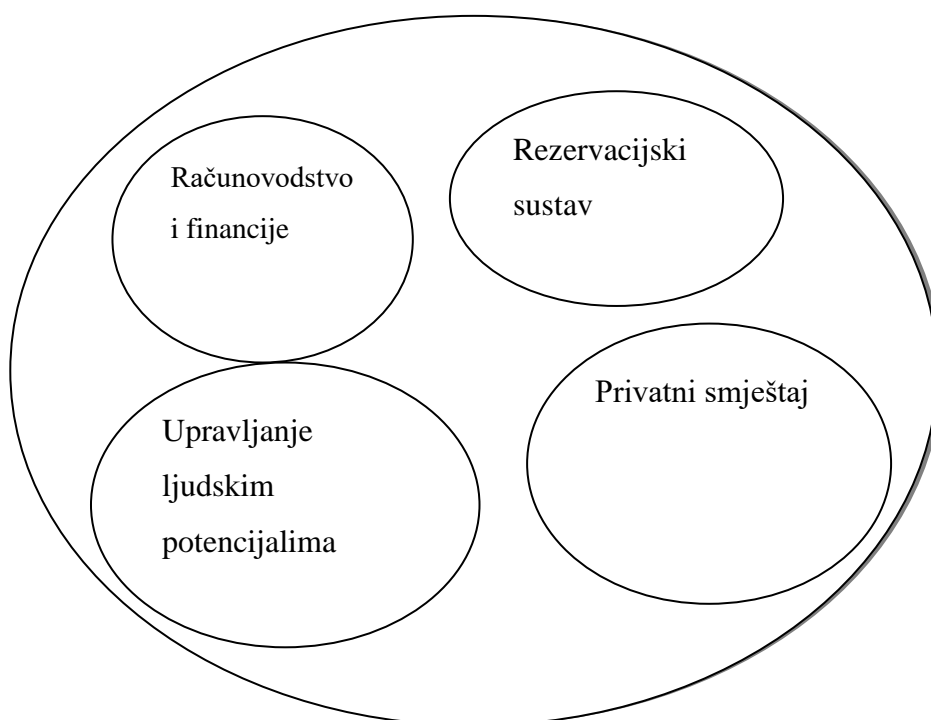
Osim jezgre domene postoje i:

- Domene za podršku poslovanju (engl. *support domain*)

To su poddomene koji neposredno podržava jezgru domene, ali nije dio nje.

- Generičke domene (engl. *generic domain*)

To su opće poznate poddomene koje svaka tvrtka ima te ih je vrlo lako nabaviti po pristupačnoj cijeni. Dobar primjer je za to upravljanje ljudskim potencijalima ili razni knjigovodstveni programi.



Slika 2.2. Prikaz podjele domene turističke agencije

Mnogi misle da je DDD uzorak, arhitekturni stil softvera ili da je metodologija razvoja softvera, međutim nije niti jedno od navedenog. Da bi softver imao obilježja DDD potrebno je jedino osigurati da se model drži izoliran u svojem sloju s implementiranom domenskom logikom, daleko od tehničke kompleksnosti. DDD je alat koji nam pruža principe i pravila da se napravi dobar kod, ali tek onda kada se uspije napraviti dobar dizajn domenskog modela. DDD nije metodologija razvoja već je korištena od neke od iterativnih softverskih metodologija.

Vjerojatno se svi pitaju kako se odlučiti kada koristiti DDD, odnosno je li ga potrebno koristiti u svim slučajevima. Odgovor je da nije, sve zavisi o kompleksnosti projekta na

kojemu se radi, odnosno sve zavisi o domeni koja može biti „bogata“ ili „siromašna“. Ukoliko nemamo nikakve domenske logike zasigurno DDD nije izbor. Ako radimo aplikaciju koja samo preslikava bazu kao što je primjerice CRUD (engl. *Create, Read, Update, Delete*) aplikacija, bolje je odabrati neki drugi pristup. Važno je i koliko će se aplikacija dalje razvijati te procijeniti kolike će biti promijene u budućnosti, odnosno da li će zahtijevati konstantne promijene. Tu postoji granica, a tu granicu je teško odrediti. To je odgovornost programera koji procjenjuju koju tehniku koristiti, a to se prije svega postiže iskustvom.

Dobre prakse i principi DDD-a [3]:

- Usmjeriti pažnju na srž domene koju možemo postići uspostavom poddomene unutar domene koja će biti najveća vrijednost za poslovanje koja će pružiti konkurentsku prednost i stvarati stvarnu vrijednost za poslovanje. Bitno je da svi unutar kompanije znaju koje je to područje.
- Učiti kroz suradnju.
DDD ističe važnost komunikacije između razvojnog tima i poslovnih stručnjaka kako bi se napravio koristan model koji rješava poslovne probleme.
- Kreirati model kroz istraživanje i eksperimentiranje.
DDD princip koristi domenski model kao spoj dizajna modela i modela koda. Navedena dva modela moraju biti cijelo vrijeme sinkronizirani. Potrebno je ostaviti odrađeno vrijeme za eksperimentiranje s modelom jer je jako teško uspostaviti iz prve najbolji model. Eric Evans ističe u svojoj knjizi [1] da za svaki dobar model moraju biti bar 3 loša što znači da za model trebamo izdvojiti dovoljno vremena.
- Komunikacija.
Domenski stručnjak i programer rade i surađuju zajedno kako bi razmijenili znanja i ideje te uspostavili zajednički jezik za poslovno područje koje treba razviti. Za ovu fazu treba izdvojiti dovoljno vremena jer je jako važna kako bi programer i domenski stručnjak imali isti pogled na problem te kako bi stečeno domensko znanje na neki način pohranili kroz razne dijagrame, nacрте i dokumente. Ovo predstavlja temeljnu vrijednost na kojoj se DDD temelji.
- Razumjeti primjenu modela.
Unutar velikih organizacija može dolaziti do dvosmislenosti u primjenama zajedničkog jezika gdje različiti dijelovi organizacije imaju različito razumijevanje zajedničkih pojmova. DDD rješava slučaj na način da se uspostavi više konteksta

unutar kojih modeli imaju značenje, odnosno konteksti uspostave jezična ograničenja unutar jednog konteksta.

- Stalno razvijati model.

Stalno držati sinkronizaciju između izvornog koda i problematične domene, te surađivati sa svim osobama unutar kompanije.

2.1 Zajednički jezik i model

Za stvaranje potpunog modela s bogatim znanjem (engl. *knowledge-rich design*) zahtijeva se stvaranje zajedničkog jezika i uspostave komunikacije sa svim akterima u procesu stvaranja modela. Domenski stručnjaci i programeri imaju drugačiji pogled na problem i način razmišljanja. Programeri su ti koji razmišljaju na apstraktan način kroz objekte i principe objektnog orijentiranog programiranja, dok domenski stručnjaci imaju limitirano znanje o tehničkom žargonu. Zbog toga se treba uspostaviti zajednički jezik koji postaje način komunikacije između svih članova tima i domenskih stručnjaka. Sredstvo komunikacije može biti putem dijagrama, nacрта, rečenica ili pak putem govora. Svi članovi tima moraju koristiti isti žargon. Jako je bitno taj jezik koristiti također u programiranju. Također treba osigurati da promjena u zajedničkom jeziku reflektira i promjenu u modelu.

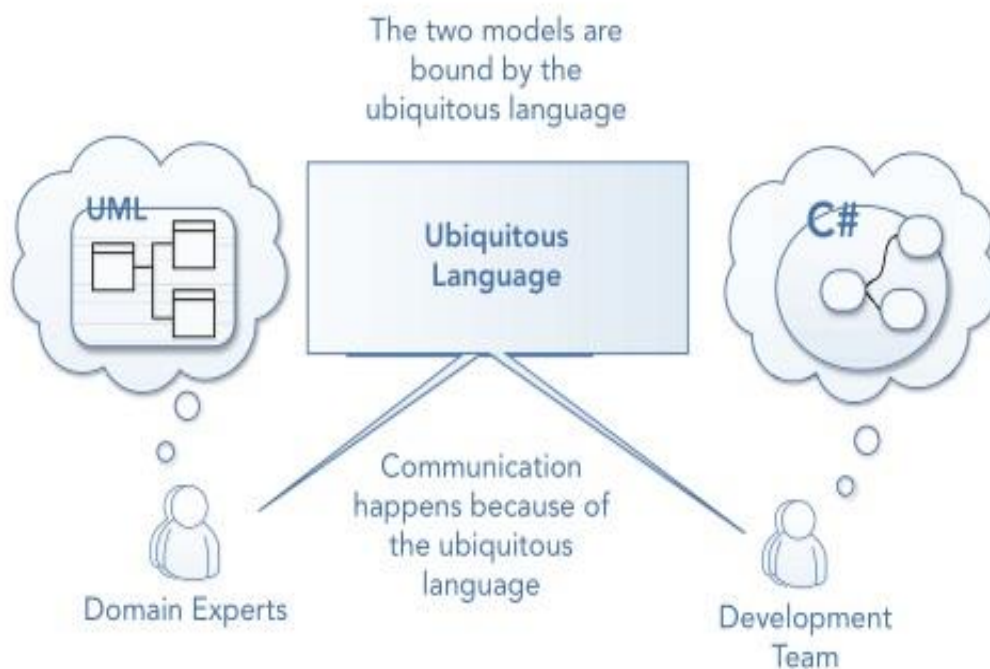
Model se u većini slučajeva opisuje UML jezikom (engl. *Unified modelling language*) koji je standardizirani jezik za planiranje razvoja softvera. Pri tome se softver modelira pomoću raznovrsnih dijagrama gdje svaki prikazuje sustav iz drugačije perspektive. Za potrebe DDD najvažniji je svakako dijagram klasa (engl. *class diagram*) koji prikazuje sustav iz perspektive objektno orijentiranog programiranja. Za model je bitno da osim veza između klasa opisuje i ponašanje sustava što se postiže kroz ubacivanje teksta u dijagrame.

2.2 Model koda

Proces koji osigurava povezivanje napravljenog domenskog modela sustava u model implementacije koda zove se Model-Driven Design (skraćeno MDD). To je proces koji je usmjeren na implementaciju i na sva ograničenja koja mogu zahtijevati promjene početnog modela [3]. Omogućuje da domensko znanje i zajednički jezik budu implementirani u modelu koda. Također je potrebno da domenski modeli dizajna sustava i koda budu cijelo vrijeme u sinkronizaciji. To ima za posljedicu podržavanje suradnje između domenskog stručnjaka i

programera jer su sposobni rješavati probleme zajedno, budući da imaju sinkronizirane i validne svoje modele.

U nekim kompanijama postoji osoba zadužena za modeliranje dizajna domenskog modela a zove se analitičar softvera (engl. *software analysts*). Navedena osoba vodi komunikaciju s domenskim stručnjacima i napravi model analize koje zatim prenese programerima. Međutim, navedeni pristup nije baš najbolje rješenje jer i programeri trebaju sudjelovati u komunikaciji s domenskim stručnjakom. Oni su ti koji će na koncu implementirati model. Sudjelovanje programera u raspravama je ključno da se odabere ispravan domenski model kojeg će biti jednostavno preslikati u kod.



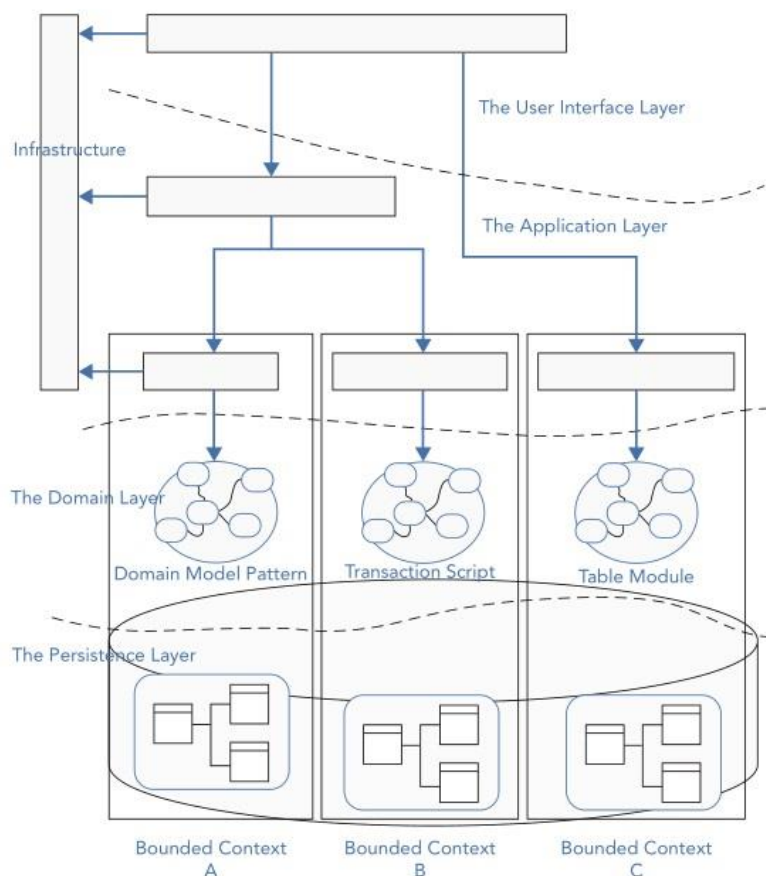
Slika 2.3. Uspostavljanje zajedničkog jezika [3]

Često se događa da se pri implementaciji koda na osnovu modela analize otkriju neke nelogičnosti i neslaganja između domenskog modela i stvarnosti, te nastavi implementirati softver bez povratne informacije domenskom stručnjaku ili analitičaru sustava. Kako programer sve više i više odmiče od domenskog modela on postane sve manje i manje koristan, te tako model koda više ne reflektira domenski model. Da do takve situacije ne bi dolazilo potrebna je stalna suradnja i izmjena znanja između svih sudionika u procesu. Na slici 2.3. može se vidjeti da svi moraju raditi kao jedan tim služeći se uspostavljenim zajedničkim jezikom za model analize (domenski model) i model koda. To ima za posljedicu

da primjerice programeri uspijevaju razumjeti sve termine domene i da razumiju kompleksnu logiku. Poslovni analitičari mogu steći iskustva u tehnikama modeliranja, dok poslovni stručnjaci isto tako mogu steći znanja o modeliranju i mogu validirati modele s programerima.

Pri prijelazu modela analize u model koda često nastaje različito korištenje zajedničkog jezika. Programeri jezik često jezik poslovanja prevode u svoj tehnički jezik. Da bi se to spriječilo potrebno je uvijek zajednički jezik implementirati i u kodu, te naučiti sve sudionike uključene u proces da koriste isti jezik. Treba shvatiti da je model reprezentiran kao model analize i model koda. Oni moraju biti identični, te MDD povezuje ta 2 modela u zajednički jezik.

Postoji više principa za reprezentaciju domenskog modela u formi koda. Sve zavisi o tome koliko je model „bogat“ ili „siromašan“.



Slika 2.4. Principi za reprezentaciju domenskog modela [3]

Iz slike 2.4. se može primijetiti da domenski sloj koji je srce aplikacije čini samo manji dio cjelokupne aplikacije. Postoji više načina implementacije domenskog modela:

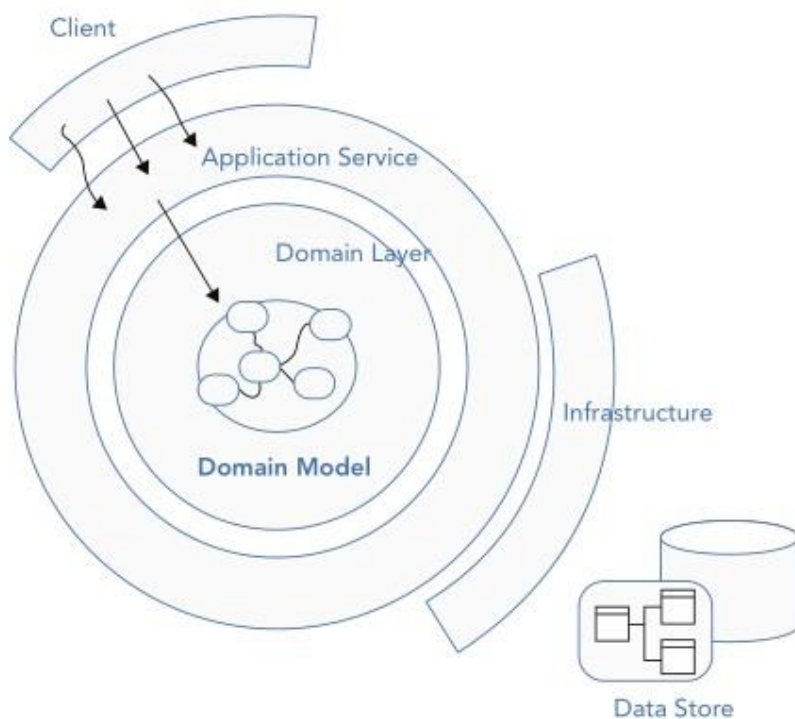
- Domenski model.
- Transakcijska skripta (engl. *transaction script*).

Princip grupira svu domensku logiku da zadovolji poslovne transakcije ili slučaj korištenja u obliku proceduralnog modula. Jednostavna procedura je kreirana za svaku poslovnu transakciju, te ona ima više odgovornosti.

- Tablični modul (engl. *table module*).

Princip reprezentira podatke u objektnoj formi. Nema neslaganja između objektnog modela i modela baze podataka.

Domenski model uzorak koristi objektno orijentirani model koji kombinira ponašanje i podatke, te uključuje poslovne procese i poveznice, pravila i bogatu domensku logiku. Bazira se na pretpostavci da ne postoji baza podataka. Klase unutar modela su oslobođene znanja o načinu spremanja i dohvaćanja podataka što se može vidjeti na slici 2.5. Ovaj uzorak se koristi uvijek kada imamo kompleksnu logiku i kod koji se često mijenja. Zbog toga ga treba znati kada koristiti budući da zahtijeva dosta znanja i truda te također programere za izvrsnim poznavanjem objektno orijentiranog programiranja.



Slika 2.5. Izdvojenost domenskog modela od ostatka aplikacije [3]

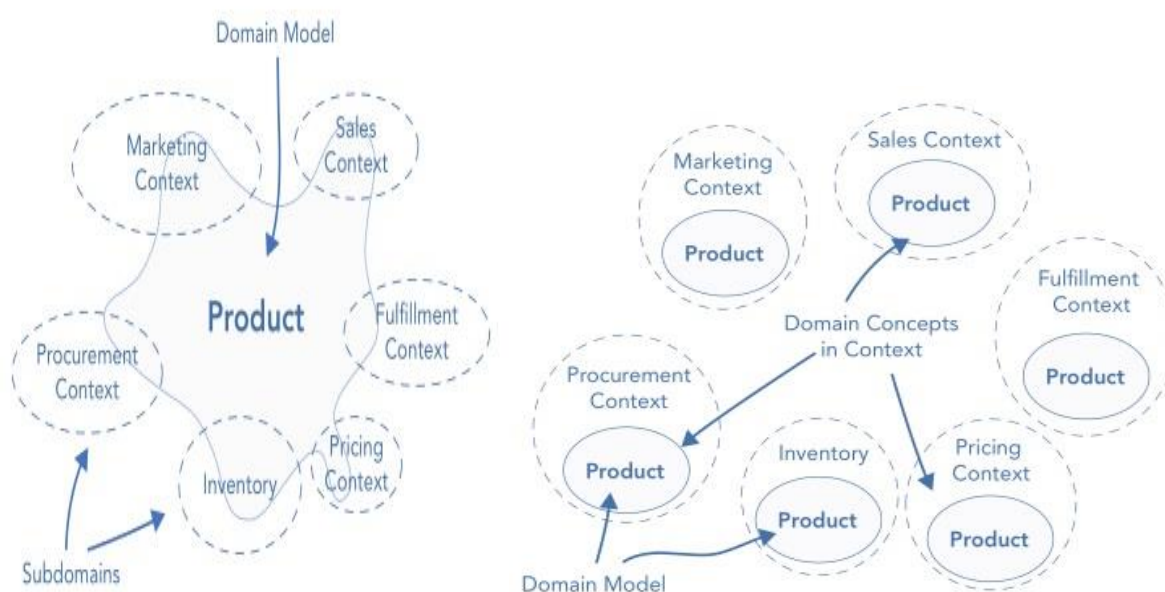
U literaturi [1] se također spominje anemični domenski model (engl. *Anemic domain model*) u kontekstu anti-uzorka. Na prvi pogled uzorak je jako sličan domenskom modelu jer stvara domenske objekte koji predstavljaju poslovnu domenu. Međutim, to su modeli koji nemaju unutar sebe nikakvo ponašanje. Ponašanje je implementirano izvan modela, predstavljajući domenske modele kao jednostavne podatkovne transfer klase. Glavni nedostatak je da domenski servisi preuzimaju ulogu ponašanja objekata te programiranje poprima svojstvo funkcionalnog, a ne objektnog. Na taj način klijent ne zna ponašanja objekata te sve što može od njega očekivati, te on mora utvrđivati da li je objekt u određenom stanju za određenu radnju. Anemični domenski princip je dobar kandidat za dijelove domenskog modela koji imaju malo logike ili za timove koji nisu iskusni sa objektno orijentiranim programiranjem. Kako je DDD princip podržan i za programere koji koriste funkcionalno programiranje, na taj način anemični model postane temelj funkcionalnog programiranja domene. Funkcionalno programiranje i anemični modeli osiguravaju da se koristi zajednički jezik domene kroz radnje odnosno funkcije. Subjekti domene su izraženi kroz anemične modele i na taj način kod još uvijek ističe sve bitne aspekte domene.

2.3 Kontekst domene i strateški principi

Kontekst je prema Evansu [1] okruženje u kojemu se riječ ili izjava nalazi te koje otkriva njihovo značenje. Svi principi do ove točke su opisivali DDD koji se temelji na jednom većem modelu. Veliki kompleksni modeli uključuju puno koncepata i poslovnih slučajeva korištenja. Kako sistem raste i u modelu se pojavljuje sve više koncepata i ovisnosti. Zbog toga ukoliko na projektu radi više timova, svi rade na istom modelu te jedan tim mora čekati drugog kako bi se napravila objava (engl. *deploy*) aplikacije.

Također se dešava da unutar jednog sustava postoji više koncepata koji u nekim slučajevima imaju isto ime, ali imaju različito značenje u različitim kontekstima aplikacije. Zbog toga je veliki model potrebno podijeliti na više manjih ograničenih konteksta (engl. *bounded contexts*) koji se razvijaju neovisno unutar svojih granica i imaju značenje unutar svojega konteksta. Na taj način se postiže da se zna točna i precizna odgovornost modela. Na slici 2.6. može se vidjeti različitost značenja termina proizvoda u različitim kontekstima. Nije dobro da isti koncept proizvoda dijele svi ograničeni konteksti jer bi dobili jednu veliku klasu koja bi imala više od jedne odgovornosti za što je namijenjena te bi održavanje koda bilo otežano.

Zbog toga se koncept proizvoda uspostavlja u svakom zasebnom kontekstu unutar kojeg ima specifično značenje.



Slika 2.6. Prikaz različitosti značenja u različitim kontekstima [3]

Ukoliko je sustav kompleksan i velik potrebno je prvo sustav podijeliti na više poddomena, a tek onda unutar svake poddomene raditi podjele na ograničene kontekste. Pri podjeli nije dobro raditi stroge podjele prema poslovanju firme, uvijek se treba bazirati prema domeni. Razlika između poddomene i ograničenog konteksta je u tome što poddomena predstavlja logičku podjelu poslovanja (organizacijsku strukturu) dok su ograničeni konteksti tehnička implementacija koja osigurava granice između modela unutar aplikacije. Svaki ograničeni kontekst posjeduje vlastitu arhitekturu koja u kontekstu DDD-a treba biti podijeljena na slojeve tako da domena bude izolirana.

2.3.1 Povezivanje ograničenih konteksta

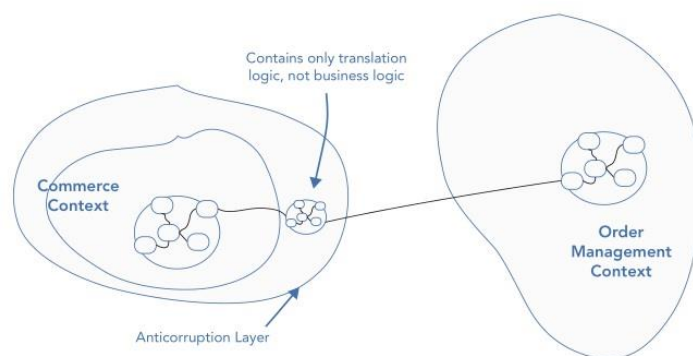
Više definiranih konteksta treba surađivati kako bi se ispunili poslovni ciljevi. Upravo zbog suradnje DDD uvodi mapu konteksta (engl. *context map*) koja pokazuje povezanost svakog ograničenog konteksta s drugima. Ona vizualizira stroge granice između različitih konteksta te pokazuje definirani način povezivanja. U većini slučajeva je određen tim zadužen za pojedini kontekst, pa tako tim mora surađivati sa ostalim timovima odnosno ostalim ograničenim kontekstima. Svaki programer unutar kompanije bi trebao poznavati mapu

konteksta. Pri tome programer ne treba poznavati tehničku implementaciju, već je bitno da poznaje samo način povezivanja svoga konteksta s drugima.

Načini povezivanja konteksta mogu biti ostvareni na više načina [4]:

- **Zajednički kanal** (engl. *Shared kernel*)
Ova vrsta je karakteristična kada dva ili više konteksta dijele zajednički kod. Pri tome svi timovi mogu mijenjati i razvijati zajednički kod kroz zajednički kanal, ali moraju paziti i surađivati sa ostalima.
- **Kupac-dobavljač** (engl. *Customer-supplier*)
Karakterističan je kada postoji jednosmjerna integracija između dva ograničena konteksta gdje se jedan ponaša kao dobavljač i drugi koji je kupac. Kupac tim predstavi svoje zahtjeve timu dobavljač koji te zahtjeve ispunjava.
- **Konformist**
Pojavljuje se kada Kupac-dobavljač odnos ne funkcionira odnosno dobavljač ne pomaže klijentu (ne izvršava njegove zahtjeve). Obično se javlja kada timovi nemaju blisku vezu što je slučaj kod integracije s vanjskim sustavima (engl. *third party systems*) gdje se domena i infrastruktura ograničenih konteksta u potpunosti razlikuju. Pri ovakvom odnosu treba se prilagoditi modelu dobavljača i čvrsto se držati za njega jer je sigurno da ga oni neće mijenjati u korist drugih.
- **Razdvajanje puteva** (engl. *Seperate ways*)
Ponekada se dešava da je integracija previše „skupa“ operacija da bi je se uspješno provelo. Pri tome programeri trebaju pronaći neki drugi način kako riješiti takvu situaciju. Situacija se obično rješava tako da je aplikacija napravljena od nekoliko manjih aplikacija koje nemaju ništa ili jako malo toga zajedničkog. Pa je tako s klijentske strane to jedna aplikacija, ali u pozadini ona se sastoji od više različito implementiranih modela. Tako programeri imaju slobodu da za svaki ograničeni kontekst koriste željenu tehnologiju implementacije. Tada su ograničeni konteksti potpuno neovisni, a povezuje (udružuje) ih jedino grafičko sučelje.
- **Open-host servis**
Kada jedan sustav koristi više drugih modela, da se ne bi kreirao za svaki model translacijski sloj, rješenje je da se vanjski sustav gleda kao skup servisa tako da svi mogu koristiti zajedničke servise.

U slučajevima kada se ovisi o starom zastarjelom kodu ili nekoj drugoj vanjskoj aplikaciji model vanjske aplikacije se ne uklapa u model aplikacije. Dobra praksa je tada izolirati svoj model kako bi se izbjegla korupcija i zaštitilo model od vanjskih utjecaja. Uspostavljamo antikorupcijski sloj (engl. *anticorruption layer*) između svog modela i vanjskog. Taj sloj koristi koncepte i akcije poznate našem modelu ali pri komunikaciji s vanjskim sustavom koristi njegov model, a ne naš. Taj sloj ima ulogu prevodioca između navedena dva modela i ne sadrži nikakvu poslovnu logiku. Sloj sadrži sučelja napisana u skladu s domenskim modelom konteksta. Možda bi ga bilo najbolje shvatiti kao granični prijelaz gdje ništa ne može ući u sustav dok ne prođe provjeru. Kada prođe provjeru tada se mora prilagoditi tome kontekstu odnosno jeziku i pravilima domene. Na slici 2.7. je prikazan primjer uspostave antikorupcijskog sloja konteksta trgovine koji komunicira s vanjskim kontekstom upravljanjem narudžbi.



Slika 2.7. Uspostavljanje antikorupcijskog sloja ograničenog konteksta [3]

Za implementaciju integracije ograničenih konteksta se najbolji pokazao međusloj poruka [2] (engl. *messaging middleware*). S tim pristupom mogu se stavljati poruke u red čekanja na koji se mogu zainteresirane strane pretplatiti i koristiti poruke. Na taj način se postiže slabo vezivanje između dva ili više konteksta. Da bi se konteksti mogli razumjeti poslano informacije potrebno je uspostaviti zajednički jezik komunikacije koji se zove objavljeni jezik (engl. *Published language*). Da bi postigli asinkronu obradu događaja koristimo Objavitelj-pretpatnik (engl. *Publish-subscribe*) uzorak. Pri tome objavitelj šalje poruku međusloju poruka sa informacijom o temi na koju se poruka odnosi. Pretplatnici se pretplaćuju na teme koje ga zanimaju. Međusloj predstavlja komunikacijsku infrastrukturu, odnosno posrednik je između objavitelja i pretplatnika, te šalje sve poruke pretplatnicima koji su se pretplatili na određenu temu.

3 TAKTIČKI PRINCIPI

Implementacija modela u kodu na osnovu modela analize je zahtjevan i kompleksan zadatak. Evans je uveo kolekciju principa koji osiguraju efikasnu kreaciju domenskog modela. To su taktički principi koji se temelje na najboljim praksama u objektno orijentiranom programiranju. Mnogi od tih principa nisu novi te su već odavno poznati ali ih je Evans grupirao i dao programerima na raspolaganje. Primjeri korištenih taktičkih principa se mogu pronaći u poglavlju 5. gdje je detaljno opisana web aplikacija za izdavanje računa.

3.1 Entitet

Entitet predstavlja koncept unutar domene koji je jedinstveno određen svojim identitetom, a ne atributima. Svaki entitet ima svoj životni ciklus te mu identitet cijelo vrijeme ostaje nepromijenjen, a jedino što mu se mijenja su njegova stanja. Kako bi spriječili izmjene identiteta entiteta potrebno je osigurati enkapsulaciju podataka kako bi podaci ostali nepoznati vanjskom svijetu. Entitet je jedan od najvažnijih taktičkih principa i građevnih elemenata implementacije domene, te pogreške u identifikaciji entiteta mogu imati jako teške posljedice u pogledu korupcije koja se reflektira kroz podatke. Da bi neki objekt iz stvarnog svijeta bio entitet, on mora biti jedinstveno identificiran nekom jedinstvenom oznakom što znači da se njegova apstraktna definicija klase te njezine odgovornosti, atributi i veze trebaju bazirati oko toga tko je objekt, a ne koje posebne attribute ima. Entiteti su u većini slučajeva povezani sa ostalim entitetima i vrijednosnim objektima (engl. *value objects*) kroz asocijacije te zbog toga ispunjavaju svoje odgovornosti kroz koordinirani rad objekata koje posjeduju. Odluka da li je nešto entitet treba se pažljivo donijeti kroz kontekst aplikacije unutar kojeg se nalazi. Za primjer možemo uzeti sjedala u busu koje treba modelirati ovisno o pojedinom slučaju. Sustav čine karta za autobus, autobus, sjedala. Sjedala u busu u gotovo svakom autobusu imaju svoj identitet koji je određen u većini slučajeva svojom numeričkom oznakom. Međutim kompanija koristi autobus za lokalni i međugradski prijevoz te su ta dva područja u poslovanju kompanije odvojena. Kod izdavanja karata za međugradski prijevoz striktno je naveden i broj dobivenog sjedala na koje bi trebao sjesti svaki putnik. Kod izdavanja karata za gradsku vožnju uopće nije naveden broj sjedala. To bi značilo da je sjedalo kod međugradске vožnje entitet, a kod gradske vožnje nije slučaj pošto nema potrebe za evidencijom sjedala, poznat je samo njihov ukupan broj. Ovo dovodi do zaključka da odluka da li je nešto entitet ili ne ovisi o kontekstu u kojem se nalazi.

3.2 Vrijednosni objekt

Kod modeliranja problematičnog domenskog područja postoje slučajevi kada radimo sa objektima za koje u kontekstu softvera nije bitan njihov identitet. Suprotno od entiteta koji su određeni svojim identitetom, postoje objekti koji su određeni svojim atributima a zovu se vrijednosni objekti. U prijašnjem slučaju kod sjedala za bus, ukoliko ne gledamo identitet sjedala već samo njegove attribute onda je takav objekt određen samo svojom vrijednosti.

Teoretski možemo svaki objekt napraviti kao entitet i pratiti njegov životni ciklus. U programiranju je potrebno raditi stvari samo onda ukoliko je stvarna potreba za tim budući da svaka promjena kakva god bila povećava kompleksnost softvera te utječe na performanse sustava. Ako bi se gledalo da je svaki objekt entitet tada se mora raditi uvijek s jednom određenom instancom objekta. Preporuka je da vrijednosni objekti budu nepromjenjivi, odnosno da ne mogu mijenjati svoja svojstva (engl. *immutable*). To se postiže jednim instanciranjem objekta kroz konstruktor, te se više ne mogu mijenjati vrijednosti za tu instancu kroz životni ciklus objekta što znači kada se zatraži nova vrijednost objekta treba se konstruirati novi. To osigurava da se vrijednosni objekti mogu dijeliti unutar programskog koda te slati kao argument u druge funkcije bez straha od njihove promjene i nastavka rada s neočekivanom vrijednosti objekta.

```
<?php

$today = new \DateTime('now');
echo "Danas je: ".date_format($today, 'Y-m-d').PHP_EOL;

$lastDateInMonth = calculateLastDateInMonth($today);

echo "Zadnji dan u mjesecu je: ".date_format($lastDateInMonth, 'Y-m-d').PHP_EOL;
echo "Danas je: ".date_format($today, 'Y-m-d');

function calculateLastDateInMonth(\DateTime $today)
{
    $lastDateInMonth = $today->modify('last day of this month');

    return $lastDateInMonth;
}
```

Result:

```
Danas je: 2017-09-14
Zadnji dan u mjesecu je: 2017-09-30
Danas je: 2017-09-30
```

Slika 3.1. Posljedice rada promjenjivog vrijednosnog objekta

Iz primjera na slici 3.1. vidimo kakve su posljedice ukoliko radimo sa jednom instancom objekta. Nakon poziva funkcije varijabla `today` se promijenila te nastavljamo s njom raditi misleći da radimo sa ispravnim datumom. Da se to ne bi događalo potrebno je osigurati da radimo sa nepromjenjivim vrijednosnim objektom. U kontekstu navedenog programa to bi bila klasa `DateTimeImmutable`. Na taj način u funkciji stvorila bi se nova varijabla prilikom njezine modifikacije.

3.3 Servis

U kontekstu programiranja servis je objekt koji obavlja neki globalni zadatak koji može biti primjerice slanje e-mailova, poruka, spremanje objekata u bazu i slične stvari. Potreba za servisima je nastala iz činjenice da je ponekada teško predstaviti koncepte i logiku unutar jedne klase, odnosno za donijeti neku odluku ili obaviti zadatak potrebno je više strana.

Postoji više vrsta servisa ovisno o korištenom mjestu unutar arhitekture aplikacije. Tako domenski servisi enkapsuliraju domensku logiku i koncepte koji nisu prirodno modelirani unutar pojedinačnih entiteta ili vrijednosnih objekata unutar domene. Domenski servisi nemaju identitet ili stanje nego su zaduženi za upravljanje poslovnom logikom koristeći entitete i vrijednosne objekte. Eric Evans u svojoj knjizi [1] kaže „Kada značajan proces ili operacija unutar domene nije prirodna odgovornost entiteta ili vrijednosnog objekta, dodaje se model kao zasebno sučelje definirano kao servis.“ Pri tome je bitno da ime servisa i logika bude u skladu uspostavljenog zajedničkog jezika. Implementacija servisa treba biti takva da servis ne prati podatke od jednog do drugog poziva, već se izlaz generira samo na osnovu ulaza ne uzimajući u obzir prijašnja stanja.

Za primjer servisa može se uzeti primjer prebacivanja novca s jednog računa na drugi. Kada imamo neki poslovni slučaj prvo na što treba obratiti pozornost čija je to odgovornost odnosno zadatak. U ovome navedenom slučaju to nije zadatak ni prvog ni drugog računa, već je to zadatak nečega globalnog a to je servis. Kako je prebacivanje s računa na račun osjetljivo područje i predstavlja domenu tvrtke radi se o domenskom servisu. Pri tome ime servisa mora iskazivati namjeru i biti u skladu s zajedničkim jezikom. Može se reći da servis predstavlja točku uspostave konekcije za mnoge objekte.

3.4 Modul

Kod velikih i jako kompleksnih aplikacija model unutar određenog područja aplikacije ima tendenciju da bude sve veći i veći. Model dostigne točku kada je jako teško povezati sve međusobne interakcije između različitih dijelova. Zbog toga se javlja potreba za grupiranjem povezanih klasa u module kako bi se postigla što veća kohezija među njima.

Postoje dva tipa kohezije:

- Komunikacijska koja se postiže kada dijelovi modula koriste iste podatke pa zbog toga postoji veza među njima.
- Funkcionalna koja se postiže kada svi dijelovi modula rade zajedno kako bi izvršili definirani zadatak.

Moduli ne odvajaju kod već samo koncepte unutar domene u definiranom kontekstu. Također trebaju imati dobro definirana sučelja koja su dostupna iz drugih modula. Sučelja sprječavaju čvrsto vezivanje (engl. *tight coupling*) što u praksi dovodi do smanjenja kompleksnosti i lakšeg održavanja koda. Na taj način je puno lakše programerima razumjeti kako sistem funkcionira. Potrebno je kao i za druge principe pridijeliti ispravne nazive kako bi bili u duhu zajedničkog jezika. Što se tiče grupiranja klasa u module mnogi jezici pružaju neki oblik alata koji omogućava grupiranje klasa i jezičnih definicija zajedno pa tako primjerice programski jezik Java ima pakete, Ruby ima module dok skriptni programski jezik PHP koji je i tema ovoga rada ima imenske prostore (engl. *namespaces*).

3.5 Agregat

Svaki entitet i vrijednosni objekt ima svoj životni ciklus. Entitet se stvori te prolazi u svom ciklusu kroz niz faza pri kojima dolazi do promjena u atributima te na kraju „umire“. Agregat je princip koji je upravo vezan za životni ciklus domenskih objekata. To je princip koji se koristi za definiranje vlasništva entiteta te njegovih granica.

Može se vidjeti iz prijašnjih poglavlja da model može sadržavati jako puno objekata i biti jako kompleksan gdje su svi objekti međusobno povezani zbog čega se je jako teško nositi sa svim promjenama kod tako razgranatih i kompleksnih relacija te garantirati konzistentnost objekata. Kao što moduli odvajaju koncepte unutar domene, agregati udružuju međusobno povezane objekte u jednu jedinicu koja se smatra kao jedna cjelina. Agregat ima glavni

korijenski čvor koji je entitet te on razdvaja unutarnje objekte od okoline. Pri tome je jedino korijenski čvor dostupan izvana te on vrši svu kontrolu i predstavlja vezu pri komunikaciji s internom strukturom. To znači da vanjski objekti ne mogu direktno mijenjati objekte unutar agregata, nego mogu komunicirati samo s korijenskim objektom te ga pitati za promjenom drugih objekata. Kada su bilo koje promjene napravljene na korijenskom objektu transakcija osigurava da se naprave i na povezanim objektima. Također ako korijenski čvor bude izbrisan iz memorije svi objekti unutar njega se također automatski brišu.

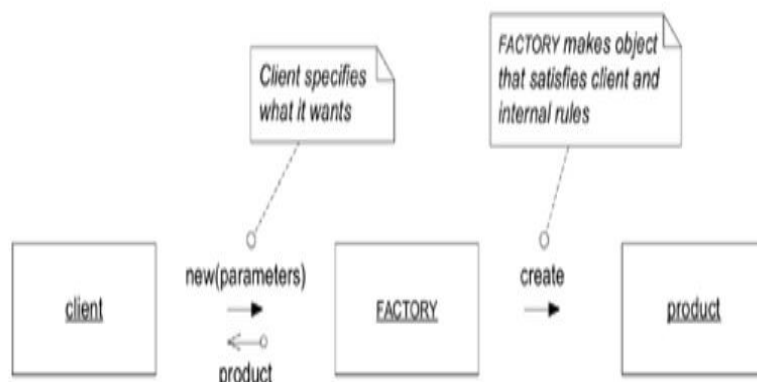
Da bi uspješno implementirali agregate postoji nekoliko pravila [1]:

- Korijenski entitet ima globalni identitet i odgovoran je za provjeru svih procesa koji se moraju izvršiti.
- Entiteti unutar korijenskog imaju lokalni identitet jedinstven samo unutar granica agregata.
- Ništa izvan granica agregata ne smije imati referencu na ništa unutar agregata nego samo na korijenski entitet. Korijenski čvor može pružiti referencu unutarnjeg entiteta drugim objektima ali samo prolazno (engl. *transient*) odnosno ukoliko objekt nije u spremljenom stanju nego samo u prolaznom (nije spremljen još u bazu). Dobra je praksa da korijenski entitet ukoliko baš mora pružiti unutarnju strukturu drugim objektima da to bude putem vrijednosnih objekata jer njihova promjena neće utjecati unutar agregata (nije bitno što će se s njime dogoditi).
- Objekti unutar agregata mogu imati referencu prema drugim agregatima.
- Kada se dogodi promjena unutar granica agregata, svi poslovni procesi cijelog agregata moraju biti zadovoljeni (engl. *invariants*).

3.6 Factory

Kada kreacija cijelog objekta ili agregata postane kompleksna ili otkriva previše unutrašnje strukture koristi se factory kako bi odvojio potrebno znanje za kreiranje kompleksnog objekta. Factory [4] obično pruža sučelje koje reflektira ciljeve klijenta. Odgovornost za kreiranje factory-a imaju posebni objekti koji sami po sebi nemaju nikakvih odgovornosti u domenskom modelu, ali su ipak dio domene. Na slici 3.2. vidljiv je način rada factory-a koji prima parametre potrebne za kreaciju objekta te prema njima i poslovnim pravilima instancira objekt i vraća ga klijentu. Uvijek je dobra praksa objekte konstruirati preko konstruktura ili ako je objekt velik i kompliciran što su u pravilu agregati, tada koristiti factory-e. Na taj način

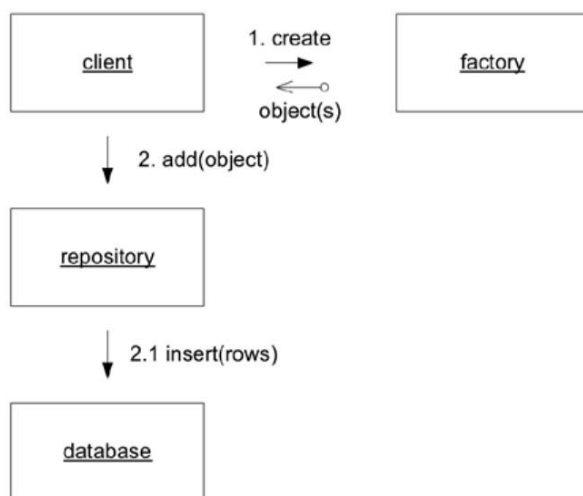
osiguravamo da se objekt kreira u potpunosti ili nikako odnosno da se dojadi greška ukoliko bilo koji dio objekta nije uspješno kreiran.



Slika 3.2. Princip rada factory-a [1]

3.7 Repozitorij

Repozitoriji se koriste za upravljanje pohranom i dohvatom objekata iz baze osiguravajući da postoji odvajanje između domenskog modela i modela podataka, odnosno ponašaju se kao spremište između klijenta i baze kao na slici 3.3. Oni sakrivaju tehnologiju potrebnu za pohranu i dohvat podataka. U kontekstu DDD-a sučelje repozitorija se nalazi u domenskom modelu jer je i sam dio domenskog modela sa implementacijom koja se nalazi u infrastrukturnom sloju. Na taj način se osigurava domenskim objektima da se baziraju samo na poslovne koncepte i logiku, odnosno da se ne zamaraju o korištenoj tehnologiji za pohranu i dohvat.



Slika 3.3. Uloga repozitorija u sustavu [1]

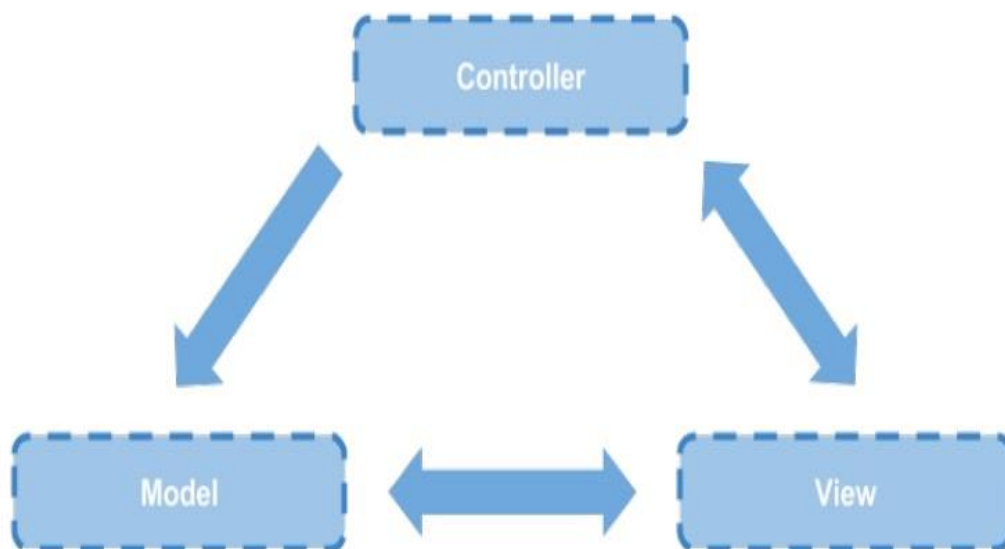
4 ARHITEKTURE APLIKACIJA

Izrada svake aplikacije je priča se sebe te zahtjeva stil arhitekture koji zadovoljava aplikacijske potrebe. DDD princip je skroz fleksibilan jer nije vezan za niti jedan arhitekturni stil, bitno je samo da su tehničke stvari odvojene od domenskih. Programeri su slobodni odabrati arhitekturu koja je najprikladnija za aplikaciju što vrijedi i za svaki ograničeni kontekst. Arhitektura je stvar arhitekta softvera ili samog programera pa tako kao što svaka kuća ili zgrada zahtjeva svoju arhitekturu tako je i sa softverom. Za neke će se kuće ili zgrade zahtijevati jednostavnije, dok će za druge arhitekture biti unikatne te kompliciranijeg dizajna. Danas je svakako najpopularniji MVC (engl. *model-view-controller*) uzorak koji dolazi implementiran putem nekog od radnog okruženja (engl. *framework*) kao što je primjerice Symfony ili Laravel napisan u PHP-u.

Dok nije bilo razdvajanje sadržaja (engl. *seperation of concerns*) softver bi se izrađivao tzv. „Big Ball Of Mud“ pristupom [2]. Taj pristup je prepoznatljiv po slučajno strukturiranom, neurednom, isprepletenom kodu koji radi nešto korisno ali nije vidljivo kako, odnosno nema naglaska na poslovne procese. Taj kod u žargonu se naziva i „špageti kod“. Kod takvog pristupa čak i male promjene funkcionalnosti postaju jako teške za implementaciju pošto dolazi do miješanja kompleksnosti domene s tehničkim aspektima softvera. Taj pristup nije uočljiv toliko u početnoj fazi projekta, ali s razvojem i životnim vijekom softvera postane jako teško s njime upravljati, pa se tako pojave potrebe za izmjenom domenske logike koja postane jako teška za implementaciju. Time se gubi puno vremena na bitnim funkcionalnostima za poslovanje kompanije. Čak i ako se stave na raspolaganje dodatni resursi (ljudi, oprema) ne dolazi do izraženije razlike. Time je životni vijek takvog softvera jako kratak pošto će se u jednom trenutku kroz analizu odlučiti da je bolje napraviti softver ispočetka, ali s puno više pažnje posvećene domenskom modelu te implementaciji koda kroz razdvajanje sadržaja.

4.1 MVC

MVC je jedan od trenutno najpoznatijih uzoraka za povezivanje korisničkog sučelja prema aplikacijskom i domenskom sloju.



Slika 4.1. Prikaz MVC uzorka [2]

Dijeli aplikaciju u tri glavna sloja kao na slici 4.1.:

- Model koji obuhvaća i centralizira domenski model, te su tu definirana sva poslovna pravila.
- Kontroler koji upravlja sa drugim slojevima, okidač je za akcije u modelu koji mijenja svoje stanje. Kontroler može također slati podatke sloju pogleda koji može izmijeniti reprezentaciju modela. U kontekstu web aplikacija kontroleri prime HTTP zahtjev i vraćaju HTTP odgovor koji može biti bilo koji oblik podataka (HTML, JSON, tekst, slika itd.).
- Pogled (engl. *view*) koji predstavlja reprezentaciju podataka, te također mijenja stanja modela.

Treba naglasiti da je MVC po svojoj prirodi uzorak te nema veze sa slojevitom arhitekturom koja striktno radi razdvajanje aplikacije na slojeve. Kod MVC-a je razdvajanje sadržaja aplikacije ipak čvrsto povezano u obliku trokuta gdje sve strane izravno komuniciraju i mijenjaju jedan drugog. Slojevitost arhitekture je pak prava linearna arhitektura gdje sve mora prolaziti kroz jedan međusloj (engl. *middleware*). Problem MVC-a proizlazi iz činjenice da

se u model guraju stvari koje ne bi trebale biti dio modela, a to su najčešće različite infrastrukturne stvari poput primjerice baze. Model ne bi trebao imati veze sa implementacijom spremanja i čitanja iz baze. Tako se prekrše pravila razdvajanja sadržaja aplikacije.

4.2 Slojevita arhitektura

Slojevita arhitektura (engl. *layered architecture*) je trenutno najpoznatija arhitektura korištena kod kompleksnijih aplikacija gdje se postiže strogo odvajanje slojeva gdje su slojevi slabo povezani (engl. *loose coupling*). Ona je neovisna o radnom okruženju te je u većini slučajeva stvar izbora arhitekta softvera ili programera koji treba procijeniti koji tip slojevite arhitekture koristiti. Autor DDD-a Eric Evans u svojoj knjizi [1] detaljno objašnjava svoju verziju slojevite arhitekture gdje se najviše bazira na domenski sloj.

Posljednjih godina pojavilo se je mnogo slojevitih arhitektura aplikacija poput:

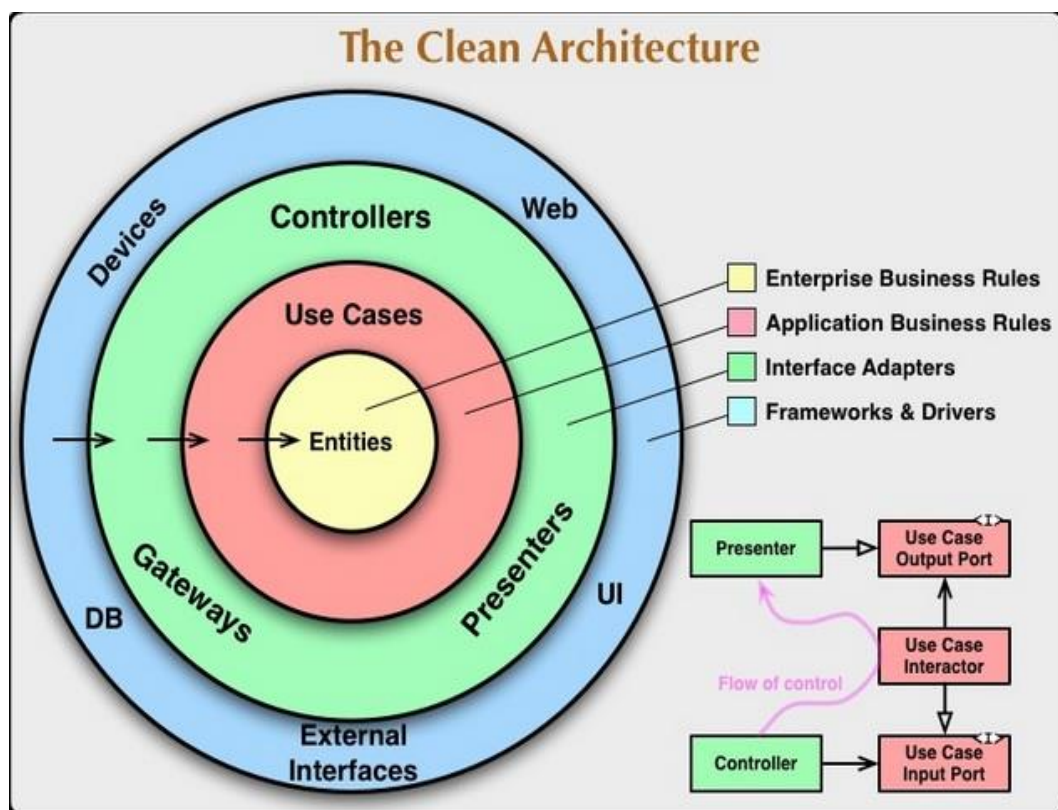
- „Clean architecture“ koju detaljno objašnjava Robert C. Martin [5]
- „Hexagonal architecture“ koju detaljno objašnjava Alistair Cockburn u svojoj knjizi „Growing Object Oriented Software“
- „Onion architecture“ čiji je autor Jeffrey Palermo

Iako se sve ove arhitekture razlikuju u nekim detaljima, jako su slične i dijele zajedničke značajke razdvajanja aspekata funkcionalnosti softvera koju postižu kroz razdvajanje softvera na slojeve.

Cilj svake slojevite arhitekture je u osiguranju sljedećih stavki [5]:

- Neovisan o radnom okruženju što znači da nam radno okruženje služi samo kao alat. U suprotnom bi ovisili pa bi ograniči svoj sustav kroz ograničenja radnog okruženja.
- Lagan za testiranje što znači da se poslovna logika može bez problema testirati bez tehničkih detalja kao što je korisničko sučelje, baza podataka ili bilo kakve druge stvari.
- Neovisan o korisničkom sučelju što znači da se sučelje može lako mijenjati bez mijenjanja ostatka sustava. Primjerice grafičko sučelje se može zamijeniti komandnim bez mijenjanja ostatka sustava.

- Neovisan o bazi podataka što znači da možemo lako promijeniti primjerice SQL Server s MySQL-om. Unutar domenskog sloja poslovna logika nije niti na koji način spojena s bazom.
- Neovisan o bilo kakvom vanjskom djelovanju. Tako domenski sloj koji je srce aplikacije ne smije znati ništa o vanjskim slojevima odnosno vanjski svijet mu je nepoznat. Ukoliko se ipak treba pozvati nešto iz drugih slojeva, postoje tehnike kojima se i to postiže koje će biti spomenute u nastavku rada.



Slika 4.2. Prikaz slojevite arhitekture [5]

Koncentrični krugovi na slici 4.2. predstavljaju različite slojeve softvera. Vanjski krugovi su više mehanizmi, dok prema unutrašnjosti ide sve više i više poslovnih pravila. Ovisnost izvornog koda može ići samo prema unutra, te ništa unutar unutarnjeg kruga ne smije znati ništa o stvarima u vanjskom krugu. To konkretno znači da se ime nečega definiranog u vanjskom krugu ne smije spominjati u unutrašnjem krugu. Isto tako formati podataka definirani u vanjskom krugu ne bi trebali biti korišteni u unutarnjem krugu, posebno ako su formati generirani putem nekog od radnog okruženja jer se ne želi ovisiti o njima. Na slici sloj poslovnih pravila odnosno entiteta predstavlja srce aplikacije te taj sloj ne smije ovisiti niti o

jednom sloju. Sloj slučajeva korištenja sadrži aplikacijska pravila. On enkapsulira i implementira sve slučajeve korištenja sistema. Ti slučajevi upravljaju tokom podataka prema i od entiteta i usmjeravaju entitete da koriste poslovna pravila specifična za poslovanje kako bi zadovoljili slučaj korištenja. Ovaj sloj ne smije ovisiti o promjenama u radnom okruženju ili bazi. Adapteri sučelja (engl. *Interface Adapters*) pretvaraju podatke iz jednog formata u drugi dvosmjerno.

Često se dešava da unutarnji sloj mora na neki način pozvati vanjski. Takav slučaj se rješava inverzijom ovisnosti (engl. *Dependency Inversion*, skraćeno DIP) na način da unutarnji sloj koristi sučelje (engl. *interface*), a vanjski sloj ga implementira. Princip se odnosi na razdvajanje modula tako da se ovisnost preokrene.

DIP princip ima svoja pravila [5]:

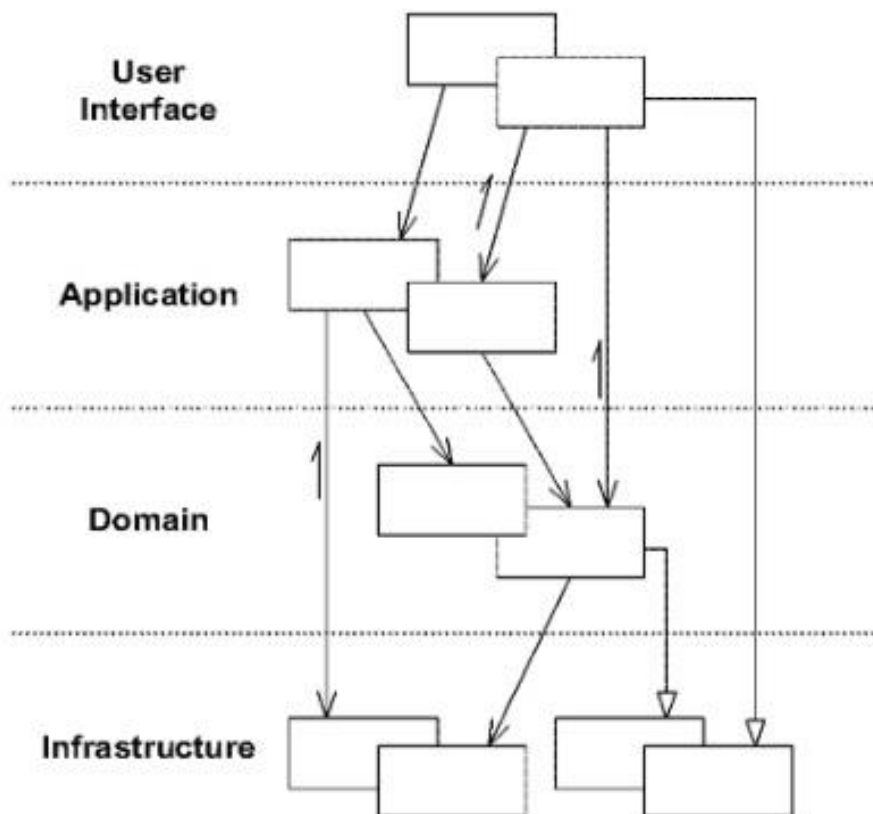
- Sloj većeg nivoa ne bi trebao ovisiti o sloju nižeg nivoa. Oba dva bi trebala ovisiti o apstrakcijama
- Apstrakcije ne bi trebale ovisiti o detaljima, dok bi detalji trebali ovisiti o apstrakcijama.

Slojevitost arhitektura osigurava da bilo koji element određenog sloja ovisi samo o elementima na istom sloju i elementima na nižem sloju. Svaki sloj je specijaliziran za određenu ulogu u programu. Iako ima mnogo varijacija unutar slojevitost arhitekture, mnoge poznate arhitekture koriste neku verziju od sljedećih konceptualnih slojeva:

- Korisničko sučelje (engl. *user interface*, skraćeno UI) naziva se još i prezentacijski sloj. Odgovoran je za prikaz informacija korisniku i interpretaciju korisnikovih naredbi. Vanjski korisnik može osim čovjeka biti i računalni sistem.
- Aplikacijski sloj (engl. *application layer*) definira zadatke koje sistem treba napraviti i usmjerava objekte domena kako bi riješili problem odnosno slučaj korištenja. Zadatci za koje je taj sloj odgovoran su značajni za poslovanje ili su potrebni za interakciju s aplikacijskim slojevima drugih sustava. Važno je da ovaj sloj bude tanak i ne sadrži poslovna pravila ili znanje domene. Može imati stanje koje reflektira napredak zadatka za korisnika sustava, ali ne smije sadržavati stanje koje reflektira poslovnu situaciju. On je klijent domenskog sloja te samo zna koga pozvati kako bi ispunio svoj posao. Može se reći da on zna što domenski sloj sve radi, ali ne zna način na koji radi. Po svojoj prirodi je proceduralan u obliku funkcija.

- Domenski sloj (engl. *domain layer*) odgovoran je za presliku koncepata poslovanja te sadrži poslovna pravila i koncepte. Temelji se na modeliranju objekata i predstavlja srce svakog sustava.
- Infrastrukturni sloj (engl. *infrastructure layer*) pruža tehničke stvari za više slojeve kao što su slanje poruka, spremanje domenskih objekata.

Neke slojevite arhitekture ne dozvoljavaju preskakanje slojeva, te su dosta striktne. Međutim postoje i slojevite arhitekture koje preskakanje slojeva dozvoljavaju i dosta su fleksibilne ovisno o aplikacijskim potrebama. Tako primjerice Eric Evans koristi u svojoj knjizi koristi slojevitom arhitekturu prikazanu na slici 4.3. [1]. Iako postoje razlike u arhitekturama, svima je zajedničko odvajanje domenskog sloja što predstavlja temelj principa DDD.



Slika 4.3. Prikaz slojevite arhitekture od Eric Evansa [1]

5 WEB APLIKACIJA ZA IZDAVANJE RAČUNA

U sklopu ovoga diplomskog rada napravljena je web aplikacija „eRačun“ za izdavanje računa čije je sučelje prikazano na slici 5.1. To je jedna manja poslovna aplikacija koja omogućava izdavanje računa specificiranih po zakonu Republike Hrvatske. Poslovne aplikacije su po svojoj prirodi većinom dosta kompleksne pri izradi, budući da su bogate poslovnim znanjem, sadrže dosta poslovnih odluka i moraju biti u skladu sa zakonom. Također su osjetljive za implementaciju budući da se radi s novcem te posljedice mogu biti velike ukoliko obračuni ne rade kako treba. Trenutno na tržištu postoji dosta aplikacija ovoga tipa koje omogućavaju efikasno i jednostavno izdavanje računa bez ikakve papirologije. Priložena aplikacija ima sve potrebne funkcionalnosti izdavanja računa, s tim da joj nedostaje funkcionalnost fiskalizacije za koju je potrebno imati testni certifikat kako bi se mogle digitalno potpisivati poruke.

The image shows two screenshots of the 'eRačun' web application. The top screenshot displays the main dashboard with a blue header containing the application name 'eRačun', navigation links 'Administracija' and 'Računi', and a user profile 'ivan@gmail.com' with a 'Odjavi se' button. The dashboard content includes a status check section, a 'Kupci i artikli' section with buttons for managing customers and articles, and a 'Podaci o tvrtki' section with buttons for managing company data, business locations, payment devices, and operators. The bottom screenshot shows the 'Izdani računi' (Issued Invoices) page, which features a search bar and a 'Unesi novog' button. Below these is a table listing three invoices with their details and action buttons.

Redni broj	Broj računa	Datum izdavanja	Ukupni iznos	Operator	Način plaćanja	Status	Opcije
1	1-POS1-1	09/09/2017	141.80 HRK	Ivan Brkovic	Gotovina	IZDAN	Storniraj Pogledaj
2	2-POS1-1	09/09/2017	44.40 HRK	Ivan Brkovic	Gotovina	STORNIRAN	Storniraj Pogledaj
3	3-POS1-1	09/09/2017	-44.40 HRK	NOVI	Gotovina	STORNO	Storniraj Pogledaj

Slika 5.1. Sučelje web aplikacije „eRačun“

Kroz navedenu aplikaciju biti će proveden u praksi dio DDD principa. Biti će pokazano kako je išao razvoj aplikacije, od razgovora s domenskim stručnjakom koji ima detaljno znanje o problematičnoj domeni izdavanja računa, do stvaranja modela sustava koji je zapisan kroz razne UML dijagrame te specifikacije. Nakon toga će ići implementacija modela sustava u model koda gdje će se koristiti taktički principi DDD-a za apstrakciju problematične domene. Budući da je ova aplikacija manja i obuhvaća domenu koja nema poddijele na poddomene te se sastoji od jednog ograničenog konteksta, strateški principi neće biti navedeni u kontekstu aplikacije. Nakon realizacije domenskog modela te njegovog odvajanja od ostatka aplikacije što je temeljni preduvjet DDD-a, grade se slojevi koji ne sadrže poslovnu logiku aplikacije ali su također bitni faktor aplikacije, a to su: aplikacijski, infrastrukturni i prezentacijski sloj.

5.1 Uspostava zajedničkog jezika

Općenito govoreći velika većina programera ima jako malo znanja o problematičnoj domeni, osim ako nisu već radili na projektima na kojima su mogli steći dovoljno znanja. Problematična domena o kojoj je ovdje riječ nije toliko kompleksna pa programeri mogu steći dovoljno znanja čitajući sve potrebne zakone te kroz otkrivanje problematike domene putem Interneta, ali naravno da će potrošiti više vremena nego kroz direktan razgovor s domenskim stručnjakom koji poznaje sve zakone te zamke koje domena krije. Moguće je također da će programeri neke stvari krivo zaključiti i pretpostaviti pa će se izrada softvera znatno produžiti. Zbog toga će se ovdje prikazati kako se može kroz razgovor s domenskim stručnjakom vrlo lako uspostaviti zajednički jezik te uspostaviti model sustava. Osoba koja ima dovoljno znanja o navedenoj specifičnoj domeni je zasigurno računovođa ili porezni savjetnik koji poznaje sve potrebne zakone države, te nam zna ukazati na bitne stvari i moguće probleme. Programer iz razgovora s računovođom zna da treba napraviti web aplikaciju za izdavanje računa koja mora biti po zakonima RH. Na početku je kod programera puno nepoznanica kako bi aplikaciju mogao napraviti, te već počinje kroz apstrakciju raditi model u svojoj glavi. Međutim još uvijek mu dosta stvari nije jasno pa ide popričati s domenskim stručnjakom. Ovdje će biti prikazan tijek razgovora programera i domenskog stručnjaka.

Programer: „Trenutno izdajete svaki račun ručno, pa želite na neki automatizirati izdavanje računa?“

Računovođa: „Tako je, trenutno nemamo podršku automatiziranog slanja fiskalnih računa, svaki račun izdajemo pojedinačno te nam to znatno usporava poslovanje. Trošimo puno

nepotrebnog vremena. Zbog toga bi napraviti web aplikaciju koja omogućava automatizirano izdavanje fiskalnih računa da kroz par klikova na jednostavan način izdamo račun. Aplikacija bi trebala pamtit i sve unesene kupce, artikle, poslovne prostore, operatere, naplatne uređaje te podatke naše firme.“

Programer: „To je dosta podataka za prilično jednostavnu funkcionalnost izdavanja računa. Znači koliko sam primijetio željeli bi ste moći unaprijed unijeti sve navedene podatke koje ste spomenuli tako da kod izdavanja računa možete samo odabrati sve navedene stavke, unijeti možda količinu i primjerice popust artikla i izdati račun?“

Računovoda: „Tako je, ne želimo svaki puta raditi dosadan posao unošenja identičnih podataka o firmi, te o kupcima i artiklima koji se većinom ponavljaju. To čak i nije toliko problem, najveći je problem što uz te sve podatke moramo raditi i kalkulaciju računa te izračunati broj računa za taj račun“

Programer: „Dobro, što se tiče artikala i kupaca to mi je sve jasno. Kalkulacija mi je isto jasna, svaki račun mora imati osnovicu, obračun poreza te ukupan iznos. Ali što je sa ovim brojem računa, to vidim na svakom računu. Kako se to definira?“

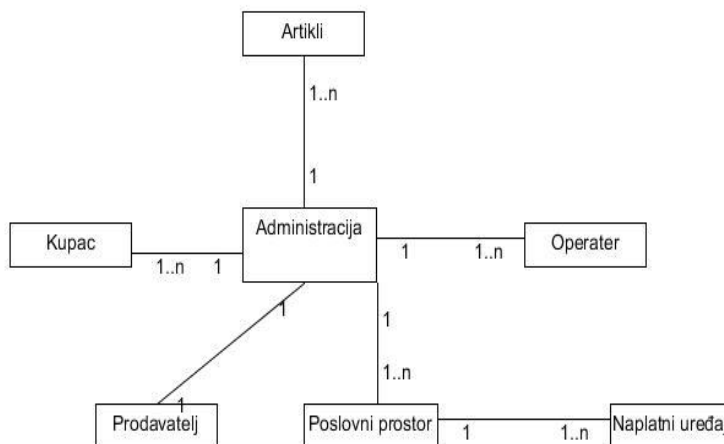
Računovoda: „Broj računa mora imati slijednost po broju naplatnog uređaja unutar poslovnog prostora, što znači da se ne smije dogoditi da dva računa izdana od istog naplatnog uređaja imaju isti redni broj. Znači format računa je u obliku X|Y|Z gdje je X redni broj, Y oznaka poslovnog prostora, dok je Z oznaka naplatnog uređaja. To bi u praksi značilo da ukoliko imate račun oblika 1|1|2 te idete izdati novi račun na istom naplatnom uređaju novi račun će imati oblik 2|1|2.“

Programer: „U redu, razumio sam. Ako možete reći samo još u kakvoj su vezi poslovni prostor i naplatni uređaj. Da li su naplatni uređaji na razini cijele firme, ili na razini poslovnog prostora?“

Računovoda: „To vam je na razini poslovnog prostora te svaki naplatni uređaj unutar poslovnog prostora mora biti jedinstven. Još vam moram napomenuti da ne bi bilo zabune, račun kada je jednom izdan više se ne smije mijenjati niti brisati. Moramo ga uvijek imati pohranjenog u stanju kakav je izdan.“

Programer: „E super što ste to rekli, onda ćemo napraviti da imate modul administracije gdje će te unositi sve potrebne podatke koji vam trebaju. Još ćete imati modul računa gdje će te izdavati račune koristeći te unesene podatke iz administracije. Da bi pristupili modulu računa morati ćete unaprijed unijeti sve potrebne podatke, inače vam program neće dopustiti pristup. Kako bi bio siguran da smo se dobro razumjeli ovdje vam prikazujem grafički dijagram

administracije“. Grafički dijagram može se vidjeti na slici 5.2. Tu programer objasni što znače veze u relacijama kako bi se on i računovođa uspješno razumjeli.



Slika 5.2. Predloženi model administracije

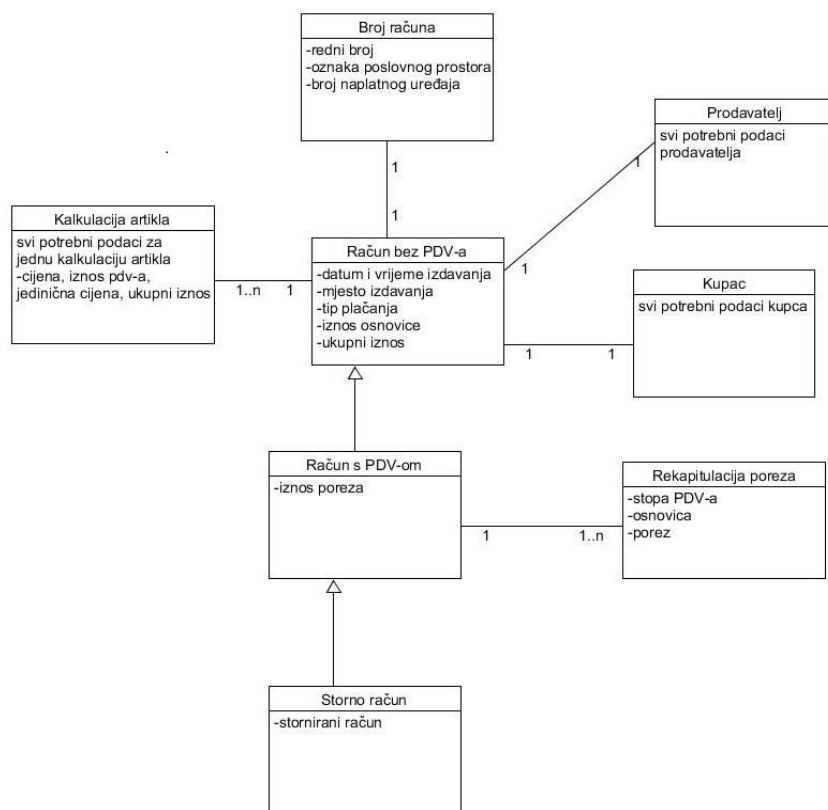
Računovođa: „Onda smo se sve razumjeli što se tiče administracije. Što se tiče računa, još bi napomenuo kako bi trebali omogućiti da se može postaviti da li je firma obveznik PDV-a. To vam znači ako firma nije obveznik PDV-a tada na računu nema obračuna PDV-a već samo ukupan iznos i osnovica koji su jednaki. Ukoliko je obveznik PDV-a tada je potrebno obračunati kalkulaciju računa da ima ukupan iznos PDV-a. Također je potrebno obračunati rekapitulaciju poreza. Rekapitulacija poreza je obračun poreza za svaku vrstu PDV-a koja u Hrvatskoj može biti 5, 13 te 25 posto. To sve specificira zakon RH te se toga moramo striktno držati.“

Programer: „Mislim da sam sve razumio što se tiče računa te koji su sve obvezni elementi. Da li postoje još neke funkcionalnosti koje bi željeli imati?“

Računovođa: „Treba biti moguće pregledati listu svih računa, te također pojedinačno. Također treba biti moguće stornirati račune. To je račun koji je jednak originalu ali s negativnim iznosima, podacima trenutnog operatera te s novim generiranim brojem računa. Također jednom stornirani račun se više ne može stornirati.“

Programer: „Koliko sam kroz razgovor primijetio, dakle imamo dvije vrste računa: s PDV-om te bez PDV-a. Ovisno o postavljenom izboru da li je prodavatelj u sustavu PDV-a, generiraju se računi sa svojim kalkulacijama. Također je za račune s PDV-om potrebno napraviti rekapitulaciju poreza tako da se točno zna koliki je iznos poreza za određeni PDV koji može biti 5, 13, te 25 posto. Ja ću sada napraviti grafički dijagram da bi se mogli bolje

međusobno shvatiti“. Grafički dijagram se može vidjeti na slici 5.3. Programer detaljno objasni domenskom stručnjaku kako je organizirao model te mu detaljno objasni sve što ga zanima. Ovo je nepotpuni model, ali je koristan u ovoj fazi razgovora. Tek će se nakon specifikacije zahtjeva napraviti potpuni model kojeg će programer napraviti u suradnji s domenskim stručnjakom. Domenski stručnjak će naučiti dovoljno o modeliranju da bi mogao sudjelovati u završnoj fazi modeliranja.



Slika 5.3. Predloženi model računa

Računovođa: „Mislim da je sve dobro napravljeno ali biti ćemo svakako još u kontaktu.“

Iz priloženog razgovora možemo vidjeti kako programer i domenski stručnjak međusobno razmjenjuju znanja na način da programer usvaja poslovna znanja, dok domenski stručnjak stekne znanja u modeliranju. Razgovor je bogat domenskim znanjem te od tu možemo jako puno korisnih informacija izvući kako bi napravili dobar model sustava. Razgovor je dodirna točka između programera i domenskog stručnjaka te tu dolazi do uspostave zajedničkog jezika koji je temelj principa DDD. Pri uspostavi modela je potrebno ignorirati nepotrebne informacije koje nisu od značaja za implementaciju. Iz razgovora s domenskim stručnjakom možemo generirati specifikaciju zahtjeva koja sadrži sve usluge koje korisnik zahtijeva od sustava te ograničenja pod kojima sustav mora raditi.

5.2 Specifikacija zahtjeva

5.2.1 Opseg aplikacije

Osnovna namjena web aplikacije „eRačun“ je omogućiti svim obveznicima fiskalizacije izdavanje računa na brz i jednostavan način. Obveznicima fiskalizacije treba omogućiti upravljanje podacima o artiklima, kupcima, poslovnim prostorima, naplatnim uređajima te operaterima naplatnih uređaja. Također im treba omogućiti izdavanje računa korištenjem prethodno unesenih podataka. Bitno je za naglasiti da sadržaj računa mora biti u skladu s zakonima Republike Hrvatske a to su:

- Opći porezni zakon [10]
- Zakon o porezu na dodanu vrijednost [11]
- Pravilnik o porezu na dohodak [12]
- Zakon o fiskalizaciji u prometu gotovinom [13]

5.2.2 Pregled sustava

eRačun web aplikacija za izdavanje računa sadrži dva odvojena modula a to su:

1. Administracija

U navedenom modulu korisnik može uređivati sve podatke o:

- Tvrtki
- Kupcima
- Artiklima
- Naplatnim uređajima
- Poslovnim prostorima
- Operaterima naplatnih uređaja

2. Računi

U navedenom modulu korisnik može upravljati izdavanjem te storniranjem računa.

Faktori kvalitete aplikacije su:

1. Jednostavnost

Nema potrebe za edukacijom korisnika, treba pružiti korisniku jasno i intuitivno sučelje koje jasno i nedvosmisleno ističe sve moguće radnje.

2. Dostupnost

Računima se može pristupiti sa bilo kojeg mjesta u svijetu, potrebna je samo Internet konekcija i neki od preglednika novije generacije.

3. Široka namjena

Aplikaciju trebaju moći koristiti razno razne djelatnosti, obrti, slobodna zanimanja te svi ostali obveznici fiskalizacije.

Što se tiče klasa korisnika podržane su dvije klase a to su:

1. Posjetitelj

Posjetitelj je akter koji nije prošao postupak prijave u sustav, te mu nije odobren rad u aplikaciji dok ne prođe registraciju u sustav. Posjetitelj može pristupiti samo prijavi i registraciji u sustav.

2. Korisnik

Korisnik je akter koji je prošao postupak registracije te nakon toga je dobio prava pristupa aplikaciji te ima pristup svim modulima aplikacije.

Aplikaciju je potrebno implementirati u PHP programskom jeziku koristeći Symfony radno okruženje. Za početak je aplikaciju moguće razvijati lokalno odabirom neke od distribucija kao što su XAMPP, Wamp i ostale.

5.2.3 Funkcionalni zahtjevi

Ispod su navedeni svi funkcionalni zahtjevi koji opisuju slučajeve korištenja sustava.

1. Registracija u sustav

Potrebno je omogućiti mogućnost registracije u sustav na način da se korisnik identificira svojim e-mailom te lozinkom. Ne smije se dogoditi da se registrira više korisnika s identičnim e-mailom. Također lozinke treba u bazu spremiti kriptirane da se ne zna njihov sadržaj.

2. Prijavljivanje u sustav

Korisnik se prijavi u sustav sa svojim nazivom email-a te lozinkom. Ukoliko prijavljivanje ne uspije sustav treba vratiti povratnu informaciju korisniku o greški.

3. Unos, uređivanje, pregled podataka o:

- **Prodavatelju**

Korisnik može uređivati podatke o tvrtki kao što su naziv tvrtke, adresa, OIB broj te svim ostalim podacima koji su nužni za izdavanje računa a propisani su zakonom. Također može urediti podatak da li se nalazi u sustavu PDV-a, te se na osnovu tog podatka određuje obračun i ispis računa. U svakom trenutku može promijeniti taj podatak.

- **Kupcima**

Moguće je dodavati nove, editirati i brisati postojeće kupce. Moguće je upravljati svim podacima kupca koji su nužni za izdavanje računa.

- **Artiklima**

Moguće je dodavati nove, editirati i brisati postojeće artikle. Moguće je unijeti naziv, cijenu te stopu PDV-a pojedinog artikla.

- **Poslovni prostorima**

Korisnik ima mogućnost dodavanja neograničenog broja poslovnih prostora. Također može promijeniti podatke određenog poslovnog prostora kao što je adresa.

- **Naplatnim uređajima**

Moguće je dodavati neograničen broj naplatnih uređaja za pojedini poslovni prostor. Broj naplatnog uređaja mora biti jedinstven na razini poslovnog prostora.

- **Operaterima naplatnih uređaja**

Moguće je unijeti neograničen broj operatera naplatnih uređaja na razini tvrtke. Također je moguće unijeti nove, editirati i brisati postojeće.

4. Kreiranje računa

Kod kreiranja novog računa potrebno je omogućiti:

- Odabir kupca, poslovnog prostora, naplatnog uređaja i operatera, načina plaćanja. Svi ti podaci postoje već uneseni u bazi te ih je ovdje moguće izabrati s popisa.
- Mogućnost odabira više artikala iz baze s mogućnosti uređivanja početne cijene, količine, PDV-a te iznosa popusta.

Prilikom spremanja računa potrebno je zapisati trenutni datum i vrijeme, izračunati jedinstveni broj računa gdje je računanje specificirano po zakonu na način da slijednost računa ide po naplatnom uređaju unutar poslovnog prostora.

5. Pregled svih te pojedinačnih računa

Korisnik u modulu računa može na jednostavan način imati uvid u sve izdane račune, te može vidjeti sadržaj pojedinog računa.

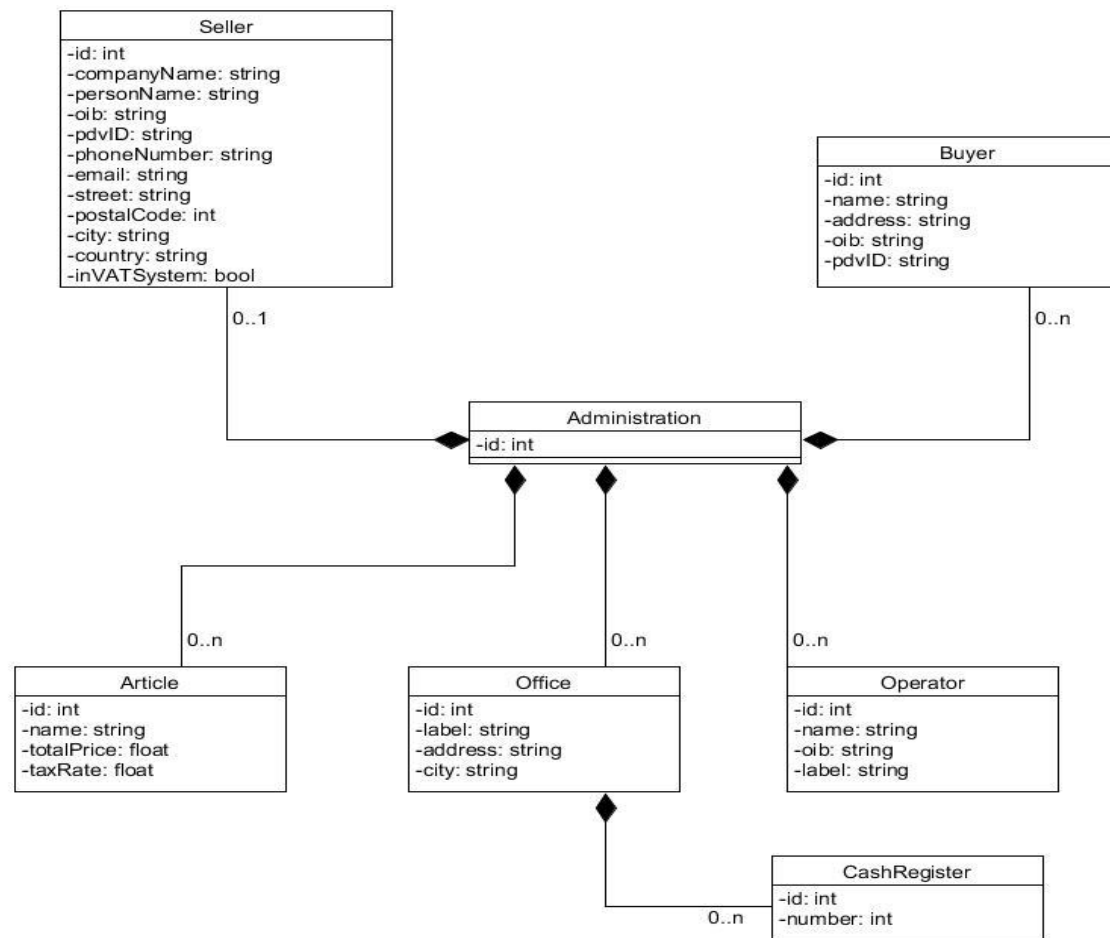
6. Storniranje računa

Svaki račun koji je izdan treba biti moguće stornirati tako da bude identičan izdanom računu ali s negativnim vrijednostima kalkulacije te novim jedinstvenim brojem računa na razini identičnog naplatnog uređaja. Bitno je za naglasiti da se izdani račun ne smije na bilo koji način mijenjati ili brisati, te on mora biti cijelo vrijeme pohranjen u bazi. Također promijene u administraciji kod artikala, kupaca te svih ostalih stavki ne smiju biti na bilo koji način povezane s računom te mijenjati bilo koju od obaveznih stavki na računu.

5.3 Dizajn domene

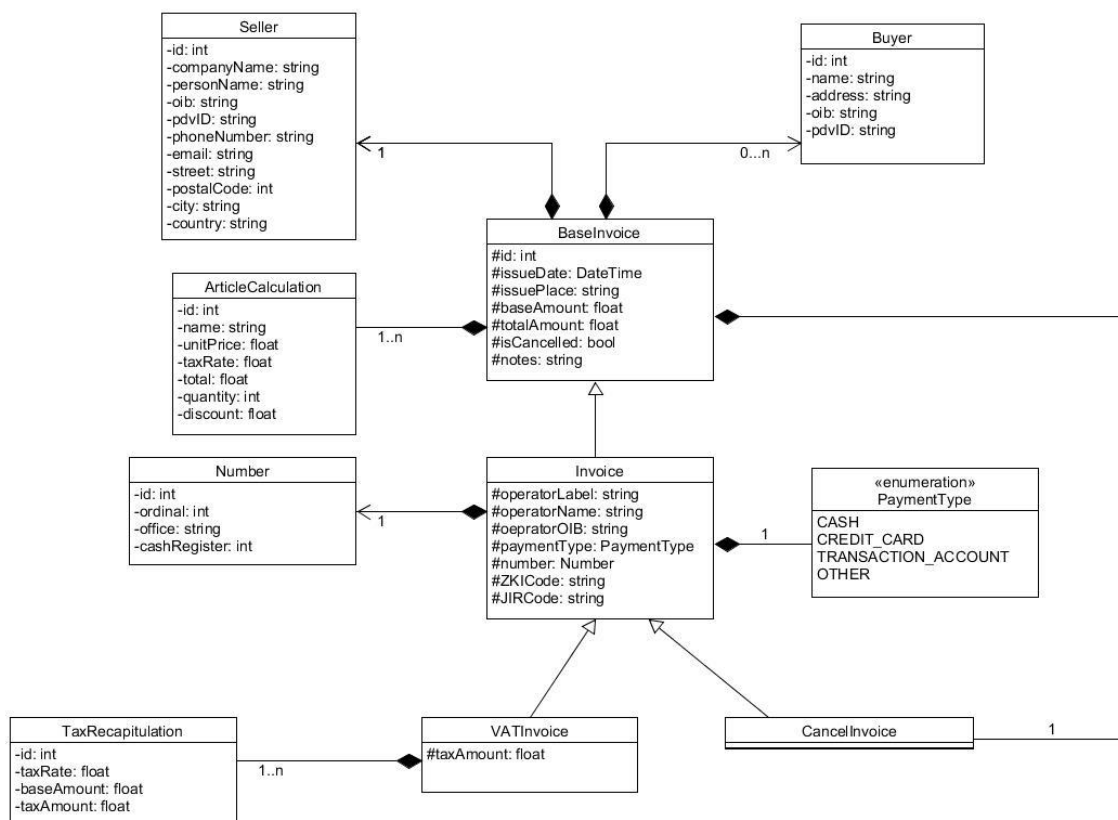
Nakon nekoliko održanih razgovora s domenskim stručnjakom možemo napraviti detaljan domenski model. U sklopu dizajna domenskog modela koristit će se UML za modeliranje. U ovome koraku je bitno održati kontakt s domenskim stručnjakom ukoliko se jave neke dvojbe oko modela.

Za modeliranje koristiti će se dijagram klasa koji pokazuje statički pogled na sustav odnosno prikazuje nasljeđivanje klasa i veze među njima. Pri modeliranju se neće prikazati sve korištene klase u jednom monolitnom modelu već u dva odvojena modula: administracija i računi. Prvi je modul administracije kojeg modeliramo iz prikupljenih informacija putem razgovora i dokumentirane specifikacije zahtjeva. Na kraju modeliranja dobijemo dijagram kao na slici 5.4.



Slika 5.4. Dijagram klasa administracije

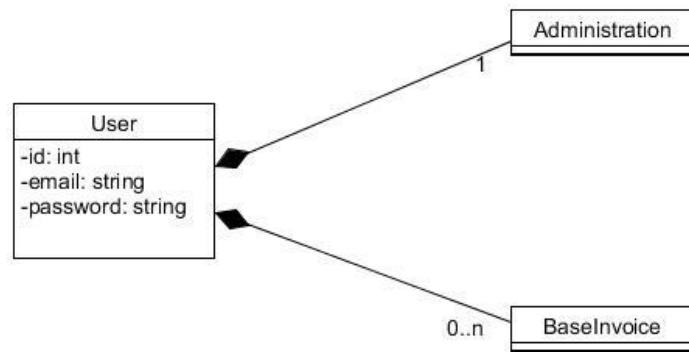
U ovoj fazi može sudjelovati i domenski stručnjak na način da mu objasnimo dijagram te pojam asocijacija, agregacija i nasljeđivanja. Sljedeći modul je modul računa čiji je dijagram prikazan na slici 5.5.



Slika 5.5. Dijagram klasa računa

Još je preostalo grafički prikazati način na koji je povezan registrirani korisnik između navedena dva modula što se može vidjeti na slici 5.6.

Kao što se može vidjeti na prikazanim primjerima, umjesto jednog velikog monolitnog dijagrama koristimo više manjih čime postizemo puno jasniju sliku strukture sustava. Kod korištenja jednog velikog dijagrama postane jako teško povezati sve u sustavu, pogotovo kod jako velikih i kompleksnih sustava gdje se je vrlo lako izgubiti u svim korištenim pojmovima.



Slika 5.6. Dijagram povezanosti korisnika sa ostatkom sustava

5.4 Tehnologije implementacije

Implementacija web aplikacije „eRačun“ bazira se na sljedećim tehnologijama koje će biti ukratko opisane:

- PHP
- Symfony framework
- Doctrine ORM
- MySQL

5.4.1 PHP

PHP (engl. *PHP: Hypertext Preprocessor*) je serverski skriptni programski jezik otvorenog koda (engl. *open-source*) korišten za izradu dinamičkih web stranica. Izvorno je napisan u programskom jeziku C te svatko može imati pristup izvornom kodu i po potrebi ga mijenjati. Stvorio ga je Rasmus Lerdorf 1994. godine [6] implementirajući brojanje posjeta na svojoj privatnoj web stranici. Prve verzije su se zvale PHP/FI (engl. *Personal Home Page Tools/Forms Interpreter*) i bile su implementirane u Perl programskom jeziku. Kasnije kada je nastala potreba za umetanjem više mogućnosti i funkcionalnosti, Lerdorf je razvio novu verziju implementiranu u C programskom jeziku. Može se reći da je PHP nastao slučajno bez nekog posebnog plana u startu. Prva verzija koja podsjeća na današnji PHP je verzija PHP 3 koja je objavljena 1998. godine. Nakon toga izlazile su verzije PHP 4, PHP 5 i PHP 7 (2015. godine) gdje je svaka imala dosta poboljšanja. Procijene su da PHP trenutno „pokreće“ 80% web stranica. Danas je PHP moderni objektno orijentirani programski jezik koji nam pruža implementaciju svih principa objektno orijentiranog programiranja.

Objektno orijentirane osobine koje PHP ima:

- Razine vidljivosti metoda i atributa
 - Public – može se pristupiti od bilo gdje (unutar i izvan klase)
 - Protected – moguć je pristup unutar klase i njezinih podklasa
 - Private – moguć je pristup samo unutar klase

- Apstraktne klase

Imaju nepotpunu implementaciju te se ne mogu instancirati. Apstraktnu klasu nasljeđuje klasa koja poprima sva svojstva i metode apstraktne klase s mogućnosti umetanja ostatka implementacije. Nakon toga možemo takvu klasu koja nasljeđuje apstraktnu bez problema instancirati.

- Nasljeđivanje

Jedna klasa može imati više podklasa, dok svaka klasa može imati najviše jednu nadklasu.

- Sučelje

To je sučelje između klase i vanjskog svijeta i predstavlja ugovor koji definira skup ponašanja koje implementirajući objekt mora imati. Dobra programerska praksa je da se što više programira prema sučelju, a ne prema implementaciji. Jedna klasa može implementirati više od jednog sučelja.

Trenutno postoji jako puno PHP radnih okruženja [14] od kojih su poznatiji: Symfony, Laravel, Codeigniter, Zend, Yii i svima im je zajedničko da ubrzavaju razvoj pružajući programerima kostur za razvoj aplikacija kroz razne gotove komponente te razne alate. Kod napisan koristeći radno okruženje je u veliko mjeri dobro organiziran te ponovno iskoristiv (engl. *reusable*). Programeri na taj način postaju puno produktivniji u razvoju aplikacija.

U sklopu ovoga rada je odabran Symfony radno okruženje koji je prepoznatljiv u korištenju kod velikih i kompleksnih poslovnih aplikacija, te zahtijeva nešto više znanja nego kod ostalih te je zbog toga nešto strmija krivulja učenja.

5.4.2 Symfony radno okruženje

Symfony [15] je potpuno (engl. *full stack*) radno okruženje za izradu aplikacija koji pruža alate za lakši razvoj svih dijelova aplikacije od korisničkog sučelja do baze podataka. Symfony se definira kao skup ponovno iskoristivih komponenti, kao radno okruženje,

filozofija i zajednica gdje svi rade zajedno u harmoniji. [8] Mnogi drugi poput Laravel frameworka koriste Symfony komponente. Symfony je u sklopu izrade web aplikacije za izdavanje računa izabran samo kao alat prezentacijskog sloja, budući da je aplikacija izrađena sljedeći princip DDD-a te slojevite arhitekture. Na taj način se može vrlo lako promijeniti radno okruženje bez velikih promjena u kodu. Ovdje će biti opisano kako se Symfony uklapa u kontekst izrađene aplikacije. Instalacijom paketa Symfony-a dobije se početna struktura kao na slici 5.7.



Slika 5.7. Početna konfiguracija projekta

Ovdje će se navesti uloga pojedinog foldera:

- app

Tu se nalazi konfiguracija infrastrukture (spajanje s bazom, mail) te konfiguracija aplikacije. Bitno je za reći da se konfiguracija infrastrukture mijenja u ovisnosti o serveru na kojemu se aplikacija izvodi (npr. razvojni ili produkcijski server), dok aplikacijska konfiguracija mijenja ponašanje aplikacije u ovisnosti o okolini. Okoline mogu biti produkcijska (brzo izvršavanje) i razvojna okolina (lakše debugiranje).

- src

U folderu src se nalazi izvorni kod aplikacije. Tu se nalazi AppBundle folder u kojemu se nalazi kod napisan pomoću Symfony-a koji se većinom odnosi na prezentaciju aplikacije, te čisti (engl. *plain*) PHP kod koji sadrži domenski, infrastrukturni i

aplikacijski sloj aplikacije za izdavanje računa. Gore navedeni konteksti trebaju biti što manje povezani kako bi se što manje ovisilo o radnom okruženju.

- Tests – koristi se za testove. Ovaj folder zrcali strukturu src direktorija.
- Vendor – koristi se za vanjske pakete koji se koriste u aplikaciji.

- Web

U navedenom folderu nalaze se Symfony prednji kontroleri koji obrađuju sve zahtjeve za web site. Prednji kontroler je jedan od uzoraka dizajna te se tu inicijalizira aplikacija i uključivanje globalnih funkcionalnosti (automatsko učitavanje klasa, keš i debugiranje). Glavni zahtjev dolazi prvo na prednji kontroler te se proslijeđuje dalje prema odgovarajućim kontrolerima i akcijama. Također se u ovom folderu nalaze Symfony Web Assets u što spadaju stvari poput CSS, javascript fileova te slika.

- Composer.json

Tu se nalazi lista svih vanjskih paketa korištenih u aplikaciji gdje je za svaki paket određena minimalna i maksimalna verzija paketa. Koristi JSON format kao na slici 5.8. Za dohvat svih vanjskih paketa je zadužen Composer koji je alat koji upravlja vanjskim paketima. U navedenom file-u je također naveden automatsko učitavanje klasa. Za potrebe aplikacije koristi se PSR-4 koji definira automatsko učitavanje klasa iz njihovog FQCN (engl. *Fully qualified class name*) imena.

```

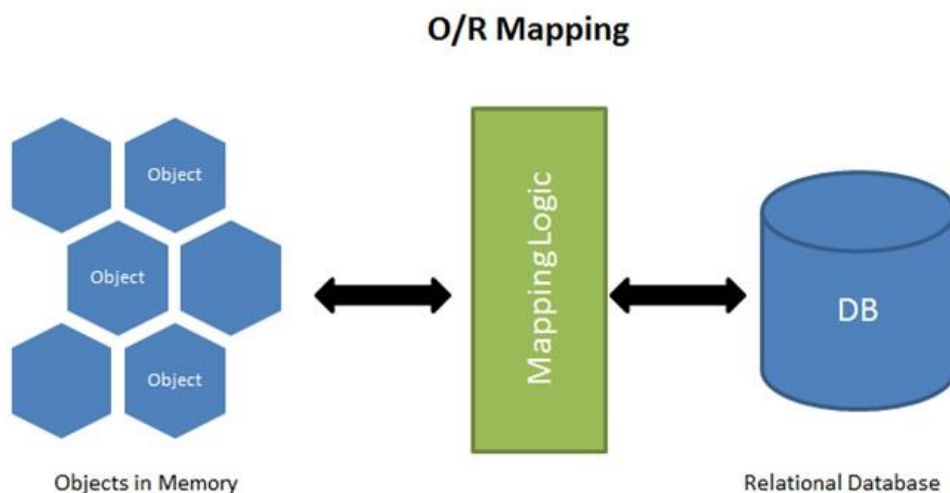
{
    "name": "diplomski",
    "license": "MIT",
    "type": "project",
    "description": "The \"Symfony Standard Edition\" distribution",
    "autoload": {
        "psr-4": { "": "src/" },
        "classmap": [ "app/AppKernel.php", "app/AppCache.php" ]
    },
    "autoload-dev": {
        "psr-4": { "Tests\\": "tests/" }
    },
    "require": {
        "php": ">=5.5.9",
        "symfony/symfony": "3.2.*",
        "doctrine/orm": "^2.5",
        "doctrine/doctrine-bundle": "^1.6",
        "doctrine/doctrine-cache-bundle": "^1.2",
        "symfony/swiftmailer-bundle": "^2.3",
        "symfony/monolog-bundle": "^3.0",
        "symfony/polyfill-apcu": "^1.0",
        "sensio/distribution-bundle": "^5.0",
        "sensio/framework-extra-bundle": "^3.0.2",
        "incenteev/composer-parameter-handler": "^2.0"
    },
    "require-dev": {
        "sensio/generator-bundle": "^3.0",
        "symfony/phpunit-bridge": "^3.0",
        "phpunit/phpunit": "^5.6.0",
        "kint-php/kint": "^2.1"
    }
}

```

Slika 5.8. Composer.json datoteka

5.4.3 Doctrine ORM

Doctrine je set PHP paketa [7] otvorenog koda koji se bave spremanjem i čitanjem podataka iz baze. ORM (engl. *object relational mapper*) pretvara podatke iz objektno orijentiranih objekata u tablice relacijskih baza i obrnuto kao na slici 5.9. Koristi Data Mapper ORM implementaciju što znači da sadrži sloj koji sadrži pretvornike za pretvaranje podataka između objekata i baze, ali gdje su objekti neovisni jedni o drugom i o samim pretvaračima. Korištenje ORM-a programerima olakšava svakodnevni rad na način da pruža fokus na objektno orijentirani model umjesto na pisanje čistog SQL-a. SQL logika je sakrivena u potpunosti od ostatka aplikacije. Doctrine također interno koristi transakcije čime se osigurava atomičnost (engl. *atomicity*), dosljednost (engl. *consistency*), izolacija (engl. *isolation*) i trajnost (engl. *durability*).



Slika 5.9. Način rada ORM-a [15]

ORM prema nekim procjenama olakšava pretvaranje objekata u tablice u 90% slučajeva, ali ponekada dolazi do problema u područjima gdje se koriste jako kompleksni upiti te velike količine podataka koji se spremaju u radnu memoriju servera. ORM nije prikladan za svaki projekt, te je potrebno ponekada detaljno poznavati njegov način rada kako bi se izbjegle zamke koje korištenje ORM-a ima.

Objekti u koje se pretvaraju tablice iz baze podataka zovu se entiteti te je primjer entiteta kupca prikazan na slici 5.10. Oni sadrže sva potrebna svojstva koje Doctrine sprema i puni iz baze podataka. Također Doctrine podržava rad s povezanim tablicama koristeći asocijacije (engl. *relationship*) te su moguće veze jedan na jedan, jedan na više, više na jedan te više na više. Kod takvih veza jako je bitno razumjeti vlasničku (engl. *owning*) i inverznu stranu jer Doctrine provjerava samo vlasničku stranu za promjene koje sprema u bazu, dok promjene na obrnutoj strani ignorira. Doctrine ima svoj jezik za upite DQL (engl. *Doctrine Query Language*) koji omogućava da se upiti šalju direktno objekt modelu. U DQL-u se umjesto tablica i stupaca koriste imena klasa i atributa. Dobra praksa u korištenju Doctrine-a je da česte operacije dohvaćanja i spremanja izdvojimo u repozitorij klase čime postizemo enkapsulaciju i ponovnu iskoristivost koda.


```

/**
 * Kupac.
 *
 * @ORM\Entity(repositoryClass="IssueInvoices\Infrastructure\Doctrine2\BuyerRepositoryImpl")
 * @ORM\Table(name="administration_buyer")
 */
class Buyer
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    private $id;

    /**
     * Ime i prezime
     *
     * @var string
     * @ORM\Column(type="string", length=50)
     */
    private $name;

    /**
     * @var string
     * @ORM\Column(name="address", type="string")
     */
    private $address;

    /**
     * @var string
     * @ORM\Column(type="string", length=15, nullable=true)
     */
    private $oib;

    /**
     * PDV ID broj
     *
     * @var string
     * @ORM\Column(name="pdv_id", type="string", length=15, nullable=true)
     */
    private $pdvID;

    /**
     * @ORM\ManyToOne(targetEntity="Administration", inversedBy="buyers")
     * @ORM\JoinColumn(name="administration_id", referencedColumnName="id")
     */
    private $administration;
}

```

Slika 5.10. Doctrine entitet sa potrebnim metapodacima

Iz slike 5.10. se može vidjeti primjer Doctrine entiteta kojemu su sve vrijednosti definirane preko anotacija koje predstavljaju metapodatke za rad ORM-a. Također je za entitet definiran i njegov repozitorij iz kojeg se vrši dohvat podataka za taj entitet na jednom mjestu. Ukoliko se ne specificira ime atributa tada Doctrine radi po svojoj konvenciji imenovanja. Postoji više načina definiranja metapodataka ORM-a. U okviru izrađene aplikacije korištena je metoda preko anotacija, ali podržane su i druge kao što je preko XML-a ili YAML formata.

5.4.4 MySQL

MySQL [9] je besplatan relacijski sustav za upravljanje bazom podataka, te je čest izbor baze za projekte otvorenog koda. Na glasu je kao vrlo stabilan i ima dobro dokumentirane module i ekstenzije za podršku od brojnih jezika. Koriste ga brojne globalne kompanije kao što su Facebook, Google, Twitter, Youtube i mnoge druge. Centralna je komponenta LAMP (engl. *Linux, Apache, MySQL, Perl/PHP/Python*) paketa otvorenog koda. U potpunosti je napisan kroz kombinaciju C i C++ programskog jezika. U sklopu priložene web aplikacije MySQL se koristi za pohranu podataka. Budući da se komunikacija s bazom odvija posredstvom Doctrine ORM-a koji ima podršku za sve poznate relacijske sustave, ORM se sam brine o tome kako će objekte pretvoriti u tablice u MySQL-u.

Aplikaciju je moguće razvijati na način da se prvo stvori nacrt baze podataka te se potom može koristeći Doctrine naredbe generirati entitete. Ali postoji i druga vrsta, a to je implementacija entiteta pa tek onda generiranje nacrtu baze na osnovu entiteta. U sklopu izrađene aplikacije korišten je drugi pristup.

5.5 Preslikavanje domene u model koda

U ovoj fazi je potrebno učinkovito preslikati domenski model sustava (definiran u poglavlju 5.2.) u model koda koji mora poštivati sva poslovna pravila specificirana u uspostavljenom modelu. Navedena faza je u razvoju softvera izuzetno bitna te je često teško preslikati domenski model u kod. Da bi preslika stvarnosti bila što bolja DDD princip daje na raspolaganje taktičke principe koji su osnova domenskog modela koda.

U kontekstu web aplikacije „eRačun“ uspostavi se domenski sloj gdje će ići sve poslovne stvari iz uspostavljenog dizajna domene u poglavlju 5.2. Navedeni sloj ne smije imati nikakvu vezu s okolinom, pa tako ne smije znati ništa o primjerice slanju poruka ili spremanju podataka. Koristeći DDD princip prvo se gleda prostor problema problematične domene, odnosno da li trebamo domenom dijeliti na poddomene. Ovdje je aplikacija dosta jednostavna s jednom domenom izdavanja računa za koju nema potrebe raditi dodatnu podjelu. Primjerice ukoliko bi imali još neke dodatne funkcionalnosti kao što je upravljanje skladištima artikala, tada bi izdavanje računa bila jedna poddomena unutar šire domene. Prema tome cijela aplikacija nema podijele na poddomene i ona predstavlja jednu domenom unutar koje se nalazi jedan ograničeni kontekst, a to je kod ove aplikacije ograničeni kontekst izdavanja računa.

DDD pruža taktički princip modula pa ga se može iskoristiti kako bi se postigla komunikacijska i funkcionalna kohezija. Tako dolazi do podjele ograničenog konteksta izdavanja računa na dva modula: administracija i računi.

Kada je gotova podjela na module, tada unutar svakog modula modeliramo podatke koji nam trebaju. Tako za modul administracije potrebno je prvo saznati koji su to elementi koji bi bili predstavljeni kao entiteti, odnosno za koje je potrebna jedinstvena identifikacija. Svi elementi iz dizajna modela administracije trebaju biti entiteti pošto svi oni moraju biti jedinstveno određeni. Kako ova aplikacija omogućuje registraciju više korisnika treba uspostaviti i entitet administracije kao na slici 5.11. koji mora biti jedinstveno određen za svakoga korisnika. Ovdje su definirane i ORM metapodaci u obliku anotacija, iako bi bolji izbor bio da definiranje metapodataka izdvojimo van klase domene primjerice preko XML formata. Na isti se način definiraju i ostale klase tako da zadovoljimo dizajn domenskog modela koji smo uspostavili pri razgovoru s domenskim stručnjakom.

```
/**
 * Administracija korisnika.
 *
 * @ORM\Entity(repositoryClass="IssueInvoices\Infrastructure\Doctrine2\AdministrationRepositoryImpl")
 * @ORM\Table(name="administration")
 */
class Administration
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    private $id;

    /**
     * @var Seller
     * @ORM\OneToOne(targetEntity="Seller", mappedBy="administration", cascade={"persist", "remove"}, orphanRemoval=true)
     */
    private $seller;

    /**
     * @var Article[]
     * @ORM\OneToMany(targetEntity="Article", mappedBy="administration", cascade={"persist", "remove"}, orphanRemoval=true)
     */
    private $articles;

    /**
     * @var Office[]
     * @ORM\OneToMany(targetEntity="Office", mappedBy="administration", cascade={"persist", "remove"}, orphanRemoval=true)
     */
    private $offices;

    /**
     * @var Buyer[]
     * @ORM\OneToMany(targetEntity="Buyer", mappedBy="administration", cascade={"persist", "remove"}, orphanRemoval=true)
     */
    private $buyers;

    /**
     * Operateri naplatnih uređaja
     *
     * @var Operator[]
     * @ORM\OneToMany(targetEntity="Operator", mappedBy="administration", cascade={"persist", "remove"}, orphanRemoval=true)
     */
    private $operators;

    /**
     * @var User
     * @ORM\OneToOne(targetEntity="IssueInvoices\Domain\Model\User\User", inversedBy="administration")
     */
    private $user;

    public function __construct()
    {
        $this->articles = new ArrayCollection();
        $this->offices = new ArrayCollection();
        $this->buyers = new ArrayCollection();
        $this->operators = new ArrayCollection();
    }
}
```

Slika 5.11. Entitet administracije

U navedenom modulu se također implementiraju sva sučelja repozitorija sa specifikacijom potrebnih metoda. Tu se ne smije nikako stavljati implementacija repozitorija jer domena treba biti izolirana od tehničkih stvari.

Drugi modul je modul računa kojeg treba modelirati također u skladu zajedničkog jezika prema uspostavljenom dizajnu domene. Kako postoji više vrsta računa najbolji izbor za model je apstraktna klasa računa `BaseInvoice` kao na slici 5.12. koja sadrži sve elemente koji svaki tip računa ima. Zbog toga se izrade dvije klase: `Invoice` koji predstavlja račun bez PDV-a, te `VATInvoice` koji predstavlja račun s PDV-om. U navedenom slučaju klasa `Invoice` nasljeđuje `BaseInvoice` kao na slici 5.13., a `VATInvoice` nasljeđuje `Invoice` pošto ima sve elemente klase `Invoice` s dodatnim atributima rekapitulacije poreza i ukupnog iznosa poreza. Što se tiče tipa računa preostaje napraviti stornirani račun koji sadrži originalni račun te nasljeđuje klasu `Invoice` koja sadrži podatke operatera koji su potrebni za stornirani račun.

```
/**
 * @ORM\Entity(repositoryClass="IssueInvoices\Infrastructure\Doctrine2\InvoiceRepositoryImpl")
 * @ORM\Table(name="invoice")
 * @ORM\InheritanceType("SINGLE_TABLE")
 * @ORM\DiscriminatorColumn(name="type", type="string")
 * @ORM\DiscriminatorMap({
 *   "FISCAL" = "Invoice",
 *   "FISCAL_VAT" = "VATInvoice",
 *   "CANCEL" = "CancelInvoice"
 * })
 */
abstract class BaseInvoice
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    protected $id;

    /**
     * Datum i vrijeme izdavanja.
     *
     * @var DateTime
     * @ORM\Column(name="issue_date", type="datetime")
     */
    protected $issueDate;

    /**
     * Mjesto izdavanja.
     *
     * @var string
     * @ORM\Column(name="issue_place", type="string")
     */
    protected $issuePlace;

    /**
     * @ORM\Column(type="float")
     */
    protected $baseAmount;

    /**
     * @ORM\Column(type="float")
     */
    protected $totalAmount;
```

Slika 5.12. Bazna klasa računa

Doctrine izravno podržava ovakav tip implementacije kroz „Inheritance Mapping“ na način da se u baznom elementu specificiraju sve klase koje nasljeđuju baznu klasu. Doctrine onda „zna“ pretvoriti entitet u izgled baze podataka na način da su svi elementi unutar bazne klase obavezni (nisu dozvoljene null vrijednosti), dok su sve vrijednosti naslijeđenih klasa opcionalne.

```
*
* @ORM\Entity
*/
class Invoice extends BaseInvoice
{
    /**
     * Oznaka operatera
     */
    * @var string
    * @ORM\Column(type="string", length=20)
    */
    protected $operatorLabel;

    /**
     * Ime i prezime operatera
     */
    * @var string
    * @ORM\Column(type="string", length=30)
    */
    protected $operatorName;

    /**
     * OIB operatera
     */
    * @var string
    * @ORM\Column(type="string", length=30)
    */
    protected $operatorOIB;

    /**
     * Tip placanja (gotovina, transakcijski račun)
     */
    * @var string
    * @ORM\Column(type="string", length=30)
    */
    protected $paymentType;

    /**
     * Broj računa.
     */
    * @var Number
    * @ORM\OneToOne(targetEntity="InvoiceNumber", cascade={"persist", "remove"}, orphanRemoval=true)
    */
    protected $number;
```

Slika 5.13. Nasljeđivanje bazne klase računa

Za entitet račun koji je središte modula se povezuju i ostali elementi koji čine pojedini račun. Račun postaje centralni entitet koji sadrži veze prema drugim elementima koje sadrži. U kontekstu DDD-a njega se modelira kao agregat koji sakriva internu strukturu od ostatka svijeta. On također osigurava ispravnost svih poslovnih pravila unutar svoje strukture. Kako se radi s agregatom, instanciranje tako velike strukture sa svim potrebnim elementima treba također sakriti od klijenta unutar domene. Zbog toga se uvodi factory računa kao na slici 5.14.

koji se brine o njegovom stvaranju. Klijent pošalje specifikaciju parametara po kojima želi da se napravi račun, a factory to i osigura.

Iz dizajna domenskog modela može se vidjeti da svaki fiskalni račun mora imati tip plaćanja. Kako za tip plaćanja nije bitan identitet, bitna je samo vrijednost da li je tip primjerice gotovina, njega modeliramo kao vrijednosni objekt kao na slici 5.15. Kako takav objekt treba biti nepromjenjiv u klasi ne smijemo dozvoliti promjenu njegove vrijednosti.

```
public function createFromData($data, $administration): BaseInvoice
{
    // Treba saznati da li je prodavatelj u sustavu PDV-a
    $seller = $administration->getSeller();
    $isInVatSystem = $seller->isInVatSystem();

    if ($isInVatSystem) {
        // Izdajemo fiskalni racun s PDV-om
        $invoice = new VATInvoice();
        $invoice->setUser($administration->getUser());
        $this->addBasicMandatoryInfo($invoice, $data, $seller);

        // Izracunamo kalkulaciju računa
        $this->doInvoiceCalculation($invoice, $isInVatSystem);

        // Izracunamo rekapitulaciju poreza
        $this->calculateTaxRecapitulation($invoice);
    } else {
        // Izdajemo fiskalni racun bez PDV-a
        $invoice = new Invoice();
        $invoice->setUser($administration->getUser());
        $this->addBasicMandatoryInfo($invoice, $data, $seller);

        // Izracunamo kalkulaciju računa
        $this->doInvoiceCalculation($invoice, $isInVatSystem);
    }

    return $invoice;
}
```

Slika 5.14. Factory računa


```

/**
 * Načini plaćanja računa
 */
class PaymentType
{
    /**
     * Naziv trenutačne opcije
     * @var string
     */
    private $name;

    /**
     * @param string $name Naziv opcije
     * @throws InvalidArgumentException Ako zadana opcije nije ispravna
     */
    public function __construct($name)
    {
        $this->name = (string) $name;
    }

    public function __toString()
    {
        return $this->name();
    }

    /**
     * Dohvaća naziv
     * @return string
     */
    public function name()
    {
        return $this->name;
    }

    /**
     * Dohvaća naslov
     */
    public function titleName()
    {
        return self::names()[$this->name];
    }
}

```

Slika 5.15. Tip plaćanja kao vrijednosni objekt

Bitan dio aplikacije je radnja generiranja broja računa. Kako generiranje računa nije, odgovornost klase računa, nego je neki „globalni“ zadatak to je dobro modelirati kao servis. U ovome kontekstu to je domenski servis koji sadrži svu potrebnu logiku kreiranja kao na slici 5.16.

```

/**
 * Domenski servis generiranja broja računa
 */
class InvoiceNumberGenerator
{
    private $invoiceRepository;

    public function __construct(InvoiceRepository $invoiceRepository)
    {
        $this->invoiceRepository = $invoiceRepository;
    }

    public function calculateOrdinalNumber(
        string $officeLabel,
        int $cashRegisterNumber,
        int $userId
    ): int {
        $startDate = (new \DateTimeImmutable('first day of January this year'))->setTime(0, 0, 0);
        $endDate = (new \DateTimeImmutable('last day of December this year'))->setTime(0, 0, 0);

        $ordinalNumber = $this->invoiceRepository->findMaxOrdinalNumberForCombination(
            $officeLabel,
            $cashRegisterNumber,
            $userId,
            $startDate,
            $endDate
        );

        if ($ordinalNumber) {
            return ++$ordinalNumber;
        } else {
            return 1;
        }
    }
}

```

Slika 5.16. Domenski servis generiranja broja računa

Iz priložene slike 5.16. može se primijetiti konstruktor klase koji prima parametre izvana, odnosno instanciranje primjerice repozitorija nije njegova odgovornost već preko dependency injection-a samo primi ovisnost.

Kao što je stvar kod modula administracije tako i ovdje treba osigurati da domena ne ovisi o tehničkim stvarima kao što je primjerice baza. Zbog toga se uspostavlja sučelje repozitorija računa kao na slici 5.17., te na taj način implementacija repozitorija koja je u nekom vanjskom sloju ovisi o definiranoj apstrakciji. Na taj način oba dva sloja ovise o uspostavljenoj apstrakciji što je osnova DIP principa.


```

interface InvoiceRepository
{
    /**
     * Traži maksimalni redni broj računa za danu kombinaciju
     *
     * @param string $officeLabel      Oznaka poslovnog prostora
     * @param int    $cashRegisterNumber broj blagajne unutar poslovnog prostora
     * @param int    $userId identifikator korisnika
     *
     * @return int|null null ako ne postoji kombinacija u bazi
     */
    public function findMaxOrdinalNumberForCombination(
        string $officeLabel,
        int $cashRegisterNumber,
        int $userId,
        \DateTimeImmutable $startDate,
        \DateTimeImmutable $endDate
    );
}

```

Slika 5.17. Sučelje repozitorija računa

5.6 Korištenje slojevite arhitekture

Kako je domena izolirana kroz presliku dizajna domene u kod, za arhitekturu će biti korištena slojevitá arhitektura koja će uz već definirani domenski sloj sadržavati još aplikacijski i infrastrukturni.

5.6.1 Aplikacijski sloj

Kao što je rečeno u poglavlju 4. za aplikacijski sloj je bitno da bude tanak i ne sadrži poslovna pravila. On je klijent domene i zna koga pozvati kako bi se slučaj korištenja sistema zadovoljio.

Iz primjera aplikacijskog servisa na slici 5.18. je vidljivo kako unutar njega nema poslovnih odluka koje su sve sakrivene u domenskom sloju. U ovome slučaju servis dobije identifikator iz prezentacijskog sloja, te osigurava da se slučaj korištenja storniranja računa odradi. Servis mora generirati iznimke ukoliko dođe do nepredviđenog ponašanja. Servis provjerava da li je račun pronađen u bazi preko sučelja definiranog u domenskom sloju. Također mora ispitati da li je račun već storniran. Ukoliko je sve zadovoljeno stornira odabrani račun te generira novi storno račun koji sadrži sve vrijednosti u minusu.

```

/**
 * Stvara storno račun originalnog računa
 *
 * @param int $invoiceId - identifikator originalnog racuna kojeg treba stornirati
 *
 * @throws CancelInvoiceException ako se probaju stornirati već stornirani racuni ili storno racuni
 * @throws InvalidArgumentException ako račun nije pronađen u bazi
 */
public function cancelInvoice(int $invoiceId)
{
    $invoice = $this->invoiceRepository->find($invoiceId);

    if (!$invoice) {
        throw new InvalidArgumentException(sprintf('Račun "%s" nije pronađen', $invoiceId));
    }
    if ($invoice->isCancelled()) {
        throw CancelInvoiceException::invoiceAlreadyCancelled();
    }
    if ($invoice->isCancel()) {
        throw CancelInvoiceException::invoiceIsCancel();
    }

    $invoice->cancel();
    $cancelInvoice = $this->invoiceFactory->createCancelFromOriginalInvoice($invoice);
    $this->entityManager->persist($cancelInvoice);
    $this->entityManager->persist($invoice);
    $this->entityManager->flush();
}

```

Slika 5.18. Aplikacijski servis storniranja računa

5.6.2 Infrastrukturni sloj

Infrastrukturni sloj sadrži sve tehničke stvari bitne za aplikaciju kao što je spremanje objekata ili slanje poruka. On je po svom stupnju sloj najniže razine te o njemu ne bi trebali ovisiti unutarnji slojevi. Unutar njega dolaze sve implementacije radnji u komunikaciji s primjerice bazom ili nekim vanjskim servisom. Tako se na slici 5.19. koristi Doctrine za implementaciju repozitorija pomoću jezika DQL.

```

class InvoiceRepositoryImpl extends EntityRepository implements InvoiceRepository
{
    public function findMaxOrdinalNumberForCombination(
        string $officeLabel,
        int $cashRegisterNumber,
        int $userId,
        \DateTimeImmutable $startDate,
        \DateTimeImmutable $endDate
    )
    {
        // Odabrati sve racune trazenog korisnika te brojeve tih racuna
        $sql = 'SELECT MAX(n.ordinal) FROM IssueInvoices\Domain\Model\Invoice\Invoice AS i JOIN i.number AS n
        WHERE i.user = :userId
        AND (i.issueDate BETWEEN :startDate AND :endDate)
        AND n.office = :officeLabel
        AND n.cashRegister = :cashRegisterNumber';

        return $this->getEntityManager()
            ->createQuery($sql)
            ->setParameters([
                'userId' => $userId,
                'startDate' => $startDate,
                'endDate' => $endDate,
                'officeLabel' => $officeLabel,
                'cashRegisterNumber' => $cashRegisterNumber
            ])
            ->getSingleScalarResult();
    }
}

```

Slika 5.19. Implementacija repozitorija putem Doctrine ORM-a

5.6.3 Symfony u ulozi prezentacijskog sloja

U sklopu aplikacije „eRačun“ za komunikaciju s klijentom aplikacije korišteno je PHP radno okruženje Symfony koji je opisan u poglavlju 5.4.2. Do ove točke aplikacija već ima domenski, aplikacijski i infrastrukturni sloj koji dosta dobro enkapsuliraju aplikaciju. Preostala je još direktna komunikacija s klijentom, a taj posao Symfony uvelike olakšava na način da pruža čitav niz gotovih ponovno iskoristivih komponenti koji nam znatno ubrzavaju i olakšavaju razvoj. Primjer takvih komponenti su forme, Dependency Injection, rutiranje, sigurnosna komponenta i brojne druge. Jedna od osnovnih komponenti je komponenta rutiranja koja omogućava pretvaranje klijentskih zahtjeva prema odgovarajućim akcijama u kontrolerima.

```
class InvoiceController extends Controller
{
    /**
     * @Route("/invoice/issueInvoice", name="AppBundle_Invoices_issueInvoice")
     */
    public function issueInvoiceAction(Request $request)
    {
        $userAdministration = $this->getUser()->getAdministration();

        $formInvoice = new FormInvoice();

        $form = $this->createForm(
            InvoiceType::class,
            $formInvoice,
            [
                'buyers' => $userAdministration->getBuyers(),
                'operators' => $userAdministration->getOperators(),
                'offices' => $userAdministration->getOffices(),
                'allArticles' => $userAdministration->getArticles()
            ]
        );

        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            // dump($form->getData());
            $formInvoice = $form->getData();
            // Poziv aplikacijskog servisa koji će izdati račun
            $this->get('app.application_service.issue_invoice')->issueInvoice($formInvoice, $userAdministration);

            return $this->redirectToRoute('AppBundle_Invoices_issuedInvoices');
        }

        return $this->render('AppBundle:Invoice:issueInvoice.html.twig', [
            'form' => $form->createView()
        ]);
    }
}
```

Slika 5.20. Kontroler računa

Na slici 5.20. je prikazana akcija izdavanja računa koja se sastoji od prikaza forme klijentu, unosa forme i njene validacije preko događaja koji upravlja zahtjevom. Nakon toga se radi poziv aplikacijskog servisa koji će izdati račun. Forme su jako bitan element svake web

aplikacije, stoga Symfony pruža komponentu forme koja pruža jako puno funkcionalnosti. Kako se u ovoj aplikaciji ne želi ovisiti o radnom okruženju za svaku formu postoji i njen model, odnosno forma se ne veže direktno na domenski model.

Tako primjerice za račun postoji forma računa koja predstavlja izgled forme i model računa koja predstavlja podatke koje forma „nosi“. Komponenta forme omogućava stvaranje kolekcije formi pa tako za formu računa postoji kolekcija formi artikala pošto je na svaki račun moguće dodavati više artikala. Tako su na slici 5.21. prikazani modeli forme računa i artikla.

```
class Invoice
{
    public $buyer;
    public $operator;
    public $office;
    public $articles = [];
    public $cashRegister;
    public $paymentType;

    public function addArticle(Article $article)
    {
        $this->articles[] = $article;
    }

    public function removeArticle(Article $article)
    {
        $key = array_search($article, $this->articles);
        if($key !== false) {
            unset($this->articles[$key]);
        }
    }
}

class Article
{
    /**
     * @Assert\NotBlank
     */
    public $article;

    public $totalPrice;

    public $taxRate;

    /**
     * @Assert\NotBlank
     * @Assert\Type(type="int")
     */
    public $quantity;

    /**
     * @Assert\Type(type="float")
     */
    public $discount;
}
```

Slika 5.21. Model forme računa i artikla

Uz model forme postoji i objekt forme koji je zadužen za konstrukciju forme. Tako je na slici 5.22. izrađen objekt forme računa koji pomoću metode `buildForm` popunjava podatke iz modela i prosljeđuje na prikaz. Za forme je specifično da mogu sadržavati osim osnovnih elemenata i druge forme, pa tako forma računa sadrži kolekciju formi artikala koji se mogu dodavati i brisati.

Odgovor servera koji se vraća korisniku je delegiran kroz sistem predloška (engl. *templating engine*) koji se u Symfony-u zove Twig i koristi se u izrađenoj aplikaciji kao na slici 5.23.

Ima brojne prednosti u odnosu na standardni PHP predložak a to su: jednostavan, intuitivan, podržava nasljeđivanje blokova, lako baratanje s varijablama i mnoge druge.

```
class InvoiceType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $buyers = $options['buyers'];
        $operators = $options['operators'];
        $offices = $options['offices'];
        $allArticles = $options['allArticles'];

        $builder
            ->add('articles', CollectionType::class, [
                'entry_type' => ArticleType::class,
                'entry_options' => ['label' => false],
                'allow_add' => true,
                'allow_delete' => true,
                'by_reference' => false,
                'entry_options' =>
                [
                    'allArticles' => $allArticles
                ]
            ])
    }
}
```

Slika 5.22. Klasa forme računa

```
{% extends 'AppBundle::layout.html.twig' %}

{% block content %}

    {% for invoice in invoices %}
        <tr>
            <td>{{loop.index}}</td>
            <td>{{invoice.number.ordinal}}-{{invoice.number.office}}-{{invoice.number.cashRegister}}</td>
            <td>{{invoice.issueDate | date("m/d/Y")}}</td>
            <td>{{invoice.totalAmount | number_format(2, '.', ',')}} HRK</td>
            <td>{{ invoice.operatorName }}</td>
            <td>{{ invoice.paymentType.titleName }}</td>
            <td>{{ invoiceStatus(invoice) }}</td>
            <td>
                <a href="{{ path('AppBundle_Invoices_cancelInvoice', {'invoiceId': invoice.id}) }}"> <button type="button" class="btn btn-xs btn-primary" {{ isDisabled(invoice)}}>Storniraj</button></a>
                <a href="{{ path('AppBundle_Invoice_viewInvoice', {'invoiceId': invoice.id}) }}"> <button type="button" class="btn btn-xs btn-info">Pogledaj</button></a>
            </td>
        </tr>
    {% endfor %}
}
```

Slika 5.23. Korištenje Twig-a

5.7 Testiranje aplikacije

Testiranje je jedan od softverskih procesa koje bi svaki softver trebao imati. To je proces koji dokazuje da program radi ono za što je namijenjen prema specifikaciji zahtijeva, odnosno koji otkriva različite pogreške napravljene u implementaciji. Postoji mnogo načina testiranja aplikacije od testiranja za vrijeme razvoja aplikacije pa sve do korisničkih testova prihvatljivosti.

Za potrebe testiranja priložene aplikacije koristiti će se testiranje bazirano na scenarijima gdje će se koristiti tipičan scenarij korištenja aplikacije koji se sastoji od koraka:

1. Registracija korisnika.

Korisnik se registrira u sustav sa svojim e-mail-om i lozinkom kao na slici 5.24.

2. Prijava u sustav.

Korisnik se prijavi u sustav sa svojim e-mail-om i lozinkom te se preusmjeri na početnu stranicu administracije gdje unosi potrebne podatke koje će koristiti za izdavanje računa (slika 5.25.)

3. Unos, ažuriranje i brisanje podataka:

- Prodavatelja
- Kupaca
- Artikala
- Poslovnih prostora
- Naplatnih uređaja
- Operatera

U ovome koraku korisnik unosi sve potrebne podatke kao što su primjerice podaci prodavatelja (slika 5.26.) i svi ostali potrebni podaci. Nakon provedenih svih unosa dobiva povratnu informaciju o uspješno unesenim svim parametrima (slika 5.27.). Također o ovome koraku vrši radnje ažuriranja i brisanja prethodno unesenih podataka.

4. Izdavanje novog računa kada je prodavatelj u sustavu PDV-a.

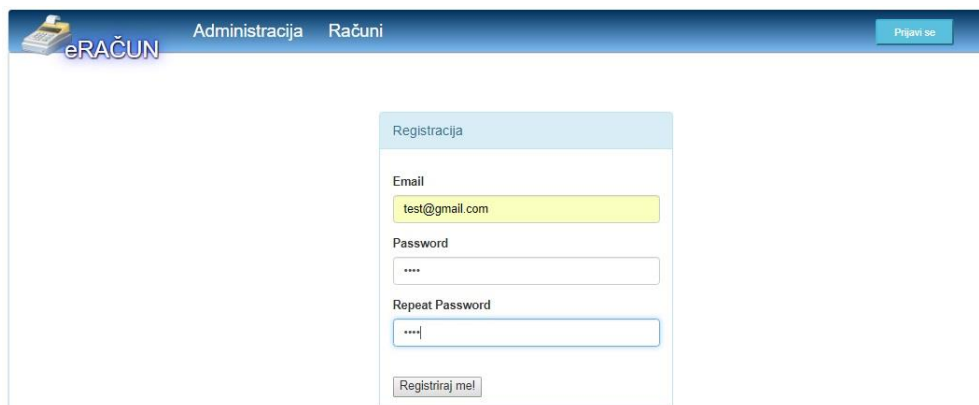
Korisnik ulazi u modul računa te mu je prikazana forma kao na slici 5.28., gdje odabire i unosi potrebne podatke i potvrdi izdavanje računa. Nakon toga koraka u popisu računa dobiva izgled računa kao na slici 5.29., budući da je trenutno odabrana opcija da je prodavatelj u sustavu PDV-a.

5. Izdavanje novog računa kada prodavatelj nije u sustavu PDV-a.

Korisnik ulazi u administraciju i odabire opciju da prodavatelj nije u sustavu PDV-a, te ponavlja radnju iz stavke 4. Nakon toga prikaže se račun kao na slici 5.30.

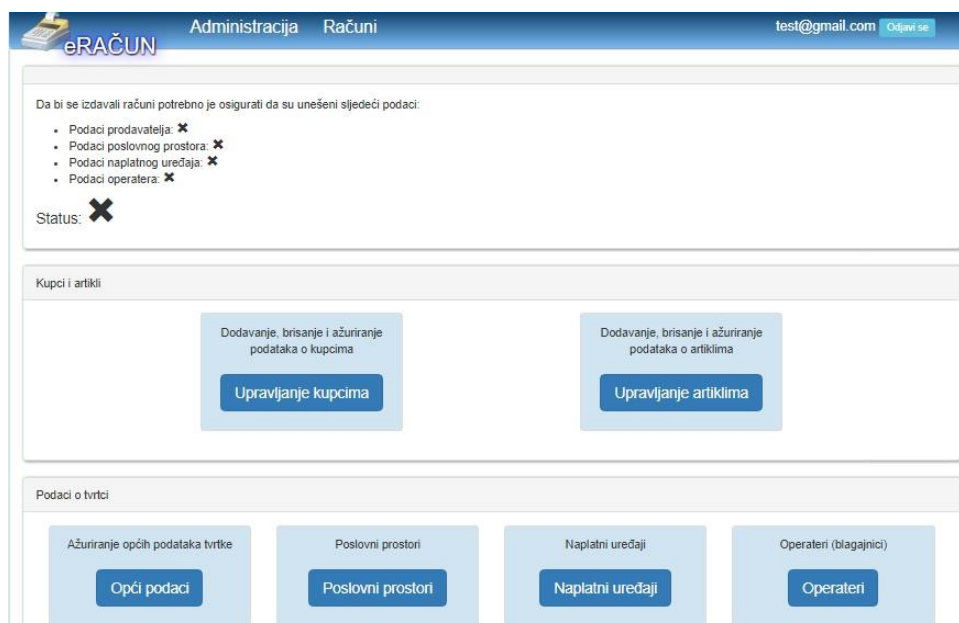
6. Storniranje računa.

Korisnik unutar modula računa gdje su prikazani svi računi odabere akciju storniranja unesenog računa iz koraka 5., te dobiva prikaz računa kao na slici 5.31.



The screenshot shows the 'Registracija' (Registration) form in the eRAČUN application. The form is titled 'Registracija' and contains the following fields: 'Email' with the value 'test@gmail.com', 'Password' with four asterisks, and 'Repeat Password' with four asterisks. A 'Registriraj me!' button is at the bottom. The application header includes the eRAČUN logo, 'Administracija' and 'Računi' tabs, and a 'Prijavi se' button.

Slika 5.24. Registracija korisnika



The screenshot shows the 'Administracija' (Administration) dashboard in the eRAČUN application. The header includes the eRAČUN logo, 'Administracija' and 'Računi' tabs, the user email 'test@gmail.com', and a 'Odjavi se' button. The main content area is divided into three sections: 1. A message box stating 'Da bi se izdavali računi potrebno je osigurati da su uneseni sljedeći podaci:' followed by a list of required data: 'Podaci prodavatelja', 'Podaci poslovnog prostora', 'Podaci naplatnog uređaja', and 'Podaci operatera', each with a red 'X' icon. Below this is a 'Status:' label with a red 'X' icon. 2. A 'Kupci i artikli' (Customers and Articles) section with two buttons: 'Upravljanje kupcima' (Customer Management) and 'Upravljanje artiklima' (Article Management). 3. A 'Podaci o tvrtci' (Company Data) section with four buttons: 'Opći podaci' (General Data), 'Poslovni prostori' (Business Spaces), 'Naplatni uređaji' (Billing Devices), and 'Operateri (blagajnici)' (Operators (Cashiers)).

Slika 5.25. Početna stranica administracije nakon prijave

Administracija Računi test@gmail.com [Odjavi se](#)

Administracija / Prodavatelj

[Uredi](#)

Naziv tvrtke	Test d.o.o.
Vlasnik	Test Testić
OIB	314141414141
PDV ID	
Broj telefona	0911111111
E-mail	test@gmail.com
Ulica	Testna 22
Poštanski broj	23206
Grad	Test
Država	Testna
U sustavu PDV-a	DA

Slika 5.26. Uspješno dodani podaci prodavatelja

Administracija Računi test@gmail.com [Odjavi se](#)

Da bi se izdavali računi potrebno je osigurati da su uneseni sljedeći podaci:

- Podaci prodavatelja: ✓
- Podaci poslovnog prostora: ✓
- Podaci naplatnog uređaja: ✓
- Podaci operatera: ✓

Status: ✓

Slika 5.27. Povratna informacija nakon unesenih svih parametara

Administracija Računi test@gmail.com [Odjavi se](#)

Administracija / Izdani računi / Novi

Kupac: TESTNI KORISN Operator: TEST TESTIĆ Poslovni prostor: TEST1 Blagajna: 1 Način plaćanja: Gotovina

Artikli:

Artikl: Testni artikl

Cijena: HRK 66

PDV: 13 %

Količina: 5

Popust: 7 %

[Izdaj račun](#)

Slika 5.28. Unos novog računa

Račun broj: 1-TEST1-1

Prodavatelj:
Test d.o.o.
Vlasnik: Test Testić
OIB: 314141414141
Testna 22
23206, Test
0911111111
test@gmail.com

Kupac:
Testni Korisnik
OIB: 41241241441
Testna 33

Operator: Test Testić
Način plaćanja: Gotovina

Mjesto isporuke: Testno
Datum i vrijeme: 09/17/2017 21:31:06

Pregled računa					
Redni broj	Naziv	Jedinična cijena	Količina	Popust	Iznos
1	Testni artikl	66 KN	5	7 %	306.90 KN
				Osnovica	267.00 KN
				PDV	39.90 KN
				Ukupno	306.90 KN
Rekapitulacija poreza:					
Stopa poreza		Osnovica	Iznos poreza		
13 %		267.00	39.90		

Slika 5.29. Prikaz računa s PDV-om

Račun broj: 2-TEST1-1

Prodavatelj:
Test d.o.o.
Vlasnik: Test Testić
OIB: 314141414141
Testna 22
23206, Test
0911111111
test@gmail.com

Kupac:
Testni Korisnik
OIB: 41241241441
Testna 33

Operator: Test Testić
Način plaćanja: Gotovina

Mjesto isporuke: Testno
Datum i vrijeme: 09/17/2017 21:42:37

Pregled računa					
Redni broj	Naziv	Jedinična cijena	Količina	Popust	Iznos
1	Testni artikl	66 KN	5	7 %	306.90 KN
				Osnovica	306.90 KN
				Ukupno	306.90 KN

Slika 5.30. Prikaz računa bez PDV-a

Račun broj: 3-TEST1-1

Prodavatelj:
Test d.o.o.
Vlasnik: Test Testić
OIB: 314141414141
Testna 22
23206, Test
0911111111
test@gmail.com

Kupac:
Testni Korisnik
OIB: 41241241441
Testna 33

Operator: NOVI
Način plaćanja: Gotovina

Mjesto isporuke: Testno
Datum i vrijeme: 09/17/2017 21:44:25

Pregled računa					
Redni broj	Naziv	Jedinična cijena	Količina	Popust	Iznos
1	Testni artikl	-66 KN	-5	7 %	306.90 KN
				Osnovica	-306.90 KN
				Ukupno	-306.90 KN

Slika 5.31. Prikaz storniranog računa

ZAKLJUČAK

Domain-Driven Design je filozofija i pristup razvoju softvera koju je prvi definirao Eric Evans u svojoj knjizi "Domain-Driven Design: Tackling Complexity in the Heart of Software". DDD pristup pruža brojne strateške i taktičke principe kako bi se implementirao što kvalitetniji softver i kako bi njegovo održavanje bilo što jednostavnije.

Za svaki softver je najbitnije da rješava slučajeve korištenja, odnosno da rješava stvarne probleme problematične domene. Mnogi kompleksni softveri sadrže i kompleksne domene za koje programeri nemaju dovoljno znanja da bi ih implementirali. Zbog toga se posebno ističe u ovome radu važnost razgovora s domenskim stručnjacima koji su stručnjaci problematične domene i poznaju sva pravila i zamke koje sadrži. Razgovorima se postiže zajednički jezik koji je temelj pristupa DDD. Zajednički jezik omogućuje da se izradi što kvalitetniji dizajn domenskog modela u obliku primjerice UML dijagrama koji su opisani i tekstom koji sadrži ponašanje. U navedenom koraku kroz razgovore sve zainteresirane strane razmjenjuju znanja, pa tako programeri steknu znanja o poslovnim procesima dok se domenski stručnjaci približe modeliranju modela. Tek kada su gore navedeni koraci zadovoljeni, kreće se u implementaciju domenskog modela koda koji mora biti konstantno u sinkronizaciji s dizajnom domenskog modela i koji ga mora precizno preslikavati. Kada se radi implementacija modela u obliku koda potrebno se je služiti taktičkim principima koji olakšavaju apstrakciju problematične domene. Najbitnije je osigurati izoliranost domenskog modela pa tek onda koristiti slojevit arhitekturu za ostale slojeve kao što su aplikacijski, infrastrukturni i prezentacijski. Pri tome je važno osigurati da domenski sloj ne zna ništa o vanjskim slojevima.

Pristup DDD-a je korišten u okviru priložene aplikacije za izdavanje računa gdje je prikazan tijek razvoja aplikacije od uspostavljanje zajedničkog jezika, dizajna domenskog modela pa sve do implementacije domenskog modela u obliku koda. Opisane su također sve korištene tehnologije implementacije kao što su: programski jezik PHP, radno okruženje Symfony, Doctrine ORM i MySQL relacijska baza podataka. Korištena je slojevita arhitektura kao temeljna arhitektura razvoja aplikacije. Također je korišteno i razvojno okruženje Symfony, ali samo kao alat prezentacijskog sloja koji znatno olakšava razvoj kroz brojne gotove komponente. Na taj način aplikacijski i domenski sloj izoliraju funkcionalnosti aplikacije. Na

taj način možemo vrlo lako zamijeniti grafičko sučelje s primjerice komandnim ili vrlo lako promijeniti radno okruženje. DDD pristup nije prikladan za svaki projekt, pa tako ukoliko domena nema puno ponašanja i aplikacija nema puno slučajeva korištenja tada DDD pristup samo usporava razvoj. Uvijek treba procijeniti kakav je softver i koliko će se u budućnosti razvijati.

LITERATURA

- [1] Evans, Eric: “Domain-Driven Design: Tackling Complexity in the Heart of Software“, Addison Wesley, 2003.
- [2] Buenosvinos, Carlos; Soronellas, Christian; Akbary, Keyvan: “Domain-Driven Design in PHP“, Rigor Books, 2016.
- [3] Millett, Scott; Tune, Nick: “Patterns, Principles, and Practices of Domain-Driven Design“, Wrox, 2015.
- [4] Avram, Abel; Marinescu, Floyd: “Domain-Driven Design Quickly“, Lulu.com, 2007.
- [5] Martin, Robert C, “The Clean Architecture“, s Interneta, <https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>, 13.08.2012.
- [6] PHP, „History of PHP“, s Interneta, <http://php.net/manual/en/history.php.php>
- [7] The Doctrine Project, “Doctrine 2 ORM 2 documentation“, s Interneta, <http://docs.doctrine-project.org/en/latest/#>
- [8] SensioLabs, “Symfony Documentation“, s Interneta, <http://symfony.com/doc/current/index.html>
- [9] Wikipedia, “MySQL“, s Interneta, <https://en.wikipedia.org/wiki/MySQL>
- [10] Zakon.hr, „Opći porezni zakon“, s Interneta, <https://www.zakon.hr/z/100/Op%C4%87i-porezni-zakon>
- [11] Zakon.hr, „Zakon o porezu na dodatnu vrijednost“, s Interneta, <https://www.zakon.hr/z/186/Zakon-o-porezu-na-dodanu-vrijednost>
- [12] Zakon.hr, „Zakon o porezu na dohodak“, s Interneta, <https://www.zakon.hr/z/85/Zakon-o-porezu-na-dohodak>
- [13] Zakon.hr, „Zakon o fiskalizaciji u prometu gotovinom, s Interneta, <https://www.zakon.hr/z/548/Zakon-o-fiskalizaciji-u-prometu-gotovinom>
- [14] NoeticForce, „PHP Frameworks: The best 10 for modern web development“, s Interneta, <http://noeticforce.com/best-php-frameworks-for-modern-web-development>
- [15] AgileCode, „Microsoft.NET O/R mapper:choose your own“, s Interneta, <https://www.agile-code.com/blog/microsoft-net-or-mapper-choose-your-own/>

POPIS OZNAKA I KARTICA

DDD	Domain Driven Design
MDD	Model Driven Design
PHP	Skriptni programski jezik (engl. <i>PHP: Hypertext Preprocessor</i>)
ORM	Uzorak za pretvaranje objekata u tablice relacijske baze (engl. <i>Object-relational mapping</i>)
UML	Ujedinjeni jezik za modeliranje (engl. <i>Unified Modelling Language</i>)
DIP	Princip obrtanja ovisnosti (engl. <i>Dependency Inversion Principle</i>)
MVC	Model-Pogled-Kontroler (engl. <i>Model-View-Controller</i>) uzorak
DQL	Doctrine jezik za upite (engl. <i>Doctrine Query Language</i>)
UI	Korisničko sučelje (engl. <i>User interface</i>)
CRUD	Unos, čitanje, ažuriranje, brisanje (engl. <i>Create, read, update, delete</i>)
XML	Jezik za označavanje podataka (engl. <i>EXtensible Markup Language</i>)

SAŽETAK

U ovome diplomskom radu u potpunosti je opisan koncept i pristup razvoju softvera poznat kao Domain-Driven Design. Opisane su detaljno sve karakteristike i principi takvog pristupa razvoja softvera te njegove prednosti i nedostaci.

Osoba koja je osmislila i najviše pridonijela populaciji ovoga pristupa je softverski arhitekt Eric Evans koji DDD pristup detaljno opisuje u svojoj poznatoj knjizi „Domain-Driven Design: Tackling Complexity in the Heart of Software“ koja je osnovna literatura na kojemu se ovaj rad temelji.

Opisano je rješenje problematične domene izdavanja fiskalnih računa primjenjujući sve strateške i taktičke principe navedenog pristupa. Pokazan je tijek razvoja softvera: od početnog razgovora, uspostavljanja zajedničkog jezika, dizajniranja domenskog modela pa sve do implementacije domenskog modela u model koda. Nakon implementacije domene korištena je popularna slojevita arhitektura kao osnova za izoliranje domenskog sloja od ostatka aplikacije. Na kraju je pokazano kako se jedno PHP radno okruženje poput Symfony-a uklapa u projekt kao osnova prezentacijskog sloja aplikacije. Naglašena je potrebna izolacija aplikacije od jednog takvog okruženja koji u okviru web aplikacije služi samo i jedino kao alat koji nam olakšava prezentaciju.

KLJUČNE RIJEČI: Web aplikacija, Domain-Driven Design, Slojevita arhitektura, PHP, Doctrine ORM, Symfony

TITLE: WEB APPLICATION DEVELOPMENT USING DOMAIN DRIVEN DESIGN APPROACH

SUMMARY

In the graduate thesis, the concept and approach to developing a software known as Domain Driven Design is fully described. All the characteristics and principles of such software development and its advantages and disadvantages are described in detail.

The person who devised and contributed most to the population of this approach is the software architect Eric Evans who describes the DDD approach in his famous book „Domain-Driven Design: Tackling Complexity in the Heart of Software“ which is the basic literature of this graduate thesis.

A solution to the problematic domain of issuing fiscal invoices is described, applying all the strategic and tactical principles of this approach. The software development trend is demonstrated: from initial conversation, establishing a ubiquitous language, designing a domain model, and implementing domain model in the code model. After the domain development, the popular layered architecture was used as the basis for isolating the domain layer from the rest of the application. In the end, it has been shown how a PHP framework like Symfony fits into the project as the basis of the presentation layer. It is emphasizes the need to isolate the application from one such environment, which within the web application only serves as the only tool that facilitates the presentation.

KEYWORDS: Web application, Domain-Driven Design, Layered architecture, PHP, Doctrine ORM, Symfony