SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

DIPLOMSKI RAD

DIZAJN UPRAVLJAN DOMENOM

Marko Mihael Maras

Split, rujan 2023.

ZAHVALA

Zahvaljujem svom mentoru, prof. dr. sc. Saši Mladenoviću za svaki savjet, pomoć i vodstvo ne samo pri izradi ovog diplomskog rada već i kroz sve godine studiranja.

Posebne zahvale mojoj obitelji na podršci koju su pružili te svojoj požrtvovnosti.

Na kraju hvala i svim kolegama, prijateljima i najmilijima koji su dane studiranja učinili lijepim i zabavnim.

Temeljna dokumentacijska kartica

Diplomski rad

Sveučilište u Splitu
Prirodoslovno-matematički fakultet
Odjel za informatiku
Ruđera Boškovića 33, 21000 Split, Hrvatska

DIZAJN UPRAVLJAN DOMENOM

Marko Mihael Maras

SAŽETAK

Rad prikazuje pristup razvoju programske podrške koristeći dizajniranje usmjereno na domenu. Principi dizajniranja usmjerenog na domenu su prikazani kroz primjer sustava za iznajmljivanje vozila. Za izradu programske podrške bilo je potrebno razumjeti poslovnu domenu i njezinu funkcionalnost, a za izradu modela kompleksne domene stvoriti sveprisutni jezik kao temelj komunikacije. U izradi programske podrške korištene su komponente kao što su agregati, entiteti i vrijednosni objekti. Za stvaranje programske podrške kao cjeline korištena je čista arhitektura u kojoj svaki sloj ima određenu funkcionalnost. Zahtjevi sa viših slojeva se obrađuju koristeći biblioteku MediatR koja radi po principu CQRS-a, a podaci se nalaze u relacijskoj bazi podataka te se manipuliraju koristeći EntityFrameworkCore. Cilj rada bio je uočiti prednosti i nedostatke DDD-a kroz konkretan primjer, te su isti izneseni u zaključku.

Ključne riječi: DDD, Agregati, Entiteti, Vrijednosni objekti, Čista arhitektura, MediatR, CQRS, Sveprisutni jezik, Web API, C#, Objektno-relacijsko mapiranje, Entity Framework Core

Rad je pohranjen u knjižnici Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu

Rad sadrži: 64 stranica, 40 grafičkih prikaza i 11 literaturnih navoda. Izvornik je na hrvatskom jeziku.

Mentor: **Dr. sc. Saša Mladenović,** *izvanredni profesor Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Ocjenjivači: **Dr. sc. Saša Mladenović,** *izvanredni profesor Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Dr. sc. Divna Krpan, *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Dino Nejašmić, *predavač Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Rad prihvaćen: **rujan 2023**

Basic documentation card

Master Thesis

University of Split
Faculty of Science
Department of Informatics
Ruđera Boškovića 33, 21000 Split, Croatia

DOMAIN DRIVEN DESIGN

Marko Mihael Maras

ABSTRACT

This thesis presents an approach to the software development using Domain Driven Design. The principles of DDD are illustrated through the example of a car rental software. To create software support it was necessary to understand the business domain and its functionality, and to create a complex domain model it was necessary to create ubiquitous language as the basis of communication. Components such as aggregates, entities and value objects were used in the development of the software. To create the software as a whole, a clean architecture was used in which each layer has a certain functionality. Requests from higher layers are processed using the MediatR library, which works according to the CQRS principle, and the data is located in a relational database and is manipulated using EntityFrameworkCore. The aim of the paper was to observe the advantages and disadvantages of DDD through a concrete example, and they are presented in the conclusion.

Key words: DDD, Aggregates, Entities, Value objects, Clean Arhitecture, MediatR, CQRS, Ubiquitous language, Web API, C#, Object–Relational Mapping, Entity Framework Core

Thesis deposited in library of Faculty of science, University of Split

Thesis consists of: 64 pages, 40 figures and 11 references

Original language: Croatian

Mentor: **Saša Mladenović, Ph.D.** Associate *Professor of Faculty of Science, University of Split*

Reviewers: **Saša Mladenović, Ph.D.** Associate *Professor of Faculty of Science, University of Split*

Divna Krpan, Ph.D. Assistant *Lecturer of Faculty of Science, University of Split*

Dino Nejašmić, Ph.D Lecturer *of Faculty of Science, University of Split*

Thesis accepted: **September 2023**

IZJAVA

kojom izjavljujem s punom materijalnom i moralnom odgovornošću da sam diplomski rad s naslovom *Dizajn upravljan domenom* izradio samostalno pod voditeljstvom dr. sc. Sašom Mladenovićem. U radu sam primijenio metodologiju znanstvenoistraživačkog rada i koristio literaturu koja je navedena na kraju diplomskog rada. Tuđe spoznaje, stavove, zaključke, teorije i zakonitosti koje sam izravno ili parafrazirajući naveo u diplomskom radu na uobičajen, standardan način citirao sam i povezao s fusnotama s korištenim bibliografskim jedinicama. Rad je pisan u duhu hrvatskog jezika.

Student

Marko Mihael Maras

Sadržaj

Uvod

U svijetu koji nas okružuje možemo vidjeti koliko naš svakodnevni život ovisi o tehnologiji. Tehnologija se svakodnevno usavršava i od svih zanimanja ima najviši rast u povijesti čovječanstva. Svijet u kojem živimo je pun različitih problema, malih i velikih, poslovnih, financijskih, životnih... U svakom od područja života postoje problemi, problemi za koje pokušavamo pronaći rješenje. Svako rješenje opet nije primjenjivo na sve probleme, ali ono što možemo je pronaći uzorke, sličnosti ili korake u pristupu rješavanju svakog problema.

Na programiranje i strukturiranje rješenja možemo gledati kao na umjetnost. Sredstva za slikanje i za izradu sustava su u suštini ista, ali ovisno o tome tko ih koristi možemo dobiti remek djelo rješenja kojem će se svi diviti i poželjeti u svojoj kolekciji odnosno poslovanju. Kako bi postigli adekvatno i bogato rješenje potrebno je koristiti određena pravila, savjete, principe i strukture.

U ovom radu će biti prikazan i objašnjen jedan od popularnih pristupa u izradi programske podrške za rješavanje poslovnih problema. Pristup o kojem će se govoriti je Dizajn upravljan domenom (engl. *Domain Driven Design*, skraćenica DDD). DDD je koncept koji pokušava ubrzat razvoj programske podrške vodeći brigu o značajkama kompleksne domene za koju se kreira. Ovaj pristup predstavio je Eric Evans u svojoj knjizi *Domain-Driven Design: Tackling Complexity in the Heart of Software,* a na osnovu nje su se uključili i mnogi drugi koji su uvidjeli prednosti ovog pristupa te je DDD postao temeljem mnogih sustava.

Glavna značajka DDD pristupa je razvoj programske podrške zasnovane na domeni, a sami razvoj domene podrazumijeva stalnu komunikaciju domenskih stručnjaka i razvojnih stručnjaka kako bi se stvorila što preciznija preslika stvarne domene u programsku domenu. Faze razvoja programske podrške zasnovane na DDD-u su iterativne te je bitno početne faze dobro proučiti kako bi daljne faze bile podložne što manjim promjenama i refaktoriranju.

Tehnike, procesi i gradivni blokovi DDD-a bit će prikazani kroz projektni primjer iznajmjivanja vozila. Sama problematika i poslovna domena bit će objašnjena u nastavku rada.

1. Dizajn upravljan domenom

Razvoj softvera se koristi kako bi se automatizirao određeni proces koji postoji u realnom svijetu ili kako bi se pružilo rješenje poslovnog problema. Samim time softver je izveden i duboko povezan sa domenom. Pošto je softver sačinjen od kôda može se dogoditi da se previše vremena utroši na kôd te se pogled na softver vidi samo kroz objekte i metode. Takav način rada moguć je kad su u pitanju jednostavni slučajevi i primjeri, ali ne možemo na taj način kreirati kompleksne softvere. (Avram & Marinescu, 2006)

Kako bi kreirali dobar softver potrebno je znati što taj softver treba raditi. Nije moguće kreirati bankovni softver bez da se razumije kako banka funkcionira, a kako bi to bilo moguće treba razumjeti domenu banke. Stoga ako se postavi pitanje je li moguće kreirati kompleksni softver nekog poslovanja bez znanja o toj poslovnoj domeni, odgovor je ne. Postavlja se pitanje kako doći do tog znanja domene i tko ga uopće posjeduje? Najbolji poznavaoci određene poslovne domene su nitko drugi nego osobe koje rade u toj poslovnoj domeni, a njih nazivamo domenskim stručnjacima. Ti stručnjaci znaju svaki detalj, probleme i pravila. Stoga je početak razvoja programske podrške domena.

Cilj softvera je preslikati određenu domenu, a kako bi to bilo moguće softver mora biti u skladu sa domenom za koju se izrađuje inače može doći do deformacije u domeni što dovodi do kvarova, kaosa i štete. Kako ne bi došlo do tog scenarija softver mora koristit glavne koncepte i elemente domene.

Domena je nešto iz stvarnog svijeta te ju nije tako lako prenijeti u programski kôd stoga je potrebno stvoriti apstrakciju domene. (Avram & Marinescu, 2006) Apstrakcija nije ništa drugo nego model, model domene. (Avram & Marinescu, 2006) Prilikom razvoja programske podrške dosta toga se može naučiti od domenskih stručnjaka, ali čak ni to znanje neće biti tako lako prenijeti u programski kôd ako nije stvoren nacrt domene u glavi. Kao i na svakom početku, slika domene nije najjasnija, ali kroz vrijeme dok se koristi i razmišlja o njoj postaje jasnija i razumljivija.

Prema Ericu Evansu, model domene nije specifični dijagram nego je ideja koju neki dijagram treba prenijeti. To nije samo znanje domenskih stručnjaka nego organizirana i odabrana apstrakcija tog znanja. Dijagram može predstavljati model kao i pažljivo napisan kôd ili rečenica na nekom jeziku.

Model je pojednostavljena slika stvarne domene u kojoj se ističu najvažnija svojstva te domene. Prilikom stvaranja modela domene postoje određeni izazovi. Potrebno je organizirati i sistematizirati sve informacije te ih podijeliti u manje dijelove, a opet i te dijelove grupirati u određene logičke cjeline. Također jedan od izazova je i izbacivanje ili izuzimanje dijelova domene zbog njihovog viška jer domena sadrži ogroman broj informacija.

Model je esencijalni dio dizajna softvera. Treba nam kako bi se moglo nositi sa složenošću. O modelu treba pričati sa domenskim stručnjacima, kolegama dizajnerima i razvojnim programerima. Model je srž softvera, ali treba imati načine kako ga izraziti i pričati o njemu s drugim ljudima. Znanje i informacije treba dijeliti na točan, kompletan i nedvosmislen način. (Avram & Marinescu, 2006)

Postoji razlika u dizajniranju softvera i dizajniraju kôda. Dizajniranje softvera se može analogno prikazati kao kreiranje kuće gdje se sve vrti oko šire slike (npr.: visina i širina zidova, broj prostorija...). S druge strane, dizajniranje kôda je rad s detaljima (npr.: lokacija slike ili ormarića). I jedan i drugi dizajn su jako važni, ali dizajn softvera je važniji. Pogrešku u dizajnu kôda je lakše ispraviti nego pogrešku na razini dizajna softvera. (Avram & Marinescu, 2006) Lakše je premjestiti sliku sa jednog zida na drugi nego srušiti dio kuće kako bi ga napravili na drukčiji način.

1.1. Modeli razvoja dizajna upravljanog domenom

U pristupu razvoju programske podrške koriste se različiti modeli koji imaju svoje prednosti i nedostatke. Svaki od modela se sastoji od četiri osnovne faze (Slika 1):

Slika 1 Osnovni model razvoja programske podrške

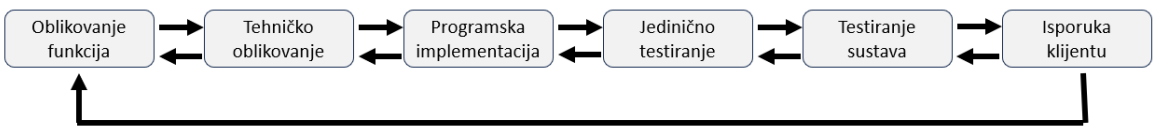
- Faza zahtjeva – prikupljanje zahtjeva koji su ključ za početak razvoja programske podrške. Ova faza zahtjeva odgovore na pitanja tko će i kako koristiti sustav, te kakvi su ulazno-izlazni podaci. Odgovorom na ta pitanja dobivaju se očekivane funkcionalnosti sustava

- Faza oblikovanja – oblikovanje sustava je rezultat zaključaka iz faze zahtjeva. U ovoj fazi stručnjaci programske podrške opisuju načine kojima će se ostvariti svaka od funkcionalnosti. Odabiru se alati koji će pomoći u izgradnji programske podrške (npr. UML dijagrami).
- Faza implementacije – kao što sam naziv govori, radi se o implementaciji rezultata faze oblikovanja koristeći programski kôd. Spada pod najdulju i fazu razvoja programske podrške. Za programere je ovo najvažnija faza jer kreiraju kôd koji će na kraju biti temelj rješavanja poslovnog problema.
- Faza testiranja/vrednovanja – programski kôd koji je napisan u fazi implementacije potrebno je testirati odnosno vrednovati kako bi se osiguralo da sve funkcionalnosti odgovaraju zahtjevima iz faze prikupljanja zahtjeva. Mogu se koristiti jedinični testovi (nad pojedinom komponentom) i testovi sustava (nad sustavom kao cjelinom).

Nakon što su sve faze dovršene, poslovno rješenje odnosno proizvod se isporučuju klijentu. Redoslijed izvršavanja ovih faza može biti različit te prema tome postoji i podjela na različite modele razvoja programske podrške. Za potrebe DDD-a koriste se iterativni razvoj (evolucijski model) te agilne metode, a uz njih još postoje izradi i popravi model, vodopadni model i v-model.

1.1.1. Iterativni razvoj ili evolucijski model

U ovom modelu uzimaju se u obzir povratne informacije svake faze i njenog prethodnika. Nakon što se kreira jedna verzija sustava, klijent je ocjenjuje te se na osnovu mišljenja i primjedbi kreira se nova verzija koja nije ništa drugo nego točnija verzija prethodne. Poželjni način korištenja ovog modela je inkrementalni iterativni način što znači da se razvoj vraća na prvu fazu razvoja nakon što se primjene i obrade informacije iz prethodne faze. Također rezultat svake iteracije je iskoristiva verzija programske podrške (Slika 2)



Slika 2 Iterativni model

Prednosti evolucijskog modela razvoja su:

- Razvoj moguć bez obzira na moguće nejasnoće u zahtjevima
- Proizvodi brza rješenja na klijentove zahtjeve
- Specifikacije se postupno usavršavaju s obzirom na klijentovo razumijevanje (Barkijević, 2021)

Nedostatci evolucijskog modela razvoja su:

- Teško ili nemoguće ocijeniti koliko posla je odrađeno te kada bi projekt mogao biti gotov
- Moguće je da izlazni sustav bude loše strukturiran i težak za održavanje obzirom na stalne promjene (Barkijević, 2021)

1.1.2. Agilne metode

Agilni razvoj programske podrške je skupina razvojnih metodologija koja se temelji na iterativnom i inkrementalnom razvoju. (Agile Software Development, 2023) Zahtjevi i rješenja se povezuju kroz suradnju između samoorganizacije i razvojnih timova. Cilj agilnih metoda je razviti programsku podršku u kratkom vremenskom periodu, pri tome je svaka iteracija samostalni projekt. Komunikacija je jedan od ključeva ovih metoda te se izvršava u realnom vremenu, a najmanji tim tvore programer i klijent. Iz ovoga je vidljivo da DDD koristi agilne metode jer jedna od bitnih značajki DDD-a je komunikacija domenskih stručnjaka sa stručnjacima programske podrške. Ovaj dio komunikacije će biti detaljnije objašnjen u nastavku rada kroz Sveprisutni jezik. Glavni pokazatelj napretka projekta je program koji radi.

Procesi koji se nalaze u agilnim metodama dobro odgovaraju DDD-u. Jedan od osnovnih agilnih procesa je provjera i prilagodba. Osnovna ideja je da se napravi mala promjena koja se pusti u korištenje. To što je napravljeno i objavljeno se ocjenjuje u razgovorima sa krajnjim korisnicima ili klijentom te na osnovu tih informacija se rade preinake, a proces se ponavlja. Kako bi se nešto smatralo agilnim najvažnije je da se krajnjem korisniku što prije predstavi mali dio sustava kojeg može koristiti i kojeg se može usavršavati na osnovu povratnih informacija.

Agilne metode se dijele na:

- Ekstremno programiranje (engl. *Extreme Programming*, skraćenica XP)
- Scrum
- Razvoj usmjeren na značajke
- RUP (engl. *The Rational Unified Process*)
- Metoda dinamičkog razvoja sistema
- Adaptivni razvoj softvera

Za potrebe DDD-a koristi se agilna metoda ekstremnog programiranja koja će biti objašnjena, a sve informacije o drugim agilnim metodama mogu se pronaći u knjizi *Agile Software Development Methods: Review and Analysis* (Abrahamsson, 2017)

1.1.2.1 Ekstremno programiranje

Ekstremno programiranje razvilo se iz problema uzrokovanih dugima razvojnim ciklusima tradicionalnih modela razvoja. U početku je bila ideja da se posao što prije napravi koristeći prakse koje su se pokazale učinkovite u procesima razvoja softvera. Samo naziv „ekstremno programiranje“ dolazi od svih kombiniranih praksi i principa koje su podignute na ekstremnu razinu (Abrahamsson, 2017).

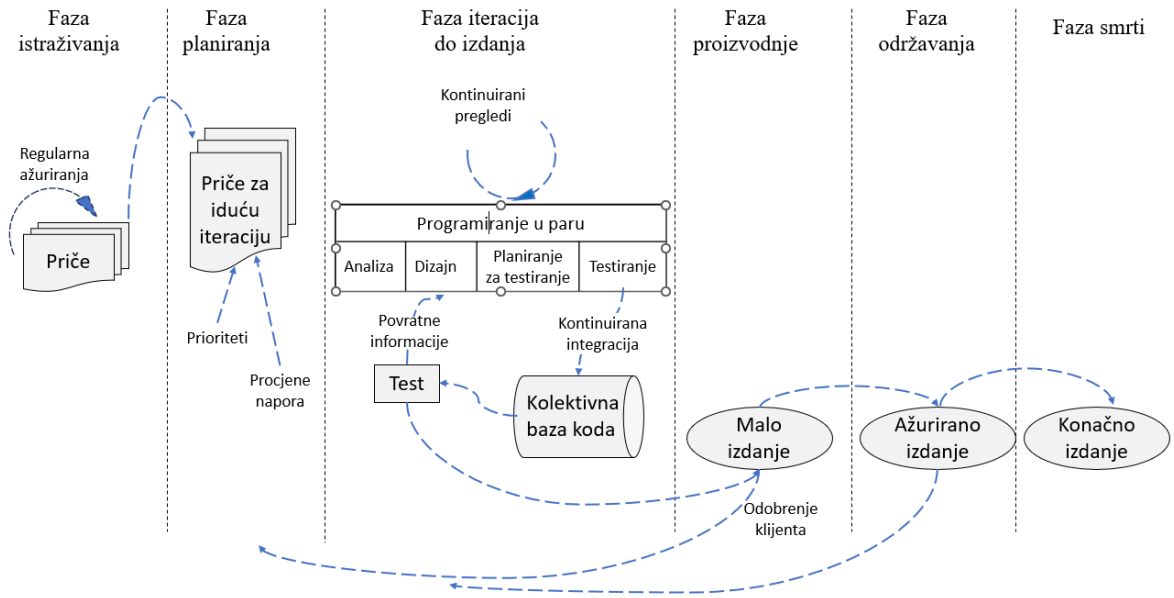
Ekstremno programiranje je skup ideja i praksi koje su izvučene iz već postojećih metodologija. Važno je naglasiti da klijent donosi poslovne odluke dok programeri odlučuju o tehničkim problemima. Ekstremno programiranje omogućava uspješni razvoj programske podrške bez obzira na nejasne ili promjenjive zahtjeve.

Glavne karakteristike ekstremnog programiranja su:

- Kratke iteracije sa malim verzijama sustava
- Brze povratne informacije
- Klijentovo sudjelovanje (cijelo vrijeme dostupan i prisutan za tim)
- Komunikacija i koordinacija
- Neprekidna integracija i testiranje
- Ograničena dokumentacija
- Jednostavan dizajn

- Kolektivno vlasništvo (bilo tko, bilo kada može promijeniti bilo koji dio kôda)
- Programiranje u paru (dvije osobe programiraju na jednom računalu)

Životni ciklus XP-a se sastoji od pet fazi (Slika 3):



Slika 3 Životni ciklus procesa ekstremnog programiranja

- Faza istraživanja – klijenti pružaju zahtjeve koje žele u prvoj verziji. Svaki zahtjev predstavlja značajku programa. U ovoj fazi se i razvojni programeri upoznavaju sa alatima, tehnologijama i praksama koje će se koristiti u projektu. Ova faza može trajati između par tjedana pa sve do par mjeseci ovisno o tome koliko su programeri upoznati sa tehnologijom.
- Faza planiranja – postavlja se raspored prioriteta i donosi se dogovor o sadržaju prve verzije programa. Programeri procjenjuju koliko će biti zahtjevan svaki od zahtjeva te se na osnovu toga postavlja raspored. Raspored za prvu verziju ne bi trebao prekoračiti dva mjeseca, a sama faza planiranja traje par dana.
- Faza iteracija do izdanja – uključuje nekoliko iteracija sustava prije izdanja prve verzije. Raspored iz faze planiranja se dijeli u nekoliko iteracija za koje će trebati između jednog i četiri tjedna za implementaciju. Prva iteracija kreira sustav sa arhitekturom cijelog sustava. Ovo se postiže uzimajući zahtjeve koji će izgraditi cijeli sustav. Klijent odabire zahtjeve koji će se izvršiti u svakoj od iteracija. Na kraju zadnje iteracije sustav je spreman za produkciju.

- Faza proizvodnje – uključuje testiranje i provjere performansi sustava prije nego što se sustav dostavi klijentu. U ovoj fazi mogu se pronaći nove promjene te treba donijeti odluku hoće li se te promjene nalazi u trenutnoj verziji. Ideje koje su ostavljene za kasnije treba dokumentirati za kasniju implementaciju, na primjer za fazu održavanja.
- Faza održavanja – nakon što je klijent dobio prvu verziju sustava potrebno je da tu verziju klijent može koristiti, a u isto vrijeme treba raditi na novim iteracijama. Faza održavanja zahtjeva zadatke korisničke podrške. U ovoj fazi moguće je da dođe do sporijeg razvoja. Također, faza održavanja može zahtijevati uključivanje novih osoba u timove te promjenu strukture timova.
- Faza kraja ili smrti – do ove faze dolazi kada klijent nema više zahtjeva za implementaciju. To znači da je sustav zadovoljio sve zahtjeve klijenta. U tom trenutno potrebno je napravit sve potrebne dokumentacije sustava pošto više nema promjena u arhitekturi, dizajnu ili kôdu. Do ove faze također može doći ako sustav ne ispunjava zahtjeve klijenta ili ako postane skup za daljnje razvijanje.(Abrahamsson, 2017)

2. Sveprisutni jezik

Kako bi se model domene razvio kako treba apsolutno je potrebno da stručnjaci programske podrške rade zajedno sa domenskim stručnjacima. Dakle, s jedne strane postoje stručnjaci programske podrške te po samom opisu to su stručnjaci koji imaju znanje i iskustvo za korištenje različitih programskih alata, paradigmi, pravila i slično te međusobno koriste specifični jezik za komunikaciju koji prelazi u žargon razumljiv samo ljudima te struke. Ovi stručnjaci ne znaju pravila određenog poslovanja odnosno domene za koju treba napraviti programsko rješenje. Svojom vlastitom logikom i razmišljanjem mogu donekle zaključiti kako određeno poslovanje funkcionira, ali to znanje je nedovoljno da bi pokrili sve elemente domene. S druge strane postoje domenski stručnjaci, osobe koje vode određeni posao te znaju svaki detalj i pravilo poslovanja, ali ne koriste terminologiju niti znanje za razvoj programske podrške. Oni također međusobno razvijaju svoj jezik za komunikaciju odnosno žargon. Kako bi se što bolje stvorila slika domene ova dva tipa stručnjaka moraju surađivati, ali na samom početku suradnje postoje komunikacijske barijere preko kojih je potrebno prijeći.

Kako bi se ta komunikacijska razlika riješila kod kreiranja modela potrebno je komunicirati da bi razmijenili ideje o modelu, elementima koji su u modelu, kako ih povezujemo, što je bitno, a što nije. Komunikacija na ovoj razini je ključ uspjeha projekta. Ako jedna osoba kaže nešto, a druga osoba ne razumije, ili još gore, razumije nešto drugo, kolike su šanse da projekt uspije? (Avram & Marinescu, 2006)

Terminologija svakodnevnih rasprava odvojena je od terminologije ugrađene u kôd (u konačnici najvažniji proizvod projekta programske podrške). Čak i osoba koristi različit jezik u razgovoru i u pisanju, tako da se najpronicljiviji izrazi domene često pojavljuju u prolaznom obliku koji nikada nije zarobljen u kôdu ili čak u pisanju. (Avram & Marinescu, 2006)

Kroz ove sesije razgovora, često se koristi prevoditelj kako bi drugi razumjeli što znače neki koncepti. Programeri će pokušati objasniti neke obrasce dizajna laičkim jezikom, ali nekad će to biti bezuspješno. Stručnjaci domene će težiti usvajaju njihovih ideja i time kreirati novi žargon. Tijekom ovog procesa komunikacija trpi, a ova vrsta prijevoda ne pomaže izgradnji znanja. (Avram & Marinescu, 2006)

Osobe uključene u izgradnju modela su sklone koristiti vlastiti dijalekt u ovim sesijama, ali niti jedan od tih dijalekata ne može biti zajednički jezik jer niti jedan ne služi potrebama svih. Zaključak je da se mora pričati istim jezikom kad god se sudionici susretu i pričaju o modelu i kako ga definirati, ali se postavljaju pitanja koji će to jezik biti? Jezik stručnjaka programske podrške? Jezik stručnjaka domene? Nešto između?

Centralni princip DDD-a je korištenje jezika zasnovanog na modelu. S obzirom da je model zajedničko područje, mjesto gdje programska podrška susreće domenu, primjereno ga je koristiti kao temelj jezika. (Avram & Marinescu, 2006)

Pošto se na model gleda kao na okosnicu jezika potrebno je zahtijevati da timovi koriste ovaj jezik u svim komunikacijama, a također i u kôdu. Baš iz razloga što je jezik prisutan u svakom dijelu procesa izgradnje modela taj jezik zovemo sveprisutni jezik. Sveprisutni jezik spaja sve dijelove dizajna i kreira premisu za razvojni tim kako bi dobro funkcionirao.

Kao i svjetski jezici, jezik korišten u razvoju modela ne nastaje preko noći. Potrebno je uložiti dosta posla i truda kako bi se prikazali ključni elementi jezika. Potrebno je pronaći ključne koncepte koji definiraju domenu i dizajn te im pridodijeliti odgovarajuće riječi i izraze. Neke elemente je jednostavno prepoznati, ali neke puno teže te je potrebno dosta vremena da se pronađu i definiraju na pravi način.

Izgradnja jezika ima jasan rezultat: model i jezik su usko povezani jedno s drugim. Promjena u jeziku bi trebala biti i promjena u modelu. Stručnjaci za domene bi se trebali usprotivi terminima ili strukturama koje su nespretne ili neadekvatne za prenošenje razumijevanja domene. Ako stručnjak za domenu ne može razumjeti nešto u modelu ili jeziku onda vrlo vjerojatno postoji problem u trenutnom modelu odnosno jeziku. S druge strane, stručnjaci za programsku podršku bi trebali paziti na nejasnoće ili nedosljednosti koje će se pojaviti u dizajnu. (Avram & Marinescu, 2006)

2.1. Kreiranje sveprisutnog jezika

Postavljaju se pitanja: Kako razviti sveprisutni jezik? Koje su metode i procesi u njegovom razvoju? Tko su osobe koje ga razvijaju?

Razvoj sveprisutnog jezika je ovisan o vremenu te se vremenom gradi i poboljšava. Naravno, kako bi se poboljšao potrebne su diskusije i analize postojećih dokumenata, riječnika i standarda.

Što se tiče samih metoda postoji podjela na četiri glavne, a to su:

- Crtanje
- Kreiranje riječnika
- Razmjena događaja (engl. *Event Storming*)
- Pregledavanje i ažuriranje

Metoda crtanja je zapravo izražavanje domene na ploči uz pomoć markera, nalijepnica i sličnih pomagala. U ovoj metodi nije bitno voditi brigu o formalnosti dizajna. Također, koristeći ovu metodu može se započeti izgradnja sveprisutnog jezika, ali ne može biti glavna referenca korištenja. Ovom metodom mogu se prikazati osnovni pojmovi te uvesti sve sudionike izgradnje programske podrške u problematiku kojom će se baviti.

Metodom kreiranja riječnika može se stvoriti prevoditelj svih izraza tako što će svaki izraz imati svoje značenje. Ova metoda je dobra kao pomoć formalnog oblika jezika jer objašnjava svaki pojam vezan za domenu, ali postoji nedostatak što ne vidimo interakciju pojmova i koncepata te time ograničavamo sliku domene. Jednom kreiran riječnik može služiti samo kao legenda pojmova prilikom čitanja složenijih metoda kreiranja sveprisutnog jezika.

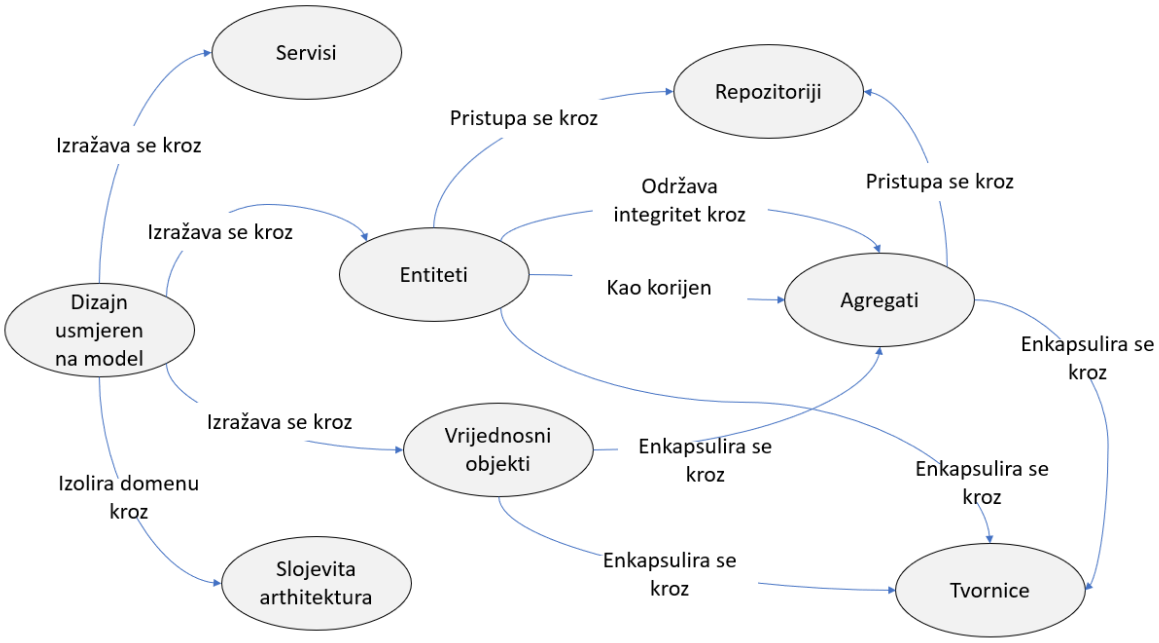
Koristeći metodu razmjene događaja domenski stručnjaci i stručnjaci za razvoj programske podrške mogu postići brz ciklus učenja poslovnih procesa što znatno olakšava razvoj sveprisutnog jezika.(Batista, 2019) Razmjena događaja je vizualni i interaktivni proces koji ohrabruje članove tima na suradnju i dijeljenje znanja kako bi se otkrili događaji u domeni te identificirala poslovna pravila. (Neto, 2023)

Metoda pregledavanja i ažuriranja se odnosi na spremnost da se na agilan način pregleda i ažurira ono što se do tad napravilo. (Batista, 2019)

3. Gradivni blokovi dizajna upravljanog domenom

Kako bi kompleksnu domenu implementirali u softver potrebno je primijeniti najbolje prakse modeliranja i dizajniranja. Određene odluke održavaju model i implementaciju u skladu jedno s drugim, a svaka odluka ojačava efektivnost druge odluke. Kako bi sve bilo u skladu potrebno je voditi brigu o detaljima svakog elementa.

Komponente koje se koriste u DDD-u predstavljaju najbolju praksu objektno-orijentiranog dizajniranja. DDD ističe modeliranje koncepata iz stvarnog svijeta, a to se dobro poklapa sa objektno-orijentiranim programiranjem koje u svom pristupu koristi enkapsulaciju podataka i ponašanja u objekte. Pažljivo kreiranje detalja individualnih elementa pruža razvojnim programerima stabilnu platformu prema kojoj mogu istraživati modele te ih prikazati kroz implementaciju. Komponente DDD-a su prikazane na Slika 4 te je vidljiva njihova povezanost i ovisnost. Svaka od komponenti ima svoju ulogu te pomaže u efektivnom modeliranju i implementaciji kompleksne poslovne domene.



Slika 4 Mapa koncepata i njihovih relacija

3.1. Slojevita arhitektura

Razvojni programeri često svoja programska rješenja pišu van konteksta neke softverske arhitekture, a bez dobro definirane softverske arhitekture programski kôd se dijeli u pakete te kao rezultat se dobije kolekcija neorganiziranog kôda koji ima manjak uloge, odgovornosti i međusobne ovisnosti. Ovakav način programiranja stvara problem pod nazivom „velika lopta blata“ (engl. *Big Ball of Mud*).

Ako programski kôd ne prati neku od formalnih arhitektura rezultat je čvrsto spojen, teško održiv i teško promjenjiv kôd bez jasne vizije ili puta ka cilju. Takav kôd je teško razumjeti bez saznanja o svim unutarnjim modulima i njihovom načinu rada i korištenja. (Richards, 2022)

Arhitektura softvera ima važnu ulogu u sposobnosti za skaliranjem i ispunjavanjem klijentovih zahtjeva kroz duži vremenski period. Ona pruža iskoristivi dizajn za različite situacije, brojne prednosti kao što su poboljšana efikasnost, produktivnost, brzina, cijena optimizacije i bolje planiranje. (Dhaduk, 2020)

Kod kreiranja programskog rješenja treba težiti savršenstvu, ali naravno tako nešto je jako teško ako ne i nemoguće s obzirom na razvoj tehnologije i promjene u zahtjevima klijenta. Kako bi se ipak u nekoj mjeri postiglo idealno programsko rješenje potrebno je pronaći odgovore na pitanja: Je li odabrana programska arhitektura skalabilna? Koje su karakteristike performansi aplikacije? Koliko brzo i jednostavno se mogu napraviti izmjene obzirom na promjene u zahtjevima? To su samo neka od pitanja koja mogu pomoći u odabiru prave arhitekture jer nije ni svaka arhitektura primjerena za svako programsko rješenje.

Neki od razloga zašto je programska arhitektura važna za aplikaciju su:

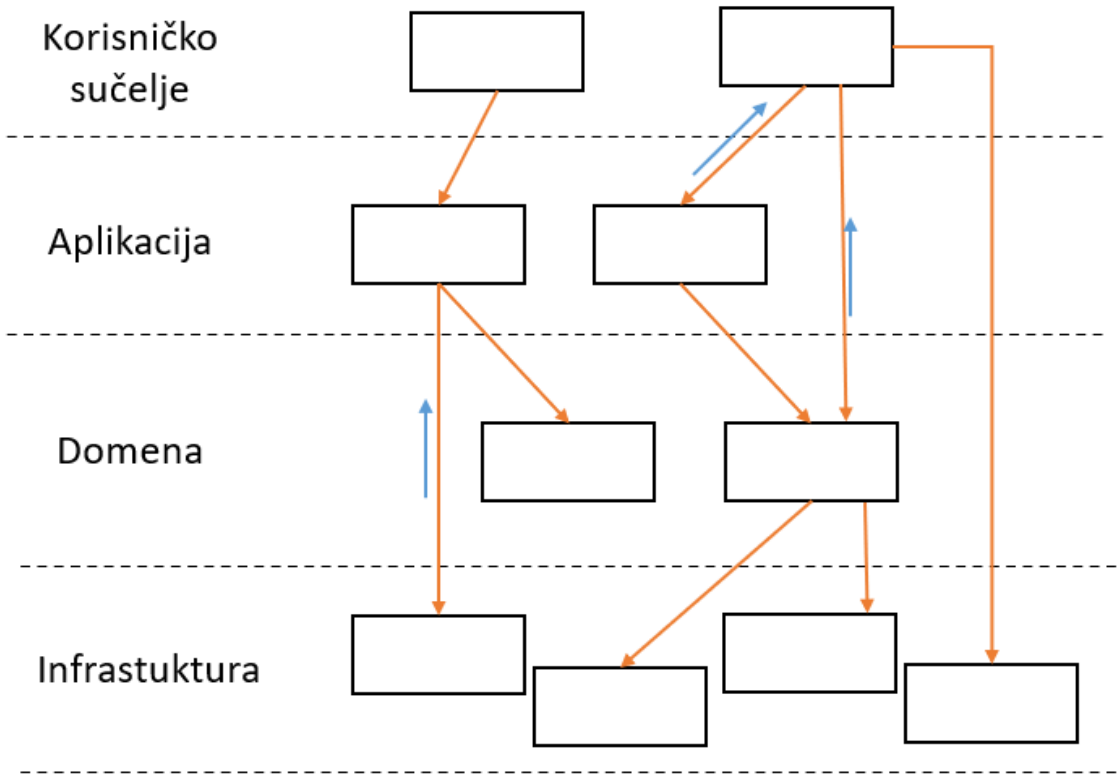
- Definiranje osnovnih karakteristika aplikacije – poznavanje karakteristika svake od arhitektura, prednosti i nedostataka su ključ u odabiru s obzirom na ciljeve poslovnog problema. Na primjer, neke arhitekture se koriste za visoko skalabilne aplikacije, a druge za agilne aplikacije
- Kvaliteta održavanja i efikasnosti – uvijek postoji vjerojatnost da će se aplikacija suočiti sa problemima kvalitete. Odabirom određene arhitekture mogu se smanjiti problemi kvalitete i efikasnosti kroz održavanje.

- Pružanje agilnosti – prirodno je da program prolazi kroz brojne modifikacije i iteracije za vrijeme razvoja, ali i nakon proizvodnje. Stoga, planiranje arhitekture pruža agilnost te olakšava buduće izmjene.
- Rješavanje problema – planiranje i poznavanje arhitektura pruža jasnu ideju kako će softver zajedno sa svim svojim komponentama funkcionirati. Razvojni tim se može prilagoditi najboljim praksama i riješiti kompleksne procese i buduće probleme.(Dhaduk, 2020)

Najčešće korištena arhitektura je slojevita arhitektura, još poznata i pod nazivom n-slojevita arhitektura. Komponente su organizirane u horizontalne slojeve, a svaki sloj ima svoju specifičnu ulogu unutar softvera. Sami naziv „n-slojevita arhitektura“ označava da broj slojeva nije ograničen niti strogo predodređen, ali ipak najčešća podjela je na četiri sloja:

- Prezentacijski
- Poslovni
- Infrastruktura
- Baza podataka

Slojevi koji su prethodno navedeni moraju biti u nekoj strukturi odnosno redosljedu te je potrebno znati njihovu međusobnu ovisnost. Njihova međusobna ovisnost je prikazana na Slika 5, a iz nje je vidljivo da svaki sloj ovisi o samom sebi ili o slojevima na nižoj razini.



Slika 5 Slojevi i njihove ovisnosti

Jedna od značajki slojevite arhitekture je razdvajanje odgovornosti(engl. *Separation of Concerns*) među komponentama. Sloj koji je nalazi na vrhu te o kome niti jedan sloj ne ovisi je prezentacijski sloj. Jedina odgovornost tog sloja je prikazivanje informacija i podataka korisniku te prihvaćanje njegovih naredbi. Prezentacijski sloj može ovisiti o svim slojevima koji se nalaze ispod njega jer o njima ovisi što će se prikazati korisniku. Idući sloj je aplikacijski sloj koji definira poslove koje softver treba napraviti, ali ne sadrži poslovnu logiku. Može služiti za komunikaciju s drugim sustavima i za koordinaciju zadataka te prosljeđivanja posla u slojeve ispod sebe. Poslovni sloj ili sloj domene je srce DDD-a. Taj sloj predstavlja poslovne koncepte, pravila i slučajeve, a modeli koji se nalaze u ovom sloju služe za transformaciju ili prema prezentacijskom sloju ili prema sloju infrastrukture. Na samom kraju je i već spomenuti sloj infrastrukture koji sadrži podršku za slojeve više razine kao što su poruke za aplikacijski sloj, pohranu i čitanje za domenski sloj, vizualne elemente za prezentacijski sloj itd.

Savjet za korištenje slojevite arhitekture u kontekstu DDD-a je da se izoliraju modeli domene i poslovna logika u poslovni sloj te se ukloni svaka ovisnost o ostalim slojevima što bi značilo da se poslovni sloj treba nalaziti u središtu te se sve gradi oko njega. Dizajn

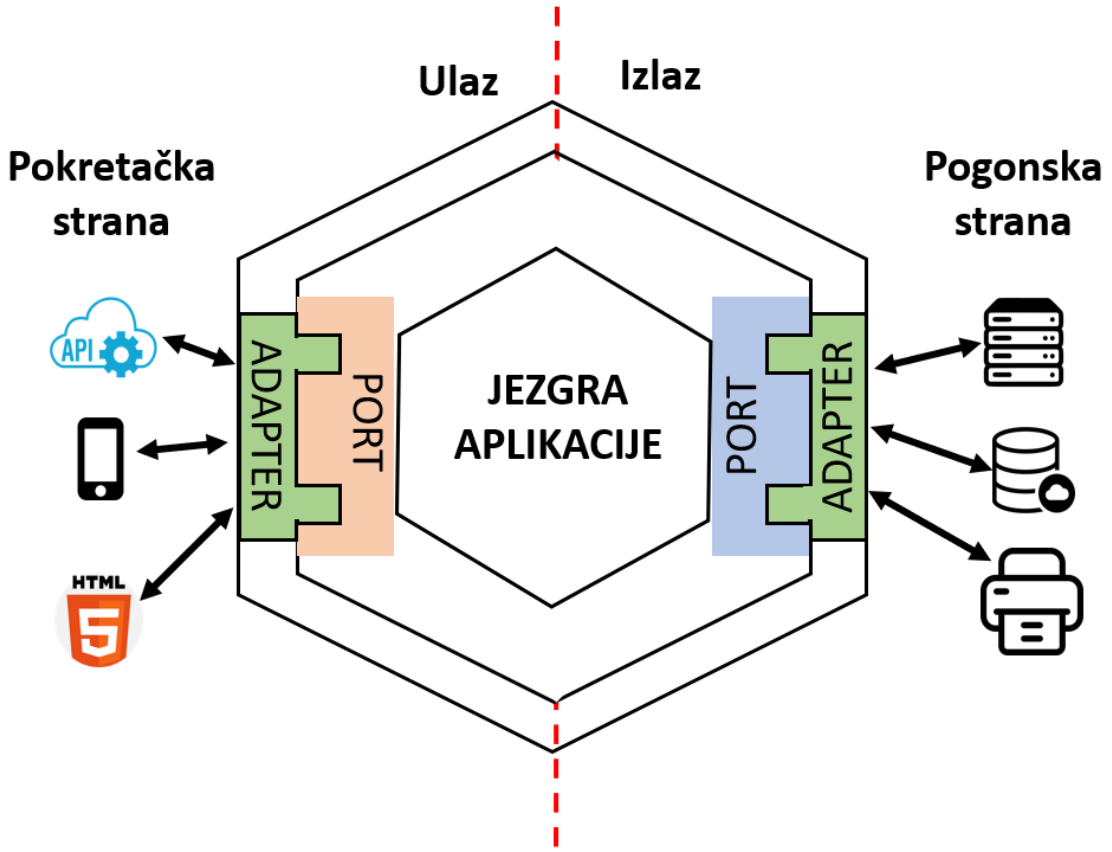
svakog sloja bi trebao biti kohezivan i ovisan samo o donjim slojevima. Poslovni sloj koji sadrži modele domene trebao bi sadržavati bogato i jasno poslovno znanje koje je moguće pravilno koristiti.

Kako bi sve ovo bilo moguće može se birati između tri tipa slojevite arhitekture, a to su čista arhitektura(engl. *Clean arhitecture*), heksagonalna arhitektura (engl. *Hexagonal arhitecture*) i onion arhitektura.

3.1.1. Heksagonalna arhitektura

Heksagonalna arhitektura je još poznata i pod nazivom arhitektura portova i adaptera. Osnovna ideja ove arhitekture je da se ulazi i izlazi postave na krajeve dizajna te bi se time izolirala centralna logika odnosno poslovna domena. Na ovaj način se omogućuje da se izlazi i ulazi koji se nalaze na krajevima mogu mijenjati bez da se mijenja poslovna domena. (Vuollet, 2018)

Osnova ove arhitekture su portovi i adapteri koji služe kao posrednici između unutarnjeg i vanjskog sloja. Sama struktura je vidljiva na Slika 6, a cilj je da se za svaki ulaz i izlaz kreira port. Port je zapravo samo apstrakcija, odnosno način na koji aplikacija može biti u interakciji sa vanjskim svijetom bez da zna s čim je točno u interakciji. To je u stvari ugovor odnosno sučelje koje definira aplikacija, a u njemu navodi na koji način želi komunicirati sa vanjskim svijetom. Portovi predstavljaju granice aplikacije. Na primjer aplikacija može imati port za komunikaciju sa bazom podataka umjesto da direktno radi sa njom. Na ovaj način aplikacija definira port koji ima sučelje za čitanje i pisanje. To sučelje je potrebno implementirati, a implementacije se rade u adapterima. Adapteri su zapravo pretvarači koji uzimaju izlaz aplikacije te ga pretvaraju u nešto što može koristiti vanjski izvor. U primjeru porta za čitanje i pisanje, aplikaciju nije briga hoće li podaci biti zapisani u bazu podataka, lokalne datoteke ili u redoslijed poruka. Adapter će uzeti podatke iz porta i pretvoriti ih u nešto što može biti zapisano u npr. Bazu podataka. Ključna stvar je da bez obzira što se promjeni u adapteru aplikacija sa svojim portom se nikada ne mijenja, odnosno promjena nema utjecaj na njih. Ovo je jako korisno jer razdvaja logiku aplikacije od baze podataka ili bilo kojih drugih vanjskih servisa. Ovaj uzorak se koristi i za ulaze u aplikaciju. Ulazni sudionici mogu biti API, red poruka, ali aplikaciju ne bi trebalo biti briga tko je pokreće jer bi ulaz definiran u portovima bi trebao biti isti. Aplikaciju možemo podijeliti na dvije strane: ulaznu i izlaznu. Ulaz pokreće aplikaciju, a izlaze pokreće sama aplikacija.



Slika 6 Struktura i komponente heksagonalne arhitekture

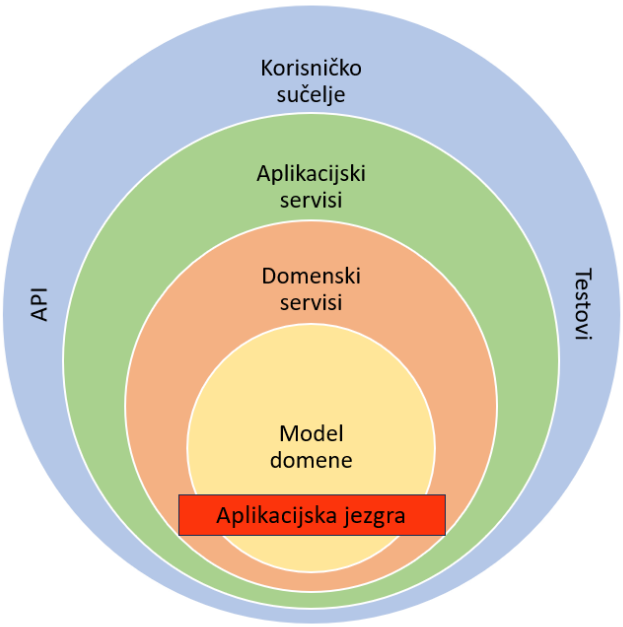
Prednosti ove arhitekture su testabilnost (sve je moguće testirati), održivost (zbog dobre izolacije lako je proširiti adaptere), identificiranje tehnološkog duga (lako izmjeriti kvalitetu kôda, trud i vrijeme). S druge strane kao nedostatak, ova arhitektura je kompleksna te traži visoku razinu timskog rada inače će se dobiti neuredan kôd, a niti jedna od prednosti neće biti zadovoljena.

3.1.2. Onion arhitektura

Cilj Onion arhitekture je izgradnja aplikacije sa boljom testabilnošću, održavanjem i pouzdanošću. Prvu ju predstavlja Jeffrey Palermo 2008. godine pokušavajući riješiti probleme održavanja aplikacijskih sustava, a pri tom naglašava princip razdvajanja odgovornosti. Prema njemu arhitektura se temelji na DDD-u i principu inverzne ovisnosti (engl. *Dependency Inversion Principle*, skraćenica DIP). Ima više slojeva u odnosu na prethodno opisanu heksagonalnu arhitekturu, a osnovno pravilo je da vanjski slojevi mogu ovisiti o središnjem sloju koji predstavlja domenu, ali središnji sloj ne smije imati nikakvu

ovisnost o slojevima oko sebe. Ukratko, na taj se način definira da ovisnosti idu izvana prema unutra.

Pošto je i ovo arhitektura koja se može koristiti za DDD, a prema tome domena je u središtu, tako i Onion arhitektura za svoje središte ima sloj domene u svojoj jezgri. Slojevi su prikazani na Slika 7, a središnji sloj je sloj domene u kojem se nalaze modeli i poslovna pravila njihovog korištenja, a predstavljaju temeljne gradivne blokove aplikacije. Iznad njega se nalazi sloj domenskih servisa čija je uloga održavanje logike domene i poslovnih pravila, a može sadržavati kompleksne izračune i algoritme koji su bitni za implementaciju slučajeva upotrebe (engl *Use Cases*) te ih koristi aplikacijski sloj. Domenski sloj i sloj domenskih servisa se mogu spojiti i u jedan, ali na ovaj način se poštuje princip razdvajanja odgovornosti. Iznad sloja domenskih servisa se nalazi aplikacijski sloj koji služi kao dodatni posrednik između sloja domene i vanjskog svijeta na način da ima definirane slučajeve korištenja. Aplikacijski sloj prima zahtjev koji obrađuje na način da orkestrira korake u provođenju, ali ne sadrži nikakvu poslovnu logiku. Prilikom obrađivanja nekog zahtjeva aplikacija prepoznaje o kojem je slučaju upotrebe riječ te poziva različite domenske servise i modele nakon čega vraća rezultat krajnjem korisniku. Obrađujući neki zahtjev ovaj sloj može biti u interakciji sa drugim servisima. Iznad aplikacijskog sloja je sloj infrastrukture, a zadužen je za interakciju sa vanjskim svijetom te ne rješava nikakve domenske probleme

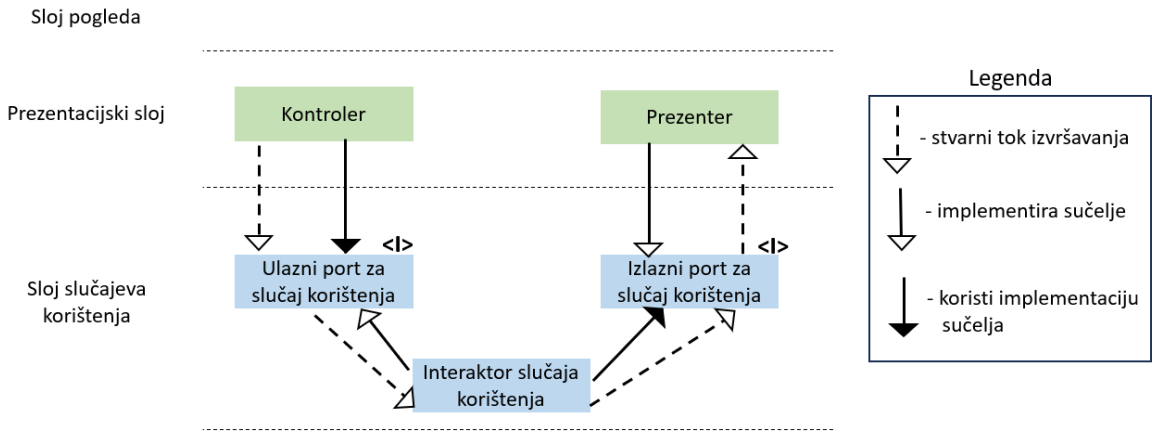


Slika 7 Struktura i slojevi Onion arhitekture

3.1.3. Čista arhitektura

Po uzoru na principe iz prethodne dvije arhitekture, 2012. godine Robert Cecil Martin (još poznat i kao Uncle Bob) je predstavio čistu arhitekturu koja je rezultat unaprjeđenja po pitanju izoliranosti, održivosti, jednostavnog testiranja i skalabilnosti. Ova arhitektura također zagovara ključne pojmove kao što su neovisnost o razvojnom okviru i tehnološkim okvirima, kretanje ovisnosti je od vanka prema unutra, slučajevi korištenja, princip inverzije ovisnosti, granice među slojevima i tehnike prosljeđivanja podataka među slojevima.

Kako bi se realiziralo pravilo ovisnosti izvana prema unutra koristi se inverzija kontrole (engl *Inversion of Control*, skraćenica IoC). Iako samo pravilo nije direktno povezano sa IoC, ali samim time što se primjenjuje pravilo ovisnosti, arhitektura je prisiljena koristiti IoC. Komponente čiste arhitekture prate tijek kontrole koji je vidljiv na Slika 8. Glavni ulaz u aplikaciju predstavlja nekakav događaj kojeg kontroler prepoznaje te poziva odgovarajuću metodu iz sučelja koje se nalazi kao ulaz u sloj slučajeva korištenja. Samim time zahtjev je prešao iz jednog sloja u drugi. U sloju slučajeva korištenja uzimaju se podaci iz zahtjeva te se prema njima pozivaju objekti i entiteti iz sloja domene. Nakon što je zahtjev obrađen rezultat je potrebno vratiti do prezentacijskog sloja. Kako bi to bilo moguće sloj slučajeva korištenja koji sadrži rezultat koristi sučelje koje je definirano kao izlaz prema prezentacijskom sloju te na taj način rezultat prenosi prezentacijskom sloju koji ga obrađuje i prosljeđuje do krajnjeg korisnika odnosno osobe koja je poslala zahtjev. Glavna ideja je da moduli više razine ne bi smjeli ovisi o tehnološkim detaljima, razvojnim okvirima, korisničkom sučelju itd.(Clean Arhitecture : Part 2 - The Clean Architecture, 2017) Inverzija kontrole, injekcija ovisnosti i inverzija ovisnosti predstavljaju pravila ovisnosti stoga je moguće zamijeniti implementaciju bilo kada. Ono što treba je poštovati sučelja koja su definirana u nižim modulima. Na ovaj način je jezgra aplikacije stabilna.



Slika 8 Tok izvršavanja koristeći inverziju kontrole

Nakon što je prikazano realiziranje jednog od glavnih obilježja čiste arhitekture, inverzija kontrole, potrebno je objasniti i razumjeti slojeve, tok izvršavanja te ovisnosti slojeva čiste arhitekture koji se mogu vidjeti na Slika 10 Primjer toka izvršavanja zahtjeva i ovisnost slojeva čiste arhitekture. Ako bi zanemarili krivulje i linije koje se nalaze na Slika 10 Primjer toka izvršavanja zahtjeva i ovisnost slojeva čiste arhitekture može se vidjeti da se čista arhitektura sastoji od pet koncentričnih slojeva.

Najvažniji sloj je sloj domene (sloj u samom središtu) te je stoga ova arhitektura jako pogodna za DDD. U ovom sloju se nalaze sva poslovna pravila zajedno sa modelima i metodama koje će održavati poslovnu logiku ispravnom. Pravila i modeli koji su definirani u ovom sloju su neovisni o vrsti tehnologije, ali su neovisna i o svim vanjskim slojevima. Modeli odnosno entiteti koji se ovdje nalaze ne služe samo za držanje podataka nego predstavljaju poslovne objekte koji učahuruju poslovna pravila. Ovaj sloj bi trebao biti otporan na vanjske promjene. Jako je bitno da je ovaj sloj ispravno napravljen jer pogreške u njegovom dizajniranju mogu rezultirati ozbiljnim posljedicama stoga se na izradu ovog sloja izdvaja jako puno vremena. Koncepti DDD-a imaju za cilj olakšati razvoj velikih poslovnih aplikacija koja se sastoje od mnoštva pravila i kompleksnosti.

Sloj iznad sloja domene je aplikacijski sloj ili sloj slučajeva korištenja. Ovaj sloj predstavlja popis aktivnosti i komunikacijskih koraka između sudionika i automatiziranog sustava koji su potrebni za izvršavanje zahtjeva. U načelu, ovaj sloj koji sadrži skup slučajeva korištenja je način na koji korisnik vidi funkcionalnosti sustava. Prilikom kreiranja ovog sloja koriste se dijagrami slučajeva korištenja. Što je dijagram detaljniji sa potrebnim operacijama to je lakše odabrati strategiju i metodologiju u razvoju arhitekture. Ovaj sloj je zadužen za orkestriranje entitetima iz sloja domene te sadrži poslovna pravila

specifična za samu aplikaciju. Aplikacijski sloj sažima i implementira sve slučajeve korištenja same aplikacije te s time aplikacija postaje automatizirani sustav. Za kreiranje ovog sloja potrebni su zahtjevi, a primjer jednog zahtjeva je vidljiv na Slika 9. Ovaj primjer predstavlja sve potrebne stvari kako bi se kreirao jedan slučaj korištenja. Vidljivo je koje podatke ovaj slučaj treba primiti kroz zahtjev, način na koji ih obrađuje putem validacije i kreiranja. Ako je kreiranje uspješno vraća se ID narudžbe, a u protivnom potrebno je vratiti poruku sa razlogom zašto nije bilo moguće napraviti narudžbu. U ovom zahtjevu nije definirano o kojem tipu baze podataka se radit, a ni o tome kroz što se šalje ovaj zahtjev ili na što će biti vraćen rezultat (mobilna aplikacija, API,...). Za aplikacijski sloj to uopće nije bitno.

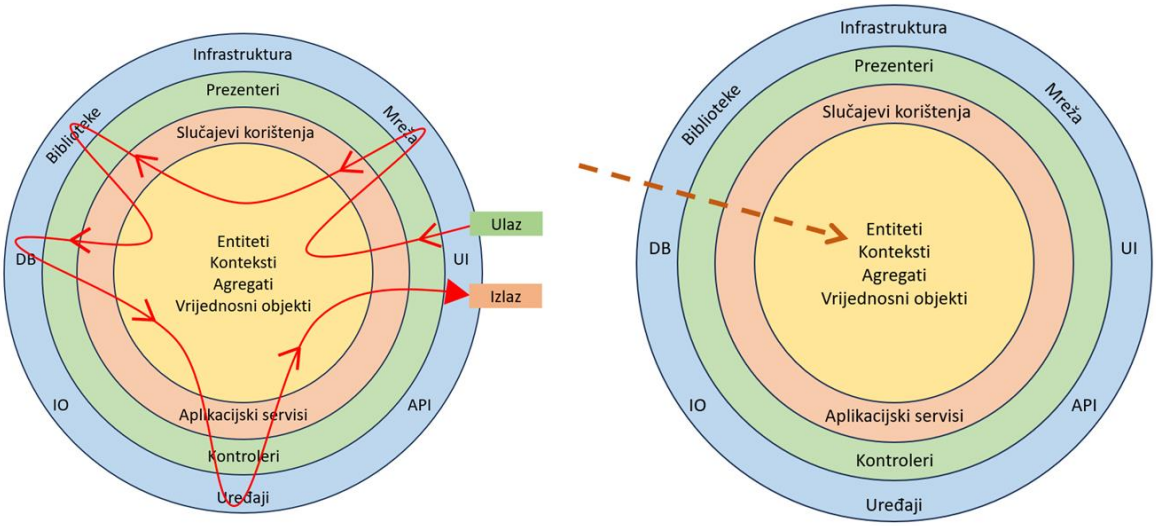
Kreiraj narudžbu	
Podaci	-Kupac_ID -Odredište_dostave -Informacije_o_plaćanju -Informacije_za_kontakt_kupca -Mehanizam_dostave
Osnovni tijek	1. Službenik za narudžbe izdaje naredbu za kreiranje narudžbe s gore navedenim podacima 2. Sustav validira sve podatke 3. Sustav kreira narudžbu i određuje ID narudžbe 4. Sustav dostavlja ID narudžbe službeniku za narudžbe
Tijek iznimke	Validacija pogrešaka: 1. Sustav dostavlja poruku pogreške službeniku za narudžbe

Slika 9 Primjer zahtjeva za jedan slučaj korištenja

Nakon aplikacijskoj sloja dolazi sloj adaptera koji za zadatak ima transformaciju podataka u prikladne strukture za unutarnji ili vanjski sloj. Ovaj sloj služi kao poveznica između vanjskoj sloja i aplikacijskog sloja. Podaci koji se prosljeđuju prema sustavu za prikaz će se ovdje transformirati na način da im sva polja budu u obliku tekstualnih tipova podataka. Podaci koji se razmjenjuju između ovog sloja i okolnih slojeva se u principu mogu smatrati modelima.(Clean Arhitecture : Part 2 - The Clean Architecture, 2017)

Zadnji sloj u ovoj arhitekturi je sloj infrastrukture te obuhvaća razvojne okvire i pokretačke programe. Ovaj sloj sadrži različite tehnologije koje potpomažu korištenju sustava, a neki od njih su baza podataka, korisničko sučelje, ulazno-izlazni uređaji biblioteke, itd. Tehnologije koje se ovdje dodaju se mogu jako brzo mijenjati te ga je teže testirati.

Zakrivljena linija sa Slika 10 predstavlja jedan od mogućih tokova izvršavanja naredbe. Moguće je da za jedan zahtjev treba više puta ulaziti prema jezgri arhitekture te određene rezultate obrađivati u vanjskim slojevima nakon čega ponovno podaci idu prema jezgri. Ovaj proces se može ponavljati koliko god je potrebno ovisno o naredbi prije nego što se odgovor vrati sloju koji je započeo ovaj proces. Desni graf sa Slika 10 prikazuje ovisnosti slojeva koje su izvana prema unutra. Na taj način promjene u vanjskim slojevima ne utječu na unutarnje slojeve, ali promjene u unutarnjim slojevima itekako imaju utjeca na vanjske slojeve. Promjenom tipa baze podataka u sloju infrastrukture ništa se ne mijenja u aplikacijskoj razini ili sloju domene, ali ako bi promijeniti nešto u sloju domene minimalna promjena bi bila u aplikacijskom sloju, a moguće i u svim drugim slojevima više razine.

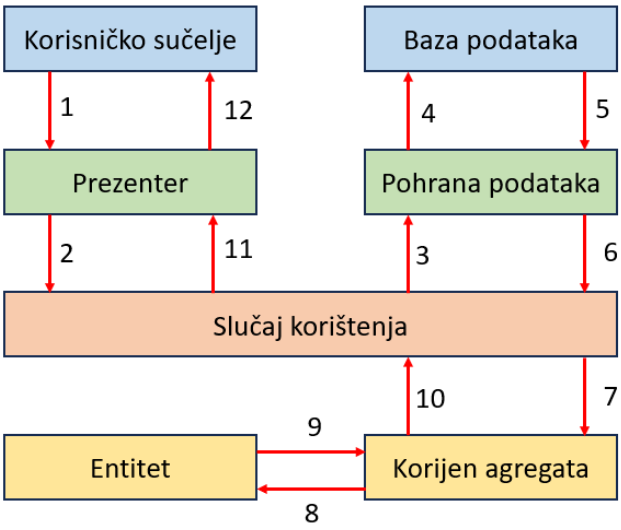


Slika 10 Primjer toka izvršavanja zahtjeva i ovisnost slojeva čiste arhitekture

Što se tiče toka izvršavanja prethodnu sliku je moguće prikazati i na drugačiji način kao što je vidljivo na Slika 11. Ovo je ujedno i najprisutniji tok izvršavanja naredbi, a koraci su sljedeći:

1. Neka akcija sa sučelja je aktivirala zahtjev prema unutarnjim slojevima
2. Događaj se prosljeđuje iz sloja pogleda u prezentacijski sloj. U ovom sloju se transformiraju podaci u prikladan oblik i prosljeđuju se do aplikacijskog sloja
3. Aplikacijski sloj kontrolira i orkestrira entitetima i drugim objektima sloja domene prema pravilima aplikacijske specifikacije, ali prije toga domenski objekti bi trebali biti kreirani na osnovu pohranjenih podataka u bazi podataka.
4. Nakon što su entiteti dohvaćeni u aplikacijski sloj, slučaj korištenja orkestrira entitetima i objektima domene prema pravilu aplikacijske specifikacije.

5. Zbog DDD-a koristi se korijen agregata kao ulazna točka u jezgru sloja domene.
6. Kada je zahtjev procesiran prema poslovnim pravilima, rezultat se vraća prema korisničkom sučelju. (Clean Arhitecture : Part 2 - The Clean Architecture, 2017)



Slika 11 Tok izvršavanja zahtjeva kroz slojeve čiste arhitekture

3.2. Entiteti

Entiteti su objekti koji imaju svoj identitet koji ostaje isti kroz sva stanja softvera. Nisu važni atributi koje sadrži već nit kontinuiteta i identiteta koji obuhvaćaju životni vijek sustava te se može proširiti izvan njega.

U kontekstu objektno-orijentiranih programskih jezika koristi se instanciranje objekata u memoriji na način da se svakom objektu dodjeljuje referenca odnosno memorijska adresa. Sama referenca je unikatna za svaki objekt u datom trenutku, ali nije uvijek ista jer se objekti konstantno kreću iz i u memoriju, šalju se, kreiraju se ili se uklanjaju stoga se referenca ne može uzeti kao identitet entiteta o kojem je ovdje riječ.

Postavlja se pitanje što je identitet i kako ga definirati? Ako bi pokušali kreirati model Osobe kroz programski jezik kreirali bi klasu Osoba koja sadrži određene atribute kao što su ime, prezime, datum rođenja, mjesto prebivališta itd. Iz tih atributa se teško može pronaći identitet. Iako je vjerojatnost mala, ali postoje osobe koje mogu imati sve te atribute iste. Objekt se mora razlikovati od ostalih objekata bez obzira što vrijednosti atributa mogu biti iste. Pogrešan identitet može dovesti do korupcije podataka. Kao dobar primjer identiteta može se uzeti osobni identifikacijski broj (skraćenica OIB) čija je

vrijednost jedinstvena te se ne može dogoditi da dvije osobe imaju isti OIB. Svaki entitet mora imati operativni način utvrđivanja svog identiteta s drugim objektom koji se može razlikovati u svojim atributima (Evans, 2003).

Stoga implementacija entiteta u softveru znači kreiranje identiteta. Identiet može biti atribut entiteta, kombinacija atributa ili atribut specijalno napravljen za čuvanje i prikaz identiteta. Mora se zadovoljiti pravilo da se dva entiteta sa različitim identitetom smatraju različitima u sustavu, a dva entiteta sa istim identitetom se smatraju istima u sustavu. Ako to pravilo nije zadovoljeno sustav može biti korumpiran. Postoje različiti načini za kreiranje unikatnog identiteta. Identifikator se može generirali automatski i koristiti internalno tako da ga korisnik ne vidi. Može se koristiti i primarni ključ baze podataka te se time osigurava unikatnost jer je taj zapis unikatan i u bazi podataka. (Avram & Marinescu, 2006)

Postoje određeni savjeti korištenja kada se objekt razlikuje po svom identitetu, a ne po atributima:

- Fokus na životni ciklus kontinuiteta i identiteta
- Definiranje načina razlikovanja objekata bez obzira na oblik ili povijest
- Oprezno raditi sa zahtjevima koji traže podudaranje objekata po atributima
- Model mora definirati što znači biti isti
- Sredstvo identificiranja može doći iz vanjskog svijeta ili biti kreiran od strane sustava

Entiteti su važni objekti modela domene i treba ih uzeti u obzir od početka procesa modeliranja. Također je važno odlučiti ako objekt treba biti entitet ili ne. (Avram & Marinescu, 2006)

3.3. Vrijednosni objekti

Entiteti mogu biti praćeni kroz sustav, ali praćenje i kreiranje dolazi uz cijenu. Kod entiteta svaka instanca ima svoj unikatni identitet, a njegovo praćenje nije jednostavno. Postoji i pitanje performansi kada bi sve objekte napravili entitetima. To bi značilo da je za svaki objekt potrebna jedna instanca, a sami rast tih instanci rezultira lošim performansama cjelokupnog sustava. U nekim slučajevima potrebno je sadržavati neke atribute elemenata

domene. U tom slučaju nije bitan objekt nego koje atribute posjeduje. Objekt koji se koristi za opisivanje određenih aspekata domene, a koji nema identitet, se naziva vrijednosnim objektom.

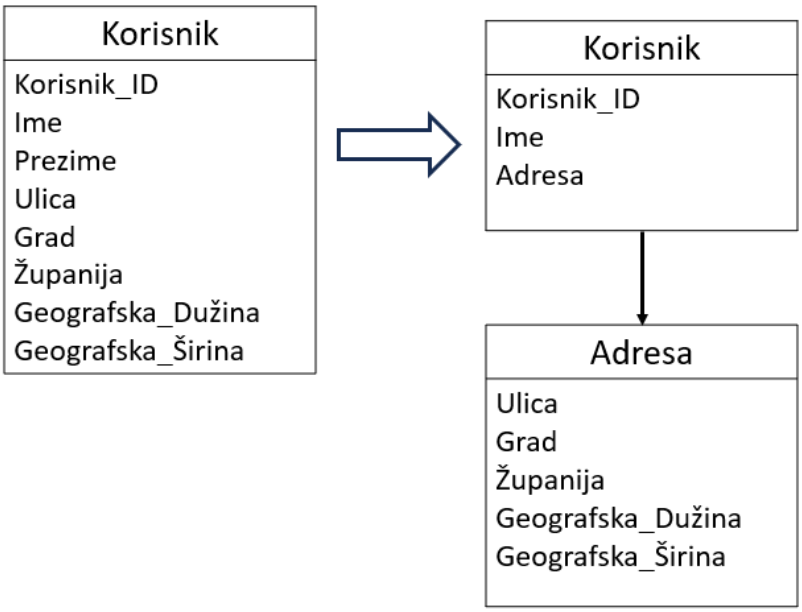
Nužno je razlikovati objekte entiteta od vrijednosnih objekata. Pogrešno je sve objekte napraviti entitetima radi uniformnosti. Ono što je preporučljivo je za objekte entitete odabrati samo one objekte koji su u skladu sa definicijom entiteta, a sve ostale objekte napraviti kao vrijednosnim objektima. Prateći ovo pravilo dobiva se pojednostavljeni dizajn koji samim time ima i druge pozitivne posljedice.

Pošto nemaju identiteta, vrijednosne objekte je lako kreirati i odbaciti. Nitko ne brine o stvaraju njegovog identiteta, a sakupljač smeća (engl. *Garbage Collector*) zbrinjava vrijednosni objekt kad ga više niti jedan drugi objekt ne poziva.(Evans, 2003)

Vrijednosni objekti bi trebali biti nepromjenjivi. Kreiraju se kroz konstruktor, a za vrijeme njihovog životnog vijeka se ne modificiraju. Ako je potrebna druga vrijednost za objekt jednostavno se kreira novi vrijednosni objekt. Ova značajka je jako bitna jer je vrijednosni objekt neovisan, a samim time se može dijeliti među različitim entitetima.

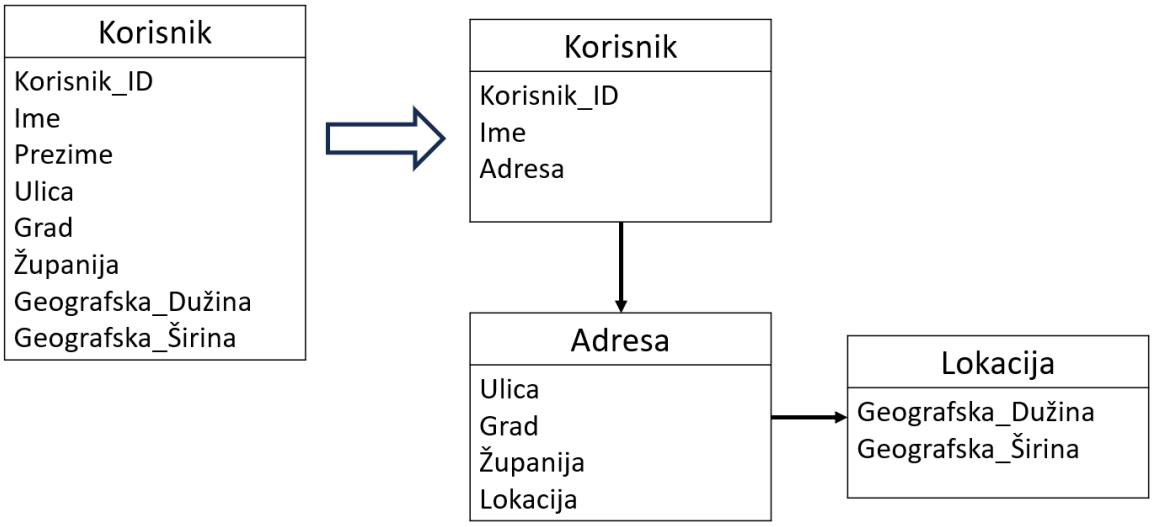
Zlatno pravilo vrijednosnih objekata je: vrijednosni objekti su djeljivi i trebaju biti nepromjenjivi. Vrijednosni objekti bi trebali biti jednostavni i mali. Kada je vrijednosni objekt potreban nekom, može se jednostavno proslijediti po vrijednosti ili se može stvoriti njegova kopija i dati kao takav. Stvaranje kopije vrijednosnog objekta je jednostavno i obično bez posljedica. Ako ne postoji identitet, može se napraviti koliko god kopija je potrebno i ukloniti ih kad nisu potrebni. (Evans, 2003)

U primjeru na Slika 12 predstavljen je entitet Korisnik koji ima svoj identitet, ime, ulicu, grad, županiju, geografsku dužinu i širinu. Ako pogledamo atribute ulica, grad, županija, geografska dužina i širina može se zaključiti da pokazuju na istu skupinu podataka odnosno podaci o mjestu stoga se tih pet atributa može odvojiti u vrijednosni objekt Adresa koji nema identitet.



Slika 12 Razdvajanje entiteta na entitet i vrijednosni objekt

Vrijednosni objekti također mogu sadržavati druge vrijednosne objekte, a u nekim slučajevima i referencu na entitete. Iako se vrijednosni objekti koriste za jednostavno sadržavanje atributa objekta domene to ne znači da bi trebao sadržavati dugačak popis atributa. Atributi se mogu grupirati u različite objekte. Atributi koji su odabrani da čine vrijednosni objekt trebaju tvoriti konceptualnu cjelinu.(Evans, 2003) Prema svemu navedenom prethodni primjer se može razdvojiti na dva vrijednosna objekta gdje će vrijednosni objekt Adresa imati referencu na vrijednosni objekt Lokacija prikazan na Slika 13.



Slika 13 Razdvajanje vrijednosnog objekta na dva vrijednosna objekta

3.4. Agregati

Objekti domene prolaze kroz skup stanja za vrijeme njihovog životnog vijeka. Kreiraju se, postavljaju se u memoriju, koriste se za proračune, nakon čega se mogu sačuvati na stalnim lokacijama kao što je baza podataka, a mogu se i u potpunosti izbrisati. Vođenje životnog ciklusa objekata domene predstavlja izazov, a ako nije vođeno kako treba može imati negativne utjecaje na model domene. Postoje tri načina koji pomažu u vođenju životnog ciklusa objekata domene, a to su agregati, tvornice (engl. *Factory*) i repozitoriji.

Agregat je domenski pristup koji služi za definiranje vlasništva i granica.(Evans, 2003). Model domene uvijek ima velik broj objekata te je neizbježno da su objekti povezani jedni s drugima te tako tvore kompleksnu mrežu relacija. Izazovi modela nisu u tome da budu dovoljno potpuni već da budu što jednostavniji i razumljiviji. Stvarne povezanosti između objekata domene prelaze u kôd, a nakon toga jako često i u bazu podataka. Postoji nekoliko tipova povezanost kao što su:

- Relacija jedan na jedan – veza između dva modela predstavlja i vezu između dvije tablice u bazi podataka, a predstavlja jedinstvenost u povezivanju dvaju modela. Jedan primjer prvog modela može pripadati najviše jednom primjeru drugog modela, a vrijedi i obrat.

- Relacija jedan na više – uključuje mnogo objekata jednog tipa koji su povezani sa najviše jednim objektom drugog tipa.
- Relacija više na više – dvosmjerna relacija. Povećava kompleksnost i čini vođenje životnog ciklusa takvih objekata mnogo težim. Broj povezivanja bi se trebao smanjiti što je više moguće. Prvi savjet je da se povezanosti koje nisu od važnosti uklone. Drugi savjet je da se višestrukost može smanjiti dodavanjem ograničenja, Treći savjet je da se dvosmjerna povezanost može transformirati u jednosmjernu primjenom poslovne logike.

Zbog količine modela i njihovih povezanosti trebaju se koristiti agregati. Agregat je skupina povezanih objekata koji se smatraju jednom jedinicom s obzirom na promjene podataka. Agregat je omeđen granicom koja dijeli objekte koji se nalazi unutar agregata od onih koji se nalaze van njega. Svaki agregat ima svoj korijen, a korijen je predstavljen entitetom te je to jedini objekt koji je dostupan vanjskom svijetu. Korijen može imati reference na bilo koji objekt koji se nalazi unutar agregata, a unutarnji objekti mogu imati reference međusobno. Vanjski objekt može sadržavati samo referencu prema korijenu. Dakle, u agregatima se nalaze vrijednosni objekti i drugi entiteti, a identitet tih entiteta je lokalan i ima smisla samo unutar agregata.

Agregat osigurava integritet podataka i provodi invarijante. Budući da vanjski objekti mogu imati samo reference na korijen to znači da ne mogu izravno mijenjati objekte koji se nalaze unutar agregata. Vanjski objekt može samo promijeniti korijen ili tražiti od njega da izvrši neke operacije. Korijen može mijenjati samo objekte koje ima ispod sebe unutar agregata. Ako se korijen izbriše to automatski znači da će se i svi objekti iz agregata izbrisati jer ne postoji niti jedan vanjski objekt koji ima referencu na njih.

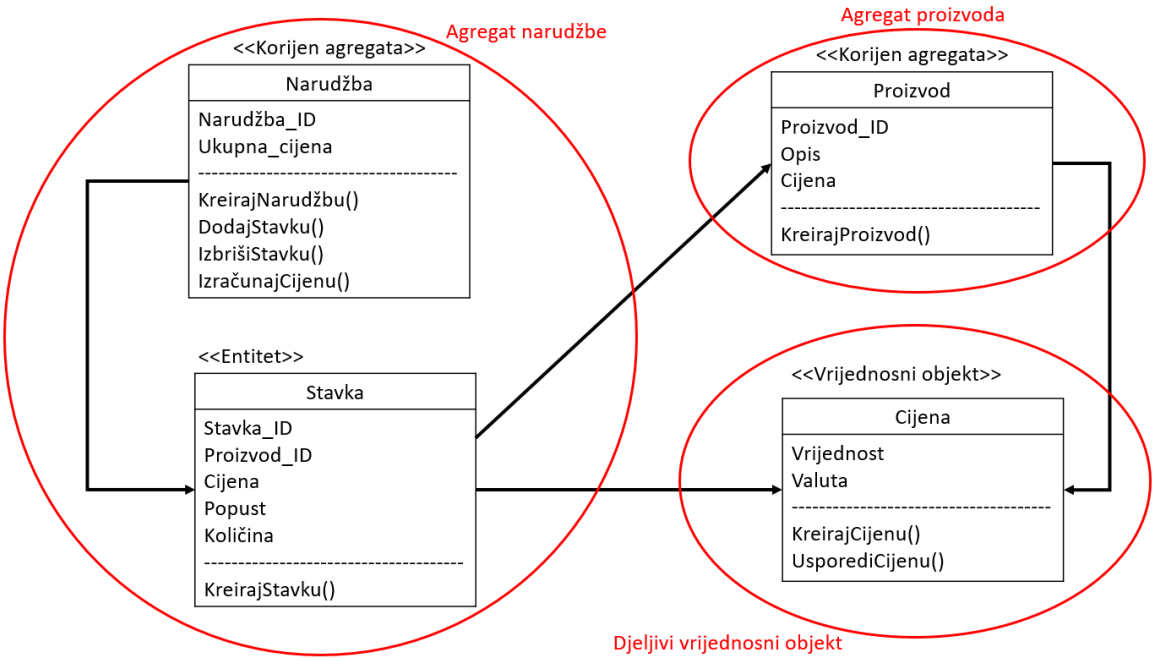
Korijen može proslijediti reference unutarnjih objekata vanjskim uz uvjet da vanjski objekti ne drže reference nakon završetka operacija. Jedan od načina za to je prosljeđivanje kopija vanjskim objektima. Nije važno što će se dogoditi s tim objektima jer to ni na koji način neće utjecati na cjelovitost agregata. (Evans, 2003)

Objekti unutar agregata mogu držati reference na korijene drugih agregata. Korijenski entiteti imaju globalni identitet i odgovoran je za održavanje invarijanti, a s druge strane interni entiteti imaju lokalni identitet koji se koristi samo unutar agregata.

Kod kreiranja agregata potrebno je grupirati entitete i vrijednosne objekte te definirati granice između svakog agregata. Potrebno je odabrati jedan entitet po agregatu koji će

služiti kao korijen agregata te će kontrolirati sav pristup objektima unutar agregata. Potrebno je dopustiti vanjskim objektima ili drugim korijenima da drže referencu samo na korijen. Ovakva organizacija pojednostavljuje nametanje invarijati za objekte u agregatu i za agregat kao cjelinu u bilo kojoj promjeni stanja.

Na Slika 14 je vidljiv primjer jednog jednostavnog agregata. Postoje dva agregata koji imaju svoje korijene, a u ovom slučaju jedan od njih ima entitet i vrijednosni objekt. Entitetom Stavka moguće je manipulirati samo kroz korijen odnosno entitet Narudžba te se na taj način primjenjuju poslovna pravila. Entitet Stavka ima referencu na korijen agregata Proizvod što je u skladu sa pravilima DDD-a i agregata. Također moguće je imat djeljive vrijednosne objekte koji se koriste u različitim entitetima i korijenima agregata.



Slika 14 Primjer jednostavnih agregata

3.5. Repozitoriji

Kako bi se nešto moglo raditi sa objektom potrebno je imati njegovu referencu. Postoje različiti načini dohvaćanja reference. Jedan način je kreiranje objekta jer će se samim time vratiti novi objekt sa svojom referencom. Drugi način je kroz pridruživanje. Od poznatog objekta se može tražiti pridruženi objekt.

Korištenje objekta znači da je objekt stvoren. Ako je objekt korijen agregata, onda je to entitet te će biti pohranjen u bazu podataka ili u neki drugi oblik pohrane. Što se tiče

vrijednosnog objekta, njega se može dobiti preko entiteta kojem je pridružen. Velik broj objekata se može dobiti iz baze podataka što rješava problem dobivanja reference objekata.

Baze podataka dio su infrastrukture. Loše je da klijent ili krajnji korisnik imaju ikakva saznanja o pristupu bazi podataka. U korištenju i kreiranju SQL upita može se vratiti skup podataka koji mogu otkriti više nego što bi trebali. U tom trenutku dolazi do ugrožavanja modela domene. Sav kôd koji bi se bavio upitima bi bio razbacan po projektima te bi se previše vremena izdvajalo na infrastrukturu umjesto na koncepte domene. Ovim načinom se stvara čvrst kôd koji nije podložan promjenama. Primjerice, ako bi se donijela odluka o promjeni vrste baze podataka sav kôd koji je koristio upite za staru vrstu baze podataka bi se trebao izmijeniti.

Softver treba omogućiti praktičan način dohvaćanja referenci na već postojeće objekte domene. Potencijalni problemi su da se logika domene prebacuje u upite, a entiteti i vrijednosni objekti postaju obični spremnici podataka, model postaje irelevantna te se gubi fokus domene i dizajn postaje ugrožen.

Kako se prethodni problemi ne bi dogodili koriste se repozitoriji, čija je svrha obuhvatiti svu logiku potrebnu za dobivanje referenci na objekte. Objekti domene neće se morati baviti infrastrukturom da bi dobili potrebne reference na druge objekte domene. Oni će ih samo preuzeti iz repozitorija i model ponovno ima svoju jasnoću i fokus.(Evans, 2003)

Korištenje i prednosti repozitorija su mnogobrojne:

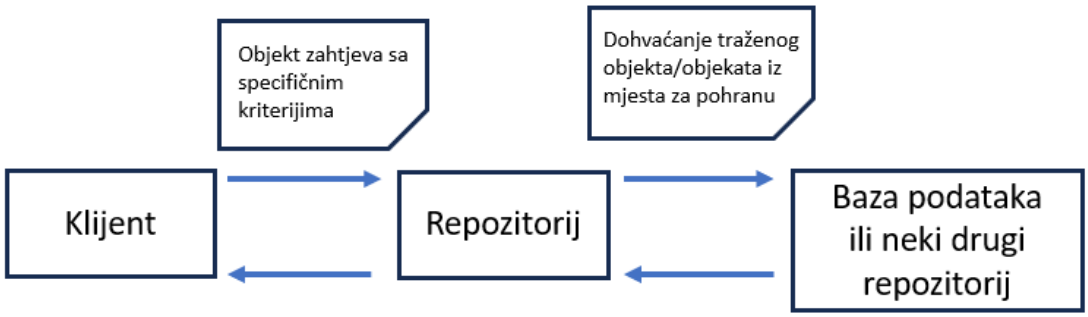
- Može pohraniti reference na neke od objekata
- Kreirani objekt može se spremiti u repozitorij i od tu ga dohvaćati za kasniju upotrebu
- Ako repozitorij ne sadrži traženi objekt dohvatit će ga iz baze podataka
- Predstavlja mjesto za pohranu globalno dostupnih objekata
- Može pristupiti jednoj ili drugoj trajnoj pohrani
- Različite lokacije za pohranu za različite vrste objekata

Savjeti u korištenju repozitorija su sljedeći:

- Za svaku vrstu objekta koji treba globalni pristup potrebno je stvoriti objekt koji pruža iluziju zbirke svih objekata te vrste u memoriji
- Postavljanje pristupa putem globalnog sučelja

- Omogućiti metode za dodavanje i uklanjanje objekata (stvarno umetanje i brisanje iz mjesta za pohranu npr. baze podataka)
- Omogućiti metode koje dohvaćaju objekte prema nekim kriterijima
- Omogućiti spremanje samo za korijene agregata koji trebaju izravan pristup
- Sučelje mora biti jednostavno

Korištenje repozitorija prema Slika 15 je vrlo jednostavno. Klijent poziva sučelje repozitorija za određenu vrstu objekta te može, a i ne mora koristiti parametre koji predstavljaju kriterij odabira objekata. Entitet se lako može dohvatiti prosljeđivanjem njegovog identiteta kao parametra, a ostali kriteriji mogu biti skup atributa tog objekta. Repozitorij uspoređuje sve objekte sa skupom kriterija i vraća one koji zadovoljavaju taj skup.



Slika 15 Pretraga objekta/objekata korištenjem repozitorija

Što se tiče definiranja sučelja repozitorija, ono bi trebalo biti dio modela domene, a sama implementacija bi trebala biti u infrastrukturi.

Kao što je objašnjeno, metode u sučelju repozitorija mogu primati parametre kao kriterije, ali na takav način moguće je da se u sučelju pojavi ogroman broj metoda zbog kompleksnosti domene. Kako bi se izbjegla takva složenost mogu se koristiti specifikacije. Opcija specifikacija omogućava kreiranje kompleksnijih kriterija te se više specifikacija može koristiti u jednoj metodi repozitorija. Taj način rezultira čistim kôdom i jednostavnim održavanjem.

3.6. Omeđeni kontekst

Kontekst je bitan kako bi se modeli mogli pravilno koristiti. Kad postoji samo jedan model onda je kontekst implicitan te ga nije potrebno definirati. Ali ako je u pitanju aplikacija koja treba komunicirati s drugom aplikacijom onda nova aplikacija ima svoj model i kontekst koji je različit od konteksta i modela aplikacije s kojom nova aplikacija treba komunicirati. Ti modeli se ne smiju kombinirati ili miješati stoga kada je u pitanju velika aplikacija potrebno je definirati kontekste za svaki model koji se stvara. Ako se modeli kombiniraju, softver postaje neispravan, nepouzdan i teško razumljiv. Sve to dovodi i to nejasne komunikacije među članovima tima jer nije jasno u kojem kontekstu se model primjenjuje.

Ne postoji formula da se jedan veliki model podijeli na manje. Potrebno je u model staviti one elemente koji su povezani i tvore prirodni pojam. Model treba biti dovoljno malen da se može dodijeliti jednom timu. Prateći ovakav način rada postiže se potpunija timska suradnja i komunikacija što je od velike pomoći za programere koji rade na istom modelu. Kontekst modela je skup uvjeta koje je potrebno primijeniti kako bi se osiguralo da pojmovi korišteni u modelu imaju određeno značenje. (Evans, 2003)

Neki od koraka u definiraju konteksta modela su:

- Definirati opseg modela (nacrtati granice njegovog konteksta)
- Pojednostavniti model
- Eksplicitno definirati kontekst unutar kojeg se model primjenjuje
- Eksplicitno postaviti granice u smislu organizacije tima, korištenja, shema baze podataka
- Održavati model strogo dosljednim unutar ovih granica

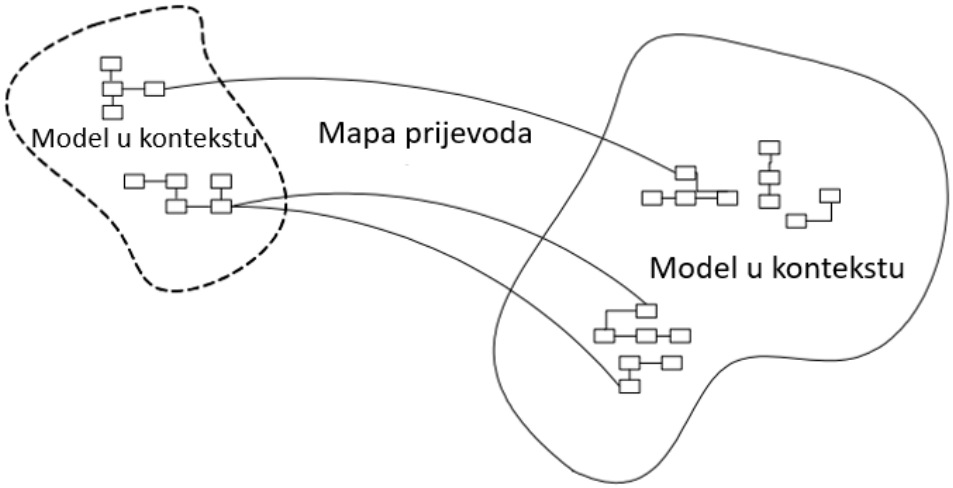
Omeđeni kontekst nije modul. Omeđeni kontekst pruža logički okvir unutar kojeg se model razvija. Moduli se koriste za organiziranje elemenata modela tako da omeđeni kontekst obuhvaća modul. (Evans, 2003)

U procesu izgradnje softvera sudjeluju različiti timovi te je potrebno da timovi ne nanose štetu drugim timovima kada rade promjene na modelu stoga je poželjno da konteksti modela budu dobro definirani te da svaki tim zna granice svog modela i da ostanu u tim granicama. Kako bi to bilo moguće model mora bit čist, jasan, dosljedan i jedinstven. U

svijetlu toga svaki model može lakše podržati refaktoriranje bez posljedica na druge modele.

Dakle, preporučeni pristup je stvaranje zasebnih modela za svaku domenu. Svi ti modeli se mogu razvijati bez brige o drugima modelima, čak svaki od modela može postat neovisna aplikacija. Važno je naglasiti da modeli mogu međusobno komunicirati, a kako bi to bilo moguće potreban je sustav razmjene poruka.

Mapa konteksta je dokument koji ocrtava različite ograničene kontekste i odnose između njih. (Evans, 2003) Mapa konteksta može biti dijagram poput ovog prikazanog na Slika 16 ili može biti bilo koji pisani dokument.



Slika 16 Mapa prevođenja modela iz jednog konteksta u drugi

Nije dovoljno imati zasebne unificirane modele već je potrebno da su ti modeli integrirani jer je funkcionalnost svakog modela dio cijelokupnog sustava. Svi ti dijelovi se na kraju moraju sastaviti, a sustav mora ispravno raditi. Ako konteksti nisu jasno definirani moguće je da će se preklapati, a ako odnosi između konteksta nije naveden onda je moguće da sustav neće funkcionirati nakon integracije. Potrebno je da svaki omeđeni kontekst ima svoj naziv te da je taj naziv predstavljen i u sveprisutnom jeziku. Na taj način se olakšava komunikacija tima. Postoji pravilo pravilo koje kaže da je prvo potrebno definirati kontekste, a zatim stvoriti module za svaki kontekst.(Evans, 2003)

4. Primjer projekta

Kao primjer projekta odabran je softver za iznajmljivanje vozila. Ova izrada je temeljena na mom poznavanju poslovanja jedne ovakve kompanije stoga u izradu ovog softvera nisu bili uključeni domenski eksperti. Projekt je subjektivan jer moje znanje predstavlja i znanje domenskog eksperta i znanje eksperta programske podrške. Bez obzira na to, bit će prikazan i proces kreiranja sveprisutnog jezika uz sve druge komponente i prakse DDD-a. Cijeli projekt je izrađen u programskom jeziku C# te je korištena čista arhitektura kroz čije slojeve će biti prikazane funkcionalnosti softvera

4.1. Zahtjevi softvera

Kao i u svakom poslovanju potrebno je znati od čega poslovanje stvara dobit bez obzira radili se o materijalnoj ili financijskoj dobiti. Softver za iznajmljivanje vozila treba sadržavati vozila jednog ili različitih tipova i kategorija. Vozila stvaraju ono što kompanija pruža kao ponudu te zbog njih ostvaruje prihode. Uz vozila postoje i različiti drugi elementi kojima kompanija ostvaruje dobit, a to su osiguranja i dodatni predmeti.

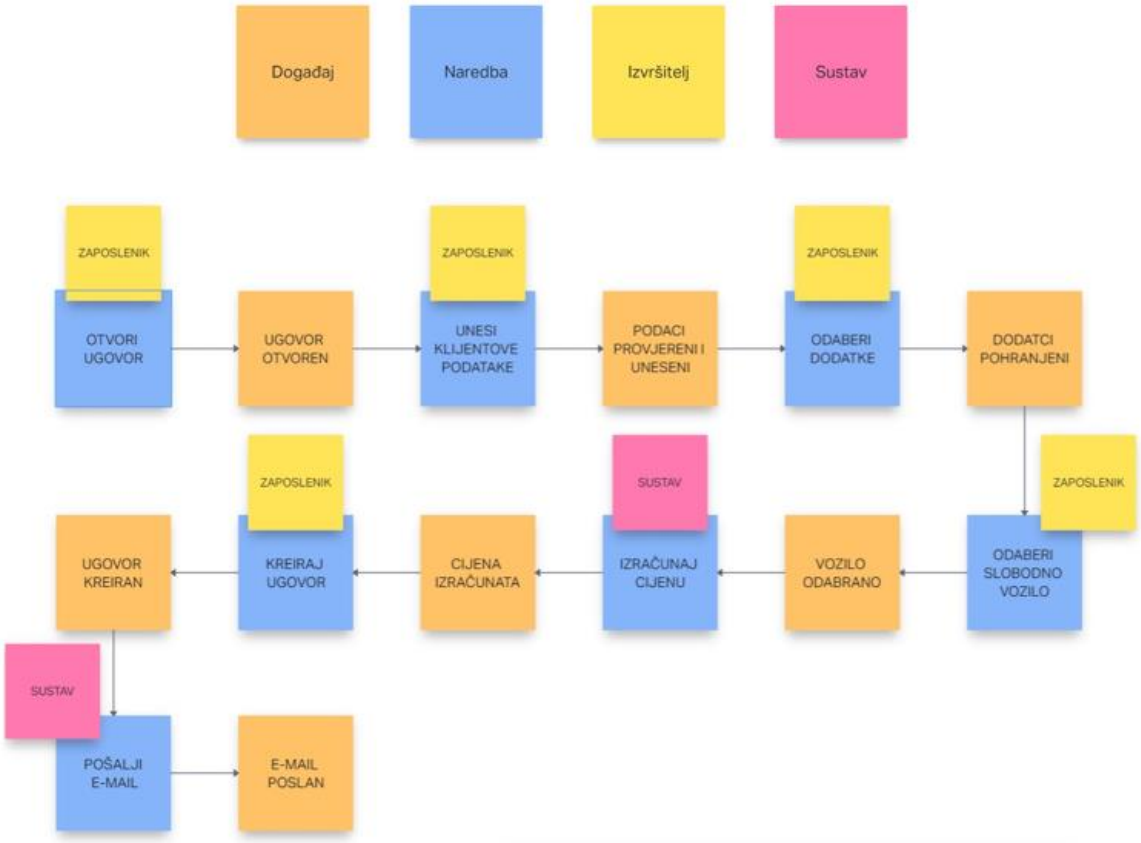
Kako bi kompanija počela ostvarivati dobit potrebno je sklapati ugovore sa klijentima koji će iznajmiti vozilo te možda uzeti neku od dodatnih mogućnosti i naravno dodatno platiti. Kako bi to sve bilo moguće potreban je softver koji će automatizirati određene procese te stvoriti jednostavan i pregledan način za manipuliranje podacima.

Centralni entiteti za problematiku ovog poslovanja su vozilo, ugovor, dodaci, podružnica. Što se tiče vozila ono mora imati marku (npr. Audi), model (npr. A6), a samo vozilo je jedinstveno svojom registracijskom oznakom. Ugovor je model koji sadrži podatke o klijentu, vremenu preuzimanja i vraćanja te o mjestu preuzimanja i vraćanja. A naravno najbitniji element je cijena. Ugovor može nastati na dva načina. Kroz rezervaciju ili bez rezervacije. Kako bi se ugovor kreirao potrebni su zaposlenici koji rade za određenu podružnicu kompanije. Prilikom kreiranja ugovora mogu se dodati i dodaci te sukladno tome cijena raste.

Ovo su u suštini glavni elementi softvera za iznajmljivanje vozila. Naravno ovaj softver je minijatura prave poslovne problematike te prikazuje osnovni rad ovakvog tipa poslovanja.

4.2. Razmjena događaja

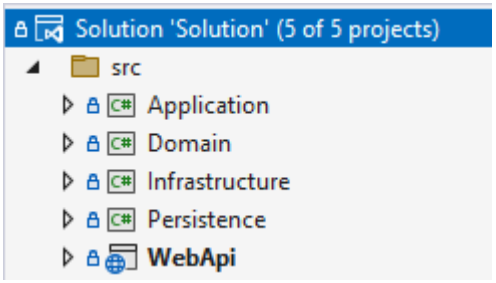
U procesu razmjene događaja stvara se razumijevanje poslovnog problema te se počinje oblikovati sveprisutni jezik. U procesu razmjene događaja sudjeluju domenski stručnjaci i razvojni programeri. Na slici (Slika 17) je prikazan proces kreiranja ugovora. Također je vidljiva i legenda što koja boja naljepnica predstavlja. Zaposlenik otvara formu za ugovor, unosi klijentove podatke koji se provjeravaju prema pravilima sustava. Zaposlenik dodaje dodatke na ugovor te odabire vozilo. Sustav računa ukupnu cijenu i cijenu najma. Ako je sve uspješno zaposlenik kreira ugovor, a sustav šalje email klijentu sa podacima na ugovoru. Koristeći razmjenu podataka vrlo jednostavno se mogu objasniti koraci u izvršavanju kompleksnih procesa nekog poslovanja. Na ovaj način stručnjaci za domenu kroz priču opisuju proces, a zajedno sa stručnjacima programske podrške kreiraju prethodno objašnjenu sliku koja se može još više detaljizirati. Ako bi se svaki proces i pojam objasnili putem razmjene događaja stvorila bi se jasna slika sustava te bi se time kreirao i sveprisutni jezik.



Slika 17 Razmjena događaja za proces kreiranja ugovora

4.3. Stuktura projekta

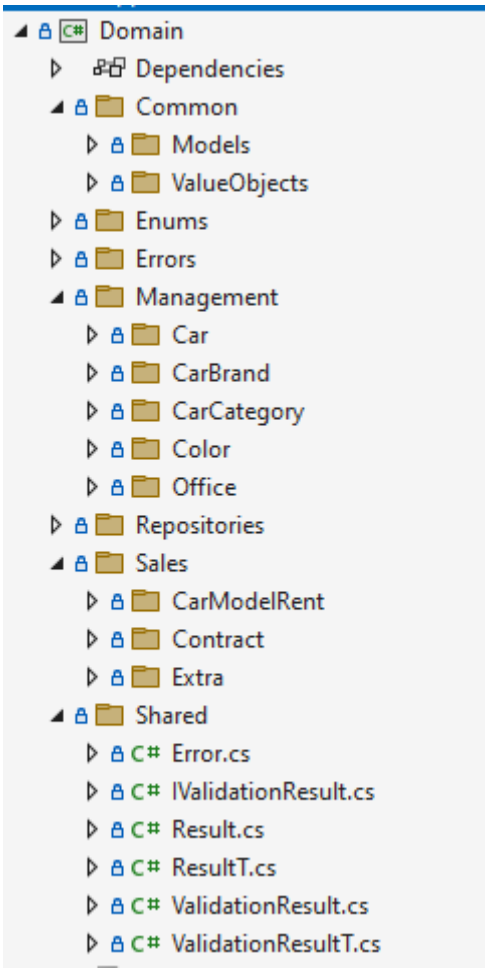
Struktura projekta je građena prema pravilima čiste arhitekture što znači da je projekt podijeljen na slojeve: sloj domene, aplikacijski sloj, sloj infrastrukture i prezentacijski sloj. Sloj infrastrukture je podijeljen na dva sloja: infrastruktura i *persistence*, a prezentacijski sloj je u obliku API-a te se slojevi mogu vidjeti na slici Slika 18. Za svaki od slojeva kreiran je jedan projekt tipa *Class Library*, a za prezentacijski sloj je korišten projekt tipa ASP.NET Core Web API. Svi projekti su verzija .NET6.0. Ovisnosti među projektima će biti objašnjene u nastavku kroz svaki sloj.



Slika 18 Slojevi u primjeru projekta

4.3.1. Sloj domene

Kao što je već spomenuto, sloj domene je centar oko kojeg se gradi čitavi softver. Nakon što su napravljene prve verzije razmjene događaja i sveprisutnog jezika kreće se na kreiranje domene sa svim potrebnim modelima, metodama i poslovnim pravilima. Ovaj sloj nije ovisan niti o jednom sloju te nema instaliranih paketa. Izgled domenskog sloja je vidljiv na slici Slika 19 te je podijeljen po mapama.



Slika 19 Sloj domene u primjeru projekta

U ovom sloju se definiraju entiteti, agregati i vrijednosni objekti. Kako bi definirali te osnovne komponente u kôdu napravio sam apstraktne klase koje će nasljeđivati modeli prema njihovoj ulozi. Za koncept entiteta je kreirana apstraktna klasa *Entity* (Slika 20) za koju se definira identitet koji je tipa *Guid* (engl. *Globally Unique Identifier*) te će ovaj tip biti korišten i kao primarni ključ u bazi podataka. Za entitet su kreirane osnovne operacije kao što su provjera jednakosti entiteta i dohvaćanje *Hash* koda. Provjera jednakosti također mora zadovoljiti pravilo entiteta koje kaže da su dva entiteta jednaka samo i samo ako su njihovi identifikatori jednaki.

```
namespace Domain.Common.Models;
22 references
public abstract class Entity : IEquatable<Entity>
{
    6 references
    protected Entity(Guid id) => Id = id;

    0 references
    protected Entity()
    {
    }

    63 references
    public Guid Id { get; private init; }

    28 references
    public static bool operator ==(Entity? first, Entity? second) =>
        first is not null && second is not null && first.Equals(second);

    2 references
    public static bool operator !=(Entity? first, Entity? second) =>
        !(first == second);

    1 reference
    public bool Equals(Entity? other)...

    0 references
    public override bool Equals(object? obj)...

    0 references
    public override int GetHashCode() => Id.GetHashCode() * 41;
}
```

Slika 20 Implementacija koncepta entiteta

Idući koncept koji je trebalo implementirati je koncept agregata. Agregat nije ništa drugo nego entitet koji drži na okupu nijedan ili više entiteta te s njima može manipulirati. Implementacija ovog koncepta je poprilično jednostavna, a napravljena je na način da je kreirana apstraktna klasa *AggregateRoot* (Slika 21) koja nasljeđuje prethodno napravljenu apstraktnu klasu *Entity* te ima svoj identifikator tipa Guid koji se prosljeđuje osnovnom konstruktoru iz klase *Entity*.

```
namespace Domain.Common.Models;
16 references
public abstract class AggregateRoot : Entity
{
    7 references
    protected AggregateRoot(Guid id)
        : base(id)
    {
    }

    0 references
    protected AggregateRoot()
    {
    }
}
```

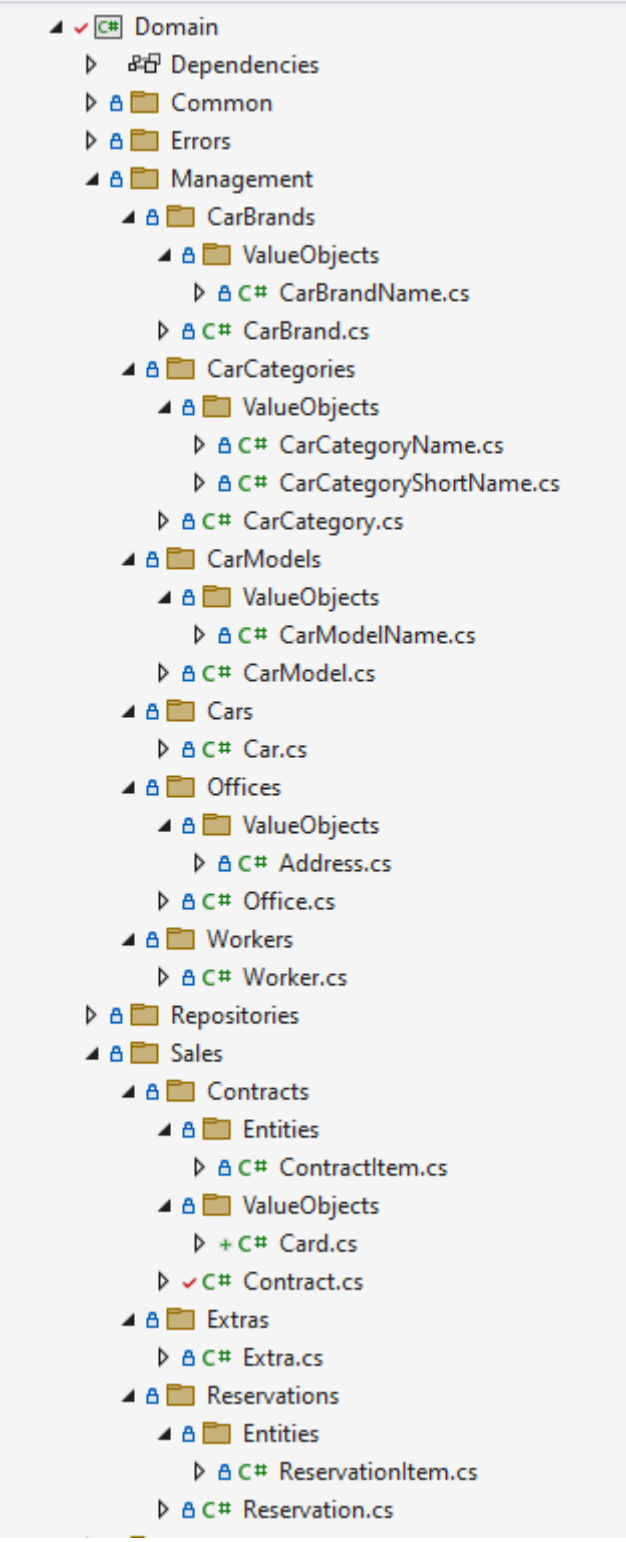
Slika 21 Implementacija koncepta agregata

Zadnji koncept koji je bitan prije nego što se krene u izradu svih modela poslovanja je vrijednosni objekt. Vrijednosni objekt nema svoj identitet nego samo vrijednost te služi za enkapsulaciju. Implementacija je slična implementaciji entiteta, ali umjesto identifikatora postoji samo vrijednost (Slika 22). Dakle, kreira se apstraktna klasa *ValueObject* koja sadrži metode za uspoređivanje vrijednosnih objekata. Također, ova implementacija zadovoljava pravilo da su dva vrijednosna objekta jednaka ako su njihove vrijednosti jednake.

```
namespace Domain.Common.Models;
14 references
public abstract class ValueObject : IEquatable<ValueObject>
{
    13 references
    public abstract IEnumerable<object> GetAtomicValues();

    2 references
    public bool Equals(ValueObject? other)
    {
        return other is not null && ValuesAreEqual(other);
    }

    0 references
    public override bool Equals(object? obj)...

    0 references
    public override int GetHashCode()...

    2 references
    private bool ValuesAreEqual(ValueObject other)...
}
```

Slika 22 Implementacija koncepta vrijednosnog objekta

Nakon što su kreirani osnovni koncepti DDD-a može se početi sa kreiranjem modela i poslovnih pravila za softver iznajmljivanja vozila. Slika (Slika 23) prikazuje sve modele korištene u rješavanju poslovnog problema. Modeli su podijeljeni na dva konteksta: *Management* i *Sales*. Kontekst *Management* se sastoji od korijenskih agregata *CarBrand*, *CarCategory*, *CarModel*, *Car*, *Office*, *Worker*. Za te agregate postoje i vrijednosni objekti kao što su *CarBrandName*, *CarCategoryName*, *CarCategoryShortName*, *CarModelName* i *Address*. S druge strane, kontekst *Sales* se sastoji od korijenskog agregata *Contract* koji koristi vrijednosni objekt *Card* te pod sobom ima entitet *ContractItem*. U ovom kontekstu također postoji i korijenski agregat *Reservation* koji pod sobom ima entitet *ReservationItem*. Uz ta dva agregata još je potreban korijenski agregat *Extra* koji služi za dodatne opcije prilikom rezerviranja ili kreiranja ugovora.



Slika 23 Modeli domene korišteni u primjeru projekta

U ovom sloju živi poslovna logika i pravila kojih se potrebno pridržavati kako bi se ispunili zahtjevi poslovanja. Za primjer poslovnih pravila prikazan je agregat *Contract* koji sadrži poprilično poslovnu logiku te se provodit na sljedeći način:

1. Unijeti podatke o klijentu

2. Odabrati način plaćanja

   2.1. Ako je odabran kartični način plaćanja potrebno je unijeti i podatke o kartici.

3. Odabrati lokaciju preuzimanja i lokaciju vraćanja vozila
4. Odabrati dostupno vozilo
5. Izračunati cijenu najma na osnovu broja dana najma i cijene modela vozila uz moguću primjenu popusta
6. Ako postoje odabrani dodaci kreirati entitete *ContractItem*
7. Izračunati ukupnu cijenu na osnovu cijene najma i cijena dodataka
8. Kreirati ugovor
9. Sustav automatizmom šalje email klijentu

Ilustracijama kôda (Slika 24, Slika 25) prikazat će se za primjer šesti i sedmi korak navedenog redoslijeda izvršavanja kreiranja ugovora. Na slici (Slika 24) su vidljivi i obavezni te opcionalni podaci koje je potrebno ispuniti kako bi se kreirao ugovor. Metoda za kreiranje radi izračune broja dana najma te ih množi sa osnovnom cijelom modela vozila. Time se trenutno dobije i ukupna cijena i cijena najma. Za vozilo je potrebno da ne bude u najmu ili na popravku odnosno da bude dostupno i slobodno za najam. Ako nije, potrebno je vratiti poruku greške. U slučaju da postoje dodaci kao što su primjerice dječja sjedalica, bežični internet itd. potrebno je kreirati stavke ugovora sa količinom dodatka. Samim time potrebno je ispraviti ukupnu cijenu na ugovoru, ali se cijena najma ne mijenja jer se ona odnosi samo na najam vozila.

```
1 reference
public static Result<Contract> Create(
    Guid id,
    FirstName driverFirstName,
    LastName driverLastName,
    Email email,
    DateTime pickUpDate,
    DateTime dropDownDate,
    Office pickUpOffice,
    Office dropDownOffice,
    Car car,
    string driverLicenceNumber,
    string driverIdentificationNumber,
    CardType? cardType,
    PaymentMethod paymentMethod,
    Card? card,
    Reservation? reservation,
    CarModel carModel,
    Worker worker)
{
    var duration = (decimal)dropDownDate.Subtract(pickUpDate).TotalDays;

    var rentalPrice = duration * carModel.PricePerDay - ((duration * carModel.PricePerDay) * carModel.Discount);

    var totalPrice = rentalPrice;

    if(car.Status == CarStatus.Repairing || car.Status == CarStatus.Rented)
    {
        return Result.Failure<Contract>(new Error(
                "CarCannotBeRented",
                $"The Car cannot be rented beacuse it is in status {car.Status}"));
    }

    return new Contract(id, driverFirstName, driverLastName, email, pickUpDate, dropDownDate,
        totalPrice, pickUpOffice.Id, dropDownOffice.Id, car.Id, driverLicenceNumber, driverIdentificationNumber,
        cardType, paymentMethod, card, reservation is null ? null : reservation.Id, rentalPrice, worker.Id);
}
```

Slika 24 Metoda za kreiranje ugovora

```
2 references
public ContractItem AddContractItem(decimal quantity, Extra extra)
{
    var detailElement = _contractItems.Where(x => x.ExtraId == extra.Id).SingleOrDefault();
    if (detailElement is not null)
    {
        _contractItems.RemoveAll(x => x.ExtraId == extra.Id);
        TotalPrice -= detailElement.Price;
    }
    var contracDetail = ContractItem.Create(Guid.NewGuid(), quantity, extra, this);
    TotalPrice += contracDetail.Price;
    _contractItems.Add(contracDetail);
    return contracDetail;
}
```

Slika 25 Metoda za dodavanje dodataka na ugovor

U prethodno navedenom primjeru vidljivo je da su neki podaci vrijednosni objekti kao što je *Card*, njegova implementacija je vidljiva na slici (Slika 26). Ovaj vrijednosni objekt također primjenjuje određena pravila poslovanja. Ako je za ugovor odabran kartični način plaćanja potrebno je ispuniti podatke o kartici. Naravno i ti podaci imaju neka ograničenja i pravila koja moraju zadovoljiti. Na primjer sigurnosni kartični kôd mora sadržavati točno tri broja. Ako su svi uvjeti ispunjeni kreira se vrijednosni objekt kartice, a inače se vraća greška sa opisom o točnom razlogu nemogućnosti kreiranja kartice.

```
public sealed class Card : ValueObject
{
    public const int CvvLength = 3;

    public const int CardYearExpirationLength = 2;

    3 references
    public string? CardName { get; private set; }
    3 references
    public string? CardNumber { get; private set; }
    3 references
    public string? CVV { get; private set; }
    3 references
    public string? CardDateExpiration { get; private set; }
    3 references
    public string? CardYearExpiration { get; private set; }

    1 reference
    private Card(string? cardName, string? cardNumber, string? cVV, string? cardDateExpiration, string? cardYearExpiration)...
    0 references
    private Card()...
    1 reference
    public static Result<Card> Create(string? cardName, string? cardNumber,
        string? CVV, string? cardDateExpiration, string? cardYearExpiration, PaymentMethod paymentMethod)
    {
        if (paymentMethod == PaymentMethod.Card)
        {
            if (string.IsNullOrEmpty(cardName))...
            else if (string.IsNullOrEmpty(cardNumber))...
            else if (string.IsNullOrEmpty(CVV) || CVV.Length != CvvLength)...
            else if (string.IsNullOrEmpty(cardName))...
            else if (string.IsNullOrEmpty(cardDateExpiration))...
            else if (string.IsNullOrEmpty(cardYearExpiration) || cardYearExpiration.Length != CardYearExpirationLength)
            {
                return Result.Failure<Card>(new Error(
                    "CardYearExpirationFormat",
                    $"The CardYearExpiration field cannot be null or longer/shorter then {CardYearExpirationLength}"));
            }
        }

        return new Card(cardName, cardNumber, CVV, cardDateExpiration, cardYearExpiration);
    }
```

Slika 26 Implementacija vrijednosnog objekta *Card*

Na sličan način je implementirano i kreiranje rezervacije uz nešto manji broj podatka koje je potrebno ispuniti. Svi korijenski agregati i entiteti u domeni imaju metode za kreiranje i ažuriranje uz to što za neke nema posebne poslovne logike. U slučaju jednostavnijih agregata kao što je *CarBrand* koji sadrži samo vrijednosni objekt *CarBrandName*, a koji ima definiranu minimalnu i maksimalnu duljinu potrebno je samo ispuniti njegov zahtjev i marka vozila se može kreirati.

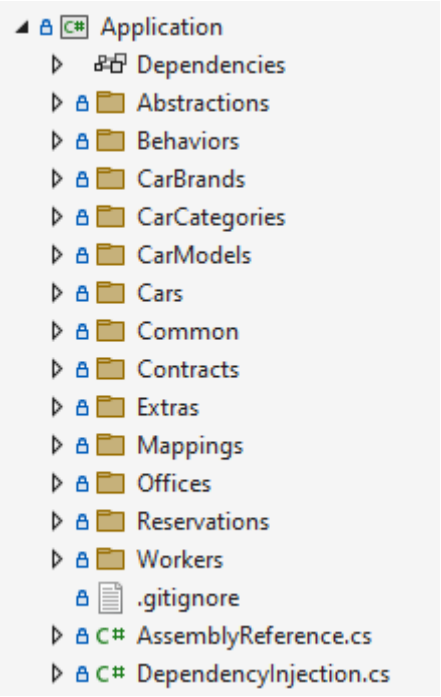
U sloju domene se također nalaze i sučelja repozitorija za svaki od prethodno navedenih agregata i entiteta, a njihova implementacija će biti objašnjena i prikazana u *Persistence* sloju. Svaki od agregata ima četiri osnovne metode u svojim sučeljima repozitorija (Slika 27), a to su dodavanje (metoda prima model), dohvaćanje svih, dohvaćanje po identifikatoru (metoda prima identifikator) i provjeravanje postojanja agregata (metoda prima unikatno polje modela).

```
13 references
public interface ICarBrandRepository
{
    3 references
    Task AddAsync(CarBrand carBrand, CancellationToken cancellationToken = default);
    2 references
    Task<List<CarBrand>> GetAllAsync(CancellationToken cancellationToken = default);
    4 references
    Task<CarBrand?> GetByIdAsync(Guid id, CancellationToken cancellationToken = default);
    2 references
    Task<bool> AlreadyExists(CarBrandName carBrandName, CancellationToken cancellationToken = default);
}
```

Slika 27 Primjer sučelja repozitorija

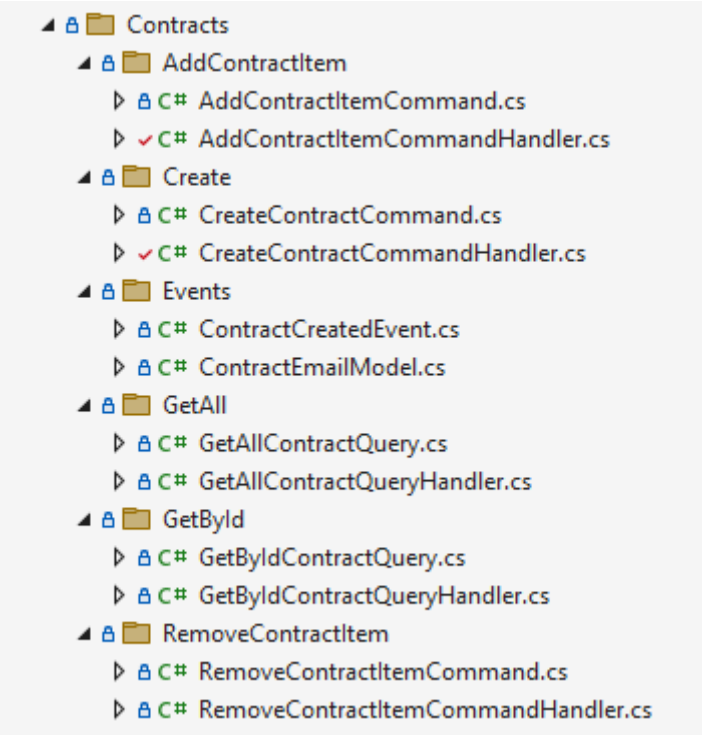
4.3.2. Aplikacijski sloj

Aplikacijski sloj (Slika 28) zajedno sa slojem domene čini jezgru čiste arhitekture. Ovaj sloj ima ovisnost samo na sloj domene. Kako je već spomenuto, ovaj sloj zaprima zahtjeve od viših slojeva te za svaki zahtjev orkestrira modelima iz sloja domene. Zahtjevi mogu zahtjevi kreiranje modela, dohvaćanje modela, ažuriranje modela ili brisanje modela. Kako bi aplikacijski sloj prepoznao svaki zahtjev korištena je biblioteka MediatR koja preimenuje pravilo razdvajanja odgovornosti kroz naredbe i upite (engl. *Command and Query Resposibility Segregation*, skraćenica CQRS). Korištenjem biblioteke MediatR smanjuju se ovisnosti među objektima. Također omogućuje slanje poruku u procesu te ne dozvoljava direktnu komunikaciju među objektima. S druge strane CQRS razdvaja rad na modele koji nešto pišu u mjesto za pohranu i na modele koji čitaju iz mjesta za pohranu. Na taj se način stvaraju jasne granice među ponašanjima sustava te se omogućuje labava povezanost kôda.



Slika 28 Aplikacijski sloj u primjeru projekta

Kako bi se sve navedeno implementiralo aplikacijski sloj sam podijelio na mape koje predstavljaju agregate iz sloja domene, a unutar svake mape se nalaze mape za kreiranje modela, dohvaćanje modela, dohvaćanje svih modela te ažuriranje modela. U primjeru na slici (Slika 29), osim navedene podjele po mapa je vidljivo i pravilo pisanja klasa koje slijedi CQRS. Ako se radi o akcijama koje nešto upisuju ili mijenjaju u mjestu za pohranu onda se prvo kreira klasa koja na kraju ima ključnu riječ *Command* te se za nju navodi i druga klasa koja će odraditi zahtjeve iz te naredbe i u svom nazivu ima ključnu riječ *CommandHandler*. Slično je za akcije koje čitaju iz mjesta za pohranu. Prvo se kreira klasa koja na kraju ima ključnu riječ *Query*, a njenu implementaciju će odrađivati klasa koja na kraju svog naziva ima ključnu riječ *QueryHandler*. Naravno ovakav način pisanja nije nužan te ću pokazati što svaka od klasa sadrži, ali na ovaj način se vizualno može zaključiti što će koja klasa raditi.



Slika 29 Primjer strukture agregata *Contract* u aplikacijskom sloju

Zbog mnoštva agregata za primjer ću pokazati kreiranje ugovora (*Contract*) kao agregata sa najviše poslovnih pravila.

Slučaj korištenja za kreiranje ugovora u aplikacijom sloju se događa u dvije klase: *CreateContractCommand* i *CreateContractCommandHandler*. Prvo se kreira naredba (Slika 30) u kojoj se definiranu podaci zahtjeva te se definira i što ovaj zahtjev traži kao povratnu vrijednost, a to je u ovom slučaju tip Guid odnosno identifikator. Nakon što se kreira zahtjev potrebno je kreirati klasu koja će rukovoditi tim zahtjevom (Slika 31). U toj klasi potrebno je navesti koji zahtjev odrađuje te što vraća kao rezultat izvršavanja. Bitno je da povratni tip bude isti kao i u zahtjevu kojeg obrađuje. U primjeru sa slike (Slika 31) vidljivi su koraci u obradi zahtjeva za kreiranje ugovora. Prvo je potrebno dohvatiti i provjeriti agregate koji su potrebni za kreiranje ugovora, a to su poslovnica preuzimanja i vraćanja, vozilo i radnik. Ovi agregati se dohvaćaju koristeći njihova sučelja repozitorija te ovaj sloj ne zna je li radi sa bazom podataka ili nekim drugim oblikom pohrane podataka. Ako nije bilo moguće dohvatiti neki od navedenih agregata potrebno je vratiti grešku. Ako je sve u redu nastavlja se sa kreiranjem vrijednosnih objekata koji su potrebni za kreiranje ugovora. I u ovom dijelu se provjerava jesu li vrijednosni objekti uspješno kreirani te ako nisu potrebno je vratiti grešku. Ako nema grešaka nastavlja se sa kreiranje instance ugovora kojoj se prosljeđuju svi prethodno dohvaćeni agregati i napravljeni vrijednosni

objekti. Svi ti podaci se šalju u agregat ugovora koji se nalazi u sloju domene te se koristi njegova statička metoda za kreiranje koja je prethodno bila pokazana, a koja će izračunati trenutnu cijenu najma i ukupnu cijenu. Važno je napomenuti kako je ugovor u ovom trenutku samo instanca koja još nije pohranjena u mjesto pohrane podataka. Nakon ovog koraka provjerava se postoje li dodaci za ugovor. Ako postoje potrebno ih je dohvatiti iz mjesta za pohranu te kreirati stavke ugovora i ispraviti ukupnu cijenu ugovora. Nakon što su svi koraci odrađeni instanca ugovora i stavke ugovora se spremaju, a klijentu se šalje email sa potvrdom i informacijama sa ugovora. Nakon što je email poslan rukovoditelj vraća identifikator ugovora.

```
public sealed record CreateContractCommand(
    string DriverFirstName,
    string DriverLastName,
    string Email,
    DateTime PickUpDate,
    DateTime DropDownDate,
    Guid PickUpOfficeId,
    Guid DropDownOfficeId,
    Guid CarId,
    string DriverLicenceNumber,
    string DriverIdentificationNumber,
    CardType? CardType,
    PaymentMethod PaymentMethod,
    Guid? ReservationId,
    Guid WorkerId,
    string? CardName,
    string? CardNumber,
    string? CVV,
    string? CardDateExpiration,
    string? CardYearExpiration,
    List<ExtrasModel>? Extras) : ICommand<Guid>;
```

Slika 30 Zahtjev za kreiranje *Contract* modela

```
internal class CreateContractCommandHandler : ICommandHandler<CreateContractCommand, Guid>
{
    variables

    0 references
    public CreateContractCommandHandler(
        ILogger<CreateContractCommandHandler> logger,ICarRepository carRepository,
        IContractRepository contractRepository,IContractItemRepository contractItemRepository,
        IExtrasRepository extrasRepository,IOfficeRepository officeRepository,
        IUnitOfWork unitOfWork,IReservationRepository reservationRepository,
        IWorkerRepository workerRepository,IPublisher publisher)...

    0 references
    public async Task<Result<Guid>> Handle(CreateContractCommand request, CancellationToken cancellationToken)
    {
        _logger.LogInformation("Started CreateContractCommandHandler");
        try
        {
            Check DropDownLocation, PickUpLocation, Worker, Car

            Create value objects

            Create Contract

            Add ContractItems
            await _contractRepository.AddAsync(newContract.Value, cancellationToken);
            await _unitOfWork.SaveChangesAsync(cancellationToken);
            Send Email
            _logger.LogInformation("Finished CreateContractCommandHandler");
            return newContract.Value.Id;
        }
        catch (Exception ex)
        {
            _logger.LogError("CreateContractCommandHandler error: {0}", ex.Message);
            return Result.Failure<Guid>(new Error(
                    "Error",
                    ex.Message));
        }

    }
```

Slika 31 Rukovoditelj zahtjevom za kreiranje *Contract* modela

4.3.3. Sloj infrastrukture

Sloj infrastrukture sadrži implementacije sučelja koje su definirane u aplikacijskom sloju ili *Persistence* sloju. Iako ovaj sloj najčešće sadrži i rad sa nekim oblikom pohrane podataka odlučio sam tu značajku izdvojiti u posebni sloj, *Persistence* sloj.

Sloj infrastrukture ima ovisnosti na aplikacijski sloj i *Persistence* sloj, a preko njih i na sloj domene. U projektu sloj infrastrukture sadrži implementacije slanja email-a te upisivanja podataka ako je mjesto pohrane podataka prazno. Popunjavanje mjesta za pohranu se radi kroz metodu *Seed* (Slika 32) te za svaki model iz sloja domene se čita json datoteka sa podacima za specifični model. Ti podaci se na kraju spremaju u mjesto pohrane.

```
public class ApplicationDataSeed
{
    1 reference
    public static async void Seed(IApplicationBuilder applicationBuilder)
    {
        using (var serviceScope = applicationBuilder.ApplicationServices.CreateScope())
        {
            Repositories
            _dbContext!.Database.EnsureCreated();
            var path = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
            var jsonRootPath = Path.Combine(path, "DataSeed\\JsonFiles");
            var jsonSettings = new JsonSerializerSettings
            {
                ContractResolver = new PrivateSetterContractResolver()
            };

            #region Office
            if (!_dbContext.Set<Office>().ToListAsync().Result.Any())
            {
                List<Office> data = new List<Office>();
                using (StreamReader r = new StreamReader(jsonRootPath + "/Office.json"))
                {
                    var json = r.ReadToEnd();
                    data = JsonConvert.DeserializeObject<List<Office>>(json, jsonSettings)!;
                }
                data.ForEach(x => _officeRepository.AddAsync(x));
            }
            #endregion
            Worker
            CarBrand
            CarCategory
            CarModel
            Car
            Extra
            Reservation
            ReservationDetail
            await _unitOfWorkRepository.SaveChangesAsync();
        }
    }
}
```

Slika 32 Punjenje podataka u mjesto pohrane

Već je objašnjeno kako aplikacijski sloj koristi slanje email-a, a to postiže na način da definira sučelje koje sloj infrastrukture implementira. Ovo je moguće zbog inverzije kontrole koja se postiže registriranjem sučelja (koje se nalazi u aplikacijskom sloju) te njegovom implementacijom (koja se nalazi u sloju infrastrukture) kako je prikazano na slici (Slika 33)

```
internal static class Startup
{
    1 reference
    internal static IServiceCollection AddMailing(this IServiceCollection services, IConfiguration config)
    {
        services.Configure<MailSettings>(config.GetSection(nameof(MailSettings)));

        services.AddTransient<IEmailTemplateService, EmailTemplateService>();
        services.AddTransient<IMailService, SmtpMailService>();

        return services;
    }

}
```

Slika 33 Inverzija kontrole za email servis

Kako sam već objasnio da se sloj infrastrukture dijeli i na *Persistence* sloj potrebno je objasniti i što se nalazi u njemu. *Persistence* sloj je rezerviran za rad sa mjestom za pohranu, a u primjeru projekta sam odlučio koristiti relacijsku bazu podataka SQL (engl. *Structured Query Language*). Ovaj sloj sadrži konfiguracije modela iz domenskog sloja u bazu podataka, implementacije sučelja repozitorija iz sloja domene, registriranje inverzne kontrole za repozitorije(Slika 35) te konstante za scheme i nazive tablica u bazi podataka(Slika 34)

```
namespace Persistence.Constants
{
    11 references
    internal static class SchemaNames
    {
        internal const string Catalog = nameof(Catalog);
        internal const string Data = nameof(Data);
        internal const string Sales = nameof(Sales);
    }
    11 references
    internal static class TableNames
    {
        internal const string Offices = nameof(Offices);
        internal const string CarBrands = nameof(CarBrands);
        internal const string CarModels = nameof(CarModels);
        internal const string CarCategories = nameof(CarCategories);
        internal const string Cars = nameof(Cars);
        internal const string Workers = nameof(Workers);
        internal const string Extras = nameof(Extras);
        internal const string Reservations = nameof(Reservations);
        internal const string Contracts = nameof(Contracts);
        internal const string ReservationItems = nameof(ReservationItems);
        internal const string ContractItems = nameof(ContractItems);
    }
}
```

Slika 34 Konstante za scheme i nazive tablica

```
public static class DependencyInjection
{
    1 reference
    public static IServiceCollection AddPersistence(this IServiceCollection services,
        ConfigurationManager configuration)
    {

        string connectionString = configuration.GetConnectionString("Database")!;


        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(connectionString));

        services.AddScoped<IOfficeRepository, OfficeRepository>();
        services.AddScoped<ICarCategoryRepository, CarCategoryRepository>();
        services.AddScoped<ICarBrandRepository, CarBrandRepository>();
        services.AddScoped<ICarModelRepository, CarModelRepository>();
        services.AddScoped<ICarRepository, CarRepository>();
        services.AddScoped<IExtrasRepository, ExtrasRepository>();
        services.AddScoped<IWorkerRepository, WorkerRepository>();
        services.AddScoped<IReservationRepository, ReservationRepository>();
        services.AddScoped<IReservationItemRepository, ReservationItemRepository>();
        services.AddScoped<IContractItemRepository, ContractItemRepository>();
        services.AddScoped<IContractRepository, ContractRepository>();
        services.AddScoped<IUnitOfWork, UnitOfWork>();


        return services;
    }
}
```

Slika 35 Inverzija kontrole za repozitorije

Za objektno-relacijsko mapiranje (engl. *Object–Relational Mapping*, skraćenica ORM) sam koristio EntityFrameworkCore biblioteku s kojim sam najbolje upoznat, a uz pomoć kojeg mogu kreirati konfiguracije i jako efikasne i jednostavne upite na bazu podataka kroz repozitorije. Samo konfiguriranje je vidljivo na slici (Slika 36). Svaki element modela vozila se može detaljno konfigurirati uz opcije kao što su maksimalna duljina, obavezno polje, broj decimala, pretvaranje elementa koji su tipa *enum,* a uz konfiguriranje elemenata mogu se jako jednostavno konfigurirati i veze među modelima.

```
internal class CarConfiguration : IEntityTypeConfiguration<Car>
{
    0 references
    public void Configure(EntityTypeBuilder<Car> builder)
    {
        builder.ToTable(TableNames.Cars, SchemaNames.Catalog);

        builder.HasKey(x => x.Id);

        builder.Property(x => x.NumberPlate)
            .HasMaxLength(10)
            .IsRequired(true);

        builder.Property(x => x.Kilometers)
            .HasColumnType("decimal(18,1)")
            .IsRequired(true);

        builder.Property(x => x.Image)
            .IsRequired(false);

        builder.Property(r => r.Status)
            .HasConversion<string>()
            .IsRequired(true);

        builder.Property(r => r.FuelType)
            .HasConversion<string>()
            .IsRequired(true);

        builder.HasOne<CarModel>(x => x.CarModel)
            .WithMany(x => x.Cars)
            .HasForeignKey(x => x.CarModelId)
            .OnDelete(DeleteBehavior.Restrict);

        builder.HasOne<Office>(x => x.Office)
            .WithMany(x =>x.Cars)
            .HasForeignKey(x => x.OfficeId)
            .OnDelete(DeleteBehavior.Restrict);
    }
}
```

Slika 36 Konfiguracija modela vozila

Konfiguriranje je isto i za korijene agregata i za entitete, ali konfiguriranje vrijednosnih objekata je nešto drugačije. Iako korijen aggregaat/entitet imaju polje koje je vrijednosni objekt, u bazi će se stvoriti polja koja vrijednosni objekt sadrži. Ako uzmemo za primjer konfiguraciju sa slike (Slika 37) gdje su prikazana dva vrijednosna objekta, Email i *Card.* Vrijednosni objekt Email sadrži samo jedno polje *Value* te se za ovaj tip vrijednosnog objekta može raditi pretvaranje koje koristi metodu vrijednosnog objekta Email za kreiranje istoga. S druge strane vrijednosni objekt *Card* je dosta složeniji te se sastoji od pet polja koja se konfiguriraju na drugačiji način od prethodnog vrijednosnog objekta. Prilikom konfiguriranja modela *Contract* koji sadrži vrijednosni objekt *Card* u bazi će se kreirati svih pet njegovih polja te se nigdje neće vidjeti da je to vrijednosni objekt *Card*, ali

prilikom dohvaćanja tih istih podataka oni će se preslikati u polje *Card* i kao takvi će se koristiti kroz sustav.

```
builder.Property(x => x.Email)
    .HasConversion(x => x.Value, v => Email.Create(v).Value)
    .IsRequired(true);

builder.OwnsOne(x => x.Card, card =>
{
    card.Property(x => x.CardName)
    .HasColumnName("CardName")
    .IsRequired(false);

    card.Property(x => x.CardNumber)
    .HasColumnName("CardNumber")
    .IsRequired(false);

    card.Property(x => x.CVV)
    .HasColumnName("CVV")
    .IsRequired(false);

    card.Property(x => x.CardDateExpiration)
    .HasColumnName("CardDateExpiration")
    .IsRequired(false);

    card.Property(x => x.CardYearExpiration)
    .HasColumnName("CardYearExpiration")
    .IsRequired(false);
});
```

Slika 37 Konfiguracija vrijednosnih objekata

Osim konfiguriranja modela za bazu podataka ovaj sloj sadrži i implementacije sučelja repozitorija koja se nalaze u sloju domene (Slika 38). Potrebno je da repozitorij naslijedi sučelje koje će onda i implementirati te vratiti ono što je definirano u sučelju. Sučelja i implementacije su poprilično jednostavna u ovom projektu, a implementacije su vezane za kreiranje modela, dohvaćanje svih modela nekog tipa, dohvaćanje modela prema njegovom identifikatoru te provjeru postojanja nekog polja modela. Sučelje i implementaciju je potrebno registrirati kako je prethodno pokazano, a na taj način se osigurava da aplikacijska razina koja poziva sučelje ne zna na koji način će implementacija odraditi zahtjev. Ovom logikom lako se može promijeniti i tip baze podataka, a da ni sloj domene ni aplikacijski sloj ne moraju voditi brigu o novom načinu rada.

```
internal sealed class CarBrandRepository : ICarBrandRepository
{
    private readonly ApplicationDbContext _dbContext;
    0 references
    public CarBrandRepository(ApplicationDbContext dbContext)
    {
        _dbContext = dbContext;
    }
    3 references
    public async Task AddAsync(CarBrand carBrand, CancellationToken cancellationToken = default)
    {
        await _dbContext.Set<CarBrand>().AddAsync(carBrand, cancellationToken);
    }
    2 references
    public async Task<bool> AlreadyExists(CarBrandName carBrandName,
        CancellationToken cancellationToken = default)
    {
        var carBrand = await _dbContext.Set<CarBrand>()
            .Where(x => x.Name == carBrandName)
            .SingleOrDefaultAsync(cancellationToken);

        return carBrand != null ? false : true;

    }
    2 references
    public async Task<List<CarBrand>> GetAllAsync(CancellationToken cancellationToken = default)
    {
        return await _dbContext.Set<CarBrand>()
            .ToListAsync(cancellationToken);

    }
    4 references
    public async Task<CarBrand?> GetByIdAsync(Guid id, CancellationToken cancellationToken = default)
    {
        var carBrand = await _dbContext.Set<CarBrand>()
            .Include(x => x.CarModels)
            .Where(x => x.Id == id)
            .SingleOrDefaultAsync(cancellationToken);
        return carBrand;
    }
}
```

Slika 38 Implementacija sučelja repozitorija

4.3.4. Prezentacijski sloj

Prezentacijski sloj je sami vrhunac čiste arhitekture te je ujedno i pokretač cijelog sustava. Prezentacijski sloj nije strogo definiran tipom projekta. Može biti *Console Application*, *Windows Forms*, mrežna aplikacija, mobilna aplikacija. U primjeru projekta je korišten ASP.NET Core Web API projekt zbog jednostavnosti u izradi i jednostavnog prikaza. Ovaj sloj ima ovisnosti o aplikacijskom sloju i sloju infrastrukture preko kojih ima i vezu prema *Persistence* sloju i sloju domene. Poziva konfiguracije svih slojeva te ih registrira. Sadrži modele koji predstavljaju ugovore zahtjeva, a koje se koristi u rutama uz pomoć kojih se šalju zahtjevi prema aplikacijskom sloju. Ti modeli najčešće odgovaraju zahtjevima iz aplikacijskog sloja te ih se jednostavno mapira jedne u druge.

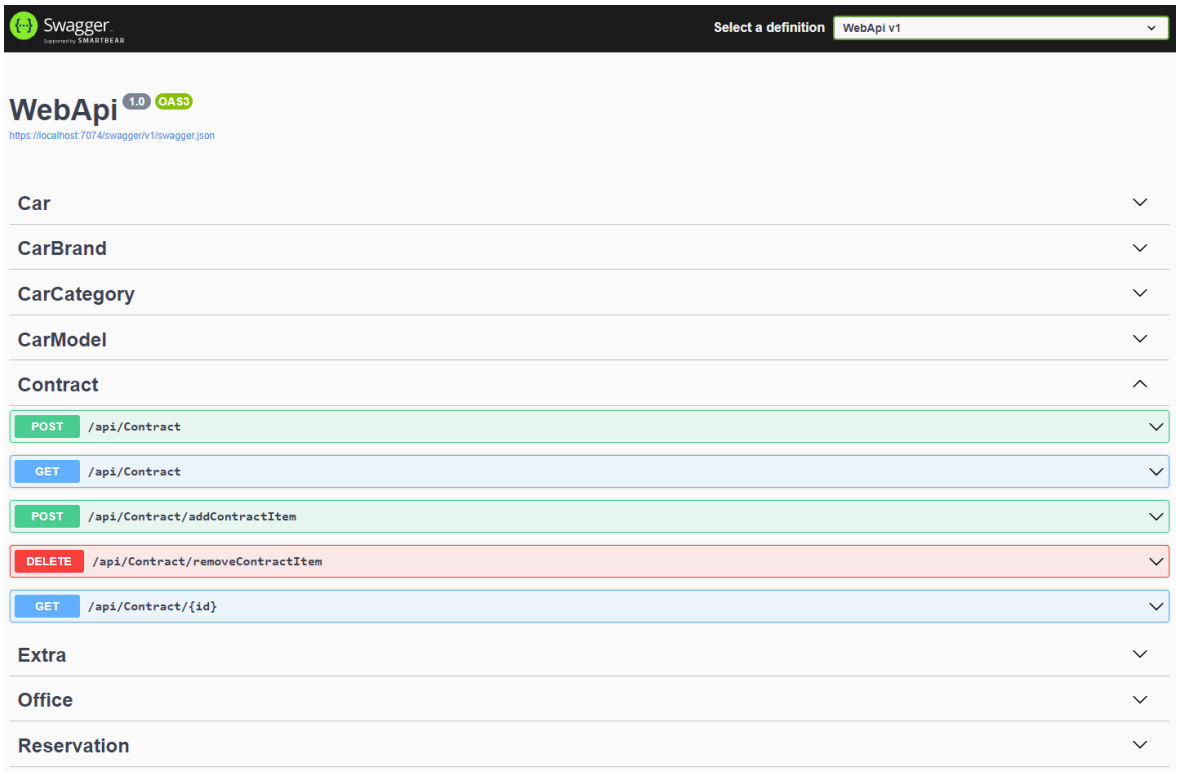
Razmatrajući primjer sa slike (Slika 39) koja predstavlja kreiranje ugovora vidljivo je da ruta prima model iz kojeg se izvlače podaci koji se prosljeđuju u naredbu zahtjeva iz

aplikacijskog sloja gdje će se odraditi sva prethodno objašnjena logika za kreiranje ugovora u kojoj aplikacijski sloj koristi modele sloja domene te inverzijom kontrole upisuje i dohvaća podatke kroz implementacije repozitorija iz *Persistence* sloja. Zahtjev se šalje koristeći MediatR, a odgovor se provjerava te sukladno njegovom statusu se vraća ili greška ili točan podatak iz aplikacijskog sloja. Na ovaj način se postiže jako jednostavna komunikacija među slojevima koja je jednostavna za održavanje.

```
[HttpPost]
3 references
public async Task<IActionResult> Create(CreateContractRequest request)
{
    _logger.LogInformation("Started ContractController.Create");
    var extrases = new List<ExtrasModel>();
    if (request.Extras != null || request.Extras.Any())
    {
        request.Extras.ForEach(x => extrases.Add(new ExtrasModel(x.ExtraId, x.Quantity)));
    }
    var commmand = new CreateContractCommand(
        request.DriverFirstName,request.DriverLastName,request.Email,
        request.PickUpDate, request.DropDownDate,request.PickUpOfficeId,
        request.DropDownOfficeId,request.CarId,request.DriverLicenceNumber,
        request.DriverIdentificationNumber,request.CardType, request.PaymentMethod,
        request.ReservationId,request.WorkerId,request.CardName,request.CardNumber,
        request.CVV,request.CardDateExpiration,request.CardYearExpiration,extrases);

    var response = await Sender.Send(commmand);
    _logger.LogInformation("Finished ContractController.Create");
    if (response.IsFailure)
    {
        return HandleFailure(response);
    }
    return CreatedAtAction(
        nameof(Create),
        new { id = response.Value },
        response.Value);
}
```

Slika 39 Ruta za kreiranje *Contract*-a

Ova vrsta projekta za prezentacijski sloj koristiti ugrađenu biblioteku Swagger preko koje je moguće upravljati cijelim sustavom (Slika 40). U prikazu su pokazani kontroleri za svaki od agregata, a unutar kontrolera se nalaze rute za manipuliranjem sustavom. Za agregat *Contract* postoje rute za kreiranje ugovora, dodavanje dodataka na već kreirani ugovor, dohvaćanje svih ugovora te dohvaćanje ugovora prema njegovom identifikatoru.



Slika 40 Upravljanje projektom kroz Swagger biblioteku

Zaključak

U ovom radu objašnjen je jedan od pristupa u razvoju programske podrške, odnosno dizajniranje upravljano domenom. Kako bi razvoj bio što uspješniji, predstavljene su različite tehnike objektno-orijentiranog programiranja. Sve navedene tehnike i obrasci su zapravo samo pomoć u izradi dizajna upravljanog domenom, ali pravo značenje DDD-a je u načinu razmišljanja i pristupu rješavanja problema koje kao rezultat ima održiv sustav.

DDD u središte stavlja domenu i njezino modeliranje. Model predstavlja presliku domene iz stvarnog svijeta te je potrebno da se proces modeliranja vodi kroz međusobnu komunikaciju stručnjaka domene i stručnjaka razvoja programske podrške. Što je model bolje definiran domenom, moguće je napraviti bolji sustav, a izradom boljeg sustava moguće je pospješiti razumijevanje same domene.

Cilj ovog diplomskog rada je bio prikazati procese i komponente u izradi sustava temeljenog na DDD-u. Kako bi se uopće odabrao ovaj pristup potrebno je imati kompleksnu domenu inače ovaj pristup gubi svoju vrijednost te zadaje nepotrebne poslove. Kao i svaki pristup i ovaj ima svoje prednosti i nedostatke. Prednosti su stvaranje sveprisutnog jezika zbog dijeljenja i razumijevanja znanja o domeni, stvaranje čistog koda i lako održivog sustava, te jednostavno uključivanje dodatnih osoba u proces izrade sustava. S druge strane ovaj pristup nije savršen te postoje nedostaci u njegovom korištenju. Kao prvi nedostatak naveo bih obveznost barem jednog stručnjaka domene u timu. Sljedeći nedostatak je što razvojni programeri nisu stručnjaci za domenu stoga moraju uložiti dodat trud u razumijevanje domene i komunikaciju sa stručnjacima domene. DDD najčešće koriste veće organizacije sa većim timovima jer je klijentu potrebno što prije isporučiti rješenje, a ovaj pristup je relativno spor zbog svih navedenih procesa koji su potrebni u korištenju DDD-a.

Literatura

Abrahamsson, P. (2017). *Agile Software Development Methods: Review and Analysis.*

Agile Software Development. (2023). Dohvaćeno iz Wikipedia: https://en.wikipedia.org/wiki/Agile_software_development

Avram, A., & Marinescu, F. (2006). *Domain Driven Design Quickly*. United States of America.

Barkijević, F. (2021). *Životni ciklus programskog proizvoda.*

Batista, F. D. (27. April 2019). *Developing the ubiquitous language*. Dohvaćeno iz https://medium.com/@felipefreitasbatista/developing-the-ubiquitous-language-1382b720bb8c

Clean Arhitecture : Part 2 - The Clean Architecture. (6. October 2017). Dohvaćeno iz Crosp: https://crosp.net/blog/software-architecture/clean-architecture-part-2-the-clean-architecture/

Dhaduk, H. (4. July 2020). *10 Software Arhitecture Patterns You Must Know About*. Dohvaćeno iz Simform: https://www.simform.com/blog/software-architecture-patterns/

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.*

Neto, J. L. (7. March 2023). *How we used Event Storming Meetings for enabling Software Domain-Driven Design*. Preuzeto 10. Srpanj 2023 iz Medium: https://medium.com/building-inventa/how-we-used-event-storming-meetings-for-enabling-software-domain-driven-design-401e5d708eb

Richards, M. (2022). *Software Architecture Patterns.*

Vuollet, P. (24. October 2018). *Hexagonal Architecture: What Is It and How Does It Work?* Dohvaćeno iz DZone: https://dzone.com/articles/hexagonal-architecture-what-is-it-and-how-does-it

Tablica slika

Skraćenice

DDD *Domain Driven Design,* Dizajn upravljan domenom

XP *Extreme Programming,* Ekstremno programiranje

DIP *Dependency Inversion Principle,* Princip inverzne ovisnosti

IoC *Inversion of Control,* Inverzija kontrole

OIB Osobni identifikacijski broj

CQRS *Command and Query Resposibility Segregation,* pravilo razdvajanja odgovornosti kroz naredbe i upite

ORM *Object–Relational Mapping,* Objektno-relacijsko mapiranje

API *Application Programming Interface,* Aplikacijsko programsko sučelje

Privitak

Instalacija programske podrške

Projekt je potrebno klonirati sa Git-a uz pomoć naredbe:

```
git clone https://github.com/mmaras-pmfst/CarRentalDDD-MasterThesis.git
```

Nakon toga je potrebno pokrenuti datoteku *Solution.sln* kojim će se dobiti svi projekti u slojevima u okruženju kao što je Visual Studio 2022. Također ovaj projekt zahtjeva .NET6 verziju kako bi svi projekti ispravno funkcionirali.

Pokretanje programa

Potrebno je postaviti početni projekt na način da se odabere WebApi projekt, desni klik miša i odabrati opciju *Set as Startup Project.*

U projektu WebApi potrebno je otvoriti datoteku *appsetings.json* te postaviti novu vrijednost u polje ConnectionStrings → Database. Važno je napomenuti da projekt radi samo sa SQL bazom podataka.

```
"ConnectionStrings": {
    "Database":
"Server=(LocalDb)\\MSSQLLocalDB;Database=CleanArhitecture0605;Trusted_Con
nection=True;"
  }
```

Kako bi i slanje email-a funkcioniralo potrebno je na stranici https://ethereal.email/ kreirati testni račun te u datoteku *appsetings.json* unijeti Password i UserName

```
"MailSettings": {
    "DisplayName": "Marko Maras",
    "From": "mmaras@diplomski.net",
    "Host": "smtp.ethereal.email",
    "Password": "vAKmWQB8CyPUBg8rBQ",
    "Port": 587,
    "UserName": "adaline.pfannerstill49@ethereal.email"
  }
```

Ako je veza na bazu podataka dobro napisana može se pokreniti progam. Program će sam kreirati bazu podataka na bazi koristeći navedeni ConnectionString te će se unijeti početni podaci u sve tablice.

Korištenje programa

Nakon pokretanja otvorit će se Swagger sučelje u pregledniku. Ako se ne otvori automatski može se koristiti veza https://localhost:7074/swagger/index.html nakon čega se slobodno mogu koristiti sve rute za rad sa sustavom.