SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI PROJEKT

RAZVOJ VOĐEN DOMENOM NA PRIMJERU UPRAVLJANJA STAMBENIM ZGRADAMA

Željko Tepšić

Mentor: Prof.dr.sc. Nikola Bogunović

Zagreb, 2010.

1. Uvod 4

2. Razvoj vođen domenom (Domain Driven Development) 5

2.1 Sveprisutni jezik 6

2.2 Građevni blokovi DDD-a 6

2.2.1 Slojevita arhitektura 8

2.2.2 Entiteti (engl. entities) 10

2.2.3 Vrijednosni objekti (engl. value objects) 12

2.2.4 Servisi (engl. services) 14

2.2.5 Moduli (engl. modules) 15

2.2.6 Agregati (engl. agregates) 16

2.2.7 Tvornice (engl. factories) 19

2.2.8 Repozitoriji (engl. repositories) 22

2.3 Kontinuirano refaktoriranje 24

2.4 Omeđen kontekst (engl. bounded context) 25

2.5 Razvoj vođen testiranjem (Test Driven Development) 26

2.5.1 Mock objekti 27

2.6 Dependency injection i Inversion of Control 28

3. Domena upravljanja stambenim zgradama 30

3.1 Opis procesa upravljanja i održavanja stambenih objekata 30

3.1.1 Temeljni dokumenti o upravljanju zgradama 31

3.1.2 Vlasništvo zgrade ... 32

3.1.3 Upravitelj ... 33

3.1.4 Predstavnik suvlasnika ... 34

3.1.5 Pričuva ... 35

3.2 Specifikacija zahtjeva ... 36

4. Izrada objektnog modela za Upravljanje stambenim zgradama primjenom DDD 38

4.1 Apstrakcije ... 38

4.2 Osobe i uloge ... 40

4.3 Zakonodavstvo ... 43

4.4 Upravljanje zgradom ... 46

4.5 Prijava kvara ... 47

4.6 Financije ... 48

5. Zaključak ... 51

6. Diplomski rad ... 52

7. Literatura ... 52

1. Uvod

Programska potpora je instrument stvoren radi rješavanja kompleksnih problema modernog života. Programska potpora mora biti praktična i korisna, jer u suprotnom ulaganje vremena i resursa u njezino stvaranje nema smisla.

Značenje riječi model u hrvatskom jeziku je vrlo slično značenju riječi model u računarskom rječniku: reprezentacija stvari iz stvarnog svijeta. U informacijskim sustavima, stvarni svijet predstavljamo objektima koji su imenovani prema konceptima s kojima se susrećemo svaki dan. Ti objekti imaju svojstva (atribute) i ponašanja slična onima koja pronalazimo u stvarnome svijetu. Pojednostavljeno rečeno, više takvih objekata čine model.

Bez modela, informacijski sustavi nemaju nikakvu vrijednost. Često puta se previše vremena troši samo na tehnologiju i implementaciju, a kontekst, odnosno domena se stavlja u drugi plan. Zbog takvog lošeg pristupa informacijski sustavi gube smisao, odnosno ugrađuju se funkcionalnosti koje su ili previše složene ili potpuno beskorisne jer se izgubio fokus na najbitnije – domenu. Stoga, da bi smo razvili dobru programsku potporu potrebno je znati radi čega se razvija programska potpora.

U ovome diplomskom projektu predstaviti ćemo razvoj vođen domenom *engl. domain driven development – DDD)*, temeljen na knjizi Erica Evansa: Domain-Driven Design: Tackling Complexity in the Heart of Software, koji u središte postavlja kontekst, odnosno domenu čiji se problem rješava. Primjena razvoja vođenog domenom biti će prikazana na primjeru upravljanja stambenim zgradama. Programski jezik koji će biti korišten za reprezentaciju modela je C#.

2. Razvoj vođen domenom (Domain Driven Development)

Razvoj programske potpore je često povezan sa automatizacijom procesa koji postoje u stvarnome svijetu ili sa pružanjem rješenja za stvarne poslovne probleme. Ta automatizacija procesa i poslovni problemi čine domenu programske potpore. Programska potpora potječe i usko je povezana sa domenom.

Programska potpora sastoji se od programskog koda. Ponekad trošimo previše vremena na kod i gledamo programsku potporu kao skup objekata i metoda.

Da bismo razvili dobru programsku potporu potrebno je znati radi čega se razvija programska potpora. Nije moguće napraviti informacijski sustav banke ukoliko ne razumijemo što je uopće banka i bankarstvo, odnosno potrebno je razumjeti domenu banke.

Razvoj vođen domenom, odnosno domain-driven design, je pristup za razvoj kompleksne programske potpore koji duboko povezuje implementaciju sa razvijajućim modelom jezgre poslovnih koncepata. Glavne pretpostavke DDD su sljedeće:

- Glavni fokus projekta postavlja se na jezgru domene i domensku logiku.
- Složeni dizajn temelji se na modelu.
- Uspostavljanje kreativne kolaboracije između eksperata domene i programera, odnosno razvojnika, radi što bližeg približavanja konceptualnoj srži problema.

DDD nije tehnologija niti metodologija. DDD pruža strukturu prakse i terminologije za donošenje odluka u dizajnu koje fokusiraju i ubrzavaju projekte programske potpore koji se bave složenim domenama.

2.1 Sveprisutni jezik

Apsolutno je nužno razvijati model domene uz pomoć programera i razvojnika sa ekspertima domene. Međutim, takav pristup obično donosi inicijalne poteškoće zbog komunikacijske barijere.

Programeri i razvojnici misle samo na razrede, metode, algoritme, obrasce i pokušavaju uvijek upariti koncepte iz stvarnog svijeta sa programskim konceptima. Eksperti domene obično ne znaju ništa o programskim i računalnim konceptima kao što su programske knjižnice, razvojni okviri, perzistencija i slično. Oni znaju samo ono u čemu su stručni i o tome govore svojim žargonom koji ponekad nije potpuno jasan vanjskim ljudima.

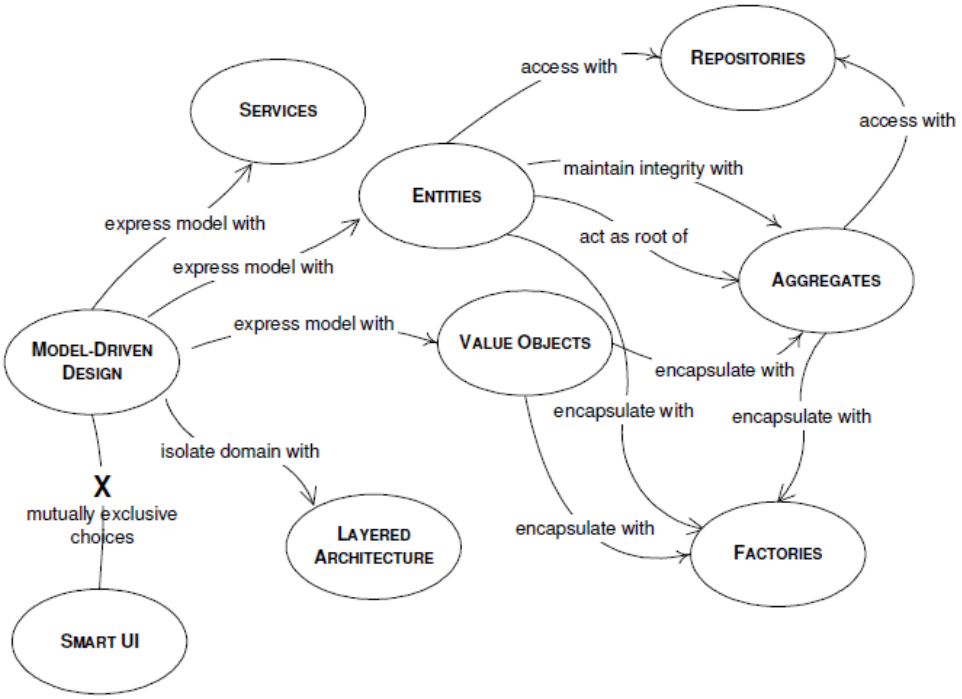
Prilikom izrade modela, potrebno je komunicirati radi razmjene ideja radi modela i radi elemenata koji su uključeni u model. Komunikacija na ovoj razini je iznimno važna za uspjeh projekta.

Jedan od osnovnih principa razvoja vođenog domenom je korištenje jezika baziranog na modelu. Potrebno je koristiti model kao kostur jezika, odnosno model se mora moći izreći jezikom domene. Takav jezik je nužno koristiti konzistentno prilikom svake komunikacije, u svim formama pa i u kodu. Iz tog razloga jezik se naziva sveprisutni jezik.

Sveprisutni jezik povezuje sve dijelove dizajna i stvara pretpostavku da dizajnerski tim odlično funkcionira.

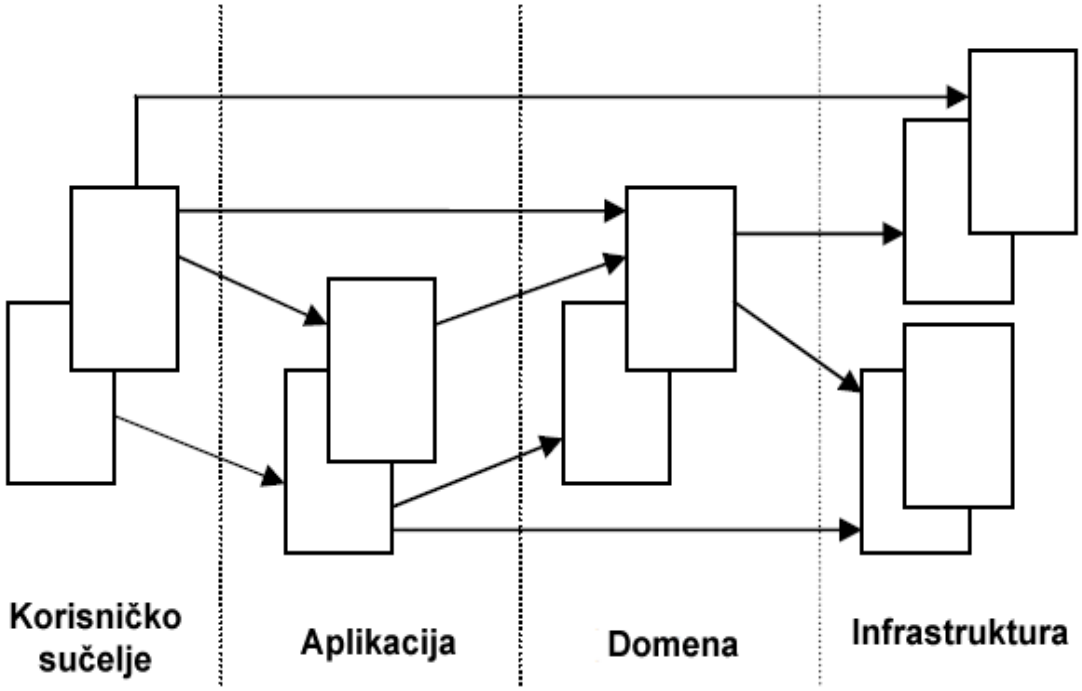
2.2 Građevni blokovi DDD-a

Sljedeća podpoglavlja će prikazati najvažnije obrasce koji se koriste u DDD-u. Svrha tih obrazaca je predstaviti ključne elemente modeliranja objekata i dizajniranja programske potpore sa stajališta DDD-a. Sljedeći dijagram predstavlja mapu obrazaca i njihovih međusobnih veza.



Slika 1 - mapa obrazaca DDD-a i njihovih međusobnih veza

2.2.1 Slojevita arhitektura



Slika 2 - slojevita arhitektura

Kada razvijamo nekakav programski sustav, veliki dio te aplikacije nije direktno vezan sa domenom, ali domena čini sastavni dio infrastrukture ili služi aplikaciji kao takva. Moguće je da domena bude mala u usporedbi sa ostatkom aplikacije s obzirom na to što tipična aplikacija sadrži veliku količinu koda koji je vezan za pristup bazi podataka, datotekama, mrežnom pristupu, interakciji sa korisnicima itd.

U objektno orijentiranom programu, korisničko sučelje, baza podataka i ostali podupirući kod se često puta ugrađuje u poslovne objekte. Poslovna logika se smješta u korisnička sučelja i kod za pristup bazi podataka.

Međutim, kada je domenski orijentiran kod pomiješan sa ostalim slojevima, postaje ekstremno teško razumjeti i razmišljati o domeni koju modeliramo. Čak i površne promjene nad korisničkim sučeljem mogu promijeniti poslovnu logiku. Isto tako, ukoliko bi željeli promijeniti poslovna pravila to bismo morali napraviti na velikom broju mjesta, a niti onda ne bismo bili sigurni da smo sve pokrili.

U tom slučaju implementacija koherentnih, modelom vođenih objekata, postaje nepraktična, a automatsko testiranje nemoguće.

Iz tih se razloga složeni programski sustavi particioniraju u slojeve. Slojeve je potrebno razvijati tako da je svaki pojedini sloj kohezivan i da ovisi samo o sloju ispod njega. Potrebno je slijediti standardne arhitekturalne obrasce radi postizanja što manje međuovisnosti o višim slojevima. Sav kod povezan sa modelom domene potrebno je smjestiti u jedan sloj i izolirati ga od korisničkog sučelja, aplikacije i infrastrukture. Objekti domene koji su oslobođeni od odgovornosti za vlastito prikazivanje i perzistenciju sada mogu biti fokusirani na izražavanje domenskog modela.

Najčešće arhitekturalno rješenje za razvoj vođen domenom sastoji se od četiri konceptualna sloja:

- **Korisničko sučelje (prezentacijski sloj)** – odgovoran za prezentiranje informacije korisniku i interpretaciju korisničkih naredbi.
- **Aplikacijski sloj** - je tanki sloj koji koordinira aplikacijske aktivnosti. Ne sadrži poslovnu logiku. Ne čuva stanje poslovnih objekata, ali može čuvati stanje napredovanja aplikacijskih zadataka.
- **Domenski sloj** – sadrži informacije o domeni. To je srce poslovnih informacijskih sustava. Stanje poslovnih objekata se čuva ondje. Perzistencija poslovnih objekata i njihovih stanja se delegira infrastrukturnom sloju.
- **Infrastrukturni sloj** – se ponaša kao podupiruća knjižnica za sve ostale slojeve.

Vrlo je važno podijeliti aplikaciju u odvojene slojeve i uspostaviti pravila njihove interakcije. Ukoliko kod nije jasno podijeljen u slojeve ubrzo će postati zamršen što otežava održavanje. Jedna jednostavna promjena u jednom dijelu koda može imati neočekivane i neželjene rezultate u drugim dijelovima aplikacije.

Domenski sloj bi trebao biti fokusiran na jezgru domene koja se modelira. Isto tako sloj domene ne bi trebao biti upleten u infrastrukturne aktivnosti. Dok s druge strane

korisničko sučelje ne bi se trebalo baviti poslovnom logikom niti zadacima koji normalno pripadaju infrastrukturnom sloju. Aplikacijski sloj je potreban u mnogim slučajevima. Mora postojati upravitelj poslovne logike koji nadgleda i koordinira sveukupnu aktivnost aplikacije.

Primjerice, tipična interakcija aplikacije, domene i infrastrukture bi mogla izgledati ovako. Korisnik želi rezervirati let i traži aplikacijski servis u aplikacijskom sloju da obavi rezervaciju leta. Aplikacijski sloj dohvaća relevantne domenske objekte iz infrastrukturnog sloja i poziva odgovarajuće metode nad njima, npr. provjerava da li je let moguće rezervirati. Kada su domenski objekti obavili sve provjere i osvježili svoja stanja, aplikacijski servis perzistira te objekte preko infrastrukturnog sloja.

2.2.2 Entiteti (engl. entities)

Mnogi objekti nisu definirani samo svojim atributima, odnosno svojstvima, već su definirani kroz niti kontinuiteta i identiteta. Većina stvari u stvarnome svijetu su definirane kroz identitet. Primjerice osoba posjeduje identitet koji se proteže od rođenja pa do smrti, možda čak i dalje. Atributi osobe se mijenjaju, a neki čak i nestanu. Kod osobe ime se može promijeniti, godine se mijenjaju, financijsko stanje isto se mijenja. Dakle ne postoji niti jedan atribut koji nije moguće promijeniti, no međutim identitet osobe ne podliježe promjenama. Pitanje koje je moguće postaviti je: da li je osoba ista sa 5 ili 55 godina? Odgovor je da se identitet osobe nije promijenio, ali njezini atributi jesu.

Svaki od objekata koji nisu primarno definirani svojim atributima predstavlja nit identiteta koja prolazi kroz vrijeme te u određenom trenutku poprima različitu reprezentaciju. Ponekad takav objekt mora biti jednak drugome objektu čak i kada se atributi razlikuju. Isto tako, neki objekt se mora razlikovati od drugih objekata ukoliko imaju jednake atribute. Zamjena identiteta može dovesti do korupcije podataka.

Objekt koji je primarno određen svojim identitetom naziva se **entitet**. Entiteti imaju specijalnu važnost prilikom modeliranja i dizajniranja. Životni ciklus entiteta može radikalno promijeniti formu i sadržaj objekta, ali identitet i kontinuitet objekta mora

biti očuvan. Identitet objekta mora biti tako definiran da se može učinkovito pratiti. Definicije razreda, odgovornosti, atributa i asocijacije kod entiteta moraju određivati što taj entitet predstavlja, tko je on, a ne samo koje atribute sadrži. Ukoliko smjestimo entitete, koji se ne mijenjaju tako radikalno ili nemaju kompliciran životni ciklus, u semantičku kategoriju to vodi k jasnim modelima i robusnoj implementaciji.

Većina entiteta u informacijskim sustavima nisu samo osobe ili entiteti u uobičajenom smislu, već su to svi objekti koji u svom životnom ciklusu moraju zadržati svoj kontinuitet te nisu definirani samo svojim atributima. Entitet može biti osoba, grad, auto ili bankovna transakcija.

Primjerice, dva depozita na isti račun, u istom danu, sa istim iznosom su dva entiteta. Dok s druge strane, objekti koji opisuju iznos su instance novčanog objekta i ne predstavljaju entitete.

Ponekad je bitno da identitet bude valjan izvan sustava, a ponekad je bitno samo da bude valjan unutar sustava.

2.2.2.1 Modeliranje entiteta

Prirodno je razmišljati o atributima i ponašanju objekta prilikom modeliranja. Međutim, osnovna odgovornost entiteta je uspostavljanje kontinuiteta. Stoga, umjesto fokusiranja na atribute i ponašanja potrebno je svesti entitet na osnovne karakteristike koje su potrebne za njegovu identifikaciju i određenost. Entitetu se dodaje samo ona ponašanja koja su nužna za njegov koncept i oni atributi koji su potrebni za ta ponašanja. Ostale atribute i ponašanja potrebno je premjestiti u druge objekte koji su povezani sa entitetom. Neki od drugih objekata mogu biti drugi entiteti ili vrijednosni objekti (*engl.* value objects).

Svaki entitet mora imati operativni način za iskazivanje svojeg identiteta u odnosu na drugi objekt, pa čak i ukoliko imaju potpuno jednake karakteristike. Identifikacijski atribut, odnosno njegova vrijednost mora biti jedinstvena unutar sustava, pa makar taj sustav bio distribuirani ili ako su objekti arhivirani.

Mnogi objektno orijentirani jezici imaju „identifikacijske“ operacije koje određuju da li dvije reference pokazuju na isti objekt tako da uspoređuju memorijske lokacije. No takav način identifikacije objekata je previše krhak. Primjerice, kod perzistencije objekata svaki puta kada se objekt dohvaća iz baze podataka nova instanca objekta se stvara, a time se gubi inicijalni identitet, memorijska lokacija – referenca. Svaki puta kada se objekt šalje putem mreže stvara se nova instanca na odredištu te se opet gubi inicijalni identitet.

Stoga se postavlja jedno od fundamentalnih pitanja: Kako znati da dva objekta predstavljaju isti konceptualni objekt? Definicija identiteta proizlazi iz modela, odnosno definiranje identiteta zahtjeva poznavanje domene.

Za osiguranje identiteta entiteta potrebno je definirati jedinstveni identifikator i pridodati ga objektu kao atribut. Identifikator može biti generiran automatski u sustavu ili može biti definiran vanjskim faktorom (npr. JMBG ili OIB). Bitno je da entitet osigura nepromjenjivost vrijednosti koja određuje njegov identitet. Dakle, jednom određen identitet/identifikator nikada se ne može promijeniti.

2.2.3 Vrijednosni objekti (engl. value objects)

Mnogi objekti nemaju konceptualni identitet. Ti objekti samo opisuju karakteristiku neke stvari.

Objekt koji predstavlja opisni aspekt domene bez konceptualnog identiteta zove se **vrijednosni objekt** (*engl.* value object). Vrijednosni objekti predstavljaju elemente u dizajnu o kojima mislimo na način što oni predstavljaju, a ne tko su oni.

Vrijednosni objekti mogu se sastojati od drugih objekata. Mogu čak i referencirati entitete. Često se prenose kao parametri poruke između objekata. Vrijednosni objekti su tranzijentne prirode, odnosno stvaraju se radi neke operacije i nakon toga mogu se odbaciti. Mogu se koristiti kao atributi entiteta ili nekih drugih vrijednosnih objekata.

Atributi koji sačinjavaju vrijednosni objekt moraju tvoriti konceptualnu cjelinu. Primjerice, ulica, grad i poštanski kod mogu predstavljati vrijednosni objekt „adresa“.

2.2.3.1 Modeliranje vrijednosnih objekata

Kod vrijednosnih objekata nije bitno koju instancu objekta imamo. Taj nedostatak ograničenja nam daje slobodu kod dizajniranja da pojednostavnimo dizajn ili optimiziramo za performanse. To uključuje donošenje odluka o kopiranju, dijeljenju i nepromjenjivosti.

Ukoliko dvije osobe imaju jednako ime, to ne znači da se radi o istoj osobi niti da je svejedno s kojom osobom zaista radimo. Ali objekt koji predstavlja ime je razmjenjiv i sasvim je svejedno s kojom instancom radimo sve dok je ime ispravno. Objekt „ime“ je moguće kopirati iz jednog objekta „osoba“ u drugi. U biti, dva objekta „osoba“ ne moraju imati vlastitu instancu objekta „ime“, već mogu jedan te isti objekt međusobno dijeliti. Međutim, takvo ponašanje je ispravno sve dok netko ne promijeni vrijednost objekta „ime“ u nekom od objekata „osoba“ jer će se tada vrijednost imena promijeniti i drugim osobama. Da bi se zaštitili od toga, odnosno da bi se vrijednosni objekti mogli sigurno dijeliti, oni moraju biti nepromjenjivi (*engl.* immutable). Vrijednosti nepromjenjivih objekata je moguće promijeniti samo njihovom zamjenom.

Isti problemi se pojavljuju kada jedan objekt preda vrijednost svojeg atributa drugome objektu kao argument ili povratnu vrijednost. Svašta se može dogoditi takvome objektu kada nije pod kontrolom svojeg vlasnika. Vrijednosni objekt se može promijeniti na način da narušava integritet svojeg vlasnika tako da krši vlasnikove invarijante. Takav problem se rješava tako da je objekt nepromjenjiv ili tako da se predaje kopija objekta. Jedna od prednosti koncepta vrijednosnih objekata i nepromjenjivosti je njihov učinak na performanse gdje samo jedan vrijednosni objekt predstavlja vrijednost tisućama drugih objekata u usporedbi sa time da svaki od tisuće objekata ima svoju vlastitu instancu vrijednosnog objekta.

Isplativost kopiranja naspram dijeljenja ovisi o implementacijskom okruženju. Premda kopije objekata mogu zagušiti sustav sa velikim brojem instanci, dijeljenje može biti usko grlo u distribuiranim sustavima. Kada se kopija šalje između dva računala tada se šalje jedna poruka i objekti nezavisno žive na svakome računalu. Ali ukoliko se jedna instanca dijeli, onda se šalje samo referenca što zahtjeva da se šalje poruka svaki puta kada se zatraži interakcija sa objektom.

Dijeljenje objekata najbolje je koristiti kada:

- Se štedi na prostoru ili je broj objekata u bazi podataka prevelik
- Kada je komunikacijsko opterećenje nisko (centralizirani poslužitelj)
- Kada je dijeljeni objekt striktno nepromjenjiv.

2.2.4 Servisi (engl. services)

Postoje važne domenske operacije koje ne mogu naći prirodan „dom“ bilo u entitetima ili vrijednosnim objektima. Najčešće se takve operacije stave u neki objekt iz razloga jer se negdje moraju staviti. Takvo prisilno guranje operacija u objekte narušava konceptualnu čistoću objekta i njegovu odgovornost što za posljedicu ima teško shvaćanje što taj objekt zaista predstavlja. Umjesto prakticiranja takvih slučajeva možemo pratiti prirodne crte problema i uključiti servise (*engl.* services) eksplicitno u model.

Servis (*engl.* service) je operacija ponuđena preko sučelja koje postoji samostalno u modelu, bez enkapsulacije stanja, kao što to rade entiteti i vrijednosni objekti. Servisi su česti obrazac u programskim okvirima, ali se također mogu primijeniti u domenski sloj.

Sam servis predstavlja vezu sa ostalim objektima. Za razliku od entiteta i vrijednosnih objekata, servis je definiran isključivo u terminima onoga što može učiniti za svog klijenta. Imenovanje servisa bi trebalo ići u pravcu imenovanja aktivnosti koju takav servis obavlja – imenovanje više u stilu glagola nego imenice.

Svaki servis ima definiranu odgovornost. Ta odgovornost zajedno sa sučeljem koje ju ostvaruje bi trebala biti dio modela.

Dobar servis ima tri karakteristike:

- Operacija se odnosi na koncept iz domene koji nije prirodni dio entiteta ili vrijednosnog objekta.
- Sučelje je definirano u okvirima drugih elemenata u domeni.
- Operacije koje servis sadrži moraju biti bez stanja (*engl.* stateless).
    - Pod pojmom bez stanja misli se da bilo koji klijent može koristiti bilo koju instancu servisa bez obzira na povijest te instance.

Servisi ne spadaju samo u model domene već neki servisi spadaju u infrastrukturni sloj. Stoga je potrebno razlikovati servise koji pripadaju modelu domene i one koji pripadaju drugim slojevima. Primjerice, banka može imati aplikaciju koja šalje e-mail svojim klijentima kada stanje računa padne ispod neke granice. Sučelje koje enkapsulira e-mail sustav je servis u infrastrukturnom sloju jer nije dio koncepta domene već samo predstavlja tehničku realizaciju slanja e-maila. Još je teže razlikovati aplikacijske servise od servisa domene. Aplikacijski sloj je zadužen za donošenje odluke za naručivanje obavještavanja korisnika, dok servis domene određuje da li je pređena zadana granica.

Mnogi domenski i aplikacijski servisi su izgrađeni na populaciji entiteta i vrijednosnih objekata. Često puta su entiteti i vrijednosni objekti prefini za prikladan pristup mogućnostima domenskog sloja stoga servisi mogu predstavljati odlično sučelje za komunikaciju sa domenskim slojem.

2.2.5 Moduli (engl. modules)

Kod velikih i kompleksnih aplikacija, modeli postaju sve veći i veći. U trenutku kada model postane prevelik otežano je razumijevanje veza i interakcija u modelu. Iz tog razloga potrebno je organizirati model u module. Moduli se koriste kao metode za organizaciju povezanih koncepata i zadataka sa ciljem reduciranja složenosti.

Moduli su u širokoj primjeni u većini projekata. Jednostavnije je dobiti sliku cijelog modela ukoliko se pogledaju moduli koje sadrži te njihove međusobne veze. Nakon što se shvate interakcije između modula onda se mogu proučavati detalji unutar modula.

Korištenje modula u dizajnu je način povećavanja kohezije i smanjenja međuovisnosti. Moduli se moraju sastojati od elementa koji logički ili funkcionalno imaju zajedničkih točaka. Moduli bi trebali imati dobro definirana sučelja preko kojih mogu komunicirati sa drugim modulima. Umjesto pozivanja tri objekta nekog modula, bolje im je pristupati preko jednog sučelja jer smanjuje međuovisnosti. Mala međuovisnost smanjuje složenost i olakšava održavanje.

2.2.6 Agregati (engl. agregates)

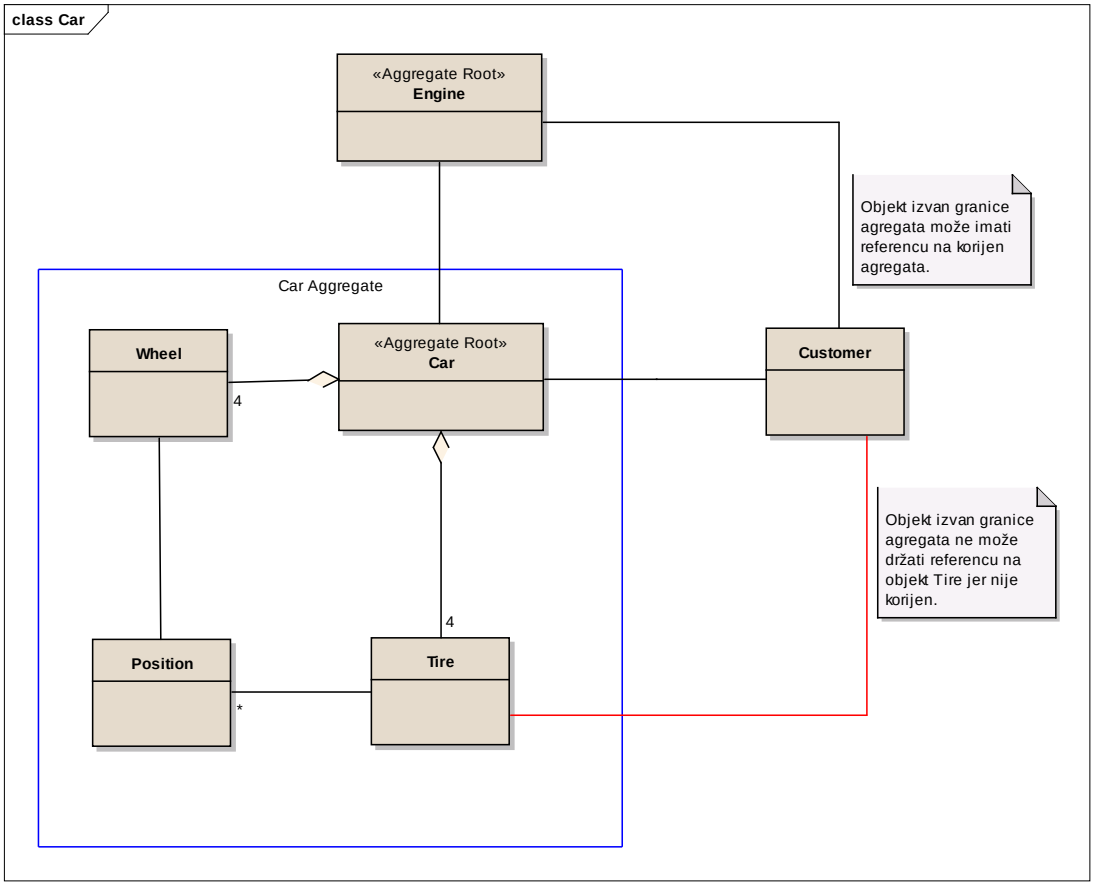
Mnoge poslovne domene imaju jako međupovezan skup objekata. Potrebno je ići po dugačkim i dubokim stazama slijedeći reference na objekte. Problem je nepostojanje oštrih granica među objektima u samoj domeni.

Čak i u izoliranoj transakciji, mreža odnosa koje objekt ima očito ograničava mogućnosti promjene. Dakle, teško je garantirati konzistentnost promjena nad objektima u modelu sa složenim asocijacijama. Osnovni problem je kako znati gdje objekt, koji je sačinjen od drugih objekata, počinje i završava, odnosno problem nedefinirane granice. Ono što je potrebno je apstrakcija za enkapsulaciju referenci unutar modela.

Agregat je skup povezanih objekata koje tretiramo kao jedinku prilikom promjene podataka. Svaki agregat određen je korijenom i granicom. Granica određuje što je unutar agregata. Korijen je jedan, specifičan entitet sadržan unutar agregata. Korijen je jedini član agregata na kojeg vanjski objekti smiju imati referencu dok objekti koji su unutar granice smiju imati međusobne reference. Svi ostali entiteti unutar agregata izuzev korijena moraju posjedovati lokalni entitet koji je bitan samo unutar agregata,

jer bilo koji vanjski objekt ne može vidjeti što se nalazi u agregatu. Dakle vanjski objekti komuniciraju samo sa korijenskim entitetom agregata.

Primjerice, model automobila se može koristiti u aplikaciji za automehaničarsku radionicu. Automobil je entitet sa globalnim identitetom jer želimo razlikovati baš taj automobil od drugih automobila na svijetu. Možemo uzeti identifikacijsku oznaku automobila kao jedinstveni identifikator koji je dodijeljen svakome novom automobilu. U aplikaciji bismo željeli pratiti koliki put je koja guma prošla, te koliko je koja guma istrošena. Da bismo znali koja je koja guma, gume moraju biti identificirani entiteti. Ali vrlo je vjerojatno da nas ne zanima identitet guma izvan konteksta određenog automobila. Ukoliko bi zamijenili staru gumu sa novom, aplikacije više neće pratiti staru gumu, odnosno neće znati više za njezin identitet. Isto tako nikoga iz vanjskog svijeta neće zanimati na kojemu automobilu se vrti koja guma. Stoga automobile predstavlja korijenski entitet agregata čija granica obuhvaća i gume. Primjerice motor automobila ima serijski broj ugraviran na sebe i ponekad se prati neovisno o automobilu pa bi u tome slučaju motor automobila bio korijen vlastitog agregata.



Slika 3 - Car agregat

Invarijante, odnosno pravila konzistentnosti se moraju održati kad god se promijene podaci. Invarijante koje vrijede unutar agregata su sigurno provedene nakon završetka transakcije.

Da bismo preveli konceptualni agregat u implementaciju, potrebno je postaviti skup pravila koji će se primjenjivati za sve transakcije:

- Korijenski entitet ima globalni identitet i odgovoran je za provjeravanje invarijanti.
- Entiteti unutar granice imaju samo lokalni identitet, jedinstven jedino unutar agregata.
    - Ništa izvan agregata ne može držati referencu na objekt unutar granice. Korijen može predati referencu na unutarnje entitete na

privremeno korištenje, odnosno vanjski objekt ne smije trajno zadržati referencu na objekt.

- Iz baze je moguće dobiti samo korijenske entitete.
- Objekti unutar agregata mogu držati referencu na korijene drugih agregata
- Prilikom brisanja agregata, operacija brisanja mora odjednom obrisati sve elemente agregata.

2.2.7 Tvornice (engl. factories)

Stvaranje objekta samo po sebi može biti složena operacija. Ali tko je zadužen za to stvaranje? Složene operacije stvaranja ne spadaju u odgovornost kreiranih objekata, odnosno objekti koji se stvaraju ne bi trebali znati stvoriti sami sebe jer to nije njihova uloga. Forsiranjem takvog principa stvaranja možemo imati za posljedicu zamršen dizajn koji je teško razumjeti.

Ukoliko bi odgovornost stvaranja objekta premjestili u nadležnost klijenta (klijentskog sloja) tada bi onečistili dizajn klijenta, otkrili unutarnju strukturu i pravila sloja domene koju želimo sakriti/enkapsulirati od ostatka svijeta te bi stvorili ovisnost stvaranja objekata iz sloja domene o klijentskom sloju. Dakle, da bi kreirali objekt njegova nam kompletna struktura mora biti poznata.

Stvaranje kompleksnih objekata je odgovornost sloja domene, no međutim taj zadatak ne pripada objektima koji izražavaju model. Doduše postoje slučajevi u kojima stvaranje objekata pripada domeni koju se modelira, kao npr. „otvori bankovni račun“. Ali stvaranje objekata obično nema nikakve veze sa domenom koja se modelira, već to spada u detalje implementacije. Da bismo riješili problem, moramo dodati konstrukte u dizajn domene koji nisu entiteti, vrijednosni objekti ili servisi, odnosno elementi koji ne predstavljaju ništa u domeni ali su sastavni dio odgovornosti sloja domene.

Svaki objektno orijentirani jezik omogućava mehanizam za stvaranje objekata, ali postoji potreba za apstraktnijim konstrukcijskim mehanizmima. Programski element čija je odgovornost stvaranje drugih objekata naziva se tvornica (*engl.* factory).

Kao što sučelje objekta enkapsulira njegovu implementaciju dozvoljavajući klijentu korištenje ponašanja objekta bez potrebe za znanjem kako objekt funkcionira, tvornica enkapsulira znanje potrebno za stvaranje kompleksnih objekata ili agregata. Tvornica pruža sučelje koje reflektira ciljeve klijenta te apstraktni pogled na stvoreni objekt.

Dakle, potrebno je premjestiti odgovornost stvaranja instanci kompleksnih objekta i agregata u posebni objekt koji možda nema direktne veze sa modelom domene ali svejedno spada u domenski sloj. Isto tako potrebno je kreirati sučelje koje enkapsulira složene operacije stvaranja i koje će omogućiti klijentu da ne referencira konkretan razred objekta koji se instancira. Agregate je potrebno stvoriti kao cjelinu, odnosno jedinku, te tako provoditi njihove invarijante.

Postoji nekoliko načina za dizajniranje tvornica, odnosno oblikovnih obrazaca poput:

- Factory method
- Apstract factory
- Builder

Dva osnovna zahtjeva za svaku dobru tvornicu su:

- Svaka operacija stvaranja je atomarna i provodi sve invarijante kreiranog objekta ili agregata. Za entitet to znači stvaranje cijelog agregata sa zadovoljavanjem svih invarijanti sa opcionalnim elementima koji se trebaju još nadodati. Za ne mijenjajuće vrijednosne objekte, to znači da su svi atributi inicijalizirani na ispravne vrijednosti. Ukoliko nije moguće ispravno stvoriti objekt tada je potrebno baciti iznimku.
- Tvornica treba biti apstrahirana prema željenom tipu, a ne prema konkretnoj klasi.

Tvornicu je moguće smjestiti na više mjesta u domeni. Ukoliko dodajemo element u postojeći agregat tada tvornicu smještamo u korijenski entitet (korištenje oblikovnog obrasca – factory method). Time sakrivamo strukturu agregata od objekata izvana. Ukoliko je neki objekt blisko uključen u kreiranje drugog objekta tvornica se stavlja u taj objekt (korištenje oblikovnog obrasca – factory method).

Kada želimo kreirati tvornicu, a ne postoji neko prirodno mjesto potrebno je stvoriti samostalni objekt tvornice ili servis. Samostalna tvornica obično stvara kompletan agregat, upravljajući referencom na korijen i osiguravajući da su prilikom stvaranja sprovedene sve invarijante.

Ponekad za stvaranje objekata je dovoljan samo konstruktor. Javni konstruktor može se koristiti u slijedećim slučajevima:

- Razred objekta nije dio nikakve hijerarhije te se ne koristi polimorfno.
- Klijentu je bitna implementacija, primjerice za odabir strategije.
- Svi atributi objekta su dostupni klijentu tako da uopće nema enkapsulacije.
- Konstrukcija objekta nije kompliciranа.
- Javni konstruktor mora biti atomarna operacija koja zadovoljava sve invarijante kreiranog objekta.

Tvornica je zadužena za provođenje svih invarijanti objekta ili agregata kojih stvara. To ne znači nužno da tvornica sama sadrži invarijante. Najčešći i najbolji slučaj je da tvornica delegira posao provedbe invarijanti objektu koji se kreira. Ponekad, posebice za vrijednosne objekte tvornica sadrži kod za provođenje invarijanti.

Tvornice entiteta se razlikuju od tvornica vrijednosnih objekata na dva načina. Vrijednosni objekti su nepromjenjivi, odnosno objekt koji se stvori predstavlja svoj finalni oblik. Pa tako operacije tvornice moraju omogućiti način definiranja potpunog objekta kojeg želimo stvoriti. Dok s druge strane tvornice entiteta dozvoljavaju zadavanje samo osnovnih atributa koji su potrebni za stvaranje ispravnog agregata. Ostali detalji se mogu dodati kasnije ukoliko nisu zahtijevani od invarijanti.

Osim za stvaranje potpuno novih objekata, tvornica se može koristiti za rekonstituciju objekta iz baze podataka u memoriju. Tvornica koja se koristi za rekonstituciju je vrlo slična tvornici koja se koristi za stvaranje, ali sa dvije razlike:

- Tvornica entiteta koja se koristi za rekonstituciju ne dodjeljuje novi identifikator. Stoga identifikacijski atributi moraju biti dio ulaznih parametara tvornice za rekonstrukciju spremljenog objekta.
- Tvornica za rekonstituciju objekta će drugačije obraditi kršenje invarijanti. Budući da objekt sigurno postoji, znači da je došlo do neke pogreške koju treba nekako razriješiti. U slučaju tvornice za stvaranje novog objekta ukoliko se ne zadovolje invarijante tvornica jednostavno prekida stvaranje.

2.2.8 Repozitoriji (engl. repositories)

Da bismo bilo što radili sa objektom, potrebno je imati referencu na njega. Pa kako ćemo doći do te reference? Svakako jedan od načina dobivanja reference je stvoriti novi objekt. Drugi način je obilazak po asocijacijama objekta – od objekta za kojeg već imamo referencu zatražimo referencu na objekt s kojim je povezan. Ali naravno moramo imati referencu na prvi objekt da bismo mogli dohvaćati druge objekte. Treći način je da izvršimo upit nad bazom da dobijemo podatke o objektu te ga pomoću tih podataka rekonstituiramo.

Pretraživanje baze podataka je globalno dostupno i omoguće dohvaćanje bilo kojeg objekta. Odluku o dohvaćanju objekata obilaskom asocijacija ili pretraživanjem baze podataka je dizajnerska odluka, svaki od načina ima svoje prednosti i nedostatke. Primjerice, da li će objekt „kupac“ imati referencu na kolekciju svih narudžbi ili će se narudžbe dohvatiti iz baze podataka na temelju identifikatora kupca?

Tehnički, dohvaćanje objekta je podskup operacija stvaranja. Ali dohvaćanje objekta događa se u sredini njegovog životnog ciklusa gdje objekt već postoji samo je spremljen u drugačijem obliku. Stoga stvaranje već postojećih objekata na temelju podataka iz baze podataka naziva se rekonstitucija.

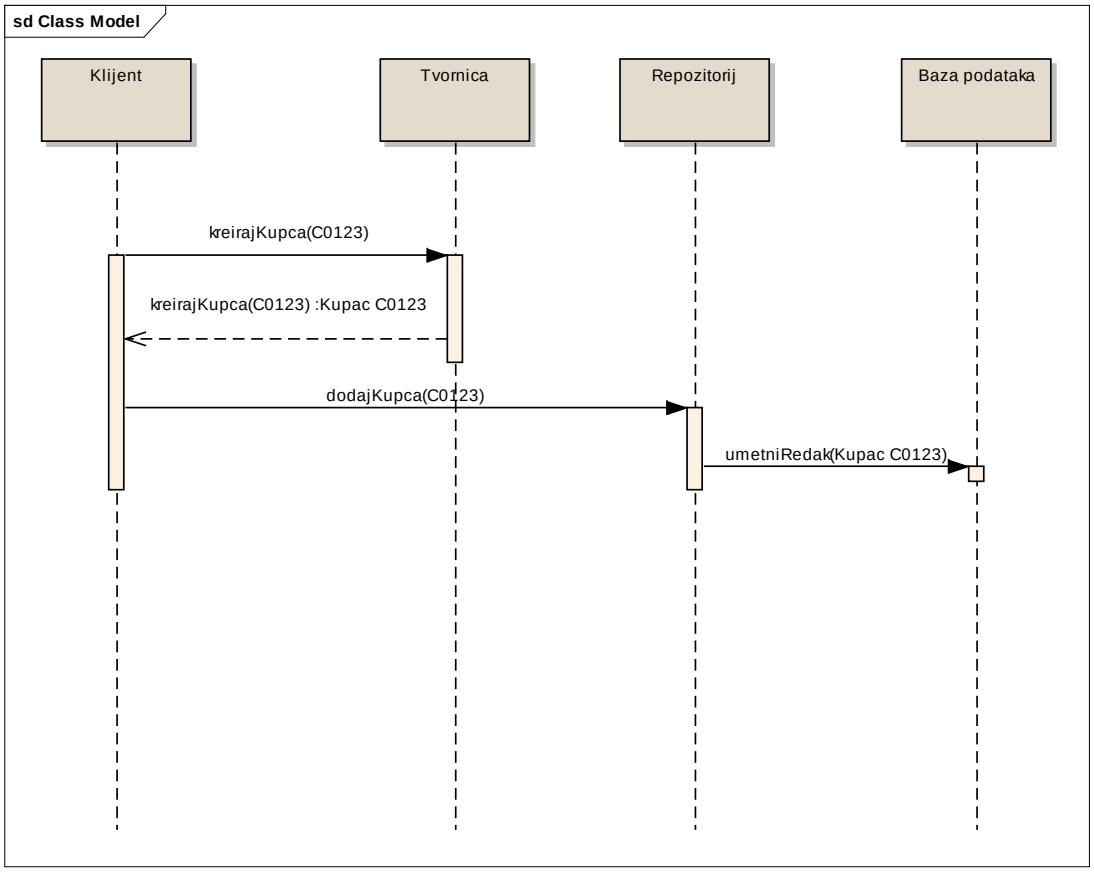
Cilj dizajna vođenim domenom je stvoriti bolji informacijski sustav gdje je fokus na modelu domene, a ne na tehnologiji. Najčešće programeri napišu neki upit, pošalju ga na bazu te dobe rezultat kao tablicu sa redcima. Programer iskorištava potrebne informacije te stvara objekt pomoću konstruktora ili tvornice. Međutim, takvim postupcima gubi se fokus na model. Počinjemo razmišljati o objektima kao o spremnicima podataka koje daju upiti, preskačemo agregate, zanemarujemo enkapsulaciju i direktno uzimamo i manipuliramo s podacima.

Klijentu su potrebni praktični načini dohvaćanja reference na već postojeće objekte domene. Drugim riječima, podskup perzistentnih objekata mora biti globalno dostupan kroz pretragu zasnovanoj na atributima objekta. Takav pristup je potreban za dohvaćanje korijena agregata do kojih nije moguće doći obilaskom po asocijacijama. To su najčešće entiteti, ponekad i vrijednosni objekti sa kompleksnom unutarnjom strukturom. Upiti na bazu mogu prekršiti enkapsulaciju na domenske objekte i agregate. Izlaganje tehničke infrastrukture i mehanizma pristupa bazi podataka komplicira klijentski dio aplikacije i zamagljuje modelom vođeno dizajniranje.

Repozitorij je mehanizam za enkapsulaciju ponašanja spremanja, dohvaćanja i pretraživanja koja emulira kolekciju objekata.

Za svaki tip objekta koji treba globalni pristup, potrebno je kreirati objekt koji će omogućiti iluziju memorijske kolekcije svih objekata tog tipa. Pristup takvim operacijama potrebno je omogućiti kroz dobro poznato globalno sučelje. U sučelje je potrebno dodati metode za dodavanje i brisanje objekata koje će enkapsulirati stvarno umetanje i brisanje podataka iz spremišta podataka. Isto tako potrebno je omogućiti metode koje će odabrati objekte na temelju nekog kriterija i kao rezultat vratiti potpuno instancirane objekte ili kolekcije objekata čiji atributi zadovoljavaju određeni kriterij te će tako enkapsulirati stvarno spremište podataka i tehnologiju upita. Repozitoriji se definiraju samo za korijenske agregate koji u stvarnosti trebaju direktni pristup.

Postoji veza između tvornice i repozitorija. Oboje su dio DDD-a i oboje nam pomažu u upravljanju životnog ciklusa domenskih objekata. Dok je tvornica posvećena stvaranju objekata, repozitorij se brine za pronalazak postojećih objekata. Tako primjerice kada želimo dodati novi objekt u repozitorij potrebno je prvo stvoriti objekt pomoću tvornice, a zatim ga predati u repozitorij koji će onda taj objekt smjestiti u bazu podataka.



Slika 4 - Odnos tvornice i repozitorija

2.3 Kontinuirano refaktoriranje

Za vrijeme trajanja procesa dizajniranja i razvoja moramo stati s vremena na vrijeme i pogledati kod. Možda je vrijeme za refaktoriranje. Refaktoriranje (*engl. refactoring*) je proces redizajniranja programskog koda radi poboljšanja modela bez bilo kakvog utjecaja na ponašanje aplikacije. Refaktoriranje se obično obavlja u

malim i kontroliranim koracima sa velikom pažnjom da ne uništimo postojeću funkcionalnost ili da ne unesemo greške. Automatizirani testovi su od velike pomoći da se osiguramo od pogrešaka.

Tradicionalno refaktoriranje koda je tehnički motivirano. Međutim refaktoriranje može biti motivirano uvidom u domenu radi odgovarajućeg poboljšanja modela ili njegove reprezentacije u kodu. Refaktoriranje motivirano uvidom u domenu proizlazi iz prelaska implicitnih koncepata u eksplicitne. Prilikom razgovora sa ekspertima domene neki koncepti domene mogu ostati ne spomenuti ili ne definirani, drugim riječima implicitni. Stoga je potrebno takve implicitne koncepte prepoznati, pretvoriti ih u eksplicitne i ugraditi u model domene.

Rezultat refaktoriranje je serija malih napredaka. Ponekad se zna desiti da ukoliko napravimo mnogo malih promjena dobivamo vrlo malo na vrijednosti dizajna, ali ponekad se zna desiti da nekoliko malih promjena ima veliki značaj na dizajn.

2.4 Omeđen kontekst (engl. bounded context)

Svaki model ima svoj kontekst. Kada se bavimo samo sa jednim modelom taj kontekst je implicitan, odnosno nije ga potrebno posebno definirati. Međutim, veliki projekti obuhvaćaju više modela. Kombinacija programskih kodova sa različitim modelima ima za posljedicu teško održavanje i ne razumijevanje modela. Komunikacija postaje ne shvatljiva, a sveprisutni jezik postaje ne održiv. Rješenje je formiranje omeđenog konteksta. Dakle potrebno je eksplicitno definirati kontekst unutar kojeg se primjenjuje model.

Za projekte koji se sastoje od više modela, odnosno omeđenih konteksta koristi se kontekstna mapa (*engl. context map*) koja predstavlja dokument koji definira sve omeđene kontekste i njihove međusobne odnose/veze.

2.5 Razvoj vođen testiranjem (Test Driven Development)

Jedan od glavnih ciljeva za vrijeme dizajniranja i refaktoriranja nad razredima domene, odnosno objekata domene je da ih je moguće testirati na razini jedinke. Objekte je lakše testirati što su manje ovisni o drugim objektima pa čak i slojevima. Razvoj vođen testiranjem (*engl. Test Driven Development*) ili TDD je pristup koji pomaže timu u ranoj identifikaciji dizajnerskih problema u projektu kao i verifikaciji da je programski kod u skladu sa modelom domene. DDD je idealan za „prvo testiraj“ razvoj jer su stanja i ponašanje sadržani u domenskim razredima te je njihovo testiranje u izolaciji jednostavno. Vrlo je važno testirati stanja i ponašanje modela domene, a što manje se fokusirati na implementacijske detalje pristupa podacima ili perzistencije.

Razvoj vođen testiranjem (*engl. Test driven development*) je metodologija oblikovanja programske potpore koja koristi unit testove[1] kao dio procesa pisanja programskih kodova. Kada se prakticira razvoj vođen testiranjem, prvo se pišu testovi, a onda programski kod. Prilikom TDD-a potrebno se pridržavati slijedećih koraka:

- Napisati unit test koji pada (Red) – prvo je potrebno napisati unit test. Unit test bi trebao iskazati očekivanje ponašanja programa. Kada se prvi put piše unit test, unit test bi trebao pasti. Test mora pasti zato jer još nije napisan nikakav programski kod koji bi zadovoljio test.

- Napisati programski kod koji će proći unit test (Green) – Potrebno je napisati dovoljno programskog koda koji će zadovoljiti unit test. Cilj je napisati kod na najbrži mogući način, bez razmišljanja o arhitekturi aplikacije. Cilj je napisati minimalnu količinu koda koja će biti dovoljna da se zadovolji test.

[1] Unit testovi - testovi malih, odnosno jediničnih dijelova izvornog koda

- Napraviti preoblikovanje svojeg koda (Refactor) – Nakon što je programski kod zadovoljio test, potrebno je napraviti korak natrag i razmisliti o arhitekturi aplikacije. Dakle potrebno je napraviti preoblikovanje programskog koda koristeći oblikovne obrasce u programiranju.

Jedan od važnih uvjeta prilikom pisanja unit testova je da se testovi moraju brzo izvoditi. Dakle normalno je da imamo na stotine pa i tisuće unit testova te da te testove svako malo pokrećemo. Iz tih razloga važno je da se testovi brzo izvode inače ne bi imali smisla. Najveći problem su primjerice baze podataka jer je bazu podataka potrebno pokrenuti, poslati upit na bazu itd., a to sve vremenski košta. Iz tog razloga uvode se mock objekti.

2.5.1 Mock objekti

Prilikom izrade unit testova često se koriste mock, odnosno lažni objekti. Mock objekti simuliraju ponašanje kompleksnih stvarnih objekata. Mock objekti su korisni kada stvarni objekt nije praktičan ili ga nije moguće ukomponirati u unit test. Ukoliko stvarni objekt ima bilo koju od sljedećih karakteristika, potrebno bi bilo koristi mock objekt prilikom testiranja:

- Objekt daje nedeterminističke rezultate (npr. trenutno vrijeme ili trenutna temperatura)
- Ima stanje koje je teško kreirati ili reproducirati (npr. mrežna greška)
- Spor je (npr. baza podataka koju je potrebno istestirati prije testa)
- Trenutno ne postoji ili može promijeniti ponašanje
- Mora sadržavati informacije i metode samo radi testiranja, a koje inače ne bi sadržavao.

Postoji nekoliko varijacija lažnih objekata:

- Fakes – fake objekti su najednostavniji objekti. Oni implementiraju isto sučelje kao i objekti koje oni predstavljaju te vraćaju predefinirane odgovore.
- Mocks – mock objekti rade isto što i fake objekti s tom razlikom da će još i provjeravati kontekst svakog poziva – primjerice provjeravat će u kojem

redoslijedu su metode pozivane, da li su metode pozvane u svome životnom ciklusu, s kojim parametrima su metode pozvane itd.

2.6 Dependency injection i Inversion of Control

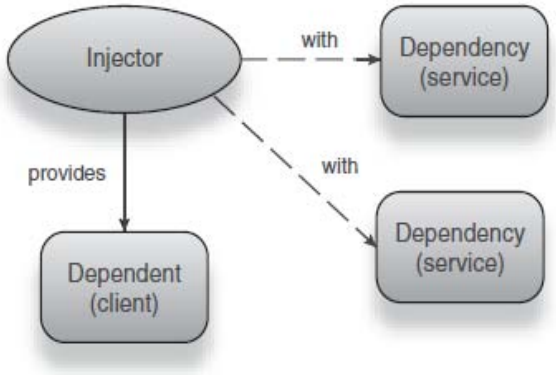
U prethodnim podpoglavljima govorili smo o građevnim blokovima DDD-a među kojima su entiteti, vrijednosni objekti, agregati, servisi, tvornice i repozitoriji. Entiteti i vrijednosni objekti grupiraju se u agregate, a tvornice su objekti koji su zaduženi za stvaranje agregata pomoću entiteta i vrijednosnih objekata te ostalih kompleksnih objekta. A što je sa servisima? Što ako neki entitet ili vrijednosni objekt za svoje ponašanje ovisi o servisima, odnosno o uslugama drugog objekta pri čemu taj objekt nije gradbeni element već samo nešto što pruža neku uslugu?

Jedan od trivijalnih rješenja je ukodirati servis u sami objekt. No takav pristup nije dobar jer objekt ovisi o implementaciji servisa, a ne samo o uslugama, odnosno apstrakcijama koje servis pruža. Isto tako otežano je testiranje takvog objekta, odnosno nije moguće koristiti mock objekte, te stvaranje servisa istog tipa sa različitim ponašanjem.

Umjesto hard kodiranja ovisnosti, objekt bi trebao objaviti koje su njegove ovisnosti. To objavljivanje ovisnosti se obavlja putem definiranja parametara konstruktora (constructor injection), preko settera (setter injection) ili javnih svojstava. Ovisnosti moraju biti definirane preko apstrakcija i to su u većini slučajeva sučelja. Takvim pristupom objekt koji je ovisan o drugim objektima ne ovisi o njihovoj implementaciji. Takav pristup zove se „ručna konstrukcija“ (*engl. contruction by hand*) jer se klijent treba brinuti da prilikom stvaranja objekta osigura njegove zavisnosti.

Dependency injection oslobađa klijenta znanja o svojim ovisnostima i o načinima kako ih stvoriti. DI koristi tzv. „The Hollywood Principle – Don't call us; we'll call you!“ gdje se programske knjižnice DI-a brinu o snabdijevanju ovisnog objekta njegovim ovisnostima, odnosno zadužene su za stvaranje i distribuciju ovisnosti u objektni graf ovisnog objekta.

Korištenjem dependency injection principa, životnim ciklusom servisa, odnosno ovisnosti, upravlja dependency provider/injector/container. Dependency injection container je zadužen za stvaranje odnosno instancijranje ovisnosti, upravlja životnim ciklusom tih ovisnosti, te je zadužen za odlučivanje koje ovisnosti se distribuiraju u koji ovisan objekt.



Slika 5 - injector osigurava objektu njegove ovisnosti

Dependency injection je poseban oblik inverzije kontrole (*engl. inversion of control*) gdje se pristup dohvaćanju ovisnosti invertira, odnosno invertira se kontrola stvaranja objekta. Dakle više ne kontrolira ovisan objekt stvaranje svojih ovisnosti, već se kontrola prebacuje na procese koji djeluju izvan ovisnog objekta.

DI nije sastavni dio DDD-a, ali predstavlja vrlo korisnu metodologiju za neovisan kod, odnosno kod sa što manje međuovisnosti čime naglasak na modelu dolazi više do izražaja. DI pruža čišći i neovisniji dizajn ubrizgavanjem u druge objekte, kao što je ubrizgavanje repozitorija i servisa u objekte domene.

3. Domena upravljanja stambenim zgradama

U ovome poglavlju biti će navedeni i objašnjeni svi procesi koji se javljaju u upravljanju i održavanju stambenih zgrada kroz prikupljene dokumente kao što su radne procedure, pravilnici i zakoni te informacije prikupljene iz prakse. Premda je opis procesa većinom pravno definiran, on je nužan za shvaćanje pojmova i poslovnih procesa koji se javljaju u upravljanju i održavanju stambenih objekata. Procesi upravljanja stambenim zgradama su pojednostavljeni kako bi mogli u okviru ovoga rada prikazati primjenu DDD-a.

U sljedećim podpoglavljima biti će specificirani zahtjevi koji se nameću na informacijski sustav temeljeni na analiziranim procesima prikupljenim iz raznih dokumenata i prakse.

3.1 Opis procesa upravljanja i održavanja stambenih objekata

Upravljanje nekretninama je umijeće gospodarenja nekretninama s ciljem očuvanja i povećanja njezine vrijednosti. Pod nekretninom se podrazumijeva zgrada i zemljište koje pripada zgradi. U nastavku teksta pod pojmom zgrada podrazumijeva se i zemljište koje joj pripada.

Neposredni sudionici upravljanja su suvlasnici i upravitelj. Vlasnici se brinu i odgovaraju za svoje vlasništvo, a izvršne poslove oko toga povjeravaju upravitelju kao zalogoprimcu. Zakonom o vlasništvu i drugim stvarnim pravima određeno je da svaka zgrada mora imati upravitelja koji je registriran za tu djelatnost. Upravitelj upravlja zgradom, održava ju, prikuplja pričuvu za zgradu te obavlja i sve druge poslove koje mu povjere suvlasnici.

Izvor prihoda kojim se osigurava i ostvaruje briga suvlasničke zajednice za njenu nekretninu jest pričuva.

3.1.1 Temeljni dokumenti o upravljanju zgradama

Pravna podloga upravljanja je Zakon o vlasništvu i drugim stvarnim pravima u kojem se propisuju okviri za uređenja odnosa među suvlasnicima.

Ovim je zakonom, kao i podzakonskim propisima također propisano da se mora plaćati minimalno 1,53 kn/m^2 mjesečno u zajedničku pričuvu zgrade. Iz zajedničke pričuve zgrade plaćaju se zakonski obavezni troškovi (poput osiguranja zajedničkih dijelova zgrade ili mjesečnog servisa dizala), kao i ostali nužni i poželjni radovi.

Iz Zakona o vlasništvu i drugim stvarnim pravima proizlaze 2 temeljna ugovora:

1. **Međuvlasnički ugovor (MU)** kojemu pripadaju prilozi „Popis suvlasnika s utvrđenim suvlasničkim udjelima i udjelima u troškovima održavanja“ i „Popis zajedničkih dijelova i uređaja zgrade“. Međuvlasničkim ugovorom uređuju se odnosi među suvlasnicima u svezi s upravljanjem i korištenjem zgrade, a posebice:
    - Veličina suvlasničkih dijelova zgrade.
    - Uvjeti i način upravljanja zgradom.
    - Pobliži podaci o osobi koja će upravljati zgradom (u daljnjem tekstu: zajednički upravitelj / upravitelj).
    - Osnivanje, uvjeti i način prikupljanja i raspolaganja sredstvima zajedničke pričuve.
    - Ime i adresa stanovanja suvlasnika ovlaštenog za predstavljanje i zastupanje suvlasnika prema upravitelju, odnosno trećim osobama (u danjem tekstu predstavnik suvlasnika) i opseg njegovih ovlasti.
    - Druga pitanja vezana uz upravljanje i korištenje zgrade

2. **Ugovor o upravljanju zgradom (UUZ)** – kojem pripadaju prilozi „Program održavanja zgrade“ za godinu u kojoj se potpisuje ugovor i „Zapisnik o primopredaji zajedničkih dijelova i uređaja zgrade“. Tim se ugovorom uređuju međusobni odnosi suvlasnika i upravitelja zgrade, a osobito:

- Sadržaj i opseg poslova redovnoga održavanja zajedničkih dijelova i uređaja zgrade prema godišnjem odnosno višegodišnjem programu održavanja.
- Poduzimanje hitnih i nužnih popravaka.
- Obveze i rokovi izrade prijedloga godišnjih odnosno višegodišnjih programa održavanja, plana prihoda i rashoda te godišnjeg izvješća o radu.
- Način osnivanja zajedničke pričuve, kao i osiguranja sredstva zajedničke pričuve za pokriće troškova upravljanja i održavanja.
- Odgovornost za obavljanje poslova

3.1.2 Vlasništvo zgrade

Vlasnici zgrade su ili pojedinačni vlasnici ili suvlasnici nekretnine. Nekretnina je zemljište i ono što je s njime relativno trajno povezano, a nalazi se u razini, ispod razine ili iznad razine zemlje. Potpuno uređen suvlasnički odnos u nekoj nekretnini postoji kada se točno utvrdi tko je vlasnik kojega dijela zgrade. Točnije, kada se utvrdi tko je vlasnik kojega posebnog dijela zgrade, a što je zajedničko vlasništvo. Kako bi se to postiglo potrebno je etažirati zgradu.

Pojam etažni vlasnik sinonim je vlasnika na posebnom dijelu nekretnine. Posebni dio nekretnine je primjerice: stan, poslovni prostor ili garaža. Zajednički dio nekretnine je primjerice: krov, stubište, dizalo.

3.1.2.1 Prava i obveze vlasnika – suvlasnika

Suvlasnici odlučuju o svim pitanjima koja se tiču zajedničkih dijelova i uređaja zgrade sukladno odredbama Zakona o vlasništvu i drugim stvarnim pravima, drugih propisa te međuvlasničkog ugovora.

Suvlasnici upravljaju zgradom donošenjem odluka o poduzimanju redovnih i izvanrednih poslova upravljanja u skladu s Međuvlasničkim ugovorom, Zakonom o vlasništvu i drugim stvarnim pravima te drugim propisima.

Svi poslovi koje suvlasnici poduzimaju na zgradi imaju karakter redovne i izvanredne uprave. O redovnoj upravi suvlasnici odlučuju većinom glasova, dok je za izvanrednu upravu potrebna suglasnost svih suvlasnika.

Poslovima redovne uprave smatraju se poslovi redovitog održavanja zajedničkih dijelova i uređaja nekretnine, te građevinski zahvati radi održavanja, zatim stvaranje zajedničke pričuve i uzimanje zajmova za pokriće troškova održavanja zgrade koji se ne mogu podmiriti iz sredstva redovite pričuve.

Izvanrednom upravom koja premašuje okvir redovne uprave smatraju se promjena namjene, dogradnja, nadogradnja, poslovi preuređenja, davanje u zakup ili najam na dulje vrijeme, osnivanje hipoteke na zgradi odnosno davanje u zalog, davanje prava građenja, otuđenje.

Odluke o poduzimanju redovnih poslova upravljanja smatraju se donesenim kada se za njih izjasne suvlasnici koji zajedno imaju većinu suvlasničkih dijelova. Za donošenje odluka o poduzimanju izvanrednih poslova upravljanja potrebna je suglasnost svih suvlasnika.

3.1.3 Upravitelj

Zakon propisuje da svaka zgrada mora imati sklopljen ugovor o upravljanju s nekim od ovlaštenih upravitelja.

Suvlasnici ovlašćuju upravitelja da u njihovo ime i za njihov račun obavlja postojeće poslove kao što su:

- Organizacija redovnog održavanja zajedničkih dijelova i uređaja u graditeljskom i funkcionalnom stanju.
- Utvrđivanje visine sredstava zajedničke pričuve koju snosi pojedini suvlasnik.
- Organizacija naplate sredstava zajedničke pričuve, uključivši i prinudnu naplatu.
- Raspolaganje sredstvima koje suvlasnici izdvajaju za pokriće troškova održavanja zajedničkih dijelova i uređaja (sredstva zajedničke pričuve).

- Upravlja i raspolaže sredstvima pričuve na računu radi zaštite njihove vrijednosti

3.1.4 Predstavnik suvlasnika

Svaka zgrada ima svog predstavnika kojega su ostali suvlasnici izabrali i ovlastili za predstavljanje. Predstavnik zgrade je osoba koja vodi brigu o zgradi (dojavljuje kvarove, hitne intervencije, male popravke, promjene vlasnika itd.) te je u stalnoj vezi s upraviteljem. Za svoj radi ima pravo na naknadu. O visini naknade odlučuju suvlasnici.

Predstavnik suvlasnika ovlašten je:

- S upraviteljem sklopiti ugovor o upravljanju zgradom.
- Pokrenuti uspostavu vlasništva posebnog dijela nekretnine odnosno pretvaranje etažnog vlasništva stečenog po prijašnjim propisima u vlasništvo posebnog dijela nekretnine prema novom sustavu sadržanom u Zakonu, odnosno naručiti u tu svrhu izradu stručnog elaborata o etažiranju.
- Zastupati suvlasnike prema upravitelju i trećim osobama u poslovima vezanim za zajedničke dijelove i uređaje zgrade, a koji nisu povjereni upravitelju.
- Organizirati naplatu zajedničkih troškova.

Predstavnik suvlasnika dužan je:

- Ovjeravati radne naloge kao potvrde izvršenja određenog posla
- Voditi brigu o provođenju kućnog reda, načinu korištenja zajedničkih prostorija i zemljišta koje služi zgradi
- Redovito izvješćivati suvlasnike o svim važnijim pitanjima vezanim u upravljanje, kao i najmanje jednom godišnje podnijeti pismeno izvješće o svom radu svim suvlasnicima.

Predstavnik suvlasnika ne može donositi odluke umjesto suvlasnika. Za svoj rad predstavnik suvlasnika odgovara suvlasnicima.

Predstavnik suvlasnika je spona između zgrade i suvlasnika te upravitelja. Predstavnik suvlasnika je bitan za rad upravitelja kada se radi o provođenju i izvođenju svih potrebnih radova i obveza na zgradi. Upravitelj za svoj rad odgovara suvlasnicima putem predstavnika. Dobar odnos između predstavnika suvlasnika i upravitelja preduvjet je dobre suradnje.

3.1.5 Pričuva

Pričuva stambene zgrade je novčani fond suvlasnika zgrade iz kojeg se plaća zakonsko obvezno, nužno i drugo održavanje zgrade te poboljšice na zgradama. Plaćanje pričuve je zakonska obveza suvlasnika, odnosno obavezno je za svakog suvlasnika.

Svaki vlasnik posebnoga dijela plaća pričuvu mjesečno. Odluku o visini pričuve po jednom četvornom metru donose suvlasnici, a ona ne može biti manja od 1,53 kn/m^2, što je zakonski minimum. Visina pričuve se određuje za svaki stan posebno prema ukupnoj površini stana. Naplatu pričuve organizira i pričuvu prikuplja upravitelj na žiro-račun zgrade, te ju raspoređuje prema potrebi sukladno Zakonu te odluci suvlasnika.

Sredstva zajedničke pričuve koriste se:

- za redovno održavanje i poboljšanje zajedničkih dijelova i uređaja zgrade
- za hitne popravke
- za nužne popravke
- za osiguranje zgrade kod osiguravajućeg društva
- za zamjenu postojećih i ugradnju novih zajedničkih dijelova i uređaja zgrade
- za naknadu upravitelju zgrade.

Važno je napomenuti da pričuva nije upraviteljeva, ne nalazi se na računu upravitelja i ne može se koristiti bez potpisa predstavnika suvlasnika.

Hitnim se popravcima smatraju popravci koji se javljaju nenadano, a njim se sprečavaju posljedice za život i zdravlje ljudi kao i veća oštećenja same nekretnine.

3.2 Specifikacija zahtjeva

- Informacijski sustav treba omogućiti izvođenje ciklusa održavanja stambenih objekata, odnosno sve procese nužne u upravljanju stambenim objektima poput upravljanja financijama i popravcima kvarova.
- Sustav treba omogućiti evidenciju matičnih podatka o zgradama (Katastarska općina i čestice), suvlasnicima, zajedničkim i osobnim prostorima (stanovi, poslovni prostori, zajedničke prostorije) itd.
- Sustav mora omogućiti definiranje veza između suvlasnika, predstavnika suvlasnika, upravitelja i njegovih kooperanata, odnosno izvođača radova. Međusobne veze omogućuju podnošenje, delegiranje, odobravanje i provođenje zahtjeva koji se nameću u poslovima upravljanja stambenim zgradama.
- Informacijski sustav treba omogućiti evidenciju, analizu i računsku obradu financija zgrade.
- Informacijski sustav treba omogućiti suvlasnicima jednostavno podnošenje prijave kvarova na zgradi, te predstavniku suvlasnika, upravitelju i njegovim kooperantima obradu kvarova. Obrada kvara sastoji se od upraviteljevog delegiranja zadataka obrade kvarova svojim kooperantima, odnosno izvođačima radova, te predstavnika suvlasnika koji će odobriti izvođenje obrade kvarova ukoliko kvarovi nisu hitne prirode. Ukoliko je kvar hitne prirode upravitelj automatski može započeti obradu kvarova.
- Sustav treba omogućiti svim suvlasnicima, predstavniku suvlasnika, upravitelju i njegovim kooperantima jednostavan prikaz stanja prijave kvara, odnosno obrade prijavljenog kvara.
- Informacijski sustav treba omogućiti upravitelju kreiranje naloga za naplatu troškova zgrade, te delegiranje naloga predstavniku suvlasnika radi njihovog odobravanja kako bi se nalozi mogli naplatiti iz pričuve.
- Informacijski sustav treba omogućiti registraciju izvođača radova u sustav te svakom upravitelju omogućiti da postavi izvođače radova kao svoje ovlaštene partnere.

- Kooperante/Izvođače radova treba kategorizirati prema djelatnosti koju obavljaju.
- Minimalna visina pričuve je zakonom određena i iznosi 1.53 kn/m^2. Visina pričuve određuje se za svaki stan posebno prema ukupnoj površini stana. Dakle, na prijedlog upravitelja suvlasnici mogu povećati visinu cijene u kunama po kvadratu (> 1.53 kn/m^2).
- Za poslove koje je potrebno obaviti za zgradu, a koji imaju karakter redovne uprave[2] suvlasnici odlučuju većinom glasova.
    - Odluke o poduzimanju redovnih poslova upravljanja smatraju de donesenim kada se za njih izjasne suvlasnici koji zajedno imaju većinu suvlasničkih dijelova. Dakle potrebno je da se pozitivno izjasne stanari koji zajedno predstavljaju 51% ukupne površine zgrade.
- Za poslove koje je potrebno obaviti za zgradu, a koji imaju karakter izvanredne uprave[3] potrebna je suglasnost svih suvlasnika.
- Predstavnik suvlasnika mora ovjeravati radne naloge kao potvrdu za izvršavanje određenog posla. Tek nakon toga posao se smatra obavljenim.
- Izvođači radova, odnosno kooperanti, moraju imati ugovor sa upraviteljem. Dakle izvođenje radova na zgradi mogu obavljati samo ovlašteni izvođači upravitelja.
- Pričuvu plaća svaki vlasnik posebnog dijela, odnosno suvlasnik.
- Za sva plaćanja troškova upravitelj mora zatražiti od predstavnika suvlasnika potvrdu/ovjeru kako bi troškove mogao naplatiti iz pričuve.
- Isti upravitelj može biti za više zgrada, odnosno ulaza.

[2] Poslovima redovne uprave smatraju se poslovi redovitog održavanja zajedničkih dijelova i uređaja nekretnine, te građevinski zahvati radi održavanja, zatim stvaranje zajedničke pričuve itd.

[3] Poslovima izvanredne uprave smatraju se poslovi koji premašuju okvir redovne uprave kao što su promjena namjene, dogradnja, nadogradnja, poslovi preuređenja, davanje u zakup ili najam na dulje vrijeme, davanje prava građenja itd.

- Isti predstavnik suvlasnika može biti za više zgrada ili ulaza. Predstavnik suvlasnika ne mora nužno biti i stanar zgrade.

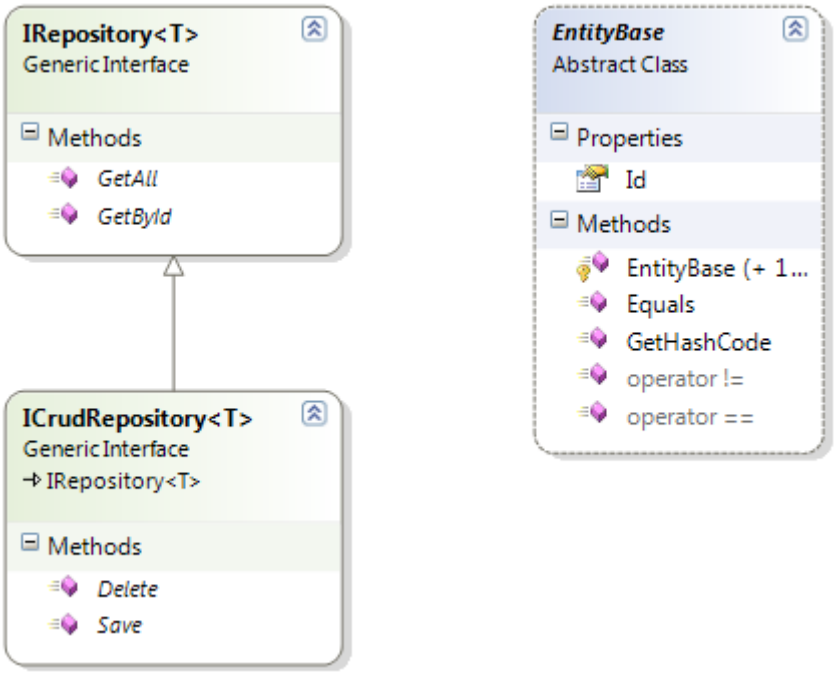
4. Izrada objektnog modela za Upravljanje stambenim zgradama primjenom DDD

U ovim podpoglavljima biti će opisana primjena DDD-a na pojednostavljenom primjeru upravljanja zgradama. Na početku opisati ćemo osnovne apstrakcije koje će nam pomoći pri modeliranju konkretnih elemenata. U nastavku ćemo opisati ostale elemente modela na način da su elementi grupirani u funkcionalne module. Prilikom modeliranja koristiti ćemo sveprisutni jezik, odnosno pokušati ćemo na temelju poglavlja 3. Strukturirana analiza i dizajn imenovati odgovarajuće objekte, te na isti način imenovati metode.

Napomenuti ćemo da je uz ovaj rad priložen programski kod s kojim je ostvaren model za upravljanje stambenim zgradama, te da su svih zahtjevi testirani unit testovima koji se također nalaze u programskom kodu.

4.1 Apstrakcije

Prije nego započnemo sa modeliranjem konkretne domene definirati ćemo apstrakcije vezane za građevne elemente DDD-a. Te apstrakcije će nam ubrzati i olakšati razvoj. Apstrakcije se sastoje od apstraktnog razred *EntityBase* koji predstavlja entitet, te repozitorija *IRepository* i *ICrudRepository*.



Slika 6 - Apstrackije DDD-a

Apstraktni razred *EntityBase* predstavlja entitet. Definirali smo identifikator koji definira entitet, te je taj identifikator moguće dodijeliti objektu isključivo za vrijeme njegovog stvaranja, odnosno preko konstruktora. Prethodno smo rekli da su dva entiteta jednaka isključivo onda kada imaju isti identifikator. Kako bi zadovoljili ovo pravilo implementirane su metode *Equals* i *GetHashCode*, te operatori *!=* i ==. Dakle svaki entitet koji ćemo kasnije kreirati mora naslijediti apstraktni razred *EntityBase*.

Što se tiče repozitorija, definirali smo dva općenita repozitorija, odnosno sučelja repozitorija:

- *IRepository* – repozitorij isključivo za čitanje. Čitanje je ostvareno preko operacija dohvaćanja entiteta preko identifikatora ili dohvaćanje svih entiteta.
- *ICrudRepository* – repozitorij koji se nasljeđuje iz *IRepository* koji osim operacija čitanja nudi i operacije koje mijenjaju stanje, odnosno operacije brisanja i spremanja.

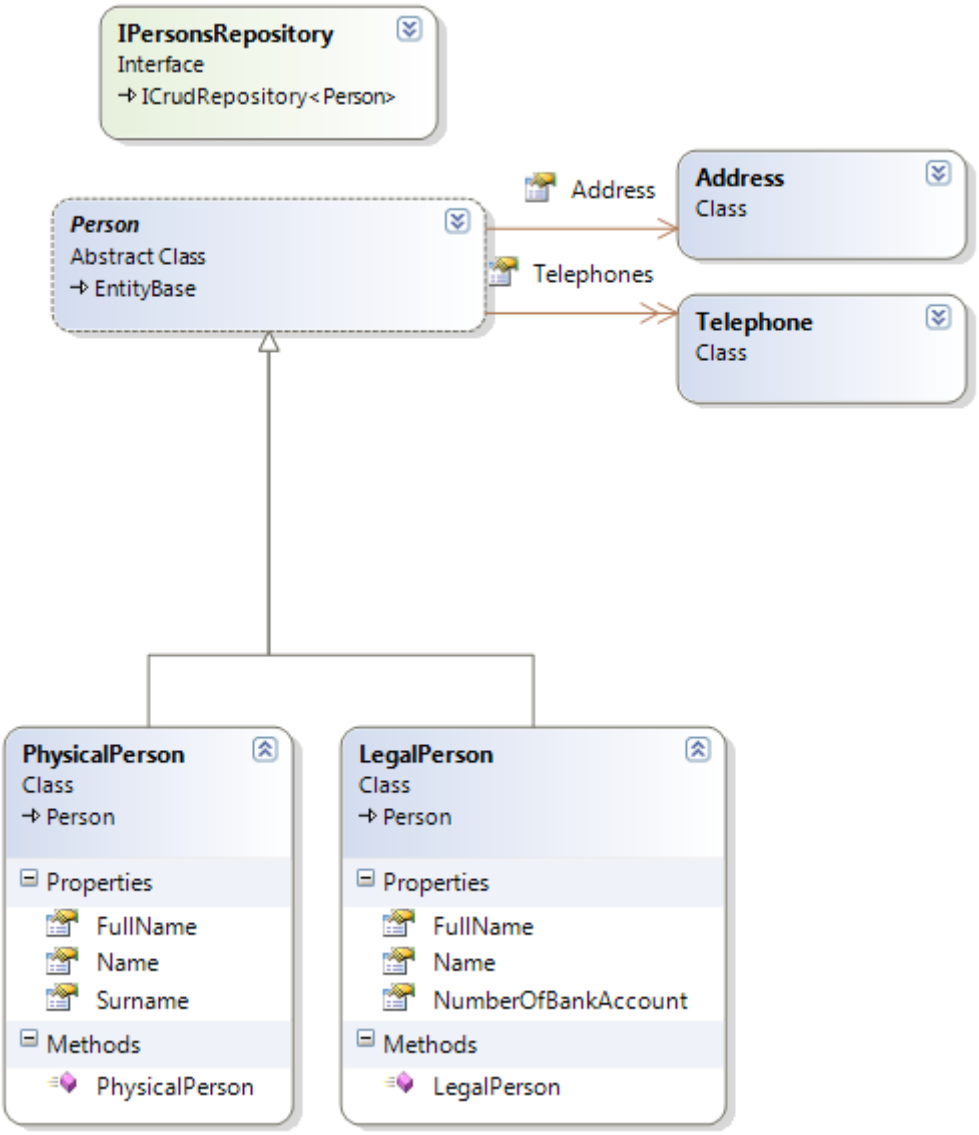
Prije smo spomenuli kako repozitoriji postoje jedino za korijenske agregate, pa kako bi osigurali primjenu takvog pravila definirali smo u definiciji repozitorija da repozitorij može isključivo postojati u službi entiteta. To smo ostvarili pomoću C# programskog jezika:

```
public interface IRepository<T> where T : EntityBase
```

4.2 Osobe i uloge

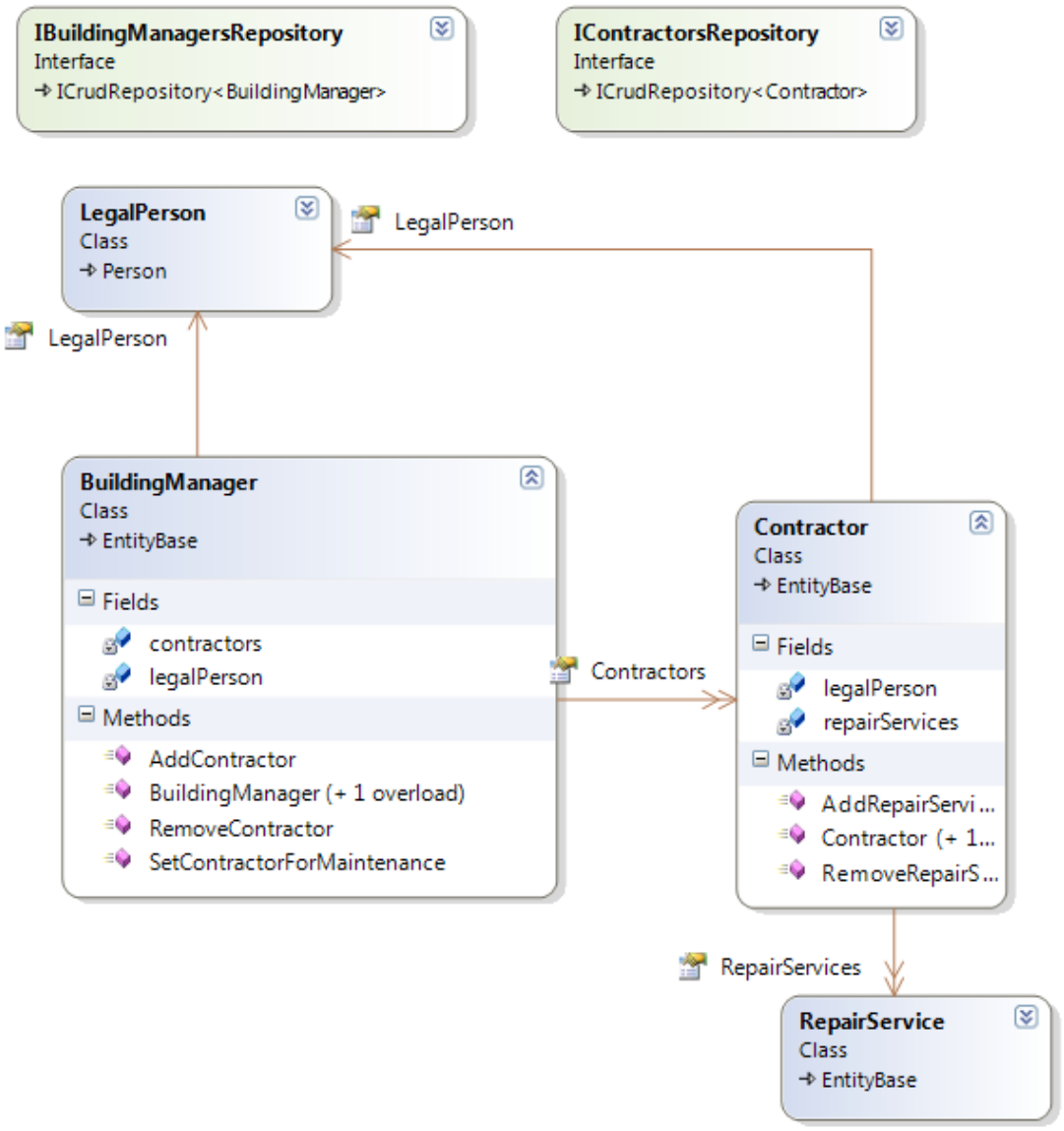
Osobe i uloge predstavlja zaseban modul u DDD-u koji obuhvaća osobe te njihove pripadajuće uloge.

Definirali smo apstraktni razred *Person* koji se nasljeđuje iz razreda *EntityBase* čime je definiran kao entitet. Razred *Person* je definiran pomoću vrijednosnih objekata *Address*, koji predstavlja adresu, te *Telephone* koji predstavlja telefonske brojeve neke osobe. Razred *Person* je konkretiziran preko fizičke osobe (*PhysicalPerson*) i pravne osobe (*LegalPerson*) jer naša domena uključuje kako stanare, odnosno vlasnike, tako i izvođače radova i upravitelja.



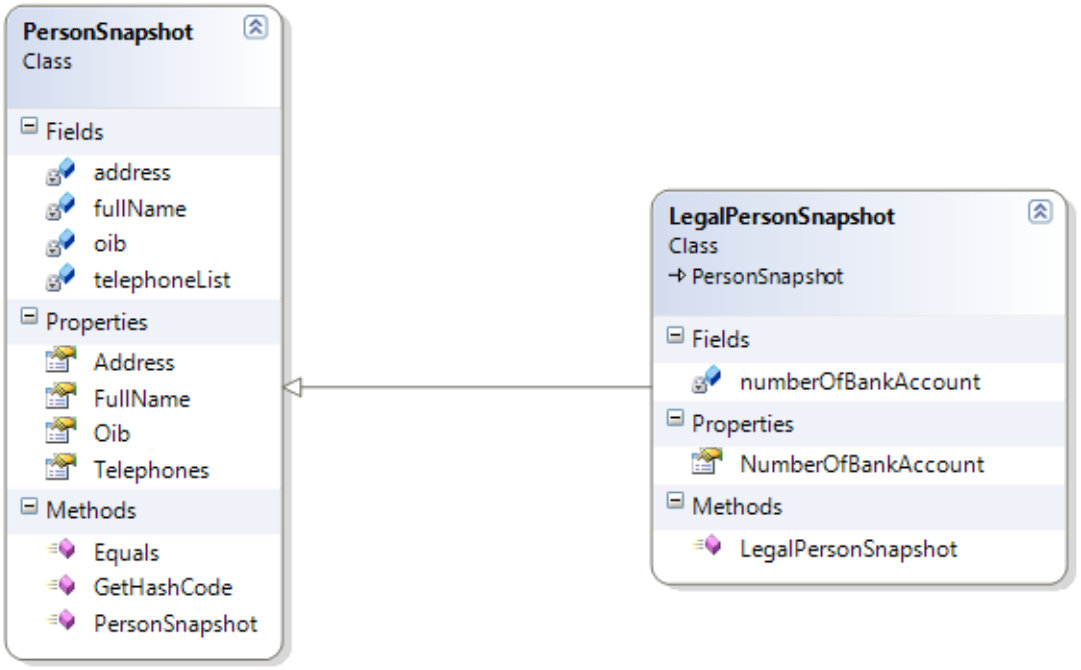
Slika 7 - Entiteti osobe i pripadajući vrijednosni objekti

Upravitelj zgrade je definiran preko razreda *BuildingManager* što je korijenski agregat. Agregat upravitelja zgrade sastoji se od pravne osobe te liste izvođača radova (*Contractor*), gdje je izvođač radova korijenski agregat za sebe.



Slika 8 - Entiteti Upravitelj i Izvođač radova

Radi očuvanja povijesti osoba definirali smo vrijednosne objekte *PersonSnaphot* i *LegalPersonSnapshot* koji predstavljaju vrijednosti neke osobe u određenom trenutku.

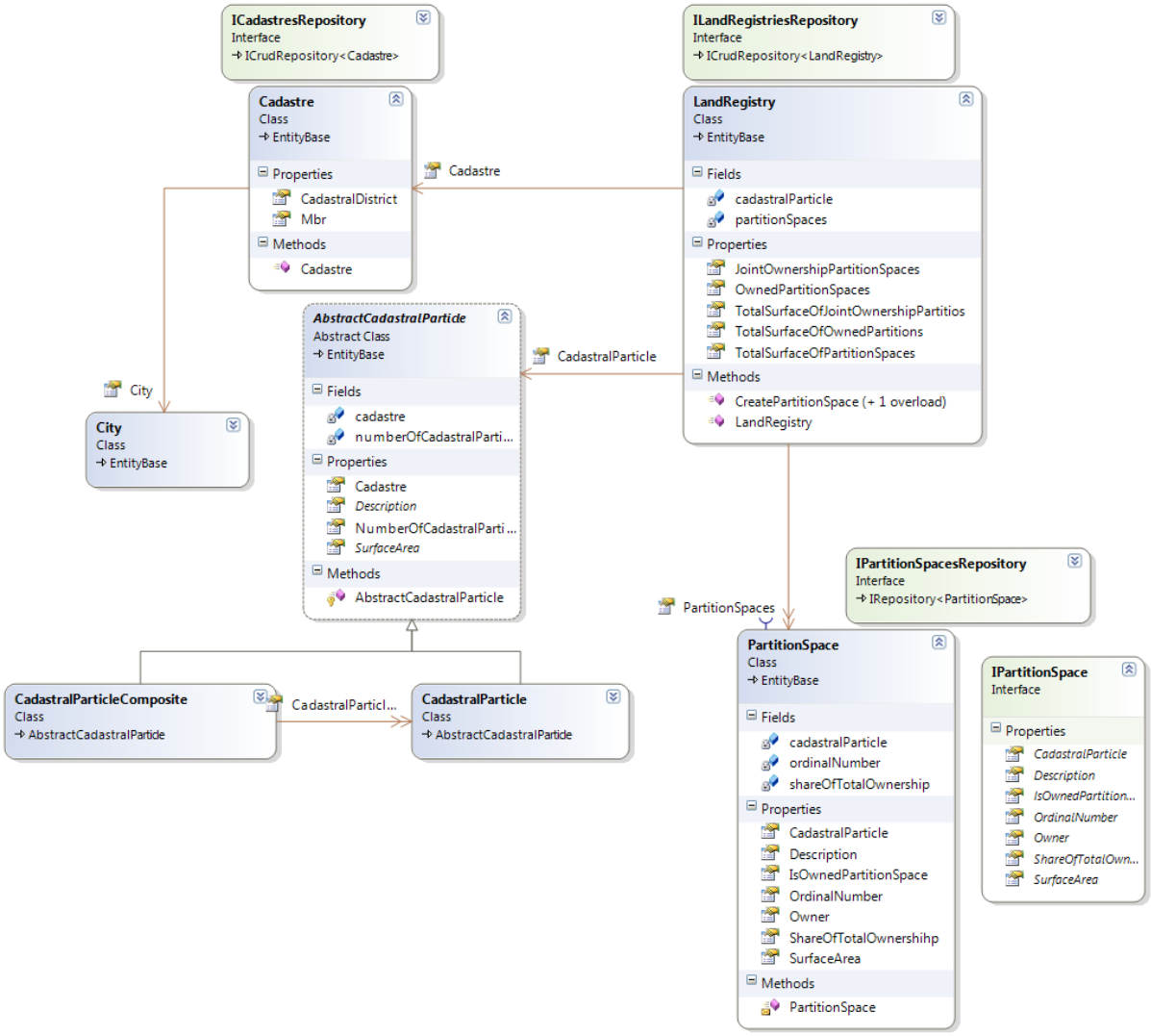


Slika 9 - Vrijednosni objekti osobe

4.3 Zakonodavstvo

Sljedeći dio domene je zakonodavstvo. Prije smo rekli da zgrada mora biti etažirana, odnosno da su određeni zajednički dijelovi zgrade kao i dijelovi pod posebnim vlasništvom. Iz tih smo razloga odlučili modelirati problematiku zakonodavstva.

Kao središte zakonodavstva definiran je korijenski agregat Zemljišna knjiga, odnosno razred *LandRegistry*. Taj se agregat sastoji od katastra (*Cadastre*), zemljišnih čestica *(*obitelj razreda *AbstractCadastralParticle*), te etaža (*PartitionSpace*).

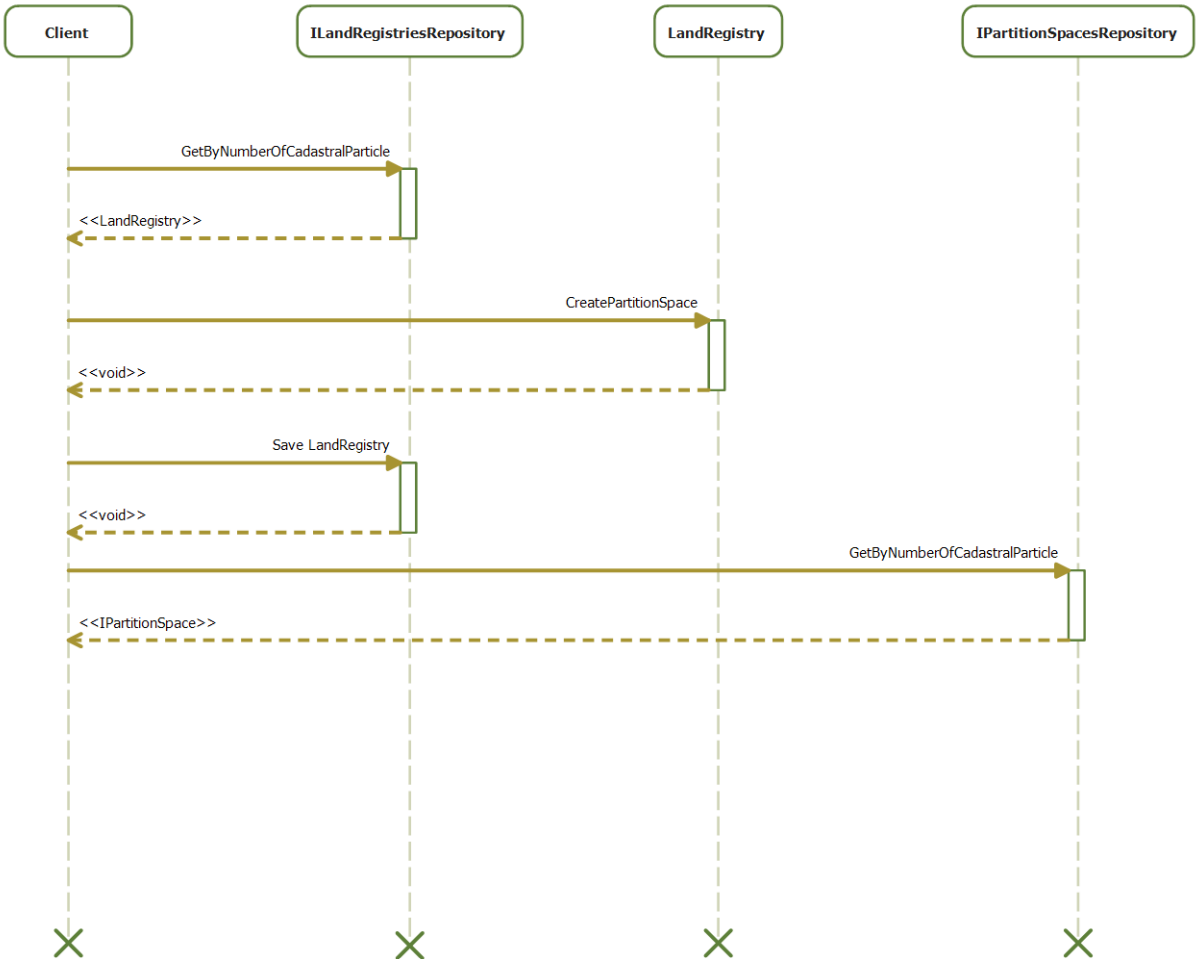


Slika 10 - Elementi zakonodavstva

Iz definicije agregata poznato je da korijenski agregat provodi sve invarijante agregata. Ovo je posebice bitno u slučaju etaža (*PartitionSpace*). Etaža je ponekad zanimljiva izvan samog agregata, odnosno zemljišne knjige. Iz tog je razloga etaža definirana kao korijenski agregat kako bismo ju mogli dohvatiti iz repozitorija. Međutim, stvaranje etaže izvan konteksta određene zemljišne knjige nema nikakvoga smisla. Stoga je etažu moguće stvoriti isključivo (ostvareno preko internog konstruktora etaže – nije vidljivo izvan prostora imena etaže) preko korijenskog agregata *LandRegistry* i to preko metode *CreatePartitionSpace*. Metoda *CreatePartitionSpace* predstavlja tvornicu ostvarenu preko oblikovnog obrasca

„Factory method“, gdje se enkapsulira stvaranje etaže. Etažu je moguće perzistirati isključivo preko repozitorija zemljišne knjige, odnosno spremanjem korijenskog agregata spremaju se svi ostali članovi agregata, a to je u ovome slučaju etaža. Eksplicitni repozitorij etaže služi isključivo radi čitanja. Etaža izvan agregata je dostupna preko sučelja *IPartitionSpace* koje ima ulogu vrijednosnog objekta. Dakle, takav objekt je moguće koristiti isključivo sa ciljem čitanja. Sve ostale promjene, kao što je promjena vlasnika etaže, obavljaju se u okvirima agregata zemljišne knjige.

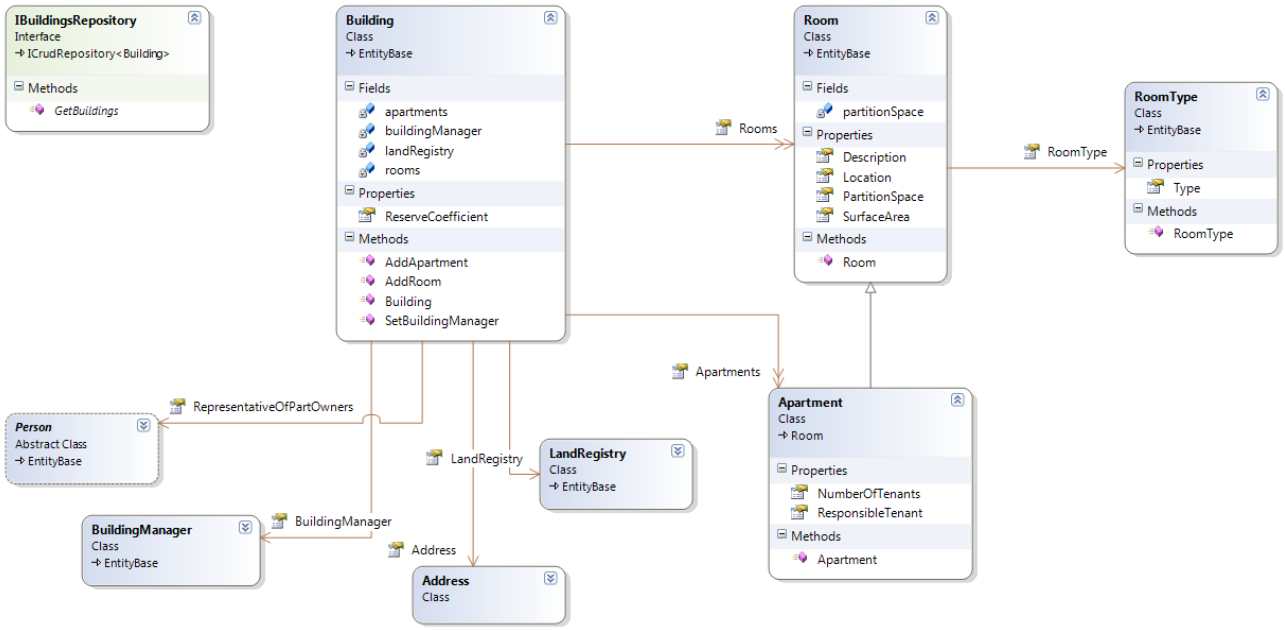
Na sljedećoj slici prikazan je prethodno opisan proces stvaranja etaže.



Slika 11 - Sekvencijalni dijagram koji opisuje proces stvaranja etaže

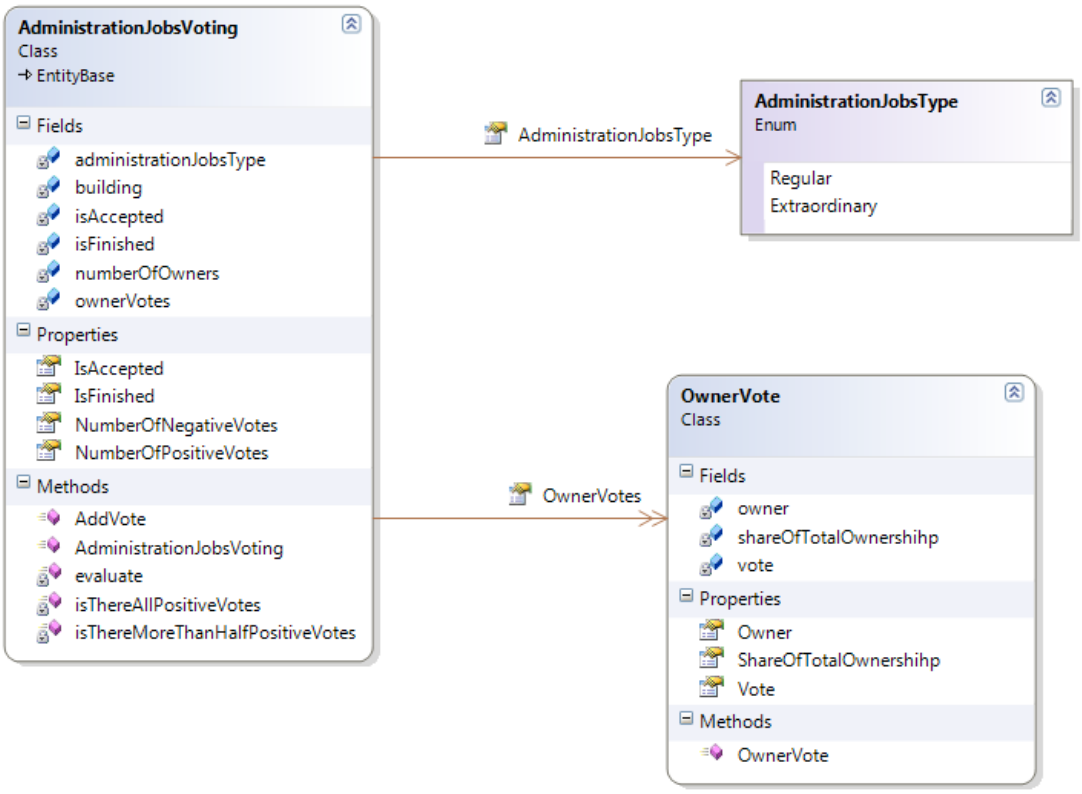
4.4 Upravljanje zgradom

Središnji korijenski agregat je zgrada, odnosno razred *Building*. Agregat se sastoji od predstavnika stanara koji je predstavljen apstraktnim razredom *Person*, upravitelja zgrade *BuildingManager*, vrijednosnog objekta *Address*, zemljišne knjige *LangRegistry*, liste stanova *Apartment* i ostalih prostorija *Room* (zajedničkih i vlasničkih).



Slika 12 - Agregat zgrade

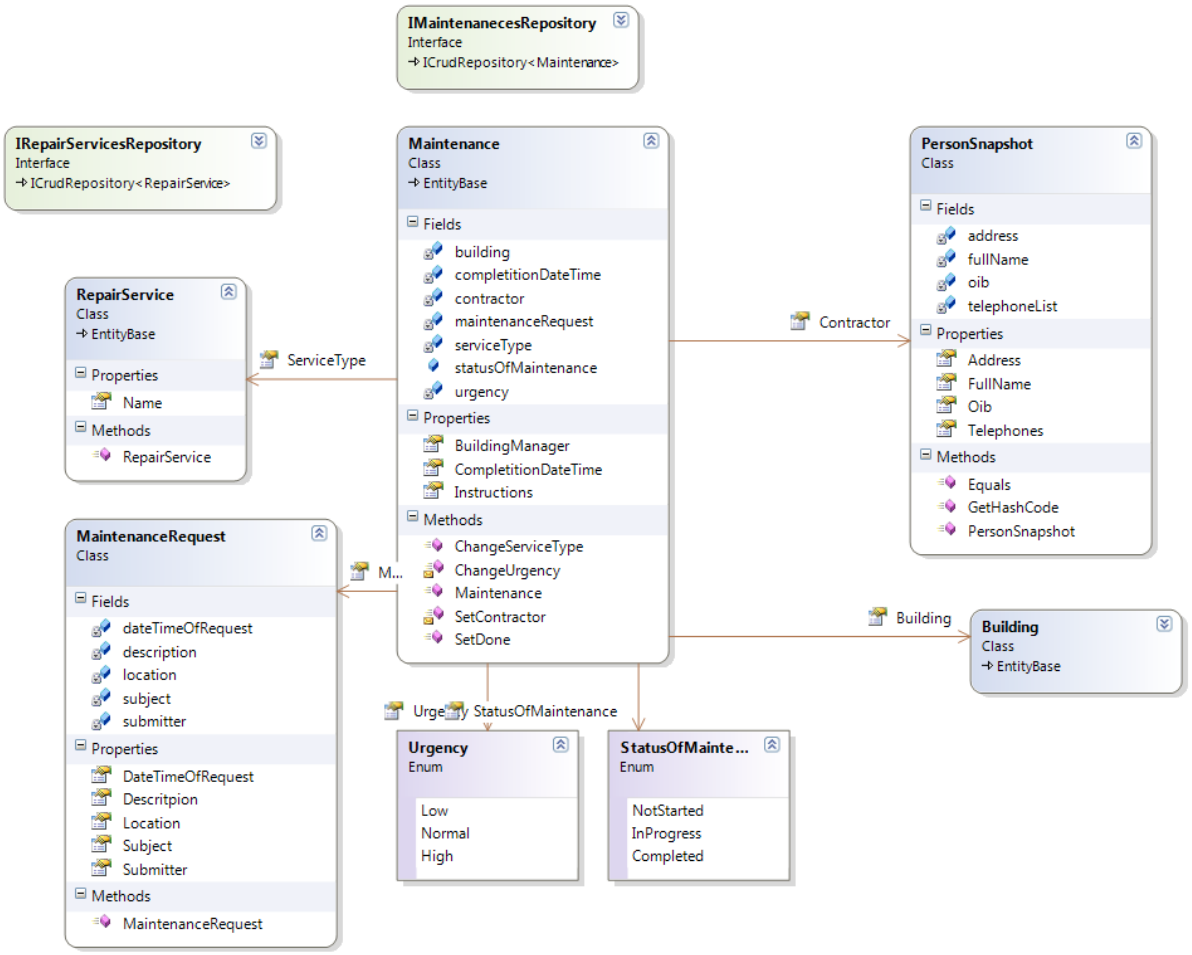
Upravljanje zgradom je također definirano poslovima redovne i izvanredne uprave. Redovna i izvanredna uprava modelirana je po principu glasovanja, odnosno definirali smo korijenski agregat *AdministrationJobsVoting* koji se sastoji od vrijednosnog objekta *OwnerVote* i vrste uprave *AdministrationJobsType*. Ovisno o vrsti uprave korijenski agregat primjenjuje invarijante agregata, odnosno kada su pravila o skupljenom broju glasova zadovoljena. Objekt *OwnerVote* predstavlja vrijednosti objekt koji jedino živi unutar granice agregata i čija je svrha predstavljanje vlasničkog glasa s obzirom na vlasnički udio.



Slika 13 - Agregat redovne i izvanredne uprave

4.5 Prijava kvara

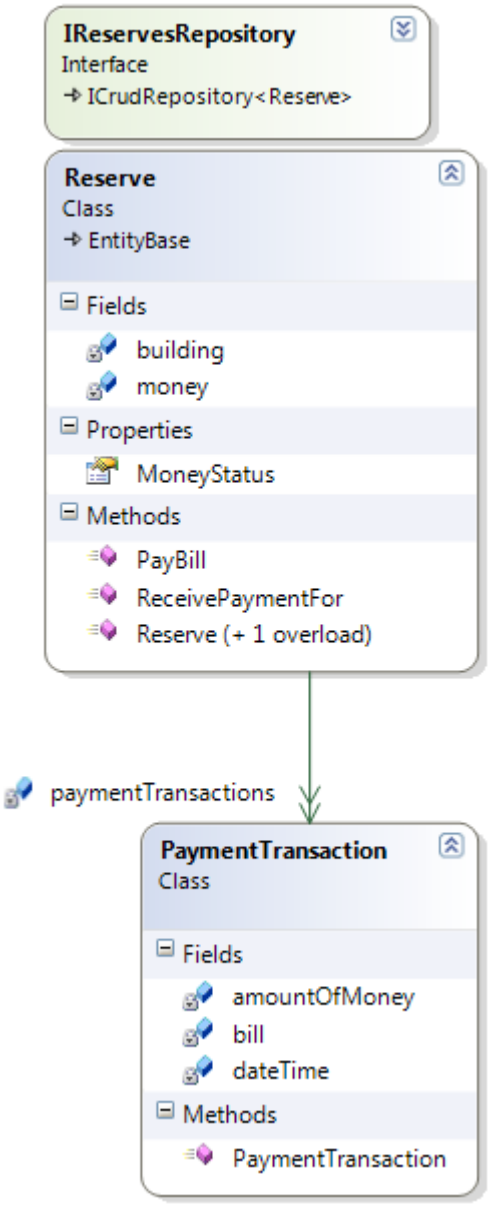
Suvlasnici mogu prijaviti kvarove upravitelju. Upravitelj za prijavljene kvarove angažira svoje kooperante za sanaciju kvarova. Predstavnik suvlasnika mora potvrditi da je posao sanacije kvara obavljen. Navedeni zahtjevi su ostvareni preko korijenskog agregata održavanja – *Maintenance*. Korijenski agregat održavanja ostvaren je preko usluge održavanja (*RepairService)*, zahtjeva za održavanjem ili popravkom (*MaintenanceRequest*), hitnosti (*Urgency*), statusa popravka (*StatusOfMaintenance*), izvođača radova (*PersonSnapshot*) i zgrade (*Building*). Stvaranje korijenskog agregata ostvareno je predajom u konstruktor parametra *MaitenanceRequest* koji je po definiciji vrijednosni objekt. Pridjeljivanje izvođača radova za sanaciju kvara je prema van omogućeno preko korijenskog agregata, dok se invarijante provode kroz članove agregata, odnosno dodjeljivanje je moguće ukoliko se izvođač radova nalazi u upraviteljevoj listi izvođača radova.



Slika 14 - Agregat održavanja

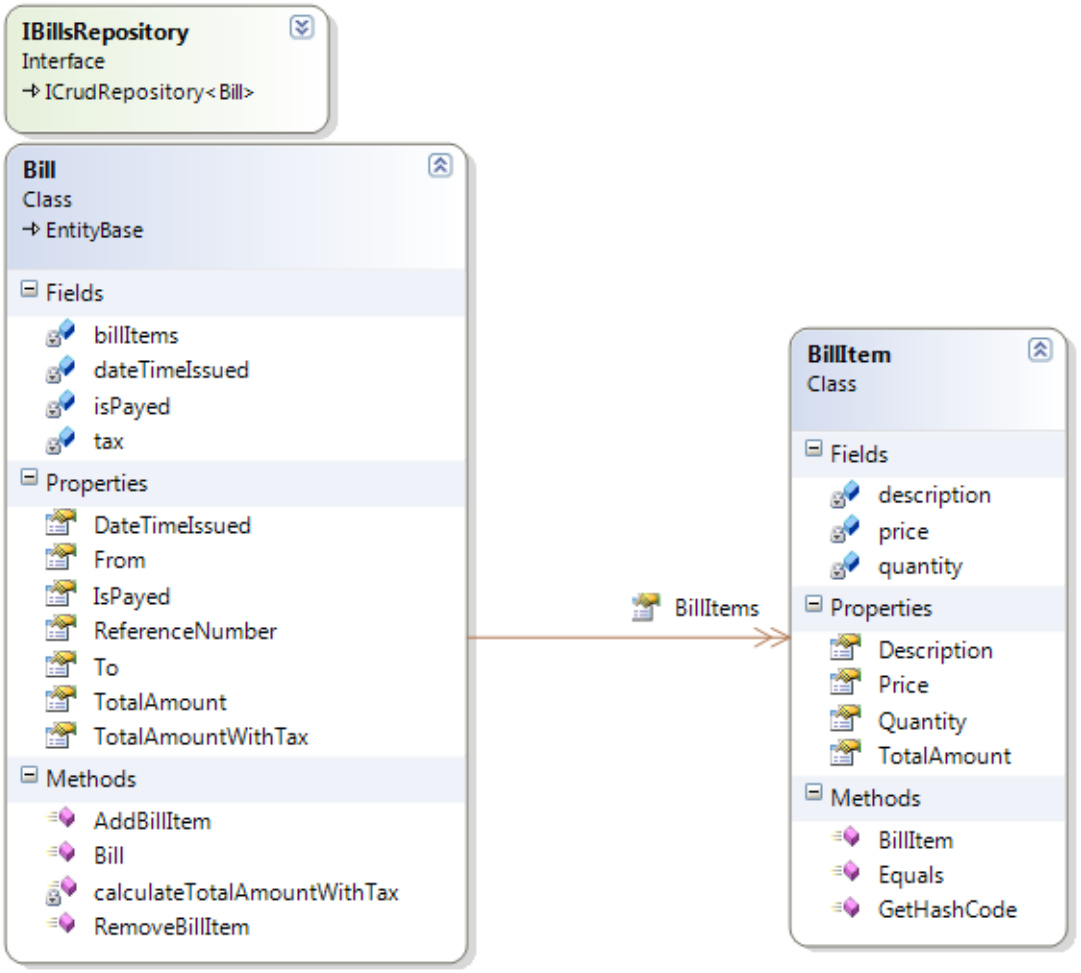
4.6 Financije

Financije u zgradi su definirane preko pričuve. Pričuva je u pravilu račun s novcem preko kojeg se plaća održavanje, te se primaju mjesečne uplate suvlasnika. Dakle, pričuva je definirana preko korijenskog agregata *Reserve* sa pripadajućim repozitorijem. Unutar pričuve, tj. agregata se nalazi povijest transakcija koja je ostvarena preko razreda *PaymentTransaction*.



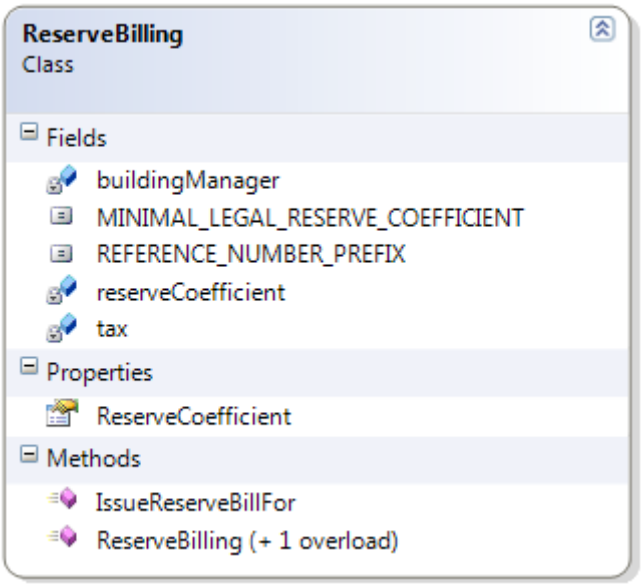
Slika 15 - Agregat pričuve

Kako bi se iz pričuve mogli plaćati računi, te primati uplate temeljem nekog računa definiran je korijenski agregat *Bill*. Agregat *Bill* sastoji se od stavki računa koje su jedino interesantne unutar okvira agregata, pa stoga nisu dobavljive preko repozitorija.



Slika 16 - Agregat računa

Kako bismo mogli izdavati račune za naplatu pričuve definirali smo servis koji će provoditi pravila formiranja cijene, a time i izdavanja računa. Navedeni servis će sukladno zakonskim propisima o minimalnom koeficijentu po površini i koeficijentu određenom od strane upravitelja formirati cijenu za naplatu pričuve. Kao rezultat servisa dobivamo račun (*Bill*).



Slika 17 - Servis za izdavanje računa za pričuvu

5. Zaključak

Razvoj vođen domenom, odnosno DDD, preporuča se za projekte sa kompleksnom domenom čime je olakšano snalaženje i održavanje cjelokupnog informacijskog sustava. Međutim, DDD je popularan i kod manjih projekata posebice prilikom korištenja razvojnih okvira koji se zasnivaju na MVC-u.

DDD zahtjeva veliku disciplinu prilikom razvijanja od strane razvojnika, jer ne postoje alati niti jezični konstrukti za provođenje pravila koje postavlja DDD. Dakle odgovornost je na razvojniku da osigura sve zahtjeve nad modelom koje zahtjeva DDD. Možemo reći da je prilikom prakticiranja DDD u razvoju informacijskog sustava potrebno dosta vremena kako bi se osjetio napredak, no kasnije se to nadoknađuje u održavanju i proširivanju informacijskog sustava.

Premda DDD zastupa pojam „Persistence ignorance“, odnosno da prilikom primjene DDD nismo opterećeni perzistencijom i infrastrukturnim slojem, već je glavni fokus isključivo na domeni, to u praksi nije posve točno. Kako sustavi i tehnologije nisu savršeni potrebno je razmišljati o implementacijskim detaljima zbog performansi

sustava prilikom modeliranja domene. Često puta ćete se naći u situaciji i razmišljati da li će korijenski agregat sadržavati listu elemenata članova agregata, ili će se ti elementi dohvaćati iz repozitorija, da li je isplativo stvaranje složenog agregata radi dohvaćanja jednog člana agregata obilaskom po objektnom grafu ili je bolje isti član dohvatiti iz repozitorija. Nekog posebnog pravila nema, odnosno potrebno je pronaći što bolji kompromis između koncepata domene i implementacijskih detalja. U takvim odlukama nam često puta pomažu koncepti poput lijene evaluacije[4] (*engl. lazy evaluation*) i od .NET 3.5 u programskom jeziku C# *expression tree*[5]*i LINQ*[6] koji može odgoditi izvođenje upita na bazu do trenutka kada je to zbilja potrebno.

6. Diplomski rad

Za diplomski rad predlažem izradu kompletnog informacijskog sustava za upravljanje zgradama, gdje je jezgra model dizajniran primjenom DDD-a. Implementacija infrastrukturnog sloja, odnosno repozitorija, ostvarila bi se pomoću objektno – relacijskog mapera NHibernate. Prezentacijski i aplikacijski sloj bio bi ostvaren pomoću MVC obrasca, odnosno ASP.NET MVC 2.

7. Literatura

1. *Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison Wesley, 2003.*

2. *Jimmy Nilsson: Applying Domain-Driven Design And Patterns: With Examples in C# and .NET, Addison Wesley, 2006.*

3. *Abel Avram & Floyd Marinescu: Domain-Driven Design Quickly, 2006.* URL: http://www.infoq.com/minibooks/domain-driven-design-quickly

[4] Lazy evaluation - http://en.wikipedia.org/wiki/Lazy_evaluation

[5] Expression tree - http://msdn.microsoft.com/en-us/library/bb397951.aspx

[6] LINQ - http://en.wikipedia.org/wiki/Language_Integrated_Query

4. *Srini Penchikala: InfoQ: Domain Driven Design and Development In Practice, 2008.*
   URL: http://www.infoq.com/articles/ddd-in-practice

5. *Domain-Driven Design Community*
   URL: http://domaindrivendesign.org/

6. *Domain Driven Design*
   URL: http://en.wikipedia.org/wiki/Domain-driven_design (2010)

7. *Djanji R. Prasanna: Dependency Injection, Manning, 2009.*

8. *Upravljanje i održavanje zgrada*
   URL: http://www.maksimus.hr/upravljanje/ (2010)