

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 167

**OBLIKOVANJE INFORMACIJSKOG
SUSTAVA ZA UPRAVLJANJE STAMBENIM
ZGRADAMA VOĐENO DOMENOM
PRIMJENE**

Željko Tepšić

Zagreb, lipanj 2011.

**SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA**

Zagreb, 17. veljače 2011.

DIPLOMSKI ZADATAK br. 167

Pristupnik: **Željko Tepšić**
Studij: **Računarstvo**
Profil: **Programsko inženjerstvo i informacijski sustavi**

Zadatak: **Oblikovanje informacijskog sustava za upravljanje stambenim zgradama vođeno domenom primjene**

Opis zadatka:

Primjenom postupka oblikovanja vođenog domenom primjene (engl. domain driven design - DDD) specificirati, modelirati i ostvariti informacijski sustav za upravljanje stambenim zgradama. Informacijski sustav ostvariti kao web primjenski program. Uz temeljni DDD pristup u ostvarenju informacijskog sustava potrebno je uporabiti i ostale moderne postupke razvoja kao što su oblikovni obrasci (engl. design patterns), integracija i preslikavanje objektne i relacijske paradigme (eng. object-relational mapping) te raspodijeljena arhitektura oblika model-izgled-nadglednik (engl. model-view-controller, MVC).

Zadatak uručen pristupniku: 25. veljače 2011.

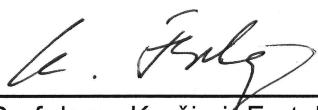
Rok za predaju rada: 10. lipnja 2011.

Mentor:



Prof.dr. sc. Nikola Bogunović

Predsjednik odbora za
diplomski rad profila:



Prof.dr.sc. Krešimir Fertalj

Djelovođa:



Doc.dr.sc. Boris Vrdoljak

Sadržaj

1.	Uvod	9
2.	Razvoj vođen domenom (engl. <i>Domain – Driven Design</i>)	11
2.1	Sveprisutni jezik	11
2.2	Građevni blokovi DDD-a.....	12
2.2.1	Slojevita arhitektura.....	13
2.2.2	Entiteti (engl. <i>entities</i>)	15
2.2.3	Vrijednosni objekti (engl. <i>value objects</i>).....	17
2.2.4	Servisi (engl. <i>services</i>).....	18
2.2.5	Moduli (engl. <i>modules</i>)	19
2.2.6	Agregati (engl. <i>aggregates</i>).....	20
2.2.7	Tvornice (engl. <i>factories</i>)	21
2.2.8	Repozitoriji (engl. <i>repositories</i>)	24
2.3	Kontinuirano refaktoriranje	25
2.4	Omeđen kontekst (engl. <i>bounded context</i>).....	26
2.5	Razvoj vođen testiranjem (engl. <i>Test Driven Development</i>)	26
3.	Perzistencija modela domene	28
3.1	Perzistencija i relacijske baze podataka	28
3.1.1	Perzistencija u objektno – orijentiranim aplikacijama	29
3.2	Objektno - relacijsko mapiranje pomoću Nhibernatea	30
3.2.1	Neusklađenost paradigm	31

3.2.2	Jedinice posla i transakcije	32
3.2.3	Arhitektura Nhibernatea	33
3.2.4	Implementacija modela domene	35
3.2.5	Definiranje meta podataka o mapiranju.....	39
3.2.6	Životni ciklus NHibernate objekata.....	52
4.	ASP.NET MVC	54
4.1	Arhitektura ASP.NET MVC aplikacije	54
5.	Strukturirana analiza i specifikacija informacijskog sustava za upravljanje stambenim zgradama.....	57
5.1	Opis procesa upravljanja i održavanja stambenih objekata	57
5.2	Specifikacija zahtjeva sustava.....	59
5.2.1	Poslovni zahtjevi	59
5.2.2	Korisnički zahtjevi.....	60
5.2.3	Funkcionalni zahtjevi	60
5.2.4	Nefunkcionalni zahtjevi.....	61
5.2.5	Poslovna pravila	61
5.3	Slučajevi korištenja.....	63
5.3.1	Registracija upravitelja ili izvođača radova u sustav.....	63
5.3.2	Registracija suvlasnika u sustav	64
5.3.3	Kreiranje stambene zgrade	65
5.3.4	Kreiranje zemljišne knjige.....	65
5.3.5	Dodavanje etaža u zemljišnu knjigu.....	66

Sadržaj

5.3.6	Kreiranje glasovanja – poslovi redovne i izvanredne uprave	67
5.3.7	Glasovanje suvlasnika za poslove redovne i izvanredne uprave	68
5.3.8	Prijava kvara	68
5.3.9	Revizija kvara	69
5.3.10	Obrada kvara.....	69
5.3.11	Potvrđivanje popravka	70
5.3.12	Izdavanje računa zgradi za obavljene popravke	70
5.3.13	Plaćanje troškova zgrade	71
5.3.14	Izdavanje računa za pričuvu	71
6.	Implementacija informacijskog sustava za upravljanje stambenim zgradama.....	72
6.1	Implementacija modela domene	73
6.1.1	Apstrakcije.....	75
6.1.2	Osobe i uloge	79
6.1.3	Zakonodavstvo	82
6.1.4	Upravljanje zgradom	84
6.1.5	Prijava kvara	86
6.1.6	Financije	88
6.2	Baza podataka	90
6.2.1	Osobe i uloge	90
6.2.2	Zakonodavstvo	91
6.2.3	Upravljanje zgradom	92

6.2.4	Prijava kvara	93
6.2.5	Financije	94
6.2.6	Korisnici i uloge	95
6.3	Implementacija infrastrukturnog sloja	97
6.3.1	Sjednice i transakcije	97
6.3.2	Repozitoriji	98
6.3.3	Autentifikacija i autorizacija	102
6.3.4	Servisi	107
6.4	Implementacija aplikacijskog i prezentacijskog sloja.....	108
6.4.1	Nadglednici (engl. <i>controllers</i>).....	108
6.4.2	Modeli pogleda	110
6.4.3	Pogledi	111
7.	Upute za korištenje informacijskog sustava za upravljanje stambenim zgradama ...	112
7.1	Upute za postavljanje sustava	112
7.2	Korisničke upute	113
7.2.1	Osnovni dijelovi korisničkog sučelja	113
7.2.2	Prijava na sustav	114
7.2.3	Registracija korisnika	115
7.2.4	Zgrade i zemljишne knjige	115
7.2.5	Rad uprave	118
7.2.6	Kvarovi	119

Sadržaj

7.2.7 Financije	121
8. Zaključak	123
9. Literatura	125
10. Naslov, sažetak i ključne riječi	126
11. Title, Abstract and keywords.....	127
12. Dodatak A – Korisnički podaci IS za bazu podataka sa prezentacijskim primjerima	
128	

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 (attribute) i ponašanja slična onima koja pronalazimo u stvarnome svijetu. Pojednostavljeni rečeno, više takvih objekata čine model.

Bez modela, informacijski sustavi nemaju nikakvu vrijednost. Poslovna logika zna biti veoma složena jer opisuje mnogo različitih slučajeva i ponašanja, a njihovo modeliranje najbolje se ostvaruje objektima.

Često puta previše se vremena troši samo na tehnologiju i implementaciju, a kontekst, odnosno domena i poslovni procesi se stavljuju 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.

Model domene stvara mrežu međusobno povezanih objekata, gdje svaki objekt predstavlja neki značajan entitet, bilo on velik kao korporacija ili mali kao linija na obrascu. Stavljanje modela domene u aplikaciju uključuje umetanje cijelog sloja objekata koji modeliraju poslovnu domenu koja se rješava. Objekti oponašaju podatke iz poslovnoga svijeta te nad njima provode poslovna pravila.

Površinski, objektno – orijentirani model domene često puta liči na model baze podataka, međutim on se uvelike razlikuje. Model domene čini spregu podataka i poslovnih procesa, posjeduje višestruke vrijednosti atributa i kompleksnu mrežu asocijacija te koristi nasljeđivanje, različite strategije i ostale oblikovne obrasce. Bogati modeli domene su odlični za rješavanje složene poslovne logike, međutim uvode komplikaciju preslikavanja u model baze podataka. Upravo zbog toga bogati modeli domene zahtijevaju korištenje

podatkovnih mapera, odnosno objektno – relacijskih mapera. Objektno – relacijski maperi će pomoći da model domene bude neovisan o bazi podatka te je najbolji pristup za razrješavanje slučajeva kada se model domene i model baze podataka razlikuju.

Informacijski sustav je u službi rješavanja kompleksnih poslovnih problema u interakciji sa korisnicima. Model – pogled – nadglednik (engl. *model – view - controller*, MVC) kao arhitekturalni obrazac razdvaja domensku logiku od korisničkog sučelja (unos i prezentacija) time omogućujući neovisan razvoj, testiranje i održavanje (engl. *separation of concerns, SoC*), što je jedna od fundamentalnih heuristika dobrog programskega dizajna.

U ovome diplomskom radu, primjenom postupaka oblikovanja vođenog domenom primjene (engl. *domain – driven design – DDD*), specificirati ćemo, modelirati i ostvariti informacijski sustav za upravljanje stambenim zgradama. Informacijski sustav ostvariti ćemo kao web primjenski program korištenjem MVC obrasca i relacijske baze podataka. Za perzistenciju bogatog modela domene koristi ćemo NHibernate kao objektno – relacijski maper. Informacijski sustav biti će implementiran programskim jezikom C# i .NET razvojnom okolinom.

Kroz nekoliko slijedećih poglavlja teoretski ćemo predstaviti glavne koncepte razvoja vođenog domenom, njegove gradbene elemente. Upoznati ćemo čitatelja sa problemima neusklađenosti objektne i relacijske paradigme u okviru perzistencije objektnog modela te rješenjem kao objektno – relacijskog mapera. Ukratko ćemo nešto reći o MVC-u i njegovoj web implementaciji u .NET razvojnoj okolini. Nadalje, kroz strukturiranu analizu specificirati ćemo zahtjeve informacijskog sustava za upravljanje stambenim zgradama. U posljednjim poglavljima opisati ćemo, korak po korak, dizajn modela domene te implementaciju informacijskog sustava koristeći se prethodno opisanim obrascima, paradigmama i tehnologijama. A na samome kraju mogu se pronaći upute za korištenje ostvarenog informacijskog sustava.

2. Razvoj vođen domenom (engl. *Domain – Driven Design*)

Razvoj programske potpore često je 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 iz kojeg se razloga 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 prepostavke DDD-a 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 uvjek 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.

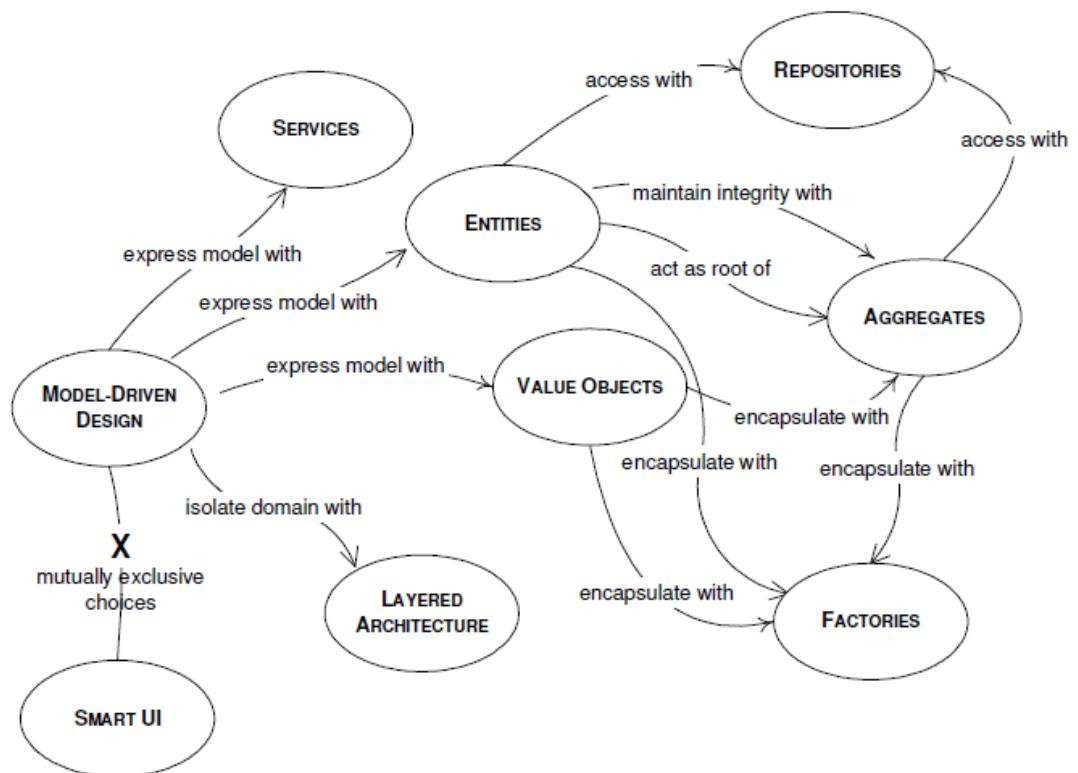
Prilikom izrade modela, potrebno je komunicirati zbog razmjene ideja i zbog elemenata koji su uključeni u model. Komunikacija na ovoj razini je iznimno važna za uspjeh projekta.

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 funkcioniра.

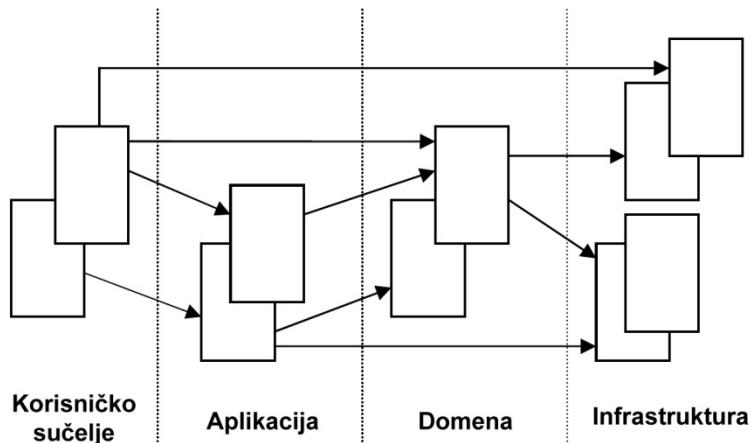
2.2 Građevni blokovi DDD-a

Sljedeća potpoglavlja 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 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 – orijentiranoj aplikaciji, korisničko sučelje, baza podataka i ostali podupirući kod se često puta ugrađuje u poslovne objekte. Poslovna logika i kod za pristup bazi podataka se smješta u korisnička sučelja.

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 mesta, 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 arhitekturne obrasce radi postizanja što manje međuvisnosti 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 arhitekturno 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. Sadrži sloj za perzistenciju podataka.

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 se ne bi 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 persistira 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. 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 attribute sadrži.

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.

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 attribute 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 attribute 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.

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 persistencije objekata svaki puta kada se objekt dohvata iz baze podataka nova instanca objekta se stvara, a time se gubi inicijalni identitet, memorijska lokacija – referenca.

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 pojednostavimo 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 rješava se 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 instance dijeli, onda se šalje samo referenca što zahtjeva da se šalje poruka svaki put 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.

Mnogi domenski i aplikacijski servisi su izgrađeni na populaciji entiteta i vrijednosnih objekata. Često puta su entiteti i vrijednosni objekti prefini za prikidan 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 međuvisnosti. Moduli se moraju sastojati od elemenata 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 međuvisnosti. Mala međuvisnost smanjuje složenost i olakšava održavanje.

2.2.6 Agregati (engl. aggregates)

Mnoge poslovne domene imaju jako međupovezan skup objekata. Potrebno je ići po dugačkim i dubokim stazama slijedeći reference na objekte. Problem je nepostojanje oštih 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.

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 aggregata 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 aggregata mogu držati referencu na korijene drugih aggregata
 - Prilikom brisanja aggregata, operacija brisanja mora odjednom obrisati sve elemente aggregata.

2.2.7 Tvornice (engl. *factories*)

Stvaranje objekta samo po sebi može biti složena operacija. 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. 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 funkcioniра, 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 konkretni 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
- Abstract factory
- Builder

Dva osnovna zahtjeva za svaku dobру 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,

upravljujući 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 komplikirana.
- 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. 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 asocijacije 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 spremlijen 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 aggregate, 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 strukturu. Upiti na bazu mogu prekršiti enkapsulaciju na domenske objekte i aggregate. Izlaganje tehničke infrastrukture i mehanizma pristupa bazi podataka komplicira klijentski dio aplikacije i zamagluje 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 memoriske kolekcije svih objekata tog tipa. Pristup takvima 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 aggregate 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.

2.3 Kontinuirano refaktoriranje

Za vrijeme trajanja procesa dizajniranja i razvoja moramo stati s vremenom 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 dizajnu.

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 eksplizitno 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 (engl. *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 persistencije.

Razvoj vođen testiranjem je metodologija oblikovanja programske potpore koja koristi *unit* testove¹ 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.
- Napraviti preoblikovanje svojeg koda (*Refactor*) – Nakon što je programski kod zadovoljio test, potrebno je napraviti korak natrag i razmisiliti o arhitekturi aplikacije. Dakle potrebno je napraviti preoblikovanje programskog koda koristeći oblikovne obrasce u programiranju.

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

3. Perzistencija modela domene

U ovome poglavlju govoriti ćemo o perzistenciji podataka te o objektno – relacijskom mapiranju kao načinu za iskorištavanje najboljeg iz svijeta relacijskih baza podataka i objektno – orijentiranog programiranja. Navesti ćemo glavne probleme neusuglašenosti obiju paradigmi te predstaviti NHibernate kao ORM za .NET razvojnu okolinu. Definirati ćemo pravila i postupke za ostvarenje mapiranja pomoću NHibernatea.

3.1 Perzistencija i relacijske baze podataka

Perzistencija predstavlja ključni element u razvoju aplikacija. Skoro sve aplikacije zahtijevaju perzistentne podatke. Perzistencija omogućava da podaci budu postojani, odnosno sačuvani, čak i kada programi koji koriste te podatke nisu pokrenuti.

Često puta suočeni smo sa odlukom perzistencije, odnosno koji mehanizam perzistencije odabrati. Jedan on jednostavnijih mehanizama perzistencije je perzistencija podataka u tekstuallnu datoteku. Međutim najčešći način perzistencije podataka su relacijske baze podataka iz razloga što su široko prihvачene, robusne su i nude velike mogućnosti pouzdanog spremanja i dohvaćanja podataka.

Sustavi za upravljanje bazama podatka (engl. *Relational database management system*) nisu definirani samo za određenu tehnologiju te jednakako tako relacijske baze podataka nisu nužno određene za jednu aplikaciju. Dakle, možemo imati nekoliko aplikacija koje pristupaju jednoj bazi podataka, od kojih su neke napisane u .NET, neke u Javi, Rubyu itd.

Relacijska tehnologija predstavlja zajedničku poveznicu nesrodnih sustava i tehnoloških platformi u međusobnom dijeljenju podataka. Relacijski podatkovni model je često puta standard u *enterprise* (poslovnoj) reprezentaciji poslovnih objekata (engl. *business objects*): poduzeća trebaju spremiti informacije o različitim stvarima poput klijenata, računa i proizvoda te je relacijska baza podataka najčešće centralno mjesto gdje su takve informacije definirane i spremljene. Upravo takve činjenice postavljaju relacijske baze podataka kao važni dio IT okoline.

3.1.1 Perzistencija u objektno – orijentiranim aplikacijama

U objektno – orijentiranim aplikacijama, perzistencija omogućuje objektu da nadživi procese ili aplikaciju koja ga je kreirala. Stanje objekta može biti spremljeno na disku te se na temelju tih podataka može ponovno stvoriti novi objekt sa istim stanjem u nekom trenutku u budućnosti. Perzistencija nije ograničena samo na jedan objekt, već je moguće spremiti i ponovno rekonstruirati kompletna objektna stabla. Određeni dio objekata se ne perzistira, to su tzv. tranzijentni objekti. Tranzijentni objekt je objekt koji ima ograničeno vrijeme života koje je određeno životom procesa koji je instancirao taj objekt.

Moderne relacijske baze podataka nude strukturiranu reprezentaciju perzistiranih podataka, omogućujući sortiranje, pretraživanje i grupiranje podataka. Sustavi za upravljanje bazama podataka su odgovorni za konkurentnost i integritet podataka. Stoga kada govorimo o perzistenciji, konkretno u kontekstu objektno – orijentiranih aplikacija koje koriste model domene, tada mislimo da sljedeće stvari:

- Spremanje, organiziranje i dohvrat strukturiranih podataka
- Konkurentnost i integritet podataka
- Dijeljenje podataka

Aplikacija sa modelom domene ne radi direktno sa relacijskom reprezentacijom poslovnih entiteta, već aplikacija ima svoj vlastiti, objektno – orijentirani model poslovnih entiteta. Tako da umjesto direktnog rada sa n-torkama i atributima relacije, odnosno redcima i stupcima tablice, poslovna logika komunicira sa objektno – orijentiranim modelom domene koji je za vrijeme rada aplikacije (engl. *runtime*) realizacija grafa povezanih objekata.

Poslovna logika se nikada ne izvršava u bazi podataka, kao spremljene SQL procedure, već je implementirana u nekom od programskih jezika. To omogućava poslovnoj logici korištenje sofisticiranih objektno – orijentiranih koncepata kao što su nasljeđivanje i polimorfizam.

Nisu sve aplikacije dizajnirane na takav način, niti to trebaju biti. Za jednostavne aplikacije nije potrebno izraditi model domene, već je dovoljno koristiti SQL upite i tablične reprezentacije perzistentnih podataka jer je takav način jednostavan i dobro utvrđen. Međutim, u slučaju aplikacija sa ne trivijalnom poslovnom logikom, model domene pomaže u boljoj iskoristivosti programskog koda i boljoj sposobnosti održavanja.

NHibernate i općenito ORM su namijenjeni aplikacijama sa modelom domene. Programski kod za perzistenciju podataka najčešće se nalazi u sloju pristupa podacima, odnosno u infrastrukturnom sloju (Vidjeti poglavlje 2.2.1 Slojevita arhitektura).

3.2 Objektno - relacijsko mapiranje pomoću NHibernatea

U realnom svijetu rijetko kad se susrećemo sa jednostavnim aplikacijama. Enterprise aplikacije sadrže mnogo entiteta sa kompleksnom poslovnom logikom i dizajnerskim ciljevima kao što su produktivnost, održavanje i performanse.

Praksa je pokazala da su relacijske baze² podataka odlične za spremanje podataka, a objektno – orijentirano programiranje je odličan pristup za izradu kompleksnih aplikacija. Pomoću objektno – relacijskog mapiranja, moguće je stvoriti translacijski sloj koji će jednostavno transformirati objekte u relacijske podatke i obratno. Osim što će mehanizam objektno – relacijskog mapiranja upravljati objektima, ono može pružiti znatno više mogućnosti kao što su caching, transakcije i kontrola konkurentnosti.

Dakle, objektno – relacijsko mapiranje je automatizirana perzistencija objekata aplikacije u tablice relacijske baze podataka, koristeći meta podatke koji opisuju preslikavanje između objekata i baze podataka. U suštini ORM predstavlja transformaciju podataka iz jedne reprezentacije u drugu.

NHibernate je besplatno (*open source*) rješenje objektno – relacijskog mapiranja za .NET platformu, odnosno pruža radni okvir za mapiranje objektno – orijentiranog modela domene u tradicionalnu relacijsku bazu podataka. NHibernate je nastao na temelju popularnog Java objektno – relacijskog mapera Hibernate.

NHibernate automatizira mnoge ponavljajuće zadatke kodiranja, ali znanje tehnologije perzistencije mora postojati i izvan NHibernate-a ukoliko se želi iskoristiti puna snaga modernih SQL baza podataka.

² Postoje i objektne baze, međutim one nisu standardizirane te se razlikuju od proizvođača do proizvođača. Uz njih postoje i objektno – relacijske baze podataka koje smanjuju neusuglašenost relacijske i OO paradigmе, ali ne potpuno (nisu podržana sučelja, statički razredi, enkapsulacija itd) i krše logičku podjelu sloja poslovne logike i sloja pristupa podacima.

3.2.1 Neusklađenost paradigm

Baze podataka predstavljaju relacijsku paradigmu (baze koje su najviše u upotrebi te s kojima se bavimo u ovome radu), dok je model domene predstavljen objektno – orijentiranom paradigmom. Trenutno ne postoji direktni način za perzistenciju objekata u n-torku, odnosno redak relacije baze podataka. Perzistencija ne bi smjela ometati modeliranje entiteta koji korektno predstavljaju problem domene. Neusklađenost paradigm (engl. *paradigm mismatch* ili *object/relational impedance mismatch*), one objektne i one relacijske, odnosi se na fundamentalne nekompatibilnosti koje postoje između dizajna objekata i relacijskih baza podataka.

3.2.1.1 Problem granularnosti

Granularnost se odnosi na relativnu veličinu objekta s kojime se radi. Kada govorimo o .NET objektima i tablicama baze podataka, problem granularnosti se javlja kada želimo perzistirati objekte koji mogu imati različite vrste granularnosti u tablice i stupce čija granularnost je inherentno ograničena.

Problem granularnosti možemo prikazati na primjeru entiteta „korisnik“ i vrijednosnog objekta „adresa“. Dakle objekt „korisnik“ sastoji se od objekta „adresa“. Kako spremiti objekt „korisnik“ sa svim pripadajućim međuvisnostima u relacijsku bazu podataka? Nameće se nekoliko rješenja:

- Uz tablicu „korisnik“, kreirati tablicu „adresa“.
 - Problem: performanse.
- Kreirati korisnički definirani tip baze podataka.
 - Problem: portabilnost.
- Spojiti podatke o adresi sa podacima o korisniku
 - Problem: nije dobar objektno – orijentirani dizajn.

Rješenje ovog problema je opisano u poglavljju 3.2.5.2 *Mapiranje komponenti*.

3.2.1.2 Problem nasljeđivanja i polimorfizma

Objektno – orijentirani jezici podržavaju pojam nasljeđivanja, ali relacijske baze podataka tipično takvu ideju ne podržavaju. Izvedene i bazne klase definiraju drugačije podatke i funkcionalnost. Stoga, kako perzistirati hijerarhiju entiteta u relacijsku bazu podataka? Isto tako kako izvršavati polimorfne upite, jer pokazivač može pokazivati na baznu ili izvedenu klasu?

Postoji nekoliko različitih rješenja prikladnih za različite situacije. Rješenja su opisana u poglavlju 3.2.5.4 *Mapiranje nasljeđivanja razreda*.

3.2.1.3 Problem identiteta

Identitet n-torce u relacijskoj bazi podataka je predstavljen pomoću vrijednosti primarnog ključa. Identitet objekta nije jednak vrijednosti primarnoga ključa. Općenito se kod relacijskih baza podataka preporuča korištenje surogatnih ključeva (primarni ključ bez nekog posebnog značenja korisniku). .NET objekti imaju intrinzični identitet koji se temelji na njihovoj memorijskoj lokaciji ili na korisničkoj definiranoj konvenciji (implementacija metode `Equals()`).

Opisano rješenje se nalazi navedenog problema nalazi se u poglavlju 3.2.4.1 *Identitet objekta*.

3.2.1.4 Problem asocijacija

U objektnom modelu, asocijacije predstavljaju veze među objektima. Asocijacije među objektima se stvaraju koristeći reference objekta. U relacijskom svijetu, asocijacija je predstavljena stranim ključem (engl. *foreign key*). Objektne reference su inherentno usmjerene: asocijacija iz jednog objekta prema drugome. Ako bi asocijacija između objekata bila dvosmjerna, potrebno je definirati asocijaciju dva puta, jednom u svakom od povezanih razreda.

Dok s druge strane, asocijacije stranim ključem nisu po prirodi usmjerene. Usmjerenost nema nikakvog značenja u relacijskom podatkovnom modelu, jer je moguće stvoriti proizvoljne podatkovne asocijacije pomoću relacijskih spajanja i projekcija.

Mapiranje asocijacija biti će objašnjeno u poglavlju 3.2.5.3 *Mapiranje asocijacija*.

3.2.2 Jedinice posla i transakcije

Kada korisnici rade na aplikacijama oni izvode jedinične operacije. Te se operacije mogu definirati kao poslovne ili aplikacijske transakcije. Aplikacija može izvoditi operacije koje se tiču mnogih entiteta. Kada će ti entiteti biti učitani ili spremljeni ovisi o kontekstu.

Prepostavimo da aplikacija obavlja kompleksne transakcije koje uključuju mnoge promjene i brisanja. Ukoliko bi ručno pratili koje entitete treba spremiti ili obrisati, a

pritom osiguravajući da je svaki entitet učitan samo jednom, stvari mogu postati vrlo komplikirane.

NHibernate koristi *Unit of Work* obrazac (konkretno *The Identity Map*) za rješavanje takvih problema. Prilikom stvaranja entiteta ti se entiteti registriraju preko NHibernatea, a kasnije NHibernate prati sva učitavanja i prema potrebi sprema promjene. Na kraju transakcije, NHibernate određuje kada će te promjene provesti kroz bazu podataka i kojim redoslijedom.

NHibernate nudi *transparent persistence* značajku. Navedena značajka nam omogućuje dodavanje novih entiteta u kolekciju, koji će se spremiti onda kada spremimo objekt koji sadrži kolekciju. Na taj način ne moramo se brinuti o persistenciji novog objekta kojeg dodajemo u kolekciju.

Isto tako zbog problema veličine kolekcije NHibernate omogućava tzv. *lazy loading* značajku koja nam daje mogućnost učitavanja objekta sa kolekcijom svih elemenata u trenutku kada nam je to zaista potrebno, odnosno u trenutku pristupanja kolekciji.

NHibernate podržava dva mehanizma zaključavanja podataka: pesimističko i optimističko zaključavanje. Pesimističko zaključavanje blokira pristup podacima svim konkurenckim procesima sve dok traje transakcija i na razini je baze podataka. Pesimističko zaključavanje je na razini aplikacije, te zapravo blokira jedinicu posla bacajući iznimku pomoću koje se korisniku može signalizirati da su podaci izmijenjeni. Pesimističko zaključavanje ostvareno je upravljivim verzioniranjem dohvaćenih podataka, drugim riječima za svaku promjenu podataka povećava se broj verzije, a promjena je moguća samo onda ukoliko su vrijednosti dohvaćene verzije i one u bazi podataka jednake.

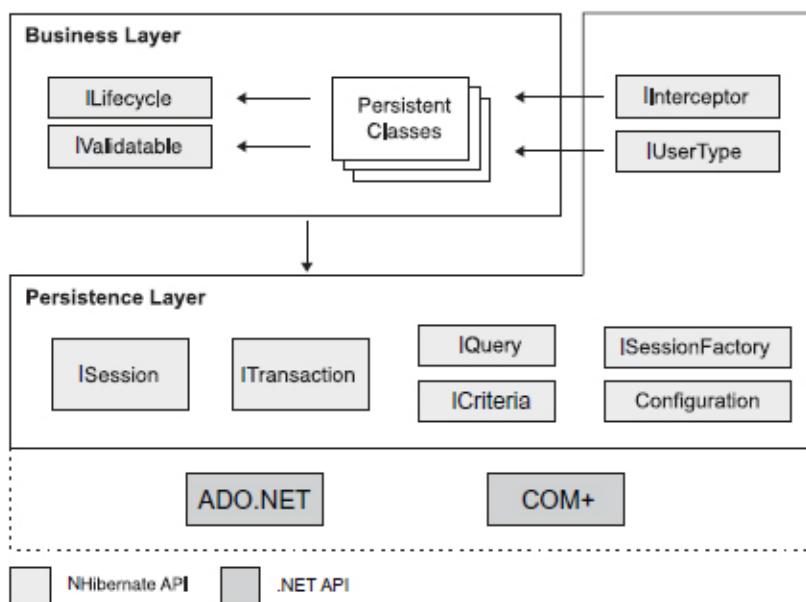
3.2.3 Arhitektura NHibernatea

NHibernate koristi postojeće .NET API-je, uključujući ADO.NET i njegov ITransaction API. ADO.NET nudi osnovnu razinu apstrakcije za funkcionalnosti relacijskih baza podataka, omogućujući NHibernateu da podrži skoro sve baze podataka sa ADO.NET upravljačkim programom.

Klasifikacija sučelja NHibernatea definira se kao:

- Sučelja koja poziva aplikacija za izvođenje osnovnih CRUD (*Create, Retrieve, Update i Delete*) operacija te operacija upita.
 - `ISession` sučelje – je primarno sučelje koje koriste NHibernate aplikacije. Ono izlaže NHibernateove metode za traženje, spremanje, ažuriranje i brisanje objekata. Instanca `ISession` je lagana i jeftina za stvaranje i uništavanje. Važno je napomenuti da NHibernate sjednice nisu dretveno sigurne (engl. *thread safe*), odnosno istovremeno samo jedna dretva može koristiti određenu sjednicu. NHibernate sjednice su nešto između konekcije i transakcije, odnosno kolekcija učitanih objekata povezana sa jedinicom posla.
 - `ISessionFactory` – sučelje zaduženo za instanciranje `ISession` sučelja. Stvaranje `ISessionFactory` instance je skupo te se tipično stvori jedna instance po aplikaciji i to prilikom pokretanja iste.
 - `ITransaction` – sučelje za upravljanje transakcijama. Nije ga obavezno koristiti, odnosno moguće je izraditi vlastitu infrastrukturu upravljana transakcijama.
 - `IQuery` – omogućuje izvođenje upita nad bazom podataka istovremeno kontrolirajući kako se upit izvodi. Osnovno sučelje za dohvaćanje podataka koristeći NHibernate. Upiti se mogu pisati u HQL-u ili u prirodnom SQL jeziku baze podataka.
 - `ICriteria` – omogućuje slične stvari kao i `IQuery` sučelje samo korištenjem objektno – orijentiranih kriterijskih upita.
- Sučelja za konfiguraciju NHibernatea, `Configuration` razred. Aplikacije koristi `Configuration` instancu za specificiranje lokacije dokumenata mapiranja.
- *Callback* sučelja koja omogućavaju aplikaciji da reagira na događaje koji se dešavaju unutar NHibernatea. Skup sučelja uključuje: `IInterceptor`, `ILifecycle` i `IValidatable`.
- Sučelja proširenja (engl. *Extension interfaces*) - Sučelja koja omogućuju proširenje NHibernate funkcionalnosti.
 - `IUserType`, `IComposite UserType` i `IIdentifierGenerator` omogućuju stvaranje vlastitih tipova podataka. NHibernate tip mapira .NET tip podatka u stupac u relacijskoj tablici.

- Generiranje primarnog ključa – `IIdentifierGenerator`
- Podrška SQL dijalektima – `Dialect`
- Caching strategije – `Icache` i `ICacheProvider`
- ADO.NET upravitelj konekcija – `IconnectionProvider`
- Upravitelj transakcija – `ITransactionFactory` i `ITransaction`
- ORM strategije – `IClassPersister`
- Strategije pristupa svojstvima – `IpropertyAccessor`
- Stvaranje posrednika - `IProxyFactory`



Slika 3 - NHibernate sučelja u slojevitoj arhitekturi

3.2.4 Implementacija modela domene

Kao što smo već rekli u poglavlju *1. Razvoj vođen domenom (engl. Domain – Driven Design)*, model domene je centralni dio aplikacije te programski kod ne bi smio ovisiti o drugim elementima aplikacije kao što su ulazno – izlazne operacije, operacije za rad sa bazom podataka itd. NHibernate omogućuje automatsku perzistenciju promjena na transparentan način u odnosu na model domene. Drugim riječima, perzistentni objekti, odnosno entiteti, nisu uopće svjesni da postoji NHibernate mehanizam za perzistenciju.

NHibernate ne zahtijeva da bilo koji razred modela domene naslijedi ili implementira bazne razrede ili sučelja. Iz tog razloga isti model domene je moguće koristiti u sustavima u kojima ne postoji NHibernate mehanizam perzistencije. Međutim, svako rješenje

automatske perzistencije nije potpuno transparentno, uključujući NHibernate, jer se nameću određeni zahtjevi na perzistentne razrede.

NHibernate zahtjeva da model domene bude implementiran koristeći POCO (*Plain Old CLR Object*) modelom programiranja. Pojednostavljeni rečeno POCO objekti su objekti implementirani isključivo .NET-om te nisu opterećeni međuvisnostima o vanjskim radnim okvirima. Par zahtjeva koje NHibernate postavlja na implementaciju modela domene su ujedno najbolje prakse za POCO model programiranja. Većina POCO objekta je kompatibilna sa NHibernateom bez bilo kakvih promjena. POCO objekti definiraju poslovne metode, koje definiraju ponašanje te svojstva koja predstavljaju stanje. Neka svojstva predstavljaju asocijacije na druge POCO objekte.

Za ostvarenje lijelog dohvaćanja podataka, odnosno *lazy fetching*, NHibernate stvara tzv. *proxy* objekte (objekti koji delegiraju pozive objektu kojeg predstavljaju). Da bi se delegirali pozivi metoda i svojstava preko proxy objekata potrebno se pridržavati nekoliko pravila prilikom dizajniranja razreda:

- Razred mora imati definiran defaultni, odnosno bezparametarski konstruktor
- Defaultni konstruktor ne smije biti privatni, odnosno mora imati najmanje *protected* modifikator pristupa.³
- Razred ne smije biti *sealed*.
- Sve javne metode i sva javna svojstva moraju biti virtualni (ključna riječ *virtual*).
- Ne smiju postojati javne članske varijable.

Svojstva koja predstavljaju kolekciju trebaju biti definirana preko sučelja kao što su primjerice `IList` ili `IDictionary`, a ne preko konkretne implementacije. NHibernate podržava sljedeće kolekcije:

Sve kolekcije mogu koristiti `ICollection` tip. Jedino `bag` i `set` se mogu koristiti u dvosmjernim vezama na strani koja je definirana kao `inverse="true"`.

Upravljanje asocijacijama prepušteno je programskom kodu unutar POCO objekata. Dakle, ukoliko želimo implementirati dvosmjernu asocijaciju, obje strane veze moraju implementirati adekvatan kod koji će to omogućiti.

³ NHibernate dozvoljava da defaultni konstruktor bude privatан samo za vrijednosne objekte, za entitete je potreban najmanje *protected* defaultni konstruktor.

Jedan od razloga zašto je popularno koristiti svojstva je zato što svojstva omogućavaju enkapsulaciju. Moguće je promijeniti skrivenu unutrašnju implementaciju, a da se javno sučelje ne promijeni. NHibernate koristi svojstva za obnavljanje stanja prilikom učitavanja objekta iz baze podataka. Ukoliko su svojstva implementirana tako da obavljaju validaciju, tada rekreiranje objekta neće biti moguće jer u trenutku validacije sva stanja možda neće biti postavljena. Iz tog razloga je moguće reći NHibernateu da prilikom rekonstrukcije stanja objekata koristi članske varijable.

NHibernate automatski detektira promjene objekata sa ciljem sinkronizacije promijenjenog stanja sa bazom podataka. Obično je sigurno vratiti drugačiji objekt preko `get` pristupnika svojstva, nego objekt koji je originalno bio postavljen preko `set` pristupnika svojstva. NHibernate uspoređuje objekte po njihovoj vrijednosti, a ne preko identiteta objekta, odnosno reference, da bi odredio koje stanje je potrebno ažurirati u bazi podataka. Međutim postoji važna iznimka, kolekcije se uspoređuju po identitetu, odnosno referenci. Za svojstvo koje je mapirano kao persistenata kolekcija, potrebno je vratiti točno istu instancu kolekcije, preko `get` pristupnika svojstva, koju je NHibernate postavio preko `set` pristupnika svojstva. Ukoliko to ne napravimo, NHibernate će ažurirati stanje u bazi podataka kada to zaista nije potrebno. Bitno je napomenuti da obično kolekcije ne bi trebale imati definirane `set` pristupnike. Ukoliko je to apsolutno potrebno, moguće je reći NHibernateu da koristi drugačiju strategiju pristupanja čitanju i postavljanju stanja svojstva kao što je direktni pristup članskoj varijabli (`field access`)⁴.

3.2.4.1 Identitet objekta

Iznimno je važno shvatiti razliku između identiteta objekta i jednakosti objekta, identiteta u bazi podataka te kako NHibernate upravlja identitetima. Identitet objekta, `object.ReferenceEquals()`, je pojam definiran na razini CLR-a, odnosno identitet objekta definira se pomoću reference. Dva objekta su jednaka ukoliko pokazuju na istu memoriju lokaciju. Dok s druge strane, jednakost objekata je pojam definiran na razini

⁴ NHibernate podržava tzv. `nosetter` strategiju pristupa podacima nekog objekta. `Nosetter` strategija specificira NHibernateu da čita podatke kroz `get` pristupnik svojstva, a da postavlja podatke preko članske varijable (npr. `set` pristupnik svojstva nije postavljen ili sadrži poslovnu logiku). Važno je napomenuti da korištenje `nosetter` strategije u slučaju kolekcija zahtjeva da `get` pristupnik svojstva vrati istu kolekciju koja je postavljena preko članske varijable.

razreda koji implementiraju metodu `Equals()` (ili operator `==`). Jednakost ili ekvivalentnost objekata znači da su dva različita objekta (imaju različiti identitet, odnosno reference) jednaka ukoliko su im vrijednosti jednakе.

Kada govorimo u kontekstu objektno – relacijske perzistencije tada objekt predstavlja reprezentaciju određenog retka tablice baze podataka u memoriji. Perzistencija uvodi neke komplikacije jer uz identitet objekta i jednakosti objekta uvodimo i identitet u bazi podataka.

NHibernate predstavlja aplikaciji identitet u bazi podataka na dva načina:

- Vrijednost svojstva koja predstavlja identitet u bazi podataka, odnosno primarni ključ n-torke u bazi podataka
- Vrijednost koju vraća metoda `ISession.GetIdentifier(object o)`

Teoretski svojstvo identiteta ne bismo trebali prikazati u modelu domene, jer se radi o problemu perzistencije podataka, a ne o poslovnom problemu. Međutim u praksi se identifikator prikazuje u modelu domene jer olakšava razvoj, kao što je npr. dohvati entiteta iz baze podataka preko identifikatora. Što se tiče NHibernatea uobičajeno je ne implementirati set pristupnik (NHibernate koristi .NET reflection za pristupanju privatnoj članskoj varijabli) jer primarni ključ generira baza podataka. Ukoliko se koriste prirodni ključevi tada je potrebno dozvoliti postavljanje identiteta, odnosno primarnog ključa. NHibernate ne dozvoljava promjenu vrijednosti identiteta nakon što je jedanput dodijeljena. Ukoliko želimo da svojstvo identifikatora nije vidljivo prema van možemo koristiti modifikatore pristupa `protected` ili `private`.

NHibernate podržava nekoliko strategija generiranja identifikatora objekta među kojima je najčešća native strategija, odnosno NHibernate prepusta generiranje identifikatora podupirućoj bazi podataka. NHibernate preporuča korištenje surogatnih primarnih ključeva, te izbjegavanje prirodnih koliko je to moguće zbog načina na koji radi NHibernate životni ciklus objekta. Također kompozitni ključevi su problematični te ih treba izbjegavati.

Moguće je da se dogodi da neka kolekcija sadrži dva objekta koji predstavljaju isti redak u tablici baze podataka, ali nemaju isti .NET identitet. Da bi se to spriječilo potrebno je

implementirati metode `Equals()` i `GetHashCode()`. Postoje tri načina implementacije metoda `Equals()` i `GetHashCode()`.

- **Uspoređivanje identiteta na temelju identiteta u bazi podataka** – Problem takvog pristupa je u tome što kada tek stvorimo objekt njemu nije dodijeljen nikakav identitet sve dok ne bude spremljen u bazu podataka. Ovo je osobito važno kod kolekcija skupa npr. `ISet`.
- **Uspoređivanje po vrijednosti** – Uspoređivanje svih svojstava perzistentnog razreda (ne uključuje kolekcije). Pri tome valja pripaziti da se samo radi operacije uspoređivanja ne koristi cijeli objektni graf. Problem kod ovakvog pristupa leži u činjenici da instance iz različitih sjednica nisu više jednake ukoliko se jedna promijeni, a dalje predstavljaju isti redak u bazi podataka.
- **Uspoređivanje po poslovnome ključu** – poslovni ključ je svojstvo ili kombinacija svojstava koji je jedinstven za svaku instancu sa jednakim identitetom u bazi podataka. Za razliku od primarnog ključa koji se nikada ne mijenja, poslovni ključ je moguće mijenjati ali to valja činiti rijetko osobito u skupovnim kolekcijama.

Kod izvedenih klasa nije dozvoljeno nadjačavanje `Equals()` i `GetHashCode()` metoda baznog razreda sa uključivanjem novih svojstava.

3.2.5 Definiranje meta podataka o mapiranju

ORM alati zahtijevaju meta podatke za specificiranje mapiranja između razreda i tablica, svojstva i stupaca, asocijacija i stranih ključeva, .NET tipova i SQL tipova. Takve informacije se nazivaju objektno – relacijski meta podaci o mapiranju. Oni definiraju transformacije između različitih podatkovnih sustava i njihove reprezentaciju njihovih veza.

Definiranje meta podataka u NHibernateu je moguće obaviti na dva načina: pomoću .NET atributa i pomoću XML datoteka mapiranja. U ovome radu ćemo opisati i koristiti metodu za definiranje meta podataka pomoću XML datoteka mapiranja iz razloga što smatramo da je takav pristup čišći i dugoročno fleksibilniji uz inicijalno više posla.

XML meta podatkovni format je čitljiv i definira korisne osnovne vrijednosti. Kada neke vrijednosti nisu definirane NHibernate koristi *reflection* nad mapiranim razredima kako bi odredio osnovne vrijednosti.

3.2.5.1 Osnovno mapiranje svojstava i razreda

Tipično NHibernate mapiranje .NET svojstva u atribut tablice baze podataka, definira:

- naziv svojstva (name)
- naziv atributa baze podataka (column)
- naziv NHibernate tipa (type)

Osnovna deklaracija omogućuje nekoliko varijacija mapiranja i optionalne postavke. Primjerice, često je moguće izostaviti naziv NHibernate tipa. Ukoliko svojstvo Description ima .NET tip String, NHibernate će koristiti NHibernate tip String bez eksplicitnog definiranja. NHibernate koristi reflection kako bi odredio tip svojstva. Sljedeća mapiranja su ekvivalentna.

```
<property name="Description" column="DESCRIPTION" type="String"/>

<property name="Description" column="DESCRIPTION" />
```

U prethodnom poglavlju vidjeli smo da je moguće izostaviti naziv atributa tablice baze podataka ukoliko naziv svojstva odgovara nazivu atributa u tablici. Ponekad je potrebno NHibernateu dati više podataka o atributu tablice osim samog imena. U tu se svrhu koristi <column> element XML dokumenta koji omogućuje veću fleksibilnost. U sljedećem primjer prikazuje ekvivalentna mapiranja:

```
<property name="Description" type="String">
    <column name="DESCRIPTION" />
</property>

<property name="Description" column="DESCRIPTION" type="String"/>
```

Određeni atributi <property> i <column> elementa koriste se radi automatskog generiranja sheme baze podataka. Posebice se preporuča korištenje atributa not-null koji omogućava NHibernateu da prijavi neispravnu null vrijednost svojstva.

```
<property name="InitialPrice" column="INITIAL_PRICE"
not-null="true" />
```

Vrijednost izvedenih svojstava izračunava se tijekom izvođenja izraza. Izraz se definira pomoću formula atributa. Primjerice:

```
<property name="TotalIncludingTax"  
formula="TOTAL + TAX_RATE * TOTAL" type="Double"/>
```

Tako definirana SQL formula se izvodi svaki put kada se entitet dohvata iz baze podataka. Dakle, baza podataka izvodi računanje umjesto .NET objekta. Takvo svojstvo nema atribut u tablici baze podataka i nikada se ne pojavljuje u SQL INSERT ili UPDATE naredbi.

access atribut elementa `<property>` specificira kako NHibernate treba pristupati vrijednostima entiteta. Osnovna strategija `<property>` elementa je korištenje `get` i `set` pristupnika svojstva. Stoga kad kod NHibernate učitava ili sprema objekt, on uvijek koristi definirane `get` i `set` pristupnike svojstva. Neki puta nije prigodno da NHibernate koristi `get` i `set` pristupnike pa se u tu svrhu koristi `field` strategija. Field strategija je korisna kada nismo definirali svojstva, kada nisu definirani `get` i `set` pristupnici ili kada `set` pristupnik sadrži kod za validaciju ili poslovnu logiku pa bi se moglo dogoditi da rekonstrukcija objekta ne uspije jer objekt nije u konzistentnom stanju pravo zbog procesa rekonstrukcije.

```
<property name="name" access="field"/>
```

Najbolja praksa preporučuje korištenje `get` i `set` pristupnika kad god je to moguće. Za problem kada `set` pristupnik sadrži poslovnu logiku moguće je definirati da NHibernate koristi `get` pristupnik za dohvat podataka, a za postavljanje podataka da koristi direktni pristup članskoj varijabli. To je tzv. `nosetter.*` strategija.

Pomoću atributa `insert` i `update` `<property>` elementa moguće je odrediti koja svojstva će se nalaziti u INSERT, a koja u UPDATE naredbi. Tako npr. vrijednost sljedećeg svojstva nikada neće biti upisana ili izmijenjena u bazi podataka:

```
<property name="Name" column="NAME" type="String"  
insert="false"  
update="false" />
```

Možemo reći da je svojstvo `Name` nepromjenjivo, odnosno moguće je pročitati vrijednost svojstva iz baze podataka, ali ga nije moguće nikako promijeniti. Ukoliko je kompletan

razred nepromijenjiv potrebno je postaviti `mutable="false"` u specifikaciji mapiranja razreda.

Dodatno, atributi `dynamic-insert` i `dynamic-update` omogućavaju NHibernateu da uključi ili ne uključi nemodificirane vrijednosti svojstva tijekom SQL `INSERT` i `UPDATE` naredbi. Navedeni atributi definiraju se u `<class>` elementu.

Odredene baze podataka podržavaju SQL sheme, odnosno grupiranje objekata baze podataka u smislene grupe. SQL shemu moguće je definirati na razini pojedinog razreda ili kompletног XML dokumenta mapiranja pomoću `schema` attributa.

Element `<property>` se nalazi unutar elementa `<class>` koji predstavlja razred kojeg se mapira u tablicu baze podataka. Da bi se razred uspješno mapirao potrebno je definirati vrijednost atributa `name`, u elementu `<class>`, odnosno potrebno je specificirati naziv razreda. Kod definiranja naziva razreda potrebno je navesti puno ime koje se sastoji od područja imena i naziva asemblerija kojemu razred pripada. Isto tako pomoću atributa `table` definira se naziv tablice koja će sadržavati objekte određenog tipa. Ukoliko se atribut `table` izostavi podrazumijeva se da se tablica u bazi podataka jednako zove kao i razred.

Za svaki razred kojeg želimo mapirati moramo specificirati njegov identifikator, odnosno kako se identifikator generira i kako se preslikava u bazi podataka. Identifikator u datoteci mapiranja je predstavljen preko `<id>` elementa koji ima standardne attribute kao što je naziv svojstva u objektu te naziv stupca u bazi podataka. Unutar samog `<id>` elementa definira se `<generator>` element koji određuje način generiranja identifikatora (npr. preko baze podataka, NHibernate ili vlastita implementacija).

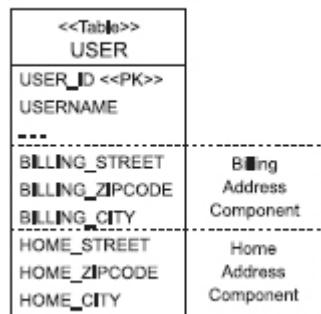
3.2.5.2 Mapiranje komponenti (fina granulacija)

Glavni cilj NHibernatea je omogućiti fino granulirane objektne modele koji su jedan od važnijih preduvjeta bogatog modela domene. U objektno orijentiranom modelu granulacija se postiže agregacijom/kompozicijom objekata. Takvim objektnim modelom postiže se bolja kohezija, veća iskoristivost koda i bolje shvaćanje modela.

Takov odnos DDD definira kroz entitete i vrijednosne objekte. Prije smo rekli da se entiteti mogu sastojati od vrijednosnih objekata, te da entitete perzistiramo jer su definirani svojim

identitetom, a vrijednosni objekti koji čine dio entiteta se perzistiraju implicitno samim entitetom dok se ostali vrijednosni objekti ne perzistiraju.

Sada se postavlja pitanje da li je za objekte koji su gradbeni elementi nekog drugog objekta nužno imati posebnu tablicu u bazi podataka ili sve podatke stavljati u istu tablicu? NHibernate za odnos kompozicije koristi jednu tablicu u bazi podataka, odnosno entitet i njegovi vrijednosni objekti su pohranjeni u istu tablicu.



Slika 4 - Tablica u bazi podataka sa podacima o entitetu i vrijednosnim objektima

Nekim jednostavnijim rječnikom mogli bismo reći da fina granulacija znači više razreda nego tablica u bazi podataka.

NHibernate za preslikavanje odnosa kompozicije među objektima koristi `<component>` XML element koji se nalazi unutar `<class>` elementa.

```
<component
    name="HomeAddress"
    class="Address">
    <property name="Street"
        type="String"
        column="HOME_STREET"
        not-null="true"/>
    <property name="City"
        type="String"
        column="HOME_CITY"
        not-null="true"/>
    <property name="Zipcode"
        type="Int16"
        column="HOME_ZIPCODE"
        not-null="true"/>
</component>
```

Slika 5 - NHibernate XML komponenta

Na Slika 5. je prikazan element `<component>` koji se sastoji od atributa `name` koji predstavlja naziv svojstva razreda koji sadrži komponentu, te atributa `class` koji specificira razred sadržanog objekta. Unutar elementa `<component>` nalaze se tipični elementi `<property>` koji predstavljaju svojstva određenog razreda.

Prethodno opisan način modelira kompoziciju jednosmjerno, odnosno moguće je doći do adrese jedino iz roditeljskog razreda, obrat ne vrijedi. Međutim NHibernate podržava jednosmjerne i dvosmjerne kompozicije. Dvosmjerne kompozicije nisu toliko česte niti korisne. Detalji o dvosmernim kompozicijama mogu se naći u NHibernate dokumentaciji.

Komponentu je moguće proglašiti *immutable* (nije moguće mijenjanje niti brisanje vrijednosti unutar objekta) sljedećom konfiguracijom u `<component>` elementu:

```
<component ... insert="false" update="false" />
```

3.2.5.3 Mapiranje asocijacija

Održavanje asocijacija između razreda i njihovih veza između tablica u bazi podataka je srž ORM-a. NHibernate podržava jednosmjerne i dvosmjerne asocijacije.

NHibernate inherentno podržava samo jednosmjerne asocijacije i to po samoj definiciji POCO modela. Da bi se podržala dvosmerna asocijacija potrebno je osigurati u svakome objektu odgovarajuće reference, ostalo je sve stvar specificirana u NHibernate XML datoteci.

NHibernate podržava sljedeće spojnosti veza:

- One to one
- One to many
- Many to one
- Many to many

Najjednostavnija veza je **One to one** veza koja povezuje dva entiteta. Postoje dva načina povezivanja, preko stranih ključeva i preko primarnog ključa. Kod veze pomoću stranih ključeva koristi `<many-to-one>` element na jednoj strani i `<one-to-one>` element na drugoj strani. Ograničenje koje nastaje iz ovakve veze je da ukoliko postoji dvije veze na objekt istog tipa, takva veza može biti samo jednosmjerna.

```
<many-to-one name="BillingAddress" class="Address" column="BAddId"
cascade="save-update" />

i

<one-to-one name="User" class="User" property-ref="BillingAddress"
/>
```

Za povezivanje preko primarnog ključa oba entiteta koriste isti primarni ključ koji je ujedno i strani. Kod takve veze postoji ograničenje da može postojati samo jedna veza na entitet određenog tipa upravo zbog dijeljenja primarnog ključa. Za takve veze konceptualno je moguć *lazy fetching* samo kada asocirani razred uvijek postoji što se definira preko atributa `constrained` (navедено je vrlo slično `not null` svojstvu).

```
<one-to-one name="Item" class="Item" constrained="true" />
```

Najčešće veze su **One to many** i **Many to one**. *One to many* veza u razredu se predstavlja preko kolekcija, odnosno svojstva tipa kolekcije. Slijedi primjer mapiranja u XML datoteci *One to many* veze (razred `Order` ima više stavki `OrderItem`):

```
<class name="Order" ...>
  <set name="Items" cascade="all-delete-orphan">
    <key column="OrderId" />
    <one-to-many class="OrderItem" />
  </set>
</class>
```

Many to one veza u razredu predstavljena je preko jednog objekta na kojeg se referencira roditeljski objekt. Prikazati ćemo XML mapiranje na prethodnom primjeru samo u suprotnome smjeru: Više objekata `OrderItem` pripada `Order` objektu.

```
<class name="OrderItem" ...>
  <many-to-one name="Order" column="OrderId" />
</class>
```

NHibernate omogućuje i dvosmjerne asocijacije te se u tu svrhu odlično uklapaju *One to many* i *Many to one* asocijacije. Sve što je potrebno napraviti je implementirati u razredima logiku sa postavljanje referenci u oba smjera. Isto tako potrebno je definirati XML

datoteke kao u prethodnim primjerima s tom razlikom što je u jednom od XML dokumenta, za navedenu asocijaciju, potrebno definirati `inverse` atribut. Dvosmjerna veza je u relacijskoj bazi predstavljena stranim ključevima, dok u objektnom modelu ona je predstavljena dvostrukim referencama. Kada ne bi definirali `inverse="true"` atribut NHibernate bi pokušao postaviti dva puta jedan te isti strani ključ, stoga pomoću `inverse` atributa određujemo za koju stranu će NHibernate kreirati upit koji će kontrolirati povezanost po stranome ključu. Najčešće najbolji način je omogućiti `<many-to-one>` strani tu kontrolu, jer generira manje SQL akcija. Tako XML dokumentu za `Order` dodajem `inverse` atribut:

```
<class name="Order" ...>
    <set name="Items" cascade="all-delete-orphan"
        inverse="true"
    >
        <key column="OrderId" />
        <one-to-many class="Orderitem" />
    </set>
</class>
```

Many to many asocijacije nisu toliko česte te ih se preporuča izbjegavati. Kod takvih se asocijacija obično koristi tablica veza (engl. *link table* ili *relationship table*) koja sadrži samo ključeve asociranih tablica. Tablica veza često puta zna dobiti dodatne atribute te postati pravi entitet, te se iz tog razloga preporuča izbjegavanje *Many to many* veza. Ta dodatna tablica se mora uzeti u obzir ukoliko se koristi *eager fetching* strategija. Svojstvo kolekcije se odnosi na tablicu veze te se dohvataju samo podaci iz tablice veza, pa u elementu `<many-to-many>` kolekcije treba specificirati strategiju vanjskog spajanja ukoliko želimo da se dohvate konkretni objekti druge relacije. Slijedi primjer XML mapiranja gdje neki predmet može imati više kategorija, te kategorija ima više predmeta.

```
<set name="Items" outer-join="true" table="Cat-Item" >
    <key column="CategoryId" />
    <many-to-many column="ItemId" outer-join="true" class="Item"
    />
</set>
```

NHibernate omogućava definiranje kaskadnog stila za svaku asocijaciju.

Tabela 1 - Kaskadni stilovi asocijacija

Kaskadni stil	Opis kaskadnog stila
cascade="none"	NHibernate ignorira asocijaciju
cascade="save-update"	NHibernate pretražuje po asocijacijama kada je transakcija uspješno završena i kada je objekt spremlijen pomoću <code>Save()</code> ili <code>Update()</code> naredbe te spremi novo instancirane tranzientne instance.
cascade="delete"	NHibernate pretražuje po asocijacijama i briše perzistentne asocijacije kada je objekt predan <code>Delete()</code> metodi.
cascade="all"	NHibernate kaskada save-update i delete.
cascade="all-delete-orphan"	Isto ako <code>cascade="all"</code> uz dodatno brisanje perzistentne instance koja je obrisana iz asocijacije (npr. brisanjem entiteta iz kolekcije).
cascade="delete-orphan"	Isto ako <code>cascade="all"</code> uz dodatno brisanje perzistentne instance koja je obrisana iz asocijacije (npr. brisanjem entiteta iz kolekcije).

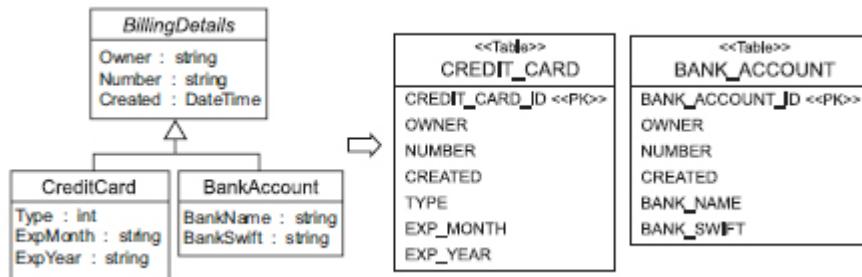
3.2.5.4 Mapiranje nasljeđivanja razreda

Nasljeđivanje je jedna od najznačajnijih razlika u neusklađenosti paradigmi objektno – orijentiranog i relacijskog svijeta. Sustavi sa objektno – orijentiranim modelom imaju „is a“ i „has a“ veze, dok relacijski sustavi imaju samo „has a“ vezu. NHibernate podržava tri pristupa za specificiranje hijerarhije nasljeđivanja:

- **Tablica za svaki konkretan razred** (engl. *Table per concrete class*) – odbacuju se veze nasljeđivanja i polimorfizma iz relacijskog modela.
- **Tablica za svaku hijerarhiju nasljeđivanja** (engl. *Table per class hierarchy*) – omogućava polimorfizam denormalizacijom relacijskog modela koristeći posebni stupac za čuvanje informacija o tipu.

- **Tablica za svaki izvedeni razred** (engl. *Table per subclass*) – predstavlja „is a“ vezu preko „has a“ veze.

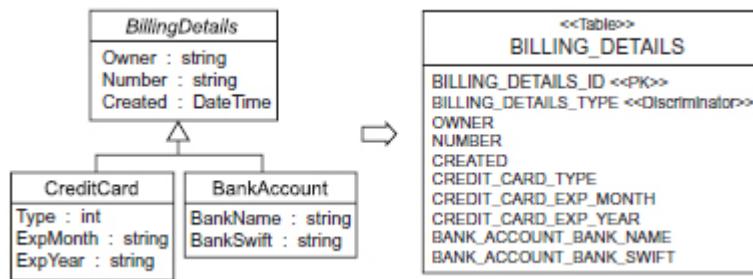
Za pristup „**Tablica za svaki konkretan razred**“ koristi se točno jedna tablica za svaki (ne apstraktni) razred. Sva svojstva razreda, uključujući naslijedena svojstva, se mapiraju u stupce tablice, kao što je prikazano na sljedećoj slici.



Slika 6 – Strategija „Tablica za svaki konkretan razred“

Glavni problem takvog pristupa je što on dobro ne podržava polimorfne asocijacije i polimorfne upite (izvršavanje nekoliko SELECT upita – jedan za svaki konkretan razred) te ukoliko promijenimo nešto u baznom razredu, takvu promjenu treba obaviti i u svim tablicama koje predstavljaju izvedene razrede. Ovakva strategija mapiranja ne zahtjeva dodatno specificiranje u XML datoteci: za svaki konkretan razred potrebno je navesti tablicu u koju se preslikava.

Strategija „**Tablica za svaku hijerarhiju nasljeđivanja**“ omogućava mapiranje hijerarhije nasljeđivanja u samo jednu tablicu. Takva tablica sadrži stupce za sva svojstva od svih razreda u hijerarhiji (bazni razred i svi izvedeni razredi). Konkretan podrazred je predstavljen posebnim *type-discriminator* stupcem koji sadrži informaciju o kojem se razredu radi (bilo bazni razred ili izvedeni). Ovakav način mapiranja je najbolja strategija što se tiče performansi i jednostavnosti, odnosno podržava polimorfne i nepolimorfne upite, kao i ad-hoc reporting upite sa zadovoljavajućim performansama. Glavni problem takvog pristupa je što svi stupci tablice moraju biti definirani da podržavaju NULL, jer bez NOT NULL ograničenja postoji problem integriteta podataka.



Slika 7 – Strategija „Tablica za svaku hijerarhiju nasljeđivanja“

Za mapiranje takve strategije u XML datoteci koristi se `<subclass>` element:

```

<hibernate-mapping>
    <class
        name="BillingDetails"
        table="BILLING_DETAILS" discriminator-value="BD">
        <id
            name="Id"
            column="BILLING_DETAILS_ID"
            type="Int64">
            <generator class="native"/>
        </id>
        <discriminator
            column="BILLING_DETAILS_TYPE"
            type="String"/>
        <property
            name="Name"
            column="OWNER"
            type="String"/>
        ...
        <subclass
            name="CreditCard"
            discriminator-value="CC">
            <property
                name="Type"
                column="CREDIT_CARD_TYPE"/>
            ...
        </subclass>
        ...
    </class>
</hibernate-mapping>

```

1 Bazni razred mapiran na tablicu
2 Discriminator stupac
3 Mapirana svojstva baznog razreda
4 Mapiranje izvedenog razreda CreditCard

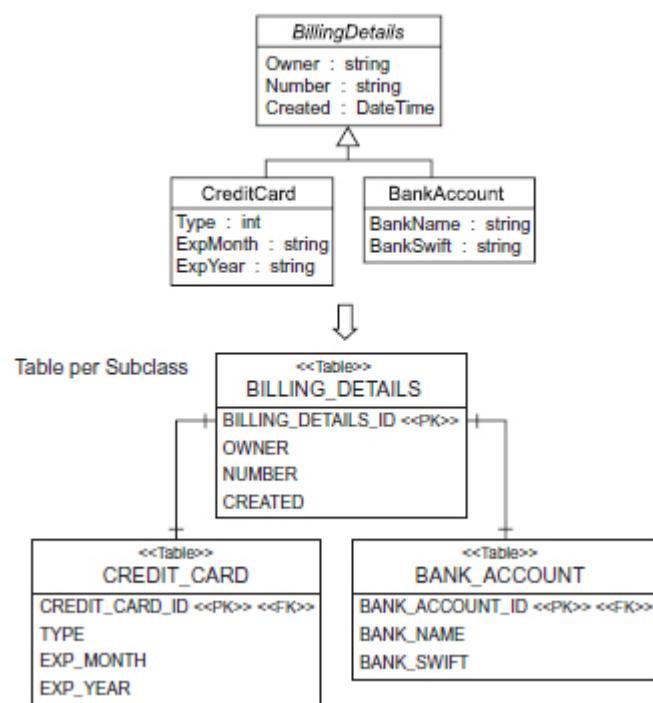
Slika 8 - Mapiranje strategije „Tablica za svaku hijerarhiju nasljeđivanja“

gdje su:

1. Bazni razred hijerarhije nasljeđivanja. Atribut `discriminator-value` sadrži vrijednost za razlikovanje od ostalih razreda u hijerarhiji nasljeđivanja.
2. Naziv i tip posebnog stupaca (*discriminator*) u tablici za razlikovanje razreda u hijerarhiji nasljeđivanja.

3. Mapirana svojstva baznog razreda.
4. Svaki izvedeni razred ima svoj vlastiti _{class} element koji sadrži definirana svojstva koja se preslikavaju u određene stupce tablice.

Posljednja strategija „**Tablica za svaki izvedeni razred**“ omogućava ostvarenje nasljeđivanja kroz relacijske veze stranog ključa. Svaki izvedeni razred koji definira perzistentna svojstva, uključujući apstraktne razrede i čak sučelja, ima svoju vlastitu tablicu. Za razliku od strategije koja koristi tablicu za svaki konkretni razred, ova tablica sadrži stupce za svako nenaslijedeno svojstvo izvedenog razreda, zajedno sa primarnim ključem, koji je ujedno i primarni ključ baznog razreda. Takav pristup prikazan je na sljedećoj slici.



Slika 9 - Strategija „Tablica za svaki izvedeni razred“

Dohvaćanje izvedenog razreda iz baze podataka omogućeno je spajanjem tablice izvedenog razreda sa tablicom baznog razreda. Glavna prednost ovakve strategije je to što je relacijski model potpuno normaliziran te su podržane sve značajke integritetskog ograničenja. U tu svrhu koristi se <joined-subclass> element XML dokumenta.

```

<?xml version="1.0"?>
<hibernate-mapping>
    <class
        name="BillingDetails"
        table="BILLING_DETAILS">
        <id
            name="Id"
            column="BILLING_DETAILS_ID"
            type="Int64">
            <generator class="native"/>
        </id>
        <property
            name="Owner"
            column="OWNER"
            type="String"/>
        ...
        <joined-subclass
            name="CreditCard"
            table="CREDIT_CARD">
            <key column="CREDIT_CARD_ID">
                <property
                    name="Type"
                    column="TYPE"/>
            ...
        </joined-subclass>
        ...
    </class>
</hibernate-mapping>

```

Slika 10 - Mapiranje strategije „Tablica za svaki izvedeni razred“

gdje su:

1. Bazni razred hijerarhije nasljeđivanja (nije potreban *discriminator*).
2. Element `<joined-subclass>` koji mapira izvedeni razred u novu tablicu. Sva svojstva definirana unutar tog elementa se mapiraju u tu tablicu.
3. Primarni ključ „CreditCard“ razreda te je ujedno i strani ključ za bazni razred.

Za ispravan odabir odgovarajuće strategije potrebno se pridržavati sljedećih uputa:

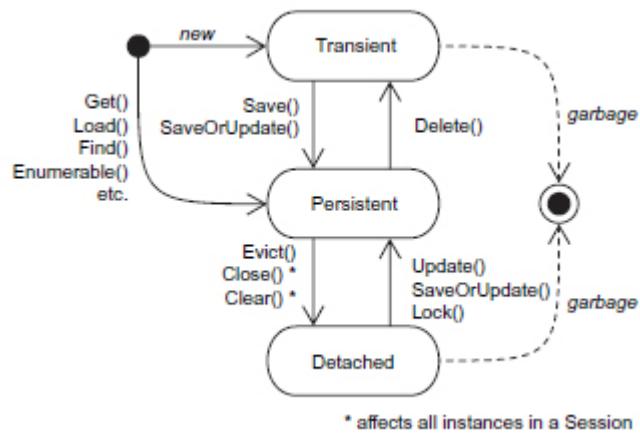
- Ukoliko nam nisu bitni polimorfne asocijacije i upiti koristiti strategiju „Tablica za svaki konkretni razred“.
- Ukoliko su potrebne polimorfne asocijacije i upiti, a izvedeni razredi definiraju relativno malo svojstava koristiti strategiju „Tablica za svaku hijerarhiju nasljeđivanja“. Koristi se kod jednostavnijih problema.
- Ukoliko su potrebne polimorfne asocijacije i upiti, a izvedeni razredi definiraju mnogo svojstava koristiti strategiju „Tablica za svaki izvedeni razred“. Koristi se kod komplikiranijih problema.

3.2.6 Životni ciklus NHibernate objekata

Aplikacija sa perzistentnim stanjem mora komunicirati sa slojem perzistencije za prijenos stanja u memoriji u bazu podataka i obratno. Da bismo to postigli potrebno je koristi NHibernate API. Kada tako komuniciramo sa slojem perzistencije, aplikacija se mora pobrinuti za stanje i životni ciklus objekata u procesu perzistencije.

Životni ciklus NHibernate objekata sastoji se od tri stanja:

- **Tranzijentni objekti** (engl. *transient*) – objekti stvoreni preko operatora new nisu odmah perzistentni, odnosno nisu povezani sa retkom tablice baze podataka. Ukoliko ih ne spremimo u bazu podataka nakon nekog vremena biti će obrisani od strane *garbage collectiona*. Perzistentni postaju eksplicitnim pozivanjem metode `Save()` ili predajući njihovu referencu nekom drugom perzistentnom objektu.
- **Perzistentni objekti** (engl. *persitent*) – su objekti koji su povezani sa bazom podataka, odnosno za koje postoji reprezentacija u bazi podataka. To su oni objekti koji su ili dohvaćeni iz baze podataka ili su iz tranzijentnog prešli u perzistentno stranje. Perzistentni objekti su uvijek vezani uz `ISession` i njihovo stanje se sinkronizira sa bazom podataka sa završetkom transakcije. Perzistentan objek je moguće učiniti tranzijentnim pomoću naredbe `Delete()`.
- **Odvojeni objekti** (engl. *detached*) – po završetku transakcije objekti postaju odvojeni objekti tako što su izgubili vezu sa upraviteljem perzistencije (`ISession`), odnosno sa zatvaranjem `ISessiona`. Time se više ne jamči sinkroniziranost trenutnog stanja sa stanjem u bazi podataka. Međutim NHibernate omogućava stvaranje nove veze sa novim upraviteljem perzistencije (metode `Update()` i `Lock()`) čime ti objekti prelaze opet u perzistentno stanje. Ponovno uspostavljanje veze između odvojenih objekata i baze obavlja se selektivno jer u većini slučajeva ne želimo uspostaviti vezu za cijelo objektno stablo.



Slika 11 - Životni ciklus NHibernate objekata

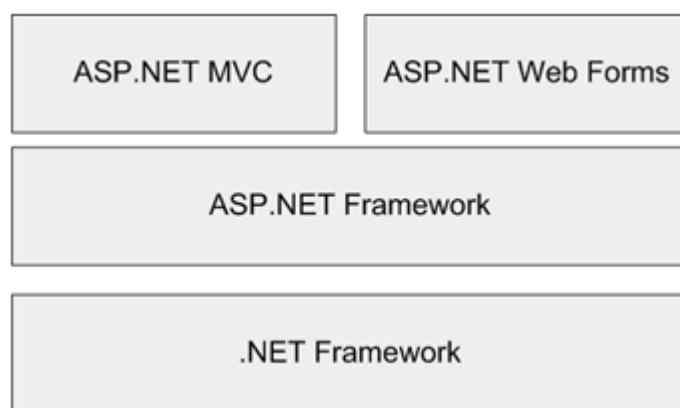
NHibernate posjeduje upravitelja perzistencije `ISession` (u danjem tekstu sjednica). Sjednica se stvara na početku svake jedinice posla (engl. *unit of work*) koristeći tvornicu sjednice te traje do kraja jedinice posla. Stvaranje sjednica je jeftino, dok je stvaranje tvornice sjednica skupo i najčešće se stvara jednom pri pokretanju aplikacije. Upravitelj perzistencije jamči da za vrijeme sjednice uvijek postoji točno jedan objekt koji predstavlja neki redak u tablici baze podataka.

4. ASP.NET MVC

ASP.NET razvojna okolina je dio .NET razvojne okoline te čini Microsoftovu razvoju okolinu za razvoj web aplikacija. Sadrži skup razreda koji su napravljeni isključivo za razvoj web aplikacija. Primjerice ASP.NET razvojna okolina sadrži razrede za implementaciju cachiranja web, autentifikaciju i autorizaciju.

Microsoft ima dvije razvojne okoline za razvoj web aplikacija nad ASP.NET razvojnom okolinom:

- ASP.NET Web Forms i
- ASP.NET MVC



Slika 12 - ASP.NET razvojne okoline

Microsoft ASP.NET MVC je najnovija Microsoft razvojna okolina za razvoj web aplikacija. ASP.NET MVC podržava razvoj programske potpore korištenjem oblikovnih obrazaca u programiranju (*engl. pattern – based software*). Web aplikacije napisane pomoću ASP.NET MVC-a pogodne su za testiranje *unit* testovima što čini ASP.NET MVC odličnom razvojnom okolinom kada se prakticira testom vođeno razvijanje (*engl. test – driven development*).

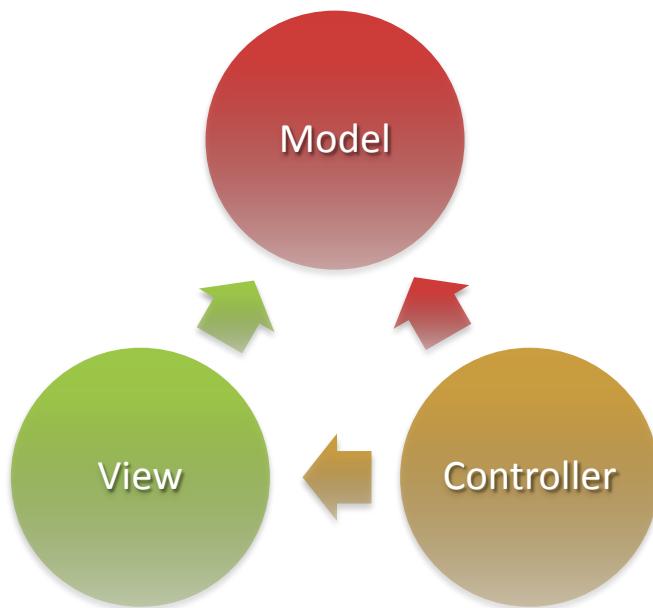
4.1 Arhitektura ASP.NET MVC aplikacije

MVC aplikacija, odnosno *Model View Controller* aplikacija, podijeljena je u tri dijela:

- **Model** – MVC model sadrži svu aplikacijsku logiku koja nije sadržana u pogledu (*engl. View*) ili nadgledniku (*engl. Controller*). To uključuje svu aplikacijsku

validacijsku logiku, poslovnu logiku i logiku pristupa bazi podataka. MVC Model sadrži model razrede koji se koriste za modeliranje objekata u domeni aplikacije.

- **View** – MVC pogled sadrži HTML oznake i logiku pogleda (*engl. view logic*). Skup pogleda čini izgled web aplikacije. ASP.NET MVC pogledi su odgovorni za prikaz HTML stranica koje će korisnici moći vidjeti kada posjete web sjedište.
- **Controller** – MVC nadglednik sadrži logiku toka i upravljanja. MVC nadglednik međusobno djeluje sa MVC modelom i pogledom za upravljanje toka izvođenja aplikacije. Kada Internet preglednik napravi zahtjev na poslužitelj sa ASP.NET MVC aplikijom, nadglednik je odgovoran za slanje odgovora za taj zahtjev. Nadglednici sadrže jednu ili više akcija. Akcija nadglednika može Internet pregledniku vratiti različite tipove rezultata izvođenja akcija. Primjerice, akcija nadglednika može vratiti pogled, datoteku ili preusmjeriti zahtjev na neku drugu akciju nadglednika.



Slika 13 - MVC struktura

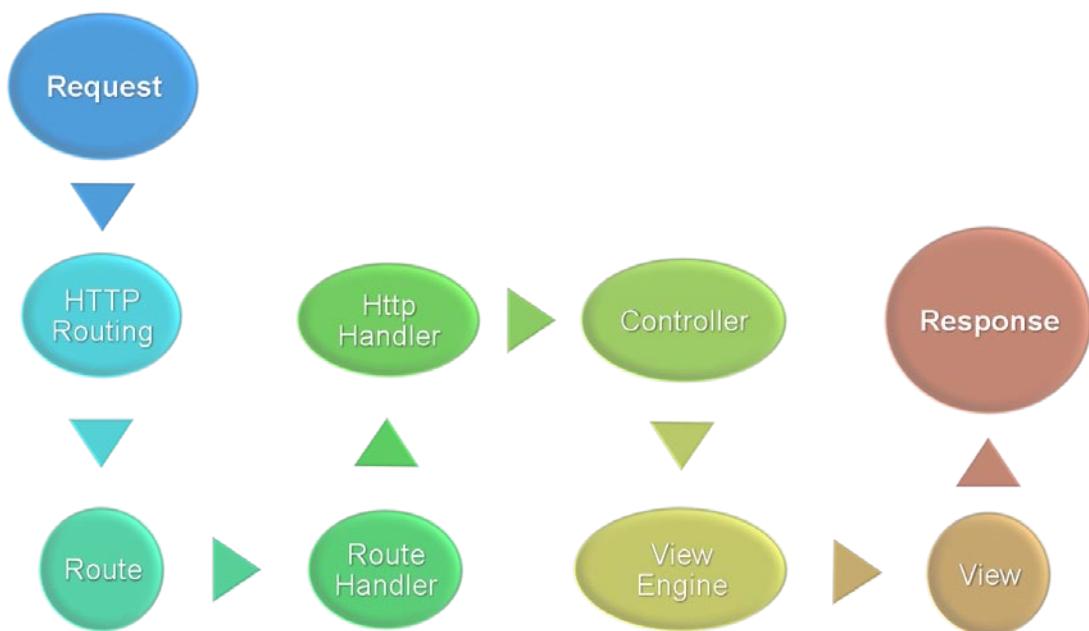
Provođenje ovakvog pristupa preko modela, pogleda i kontrolera se pokazalo kao vrlo koristan način strukturiranja web aplikacije.

Strogo odvajanje pogleda od ostatka web aplikacije omogućuje preoblikovanje izgleda aplikacije bez diranja jezgre aplikacije. Dizajner web stranice može modificirati poglede neovisno od programskih inženjera koji izrađuju poslovnu logiku i logiku pristupa

podacima. Ljudi sa različitim vještinama i ulogama mogu modificirati različite dijelove aplikacije bez direktnog utjecaja jedni na druge. Isto tako, odvajanje pogleda od ostatka web aplikacije omogućava jednostavnu promjenu tehnologije prikaza u budućnosti.

Odvajanjem upravljačke logike od ostatka aplikacijske logike se pokazao kao jedan od korisnih obrazaca za razvoj web aplikacija. Često morate promijeniti način na koji korisnik komunicira sa aplikacijom, a jednostavno ne želite dirati poglede ili modele prilikom mijenjanja toka izvođenja vaše aplikacije.

Slijedećom slikom prikazan je tok izvođenja MVC aplikacije.



Slika 14 - tok izvođenja MVC aplikacije

5. Strukturirana analiza i specifikacija informacijskog sustava za upravljanje 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.

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

U posljednjim potpoglavlјima definirati će se poslovni procesi koje će informacijski sustav moći provoditi na temelju definiranih zahtjeva.

5.1 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 zalogoprincu. 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. Svaki vlasnik posebnoga dijela plaća pričuvu mjesечно. Odluku o visini pričuve po jednom četvornom metru donose suvlasnici, a ona ne može biti manja od $1,53 \text{ kn/m}^2$, što je zakonski minimum. Visina pričuve se određuje za svaki stan posebno prema ukupnoj površini stana. Naplatu pričuve organizira i pričuvu prikuplja upravitelj. Sredstva zajedničke pričuve koriste se: za redovno održavanje i poboljšanje zajedničkih dijelova i uređaja zgrade, za hitne i nužne popravke. Važno je napomenuti da pričuva nije

Strukturirana analiza i specifikacija informacijskog sustava za upravljanje stambenim zgradama

upraviteljeva, ne nalazi se na računu upravitelja i ne može se koristiti bez potpisa predstavnika suvlasnika.

Vlasnici zgrade su ili pojedinačni vlasnici ili suvlasnici nekretnine. 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.

Suvlasnici odlučuju o svim pitanjima koja se tiču zajedničkih dijelova i uređaja zgrade. Suvlasnici upravljaju zgradom donošenjem odluka o poduzimanju redovnih i izvanrednih poslova upravljanja.

Svi poslovi koje suvlasnici poduzimaju na zgradi imaju karakter redovne i izvanredne uprave. O redovnoj upravi suvlasnici odlučuju većinom glasova (kada se za njih izjasne suvlasnici koji zajedno imaju većinu suvlasničkih dijelova), 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. Izvanrednom upravom koja premašuje okvir redovne uprave smatraju se promjena namjene, dogradnja, nadogradnja, poslovi preuređenja, davanje u zakup ili najam na dulje vrijeme.

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 održavanja zajedničkih dijelova i uređaja u graditeljskom i funkcionalnom stanju, organizacija naplate sredstava zajedničke pričuve.

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. Predstavnik suvlasnika dužan je ovjeravati radne naloge kao potvrde izvršenja određenog posla

5.2 Specifikacija zahtjeva sustava

5.2.1 Poslovni zahtjevi

Poslovni zahtjevi predstavljaju ciljeve organizacije ili korisničke zahtjeve na višoj razini, opis problema koje treba riješiti. U nastavku slijede poslovni zahtjevi za upravljanje stambenim objektima:

- 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, suvlasnicima, zajedničkim i osobnim prostorima (stanovi, 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.
- Sustav mora okupljati na jednome mjestu više različitih tvrtki upravitelja te tvrtki izvođača radova.
- Informacijski sustav treba omogućiti istovremeni rad velikog broja korisnika.
- Informacijski sustav treba biti dostupan s bilo kojeg računala spojenog na Internet.
- Pristup informacijskom sustavu treba biti dostupan suvlasnicima, predstavniku suvlasnika, upravitelju, izvođačima radova i ostalim relevantnim korisnicima sustava sa definiranim razinama autorizacije.
- Sigurnost i privatnost podataka o pojedinim stambenim objektima, njihovim upraviteljima i stanarima.
- Omogućiti dionicima uvid u njihove podatke

5.2.2 Korisnički zahtjevi

Korisnički zahtjevi opisuju zadatke koje korisnici moraju moći obaviti služeći se informacijskim sustavom. Sadržani su u opisima slučajeva korištenja. U nastavku slijede korisnički zahtjevi za upravljanje stambenim objektima:

- Brži i jednostavniji pristup uslugama potrebnim za upravljanje stambenim zgradama od strane svih dionika sustava (svlasnici, predstavnik svlasnika, upravitelj, izvođači radova).
- Informacijski sustav mora biti jednostavan za korištenje.
- Informacijski sustav treba omogućiti svlasnicima jednostavno podnošenje prijave kvarova na zgradi, te predstavniku svlasnika, upravitelju i njegovim kooperantima obradu kvarova.
- Sustav treba omogućiti svim svlasnicima, predstavniku svlasnika, upravitelju i njegovim kooperantima jednostavan prikaz stanja prijave kvara, odnosno obrade prijavljenog kvara.
- Informacijski sustav treba omogućiti upravitelju i izvođačima radova kreiranje naloga za naplatu troškova zgrade, te delegiranje naloga predstavniku svlasnika 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.

5.2.3 Funkcionalni zahtjevi

Funkcionalni zahtjevi definiraju funkcionalnost (očekivano ponašanje i operacije koje sustav može izvoditi) programske podrške, odnosno informacijskog sustava, koju treba ugraditi u proizvod da bi se omogućilo korisnicima obavljanje njihovih zadataka. U nastavku slijede funkcionalni zahtjevi za upravljanje stambenim objektima:

- Sustav treba evidentirati za zgradu
 - Katastarska općinu i česticu
 - Pripadajuće zemljiste
 - Adresu zgrade
 - Ukupnu površinu zgrade

- Stanove
- Zajedničke prostorije
- Sustav treba evidentirati sve prostore zgrade poput stanova, poslovnih prostora i zajedničkih prostorija.
 - Za svaki prostor potrebno je znati:
 - Površinu prostora
 - Vlasništvo
- Sustav treba omogućiti evidenciju svih kvarova, od prijave do rješavanja kvara.
- Sustav treba omogućiti evidenciju računa za izvršene radove na zgradama.
- Evidencija prihoda i rashoda zgrade.
- Sustav treba omogućiti slanje obavijesti putem emaila. To obuhvaća prijavu kvarova, svaku promjenu stanja kvarova, itd.
- Sustav treba evidentirati upraviteljeve ovlaštene izvođače radova. Omogućiti unos novih i ažuriranje postojećih kooperanata.
- Kooperante/Izvođače radova treba kategorizirati prema djelatnosti koju obavljaju.

5.2.4 Nefunkcionalni zahtjevi

Nefunkcionalni zahtjevi predstavljaju standarde, pravila i ugovore kojih se proizvod mora pridržavati npr. Zahtjevi na performanse, ograničenja na dizajn i implementaciju, te svojstva kvalitete. U nastavku slijede nefunkcionalni zahtjevi za upravljanje stambenim objektima:

- Informacijski sustav je potrebno izraditi kao web aplikaciju.
- Sustav treba biti izrađen pomoću programskog jezika C# i Microsoft SQL Server baze podataka.
- Osigurati persistenciju podataka.

5.2.5 Poslovna pravila

Poslovna pravila su operativni principi poslovnih procesa. Oni nisu funkcionalni zahtjevi. U nastavku slijede poslovna pravila za upravljanje stambenim objektima:

- 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

Strukturirana analiza i specifikacija informacijskog sustava za upravljanje stambenim zgradama

prijedlog upravitelja suvlasnici mogu povećati visinu cijene u kunama po kvadratu ($> 1.53 \text{ kn/m}^2$).

- Za poslove koje je potrebno obaviti za zgradu, a koji imaju karakter redovne uprave⁵ 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 uprave⁶ potrebna je suglasnost svih suvlasnika.
- Predstavnik suvlasnika sve prijave kvarova mora odobriti kako bi ih upravitelj mogao početi obrađivati.
 - Izuzetak su hitni kvarovi, odnosno hitne intervencije kada upravitelj reagira neovisno o potvrdi predstavnika 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.
- Pričuva se plača mjesečno.
- Pričuvu plača 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.

⁵ Poslovima redovne uprave smatraju se poslovi redovitog održavanja zajedničkih dijelova i uređaja nekretnine, te građevinski zahvati radi održavanja, zatim stvaranje zajedničke pričuve itd.

⁶ 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.

5.3 Slučajevi korištenja

Zahtjevi koji se postavljaju nad informacijski sustav za upravljanje stambenim zgradama ukomponirati ćemo u slučajeve korištenja koji će dati opise koraka ili akcija između korisnika i informacijskog sustava čime ćemo postići bolju predodžbu poslovnih procesa. Poslovni slučaj korištenja opisan je ne tehničkom terminologijom koja informacijski sustav tretira kao crnu kutiju i opisuje poslovne procese koje provode poslovni sudionici (ljudi ili sustavi) za postizanje određenih ciljeva. Takav slučaj korištenja opisuje proces koji pruža vrijednost poslovnome sudioniku te opisuje što proces zaista radi.

U nastavku opisati ćemo samo glavne slučajeve korištenja:

1. Registracija upravitelja ili izvođača radova u sustav
2. Registracija suvlasnika u sustav
3. Kreiranje stambene zgrade
4. Kreiranje zemljišne knjige
5. Dodavanje etaža u zemljišnu knjigu
6. Kreiranje glasovanja – poslovi redovne i izvanredne uprave
7. Glasovanje suvlasnika za poslove redovne i izvanredne uprave
8. Prijava kvara
9. Revizija kvara
10. Obrada kvara
11. Potvrđivanje popravka
12. Izdavanje računa zgradi za obavljene popravke
13. Plaćanje troškova zgrade
14. Izdavanje računa za pričuvu

5.3.1 Registracija upravitelja ili izvođača radova u sustav

Primarni sudionik: Upravitelj i izvođači radova (pravna osoba)

Dionici: upravitelj, izvođači radova, suvlasnici

Preduvjeti: Registracija je moguća samo za pravne osobe koje su ili upravitelj ili izvođač radova.

Uvjeti po završetku: Uspješno registriran upravitelj ili izvođač radova.

Strukturirana analiza i specifikacija informacijskog sustava za upravljanje

stambenim zgradama

Glavni uspješni scenarij:

1. Pravna osoba pristupa početnoj web stranici.
2. Pravna osoba odabire odgovarajuću poveznicu na stranicu za registraciju u ovisnosti da li se radi o upravitelju ili izvođaču radova.
3. Sukladno odabranoj stranici za registraciju upravitelja ili izvođača radova prikazuju se specifična polja pravne osobe (naziv pravne osobe, OIB, usluge) i standardna polja za korisnika (korisničko ime, lozinka, e-mail) koja je potrebno ispravno ispuniti.
4. Nakon ispravnog popunjavanja forme za registraciju, pravna osoba šalje zahtjev sa podacima.
5. Pravna osoba je uspješno registrirana.
6. Pravna osoba je odmah prijavljena u sustav.

Alternativni scenarij:

3a. Pravna osoba je već registrirana i njezini podaci postoje u sustavu.

3a1. Na temelju upisanog OIB-a provjerava se da li već postoji osoba sa istim OIB-om. Ukoliko postoji korisnik je upozoren porukom na ekranu da osoba već postoji. Korisnik se može ili prijaviti sa korisničkim imenom te osobe ili može unijeti novi OIB i sa njime nove podatke.

5.3.2 Registracija suvlasnika u sustav

Primarni sudionik: Suvlasnik (pravna ili fizička osoba)

Dionici: upravitelj, suvlasnici

Preduvjeti: Registracija je moguća samo za suvlasnike koji mogu biti pravne ili fizičke osobe.

Uvjeti po završetku: Uspješno registriran suvlasnik sa mogućim dodijeljenim stanom. Ukoliko određena etaža nije asocirana sa fizičkom ili pravnom osobom suvlasnik je u određenom vremenu definiran isključivo u kontekstu korisnika, odnosno nema stanova.

Glavni uspješni scenarij:

1. Osoba pristupa početnoj web stranici.
2. Osoba odabire poveznicu za registraciju suvlasnika.
3. Prikazuje se web forma za registraciju suvlasnika koja se sastoji od dijela sa korisničkim podacima i djela o podacima o osobi.
4. Nakon ispravnog popunjavanja forme za registraciju, osoba šalje zahtjev sa podacima.
5. Suvlasnik je uspješno registriran.
6. Osoba je odmah prijavljena u sustav kao suvlasnik.

Alternativni scenarij:

3a. Osoba je već registrirana i njezini podaci postoje u sustavu.

3a1. Na temelju upisanog OIB-a provjerava se da li već postoji osoba sa istim OIB-om. Ukoliko postoji korisnik je upozoren porukom na ekranu da osoba već postoji. Korisnik se može ili prijaviti sa korisničkim imenom te osobe ili može unijeti novi OIB i sa njime nove podatke.

5.3.3 Kreiranje stambene zgrade

Primarni sudionik: Upravitelj

Dionici: upravitelj, suvlasnici

Preduvjeti: Uspješno prijavljen upravitelj.

Uvjjeti po završetku: Uspješno kreirana zgrada.

Glavni uspješni scenarij:

1. Upravitelj pristupa stranici za kreiranje stambenih zgrada.
2. Upravitelj unosi adresu zgrade (ulica i grad).
3. Sprema podatke.
4. Temeljem podataka kreira se zgrada.

5.3.4 Kreiranje zemljišne knjige

Primarni sudionik: upravitelj

Dionici: upravitelj, suvlasnici

Strukturirana analiza i specifikacija informacijskog sustava za upravljanje

stambenim zgradama

Preduvjeti: Uspješno prijavljen upravitelj na sustav. Kreirana zgrada od strane prijavljenog upravitelja. Upisani podaci o gradovima. Upisani podaci o katastarskim uredima.

Uvjeti po završetku: Stvorena otključana zemljišna knjiga koja je dodijeljena zgradi, te je omogućeno dodavanje etaža. Dok je zemljišna knjiga otključana omogućeno je dodavanje etaža.

Glavni uspješni scenarij:

1. Upravitelj pristupa stranici sa stvorenom zgradom.
2. Upravitelj odabire poveznicu za stvaranje zemljišne knjige za zadanu zgradu.
3. Upravitelj popunjava podatke za zemljišnu knjigu.
4. Sprema upisane podatke o zemljišnoj knjizi.
5. Uspješno stvorena zemljišna knjiga za zadanu zgradu.

Alternativni scenarij:

4a. Upisani podaci o zemljišnoj knjizi nisu ispravni ili nisu potpuni.

4a1. Korisnik je unio nepotpune ili neispravne podatke o zemljišnoj knjizi. Iz tih razloga korisniku se prikazuju odgovarajuće poruke upozorenja i mogućnost da ispravi nedostatke čime se korisnik nalazi na trećoj točki.

5.3.5 Dodavanje etaža u zemljišnu knjigu

Primarni sudionik: upravitelj

Dionici: upravitelj, suvlasnici

Preduvjeti: Uspješno prijavljen upravitelj na sustav. Kreirana otključana zemljišna knjiga. Do zemljišne knjige moguće je doći isključivo preko stambene zgrade.

Uvjeti po završetku: Upisane su sve etaže čime je zemljišna knjiga zaključana, odnosno nije moguć danji unos etaža. Definirani su vlasnici etaža i kao osobe spremljene u sustav.

Glavni uspješni scenarij:

1. Upravitelj pristupa stranici zemljišne knjige preko stranice stambene zgrade.

2. Upravitelj odabire opciju za unos nove vlasničke ili zajedničke etaže u zemljišnu knjigu.
3. U ovisnosti o izabranoj vrsti etaže upravitelju se prikazuju odgovarajuća polja za unos podataka o etaži.
4. Upravitelj unosi odgovarajuće podatke o etaži.
5. Upravitelj sprema podatke o etaži.
6. Etaža je upisana u zemljišnu knjigu.
7. Upravitelj ponavlja korake od 1 – 6 dok ne unese sve etaže stambene zgrade.
8. Upravitelj zaključava zemljišnu knjigu.

Alternativni scenarij:

5a. Upisani podaci o etaži nisu ispravni ili nisu potpuni.

5a1. Korisnik je unio nepotpune ili neispravne podatke o etaži. Iz tih razloga korisniku se prikazuju odgovarajuće poruke upozorenja i mogućnost da ispravi nedostatke čime se korisnik nalazi na četvrtoj točki.

5a2. Upravitelj unosi podatke o vlasniku, a vlasnik sa istim OIB-om je već upisan u sustav. Korisnika se upozorava da korisnik već postoji te se upisani podaci zamjenjuju sa onim podacima iz sustava čime se korisniku onemogućava njihovo mijenjanje. Korisnik nastavlja na točku pet.

5.3.6 Kreiranje glasovanja – poslovi redovne i izvanredne uprave

Primarni sudionik: predstavnik suvlasnika

Dionici: upravitelj, suvlasnici, predstavnik suvlasnika

Preduvjeti: Definirana stambena zgrada. Uspješno prijavljeni predstavnik suvlasnika za zgradu za koju se kreira glasovanje.

Uvjeti po završetku: Definirano glasovanje za zadanu stambenu zgradu.

Glavni uspješni scenarij:

1. Predstavnik suvlasnika pristupa stranici za kreiranje glasovanja za zadanu stambenu zgradu.

Strukturirana analiza i specifikacija informacijskog sustava za upravljanje stambenim zgradama

2. Predstavnik suvlasnika odabire vrstu posla: redovna ili izvanredna uprava, definira datum početka i datum završetka glasovanja, temu i opis.
3. Sprema podatke o glasovanju.
4. Uspješno kreirano glasovanje.

5.3.7 Glasovanje suvlasnika za poslove redovne i izvanredne uprave

Primarni sudionik: suvlasnici

Dionici: upravitelj, suvlasnici, predstavnik suvlasnika

Preduvjeti: Definirano glasovanje. Suvlasnik mora biti suvlasnik prostora zgrade za koju je definirano glasovanje

Uvjeti po završetku: Uspješno glasovanje suvlasnika. Više nije moguće glasovanje suvlasnika koji je već glasovao.

Glavni uspješni scenarij:

1. Suvlasnik dolazi na stranicu za glasovanje vezano za glasovanje redovne ili izvanredne uprave.
2. Suvlasnik odabire opciju „Da“ ili „Ne“
3. Suvlasnik je uspješno obavio glasovanje.

Alternativni scenarij:

- 1a. Suvlasnik se nije odazvao glasovanju, njegov glas se tumači kao negativan.

5.3.8 Prijava kvara

Primarni sudionik: suvlasnici, predstavnik suvlasnika, upravitelj

Dionici: suvlasnici, predstavnik suvlasnika, upravitelj

Preduvjeti: Uspješno prijavljen suvlasnik ili predstavnik suvlasnika ili upravitelj.

Uvjeti po završetku: Uspješno kreirana prijava kvara koja je spremna za obradu od strane upravitelja.

Glavni uspješni scenarij:

1. Korisnik pristupa stranici za podnošenje prijave kvara.
2. Korisnik popunjava nužne podatke za kreiranje prijave kvara.
3. Šalje prijavu.

5.3.9 Revizija kvara

Primarni sudionik: upravitelj

Dionici: suvlasnik, predstavnik suvlasnika, upravitelj, izvođač radova

Preduvjeti: Kreirana prijava kvara.

Uvjeti po završetku: Dodijeljen izvođač radova te stanje prijave kvara je „započeto“.

Glavni uspješni scenarij:

1. Upravitelj pristupa stranici za zadani kvar.
2. Ovisno o usluzi koja je zatražena dodaje izvođača radova s kojima surađuje.
3. Opcionalno definira dodatne instrukcije izvođačima radova.
4. Sprema definirane podatke.

5.3.10 Obrada kvara

Primarni sudionik: izvođač radova

Dionici: suvlasnik, predstavnik suvlasnika, upravitelj, izvođač radova

Preduvjeti: Obrada kvara je u tijeku od strane izvođača radova.

Uvjeti po završetku: Obrada čeka na potvrdu predstavnika suvlasnika.

Glavni uspješni scenarij:

1. Izvođač radova pristupa stranici za određeni kvar.
2. Proglašava kvar popravljenim.
3. Upisuje zaključak sa opisom rješenja i postupcima.
4. Sprema podatke.

5.3.11 Potvrđivanje popravka

Primarni sudionik: predstavnik suvlasnika

Dionici: suvlasnik, predstavnik suvlasnika, upravitelj, izvođač radova

Preduvjeti: Kvar mora imati status „završeno“.

Uvjeti po završetku: Popravak je potvrđen te izvođač radova može izraditi račun za popravak. Alternativno, popravak nije potvrđen, kvar se stavlja u stanje „započeto“ gdje izvođač radova mora još doraditi popravak.

Glavni uspješni scenarij:

1. Predstavnik suvlasnika pristupa stranici za zadani kvar, odnosno popravak.
2. Radi evaluaciju na temelju dostupnih podataka.
3. Potvrđuje popravak.
4. Sprema podatke.

Alternativni scenarij:

- 3a. Predstavnik suvlasnika nije zadovoljan popravkom kvara. Postavlja status na „započeto“ i specificira napomene vezano za ne slaganje sa obavljenim poslom.

5.3.12 Izdavanje računa zgradi za obavljene popravke

Primarni sudionik: izvođač radova

Dionici: suvlasnik, predstavnik suvlasnika, upravitelj, izvođač radova

Preduvjeti: Potvrđen popravak za kojeg se izdaje račun.

Uvjeti po završetku: Izdan račun zgradi za obavljen popravak.

Glavni uspješni scenarij:

1. Izvođač radova pristupa stranici za izdavanje računa.
2. Odabire popravak za kojeg nije izdan račun.
3. Popunjava podatke, dodaje stavke računa.
4. Izdaje račun.

5.3.13 Plaćanje troškova zgrade

Primarni sudionik: predstavnik suvlasnika

Dionici: suvlasnik, predstavnik suvlasnika, upravitelj, izvođači radova

Preduvjeti: Postoje računi za naplatu određene zgrade.

Uvjeti po završetku: Plaćeni račun. Umanjen iznos novca u pričuvi za iznos računa.

Glavni uspješni scenarij:

1. Predstavnik suvlasnika pristupa stranici sa neplaćenim računima.
2. Odabire račun koji želi platiti.
3. Plaća račun.
4. Račun je plaćen.

Alternativni scenarij:

- 4a. Nema dovoljno novaca u pričuvi za plaćanje računa. Račun nije plaćen. Korisnik može ponoviti korake od početka kada bude dovoljno novaca u pričuvi.

5.3.14 Izdavanje računa za pričuvu

Primarni sudionik: sustav

Dionici: suvlasnik, predstavnik suvlasnika, upravitelj

Preduvjeti: Mjesečno izdati račune za pričuvu svim suvlasnicima zgrade ukoliko im već nisu računi izdani.

Uvjeti po završetku: Izdani računi za svakog suvlasnika u tekućem mjesecu.

Glavni uspješni scenarij:

1. Automatski pokrenuti izdavanje računa za plaćanje pričuvu u tekućem mjesecu.

6. Implementacija informacijskog sustava za upravljanje stambenim zgradama

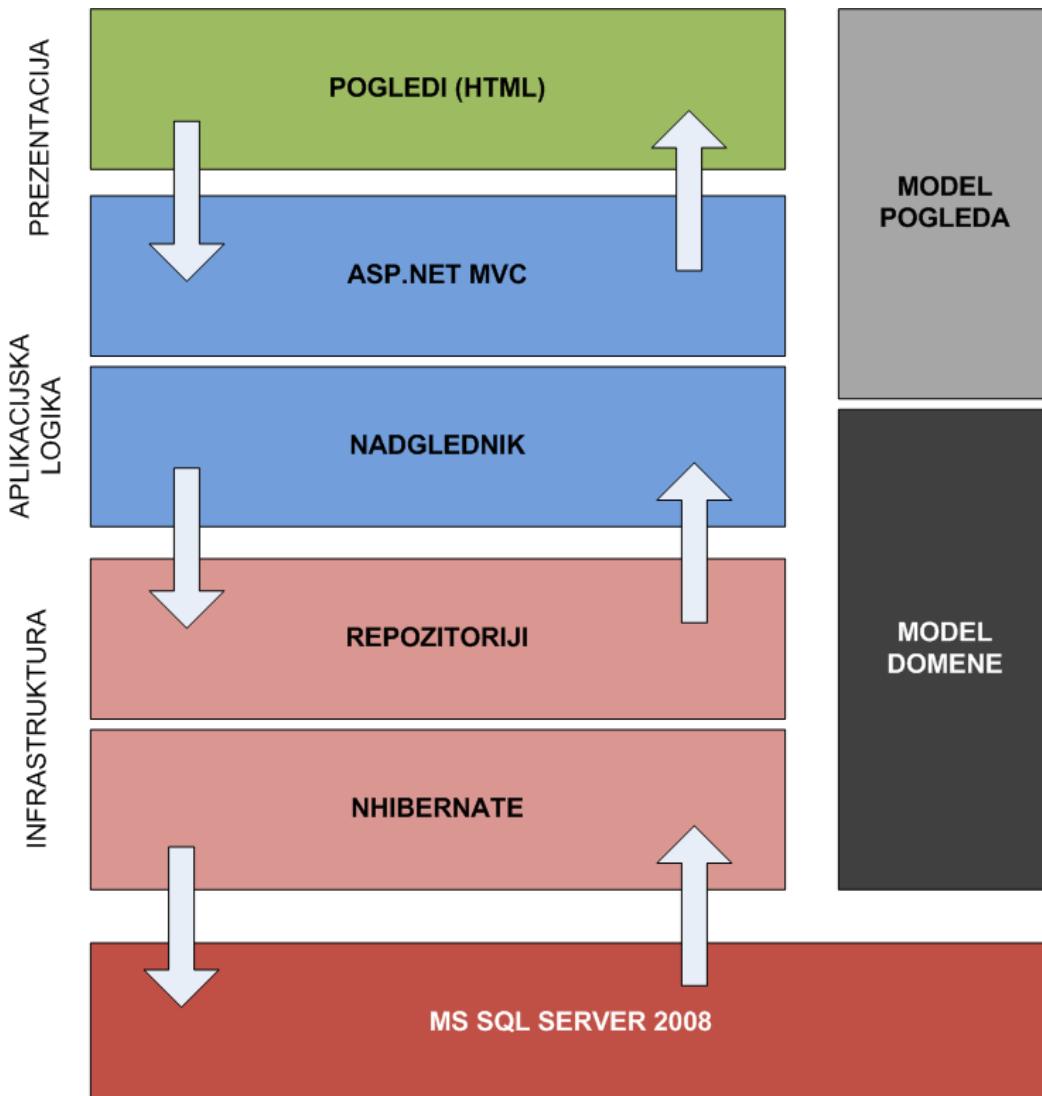
U sljedećim poglavljima opisati ćemo kako smo iskoristili, implementirali i integrirali prethodno definirane koncepte i tehnologije poput razvoja vođenog domenom, objektno – relacijskog mapiranja i oblikovnog obrasca model-pogled-preglednik u informacijski sustav u obliku web primjenskog programa.

Informacijski sustav biti će ostvaren kao web aplikacija uz pomoć ASP.NET MVC 3 razvojne okoline. Za objektno – relacijsko mapiranje korišten je NHibernate verzije 3.0. Programski jezik korišten za programsко ostvarenje je C# , u okolini .NET 4.0.

Implementacija informacijskog sustava za upravljanje stambenim zgradama ostvarena je kroz tri sloja:

- Sloj modela domene (DDD)
- Infrastrukturni sloj (NHibernate)
- Aplikacijski i prezentacijski sloj (ASP.NET MVC)

Za grafički prikaz koristi se *Razor view engine* koji je optimiziran za generiranje HTML-a. Zahtjevi sa grafičkog sučelja prolaze kroz ASP.NET MVC web okolinu i dolaze do nadglednika koji upravljaju tokom izvođenja aplikacije i provode aplikacijski logiku. Nadglednici dohvaćaju podatke preko infrastrukturnog sloja, odnosno repozitorija koji su izgrađeni nad funkcionalnošću objektno relacijskog sustava mapiranja NHibernate. Podaci su pohranjeni u MS SQL Server 2008 bazu podataka i prenose se između slojeva u oblicima prilagođenom pojedinom sloju. Podaci se iz modela domene transformiraju u modele pogleda koji su prilagođeni isključivo prezentaciji.



Slika 15 - Arhitektura sustava

6.1 Implementacija modela domene

Na početku ovoga diplomskog rada definirali smo što je razvoj vođen domenom, njegove gradbene elemente i rekli smo da model domene mora biti neovisan o drugim razvojnim okolinama, neovisan o sloju perzistencije. Takav pristup pomalo je utopijski jer u stvarnom svijetu nije moguće model domene totalno izolirati od ostalih slojeva. Drugim riječima potrebna su određena znanja o slojevima i razvojnim okolinama u kojima će se model domene koristiti, radi međusobne kompatibilnosti i učinka na performanse, i prema tome prilagoditi model domene.

Kako bi što bolje iskoristili bogati model domene i perzistenciju njegovih elemenata u relacijsku bazu podataka koristiti ćemo objektno – relacijski maper NHiberante. U

prethodnim poglavljima objasnili smo da NHibernate zahtjeva primjenu određenih pravila prilikom modeliranja modela domene radi preslikavanja u relacijsku bazu podataka. Većina pravila koja NHibernate nameće prilikom dizajniranja modela domene spadaju u dobre prakse objektno – orijentirane paradigme.

Zbog načina na koji NHibernate sprema i dohvaća entitete, što je posljedica definiranih asocijacija među objektima u nekim slučajevima se odstupa od DDD principa, te se pojedini entiteti proglašavaju korijenskim agregatima (premda to prirodno ne bi bili) kako bi se zadovoljile performanse. Primjerice, pretpostavimo da postoji jedan korijenski agregat koji sadrži kolekciju(skup) entiteta koji su članovi tog agregata. Po pravilima DDD-a u repozitorij je moguće samo dodavati korijenske aggregate. Što nas dovodi do činjenice da entitete koji su elementi kolekcije korijenskog agregata možemo persistirati isključivo ukoliko ih dodamo u kolekciju preko korijenskog agregata što savršeno zadovoljava DDD. Međutim problem nastaje u performansama, odnosno NHibernate će morati dohvatiti sve elemente kolekcije kako bi se provjerio da li je element koji se dodaje duplikat, odnosno da li zadovoljava pravila skupa. Stoga je u ovome slučaju prihvatljivo proglašiti element kolekcije kao vlastiti korijenski agregat koji se onda može persistirati kroz vlastiti repozitorij.

Repozitoriji su koncept iz modela domene te sučelja koja oni predstavljaju moraju biti u skladu sa sveprisutnim jezikom, odnosno dio domene. Međutim, zbog učinaka na performanse, u praksi se u ta sučelja uvode elementi tehničke prirode koji ne opisuju pojam iz domene već sudjeluju u službi učinaka na performanse.

U narednim potpoglavlјima opisati ćemo ostvarenje domene na primjeru upravljanja stambenim zgradama, primjenom DDD-a sa naglaskom na potporu za NHibernate. Na početku ćemo opisati osnovne apstrakcije koje će nam pomoći pri modeliranju konkretnih elemenata i poštivanju određenih pravila koja se nameću pri povezivanju sa NHibernateom kao objektno – relacijskim maperom. 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 poglavљa 5. *Strukturirana analiza i specifikacija informacijskog sustava za upravljanje stambenim zgradama* imenovati odgovarajuće objekte, te na isti način imenovati metode.

Za svaki korijenski agregat definirana je posebna xml datoteka koja sadrži podatke o mapiranju između modela domene i relacijske baze podataka. Sadržaj tih datoteka nećemo posebno komentirati. Svi detalji dostupni su u izvornome kodu informacijskog sustava priloženog uz ovaj rad.

6.1.1 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 te omogućiti podršku za NHibernate.

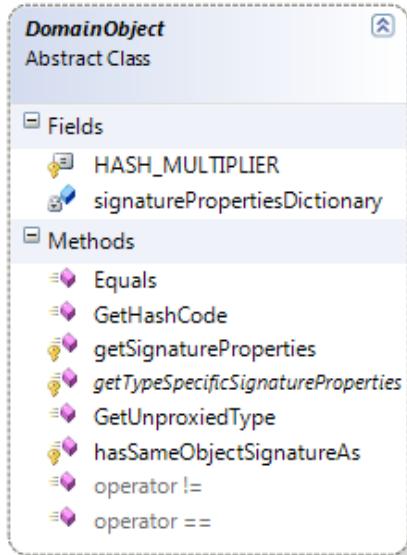
Ono u čemu se DDD i NHibernate savršeno slažu je koncept entiteta (pitanje identiteta) i vrijednosnog objekta. NHibernate podržava nekoliko načina generiranja identifikatora. Jedan od čestih načina i onaj koji se koristi u ovome radu je prepuštanje bazi podataka generiranje identifikatora. Problem koji se javlja je činjenica da entitet nema dodijeljen identifikator po svojem stvaranju, već ga dobiva persistencijom u bazu podataka. Takav problem može imati za posljedicu neispravan rad aplikacije ukoliko dođe do usporedbe ne persistenih entiteta. Iz tog se razloga definira poslovni ključ određenog entiteta, koji je određen pažljivim odabirom određenih svojstava tog entiteta koji ga čine jedinstvenim, barem u velikoj mjeri. Što se tiče vrijednosnog objekta možemo zaključiti da su dva vrijednosna objekta jednakia ukoliko su im sva svojstva jednaka.

Vidimo da ključnu stvar u ostvarenju entiteta i vrijednosnih objekata čine operacije uspoređivanja, odnosno operacije `Equals()` i `GetHashCode()`. Ono što je zajedničko vrijednosnom objektu i ne persistiranome entitetu je usporedba po svojstvima entiteta i usporedba po tipu, odnosno razredu. Kako bi NHibernate omogućio lijeno dohvaćanje svaki razred omotava pomoću proxy objekta koji se nasljeđuje iz objekta kojeg „štiti“, stoga problem nastaje kada želimo uspoređivati takve objekte po tipu. Na primjer, kada uspoređujemo bazni entitet i proxy entitet koji je izведен iz baznog entiteta. Naizgled oni nisu jednakii jer nisu istog tipa, međutim logički predstavljaju isti tip, pa čak i u određenim slučajevima isti entitet.

Na temelju takvih činjenica definirati ćemo apstraktnu hijerarhiju naslijedenih objekata koji će riješiti problem identiteta i jednakosti objekata. Korijenski apstraktan razred predstavlja `DomainObject`, odnosno domenski objekt iz kojeg će se nasljeđivati koncepti poput entiteta i vrijednosnog objekta.

Implementacija informacijskog sustava za upravljanje stambenim zgradama

DomainObject definira metodu `GetUnproxiedType()` koja omogućuje dohvaćanje izvornog tipa čime se rješava problem usporedbe izvornih tipova ukoliko su predstavljeni proxy objektom.



Slika 16 - Domenski objekt

Razred definira apstraktnu metodu `getTypeSpecificSignatureProperties()` čija je zadaća dohvat onih svojstava koja su podobna za usporedbu. Implementacija metode `Equals()` koristi prethodno navedene metode, odnosno sastoji se od nekoliko koraka:

- Usporedba po referenci
- Usporedba po tipovima (`GetUnproxiedType()`)
- Usporedba po potpisu određenog tipa, odnosno njegovim svojstvima koja predstavljanju njegovu jedinstvenost (`hasSameObjectSignatureAs()`)
 - Metoda `hasSameObjectSignatureAs()` poziva metodu `getTypeSpecificSignatureProperties()` te izvodi usporedbu svojstava dvaju objekata.

Pomoću metode `Equals()` implementirani su operatori jednakosti `==` i nejednakosti `!=`.

Metoda `GetHashCode()` implementirana je na način da se *hash code* računa iz svih svojstava za usporedbu, *hash codea* tipa objekta i slučajnog množitelja `HASH_MULTIPLIER` koji osigurava jedinstvenost.

Entitet je predstavljen generičkim razredom `Entity<TId>` koji nasljeđuje `DomainObject` te implementira metodu `getTypeSpecificSignatureProperties()`. Ta metoda kao rezultat vraća svojstva koja predstavljaju poslovni ključ.

Poslovni ključ entiteta definira se dekoriranjem svojstava entiteta pomoću atributa `BusinessKeyOfEntityAttribute`. Osim poslovnog ključa, definirano je generičko svojstvo `Id` koje predstavlja identitet entiteta.

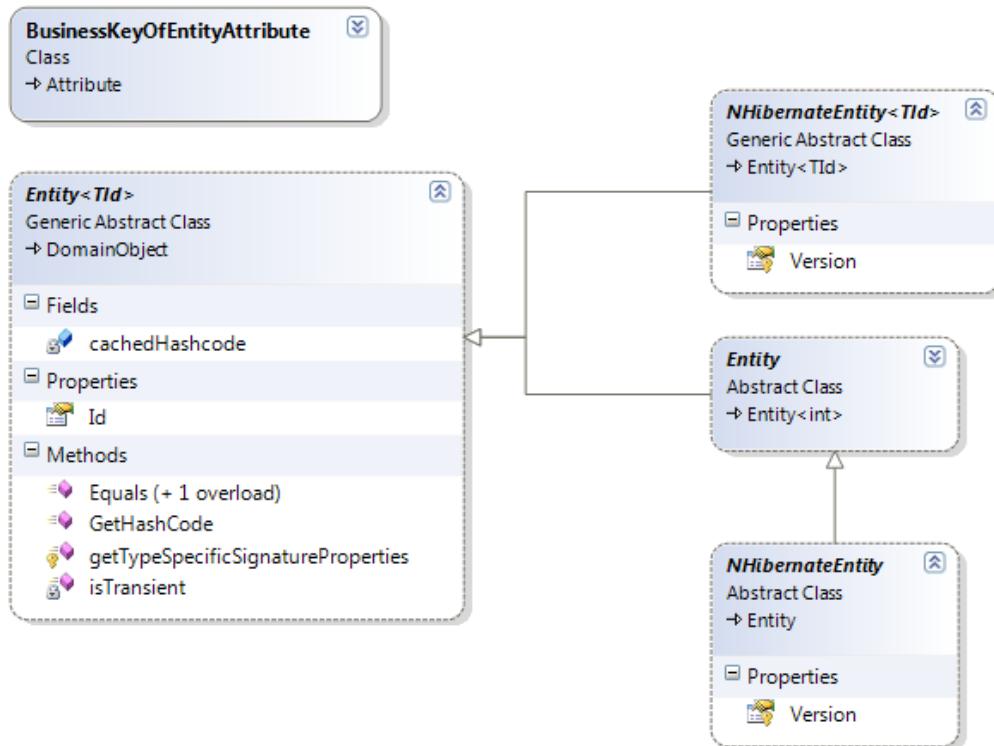
Zbog činjenice da postoje tzv. dvije strategije usporedbe, jedna kada je objekt tranzijentan (usporedba po poslovnom ključu) i jedna kada objekt nije tranzijentan (usporedba po identifikatoru dodijeljenome od baze podataka) potrebno je ponovno implementirati metodu `Equals()`. Metoda u ovisnosti o tranzijentnosti uspoređuje po identifikatoru ili se oslanja jednim dijelom na implementaciju metode `Equals()` baznog razreda.

Zbog dvojne strategije usporedbe ovisno o stanju tranzijentnosti objekta potrebno je nanovo implementirati `GetHashCode()` metodu. Karakteristika te metode je da za jedan objekt uvijek vraća istu vrijednost, ali zbog dvojne strategije usporedbe, metoda bi trebala vraćati različite vrijednosti u ovisnosti o stanju tranzijentnosti. Kako bi zadovoljili prethodno spomenutu karakteristiku odlučili smo se za čuvanje vrijednosti *hash codea*, odnosno za vrijeme trajanja životnog ciklusa metoda će vraćati isti kod koji je proizveden prilikom stvaranja objekta. Tako će prilikom prvog stvaranja entiteta, pa i nakon perzistencije, metoda vraćati *hash code* izračunat na temelju svojstava koja predstavljaju poslovni ključ, a u novom životnom ciklusu objekta koji je nastao rekonstitucijom iz perzistencije metoda će vraćati *hash code* izračunat na temelju identifikatora.

Kako je razred `Entity<TId>` generički razred, definiran je poseban razred `Entity` koji se nasljeđuje iz generičkog razreda sa cjelobrojnim tipom podatka. Drugim riječima, razred `Entity` radi sa cjelobrojnim identifikatorima koji su najčešći u slučajevima kada je baza zadužena za generiranje identifikatora.

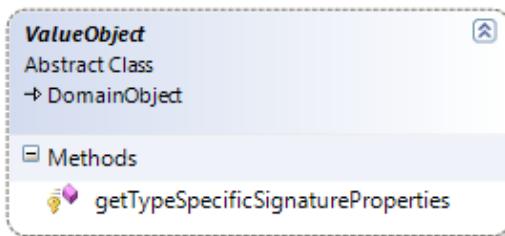
```
public abstract class Entity : Entity<int> { }
```

Zbog načina na koji NHibernate radi sa konkurentnim transakcijama, odnosno kako radi optimističko zaključavanje, implementiran je poseban razred `NHibernateEntity<TId>` nastao nasljeđivanjem iz `Entity<TId>` koji uvodi novo svojstvo `Version` koje nema nikakvo značenje za domenu, već isključivo ima značenje za NHibernate. To svojstvo NHibernate sam ažurira.



Slika 17 - Apstraktni razredi entiteta

Implementacija vrijednosnog objekta ostvarena je pomoću razreda `ValueObject` koji se nasljeđuje iz razreda `DomainObject`. Razred implementira metodu `getTypeSpecificSignatureProperties()` koja kao rezultat vraća listu svih svojstava tog objekta. Ponovimo, dva vrijednosna objekta su jednaka ukoliko su im sva svojstva jednaka.

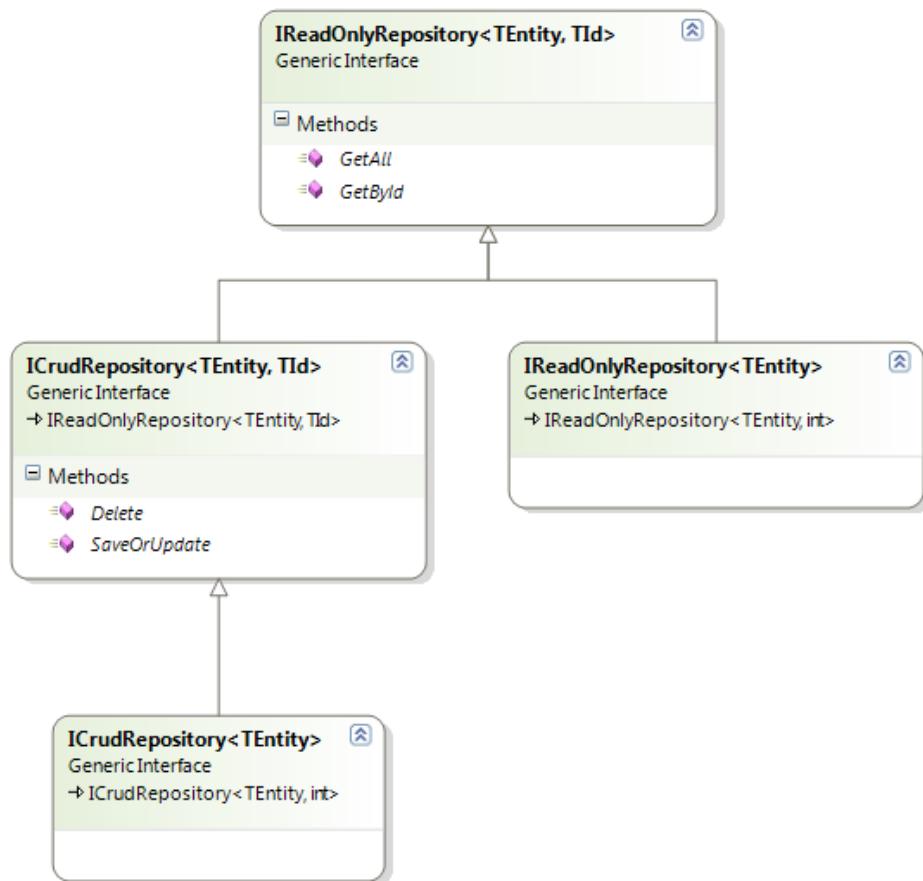


Slika 18 - Apstraktni vrijednosni objekt

Što se tiče repozitorija, definirali smo dva općenita repozitorija, koji su isključivo u službi entiteta odnosno korijenskih agregata, i njihove specijalizacije za cjelobrojne identifikatore:

- `IReadOnlyRepository< TEntity, TId >` – repozitorij isključivo za čitanje odnosno dohvaćanje entiteta. Čitanje je ostvareno preko operacija dohvaćanja entiteta preko identifikatora ili dohvaćanje svih entiteta.

- `IReadOnlyRepository< TEntity >` - specijalna vrsta repozitorija za čitanje koji radi sa cjelobrojnim identifikatorima entiteta.
- `ICrudRepository< TEntity, TId >` – repozitorij koji se nasljeđuje iz `IReadOnlyRepository` koji osim operacija čitanja nudi i operacije koje mijenjaju stanje, odnosno operacije brisanja i spremanja.
 - `ICrudRepository< TEntity >` - specijalna vrsta repozitorija čitanje i pisanje koji radi sa cjelobrojnim identifikatorima entiteta.



Slika 19 - Apstraktni repozitoriji

6.1.2 Osobe i uloge

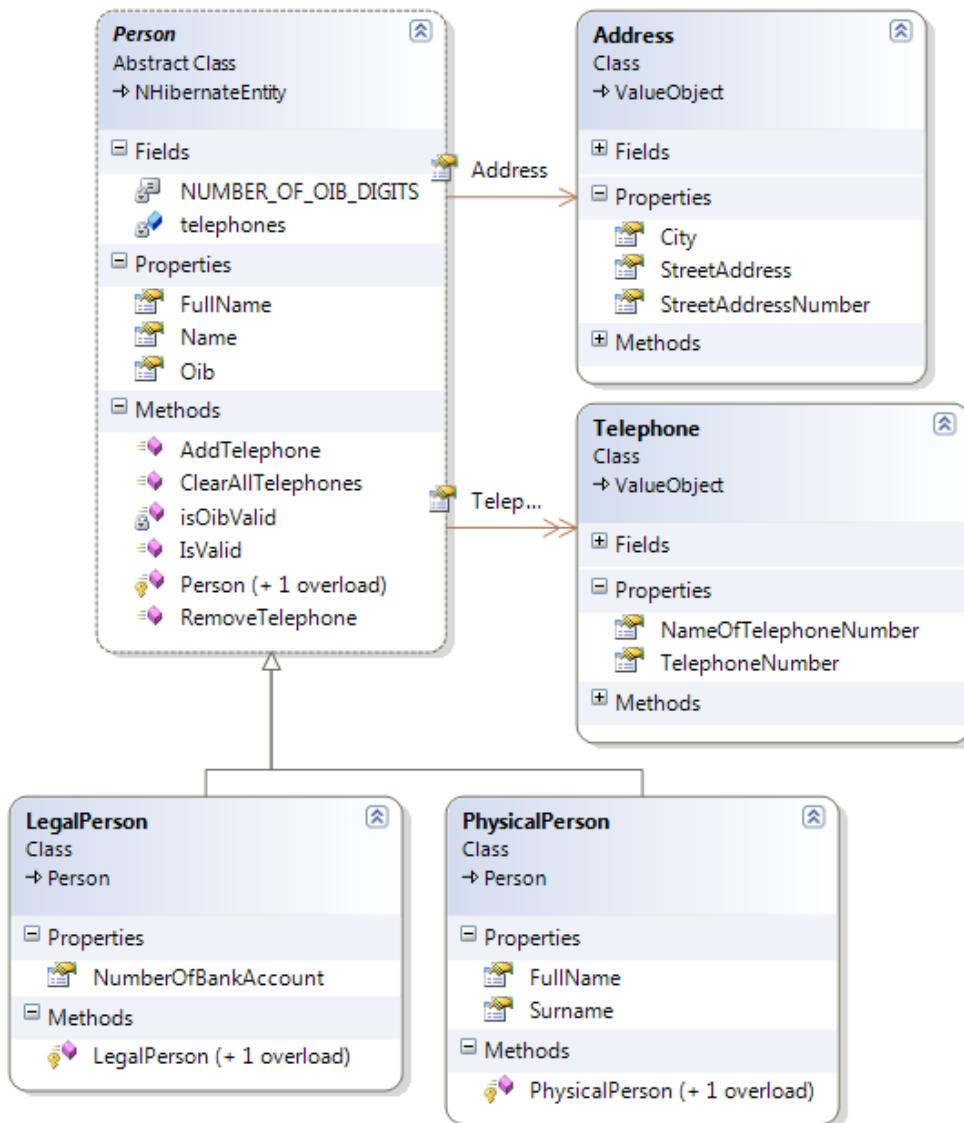
Osobe i uloge predstavlja zaseban modul u DDD-u koji obuhvaća osobe te njihove pripadajuće uloge.

Definirali smo apstraktni razred `Person` koji se nasljeđuje iz razreda `NHibernateEntity` 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

Implementacija informacijskog sustava za upravljanje stambenim zgradama

(PhysicalPerson) i pravne osobe (LegalPerson) jer naša domena uključuje kako stanare, odnosno vlasnike, tako i izvođače radova i upravitelja.

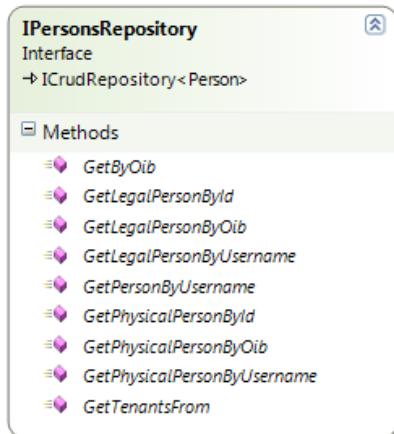
Slika 20. predstavlja agregat osoba u kojem razred Person i njegove specifikacije predstavljaju korijenski agregat.



Slika 20 - Entiteti osobe i pripadajući vrijednosni objekti

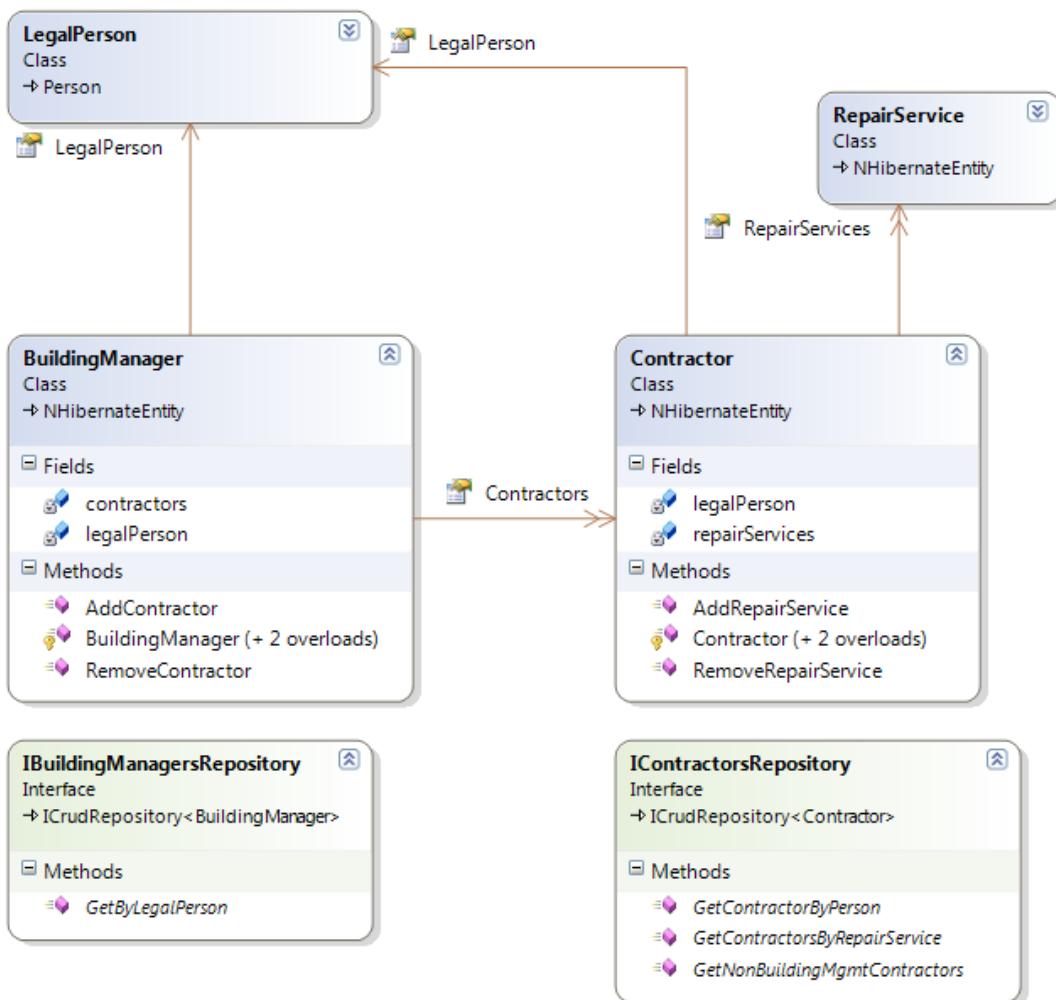
Vidimo da su osobe predstavljene kroz hijerarhiju nasljeđivanja stoga je takav odnos bilo potrebno definirani preko NHibernate xml datoteke mapiranja. Za strategiju NHibernate mapiranja odabrana je „Tablica za svaku hijerarhiju nasljeđivanja“, odnosno denormalizacija relacijskog modela koristeći posebni stupac za čuvanje informacija o tipu.

Repozitorij osoba definiran je kao repozitorij IPersonsRepository koji omogućava čitanje i pisanje, te cijeli niz metoda koji omogućuju dohvaćanje osoba po nekom kriteriju.



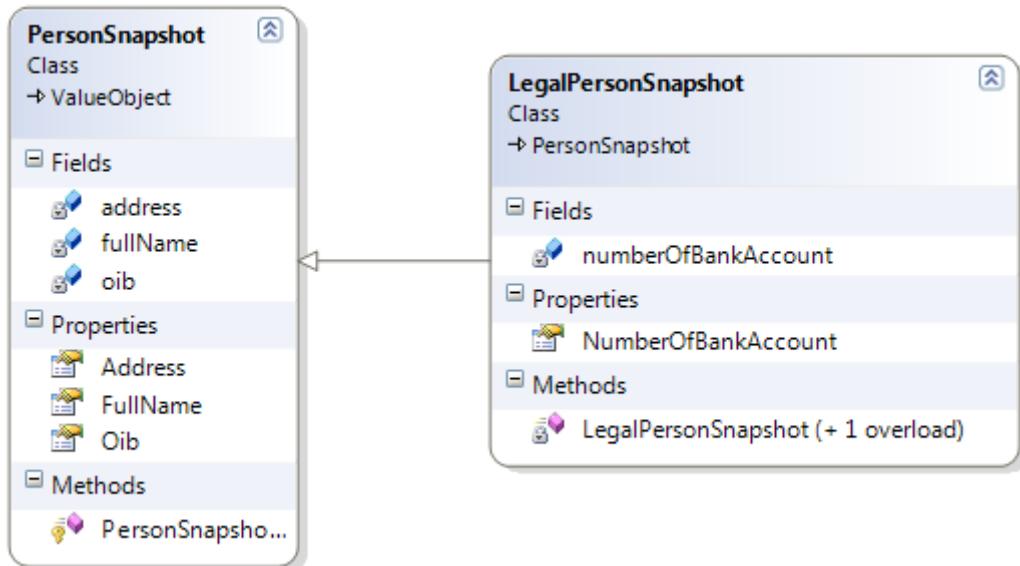
Slika 21 - Repozitorij osoba

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. Za sve korijenske aggregate definirani su i njihovi repozitoriji sa pripadajućim metodama.



Slika 22 - Entiteti Upravitelj i Izvodač radova

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

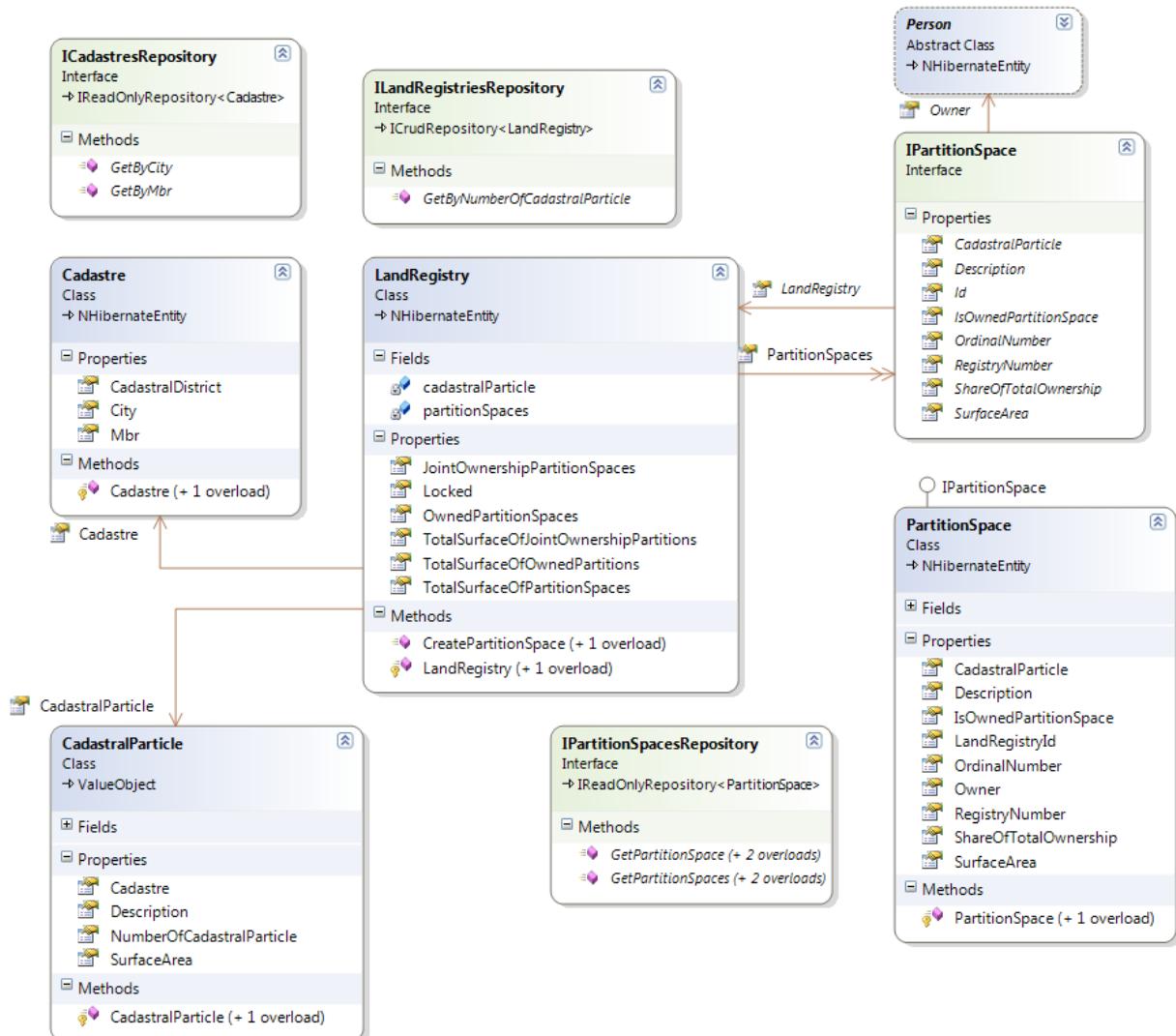


Slika 23 - Vrijednosni objekti osobe

6.1.3 Zakonodavstvo

Sljedeći dio domene je zakonodavstvo. Rekli smo 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), zemljišne čestice (AbstractCadastralParticle), te etaža (PartitionSpace).



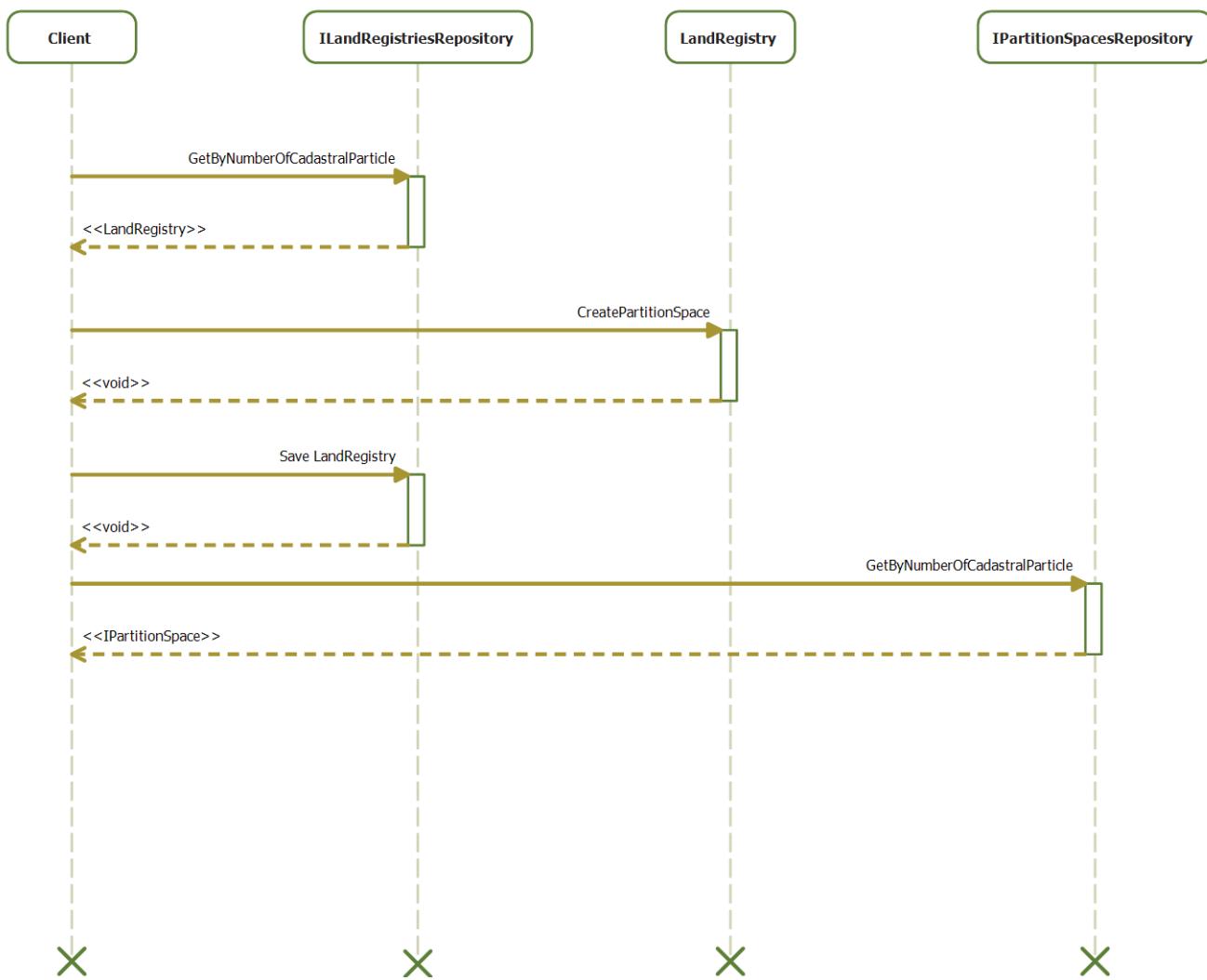
Slika 24 - Elementi zakonodavstva

Iz definicije agregata poznato je da korijenski agregat provodi sve invarijante agregata. Ovo je posebice bitno u slučaju etaže (**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 preko korijenskog agregata **LandRegistry** i to preko metode **CreatePartitionSpace()**. Metoda **CreatePartitionSpace()** predstavlja tvornicu ostvarenju preko oblikovnog obrasca „Factory method“, gdje se enkapsulira stvaranje etaže. Etažu je moguće persistirati 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

Implementacija informacijskog sustava za upravljanje stambenim zgradama

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.



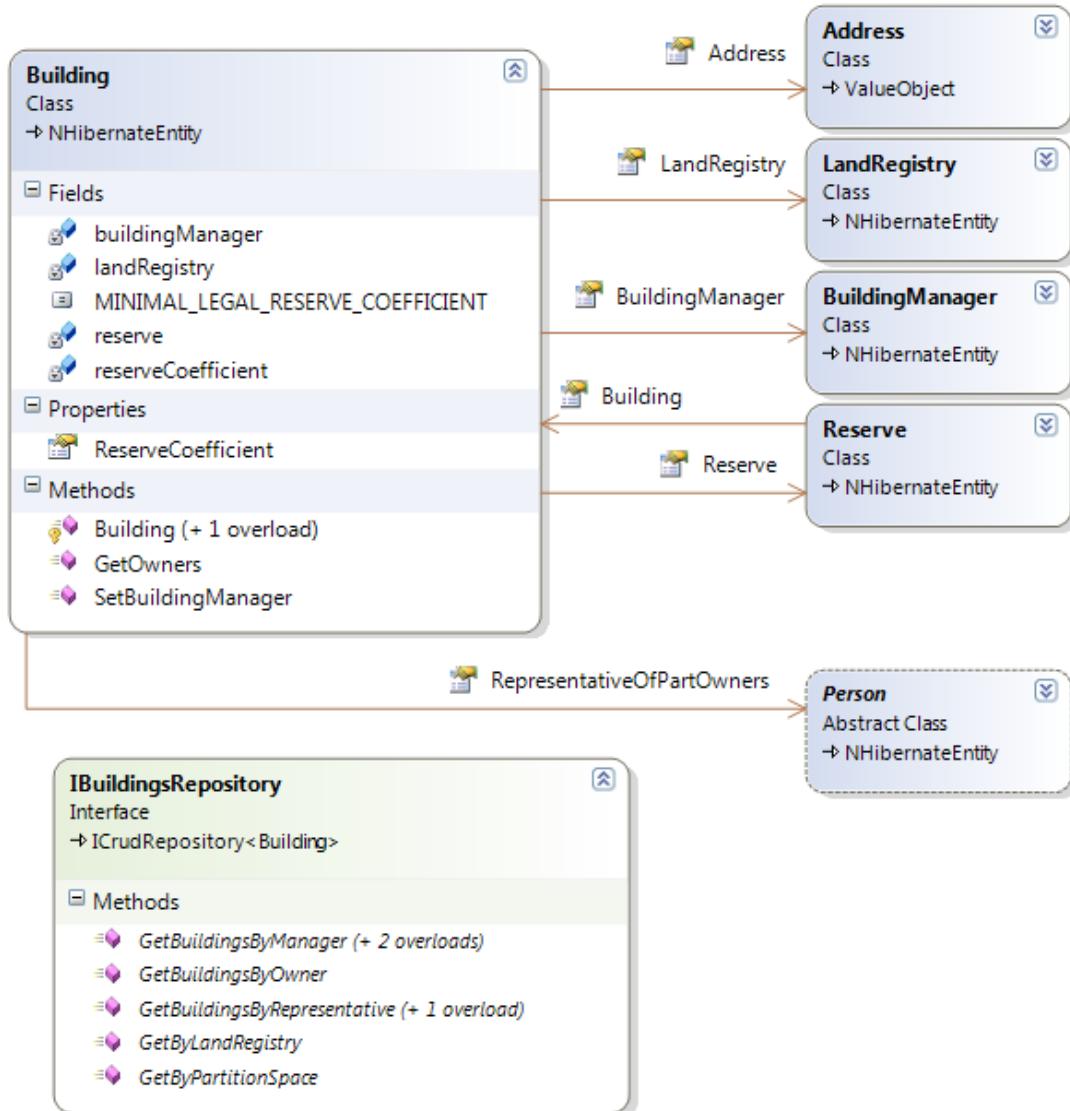
Slika 25 - Sekvencijski dijagram koji opisuje proces stvaranja etaže

6.1.4 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 `LandRegistry` s kojom je ostvarena dvosmjerna veza te pričuvom `Reserve`. Preko konstruktora zgrade,

Oblikovanje IS-a za upravljanje stambenim zgradama vođeno domenom primjene

odnosno prilikom stvaranja zgrade stvara se i pričuva. Drugim riječima pričuvu nije moguće kreirati izvan konteksta zgrade.



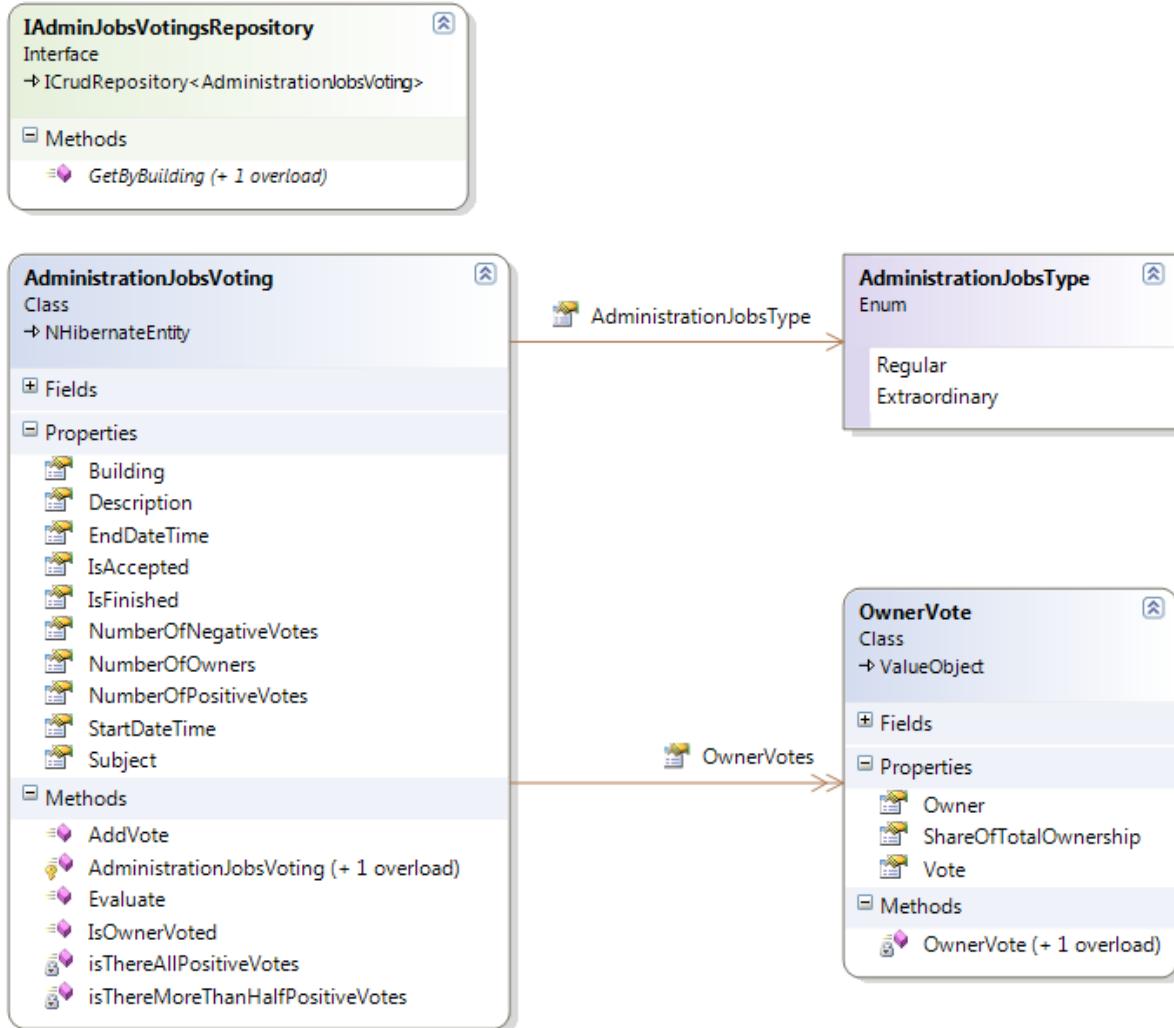
Slika 26 - Agregat zgrade

Za korijenski agregat definiran je repozitorij **IBuildingsRepository** koji omogućuje dohvaćanje zgrada po upravitelju zgrade, suvlasniku, predstavniku suvlasnika, zemljišnoj knjizi i etažama.

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

Implementacija informacijskog sustava za upravljanje stambenim zgradama

broju glasova zadovoljena. Objekt OwnerVote predstavlja vrijednosti objekta koji jedino živi unutar granice agregata i čija je svrha predstavljanje vlasničkog glasa s obzirom na vlasnički udio.



Slika 27 - Agregat redovne i izvanredne uprave

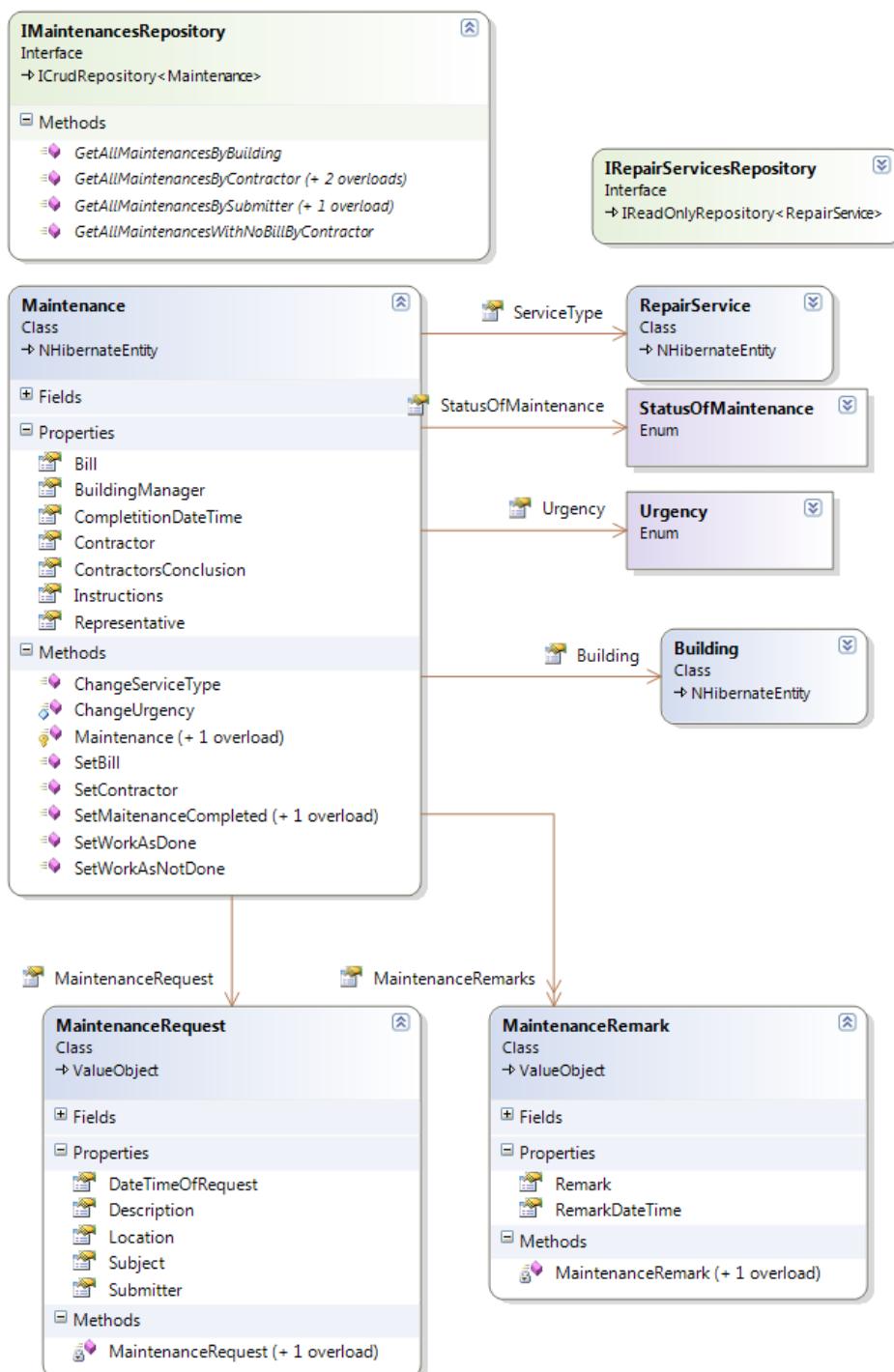
6.1.5 Prijava kvara

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),

Oblikovanje IS-a za upravljanje stambenim zgradama vođeno domenom primjene

statusa popravka (StatusOfMaintenance), izvođača radova (PersonSnapshot) i zgrade (Building). Stvaranje korijenskog agregata ostvareno je predajom u konstruktor parametra MaintenanceRequest 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.

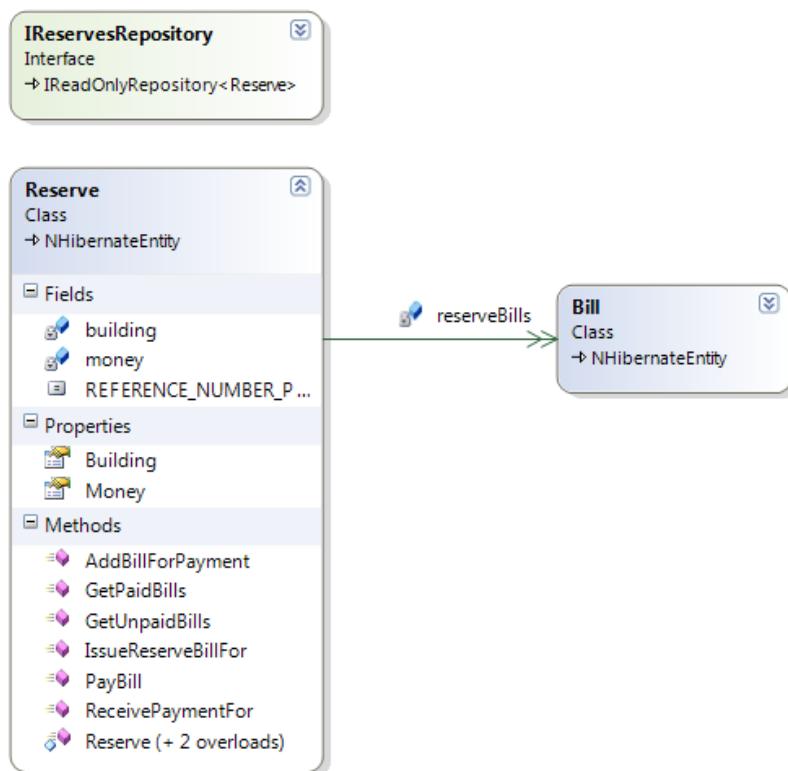


Slika 28 - Agregat održavanja

6.1.6 Financije

Financije u zgradi su definirane preko pričuve. Pričuva je u pravilu račun s novcem određene zgrade preko kojeg se plaća održavanje, te se primaju mjesecne uplate suvlasnika. Dakle, pričuva je definirana preko korijenskog agregata Reserve sa pripadajućim repozitorijem. Za pričuvu ne postoji javni konstruktor, već je pričuvu moguće instancirani isključivo u okviru zgrade za koju se kreira. Stoga možemo reći da se pričuva stvara kroz konstruktor zgrade što je ujedno i tvornica pričuve.

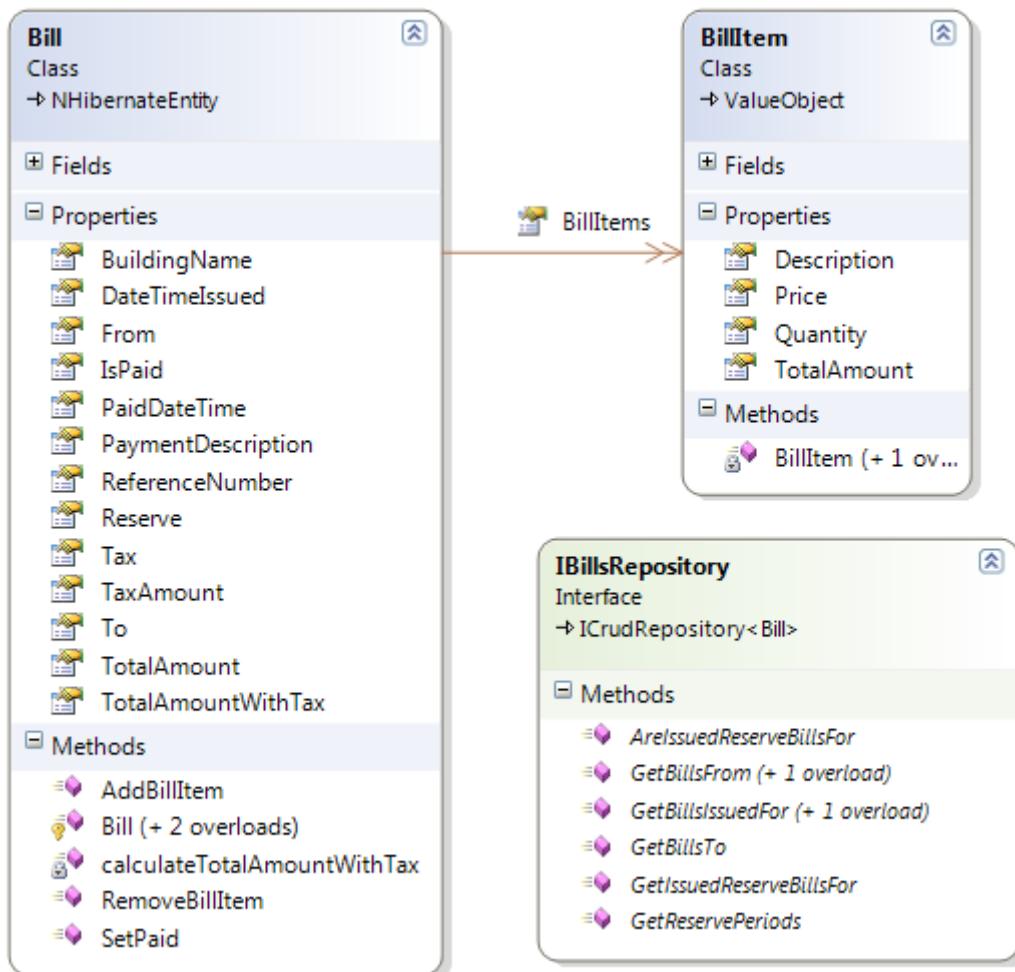
Pričuva omogućava plaćanje računa koji su izdani sa svrhom naplate održavanja zgrade kroz metode `AddBillForPayment()` koja dodaje račun za naplatu, te metode `PayBill()` pomoću koje se račun plaća novcem iz pričuve. Pomoću metode `IssueReserveBillFor()` omogućeno je izdavanje računa za pričuvu stanarima zgrade. Naplata izdanih računa za pričuvu omogućena je kroz metodu `ReceivePaymentFor()` preko koje se povećava stanje novca pričuve.



Slika 29 - Agregat pričuve

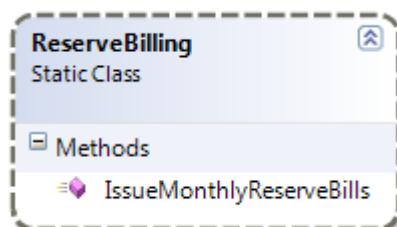
Kako bi se iz pričuve mogli plaćati računi, te primati uplate temeljem nekog računa definiran je korijenski agregat Bill. Agregat Bill sastoji se od stavki računa BillItems

koje su jedino interesantne unutar okvira agregata, pa stoga nisu dobavljive preko repozitorija.



Slika 30 - Agregat računa

Kako bismo mogli izdavati mjesecne račune za naplatu pričuve definirali smo servis ReserveBilling. Navedeni servis izdaje račune za trenutni mjesec pri tome pazeći da ne izda više puta isti račun za određeni mjesec.



Slika 31 - Servis za izdavanje računa za pričuvu

6.2 Baza podataka

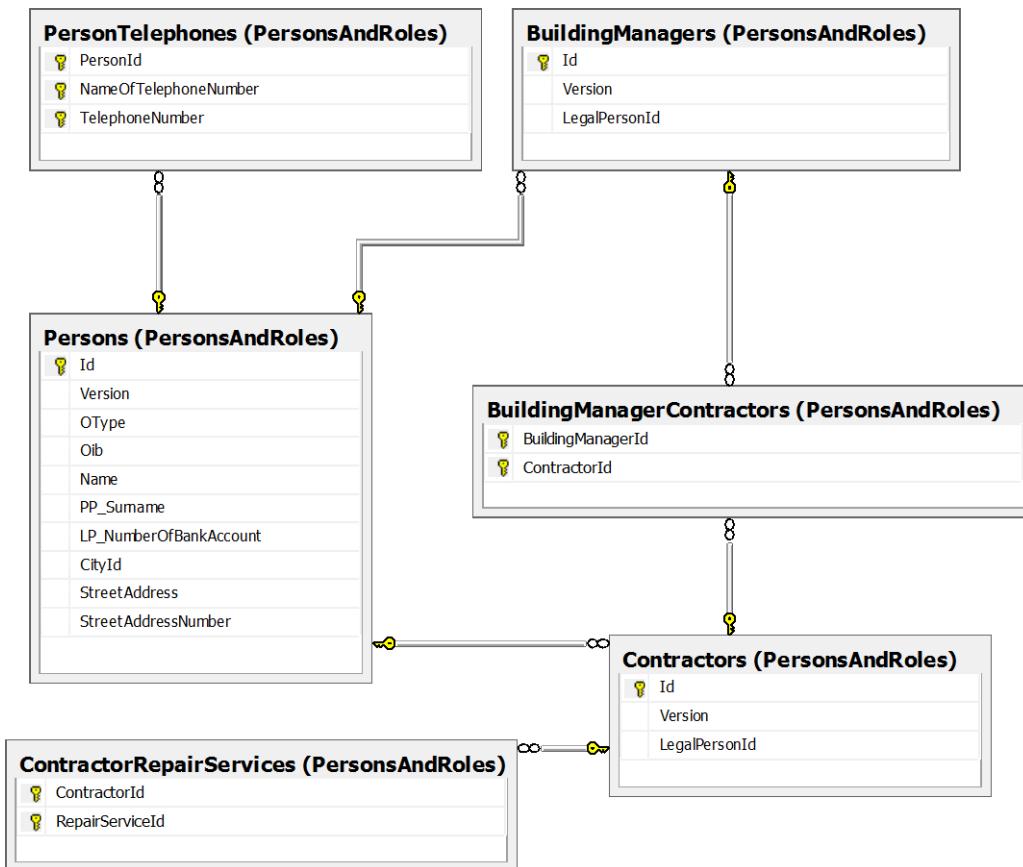
Baza podataka izrađena je ručno iz NHibernate specifikacije mapiranja. Model baze vrlo je sličan modelu domene jer je iz njega i nastao. Baza podataka je podijeljena na nekoliko shema po uzoru na module modela domene.

SQL naredbe za stvaranje odgovarajuće baze podataka sa pripadajućim tablicama priložen je uz ovaj rad.

6.2.1 Osobe i uloge

Model baze podataka sastoji se od sljedećih tablica/entiteta:

1. **Persons** – tablica sa entitetima osoba. Tablica je polimorfna, odnosno u tablicu je moguće spremiti pravne i fizičke osobe.
2. **PersonTelephones** – tablica sa vrijednosnim objektima telefonskih brojeva.
3. **BuildingManagers** – tablica sa entitetima upravitelja zgrada.
4. **Contractors** – tablica entitetima izvođača radova.
5. **BuildingManagerContractors** – predstavlja tablicu veza, konkretno many-to-many, između upravitelja zgrade i izvodača radova .
6. **ContractorRepairServices** – tablica veze između izvođača radova i njihovih usluga.

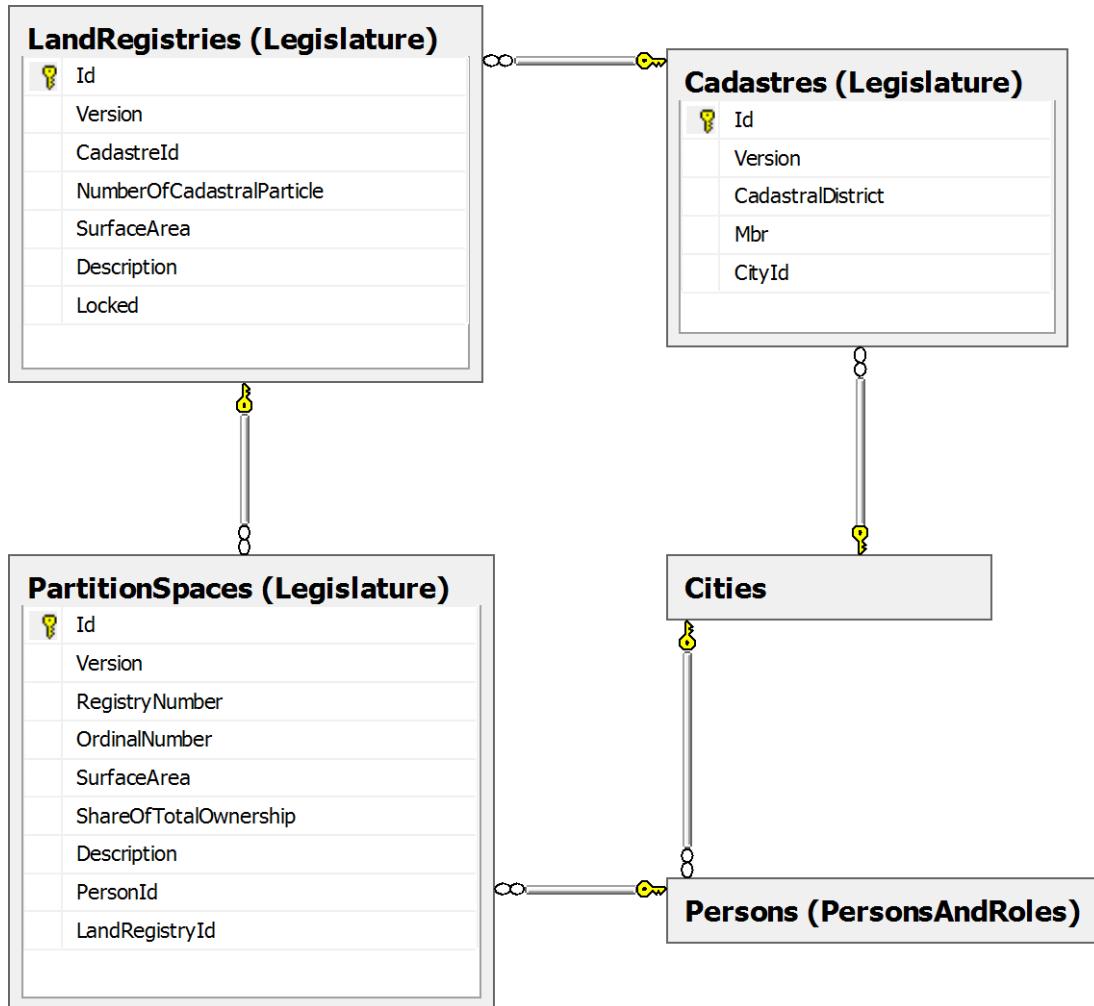


Slika 32 - Model baze podataka za osobe i usluge

6.2.2 Zakonodavstvo

Model baze podataka sastoji se od sljedećih tablica/entiteta:

1. **LandRegistries** – tablica sa entitetima zemljišnih knjiga.
2. **Cadastres** – tablica sa entitetima katastarskih općina.
3. **PartitionSpaces** – tablica sa entitetima etaža.
4. **Cities** – tablica entitetima gradova.

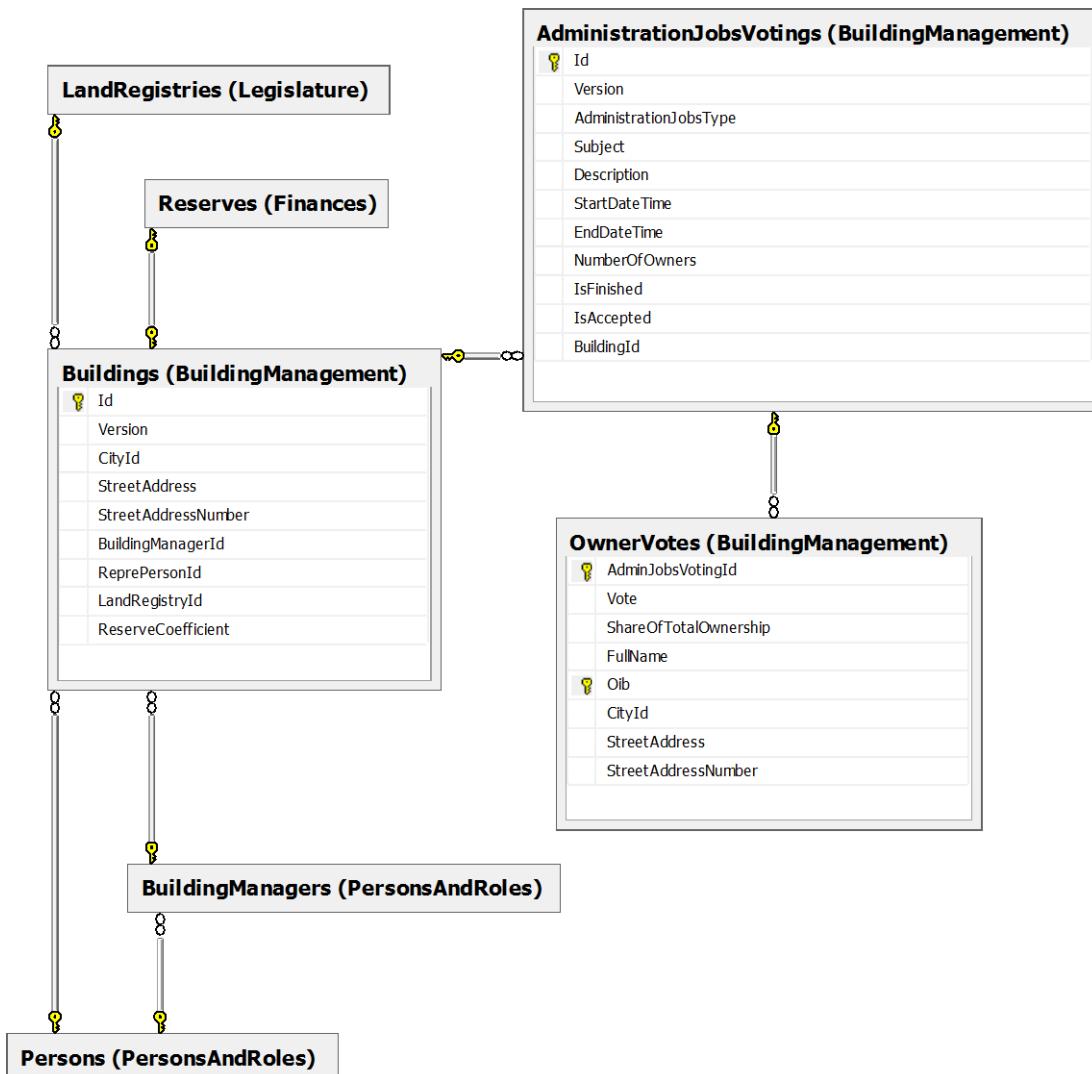


Slika 33 - Model baze podataka za zakonodavstvo

6.2.3 Upravljanje zgradom

Model baze podataka sastoji se od sljedećih tablica/entiteta:

1. **Buildings** – tablica sa entitetima zgrade.
2. **AdministrationJobsVotings** – tablica sa entitetima rada uprave.
3. **OwnerVotes** – tablica sa vrijednosnim objektima glasova suvlasnika.

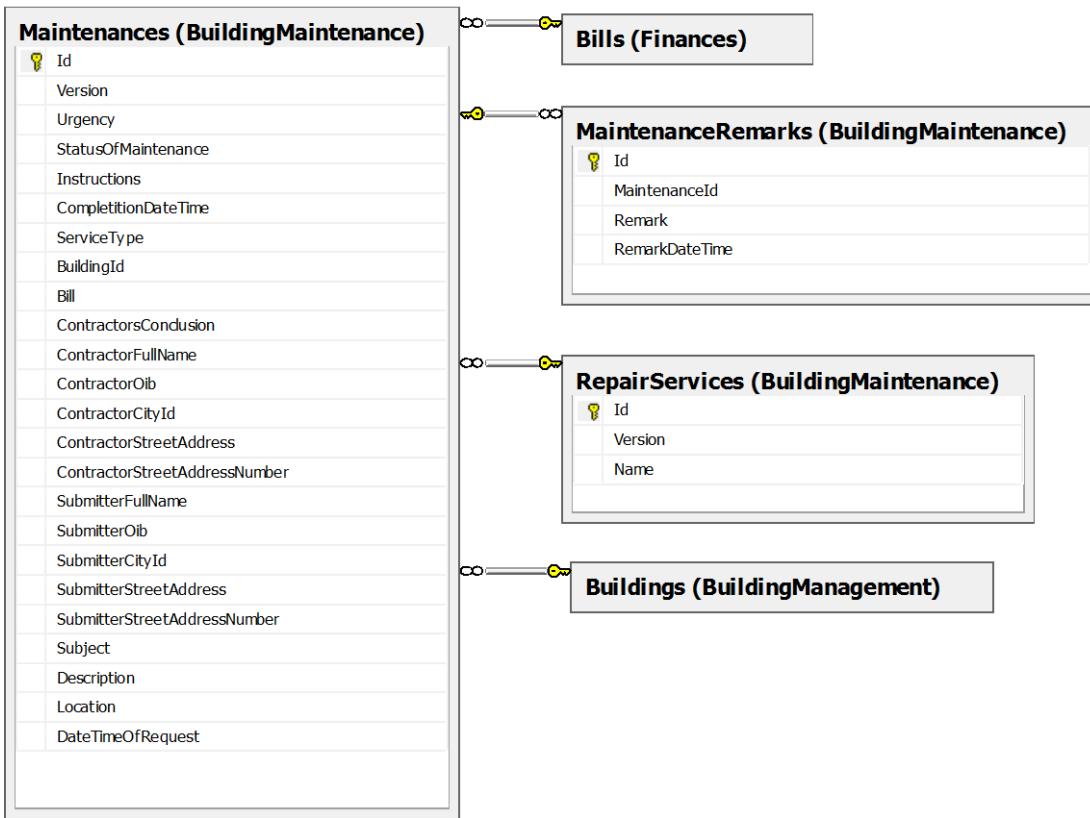


Slika 34 - Model baze podataka za upravljanje zgradom

6.2.4 Prijava kvara

Model baze podataka sastoji se od sljedećih tablica/entiteta:

1. **Maintenances** – tablica sa entitetima za održavanje/kvarove.
2. **MaintenanceRemarks** – tablica sa vrijednosnim objektima koji predstavljaju napomene predstavnika suvlasnika.
3. **RepairServices** – tablica sa entitetima usluga održavanja.

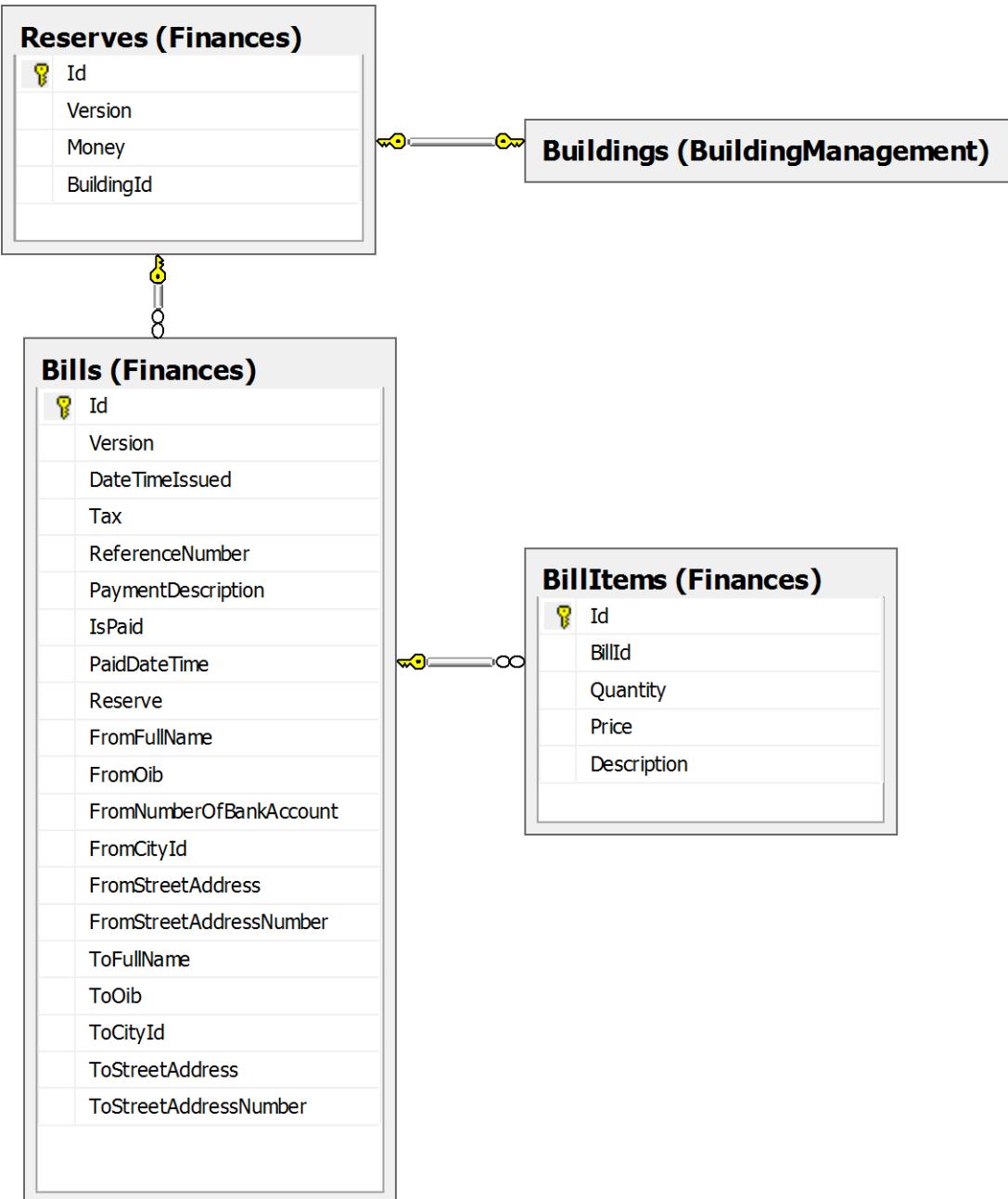


Slika 35 - Model baze podataka za prijavu kvara

6.2.5 Financije

Model baze podataka sastoji se od sljedećih tablica/entiteta:

1. **Reserves** – tablica sa entitetima pričuve zgrada.
2. **Bills** – tablica sa entitetima računa.
3. **BillItems** – tablica sa vrijednosnim objektima koji predstavljaju stavke računa.



Slika 36 - Model baze podataka za financije

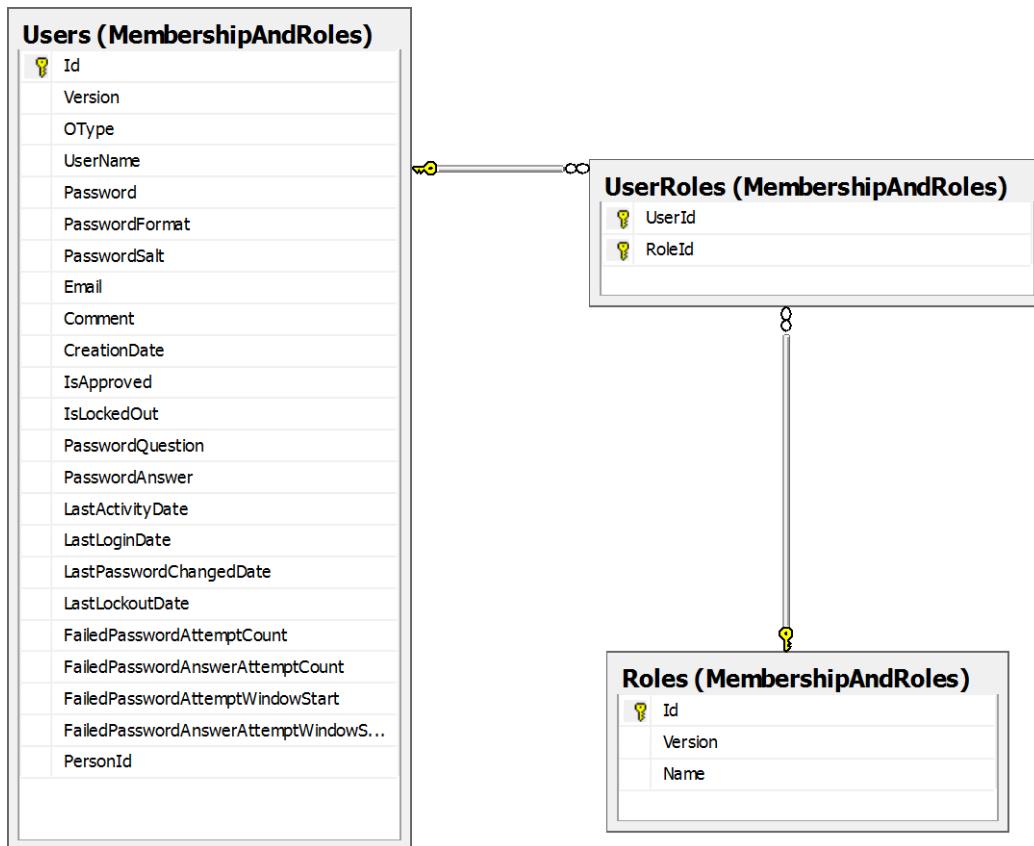
6.2.6 Korisnici i uloge

Tablice baze podataka koje pripadaju shemi *MembershipAndRoles* odnose se na korisnike, odnosno korisničke račune informacijskog sustava sa pripadajućim ulogama. Navedene tablice imaju svoju reprezentaciju u obliku objekata. Objekti koji predstavljaju korisnike i njihove uloge nisu bili do sada objašnjeni jer ne pripadaju domeni stambenih zgrada, te će biti objašnjeni u poglavlju 0

Implementacija infrastrukturnog sloja.

Model baze podataka sastoji se od sljedećih tablica/entiteta:

1. **Users** – tablica sa entitetima korisnika informacijskog sustava.
2. **Roles** – tablica sa entitetima uloga.
3. **UserRoles** – tablica veza koja definira koje uloge su dodijeljene pojedinim korisnicima.



Slika 37 - Model baze podataka koji predstavlja korisnike i uloge

6.3 Implementacija infrastrukturnog sloja

6.3.1 Sjednice i transakcije

NHibernate prepušta upravljanje sjednicama i transakcijama aplikaciji. Postoji nekoliko različitih načina za upravljanje sjednicama i transakcijama koji uvelike ovise o specifičnoj arhitekturi aplikacije. Kako je informacijski sustav za upravljanje stambenim zgradama ostvaren kao web aplikacija uobičajeni način za upravljanje NHibernate sjednicama je *sjednica po zahtjevu* (engl. *session per request*). Takav način upravljanja NHibernate sjednicama radi na principu da se za svaki zahtjev stvori nova sjednica koja traje sve dok se ne pošalje odgovor klijentu, odnosno kada se zahtjev obavi.

NHibernate značajka kontekstualne sjednice omogućuje da se sjednica asocira sa specifičnim djelokrugom (engl. *scope*) aplikacije koja aproksimira jedinicu posla. U slučaju web aplikacije uobičajeno je da se sjednica asocira sa web zahtjevom. Navedena se značajka konfigurira u NHibernate konfiguracijskoj datoteci na sljedeći način:

```
<property name="current_session_context_class">web</property>
```

Uz korištenje kontekstualnih sjednica, NHibernate automatski ne otvara ili zatvara sjednice. Potrebno je asocirati sjednicu sa trenutnim web zahtjevom korištenjem metoda `CurrentSessionContext.Bind()` i `Unbind()`.

Uobičajeno je da se asocijacija NHibernate sjednica sa web zahtjevom obavlja odmah pri dolasku web zahtjeva. Međutim kako se ovaj informacijski sustav zasniva na MVC paradigmu možemo poistovjetiti jedinicu posla sa akcijom određenog kontrolera. Stoga bi bilo poželjno da se asocijacija uspostavi što kasnije, odnosno onda kada je to zaista potrebno, a to je u ovome slučaju poziv metode/akcije kontrolera.

Vodeći se takvom logikom kreirali smo ASP.NET MVC akcijski filter `NHibernateTransactionAttribute`, pomoću kojeg je moguće dekorirati akciju kontrolera i u odgovarajući trenutak otvoriti i zatvoriti sjednicu i transakciju. Prije nego li se pozvana akcija kontrolera izvede, ASP.NET MVC će pozvati metodu `OnActionExecuting()` razreda `NHibernateTransactionAttribute`. U navedenoj metodi otvara se NHibernate sjednica i transakcija. Sjednica se zatim povezuje sa web zahtjevom koristeći značajku o kontekstualnoj sjednici. Kada pozvana akcija završi sa

izvođenjem pozvati će se metoda filtera `OnActionExecuted()`. Metoda će potvrditi transakciju ukoliko nije došlo do iznimke i ukoliko su iznimke obrađene ili će poništiti transakciju ukoliko je došlo do iznimke i ako je stanje modela u nekonzistentnom stanju. Nakon zatvaranja transakcije potvrđivanjem ili poništavanjem zatvara se sjednica i uništava se asocijacija sa web zahtjevom.

Takvim pristupom uklonili smo eksplisitno otvaranje sjednica i transakcija. Osim primjene filtera na akcijama kontrolera, filter je moguće primijeniti i na sam kontroler ukoliko sve akcije zahtijevaju pristup bazi podataka.

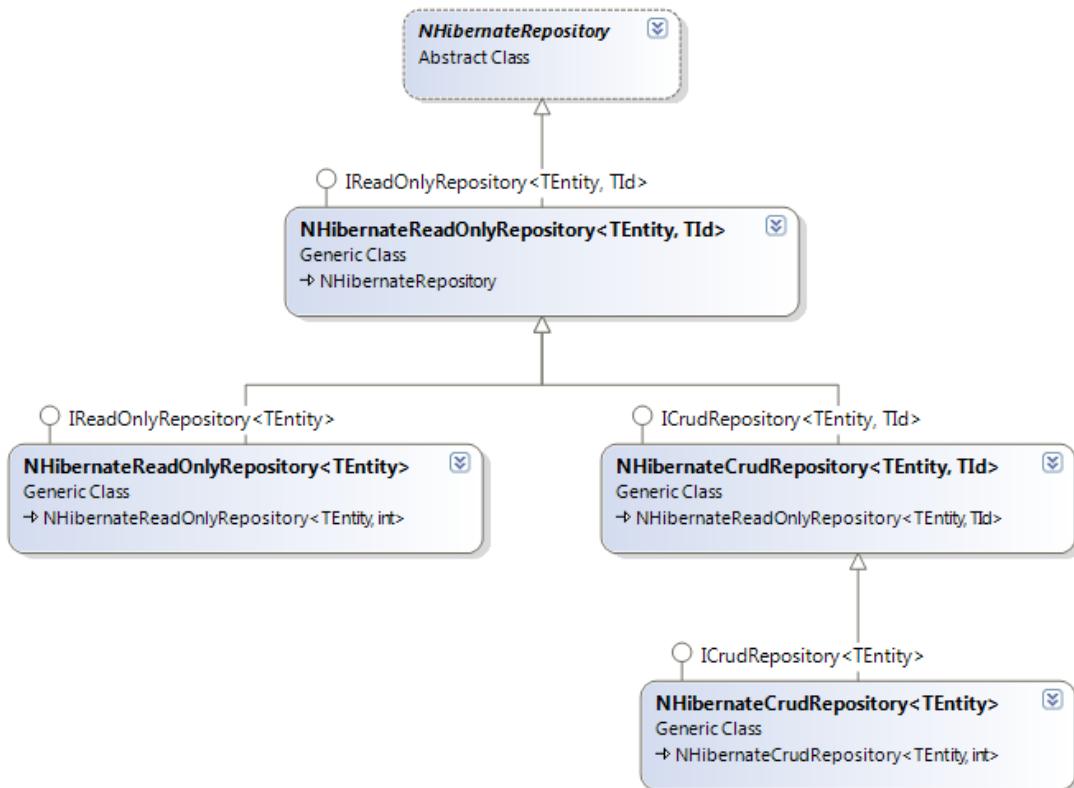
Važno je napomenuti da je sjednica otvorena samo unutar akcije kontrolera te da je moguće samo unutar sjednice izvoditi lijeno dohvaćanje. Po zatvaranju sjednice entitet s kojim radimo prelazi u tzv. *detached* stanje, odnosno nije asociran sa sjednicom te nije moguće lijeno dohvaćati asocirane entitete ili kolekcije entiteta. Ova činjenica je osobito važna za prikaz podataka korisniku, odnosno kada u pogled predajemo entitet, jer ukoliko se prezentiraju podaci koji nisu dohvaćeni dešava se događaj lijelog dohvaćanja koji rezultira iznimkom. Problem se rješava ili eksplisitnim navođenjem što se sve mora dohvatiti (engl. *eager fetching*) preko repozitorija ili obilaskom po objektnom stablu, odnosno asocijacijama i to uz pomoć mapera u dto(engl. *data transfer object*). Više o tome u poglavlju *6.4 Implementacija aplikacijskog i prezentacijskog sloja*.

Stvaranje same sjednice je jeftina operacija, dok je stvaranje tvornice iznimno skupa operacija. Zbog toga je uobičajena praksa da se tvornica sjednice stvari jednom i to prilikom pokretanja aplikacije. Kako bismo osigurali provođenje takvog pravila operaciju stvaranja tvornice sjednice omotali smo u *singleton* objekt `NHibernateSessionProvider` koji se inicijalizira pri pokretanju aplikacije. Inicijalizacija *singleton* objekta sastoji se od stvaranja konfiguracijskog objekta te temeljem istog stvaranje tvornice sjednica.

6.3.2 Repozitoriji

U poglavlju *6.1 Implementacija modela domene* opisali smo domenske apstrakcije repozitorija i njihove konkretizacije u obliku sučelja. U ovome poglavlju opisati ćemo njihove konkretne implementacije.

Svaki upit na bazu podataka izvodi se u kontekstu NHibernate sjednice. Stoga svaki repozitorij mora izvoditi svoje akcije nad trenutnom sjednicom. U takvom smjeru definiran je apstraktan razred `NHibernateRepository` koji preko svojeg konstruktora postavlja tvornicu sjednice u člansku varijablu preko koje je moguće dohvatiti trenutnu sjednicu. Konkretno, definirana su dva konstruktora, jedan prima kao parametar tvornicu sjednice (pogodno za unit testove) dok drugi tvornicu sjednice dobiva iz *singleton* objekta `NHibernateSessionProvider`. Temeljem tog baznog apstraktnog `NHibernate` repozitorija i domenskih apstraktnih sučelja repozitorija implementirani su generički repozitoriji. Drugim riječima, implementiran je repozitorij `NHibernateReadOnlyRepository< TEntity, TId >` samo za čitanje koji se bazira na razredu `NHibernateRepository` i domenskom sučelju `IReadOnlyRepository`. Isto tako implementiran je repozitorij `NHibernateCrudRepository< TEntity, TId >` za čitanje i pisanje baziran na razredu `NHibernateReadOnlyRepository` i domenskom sučelju `ICrudRepository`. Gdje `TEntity` predstavlja entitet za kojeg je repozitorij definiran, a `TId` tip podatka koji će predstavljati identifikator tog entiteta.



Slika 38 - Bazni razredi NHibernate implementacije repozitorija

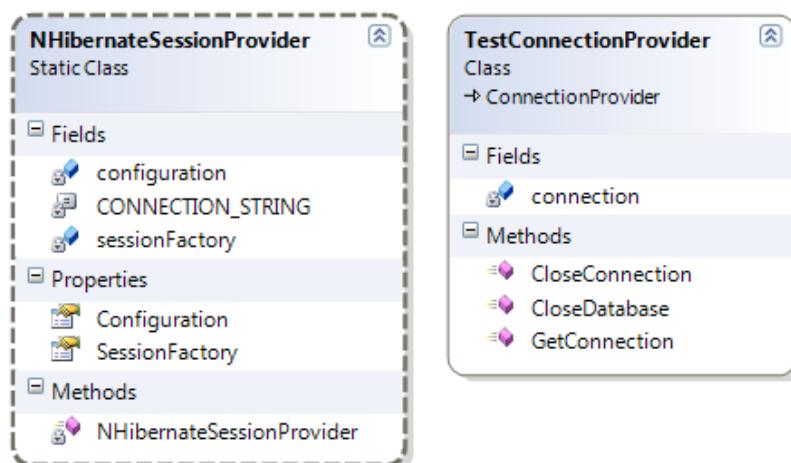
Implementacija informacijskog sustava za upravljanje stambenim zgradama

Temeljem tih osnovnih NHibernate implementacija repozitorija ostvarene su ostale domenski specifične NHibernate implementacije repozitorija. Upit na bazu u većini repozitorija ostvaren je korištenjem HQL-a i QueryOver NHibernate upita.

Testiranje repozitorija ostvareno je pomoću NUnit okvira za testiranje i SQLite baze podataka čiji se podaci čuvaju u memoriji. SQLite baza podataka odabrana je iz razloga što je lagana i ne zahtijeva preveliku konfiguraciju, omogućava čuvanje podataka u memoriji čime se povećava brzina što je izrazito bitno kada govorimo o unit testovima. Cilj takvih testova je provjeriti ispravnost specificiranih mapiranja i mogućnost obavljanja osnovnih operacija repozitorija kao što je spremanje i dohvaćanje entiteta.

Za podršku testiranju na SQLite bazi podataka kreiran je `NHibernateSession Provider` razred. Razred učitava NHibernate konfiguraciju iz glavne NHibernate konfiguracije definiranu u glavnom projektu te ujedno postavlja SQL dialect, driver, pružatelja konekcije i connection string specifične za SQLite memorijsku bazu. Nakon kreiranja konfiguracije kreira se tvornica sjednica.

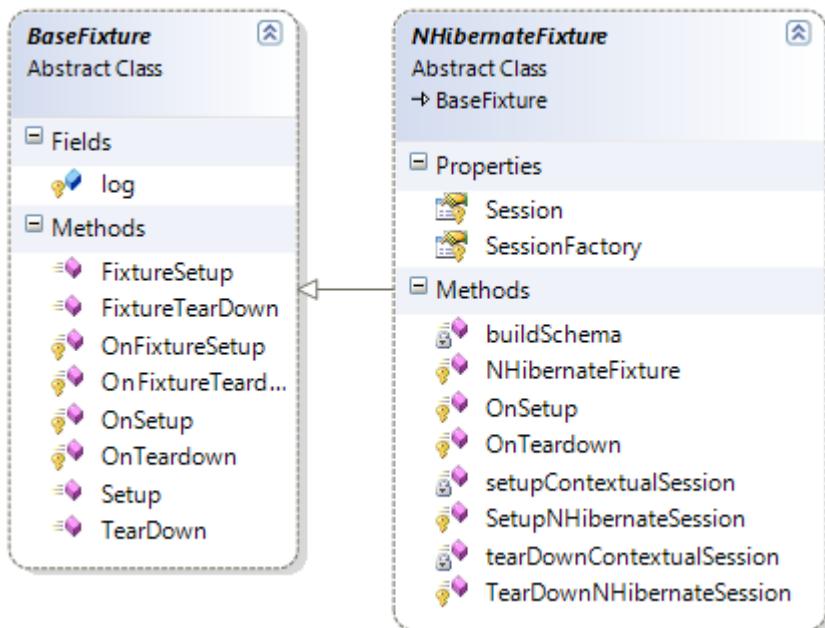
Ključni razred koji omogućuje testiranje repozitorija je `TestConnection Provider` koji se nasljeđuje iz `ConnectionProvider` razreda i predstavlja pružatelja konekcije. Tipično pružatelj konekcije vraća novu konekciju svaki put kada se pozove `GetConnection()` metoda, a zatvara ju kada se pozove `CloseConnection()` metoda. Međutim, SQLite memorijska baza podataka podržava samo jednu konekciju, odnosno za svaku novu konekciju kreira se nova baza podataka u memoriji. Kada se konekcija zatvori uništava se baza podataka.



Slika 39 - Pružatelji NHibernate sjednice i konekcije na bazu podataka

Pri pokretanju svakog testa, zatvaraju se sve aktivne konekcije čime osiguravamo da test radi sa novom i praznom bazom podataka. Kada NHibernate prvi puta pozove metodu `GetConnection()`, otvara se nova konekcija. Ista sjednica se vraća za svaki ponovni poziv te metode. Pozivi metode `CloseConnection()` se ignoriraju. Konačno kada test završi konekcija baze podatka se uništava čime se i oslobođa memorija za SQLite bazu podataka. Takav način pruža čistu bazu podataka za svaki test čime se osigurava da svaki prethodni test ne utječe na trenutni.

U razredu `BaseFixture` konfigurira se log4net okvir za upravljanje logovima i postavljaju se virtualne metode dekorirane NUnit atributima koje se mogu nadjačati u naslijedenim razredima. Razred `NHibernateFixture` nasljeđuje se iz `BaseFixture` razreda te implementira `OnSetup()` metodu koja se izvodi prilikom svakog testa. Unutar metode `OnSetup()` otvara se sjednica koja se asocira sa trenutnim kontekstom te se ujedno automatski izrađuje shema baze podataka. Navedene akcije otvaraju konekciju na bazu podataka čime se ujedno stvara nova baza podataka u memoriji. Implementacijom metode `OnTeardown()` koja se izvodi nakon svakog testa uklanja se asocijacija sjednice sa trenutnim kontekstom, zatvara se sjednica i konekcija na bazu, čime se oslobođa memorija namijenjena bazi podataka.



Slika 40 - Podupirući razredi testiranju NHibernate repozitorija

6.3.3 Autentifikacija i autorizacija

ASP.NET MVC se zasniva na ASP.NET web razvojnoj okolini koja donosi autentifikaciju putem web forme odnosno *Forms Authentication*. Forms Authentication koristi *cookies* Internet preglednika kako bi se autentifikacijski status prenio iz jednog web zahtjeva u drugi. Podatke iz web forme potrebno je provjeriti nad nekim oblikom korisničkih podataka. Od ASP.NET-a verzije 2.0 uvedena je standardna infrastruktura korisničkih računa predstavljena preko nekoliko dijelova programskih sučelja (API):

- *Membership* – zadužen za registraciju korisničkih računa i pristup njihovim podacima poput korisničkog imena i lozinke, točnije autentifikaciju.
- *Roles* – zadužen za definiranje i postavljanje uloga registriranim korisnicima, točnije autorizaciju.
- *Profiles* – omogućuje spremanje proizvoljnih podataka o korisniku.

Implementacija pojedinog programskog sučelja ostvarena je kroz pojam pružatelja (engl. *provider*). Svaki taj pružatelj je odgovoran za svoje podatkovno spremište. ASP.NET web razvojni okvir dolazi sa standardnim pružateljima koji spremaju podatke u Microsoft SQL Server u već predefinirane podatkovne sheme.

Prednosti korištenja predefiniranih programskih sučelja vezanih za Membership, Roles i Profiles jesu: dobra integracija sa ostatkom ASP.NET razvojne okoline, dobro razrađen i dizajniran model upravljanja korisnicima, iskoristivost na drugim ASP.NET baziranim web aplikacijama.

Međutim problemi koji se javljaju vezani su za ugrađene SQL programske knjižnice koje zahtijevaju direktni pristup bazi podataka što onečišćuje model domene i otežava korištenje objektno – relacijske tehnologije. Ugrađene SQL programske knjižnice zahtijevaju specifičnu podatkovnu shemu koju nije jednostavno uklopiti u podatkovnu shemu ostatka aplikacije.

Upravo zbog ovakvih razloga odlučili smo se za korištenje Membership i Role ASP.NET API-a sa vlastitom implementacijom orijentiranoj k NHibernateu. Profile API odlučili smo ne podržati jer nije bilo potrebe za njim.

Da bismo ostvarili vlastitu Membership implementaciju potrebno je da naš razred nasljeđuje bazni apstraktни razred `MembershipProvider`. Navedeni apstraktni razred

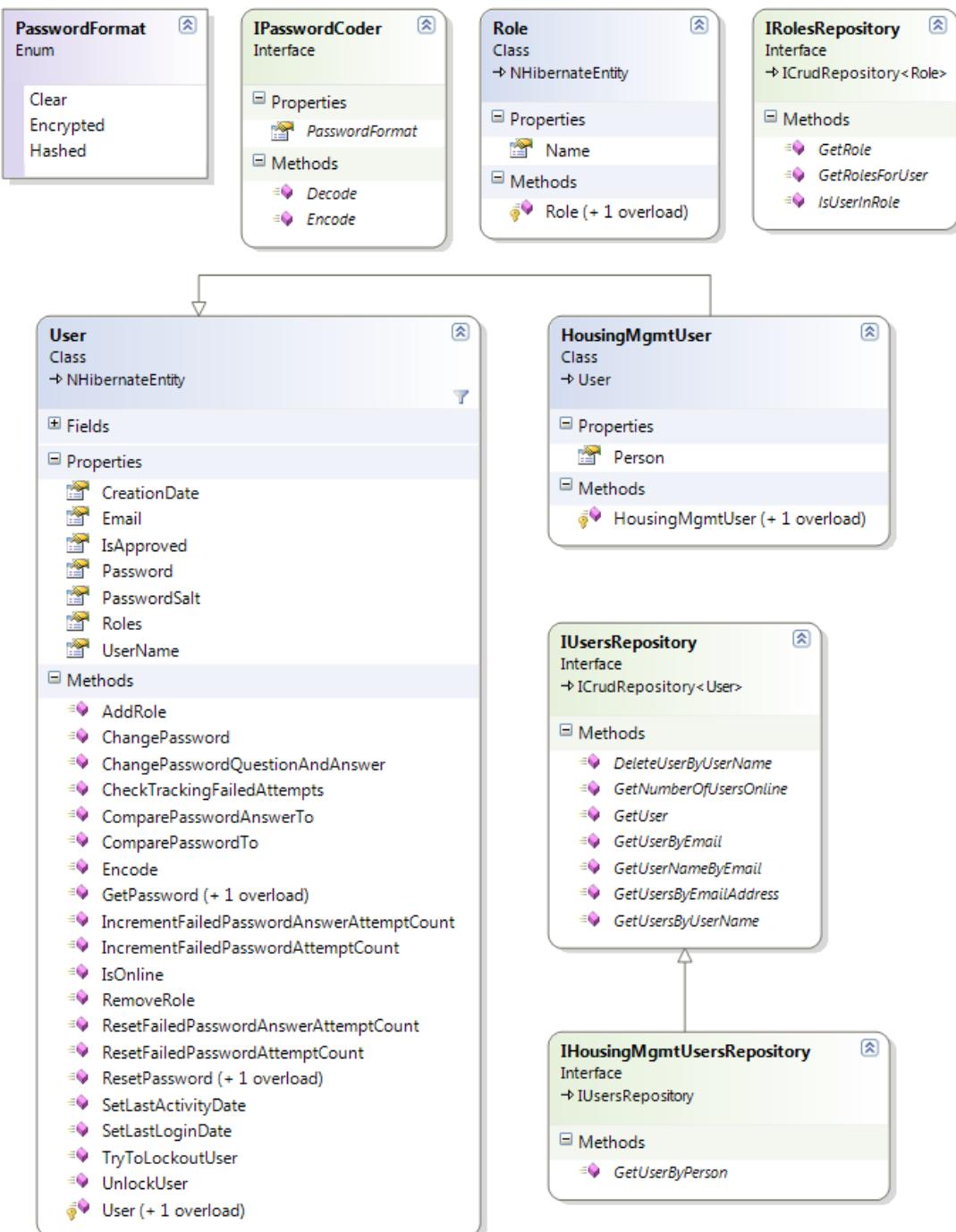
definira sučelje čija implementacija omogućuje jednostavnu integraciju u ASP.NET razvojni okvir. Glavnu metodu koju je potrebno implementirati je metoda `ValidateUser()`, koja predstavlja temelj za uspješnu autentifikaciju korisnika. Slična stvar vrijedi za implementaciju uloga, dakle potrebno je da se naš razred uloga nasljeđuje iz apstraktnog razreda `RoleProvider` koji također definira sučelje koje je potrebno implementirati. Glavnu metodu koju je potrebno implementirati u razredu uloga je `GetRolesForUser()` što čini temelj uspješne autorizacije.

Kako je naš cilj implementacija zasnovana na NHibernateu, prvo smo definirali entitet `User` koji sadrži podatke i metode slične onima definiranim kroz sučelje `MembershipProvider`. Jedan od algoritama za kodiranje lozinke duboko je ugrađen u ASP.NET razvojnu okolinu stoga smo taj mehanizam apstrahirali kroz sučelje `IPasswordCoder`. Sučelje `IPasswordCoder` je parametar određenih metoda entiteta `User` koje rade sa lozinkom. Problematika korisničkih računa spada u posebnu domenu koja je odvojena od domene upravljanja stambenim zgradama. Kako bi te dvije domene međusobno povezali kreiran je razred `HousingMgmtUser` naslijeden iz `User` razreda, gdje taj razred čini kompoziciju sastavljenu od razreda `Person` koji čini korijenski agregat domene za upravljanje stambenim zgradama.

Razred `HousingMgmtUser` nastao je specijalizacijom iz razreda `User` stoga je specifikacija mapiranja definirana u okviru strategije „Tablica za svaku hijerarhiju nasljeđivanja“, odnosno denormalizacija relacijskog modela koristeći posebni stupac za čuvanje informacija o tipu.

Uz entitet korisnika definiran je entitet `Role` koji predstavlja ulogu korisnika sa pripadajućim rezervorijem. Entitet nema nikakvo ponašanje te je predstavljen isključivo svojim nazivom. Svakom entitetu `User` moguće je dodijeliti više uloga čime entitet `Role` čini dio agregata korisnika.

Implementacija informacijskog sustava za upravljanje stambenim zgradama



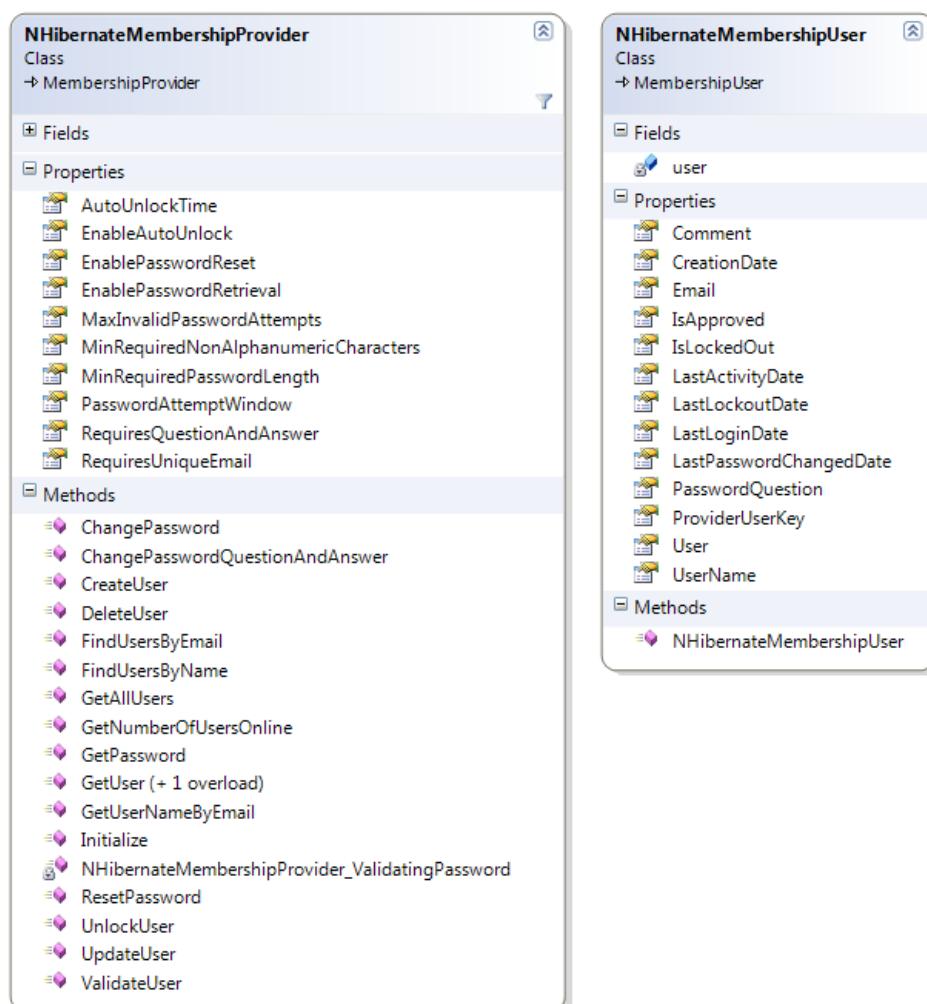
Slika 41 - Entiteti i rezervorij korisničkog računa

Nakon što smo definirali entitete koji sačinjavaju domenu korisničkih računa ostvarili smo implementaciju njihovih rezervorija pomoću NHibernatea i to:

- UsersNHRepository – NHibernate implementacija rezervorija korisnika
- HousingMgmtUsersNHRepository – NHibernate implementacija specijaliziranog korisnika u kontekstu upravljanja stambenim zgradama
- RolesNHRepository – rezervorij uloga.

Implementacijom repozitorija domene korisničkih računa i entitetima nad kojima repozitoriji rade stvorili smo sve preduvjete na implementaciju NHibernate Membership knjižnice. Prethodno smo spomenuli da metode User entiteta vrlo vjerno preslikavaju MembershipProvider sučelje što nam olakšava samu integraciju u Membership knjižnicu. Pozivom metoda definiranim MembershipProvider sučeljem delegiraju se operacije perzistencije repozitoriju te ostale domenske operacije User entitetu.

Svaki MembershipProvider kao rezultat vraća MembershipUser objekt koji predstavlja reprezentaciju korisnika, točnije samo one informacije koje su relevantne autentifikaciji korisnika. Radi povećanja dostupnih informacija o korisniku implementiran je NHibernateMembershipUser razred koji osim standardnih informacija o korisniku uvodi informacije o osobi koju on predstavlja. NHibernateMembershipUser nasljeđuje se iz MembershipUser razreda.

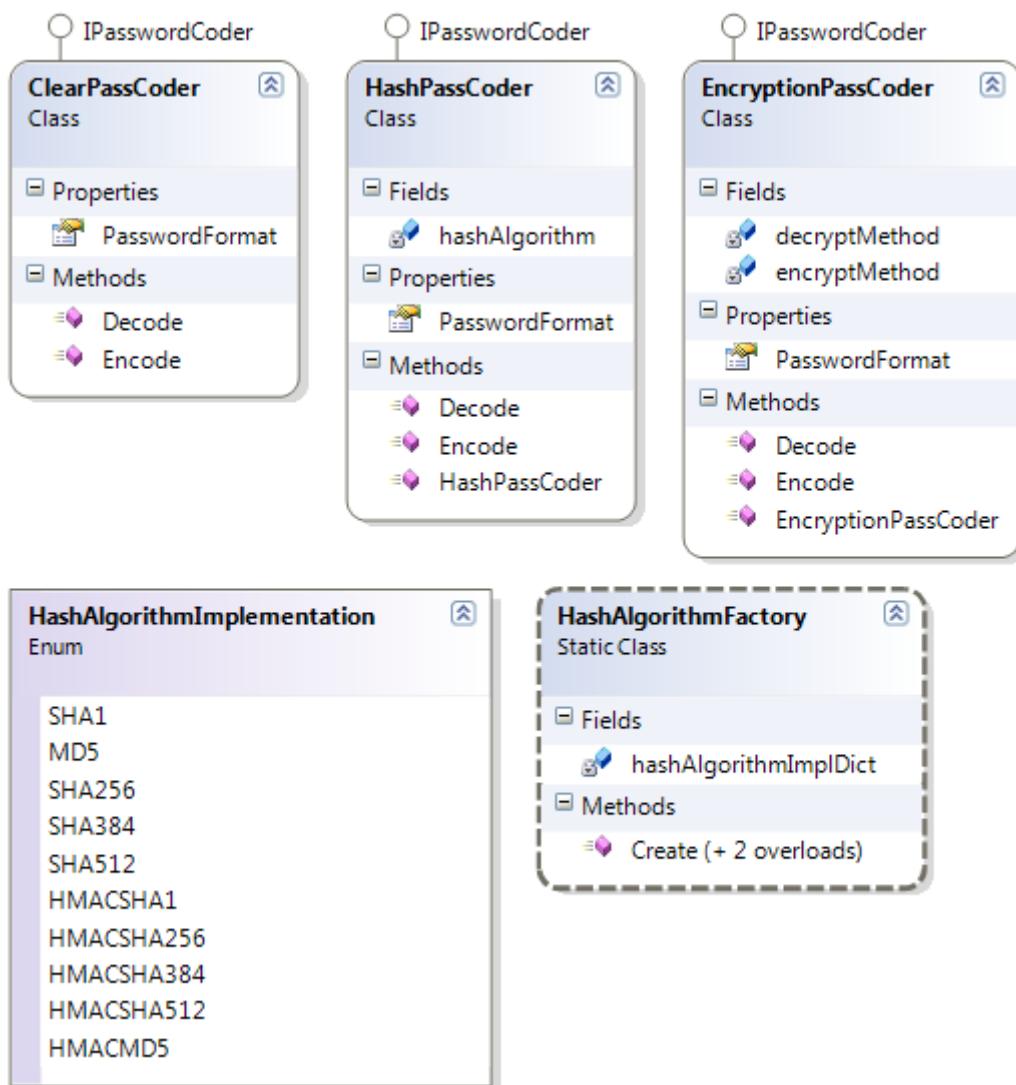


Slika 42 - Vlastita implementacija MembershipProvidera

Implementacija informacijskog sustava za upravljanje stambenim zgradama

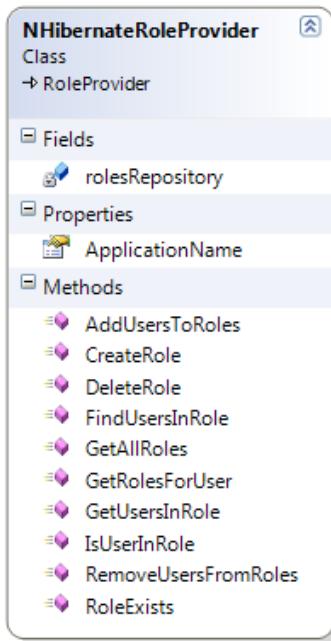
Kao implementaciju sučelja `IPasswordCoder` za kodiranje lozinke ostvarena su tri razreda čije strategije su prirodno podržane u osnovnoj implementaciji `MembershipProvider` razreda:

- `ClearPassCoder` – lozinka je predstavljena čistim tekstom
- `HashPassCoder` – u ovisnosti o izabranom algoritmu sažetka generira se sažetak poruke (`HashAlgorithmFactory` je tvornica zadužena za instanciranje odgovarajućeg hash algoritma)
- `EncryptionPassCoder` – temeljem privatnog ključa lozinka se kriptira



Slika 43 - Implementacija `IPasswordCoder`a

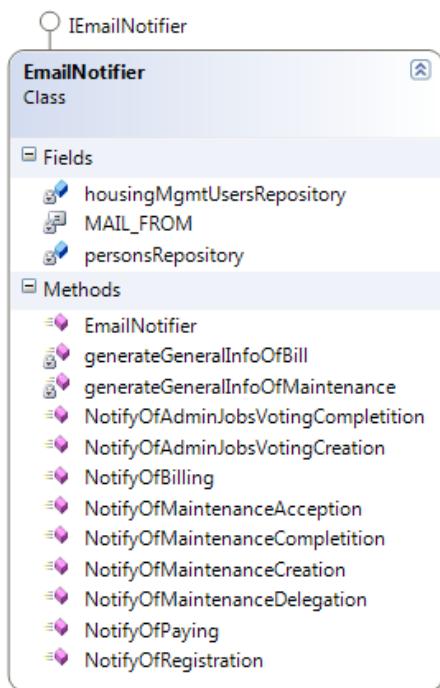
Za vlastitu implementaciju pružatelja uloga ostvaren je `NHibernateRole Provider`. Navedena implementacija zasniva se na repozitoriju uloga.



Slika 44 - NHibernate implementacija pružatelja uloge

6.3.4 Servisi

Od infrastrukturnih servisa implementiran je `EmailNotifier` čije je svrha slanje obavijesti ovisno o scenarijima iz modela domene. `EmailNotifier` definiran je u prvome redu kroz sučelje `IEmailNotifier` koje definira događaje u kojima je potrebno obavijestiti korisnike informacijskog sustava.



Slika 45 - Infrastrukturni servis za slanje obavijesti putem e-maila

6.4 Implementacija aplikacijskog i prezentacijskog sloja

Nakon dizajna modela domene i implementacije infrastrukture koja se bavi persistencijom stanja modela domene potrebno je ostvariti aplikacijski i prezentacijski sloj koji će biti zadužen za koordinaciju aktivnosti poslovnih procesa te odgovoran za prezentaciju informacija korisniku i interpretaciju korisničkih naredbi.

Prema zahtjevima informacijskog sustava taj sloj je definiran kao web aplikacija, točnije korištenjem ASP.NET MVC 3 web razvojnog okvira.

Koordinacija aktivnosti poslovnih procesa i interpretacija korisničkih naredbi ostvarena je pomoću nadglednika koji rade sa konceptima iz domene dok za prezentaciju informacija su zaduženi pogledi i modeli pogleda (engl. *ModelView*).

Za konfiguraciju NHibernatea za pristup bazi podataka najčešće korištena je hibernate.cfg.xml datoteka.

6.4.1 Nadglednici (engl. controllers)

Interakcija korisnika i informacijskog sustava dešava se posredovanjem nadglednika (engl. *controller*). Zahtjevi korisnika se preslikavaju u pozive akcija nadglednika. Akcije nadglednika su zadužene za koordinaciju aktivnosti aplikacije. Točnije akcije nadglednika ovisno o poslovnom procesu koji rješavaju dohvaćaju entitete, odnosno korijenske aggregate iz repozitorija, obavljaju promjene nad modelom domene. Nastale promjene se persistiraju pomoću repozitorija, a rezultati se prikazuju korisniku u obliku html prezentacije.

Svi nadglednici ovog informacijskog sustava definiraju ovisnosti o repozitorijima i aplikacijskim servisima preko parametara konstruktora. Ti parametri definirani su u obliku ovisnosti o sučeljima. U provođenju takvog pristupa odgovornost za instanciranje repozitorija i aplikacijskih servisa predana je izvan dosega samog nadglednika. Drugim riječima prakticirao se pristup IoC (engl. *inversion of control*).

U tu svrhu korištena je programska knjižnica *Ninject* za MVC koja predstavlja spremnik sa svim ovisnostima. Princip rada *Ninject* knjižnice je slijedeći:

- Definiraju se asocijacije između apstrakcija, koje su definirane preko sučelja, sa konkretnim implementacijama. Stoga kada neki razred ovisi o drugim servisima

definiranim kroz sučelja, temeljem tog sučelja *Ninject* spremnik zna koju implementaciju mora injektirati da zadovolji ugovor.

- Prilikom prvog pokretanja aplikacije instancira se *Ninject* objekt koji sadrži sve asocijacije i osigurava instanciranje konkretnih implementacija.
- Kada se pošalje http zahtjev i iz zahtjeva se odredi koji nadglednik treba instancirati, tvornica zadužena za instanciranje nadglednika traži *service locator* da pribavi potrebne ovisnosti koje su potrebne da bi se nadglednik uspješno instancirao. *Service locator* se pak oslanja na *Ninject* instancu.

Na takav način nadglednik dobiva se potrebne instancirane ovisnosti poput repozitorija i aplikacijskih servisa.

Akcije nadglednika obavljaju poslovne operacije isključivo sa elementima domene, odnosno sa modelom domene. Često puta model domene je presložen za prezentaciju u pogledu, stoga se elementi domene preslikavaju u jednostavnije modele pogleda. Nadglednik je po tom pitanju odgovoran da se složeni elementi domene preslikaju u jednostavnije modele koji se zatim predaju pogledu na prikaz.

Implementirani su sljedeći nadglednici:

- `AccountController` – nadglednik koji omogućuje operacije nad korisničkim računima.
- `BuildingManagementController` – nadglednik koji omogućuje operacije koje se tiču upravljanja stambenim zgradama.
- `BuildingManagerController` – nadglednik koji je usko vezan za specifične operacije upravitelja zgrade.
- `ContractorController` – nadglednik koji je usko vezan za specifične operacije izvođača radova.
- `DashboardController` – nadglednik koji prikazuje sažetak informacija vezanih za upravljanje stambenim zgradama.
- `FinancesController` – nadglednik koji omogućuje operacije vezane za financije zgrade.
- `LegislatureController` – nadglednik koji omogućuje operacije nad zemljišnim knjigama.

- MaintenanceController – nadglednik koji omogućuje operacije vezane za održavanje stambene zgrade.
- OwnerController – nadglednik koji je usko vezan za specifične operacije suvlasnika stambene zgrade.
- PersonController – nadglednik koji prikazuje informacije o osobama.

6.4.2 Modeli pogleda

Elementi modela domene često puta su previše kompleksni da bi se direktno koristili u prikazu informacija korisniku. Podaci potrebni za generiranje pogleda mogu se sastojati od velikog broja raznovrsnih podataka koji nisu nužno dio domene. Iz tih se razloga definiraju modeli pogleda. Za svaki pogled definiran je jedan poseban razred koji sadrži sve informacije koje će se prezentirati u tom pogledu. Ti razredi ne sadrže nikakvo ponašanje već samo podatke.

Dodatni problem koji se javlja u prikazu objekata domene u pogledu je doseg NHibernate sjednice i lijeno dohvaćanje entiteta. Prethodno smo rekli da je životni ciklus sjednice definiran kroz vrijeme izvođenja akcije te da je asocijacija sa bazom podatka ostvarena sve dok traje sjednica. Kada bi pokušali pristupiti kolekciji elemenata u trenutku kada je sjednica zatvorena desilo bi se lijeno dohvaćanje koje bi za posljedicu imalo iznimku. Drugim riječima, lijeno dohvaćanje nije moguće ukoliko je sjednica zatvorena. Taj problem postoji ukoliko bi element iz domene predali pogledu za prikaz. Evaluacija prikaza obavlja se nakon izvršavanja akcije nadglednika i moglo bi se desiti da će pogled pokušati pristupiti kolekciji ili drugim elementima entiteta koji nisu odmah dohvaćeni što za posljedicu ima grešku.

Kao rješenje savršeno se uklapa model pogleda koji preslikava samo potrebne podatke iz modela domene. To preslikavanje podataka obavlja se za vrijeme trajanja akcije nadglednika čime se osigurava da svi potrebni podaci budu dohvaćeni za vrijeme trajanja sjednice.

Umjesto ručnog preslikavanja podataka iz modela domene u model pogleda koriste se gotove programske knjižnice za mapiranje. U ovome radu koristili smo AutoMapper. Prilikom pokretanja informacijskog sustava obavlja se konfiguracija AutoMappera u kojoj je definirano koji razred iz domene se preslikava u koji razred modela pogleda i što je sve

potrebno preslikati. Kada AutoMapper zna kako preslikati određene instance spreman je za korištenje u akciji nadglednika.

6.4.3 Pogledi

Pogledi predstavljaju prikaz informacija korisnicima informacijskog sustava. Za svaku akciju nadglednika definiran je pripadajući pogled. U načelu pogled se sastoji od deklarativnog HTML koda i programskog koda isključivo sa svrhom ispisa podataka. Svaki pogled se ugrađuje u zajednički obrazac prikaza.

7. Upute za korištenje informacijskog sustava za upravljanje stambenim zgradama

U ovome poglavlju dati ćemo upute za postavljanje informacijskog sustava kako bi bio spreman za rad. Informacijski sustav zahtjeva Microsoft SQL Server 2008 bazu podataka te minimalno IIS 7.5 web poslužitelj. Isto tako dati ćemo osnovne upute korisnicima za korištenje sustava.

7.1 Upute za postavljanje sustava

Informacijski sustav za upravljanje stambenim zgradama predan je u obliku Visual Studio 2010 Solutiona uz SQL skripte za kreiranje baze podataka i već kreirane baze podatka za demonstraciju. Baza podataka korištena za informacijski sustav je Microsoft SQL Server 2008. Prije samog pokretanja aplikacije potrebno je kreirati bazu podatka koristeći priložene SQL skripte ili iskoristiti već kreiranu i priloženu bazu podataka. Kreiranjem baze podataka pomoću SQL skriptnih datoteka dobiva se prazna baza podataka sa formiranim tablicama i potrebnim podacima, dok se sa već kreiranom bazom dobivaju kreirani prezentacijski primjeri zgrada, suvlasnika, upravitelja i izvođača radova.

U direktoriju `source/database/sql_scripts` nalaze se dvije SQL skriptne datoteke koje je potrebno izvesti slijedećim redoslijedom:

- `database_creation.sql` – sadrži SQL za kreiranje baze podataka i schema.
- `table_creation.sql` – sadrži SQL naredbe za kreiranje tablica baze podataka sa odgovarajućim podacima

Već kreirana baza podataka nalazi se u direktoriju `source/database` u obliku dvije datoteke: `HousingMgmt.mdf` i `HousingMgmt_log.ldf`. Navedene datoteke potrebno je pridružiti sustavu za upravljanje bazama podataka i to pomoću Microsoft SQL Server Management Studioa.

Kao što je već spomenuto informacijski sustav dolazi u izvornome kodu, odnosno u obliku VS 2010 Solutiona. Za pokretanje aplikacije potrebno je otvoriti Solution i pokrenuti Debug ili pokrenuti Build te objaviti, pomoću alata Visual Studia, aplikaciju na IIS poslužitelj (IIS poslužitelj mora biti minimalno od verzije 7.5 na više).

Prije samog pokretanja aplikacije potrebno je postaviti informacije za spajanje na bazu podataka u datoteci web.config.

7.2 Korisničke upute

U nastavku opisati ćemo osnovne dijelove korisničkog sučelja te glavne ekrane za manipulaciju i prikaz podataka. U same poslovne procese i uloge nećemo detaljno ulaziti jer su oni definirani u poglavlju 5.3 *Slučajevi korištenja* te ih je moguće provesti sa danim korisničkim uputama.

7.2.1 Osnovni dijelovi korisničkog sučelja

Postoje tri glavna područja korisničkog sučelja:

1. **Korisničke opcije** – korisniku je omogućena odjava, te promjena korisničkih podataka kao što je promjena lozinke, adrese prebivališta itd.
2. **Glavni izbornik** – dijelovi glavnog izbornika određeni su ulogama prijavljenog korisnika. Glavni izbornik ima definirane dvije razine poveznica. U prvoj razini (pričinjano na sljedećoj slici) prikazane su poveznice koje su usko povezane sa prijavljenim korisnikom ovisno o njegovoj ulozi. Drugim riječima, prikazane su one poveznice koje vode na određene stranice koje nisu određene specifičnom zgradom ili stanom, već sveukupno. Tako se primjerice za poveznicu „Moji računi“ svlasnika prikazuju izdani računi pričuve svih stanova u vlasništvu prijavljenog korisnika. U drugoj razini glavnog izbornika prikazane su poveznice koje vode na specifične stranice vezane za određenu zgradu ili stan.
3. **Područje za sadržaj** – centralno mjesto na kojem se prezentira glavni sadržaj.

Naslovna stranica prikazuje posljednje dodane zgrade, stanove ili popravke za svaku postojeću ulogu prijavljenog korisnika. Klikom na poveznicu više dobiva se kompletan lista stanova, zgrada ili popravaka.

Princip po kojemu se sadržaj prikazuje slijedi *master – detail* logiku, tj. preko stranice sa listom elemenata dolazi se na stranicu sa detaljima tog elementa.



Slika 46 - Osnovni dijelovi korisničkog sučelja

7.2.2 Prijava na sustav

Registracija upravitelja ili izvođača radova

Ukoliko ste upravitelj ili izvođač radova, a nemate otvoren korisnički račun možete se registrirati na sljedećim linkovima:

- [Registracija suvlasnika](#)
- [Registracija upravitelja](#)
- [Registracija izvođača radova](#)

Prijava na sustav

Molimo vas da unesete svoje korisničko ime i lozinku.

Podaci za prijavu	
Korisničko ime	<input type="text"/>
Lozinka	<input type="password"/>
<input type="checkbox"/> Zapamti me?	
<input type="button" value="Prijava"/>	

Slika 47 - Prijava korisnika i početna stranica

Početna stranica sustava prikazuje formu za prijavu postojećih korisnika na sustav. Osim forme za unos korisničkih podataka registriranih korisnika, postoje poveznice za registraciju ne registriranih korisnika. Nakon uspješne prijave korisniku se prikazuje stranica „Naslovnica“ opisana u prethodnom poglavlju.

7.2.3 Registracija korisnika

Registrirati se mogu suvlasnici, upravitelji i izvođači radova. Dolazak na stranicu sa registracijama moguć je odabirom poveznice na početnoj stranici ukoliko korisnik već nije prijavljen u sustav. Na slijedećoj slici prikazana je web forma za registraciju upravitelja. Slične web forme za registraciju postoje za suvlasnike i izvođače radova.

Registracija upravitelja

Iskoristite formu ispod za registraciju upravitelja.

Lozinka treba biti minimalno dugačka 6 znakova.

Podaci o korisničkom računu

Korisničko ime

Email adresa

Lozinka

Ponovi lozinku

Podaci o pravnoj osobi

Ime pravne osobe

OIB

Broj bankovnog računa

Ulica

Broj

Grad

 --Odaberi ▾

Telefon

Mobitel

Registriraj se

Slika 48 - Primjer stranice za registraciju upravitelja

7.2.4 Zgrade i zemljишne knjige

Kreiranje zgrade započinje klikom na poveznicu „Nova stambena zgrada“ koji se nalazi na stranici sa listom zgrada kojima upravitelj upravlja.

Dobrodošli **upravitelj01!** [[Uredi](#)] [[Odjava](#)]

NASLOVNICA

UPRAVITELJ

Izvođači radova

Računi

Nova stambena zgrada

Podaci o stambenoj zgradi

Ulica	Broj	Grad
Ilica	123	Zagreb

Spremi

Slika 49 - Kreiranje stambene zgrade

Upute za korištenje informacijskog sustava za upravljanje stambenim zgradama

Nakon kreiranja zgrade potrebno je kreirati zemljišnu knjigu. Za kreiranje zemljišne knjige potrebno je ispuniti sljedeći obrazac.

NASLOVNICA
« UPRAVITELJ
ZGRADA
Rad uprave
Kvarovi
Upraviteljevi računi
Pričuva
Izdani računi pričuve

Nova zemljišna knjiga

Podaci o zemljišnoj knjizi

Grad katastarske općine: Zagreb Katastar: Črnomerec

Broj katastarske čestice: 4037/15 Površina [m²]: 1365

Opis:
Stambena zgrada Ilica br. 123 i dvorište.

Spremi

Slika 50 - Kreiranje zemljišne knjige

Po kreiranju zemljišne knjige prikazuje nam se ekran kao na sljedećoj slici. Ekran, odnosno stranica sadrži sve informacije o zemljišnoj knjizi sa popisom vlasničkih i zajedničkih etaža.

NASLOVNICA
« UPRAVITELJ
ZGRADA
Zemljišna knjiga
Rad uprave
Kvarovi
Upraviteljevi računi
Pričuva
Izdani računi pričuve

Zemljišna knjiga za katastarsku česticu: 4037/15

Zaključaj zemljišnu knjigu

Katastarska čestica

Broj katastarske čestice: 4037/15 Katastar: Črnomerec (335266), Zagreb

Površina: 1.365,00 m²

Opis: Stambena zgrada Ilica br. 123 i dvorište.

Etaže

Ukupna površina zajedničkih etaža: 97,34 m² Ukupna površina vlasničkih etaža: 926,77 m²

Ukupna površina etaža: 1024,11 m²

Zajedničke etaže:
Dodaj zajedničku etažu

Redni broj	Površina [m ²]	Opis
1	30,00	Biciklarna
2	25,34	Alatnica
3	42,00	Soba za sastanke

Vlasničke etaže:
Dodaj vlasničku etažu

Redni broj	Površina [m ²]	Omjer	Opis
4	34,15	0,0368	Jednosoban stan u prizemlju zgrade, ukupne površine 34.15 m2.
5	32,45	0,0350	Jednosoban stan u prizemlju zgrade, ukupne površine 32.45 m2.
6	46,23	0,0499	Dvosoban stan u prizemlju zgrade, ukupne površine 46.23 m2. Stan uključuje jednu dnevnu sobu, jednu spavaču sobu, kupuoniku, hodink, ...

Slika 51 - Zemljišna knjiga

Oblikovanje IS-a za upravljanje stambenim zgradama vođeno domenom primjene

Samo upravitelj ima mogućnost dodavanje novih etaža sve dok zemljišna knjiga nije zaključana.

NASLOVNICA
« UPRAVITELJ
ZGRADA
Zemljišna knjiga
Rad uprave
Kvarovi
Upraviteljevi računi
Pričuva
Izdani računi pričuve

Dodavanje zajedničke etaže u zemljišnu knjigu: 4037/15

Podaci o zajedničkoj etaži

Broj uloška 17290	Broj poduloška 1
Površina [m ²] 30	
Opis Biciklarna	

Dodaj

Slika 52 - Dodavanje etaže u zemljišnu knjigu

NASLOVNICA
« UPRAVITELJ
ZGRADA
Zemljišna knjiga
Rad uprave
Kvarovi
Upraviteljevi računi
Pričuva
Izdani računi pričuve

Zgrada: Ilica 123, 10000 Zagreb

Podaci o zgradbi

Zemljišna knjiga: 4037/15	Katastar: Črnomerec (335266), Zagreb
Adresa: Ilica 123, 10000 Zagreb	Površina: 1365,00 m ²
Opis: Stambena zgrada Ilica br. 123 i dvorište.	
Predstavnik suvlasnika: Suvlasnik Jedan [Uredi]	Upravitelj zgrade: Upravitelj Jedan d.o.o.
Koefficijent pričuve: 2,69 kn/m ²	Pričuva: 253,66 kn [Detalji]

Rad uprave:

Tema	Vrsta	Početak	Kraj	Glasalo	Završeno	Prihvaćeno
Izmjena ulaznih vrata	Izvanredna	22.5.2011.	25.5.2011.	1 / 12	Ne	Ne
Ličenje zidova zgrade	Regularna	22.5.2011.	23.5.2011.	2 / 12	Ne	Ne

[Više ...](#)

Prostorije i stanovi

Zajedničke etaže:

Etaža	Opis	Površina [m ²]
1	Biciklarna	30,00
2	Alatnica	25,34
3	Soba za sastanke	42,00

Vlasničke etaže:

Etaža	Vlasnik	Opis	Površina [m ²]
4	Suvlasnik Jedan	Jednosoban stan u prizemlju zgrade, ukupne površine 34,15 m ² .	34,15

Slika 53 - Detalji zgrade

Upute za korištenje informacijskog sustava za upravljanje stambenim zgradama

Na stranici „Zgrada“ moguće je vidjeti detalje zgrade koji se sastoje od podataka iz zemljišne knjige, predstavnika suvlasnika, upravitelja, svih etaža i rada uprave.

7.2.5 Rad uprave

Predstavnik suvlasnika može kreirati rad uprave za odabranu zgradu. Za rad uprave predstavnik odabire vrstu uprave, datum do kojega je moguće glasovati za rad uprave, temu i opis.

NASLOVNICA
« PRESTAVNIK SUVLASNIKA
ZGRADA
Zemljišna knjiga
Rad uprave
Kvarovi
Pričuva
Izdani računi pričuve

Novi rad uprave

Podaci o radu uprave

Vrsta uprave: Rok:
Redovna 23.05.2011

Tema rada uprave
Ličenje zidova zgrade

Opis rada uprave
Zbog dotrajalosti boje na zidovima odlučili bi se na radeve bojanja i ličenja zidova zgrade. Molimo glasajte ako ste za ili protiv.

Spremi

Slika 54 - Kreiranje rada uprave

Svaki suvlasnik određene zgrade može glasovati za sve poslove uprave. Svoj glas daje tako da se izjašnjava pozitivno ili negativno putem forme prikazane na sljedećoj slici.

 Ne: '. A 'Glasaj' button is at the bottom."/>

NASLOVNICA
« SUVLASNIK
STAN
Rad uprave
Moji kvarovi
Kvarovi
Moji računi
Pričuva

Glasaj za: Izmjena ulaznih vrata

Podaci o radu uprave

Tema rada uprave
Izmjena ulaznih vrata

Opis rada uprave
Zbog dotrajalosti ulaznih vrata potrebna je zamjena. Nova vrata bila bi izrađena od PVC-a i bijele boje. Molim da se izjasnite.

Glas
Da: Ne:

Glasaj

Slika 55 - Glasovanje za rad uprave

7.2.6 Kvarovi

Kvarove pojedine zgrade mogu objaviti svi suvlasnici te upravitelj zgrade. Prijava kvara sastoji se od definiranja lokacije kvara u zgradi, odabira vrste popravka, naziva i opisa kvara te definiranja hitnosti.

Prijava kvara

Podaci o kvaru

Zgrada: Ilica 123, 10000 Zagreb Lokacija: Glavni ulaz

Vrsta popravka: Održavanje rasvjete i drugih električnih uređaja

Kvar: Neispravan rad portafona

Hitnost: Normalna

Opis kvara:
Na ulazu u zgradu ne radi portafon što se manifestira kroz nemogućnost udaljenog otključavanja vrata.

Kreiraj

Slika 56 - Prijava kvara

Na stranici sa kvarovima prikazuje se lista kvarova zgrade podijeljena prema statusima kvarova. Tako razlikujemo nove kvarove, odnosno kvarove koji čekaju na obradu, kvarove u obradi, kvarove koji čekaju na potvrdu od strane predstavnika suvlasnika te riješene kvarove.

Upute za korištenje informacijskog sustava za upravljanje stambenim zgradama

The screenshot shows a navigation menu on the left with the following items:

- NASLOVNICA
- « PREDSTAVNIK SUVLASNIKA
- ZGRADA
- Zemljišna knjiga
- Rad uprave
- Kvarovi** (highlighted)
- Pričuva
- Izdani računi pričuve

On the right, there are several sections:

- Kvarovi za zgradu: Ilica 123, 10000 Zagreb**
 - [Prijava kvar](#)
 - Novi kvarovi**

Kvar	Zgrada	Vrsta kvara	Hitnost	Status
Neispravan rad portafona	Ilica 123, 10000 Zagreb	Održavanje rasvjete i drugih električnih uređaja	Normalna	Nije započeto
 - Kvarovi u obradi**

Kvar	Zgrada	Vrsta kvara	Hitnost	Status
Dizalo ne radi	Ilica 123, 10000 Zagreb	Servis dizala	Visoka	Čeka na potvrdu
 - Kvarovi koji čekaju na potvrdu**

Kvar	Zgrada	Vrsta kvara	Hitnost	Status
Dizalo ne radi	Ilica 123, 10000 Zagreb	Servis dizala	Visoka	Čeka na potvrdu
 - Riješeni kvarovi**

Kvar	Zgrada	Vrsta kvara	Hitnost	Status
Dizalo ne radi	Ilica 123, 10000 Zagreb	Servis dizala	Visoka	Čeka na potvrdu

Slika 57 - Lista kvarova zgrade

Za svaki pojedini kvar iz liste kvarova zgrade moguće je vidjeti detalje kvara, te povijest napomena predstavnika suvlasnika. Naime, po završetku popravka izvođač radova proglašava kvar popravljenim nakon čega predstavnik suvlasnika potvrđuje ili ne potvrđuje obavljeni posao uz pisanje napomene.

The screenshot shows a navigation menu on the left with the following items:

- NASLOVNICA
- « PREDSTAVNIK SUVLASNIKA
- ZGRADA
- Zemljišna knjiga
- Rad uprave
- Kvarovi** (highlighted)
- Pričuva
- Izdani računi pričuve

On the right, there are two main sections:

- Kvar: Dizalo ne radi**
 - Podaci o prijava kvara**

Zgrada: Ilica 123, 10000 Zagreb	Lokacija: Cijela zgrada
Datum prijave: 22.5.2011.	Datum završetka: 22.5.2011.
Prijavio: Suvlasnik Tri	Upravitelj: Upravitelj Jedan d.o.o.
Izvođač radova: Izvođač Jedan d.o.o.	Račun: račun nije formiran
Status: Završeno	Hitnost: Visoka
Kvar: Dizalo ne radi	Vrsta kvara: Servis dizala
Opis kvara: Dizalo ne radi nekoliko dana, čini se da je zaglavljeno.	
 - Detalji**
 - Upovititelje instrukcije izvođaču radova:**
Molim provjerite osvjetljenja tipki za odabir katova.
 - Zaključak izvođača radova:**
Dizalo je popravljeno te su sredena osvjetljenja tipki. Osvjetljenje tipke 1 je popravljeno.
 - Napomene predstavnika suvlasnika:**

22.05.2011., 11:25:12	Na tipki 1 svjetlo nije ispravno.
22.05.2011., 11:26:14	Odlično obavljen posao

Slika 58 - Detaljan prikaz kvara

7.2.7 Financije

Za svaki obavljeni popravak od strane izvođača radova te ostalih troškova upravitelja zgrade potrebno je izdati račun zgradi radi koje su se obavile određene usluge. Izdavanje računa omogućeno je izvođačima radova i upravitelju čija se stranica za izdavanje računa znatno ne razlikuje. Izdavatelj računa definira opis plaćanja te dodaje stavke računa specifirajući količinu, cijenu po komadu, te opis stavke.

NASLOVNICA	Izdavanje računa										
IZVOĐAČ RADOVA											
Kvarovi	Podaci za račun										
Računi	Popravak/održavanje za koji se izdaje račun:										
Usluge popravka	Dizalo ne radi ▾										
	Opis plaćanja:										
	Popravak dizala i zamjena osvjetljenja za tipke unutar dizala.										
	Stavke računa										
	Dodaj stavku Obriši zadnju stavku										
	<table border="1" style="width: 100%;"><thead><tr><th>Količina</th><th>Cijena kom(bез PDV-a)</th><th>Opis</th></tr></thead><tbody><tr><td>1</td><td>1356,45</td><td>Popravak dizala</td></tr><tr><td>2</td><td>120</td><td>Osvjetljenje za tipke</td></tr></tbody></table>		Količina	Cijena kom(bез PDV-a)	Opis	1	1356,45	Popravak dizala	2	120	Osvjetljenje za tipke
Količina	Cijena kom(bез PDV-a)	Opis									
1	1356,45	Popravak dizala									
2	120	Osvjetljenje za tipke									
	Izdaj račun										

Slika 59 - Izdavanje računa

Svi izdani računi vidljivi su za svakog izdavača posebno uz grupiranje neplaćenih i plaćenih računa.

Svi računi izdani za određenu zgradu plaćaju se iz pričuve. Na sljedećoj slici prikazana je stranica sa detaljima pričuve, odnosno stanje novca te računi koje je potrebno platiti i koji su već plaćeni novcem pričuve.

Upute za korištenje informacijskog sustava za upravljanje stambenim zgradama

NASLOVNICA

« PREDSTAVNIK SVVLASNIKA

ZGRADA

Zemljišna knjiga

Rad uprave

Kvarovi

Pričuva

Izdani računi pričuve

Pričuva za zgradu: Ilica 123, 10000 Zagreb

Podaci o pričuvu:

Zgrada: Ilica 123, 10000 Zagreb Stanje novca: 253,66 kn

Neplaćeni računi zgrade:

Broj računa	Poziv na broj	Izdao	Iznos	Iznos+PDV	Plaćeno	Akcija
26	1-2-2011-05-22	Izvođač Jedan d.o.o.	1596,45 kn	1963,63 kn	Ne	Plati

Plaćeni računi zgrade:

Broj računa	Poziv na broj	Izdao	Iznos	Iznos+PDV	Plaćeno
25	1-1-2011-05-22	Upravitelj Jedan d.o.o.	456,99	562,0977	Da

Slika 60 - Prikaz pričuve i povijest potrošnje

Za primitak novca u pričuvu mjesečno se izdaju računi pričuve svakom suvlasniku zgrade. Na stranici „Izdani računi pričuve“ moguće je vidjeti za svaki mjesec listu izdanih računa pričuve. Lista je također grupirana na naplaćene račune i na one nenaplaćene. Prikaz stranice dostupan je isključivo predstavniku suvlasnika i upravitelju, dok samo upravitelj može evidentirati i potvrđivati uplate.

NASLOVNICA

« UPRAVITELJ

ZGRADA

Zemljišna knjiga

Rad uprave

Kvarovi

Upraviteljevi računi

Pričuva

Izdani računi pričuve

Izdani računi pričuve 05. 2011. za zgradu: Ilica 123, 10000 Zagreb

Računi

Nenaplaćeni računi:

Broj računa	Poziv na broj	Za	Iznos	Iznos+PDV	Plaćeno	Akcija
14	333-5-2011-05-22	Suvlasnik Dva	87,29 kn	107,37 kn	Ne	Potvrdi naplatu
15	333-6-2011-05-22	Suvlasnik Tri	124,36 kn	152,96 kn	Ne	Potvrdi naplatu
16	333-7-2011-05-22	Suvlasnik Jedan d.o.o.	129,68 kn	159,51 kn	Ne	Potvrdi naplatu
17	333-8-2011-05-22	Suvlasnik Dva d.o.o.	156,37 kn	192,34 kn	Ne	Potvrdi naplatu
19	333-10-2011-05-22	Suvlasnik Pet	288,64 kn	355,03 kn	Ne	Potvrdi naplatu
20	333-11-2011-05-22	Suvlasnik Šest	161,94 kn	199,19 kn	Ne	Potvrdi naplatu
21	333-12-2011-05-22	Suvlasnik Sedam	257,16 kn	316,31 kn	Ne	Potvrdi naplatu
22	333-13-2011-05-22	Suvlasnik Tri d.o.o.	287,29 kn	353,37 kn	Ne	Potvrdi naplatu
23	333-14-2011-05-22	Suvlasnik Osam	337,06 kn	414,58 kn	Ne	Potvrdi naplatu

Naplaćeni računi:

Broj računa	Poziv na broj	Za	Iznos	Iznos+PDV	Plaćeno
13	333-4-2011-05-22	Suvlasnik Jedan	91,86 kn	112,99 kn	Da
18	333-9-2011-05-22	Suvlasnik Četiri	250,01 kn	307,51 kn	Da
24	333-15-2011-05-22	Upravitelj Jedan d.o.o.	321,35 kn	395,26 kn	Da

Slika 61 - Lista izdanih računa pričuve

8. Zaključak

U okviru ovog diplomskog rada proučen je temeljni pristup u oblikovanju informacijskog sustava vođenog domenom primjene (eng. *Domain – Driven Design*, DDD). Razvoj vođen domenom primjene je postupak oblikovanja informacijskog sustava koji je usko uskladen sa poslovnim zahtjevima i procesima. Viziju takvog pristupa čini zajedničko razumijevanje onoga što zapravo pokušavamo stvoriti, odnosno probleme koje želimo riješiti. Cilj treba biti jasan i razumljiv podjednako poslovnim ekspertima i razvojnicima informacijskog sustava, a njihova suradnja mora biti obostrana. Domena čini jezgru informacijskog sustava i predstavljena je sveprisutnim jezikom. Domena je enkapsulirana modelom domene koji se sastoji od nekoliko koncepata poput entiteta, vrijednosnih objekata, agregata, tvornica, repozitorija i servisa te njihovih međusobnih veza. Sustav se može sastojati od nekoliko različitih domena od kojih svaka živi unutar svojeg omeđenog konteksta.

Uvodi se pojam objektno – relacijskog mapera kao mosta za smanjenje neusuglašenosti objektne i relacijske paradigme u okviru perzistencije bogatog modela domene. Objektno – relacijski maperi omogućuju da model domene bude neovisan o bazi podataka i da se sačuvaju sve pozitivne značajke objektno – orijentirane paradigmе.

Specificiran je, modeliran i implementiran informacijski sustav za upravljanje stambenim zgradama kao web primjenski program. Prikazani su i pojedinačno obrađeni bitni elementi nužni za ostvarenje informacijskog sustava vođenog domenom primjene kao što su apstrakcije za ostvarenje modela domene, koncepti iz domene, infrastruktura u obliku implementiranih NHibernate repozitorija sa potporom za NHibernate sjednice i transakcije te aplikacijski i prezentacijski sloj u okviru ASP.NET MVC web razvojnog okvira. Naposljetku, dane su upute za postavljanje sustava kao i upute za korištenje informacijskog sustava za upravljanje stambenim zgradama.

DDD nije uvijek najbolje rješenje za ostvarenje informacijskog sustava. Koristi se onda kada postoji značajna složenost poslovnih procesa i kada postoji značajan fokus na dobro definiran poslovni model, kada postoji suradnja sa ekspertima domene te za one informacijske sustave za koje je poznato da će se dalje razvijati i imati dug životni vijek. 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 koja postavlja DDD. Odgovornost je na razvojniku da osigura sva pravila 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.

Prednosti koje DDD uvodi sačinjavaju visoka kohezija i mali stupanj međuvisnosti u aplikacijskom kodu, jednostavno testiranje komponenti i izoliranost poslovne logike od ostalih slojeva (prezentacija, infrastruktura). Međutim, izolacija nije totalna jer zbog nesavršenosti tehnologija potrebna su određena znanja o slojevima i razvojnim okolinama u kojima će se model domene koristiti, radi međusobne kompatibilnosti i učinaka na performanse. Premda DDD zastupa pojam „Persistence ignorance“ preko repozitorija, odnosno tvrdnju da prilikom primjene DDD nismo opterećeni persistencijom i infrastrukturnim slojem, već je glavni fokus isključivo na domeni, to u praksi nije posve točno jer učinkovitost informacijskog sustava ovisi o performansama. Pitanje koje se često puta pojavljuje jest 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. U obzir treba uzeti činjenice da je potrebno uspostaviti asocijacije među objektima kako bi se iskoristile mogućnosti objektno – relacijskog mapera NHibernate. Nekog posebnog pravila nema, odnosno potrebno je pronaći što bolji kompromis između koncepata domene i implementacijskih detalja. Repozitoriji trebaju osim konceptima iz domene omogućiti specificiranje načina dohvaćanja (*eager* ili *lazy*) asociranih objekata, a za kolekcije, ovisno o kontekstu, omogućiti dohvaćanje podskupa kolekcije. Idealno bi bilo kada bi se nad NHibernate proxy kolekcijom specificirao takav upit koji se pretvara u stablo izraza i u trenutku obilaska po članovima kolekcije evaluira. Time bi se omogućilo dohvaćanje samo određenog podskupa cijele kolekcije bez udara na performanse, ukoliko se radi o velikoj količini podataka kolekcije, i to bez posredovanja repozitorija.

9. 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. Srinivas Penchikala: *InfoQ: Domain Driven Design and Development In Practice*, 2008.
URL: <http://www.infoq.com/articles/ddd-in-practice>
5. *Domain Driven Design*
URL: http://en.wikipedia.org/wiki/Domain-driven_design
6. Martin Fowler: *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002.
7. Djanji R. Prasanna: *Dependency Injection*, Manning, 2009.
8. Pierre Henri Kuaté, Tobin Harris, Christian Bauer, Gavin King: *NHibernate in Action*, Manning, 2009.
9. Jason Dentler: *NHibernate 3.0 Cookbook*, Packt Publishing, 2010.
10. Steven Sanderson: *Pro ASP.NET MVC 2 Framework*, Apress, 2010.
11. Stefan Schackow: *Professional ASP.NET 2.0 Security, Membership, and Role Management*, Wiley Publishing, 2006.
12. *Upravljanje i održavanje zgrada*
URL: <http://www.maksimus.hr/upravljanje/> (2010)

10. Naslov, sažetak i ključne riječi

Naslov:

Oblikovanje informacijskog sustava za upravljanje stambenim zgradama vođeno domenom primjene

Sažetak:

Cilj ovog diplomskog rada je primjenom postupka oblikovanja vodenog domenom primjene specificirati, modelirati i ostvariti informacijski sustav za upravljanje stambenim zgradama kao web primjenski program. Čitatelja se uvodi u problematiku razvoja informacijskih sustava i rješenja u kojemu se domena poslovnih procesa i njezin model postavljaju u fokus tog razvoja.

Proučen je i teorijski opisan temeljni pristup u oblikovanju informacijskog sustava vođenog domenom primjene. Uveden je i opisan pojam objektno – relacijskog mapera kao mosta za smanjenje neusuglašenosti objektne i relacijske paradigme u okviru perzistencije bogatog modela domene.

Prikazani su i pojedinačno obrađeni bitni elementi nužni za ostvarenje informacijskog sustava vođenog domenom primjene kao što su apstrakcije za ostvarenje modela domene, koncepti iz domene, infrastruktura u obliku implementiranih NHibernate rezervorija sa potporom za NHibernate sjednice i transakcije te aplikacijski i prezentacijski sloj u okviru ASP.NET MVC web razvojnog okvira.

Dane su upute za postavljanje informacijskog sustava kao i upute za njegovo korištenje. Naposljetku dan je zaključak.

Ključne riječi:

razvoj vođen domenom, DDD, arhitektura, informacijski sustav, objektno – orijentirana paradigma, objektno – relacijsko mapiranje, testom vođeno razvijanje, TDD, oblikovni obrasci, injektiranje ovisnosti, inverzija kontrole, NHibernate, .NET, ASP.NET MVC, SQL, stambene zgrade, upravljanje

11. Title, Abstract and keywords

Title:

Domain driven design of information system for housing management

Summary:

The goal of this thesis is to specify, model and implement an information system for housing management as a web application by applying domain – driven design. We introduce the reader to the problem of the information system development and the solution in which the business process domain and its model are in the focus of development.

We study and theoretically describe the basic approach to domain – driven information system design. We introduce and describe the notion of the object – relational mapper as a bridge used to decrease gaps between object and relational paradigms within a persistence context of a rich domain model.

We show and individually present essential elements necessary to implement an information system lead by domain – driven design, such as abstractions for the domain model implementation, domain concepts, infrastructure in the form of implemented NHibernate repositories with NHibernate session support and transactions, as well as the application and the presentation layer within the ASP.NET MVC web development framework.

We give instructions for the information system setup as well as usage instructions. Finally, we present a conclusion.

Keywords:

domain – driven design, DDD, architecture, information system, object – oriented paradigm, object – relational mapping, test driven development, TDD, design patterns, dependency injection, inversion of control, NHibernate, .NET, ASP.NET MVC, SQL, housing, management

12. Dodatak A – Korisnički podaci IS za bazu podataka sa prezentacijskim primjerima

Sljedeća tablica sadrži korisničke račune informacijskog sustava koji se nalaze uz priloženu i kreiranu bazu podataka sa prezentacijskim primjerima.

Tabela 2 - Korisnički podaci IS za bazu podataka sa prezentacijskim primjerima

Korisničko ime	Lozinka	Uloga
upravitelj01	password	Upravitelj zgrade, suvlasnik
suvlasnik01	password	Predstavnik suvlasnika, suvlasnik
suvlasnik02	password	Suvlasnik
suvlasnik03	password	Suvlasnik
izvodac01	password	Izvođač radova
izvodac02	password	Izvođač radova
izvodac03	password	Izvođač radova