UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA

MAGISTRSKO DELO

ANALIZA RAZVOJA PROGRAMSKE OPREME Z UPORABO OGRODIJ ORM

Ljubljana, februar 2017 MATEJ STARE

IZJAVA O AVTORSTVU

Podpisani Stare Matej, študent Ekonomske fakultete Univerze v Ljubljani, avtor predloženega dela z naslovom Analiza razvoja programske opreme z uporabo ogrodij ORM, pripravljenega v sodelovanju s svetovalcem red. prof. dr. Jurijem Jakličem

IZJAVLJAM

1. da sem predloženo delo pripravil samostojno;

2. da je tiskana oblika predloženega dela istovetna njegovi elektronski obliki;

3. da je besedilo predloženega dela jezikovno korektno in tehnično pripravljeno v skladu z Navodili za izdelavo zaključnih nalog Ekonomske fakultete Univerze v Ljubljani, kar pomeni, da sem poskrbel, da so dela in mnenja drugih avtorjev oziroma avtoric, ki jih uporabljam oziroma navajam v besedilu, citirana oziroma povzeta v skladu z Navodili za izdelavo zaključnih nalog Ekonomske fakultete Univerze v Ljubljani;

4. da se zavedam, da je plagiatorstvo – predstavljanje tujih del (v pisni ali grafični obliki) kot mojih lastnih – kaznivo po Kazenskem zakoniku Republike Slovenije;

5. da se zavedam posledic, ki bi jih na osnovi predloženega dela dokazano plagiatorstvo lahko predstavljalo za moj status na Ekonomski fakulteti Univerze v Ljubljani v skladu z relevantnim pravilnikom;

6. da sem pridobil vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v predloženem delu in jih v njem jasno označil;

7. da sem pri pripravi predloženega dela ravnal v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil soglasje etične komisije;

8. da soglašam, da se elektronska oblika predloženega dela uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;

9. da na Univerzo v Ljubljani neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve predloženega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja predloženega dela na voljo javnosti na svetovnem spletu preko Repozitorija Univerze v Ljubljani;

10. da hkrati z objavo predloženega dela dovoljujem objavo svojih osebnih podatkov, ki so navedeni v njem in v tej izjavi.

V Ljubljani, dne _____ Podpis študenta:_____

KAZALO

KAZALO SLIK

UVOD

Večina poslovnih aplikacij shranjuje podatke. Shranjevanje in poizvedovanje po podatkih je eden od temeljnih konceptov razvoja aplikacij. Informacijski sistemi morajo trajno shranjevati podatke, sicer niso uporabni. Razvijalci dnevno uporabljajo relacijske podatkovne baze in poizvedujejo po njih. Leonard (2013) navaja, da je na tržišču več različnih vrst podatkovnih baz, vendar so v poslovnem svetu še vedno najbolj uporabljane relacijske podatkovne baze. Podatki so ključnega pomena za podjetja in relacijske podatkovne baze omogočajo njihovo uporabo in trajno shranjevanje.

Žontar (2009, str. 6) navaja: »Običajno se v arhitekturi informacijskih sistemov zgradi poseben nivo, ki je zadolžen za komunikacijo z viri podatkov in pretvarjanjem surovih podatkov v objekte«. Podatkovni sloj mora poskrbeti za trajno shranjevanje objektov iz programske kode v podatkovno bazo. Cilj magistrskega dela je razložiti problematiko razvoja podatkovnega sloja in dostopanja do podatkov v relacijskih podatkovnih bazah.

Shranjevanje podatkov je bila vedno zanimiva tema v razvijalskih skupnostih Java in .Net. Nekateri razvijalci zagovarjajo shranjevanje podatkov zgolj s pomočjo strukturiranega povpraševalnega jezika (angl. *Structured Query Language,* v nadaljevanju SQL*)* in shranjenih procedur, drugi zagovarjajo uporabo objektno-relacijskih preslikovalnikov (v nadaljevanju ORM). Nekateri razvijalci zagovarjajo način ročnega dela in pisanja vseh poizvedb za branje, pisanje, posodabljanje in brisanje, drugi so bolj naklonjeni avtomatiziranju ponavljajočega se dela. Razvijalci imajo tudi različne poglede, kako doseči prenosljivost podatkovnega sloja zaradi različnih narečij SQL. Razprava se nikoli ne konča, zelo priljubljena rešitev pa je uporaba objektno-relacijskih preslikovalnikov (Bauer, King, & Gregory, 2016).

V programskih jezikih so podatki shranjeni v objektih, v relacijskih podatkovnih bazah pa v vrsticah podatkovnih tabel. Shranjevanje objektov v relacijske podatkovne baze predstavlja za razvijalce veliko dela. Ogrodja ORM poskušajo premostiti vrzel med relacijskim in objektnim svetom ter prihraniti količino dela, potrebnega za razvoj podatkovnega sloja. Objektni programski jeziki podpirajo točno določene podatkovne tipe. Podatkovni tipi v relacijskih podatkovnih bazah se razlikujejo od tipov v objektnih jezikih. Ogrodja ORM so zdaj široko sprejeta in uporabljana tako v skupnosti Java kot v skupnosti .Net.

V preteklosti so se razvijalci ukvarjali s pisanjem programske kode za dostopanje do podatkov. Po nekaterih podatkih je te kode do 50 odstotkov celotne programske kode (Mostarda, Sanctis, & Bochicchio, 2011). Večina današnjih poslovnih programskih rešitev uporablja podatke iz relacijskih podatkovnih baz, po katerih se poizveduje s pomočjo jezika SQL. Ročno napisana programska koda za dostop do podatkovne baze je zelo prilagodljiva, saj imamo nadzor nad poizvedbami SQL. Prilagodimo jo lahko določeni podatkovni bazi in uporabimo vse njene specifične funkcionalnosti. Največja težava pri ročno napisani kodi je v veliki količini vloženega dela za delo s podatki. Za vsako tabelo v podatkovni bazi je treba

napisati kodo in poizvedbo SQL za branje, dodajanje, brisanje in popravljanje podatkov. To je ponavljajoče se delo, ki se ga razvijalci izogibamo. Pri menjavi podatkovnega strežnika je treba preveriti ter po potrebi popraviti vse poizvedbe SQL. Na Statističnem uradu Republike Slovenije (v nadaljevanju SURS) večina aplikacij uporablja podatkovno bazo kot vir podatkov. Uporabljamo podatkovni bazi Microsoft SQL Server ter Oracle. V prihodnosti želimo biti čim bolj neodvisni od obeh podatkovnih baz, ki ju trenutno uporabljamo. Iskali smo način, kako bi si olajšali delo, in se odločili preizkusiti različna ogrodja ORM.

Ogrodja ORM omogočajo shranjevanje objektov v relacijsko podatkovno bazo in branje teh objektov. Bistvo tovrstnih orodij je preslikava objektnega sveta v relacijski svet, pomagajo nam zmanjšati neusklajenost med podatki v relacijskih bazah in poslovnimi objekti (Stare, 2013). Ogrodja zmanjšajo količino programske kode, potrebne za dostopanje do podatkov. Podatke tudi preslikajo iz relacijske podatkovne baze v objekte v programski kodi (Stare, 2013).

Glavne prednosti uporabe ogrodij ORM so povečanje produktivnosti pri njihovi uporabi, zmanjšanje pisanja ponavljajoče se kode in lažja menjava podatkovnega strežnika. Podjetja uporabljajo različne ponudnike podatkovnih baz. Programsko opremo, namenjeno različnim podjetjem, je treba prilagoditi podatkovnim bazam, ki jih podjetja uporabljajo. Zaradi dragih licenc, zmogljivosti, produktivnosti, dodatnih funkcionalnosti, ki jih omogoča druga podatkovna zbirka, se podjetja odločajo za menjavo podatkovnega strežnika. Zaradi dragih licenc je v določenih primerih ceneje prilagoditi programsko opremo drugemu podatkovnemu strežniku kot kupiti licenco za uporabo drugega podatkovnega strežnika. Ogrodja ORM prilagodijo poizvedbe SQL uporabljani podatkovni bazi, s čimer se izognemo popravljanju poizvedb in prihranimo veliko časa. Ogrodja ORM omogočajo lažjo menjavo podatkovnega strežnika, vendar obstajajo določene omejitve, ki sem jih pojasnil v nadaljevanju.

Na trgu je veliko ogrodij ORM, nekatera so plačljiva, druga odprtokodna. V svojem magistrskem delu sem predstavil ORM-ogrodja NHibernate in Microsoft Entity Framework. Ogrodje NHibernate je odprtokodna ORM-rešitev za ogrodje .Net. NHibernate izvira iz ogrodja Hibernate, ki je zelo poznano pri razvijalcih v okolju Java in je zrelo ter preverjeno ogrodje (Schenker & Cure, 2011).

V magistrskem delu sem primerjal razvoj aplikacij z dostopom do podatkovne baze s poizvedbami SQL z razvojem z uporabo ogrodij ORM. Analiziral sem težave in prednosti, ki jih imamo z uporabo ogrodij ORM pri razvoju aplikacij. Primerjal sem razvoj lastnega podatkovnega sloja z razvojem s pomočjo ogrodja ORM, primerjal sem tudi vzdrževanje podatkovnega sloja oziroma programske kode. Vzdrževanje programske opreme je lahko bolj zahtevno kot sam razvoj aplikacij, zelo pomembni sta dobra dokumentacija in dobro napisana programska koda.

Opisal sem tudi upravljanje izvorne kode. Pri razvoju in vzdrževanju aplikacij potrebujemo shrambo izvorne kode. Pod pojmom izvorna koda mislimo na programsko kodo, nastavitvene

datoteke, dokumentacijo itd. V shrambo je priporočljivo shranjevati vse, kar potrebujemo za prevajanje in nameščanje aplikacije.

Predstavil sem podatkovni sloj, ki je zaključena enota programske kode, ta pa je zadolžena za dostopanje do podatkov, ki so največkrat shranjeni v podatkovni bazi. Vse metode in funkcije, ki berejo in pišejo podatke v podatkovni vir, so del tega sloja. Podatkovni sloj lahko razvijemo sami, poskrbeti moramo za shranjevanje podatkov, poizvedovanje po podatkih, pravilno izvajanje transakcij itd. To je sicer izvedljivo, vendar zahteva veliko ponavljajočega se dela, ki pa ni povezano s ključnimi poslovnimi zahtevami. Večino dela predstavlja pisanje kode za delo s podatki.

Namen magistrskega dela je izboljšati razvoj programske opreme na SURS-u. Pridobljena znanja bomo uporabili pri vsakodnevnem delu. V praktičnem delu sem razvil aplikacijo za upravljanje dokumentov na SURS-u. Podatkovni sloj aplikacije sem razvil s pomočjo ogrodja ORM in ga primerjal z lastno razvitim podatkovnim slojem.

V magistrskem delu sem iskal odgovore na naslednji vprašanji:

1. **Ali uporaba ogrodij ORM prispeva k bolj učinkovitemu in kakovostnejšemu razvoju programske opreme?**
2. **Kako uporaba ogrodij ORM omogoča hitrejšo menjavo podatkovnega strežnika?**

Osnovni cilj magistrskega dela je podrobna analiza različnih načinov razvoja podatkovnega sloja in njihova primerjava. Analiziral sem razvoj lastnega podatkovnega sloja in razvoj z uporabo ogrodja ORM. Opisal sem tudi ogrodje Entity Framework podjetja Microsoft in odprtokodno ogrodje ORM NHibernate. Na praktičnem primeru sem analiziral prednosti in slabosti obeh vrst razvoja.

Pri izdelavi magistrskega dela sem uporabil znanje, pridobljeno na podiplomskem študiju, in znanja, pridobljena z delom na SURS-u. Preučil sem domačo in tujo literaturo s področja razvoja programske opreme in podatkovnih baz.

Na začetku sem predstavil zgodovino podatkovni baz, nadaljeval z opisom ogrodja Ado.Net, nato pa predstavil še ogrodja ORM. Opisal sem tudi različne vidike varnosti in zmogljivosti programske kode ter primerjal različne načine poizvedovanja po podatkovnih bazah. Predstavil sem tudi uporabo ogrodij ORM v večslojnih aplikacijah ter spletne storitve.

Na primeru aplikacije za delo z dokumenti sem analiziral prednosti in slabosti razvoja podatkovnega sloja s pomočjo ogrodij ORM. Primerjal sem razvoj lastnega podatkovnega sloja z ogrodjem Ado.Net z razvojem podatkovnega sloja s pomočjo ogrodij ORM. Na praktičnem primeru sem predstavil razvoj dokumentnega sistema.

V praktičnem delu sem razvil aplikacijo Dokumentni sistem. Medvešek (2013, str. 1) navaja: »Poslovanje podjetja je tesno povezano z dokumenti, ki jih podjetje prejme oziroma nastanejo v okviru posameznih poslovnih procesov«. Podatkovni sloj aplikacije sem razvil dvakrat, in

sicer s pomočjo ogrodja Entity Framework, ter primerjal tak način razvoja z razvojem podatkovnega sloja s pomočjo ogrodja Ado.Net. Opravil sem analizo prednosti, slabosti, priložnosti in nevarnosti (angl. *strengths, weaknesses, opportunities, threats*, v nadaljevanju SWOT) razvoja podatkovnega sloja s pomočjo ogrodja ORM in lastno razvitega podatkovnega sloja. Na podlagi analize SWOT sem podal predloge za izboljšanje razvoja programske opreme na SURS-u.

Magistrsko delo je razdeljeno na šest vsebinskih poglavij. V uvodnem poglavju je predstavljena problematika razvoja programske opreme in v drugem podatkovni sloj. V poglavju Objektno-relacijski preslikovalniki sta predstavljeni ogrodji NHibernate in Entity Framework. Četrto poglavje predstavlja praktični del magistrskega dela, v katerem so predstavljene ugotovitve razvoja aplikacije Dokumentni sistem. Naslednje poglavje obsega primerjalno analizo razvoja podatkovnega sloja z ogrodjem ORM in lastno razvitega podatkovnega sloja. Sklepni del magistrskega dela povzema ključne ugotovitve.

1 PODATKOVNI SLOJ

1.1 Zgodovina

Podatkovne baze so del našega življenja, uporabljamo jih vsakodnevno, čeprav se tega morda niti ne zavedamo. Vsi dvigi na bankomatu, telefonski klici, rezervacije letalskih kart itd. so povezani s podatkovnimi bazami. Podatkovne baze v današnjem času shranjujejo tudi veliko število osebnih podatkov, ki jih nato podjetja uporabljajo za različne analize.

Prvi sistemi za shranjevanje podatkov so se pojavili konec 19. stoletja. Herman Hollerith je uporabil preluknjane kartice za popis prebivalstva v Združenih državah Amerike. Hollerith je pozneje ustanovil podjetje International Business Machines (v nadaljevanju IBM), ki je imelo monopolni položaj na trgu v začetku 20. stoletja. Ukvarjali so se s shranjevanjem podatkov na kartice in pripravo izpisov na podlagi shranjenih podatkov (Celko, 2015).

Po drugi svetovni vojni je ameriški statistični urad uporabljal računalnike s karticami za delo z večjimi količinami podatkov (Celko, 2015).

Dr. E. F. Codd se je ukvarjal s teorijo množic, ki je vsebovala tudi osnovne koncepte relacij. Relacijske podatkovne baze so poenostavile delo s podatkovnimi bazami. Programerji v COBOL-u so morali poznati strukturo celotne podatkovne baze, če so želeli poizvedovati po podatkih. Te so morali izpisovati v formatu, ki je bil razumljiv ljudem. Dr. Codd si je prizadeval za poenostavitev poizvedovanja po podatkih in njihovega spreminjanja. S pomočjo sodelavcev je razvil poizvedovalni jezik SQL, ki je v današnjem času poizvedovalni jezik, podprt s strani vseh glavnih ponudnikov podatkovnih baz (Celko, 2015).

Mohorič (1991, str. 151) navaja: »[…] leta 1974 je bil definiran in tudi eksperimentalno implementiran strukturiran povpraševalni jezik za delo s podatkovnimi bazami SQL«.

Konec sedemdesetih let prejšnjega stoletja sta podjetji Oracle in Sybase razvili prve različice komercialnih relacijskih podatkovnih baz. V osemdesetih letih prejšnjega stoletja je ameriški urad za standardizacijo pripravil standard ANSI-SQL, kar je omogočilo večjo skladnost med različnimi ponudniki podatkovnih baz. Vsi ponudniki se poskušajo čim bolj držati standarda, vendar obstajajo vseeno različne izjeme pri sintaksah SQL različnih podatkovnih baz. Prednosti standardizacije za razvijalce so v tem, da znanje, pridobljeno z učenjem sintakse določene podatkovne baze, lahko uporabijo tudi drugje.

Grad in Jaklič (1996, str. 100) navajata: »Relacijski podatkovni model temelji na matematični teoriji relacij, podatki so organizirani v relacijske podatkovne strukture, ki so fizično predstavljene z dvodimenzionalnimi tabelami. Relacijske podatkovne baze so v 80. letih postale izjemno razširjene in še danes je to v poslovnih informacijskih sistemih najpogosteje uporabljen podatkovni model«.

Leta 1978 je podjetje Oracle izdelalo prvo različico podatkovne baze, ki pa ni šla v prodajo. Druga različica je bila prva komercialna podatkovna baza na tržišču, tretja pa je bila napisana v programskem jeziku C in je delovala na različnih platformah.

Med bolj priljubljenimi podatkovnimi bazami je tudi Microsoft SQL Server. Razvil se je s pomočjo partnerskega podjetja Sybase leta 1989. Partnerstvo med podjetjema se je končalo z različico SQL Server 7, ki je bil čisto nov produkt, nepovezan s starimi različicami. Ena glavnih slabosti Microsoft SQL Serverja je v tem, da deluje zgolj na operacijskih sistemih Windows. Ti sistemi so v preteklosti veljali za manj zanesljive kot sistemi UNIX, zato je bil posledično tudi SQL Server namenjen manjšim in srednjim podjetjem. Strežnik SQL Server slovi po relativno nizki ceni in enostavni uporabi.

Podatkovne baze so rešile veliko težav z integriteto in podvajanjem podatkov, ki se pojavljajo v datotečnih sistemih. Celko (2015, str. 266) je zapisal: »Pravilno dizajnirana podatkovna baza shranjuje podatke zgolj enkrat na eno mesto, podatki so dostopni vsem uporabnikom, ki imajo pravice za branje«.

1.2 Podatkovni sloj

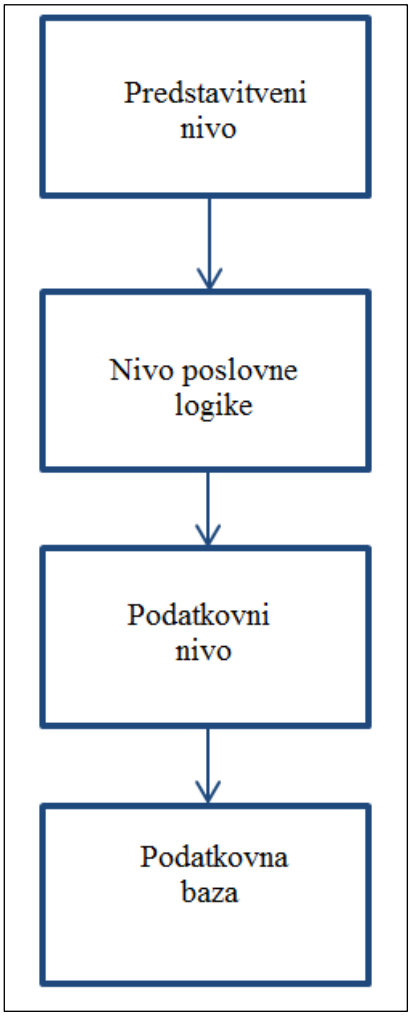
Podatkovni sloj je zaključena enota programske kode, ki je zadolžena za dostopanje do podatkov, ki so največkrat shranjeni v relacijski podatkovni bazi. Vse metode in funkcije, ki berejo in pišejo podatke v podatkovni vir, so del tega sloja.

Klasičen večslojni sistem je sestavljen iz treh slojev. Žontar (2009, str. 8) navaja: »Trislojna arhitektura temelji na ločevanju informacijskih sistemov na sloje, ti sloji so med seboj šibko sklopljeni, kar pomeni, da posamezen sloj komunicira le s spodaj ležečim slojem«.

Predstavitveni sloj vsebuje uporabniške vmesnike, poslovni sloj skrbi za poslovno logiko aplikacije, v podatkovnem sloju izvajamo poizvedbe SQL. Podatkovni sloj lahko uporabimo v več različnih aplikacijah, kar nam prihrani veliko dela. Razvoj podatkovnega sloja je odvisen

od zahtev uporabnikov, v njem so shranjene poizvedbe, povezave do podatkovne baze in drugi bazni objekti. Spodnja Slika 1 prikazuje trislojno arhitekturo.

Slika 1: Trislojna arhitektura



Vir: D. Esposito, Programming Microsoft ASP.NET 4, 2011, str. 593.

Podatkovni sloj je samostojen sloj aplikacije, njegova naloga je upravljanje s podatki. Ločen je od preostalih slojev aplikacije in uporabe podatkov v aplikaciji – to nam omogoča uporabo podatkovnega sloja v različnih aplikacijah. Pri razvoju podatkovnega sloja se lahko odločimo za uporabo ogrodij ORM, lahko pa zgradimo lasten podatkovni sloj. Glavni namen podatkovnega sloja je v ločevanju uporabniškega vmesnika, poslovne logike in dostopanja do podatkov. Preostali sloji uporabljajo zgolj metode v podatkovnem sloju, ne dostopajo pa neposredno do podatkov. Pri nekaterih aplikacijah je predpisana uporaba shranjenih procedur, in sicer zaradi varnosti ali kakšnih drugih razlogov. Shranjene procedure nam otežijo prenos podatkov v podatkovno bazo drugih ponudnikov (migracija). Težje je tudi vzdrževanje shranjenih procedur, morebitne napake pri spremembi shranjene procedure ugotovimo šele pri izvajanju aplikacije.

Danes prevladujejo večslojne aplikacije. Večslojna arhitektura razdeli programsko kodo v več slojev, ki so namenjeni reševanju določene problemske domene. Zgornji sloj vsebuje uporabniški vmesnik, sredinski sloj je namenjen poslovnim pravilom, spodnji sloj pa dostopu do podatkov.

Prednosti uporabe večslojne arhitekture so predvsem v možnosti popravkov zgolj na določenem sloju brez vpliva na druge sloje. Komunikacija poteka vedno med dvema sosednjima slojema. Pri odločanju za večslojno arhitekturo moramo imeti v mislih tudi velikost aplikacije.

Neodvisnost od podatkovne baze v grobem pomeni, da lahko uporabljamo različne ponudnike podatkovnih baz. Neodvisnost od podatkovne baze je pogosta zahteva, še posebej pri razvijalcih programske opreme, ki izdelujejo aplikacije, namenjene različnih podjetjem.

Neodvisnost lahko dosežemo s podatkovnim slojem, ki ima za vsako podatkovno bazo, ki jo želimo podpreti, ločeno programsko kodo. Katero programsko kodo naj podatkovni sloj uporabi, določimo v nastavitvenih datotekah. Slabost te rešitve je v podvajanju programske kode.

V zadnjem času se za doseganje neodvisnosti uporabljajo objektno-relacijski preslikovalniki. Ogrodja ORM imajo standardiziran vmesnik za dostopanje do podatkovne baze, do katere točno, pa se določi v nastavitvenih datotekah.

Glavne naloge podatkovnega sloja so branje, shranjevanje in urejanje podatkov. Pri lastnem razvoju podatkovnega sloja moramo biti pozorni na vse zgoraj omenjene zahteve. Uporaba ogrodij ORM nam olajša delo, vendar je vseeno treba preveriti, kaj vse podpira ogrodje ORM.

Problem sočasnega popravljanja istih podatkov se pojavi v več uporabniških sistemih, pogost je predvsem pri spletnih aplikacijah. Uporabnik lahko prebere določene podatke, in medtem ko jih popravlja, iste podatke popravi že drug uporabnik. V tem primeru se lahko odločimo za sistem »zadnji zmaga« in podatke povozimo. To je sicer najenostavnejše, vendar v veliko primerih ne zadostuje.

1.3 Lasten razvoj podatkovnega sloja

Podatkovni sloj lahko razvijemo sami, pri čemer moramo poskrbeti za shranjevanje podatkov, poizvedovanje po podatkih, pravilno izvajanje transakcij idr. To je sicer izvedljivo, vendar zahteva veliko ponavljajočega se dela, ki ni povezano s ključnimi poslovnimi zahtevami. Večino dela zahteva pisanje programske kode za delo s podatki. Pri gradnji lastnega podatkovnega sloja moramo poskrbeti za preslikavo metod v poizvedbe SQL, kar nam omogoča hitrejše poizvedbe, vendar zahteva tudi več dela.

V nekaterih primerih so zaradi varnostnih zahtev, lažjega vzdrževanja in bolj berljive kode stavki SQL za delo s podatki napisani v shranjenih procedurah. Shranjena procedura (angl.

stored procedure) je podprogram, ki se izvaja znotraj relacijske podatkovne baze.

V preteklosti so bili najbolj zmogljivi računalniki namenjeni relacijskim podatkovnim bazam, v katerih so se shranjevali podatki. Veliko razvijalcev je za poslovno logiko v aplikaciji uporabljalo shranjene procedure, ki so bile v primerjavi s poizvedbami, pisanimi v nizu, veliko boljša in varnejša rešitev. Razvijalci pogosto nimamo dovolj znanja o poizvedovalnem jeziku SQL in njegovi uporabi. Pri večjih projektih so v razvojni skupini sodelavci zadolženi za pisanje zapletenih shranjenih procedur.

Poizvedovalni jezik SQL je namenjen branju in popravljanju podatkov. Podatkovna zbirka dobi zahtevo v nizu in jo nato izvede. Poizvedba je prevedena, zgradi se tudi načrt izvajanja. Naslednjič enaka poizvedba uporabi že obstoječ načrt izvajanja iz spomina, zato se izvede hitreje.

Shranjene procedure so nekoliko hitrejše, vendar samo pri prvi uporabi, ker že obstaja načrt izvajanja (Vieira, 2009). Pri pripravi shranjene procedure je treba zgraditi tudi načrt izvajanja, ki se nato uporablja. Shranjene procedure je treba na določeno časovno obdobje ponovno prevesti, kajti v nekaterih primerih se lahko zgodi, da načrt izvajanja ni več optimalen.

Pri programski opremi, pri kateri so varnostne zahteve zelo visoke, ni dovoljen dostop do podatkovne zbirke prek poizvedb v nizu, ampak zgolj z uporabo shranjenih procedur. Slabost takšnega pristopa je v tem, da potrebujemo za vsako tabelo vsaj štiri shranjene procedure, kar lahko hitro postane zapleteno in nepregledno. Pri večji podatkovni zbirki hitro dobimo nekaj sto shranjenih procedur. Če želimo napisati dobre shranjene procedure, moramo vedeti, kako podatkovna zbirka deluje in kako se izvajajo poizvedbe po podatkih.

1.4 Ogrodje Ado.Net

Pri lastno razvitem podatkovnem sloju s pomočjo ogrodja Ado.Net imamo nadzor nad poizvedbami SQL.

Ogrodje Ado.Net. je na voljo od leta 2002 kot del ogrodja .Net. Je zbirka orodij in programskih knjižnic, ki so v pomoč razvijalcem pri dostopanju do podatkov. Vsebuje vmesnike za delo z večino priljubljenih podatkovnih strežnikov. Ogrodje Ado.Net vsebuje mnogo objektov, v nadaljevanju bom opisal pomembnejše med njimi.

Prednost tehnologije Ado.Net pred prejšnjimi tehnologijami za dostop do podatkov je v tem, da je napisana v »nadzorovani programski kodi« (angl. *managed code*). Zaradi tega se nam ni treba ukvarjati s sproščanjem objektov po uporabi, ker za to poskrbi t. i. smetar (angl. *garbage collector*). V programskem jeziku C++ je treba vsak objekt, ki ni več v uporabi, sprostiti, sicer lahko pride do puščanja pomnilnika (angl. *memory leak*). Programske knjižnice omogočajo povezovanje na podatkovne vire, izvajanje poizvedb SQL, shranjevanje in spreminjanje podatkov. Druga zelo pomembna prednost je, da omogočajo delo v t. i. odklopljenem načinu od vira podatkov.

Ogrodje Ado.Net vsebuje različne ponudnike za dostopanje do podatkov, med njimi so Microsoft SQL Server, ODBC in OLEDB. Dodatne ponudnike so razvile zunanje družbe za dostopanje do njihovih podatkovnih baz. Tako obstaja ponudnik za večino glavnih podatkovnih baz, kot so Oracle, MySQL, PostgreSQL, DB2 in Sybase.

Za branje podatkov v povezanem načinu se uporablja objekt DataReader, po podatkih se lahko pomikamo samo naprej in namenjeni so samo branju.

Objekt DataReader je smiselno uporabljati tudi za poizvedbe, ki vrnejo zgolj eno vrednost, npr. štetje zapisov.

Gačnik (2005, str. 23) je zapisal: »Dataset ima več dobrih lastnosti, načrtovan je tako, da ga je mogoče preprosto in hitro prenašati med sloji porazdeljene aplikacije«.

Odklopljen način dostopa do podatkov je ključen pri povečevanju zmogljivosti programske opreme, kajti podatke lahko shranimo v pomnilnik (angl. *cache*), ti pa so tako veliko hitreje dostopni kot pri povezovanju na podatkovno bazo. Podatkovna baza je tudi manj obremenjena.

V objektu DataSet so shranjene izvorne in spremenjene vrednosti podatkov. Pri shranjevanju podatkov nazaj v podatkovno bazo lahko s pomočjo izvornih vrednosti shranimo zgolj vrstice, ki so bile spremenjene.

Objekt DataTable je del arhitekture Ado.Net, lahko se uporablja tudi neodvisno od DataSet-a, sestavljen je iz množice stolpcev, vrstic in omejitev. Stolpci in omejitve sestavljajo shemo, podatki pa se nahajajo v vrsticah. Objekt DataTable je sestavljen iz stolpcev in vrstic, podobno kot tabela v podatkovni bazi. Objekt DataSet je sestavljen iz enega ali več objektov DataTable. V DataSet lahko shranimo več podatkovnih tabel iz baze, vse spremembe se shranijo v vrstici.

Pri ogrodju Ado.Net imamo dve možnosti za dostopanje do podatkov, objekt DataReader je manj potraten glede pomnilnika, je tudi hitrejši za branje, vendar bere vrstico po vrstico podatkov. V primeru velikega števila sočasnih uporabnikov je DataReader bolj primerna rešitev. Pri uporabi DataSetov si lahko pomagamo s shranjevanjem v pomnilnik, ki nam omogoča boljše zmogljivosti. Za shranjevanje v spomin so primerni podatki, ki se ne spreminjajo veliko. DataSet ima podatke shranjene v razširljivem označevalnem jeziku (angl. *extensible markup language,* v nadaljevanju XML)

Podatkovni strežnik Microsoft SQL Server nam tudi omogoča povezavo med podatki v bazi in podatki v pomnilniku. Podatkovni strežnik spremlja spremembe glede na poizvedovalni stavek SQL in zamenja vrednosti v pomnilniku v primeru sprememb. Nastavimo lahko tudi časovno omejitev, do kdaj so podatki v pomnilniku veljavni, po preteku časovnega intervala se podatki sami osvežijo.

Tipiziran Dataset temelji na shemi XML (angl. *XML Schema Definition,* v nadaljevanju XSD). Shema XSD je jezik za opis podatkovnih struktur in pravil. Aplikacije lahko uporabljajo

podatke iz sheme pri upravljanju s podatki. XML-shema je XML-dokument, ki definira strukturo drugega XML-dokumenta ter vrsto podatkov, ki jih dokument lahko vsebuje. Glavne prednosti shem XML so v tem, da so napisane v jeziku XML in jih lahko na relativno enostaven način preberemo, prevajalniki in aplikacije pa jih uporabijo pri delu s podatki.

V shemi XSD definiramo pravila, s katerimi mora biti določen XML-dokument skladen, da je veljaven glede na shemo. Preverjanje pravilnosti XML-dokumenta je največkrat uporabljena možnost shem XML. XML-dokumente preverjamo pri izmenjavi različnih dokumentov, pri preverjanju pravilnosti podatkov pred vnosom v zaledne sisteme itd. Različni tipi shem so za različne vrste preverjanja, za bolj zahtevno preverjanje je bolj primeren drug način.

Tipiziran DataSet lahko zgradimo z uporabo razvojnega orodja Microsoft Visual Studio s pomočjo čarovnika, brez potrebnega znanja XML-shem. S shemami XML se je smiselno seznaniti, kajti v nekaterih zahtevnejših primerih uporaba čarovnika ni dovolj.

Prednost uporabe tipiziranih objektov DataSet je predvsem v tem, da nam že shema XSD preverja pravilnost vnesenih podatkov. Določimo lahko ključe, omejitve in povezave. Glavna prednost za razvijalce je v lažjem dostopanju do vrednosti.

Primer branja objekta DataSet:

- DataSet
  string Ime = DataSet.Tables[»Oseba«].Rows[0][»Ime«];
- Tipiziran DataSet
  string Ime = DataSet.Oseba[0].Ime;

V drugem primeru ni tekstovnih vrednosti, kar zmanjša možnost napak pri razvoju.

Objekt DataAdapter je vmesni člen med podatkovno bazo in odklopljenimi objekti, kot sta DataTable in DataSet. Poskrbi za prenos podatkov iz baze v objekt DataSet, s katerim lahko nato delamo v odklopljenem načinu. Spremembe lahko s pomočjo DataAdapterja znova shranimo v podatkovno bazo. DataSet je kopija podatkov v spominu, po katerih lahko poizvedujemo in jih razvrščamo. Pri delu z odklopljenimi podatki ne potrebujemo povezave na podatkovno bazo, pri čemer pa ne dobimo nujno zadnjih podatkov. Drugi uporabniki podatkovne baze namreč lahko spremenijo določene podatke in jih mi nato ne vidimo.

Objekt DataSet vsebuje tabele in povezave med njimi, zato se pogosto imenuje tudi podatkovna baza v računalniškem spominu. Ado.Net uporablja za popravljanje podatkov t. i. optimistični pristop, pri zapisu spremenjenih vrstic v DataSet-u se preverijo vrednosti polj, in če so te vrednosti enake vrednostim v podatkovni bazi, se izvedejo popravki. V primeru spremenjenih vrednosti v podatkovni bazi lahko prepišemo te vrednosti z novimi vrednostmi ali uporabniku pošljemo opozorilo, da so bile vrednosti že spremenjene.

Pri delu z ogrodjem Asp.Net nam shranjevanje objektov v pomnilnik (angl. *caching*) lahko zelo pospeši delovanje spletnih strani. Pri bolj obremenjenih aplikacijah si lahko pomagamo z

več strežniki, pri čemer se vsaka zahteva izvede na naključnem strežniku v gruči. V tem primeru je treba izbrati rešitev, ki omogoča dostopanje do predpomnilnika iz različnih strežnikov.

Shranjevanje objektov v pomnilnik nam zagotovi hiter dostop do podatkov, brez vsakokratnega dostopanja do podatkovne baze. To je smiselno za podatke, ki se ne spreminjajo pogosto in se pogosteje uporabljajo.

Podatkovni strežnik Microsoft SQL Server ima od različice 2005 naprej možnost obveščanja pomnilnika o spremembah v podatkih.

1.5 Transakcije

Pomemben del zagotavljanja integritete podatkov so tudi transakcije. Klasičen primer transakcije, ki je pogosto naveden v literaturi, je prenos denarja iz računa A na račun B (Vieira, 2009).

Težava nastane, če se ukaz SQL za dvig denarja iz računa A izvrši, ukaz za polog denarja na račun B pa ne. Stranka v tem primeru ostane brez denarja in posledično se zmanjša ugled banke. Vsem dejavnikom, ki lahko povzročijo nepravilno izvedbo stavkov, se ne moremo izogniti, zato potrebujemo transakcije.

Transakcije morajo biti (Henderson & Celko, 2000):

- atomarne. Po uspešni transakciji se vse spremembe trajno shranijo, v primeru neuspeha se spremembe vrnejo v začetno stanje. Na primer transakcija vsebuje deset ukazov za brisanje, prvih devet se jih uspešno izvede, zadnji pa ne, zato se mora prvih devet sprememb vrniti v prvotno stanje.
- Konsistentne. Transakcija je konsistentna, če zagotavlja, da podatki niso nikoli v neveljavnem stanju. Npr. z ukazom SQL popravimo vrednosti, vendar se transakcija ne izvede uspešno. Zunanje poizvedbe nimajo možnosti dostopa do vmesnih spremenjenih vrednosti.
- Izolirane. Vsaka transakcija je samostojna, transakcije ne smejo vplivati ena na drugo.
- Trajne. Rezultati uspešno izvedene transakcije so trajno shranjeni, največkrat na podatkovni disk. Vmesne spremembe se shranjujejo v transakcijski dnevnik (angl. *transaction log*), kar omogoča vrnitev v prvotno stanje, če se transakcija ne konča uspešno.

Ogrodje Ado.Net nudi podporo za izvajanje transkacij. Glavni metodi transakcijskega razreda sta Commit in Rollback.

Metoda Commmit sporoči podatkovni bazi, da so se vsi ukazi SQL pravilno izvršili, in sproži ukaz Commit na podatkovni bazi. Metoda Rollback se izvede v primeru neuspeha pri izvajanju ukazov SQL.

Transakcije je smiselno uporabljati le pri spreminjanju podatkov oziroma takrat, ko lahko zaporedje neuspešno izvedenih stavkov SQL povzroči nepravilno stanje podatkovne baze. Nepotrebna uporaba transakcij povzroči nepotrebno zaklepanje tabel in poslabša zmogljivost podatkovne baze.

Transakcije se lahko izvajajo s pomočjo ogrodja Ado.Net, lahko pa tudi na podatkovnem strežniku v shranjenih procedurah. Dobra praksa pri razvoju podatkovnega sloja so čim krajše transakcije, dolge transakcije namreč lahko zadržujejo druge uporabnike pri dostopu do podatkov.

1.6 Podatkovni model

Namen razvoja aplikacij je izboljšanje poslovnih procesov oziroma lažje in bolj učinkovito delo za zaposlene. Pri razvoju aplikacij potrebujemo dobro razumevanje poslovne domene. Dobro aplikativno rešitev lahko razvijemo zgolj z dobrim poznavanjem procesov v podjetju. Za lažje razumevanje poslovnih zahtev si pomagamo s poslovnim modelom, ki je način za opis stanja, preslikava poslovnega sveta v entitete.

Model je sestavljen iz pomembnejših opisov poslovnega sveta in povezav med njimi. Opisi so največkrat poenostavljena slika realnosti. Poslovne procese opišemo zgolj v določenem obsegu, namenjenem razvoju naše aplikacije. S pomočjo modela se lahko osredotočimo na bistvene zahteve naročnikov programske opreme, lažje se tudi pogovarjamo s poslovnimi analitiki. Model je poskus opisa določene poslovne domene. Poslovni analitiki in arhitekti podatkovnih baz skupaj pripravijo podatkovni model, ki je sestavljen iz entitet.

Entiteta je objekt, ki ga lahko enolično določimo z njegovimi atributi. Pogoste entitete v poslovnih aplikacijah so stranka, naročilo, izdelek itd. Entitete imajo ključno vlogo v poslovnem modelu, predstavljajo pomembne koncepte poslovnih zahtev.

V aplikacijah pogosto uporabljamo t.i. nadomestne ključe. Nadomestni ključ je največkrat zaporedna številka. Naravni ključi se v daljšem časovnem obdobju lahko spremenijo, zato je pogosta uporaba nadomestnih ključev.

2 OBJEKTNO-RELACIJSKI PRESLIKOVALNIKI

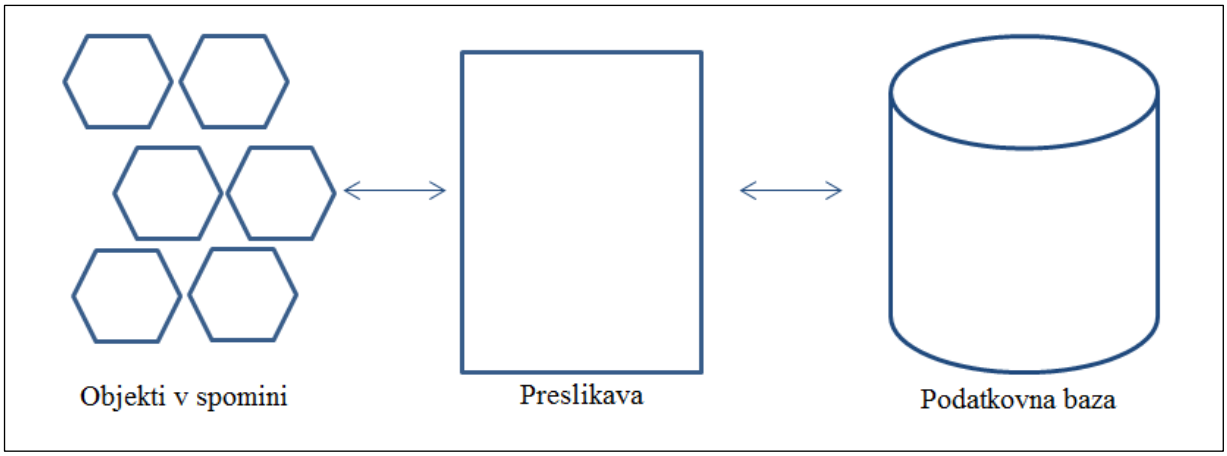
V preteklosti so se razvijalci ukvarjali s pisanjem programske kode za dostopanje do podatkov. Pisanje te kode je bilo ponavljajoče, razvijalci so morali za vsako tabelo v podatkovni bazi pisati podobno programsko kodo, česar so se naveličali. Nekateri so si pomagali z generatorji programske kode, ki s pomočjo metapodatkov zgradijo podatkovni sloj za dostopanje do podatkov. Ta način je bil primeren predvsem za točno določene podatkovne baze, pri vsaki spremembi podatkovnega modela je bilo potrebno ponovno generiranje podatkovnega sloja.

Na SURS-u večina aplikacij uporablja relacijsko podatkovno bazo kot vir podatkov. Podatkovni sloj smo razvijali s pomočjo ogrodja Ado.Net. Pogosto so se dogajale napake pri izvajanju aplikacij. Razlogi so bili različni, najpogosteje je prišlo do napak zaradi sprememb na podatkovnem modelu in nepravilnih stavkov SQL. V strokovni literaturi smo zasledili vse večjo priljubljenost ogrodij ORM, zato smo se odločili, da jih preizkusimo.

Ogrodja ORM omogočajo razvijalcem večjo produktivnost, zmanjša se pisanje ponavljajoče se programske kode. Aplikacije z manj programske kode so tudi lažje razumljive in bolj enostavne za preoblikovanje (angl. *refactoring*).

Ogrodja ORM nam sicer zelo pomagajo pri dostopanju do podatkov, vendar je znanje jezika SQL vseeno potrebno za kakovostnejše delo. Ogrodja v določenih primerih ne generirajo najboljšega poizvedovalnega stavka, zato potrebujemo znanje jezika SQL za preverjanje in izboljšanje poizvedb. Slika 2 prikazuje preslikavo objektov v podatkovno bazo.

Slika 2: Objektno-relacijsko preslikovanje



Vir: J. Lerman, Programming Entity Framework, 2010, str. 29.

Kot opisuje Dino Esposito (Esposito & Saltarello, 2014) v knjigi Architecting Applications for the Enterprise, njihova razvojna skupina uporablja ogrodja ORM že dalj časa. Uporaba jim je prinesla veliko prednosti, vendar jih niso znali ceniti, dokler niso razvijali programske opreme, pri kateri je bil zahtevan razvoj podatkovnega sloja brez uporabe ogrodja ORM. Projekt so le s težavo končali pravočasno, delo je bilo tudi zelo naporno.

Objekti s podatki se morajo preslikati v relacijski podatkovni model in se v programski kodi razlikujejo od tabel v podatkovni bazi. Ogrodja ORM poskrbijo za preslikavo objektov v relacijsko podatkovno bazo.

Uporaba objektno-relacijskega povezovanja omogoča preslikavo podatkov v relacijski obliki v objekte in shranjevanje nazaj v relacijsko podatkovno bazo. V metapodatkih so shranjene preslikave med objekti in podatkovno bazo.

2.1 Ogrodje Entity Framework

Ogrodje Entity Framework (v nadaljevanju EF) je objektno-relacijski preslikovalnik podjetja Microsoft. Podjetje Microsoft (Entity Framework, 2015) navaja: »Ogrodje Entity Framework je objektno-relacijski preslikovalnik, ki omogoča .Net razvijalcem delo z relacijskimi podatki z uporabo domenski objektov«.

Ogrodje EF razvija podjetje Microsoft v sodelovanju z zunanjimi razvijalci. Ti lahko dodajo svojo programsko kodo, ki jo nato preverijo člani ekipe, odgovorne za razvoj ogrodja EF, in jo dodajo v ogrodje. Na ta način dobimo po mojem mnenju odprtokodno rešitev z manjšo verjetnostjo napak.

Prednosti uporabe ogrodja Entity Framework v primerjavi z lastnim podatkovnim slojem (Mostarda et al., 2011):

- produktivnost. Po izkušnjah avtorjev knjige Entity Framework 4 in Action (Mostarda et al., 2011) porabimo za pisanje programske kode za dostop do podatkov do 35 odstotkov časa, namenjenega razvoju aplikacij. Uporaba ogrodij ORM lahko zmanjša ta čas na pet odstotkov v nekaterih skrajnih primerih, v večini primerov pa se ti odstotki gibljejo med 15 in 20 odstotkov. Orodje Visual Studio nam s pomočjo grafičnega orodja olajša povezovanje podatkovne baze in logičnega modela v ogrodju EF.
- Vzdrževanje. Manj kot imamo programske kode, manj je moramo vzdrževati. To se pozna na dolgi rok pri uporabi preoblikovanja (angl. *refactoring*) programske kode, pri čemer je manjšo količino kode lažje preveriti. Ogrodja ORM imajo možnost zmanjšanja vrzeli med objektnimi in relacijskimi modeli, kar nam omogoča popravljanje podatkovne strukture brez poseganja v aplikacijsko kodo. Brez uporabe ogrodij ORM bi vsaka sprememba podatkovne strukture povzročila veliko ročnega popravljanje kode.
- Vključenost ogrodja Entity Framework v ogrodje Microsoft .Net. Podjetje Microsoft v celoti podpira EF in zagotavlja odpravo napak, dokumentacijo in popravke, čemur konkurenčni produkti težko sledijo. Razvijalci se pogosto odločajo za EF namesto za NHibernate, ker je EF del .Net ogrodja – to pa zagotavlja kakovost celotne programske kode. Ogrodje .Net je brezplačno, kar pomeni, da je tudi ogrodje EF brezplačno.
- Neodvisnost od podatkovne baze. Ogrodje EF lahko uporabljamo za vse podatkovne baze, za katere so napisani ponudniki. Trenutno so na voljo ponudniki za vse bolj uporabljane podatkovne baze, kot so Microsoft SQL Server, Oracle, IBM, MySQL itd. Ogrodje EF zagotavlja prevod metod za pridobivanje podatkov v stavke SQL, ki jih podatkovna baza razume. Vse moderne podatkovne baze podpirajo standard SQL, vendar so kljub temu velike razlike med stavki SQL v različnih podatkovnih bazah.

Slabosti uporabe ogrodja Entity Framework v primerjavi z lastnim podatkovnim slojem (Mostarda et al., 2011):

- zmogljivost. Slabša zmogljivost v primerjavi s klasičnimi dostopanjem do podatkov je zelo pogost očitek ogrodjem ORM. Ogrodja so zaradi svoje zapletenosti slabše zmogljiva, vendar je to v večini primerov sprejemljivo. Le v redkih primerih so ogrodja prepočasna in zato neprimerna za uporabo.
- Vstavljanje večje količine podatkov.
- Zapletene poizvedbe SQL.

2.1.1 Ogrodje Entity Framework in večslojne aplikacije

Pri enostavnejših enoslojnih aplikacijah so povezani uporabniški vmesnik, poslovna logika in dostop do podatkov. V tem primeru izvajamo operacije nad podatki in jih nato shranimo v podatkovno bazo. Entitete imajo lastnosti, ki nam povejo, kateri podatki so se spremenili, in te podatke nato shranimo v podatkovno bazo (t. i. povezan način delovanja).

V večslojnih aplikacijah imamo odklopljen način delovanja, kar pomeni, da se entitete ne zavedajo stanja podatkov. Tak način dela je bolj težaven, ker ogrodje EF ne skrbi več za sledenje spremembam na entitetah. Podatki se preberejo iz podatkovne baze in shranijo v entitete. Vrednosti v entitetah spreminjamo na uporabniškem vmesniku, nato jih pošljemo do podatkovnega sloja, ki poskrbi za shranjevanje sprememb.

Preverjanje pravilnosti podatkov z vnosnih mask izvajamo na različnih slojih. Pri spletnih aplikacijah preverjamo vnosne maske na uporabniškem vmesniku s pomočjo jezika JavaScript. Smiselno je preverjati vrednosti tudi na podatkovnem sloju, kajti JavaScript lahko na brskalniku izklopimo. Lastnosti razreda opremimo z atributi, kot so obvezno polje, največja dolžina, podatkovni tip. Pri shranjevanju entitete se preverja pravilnost vnosa podatkov glede na podane atribute.

Aplikacije so tesno povezane s spodaj ležečimi podatki. Za vsako spremembo na strukturi podatkov je potreben poseg v programsko kodo, kar povzroči veliko dodatnega dela. V primeru spremembe imena stolpca v podatkovni bazi moramo zamenjati stolpec v vseh poizvedbah ter pri prebiranju vrednosti. Napako pogosto odkrijemo šele pri izvajanju aplikacije. Prevajalnik kode ne more pomagati, ker so imena stolpcev podana v nizu.

V današnjem času je večina jezikov objektno usmerjenih, proceduralni jeziki, kot je na primer COBOL, še obstajajo, vendar so v manjšini. V večini današnjih aplikacij se uporabljajo razredi (angl. *class*) v programski kodi. Razredi predstavljajo podatke s pomočjo lastnosti.

Prednosti uporabe razredov so predvsem (Mostarda et al., 2011):

- preverjanje napak pri prevajanju. Razredi uporabljajo lastnosti za dostop do podatkov, ne uporabljajo niza znakov oziroma zaporedne številke. Če vpišemo napačno ime lastnosti, takoj dobimo napako pri prevajanju. Ni nam več treba zaganjati aplikacije za iskanje tipkarskih napak.

- Enostaven razvoj. Urejevalniki, kot je Visual Studio, nudijo pomoč (angl. *Intellisense*) pri pisanju programske kode za pospešitev razvoja in razvijalcem ponujajo namige za lastnosti razreda.

2.1.2 Poizvedovalni jezik Entity SQL

Ogrodje EF je bilo prvič predstavljeno na razvojni konferenci TechEd leta 2006 v Združenih državah Amerike. Takrat še ni obstajal poizvedovalni jezik Language-Integrated Query (v nadaljevanju LINQ), zato so se odločili narediti različico poizvedovalnega jezika SQL, ki so jo poimenovali Entity SQL. Sintaksa je zelo podobna poznani sintaksi SQL. Jezik SQL je namenjen poizvedovanju po relacijskih podatkovnih bazah in nima podpore za dedovanje in preostale objektne lastnosti, ki so na voljo v ogrodju EF. Ogrodje je moralo podpreti te možnosti, zato se je razvil nov poizvedovalni jezik Entity SQL (v nadaljevanju ESQL) (Mostarda et al., 2011).

ESQL omogoča poizvedovanje po konceptualnem modelu. Podatki so predstavljeni kot entitete in relacije, ESQL pa omogoča poizvedovanje po njih na podoben način kot v jeziku SQL. Poizvedba ESQL je s pomočjo vmesnika prevedena v poizvedbo SQL za podatkovno bazo.

S pojavom jezika LINQ je bil Entity SQL postavljen bolj v ozadje, ker je postal drugi poizvedovalni jezik. ESQL sicer omogoča več, vendar je LINQ zaradi večje preglednosti in produktivnosti bolj priljubljen. LINQ nam ne omogoča uporabe vseh funkcij podatkovne baze, zato je takrat smiselno uporabiti ESQL. Pri vnaprej znani podatkovni bazi lahko uporabimo tudi običajen SQL, vendar potem izgubimo možnost povezovanja aplikacije na drugo vrsto podatkovne baze.

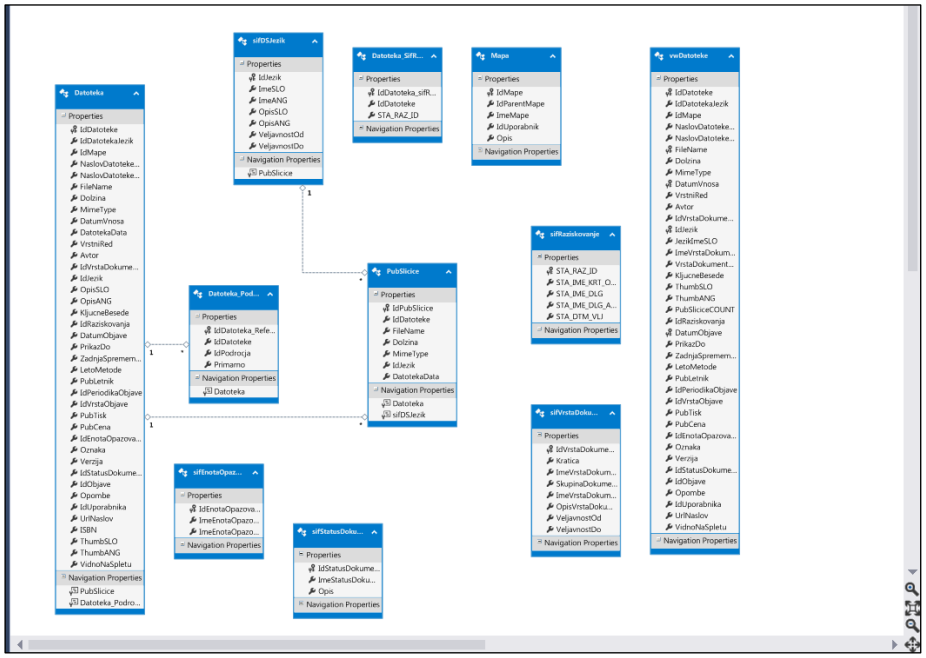
ESQL pride do izraza pri dinamičnem generiranju poizvedb. Poizvedovalni stavek zgradimo med izvajanjem, odvisno od želja uporabnika aplikacije. Kadar je poizvedba LINQ zelo kompleksna, je zaradi preglednosti smiselno uporabiti ESQL. To je zelo robusten jezik in omogoča široko uporabo.

2.1.3 Pristopi razvoja z ogrodjem Entity Framework

Ogrodje EF ponuja tri možnosti razvoja. Prva je razvoj podatkovnega modela (angl. *Model First*), druga razvoj z že obstoječo podatkovno bazo (angl. *Database First*), tretja pa razvoj modela v programski kodi (angl. *Code First*).

Pristop Model First je namenjen razvijalcem, ki pri svojih projektih še nimajo določene podatkovne baze. Ko končamo z razvojem logičnega modela podatkovne baze, ogrodje poskrbi za generiranje skript z ukazi, ki jih nato zaženemo na podatkovnem strežniku in dobimo tudi fizično podatkovno bazo. Poskrbi tudi za preslikavo podatkovnih tipov .Net v podatkovne tipe v podatkovni bazi. Na spodnji Sliki 3 je primer enostavnega podatkovnega modela v ogrodju EF.

Slika 3: Podatkovni model pri pristopu Model First



Pri razvoju po metodologiji Code First je treba upoštevati predpisana navodila glede poimenovanja. Razred mora imeti atribut, ki je ključ. Atribut, ki vsebuje v imenu Id, se preslika kot ključ v tabelo v podatkovni bazi. Pri izdelavi razredov je najbolje uporabiti enostavne razrede (angl. *Plain Old CLR Object*, v nadaljevanju POCO), ki so neodvisni od ogrodja EF in jih lahko neodvisno od drugih razredov uporabljamo v vseh slojih aplikacije. Razred POCO je predstavljen na Sliki 4.

Slika 4: Primer razreda POCO

```
0 references
public class Datoteka
{
    0 references
    public int IdDatoteke { get; set; }
    0 references
    public Nullable<int> IdDatotekaJezik { get; set; }
    0 references
    public Nullable<int> IdMape { get; set; }
    0 references
    public string NaslovDatotekeSLO { get; set; }
    0 references
    public string NaslovDatotekeANG { get; set; }
    0 references
    public string FileName { get; set; }
    0 references
    public Nullable<long> Dolzina { get; set; }
    0 references
    public string MimeType { get; set; }
    0 references
    public System.DateTime DatumVnosa { get; set; }
    0 references
    public byte[] DatotekaData { get; set; }
    0 references
    public Nullable<byte> VrstniRed { get; set; }
    0 references
    public string Avtor { get; set; }
    0 references
    public Nullable<int> IdEnotaOpazovanja { get; set; }
    0 references
    public string Oznaka { get; set; }
```

Pri pristopu Code First obstajajo določene omejitve. Ne omogoča sinhronizacije sprememb modela na bazi, vendar naj bi to odpravili v naslednji različici ogrodja EF. Težava je tudi pri povezovanju na shranjene procedure, saj jih moramo pri generiranju podatkovne baze iz razredov dodati ročno. Shranjene procedure lahko izvajamo tudi s pomočjo ukaza SQLQuery, vendar potem izgubimo možnost preverjanja pravilnosti programske kode pri prevajanju. Napake se pojavijo šele pri izvajanju programske kode, kar nam lahko povzroči nepričakovane težave.

Odločitev za uporabo pristopa Code First je predvsem odvisna od potreb razvijalcev. Če že imamo podatkovno bazo in želimo razvijati aplikacijo, je najbolj smiselno uporabiti pristop Database First.

2.2 Ogrodje NHibernate

Po besedah Cure in Schenker (Cure & Schenker, 2011, str. 7) je »ogrodje NHibernate odprtokodna ORM rešitev za ogrodje .NET. Izvira iz ogrodja Hibernate, ki je zelo poznano pri razvijalcih v Java okolju in je zrelo in preverjeno ogrodje.«

NHibernate je odprtokodno ogrodje, namenjeno delu s podatkovnimi bazami, ki omogoča shranjevanje objektov v podatkovno bazo in poskrbi za generiranje ustreznih stavkov SQL. Pri uporabi ogrodja NHibernate nam ni treba skrbeti za impedančno nesorazmerje med podatkovnimi bazami in programskimi jeziki .Net.

Podatkovna baza je relacijska in programski jeziki so večinoma objektni. Objektov ne moremo neposredno shraniti v podatkovno bazo, temveč jih moramo pretvoriti v bazi razumljivo obliko in jih vstaviti s pomočjo stavkov SQL – pri čemer si pomagamo z ogrodjem NHibernate.

Za poizvedovanje po podatkih je na voljo več možnosti. Prvi poizvedovalni jezik je bil Hibernate Query Language (v nadaljevanju HQL), ki je zelo podoben poizvedovalnemu jeziku SQL. Pri jeziku HQL poizvedujemo po objektih, pri jeziku SQL pa po podatkovnih tabelah. V določenih primerih nam jezik HQL ne omogoča uporabe vseh funkcij podatkovne baze – takrat moramo uporabiti jezik SQL.

2.2.1 NHibernate predpomnjenje

Shranjevanje objektov v pomnilnik (angl. *caching*) nam lahko zelo pohitri poizvedovanje po podatkih, ki so shranjeni v računalniškem pomnilniku. Namesto vsakokratnega branja iz podatkovnega strežnika vzamemo podatke iz spomina in jih prikažemo uporabniku. V spominu lahko shranjujemo rezultate določenih poizvedb, v primeru šifrantov je smiselno shranjevati kar celotno tabelo. Težave lahko nastanejo v primeru spremembe podatkov na podatkovnem izvoru.

NHibernate uporablja shranjevanje objektov v pomnilniku (angl. *caching*) za bolj učinkovito delo s podatki, pomnilnik omogoča pregled entitet brez dostopanja do podatkovne baze, če so entitete že v pomnilniku.

Shranjevanje podatkov v spominu je zelo primerno za spletne strani, z nekaj vrsticami programske kode lahko bistveno izboljšamo hitrost delovanja aplikacij. Aplikaciji ni treba dostopati do podatkovne baze vsakič, ko potrebuje iste podatke. Podatkovni strežnik je tudi manj obremenjen. Uporaba pomnilnika ni smiselna pri podatkih, ki se pogosto spreminjajo.

Ogrodje NHibernate uporablja predvsem tri vrste shranjevanja v spomin (Kuaté & Harris, 2009):

- transakcijski način. Podatki so shranjeni v spominu, dokler se izvaja določena operacija.
- Procesni način. Podatki so shranjeni v spominu, do njih dostopajo različni uporabniki z različnimi operacijami.
- Podatki v gruči (angl. *cluster*). Podatke si delijo različni procesi na istem strežniku, lahko pa tudi različni strežniki med seboj. Potrebujemo možnost branja istih podatkov na različnih strežnikih.

Pri uporabi gruče strežnikov (angl. *cluster*) moramo zagotoviti dostopnost podatkov v spominu vsem strežnikom. Rešitvi, ki sta nam na voljo, sta uporaba dodatnega strežnika, namenjenega podatkom v spominu, ali shranjevanje vrednosti v podatkovno bazo. Vsako možnost je treba preučiti in preveriti. Za manjše sisteme lahko uporaba gruče strežnikov prinese več težav kot koristi.

2.2.2 Poizvedovalni jezik HQL

Poizvedovalni jezik HQL je objektno naravnan poizvedovalni jezik, podoben jeziku SQL, vendar HQL namesto tabel in stolpcev uporablja objekte in njihove lastnosti. Poizvedbe v jeziku HQL se prevedejo v poizvedbe v jeziku SQL s pomočjo ogrodja NHibernate.

V ogrodju NHibernate lahko seveda uporabljamo tudi klasične poizvedbe SQL, vendar nato težje zamenjamo spodaj ležečo podatkovno bazo. Prednosti uporabe klasičnih poizvedb so v možnosti uporabe specifičnih funkcionalnosti, ki jih omogoča podatkovna baza.

Ogrodje NHibernate omogoča preslikavo razredov .Net v tabele v relacijski podatkovni bazi. Jezik HQL omogoča poizvedovanje po razredih v jeziku, podobnem jeziku SQL.

Jezik HQL ni nadomestek jezika SQL in omogoča pisanje objektno usmerjenih poizvedb. Poizvedbe HQL se prevedejo v stavke SQL, ki se nato izvedejo na podatkovni bazi. HQL je smiselno uporabljati za pisanje večine poizvedb, poizvedbe SQL uporabimo zgolj v primeru posebnih funkcij, ki jih omogoča točno določena podatkovna baza (Cure & Schenker, 2011).

Preden začnemo uporabljati katero koli ogrodje ORM, moramo poznati vsaj osnove jezika SQL. Ogrodja generirajo boljše ali slabše poizvedbe, poznavanje jezika SQL nam omogoči pregled nad poizvedbami. Znanje jezika SQL je potrebno tudi, če želimo izkoristiti vse možnosti določene podatkovne baze. Ogrodja nam pomagajo pri mnogih ponavljajočih se nalogah, vendar je končni cilj dobrih aplikacij robustno in učinkovito delo s podatki.

NHibernate vsebuje gradnike za hitro in enostavno izgradnjo podatkovnega sloja. Kadar želimo imeti popolni nadzor nad entitetami, jih je najbolje napisati ročno. To sicer zahteva malo več dela, vendar tako nismo odvisni od drugih razredov. Izdelan je z namenom učinkovitega shranjevanja podatkov, je zrelo ogrodje z vsemi potrebnimi funkcionalnostmi, namenjenimi enostavnemu in učinkovitemu shranjevanju podatkov (Cure & Schenker, 2011).

Bolj zapletene poslovne aplikacije vsebujejo veliko entitet z zapleteno poslovno logiko. Pri razvoju programske opreme so naši glavni cilji produktivnost, enostavno vzdrževanje in zmogljivost. Razlike med objektnim svetom in relacijskimi podatkovnimi bazami odpravljajo ogrodja ORM, med njimi je zelo priljubljeno tudi ogrodje NHibernate.

Ogrodje poskrbi tudi za neodvisnost aplikacije od podatkovne zbirke. Podpira vse bolj priljubljene podatkovne zbirke, kot so Microsoft SQL Server, Oracle, PostreSQL, MySQL itd. Podpora vsem tem podatkovnim zbirkam nam omogoča lažji prenos aplikacij med različnimi podatkovnimi okolji (Kuaté & Harris, 2009).

Ogrodja ORM niso rešitev za vse težave s shranjevanjem podatkov, nam pa prihranijo veliko dela, potrebnega za pisanje poizvedb SQL in programske kode za delo z objekti Ado.Net.

2.2.3 Povezovanje entitet s podatkovno bazo

Ogrodje NHibernate omogoča tri načine povezovanja entitet s podatkovno bazo (Cure & Schenker, 2011):

- preslikava XML. Entitete preslikamo v tabelo v podatkovni bazi s pomočjo datoteke XML, struktura te datoteke je definirana v shemi XSD z imenom Nhibernate-maping. Tak način je zelo prilagodljiv, vendar zahteva veliko dela. Za vsako entiteto je treba narediti svoj zapis v datoteki XML. Pri večjem številu podatkovnih entitet postane datoteka XML težko berljiva, vendar nam ta način v primerjavi z drugimi pušča največ možnosti.
- Preslikava na podlagi atributov. Entitete opremimo s pomočjo atributov, ki s pomočjo metapodatkov poskrbijo za preslikavo. Atributi so lažje razumljivi kot datoteka XML. Z njihovo pomočjo lahko dosežemo bolj enostavne preslikave, pri bolj zahtevnih preslikavah pa potrebujemo datoteko XML.
- Preslikava s pomočjo vmesnika (angl. *Fluent mapping*). Preslikava entitet se lahko naredi tudi s pomočjo vmesnika Fluent (angl. *Fluent API*). Prednost je v ločeni datoteki za preslikavo. Za vsako entiteto v modelu potrebujemo svoj razred, kar pri večjih modelih pomeni veliko dela.

Nalaganje na zahtevo (angl. *lazy loading*) je način branja podatkov v povezanih podatkovnih tabelah šele takrat, ko te podatke potrebujemo. Namen sprotnega nalaganja je odlog zahtevnejših poizvedb na čas, ko jih res potrebujemo. Ogrodje NHibernate prebere podatke o entiteti, povezane entitete pa prebere po potrebi.

S pomočjo ogrodja NHibernate lahko preberemo določeno entiteto in vključimo še podatke iz povezanih entitet, vendar takšen način pogosto ni primeren, kajti branje prevelike količine podatkov z zapletenimi poizvedbami močno obremeni podatkovni strežnik. Pri branju osebe iz podatkovne baze na primer preberemo samo osnovno tabelo o osebi, povezane podatke pa šele takrat, ko jih potrebujemo.

Sprotno nalaganje je zelo primerno za izboljšanje hitrosti delovanja aplikacije. Lahko se izognemo preveč zapletenim poizvedbam, kar nam skrajša odzivne čase in zmanjša količino prenesenih podatkov. Brez uporabe sprotnega nalaganja se lahko zgodi, da v aplikaciji preberemo določene povezane podatke, čeprav jih sploh ne potrebujemo.

2.3 Poizvedovalni jezik LINQ

Ogrodji Entity Framework in NHibernate omogočata poizvedovanje s pomočjo jezika LINQ. Lerman (2010) navaja, da je LINQ ena od ključnih prednosti uporabe ogrodja ORM. Uporaba LINQ-a nam poveča produktivnost, s pomočjo prevajalnika odkrijemo napake hitreje kot pri uporabi klasičnih poizvedb SQL.

S programskim jezikom LINQ lahko poizvedujemo po različnih podatkovnih virih z zelo podobnimi ukazi. Razvijalci smo bili vajeni dostopanja do relacijskih podatkovnih baz s pomočjo poizvedb SQL, za prebiranje datotek XML pa smo uporabili druge metode.

Različni načini dostopanja do podatkov so prisilili razvijalce k učenju različnih tehnologij. Te tehnologije se med seboj zelo razlikujejo, dobro znanje poizvedovalnega jezika SQL nam ne pomaga prav veliko pri poizvedovanju po datotekah XML. V današnjem poslovnem svetu, v katerem imamo zelo kratke roke za izdelavo programske opreme, je zelo težko dobro usvojiti toliko različnih tehnologij. LINQ je olajšal delo razvijalcem s poenotenim programskim vmesnikom za dostopanje do različnih virov podatkov.

Med bolj priljubljeni različicami jezika LINQ je LINQ to XML, ki nam omogoča delo z datotekami XML. Omogoča nam poizvedovanje, gradnjo ter spreminjanje datotek XML. Za delo s tovrstnimi datotekami so na voljo različne tehnologije, kot so XPath, XQuery, XSLT in XMLDOM. Cilj različice LINQ to XML je omogočiti objektno orientiran pristop za delo z datotekami XML, kar reši veliko težav, povezanih z manipulacijo datotek XML s pomočjo drugih tehnologij (Russo & Pialorsi, 2010).

Jezik LINQ nam omogoča poizvedovanje po različnih vrstah podatkov z enakimi poizvedbami. Struktura poizvedb je v določeni meri podobna poizvedovalnemu jeziku SQL.

Večina aplikacij na tak ali drugačen način dostopa do podatkov, ki so lahko shranjeni v različnih formatih. V preteklosti smo bili razvijalci prisiljeni v učenje programskega jezika, namenjenega razvoju aplikacij, za dostop do podatkov pa smo potrebovali znanje jezika SQL.

LINQ je vmesni člen med podatki, shranjenimi v različnih oblikah, in programskima jezikoma C# in Visual Basic. Enoten način dostopanja do podatkov v različnih virih nam olajša poizvedovanje. S poizvedbami SQL lahko poizvedujemo po podatkovnih bazah, s poizvedbami XPath po strukturi XML itn., vendar moramo poznati vsak način. Pri jeziku LINQ se naučimo enega načina dela in s tem znanjem lahko pokrijemo različne vrste podatkov.

Glavni namen razvoja poizvedb LINQ je bil v poenotenju načina dostopanja do različnih tipov podatkov. Pridobljeno znanje lahko uporabimo na različnih področjih. Slika 5 prikazuje poizvedbo LINQ.

Slika 5: Primer izpisa poizvedbe LINQ

```
var rez = from objava1 in db.Objave
          where objava1.IdObjave == iIdObjave
          select new entObjave
          {
              IdObjave = objava1.IdObjave,
              IdPodrocja = objava1.IdPodrocja,
              IdReferencnoObdobje = objava1.Objava_ReferencoObdobje.FirstOrDefault().IdReferencnegaObdobja,
              //ReferencnoObdobje = objava1.Objava_ReferencoObdobje.FirstOrDefault().IdReferencnegaObdobja.ToString(),
              ReferencnoLetoPodatkov = objava1.Objava_ReferencoObdobje.FirstOrDefault().ReferencnoLeto,
              PodrocjeNaslovSLO = objava1.Podrocje.NaslovSlo,
              IdVrstaObjave = objava1.IdVrstaObjave,
              DatumIzidaPlaniran = objava1.DatumIzidaPlaniran,
              DatumIzidaPotrjen = objava1.DatumIzidaPotrjen,
              ImeObjaveSLO = objava1.ImeObjaveSLO,
              ImeObjaveANG = objava1.ImeObjaveANG
          };
```

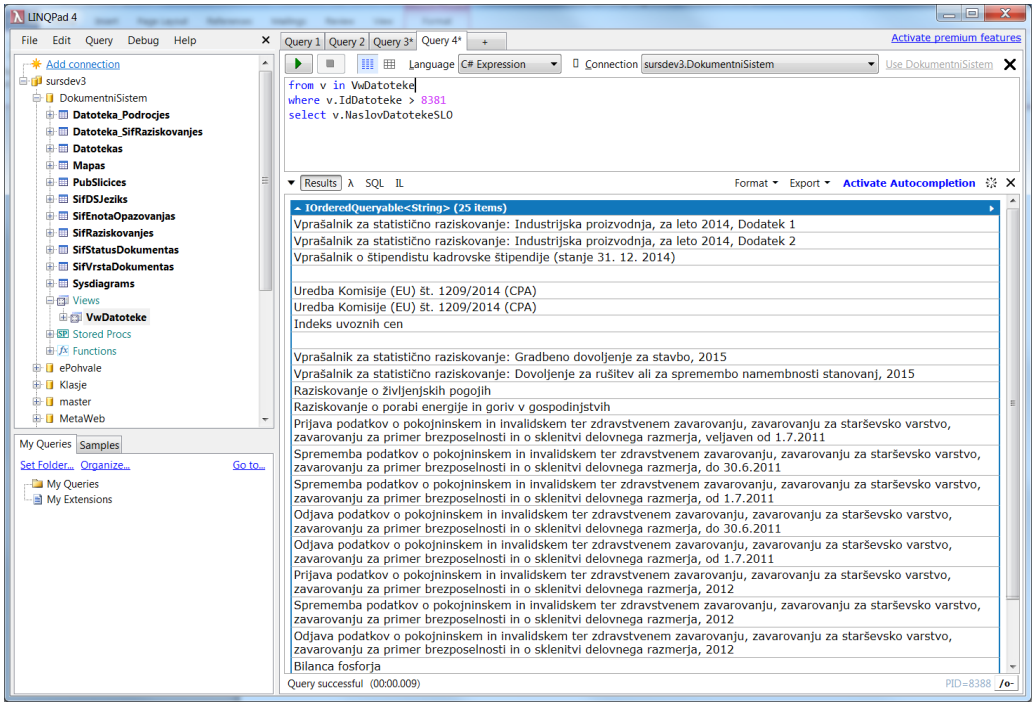
Podjetje Microsoft si je pri razvoju poizvedovalnega jezika LINQ zadalo nekaj glavnih ciljev (Marguerie, Eichert & Wooley, 2008):

- poizvedovanje po objektih, relacijskih podatkovnih bazah in XML-strukturah. Z enotno sintakso za poizvedovanje po različnih podatkovnih virih se izognemo uporabi različnih poizvedovalnih jezikov za različne podatkovne vire. Cilj je tudi enoten model za obdelavo podatkov ne glede na to, kje so podatki shranjeni.
- Prenašanje moči jezikov SQL in XQuery ter vključevanje poizvedovalne sposobnosti neposredno v programski jezik.
- Možnost razvoja vmesnikov tudi za druge podatkovne vire.
- Preverjanje pravilnosti pri prevajanju. Prevajalnik preveri pravilnost programske kode že pri prevajanju, s čimer se izognemo napakam, ki smo jih pred pojavom jezika LINQ odkrili šele pri izvajanju programske kode.
- Podpora dopolnjevanju programske kode. Pomaga razvijalcem pri pisanju poizvedb z namenom izboljšanja produktivnosti.

- Objektno-relacijsko preslikovanje. Relacijske podatkovne baze in objektni programski jeziki ne vsebujejo enakih podatkovnih tipov. Niz znakov v relacijski podatkovni bazi je največkrat omejene dolžine, niz v programskih jezikih pa nima omejitve.

Za lažje delo s poizvedbami LINQ je na voljo orodje LINQPad. Orodje je predstavljeno na Sliki 6.

Slika 6: Orodje LINQPad



To je brezplačno orodje za poizvedovanje po podatkovnih bazah v jeziku LINQ. Osnovna različica je na voljo brezplačno, plačljiva različica pa omogoča dopolnjevanje ukazov, kar nam olajša delo. Uporabljamo ga lahko kot nadomestek za orodja, namenjena poizvedovanju po podatkovnih bazah. Omogoča nam tudi izvajanje programske kode.

2.4 Ogrodje Asp.Net MVC

Ogrodje Asp.Net MVC sem uporabil pri prikazu datotek na spletni strani, pri čemer sem želel biti čim bolj neodvisen. V prihodnosti imamo namen določen del SURS-ove spletne strani pripraviti za mobilne odjemalce, in takrat bomo lahko uporabili vzorec »model-pogled-kontroler« (angl. *Model-View-Controller,* v nadaljevanju MVC).

Vzorec MVC je bil prvič uporabljen v programskem jeziku Smalltalk v osemdesetih letih prejšnjega stoletja. Vzorec je izumil Trygve Reenskaug (Yener & Theedom, 2015) pri obisku razvojnega centra Xerox. Prvi članek je predstavil leta 1978, prvotno je bil uporabljen kot vzorec pri razvoju grafičnih vmesnikov.

Po Reenskaugovi (Yener & Theedom, 2015) definiciji je vzorec sestavljen iz modela, v katerem so shranjeni podatki, pogleda, ki vsebuje vizualizacijo pogleda, in krmilnika, ki skrbi za povezavo med uporabnikom in sistemom. Priljubljen je postal pri razvoju spletnih aplikacij leta 2003 z ogrodjem Ruby on Rails.

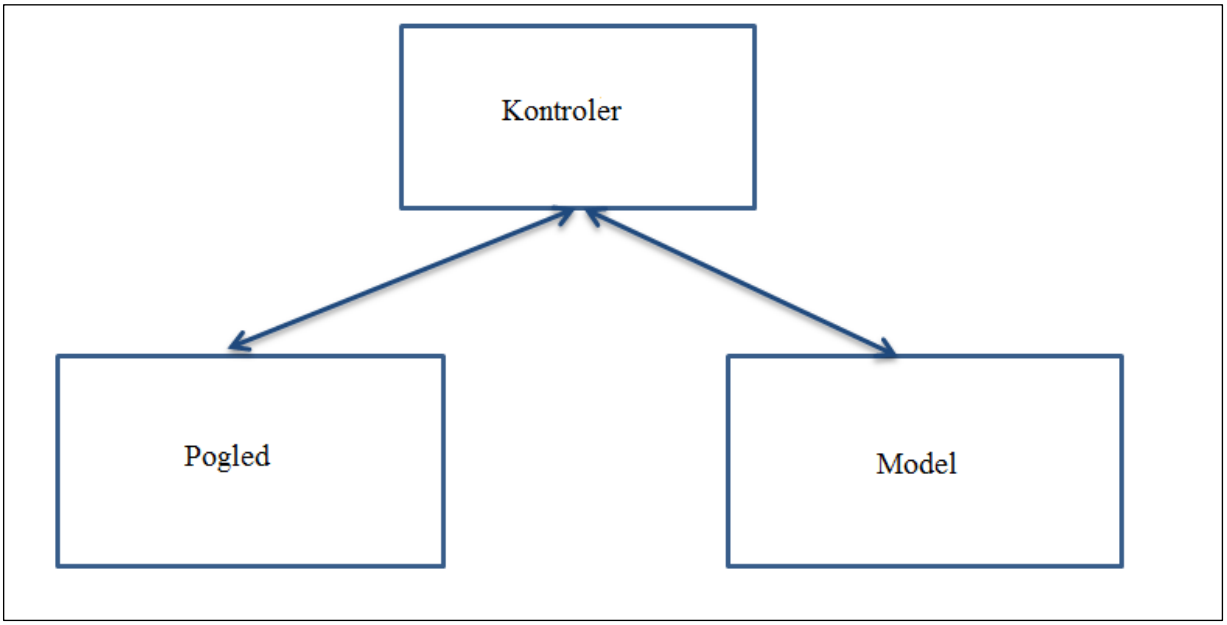
Podjetje Microsoft je svojo različico ogrodja MVC predstavilo leta 2009, poimenovali so ga Asp.Net MVC. Osnutek ogrodja je pripravil glavni razvijalec Scott Guthrie (Freeman, 2013) in ga predstavil na konferenci v mestu Redmond jeseni 2007.

Po predstavitvi se je pokazalo veliko zanimanja zanj, zato so se lotili njegovega razvoja. MVC ni klasično Microsoftovo razvojno ogrodje, ponujeno je kot odprtokodna rešitev. Pred končno različico je bilo devet poskusnih različic, kar je omogočilo sodelovanje razvijalcev z željami in pripombami za izboljšave. Razvojno ogrodje je na voljo z vso programsko kodo, kar nam omogoča tudi popravljanje in spreminjanje ogrodja.

Pri razvoju programske opreme se veliko časa ukvarjamo s popravljanjem in posodabljanjem aplikacij. Programsko kodo je lažje popravljati, če so moduli med seboj šibko sklopljeni, saj tako lahko popravljamo določen modul brez vpliva na preostale module.

Vzorec MVC temelji na treh komponentah, z močnim poudarkom na objektnem načinu razvoja programske opreme. Model vsebuje metode, ki skrbijo za branje in spreminjanje podatkov, krmilnik pa skrbi za delovanje aplikacije, odziva se na uporabniške zahteve in posreduje odgovore s pomočjo pogledov. Slika 7 predstavlja gradnike vzorca MVC.

Slika 7: Vzorec MVC



Vir: M. Yener & A. Theedom, Professional Java EE Design Patterns, 2015, str. 121.

Glavne prednosti uporabe pristopa MVC so popoln nadzor nad kodo za označevanje nadbesedila (angl. *hypertext markup language*, v nadaljevanju HTML), ki jo dobimo s

pomočjo pogledov, možnost prijaznejših spletnih naslovov (angl. *friendly URL*), lažje testiranje uporabniških vmesnikov in programske kode s pomočjo testov. V modernih spletnih aplikacijah mora biti koda HTML neodvisna od brskalnika, standardizirana in prilagojena tudi za uporabo na mobilnih napravah. Prijaznejši spletni naslovi so pomembni zaradi spletnih iskalnikov.

Model skrbi za poslovno logiko v aplikaciji, v njem dostopamo do baze, kličemo zunanje spletne storitve. Krmilnik povezuje model in pogled in ločuje uporabniški vmesnik v pogledu od poslovne logike v modelu ter skrbi za odgovarjanje na zahteve v uporabniškem vmesniku. Pogled vsebuje uporabniški vmesnik, v njem so shranjeni: koda HTML, datoteke, ki skrbijo za obliko, in knjižnice Javascript.

Model, krmilnik in pogled so ločeni, kar nam omogoča testiranje tudi samo dela aplikacije. MVC nam v osnovi ponuja tristopenjsko arhitekturo, posledično se zmanjša podvajanje kode, lažje tudi razdelimo naloge pri razvoju. Pri aplikaciji, namenjeni različnim odjemalcem, se naredi več pogledov, krmilnikov in modelov pa ni treba spreminjati. V današnjem času se pojavljajo zahteve za delovanje aplikacij na različnih odjemalcih, kot so spletni brskalniki, mobilne naprave in tablični računalniki.

MVC je dodatek in tudi alternativa klasičnemu razvoju z ogrodjem Asp.Net.

2.5 Ogrodje Asp.Net WebForms

Zaledno aplikacijo za upravljanje dokumentov v aplikaciji Dokumentni Sistem sem razvil s pomočjo ogrodja Asp.Net WebForms. Tehnologija WebForms je zrela in je v uporabi od leta 2002.

Leta 1997 se je podjetje Microsoft odločilo za razvoj nove platforme za razvoj spletnih strani, naloga je bila dodeljena dvema mladima raziskovalcema. Mark Anders in Scott Guthrie sta se poskušala pri razvoju nove platforme čim bolj približati željam razvijalcev (Novak et al., 2010).

Prva različica razvojne platforme Asp.Net je prišla na tržišče leta 2002, in sicer kot del prve različice ogrodja .Net. Microsoft je naredil spletno platformo, pri kateri je delo podobno kot pri klasičnih namiznih aplikacijah. Kontrole lahko postavimo na spletno stran brez potrebnega znanja jezika HTML. Razvijalci, vajeni dela z namiznimi aplikacijami, so se lažje naučili nove tehnologije.

Ogrodje Asp.Net je naslednik klasičnega ASP-ja, pri katerem sta vsebina in prikaz združena. Glavne prednosti so v možnostih pisanja bolj pregledne programske kode kot v klasičnem ASP-ju; za razvoj imamo na voljo veliko programskih jezikov, med njimi sta najbolj znana Visual Basic in C#. Zmanjšali so tudi težave s pomnilnikom (angl. *memory leak*) in izboljšali robustnost ter poenostavili odpravljanje napak.

Ogrodje je namenjeno hitremu razvoju aplikacij in ponuja velik nabor spletnih gradnikov, s katerimi na hiter in enostaven način zgradimo aplikacijo. Na voljo so tudi spletni gradniki drugih ponudnikov, na SURS-u uporabljamo gradnike podjetja Telerik. Ti gradniki nam omogočajo hiter razvoj, primerni so predvsem za interne aplikacije.

2.6 Primerjava ogrodij WebForms in MVC

Na SURS-u pri razvoju uporabljamo ogrodji MVC in WebForms. V nadaljevanju bom predstavil prednosti in slabosti obeh tehnologij.

Prednosti tehnologije Web Forms so (Berardi, Katawazi, & Bellinaso, 2009):

- zrela tehnologija,
- možnosti hitrega razvoja aplikacij (angl. *rapid application development*),
- dober vmesnik v orodju Visual Studio,
- bogata zbirka spletnih gradnikov različnih ponudnikov,
- podoben razvoj kot pri namiznih aplikacijah.

Slabosti tehnologije Web Forms so (Berardi et al., 2009):

- združena koda za prikaz in poslovno logiko v eni datoteki,
- težje izvajanje testov programske kode zaradi združene kode za prikaz in poslovno logiko,
- spletni gradniki so lahko zelo potratni in zelo povečajo velikost spletne strani.

Spletni gradniki uporabljajo za shranjevanje podatkov skrita HTML-polja (angl. *ViewState*). Vsebina skritih HTML-polj lahko zelo naraste, največkrat se to pokaže pri bolj obsežnih spletnih gradnikih. Spletni strežnik nima povezave med različnimi zahtevki, zato potrebujemo skrita HTML-polja za ohranjanje sledljivosti med zahtevki.

Tehnologija Web Forms je bila pri razvijalcih vse slabše sprejeta zaradi pomanjkanja nadzora nad kodo HTML. Pri razvoju aplikacij uporabljamo spletne gradnike, na njihovo kodo HTML pa nimamo vpliva.

Težave so se pojavljale tudi pri razvoju spletnih strani, namenjenih mobilnim napravam. Razvijalci so se vse pogosteje odločali za druge spletne tehnologije, ki so uporabljale pristop MVC. Podjetje Microsoft se je leta 2007 odločilo narediti novo ogrodje z imenom Asp.Net MVC z namenom lažjega testiranja spletnih vmesnikov in večjega nadzora nad kodo HTML.

Prednosti platforme MVC v primerjavi s tehnologijo Web Forms so (Berardi et al., 2009):

- boljši nadzor nad kodo HTML,
- ločevanje poslovne logike od prikaza – lažje izvajanje testov programske kode,
- enostavna integracija z ogrodji Javascript,

- omogoča uporabo različnih načinov prikazovanja,
- lažja prilagoditev spletnih strani za mobilne naprave.

Slabosti platforme MVC v primerjavi s tehnologijo Web Forms so (Berardi et al., 2009):

- drugačen koncept razvoja, težje razumljiv za obstoječe razvijalce Web Forms,
- potrebujemo vsaj osnovno znanje tehnologij HTML in Javascript,
- manjši nabor gradnikov,
- ne moremo nadgraditi obstoječe aplikacije Web Forms.

3 PRENOVA APLIKACIJE ZA UPRAVLJANJE Z DOKUMENTI

Vizija SURS-a je pravočasno objavljanje kakovostnih in mednarodno primerljivih statističnih podatkov. S pomočjo referenčnih metapodatkov bodo podatki še bolj kakovostno interpretirani, mogoča bo še kakovostnejša primerjava med podatki različnih raziskovanj ali skozi različna časovna obdobja (SURS, 2015).

Na SURS-u smo se odločili razviti manjši dokumentni sistem z namenom združitve različnih manjših podsistemov za upravljanje dokumentov na spletni strani. Za shranjevanje dokumentov se je uporabljalo več različnih sistemov, nekateri dokumenti so se nahajali zgolj v mapah na datotečnem sistemu brez potrebnih metapodatkov. Z aplikacijo smo centralizirali upravljanje z dokumenti na prenovljeni spletni strani.

Dokumentni sistem sem razvijal iterativno. Odločili sem se za prototipni pristop, kajti zahteve niso bile povsem jasne. Struktura podatkovne baze se je zelo spreminjala, kajti dodajale so se nove funkcionalnosti. Odločil sem se za razvoj podatkovnega sloja s pomočjo ogrodja ORM.

3.1 Dokumentni sistem

Dokumentni sistemi (Medvešek, 2013) so namenjeni učinkovitemu upravljanju z dokumenti podjetja.

Na intranetu uporabljamo rešitev Microsoft Sharepoint kot dokumentni sistem, namenjen shranjevanju internih gradiv in drugih dokumentov. Dokumentni sistem Sharepoint je bil za naše potrebe preobsežen in manjkale so mu določene funkcionalnosti. Potrebovali smo sistem, pri katerem za določen dokument lahko nastavimo čas objave. To je v portalu Sharepoint sicer izvedljivo, vendar zahteva dodaten razvoj. Na SURS-u smo se odločili za razvoj nove lastne rešitve, namenjene upravljanju z dokumenti.

Downing (2006) navaja prednosti uporabe aplikacije za upravljanje z dokumenti:

- hitrejše in lažje iskanje informacij,

- boljše možnosti izmenjave dokumentov z možnostjo dostopa več uporabnikov do istega dokumenta,
- višja stopnja varnosti dokumentov,
- večja zaščita dokumentov zaradi boljših možnosti varnostnega kopiranja,
- boljše poročanje, saj se informacije nahajajo v centraliziranem repozitoriju s polno revizijsko sledjo,
- prihranek prostora za shranjevanje,
- boljše možnosti sprejemanja odločitev zaradi enotnega shranjevanja in dostopa do informacij.

Rudolf in Zorman (2004) opozarjata, da lahko podjetja ob podpori elektronskih sistemov za obvladovanje dokumentov zmanjšajo nekatera tveganja, ki so posledica:

- neučinkovite organizacije dela,
- pomanjkljivosti pri obvladovanju različic,
- slabega nadzora sprememb,
- nepooblaščenih dostopov,
- izgube dokumentov,
- neupoštevanja sistema zagotavljanja kakovosti.

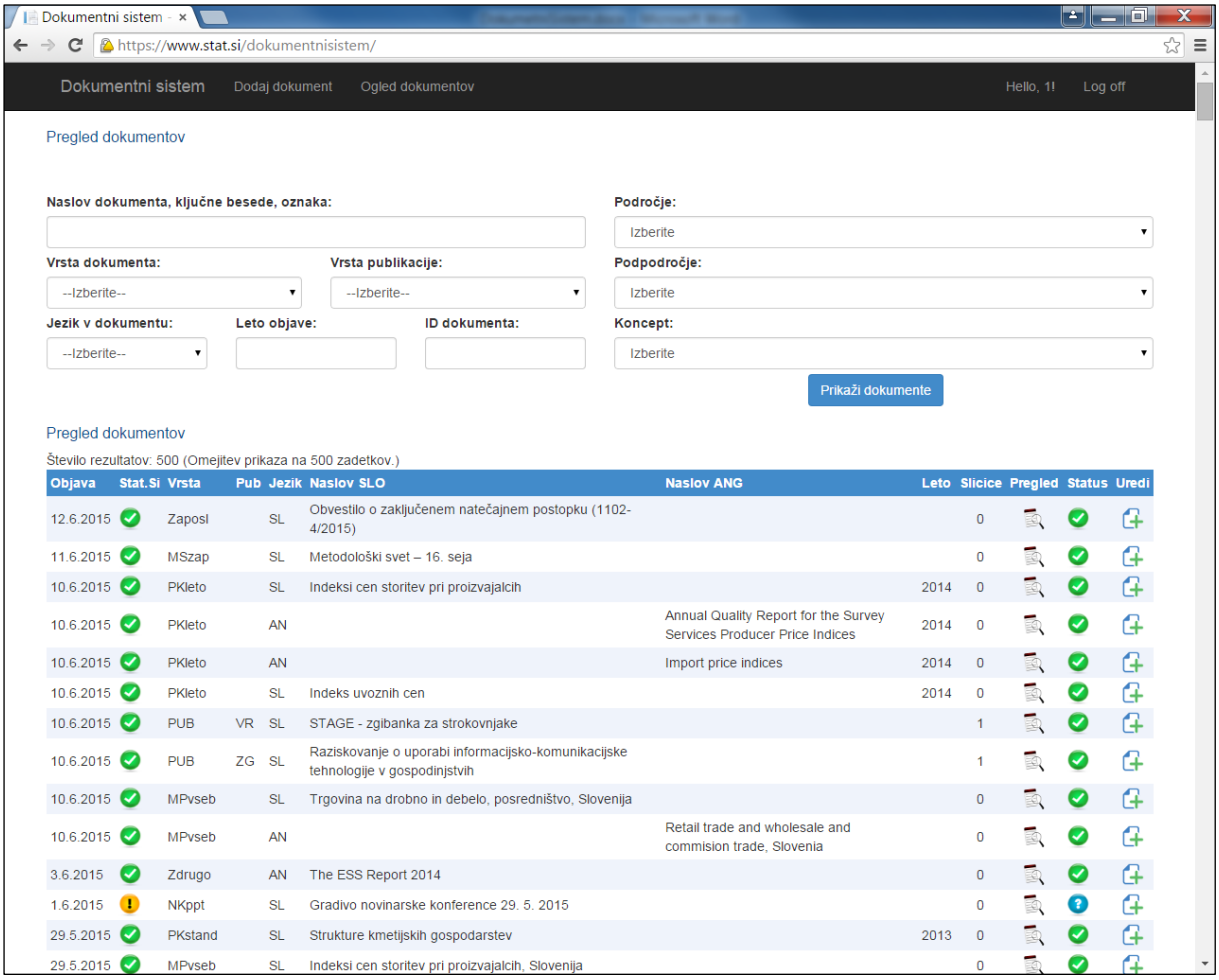
Umek Luzar (2005, str. 18) pravi: »Pravilno definiran nabor atributov na dokumentu je zelo pomemben za uspešno uporabo dokumentacijskega sistema. Neprimerno definirani atributi lahko povzročajo probleme pri iskanju dokumentov in s tem nedostopnost do njihove vsebine. Zatorej je potrebno zagotoviti, da so tisti atributi, ki so zelo pomembni pri iskanju dokumenta ali pripravi poročil za vodstvo, vedno obvezni pri izdelavi dokumenta v sistem«.

Dokumentni sistem je enostaven za uporabo, zaposleni so ga hitro usvojili in ga po njihovih zagotovilih z veseljem uporabljajo. Glavna pomanjkljivost sistema je v tem, da ne omogoča prikaza zgodovine popravkov istega dokumenta (angl. *versioning*). Pomanjkljivost nameravam odpraviti v naslednji različici aplikacije. Dokumentni sistem omogoča shranjevanje različnih vrst dokumentov – med najbolj pogostimi na SURS-u so dokumenti pisarniške zbirke Microsoft Office in datoteke s končnico pdf.

V dokumentni sistem se shranjuje 40 različnih metapodatkov. V sistemu je omogočeno iskanje po enajstih različnih metapodatkih.

Uporabniki se v aplikacijo prijavijo s pomočjo domenskega uporabniškega imena. Sistem omogoča sledenje spremembam. Podatek, kdo in kdaj je popravljal določen dokument, se shrani v podatkovni tabeli. V sistemu nimamo shranjenih dokumentov, ki bi bili kritični z varnostnega vidika. Vsi uporabniki vidijo vse dokumente, v prihodnji različici aplikacije bom najverjetneje dodal možnost dodajanja uporabniških skupin za lažje in boljše upravljanje s pravicami uporabnikov. S tem bi zagotovil dostop do določenih dokumentov le pooblaščenim uporabnikom. Slika 8. prikazuje zaslonsko masko za iskanje po dokumentih.

Slika 8: Iskanje po dokumentih



Na podatkovnem strežniku se redno izvaja izdelava varnostnih kopij podatkovne baze (angl. *backup*) in ta baza se tudi mesečno arhivira.

Uporabniški vmesnik je narejen s pomočjo tehnologije Asp.Net WebForms, prikaz datotek pa v tehnologiji MVC – obe bom opisal v nadaljevanju. Za tehnologijo WebForms smo se odločili zaradi izkušenj in hitrega in enostavnega razvoja. Tehnologijo MVC smo uporabili pri prikazu datotek zaradi bolj prilagojenih spletnih naslovov in posledično boljših rezultatov v iskalnikih.

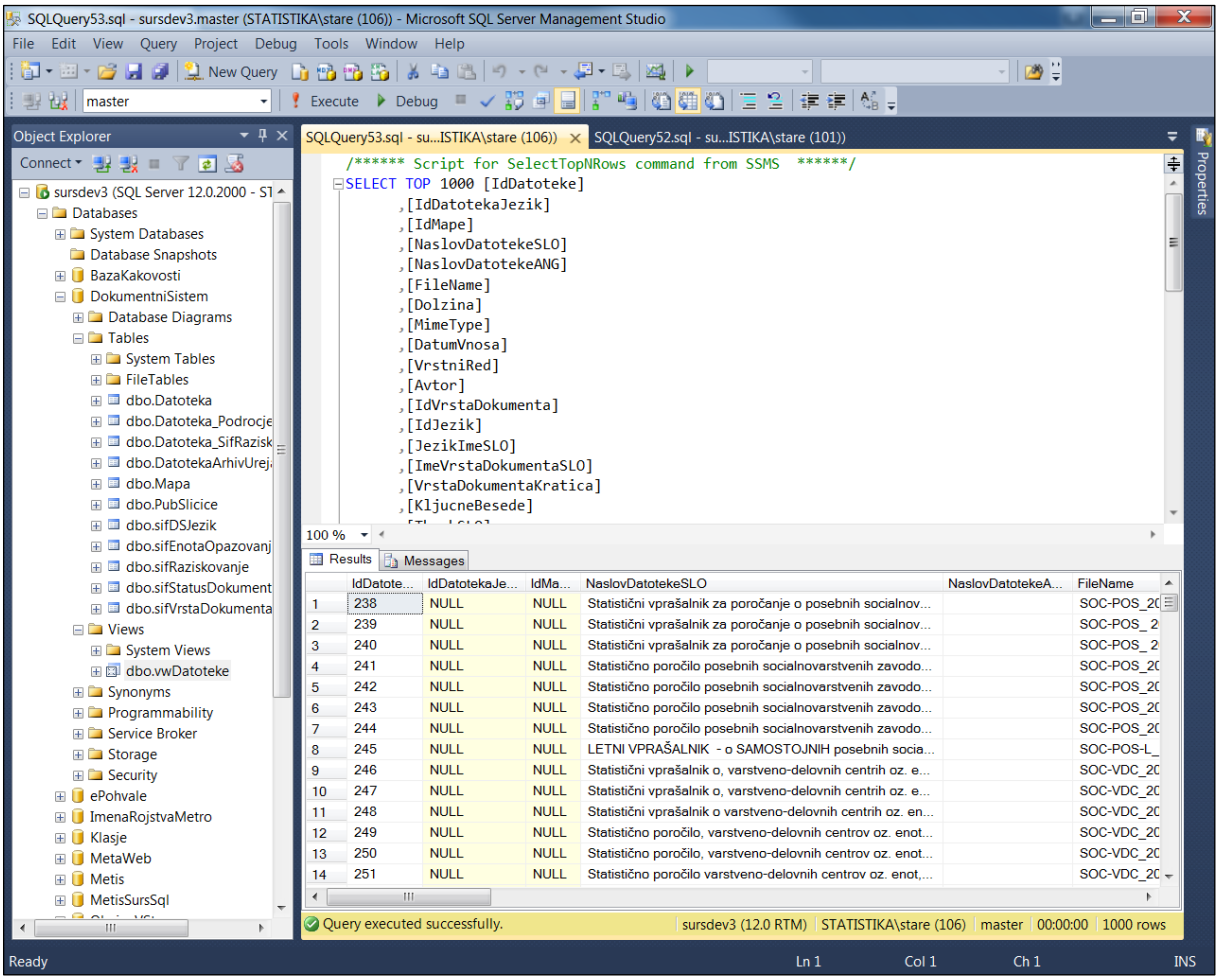
3.2 Podatkovni strežnik Microsoft SQL Server

Na SURS-u skoraj vse aplikacije uporabljajo relacijsko podatkovno bazo, predvsem podatkovni bazi Oracle in Microsoft SQL Server. Pri razvoju aplikacije Dokumentni sistem smo uporabili podatkovni strežnik Microsoft SQL Server.

Podatkovni strežnik SQL Server je na voljo od leta 1988, leta 2000 je s pojavom različice SQL Server 2000 postal pomemben igralec na tržišču podatkovnih baz. Izboljšani sta bili zanesljivost in razširljivost (Dewson, 2012).

Microsoft SQL Server je sestavljen iz treh glavnih komponent: podatkovne baze, orodij, namenjenih poslovnemu obveščanju, in orodja Management Studio, namenjenega poizvedovanju po podatkovni bazi. Uporabljajo ga tako razvijalci kot skrbniki podatkovne baze. Slika 9 prikazuje orodje Management Studio.

Slika 9: Management Studio



Podatkovni strežnik Microsoft SQL Server se je od svoje prve različice iz leta 1989 zelo razvil. Podjetja se večinoma odločajo med tremi glavnimi ponudniki podatkovnih baz: Oracle, IBM in Microsoft. Podatkovni bazi Oracle in IBM sta bili v preteklosti veliko bolj napredni in robustni kot SQL Server (Dewson, 2012).

Trenutno je na voljo različica SQL Server 2014, ki smo jo tudi mi uporabili pri razvoju aplikacije Dokumentni sistem. Glavne prednosti, ki jih ima SQL Server pred podatkovno bazo Oracle, so predvsem veliko boljša orodja, ki so na voljo poleg strežnika. SQL Server je na voljo v različnih izdajah – različica Enterprise omogoča vse funkcionalnosti, glavne prednosti so predvsem v visoki razpoložljivosti in poslovnem obveščanju. SURS se je odločil za različico Standard, predvsem zaradi cene.

3.3 Prenos podatkov med strežniki

SURS uporablja za aplikacijo Dokumentni sistem dva ločena podatkovna strežnika, in sicer zaradi varnosti in lažje prijave v sistem. Prijava poteka s pomočjo domenskega uporabniškega imena (angl. *Active Directory*). Uporabnik za prijavo ne potrebuje dodatnega gesla in gesel nam tudi ni treba shranjevati v podatkovno bazo.

Na notranjem strežniku se dodajajo in urejajo datoteke. V notranjem omrežju je notranji strežnik namenjen tudi prikazovanju datotek. Zunanji uporabniki sistema dostopajo do zunanjega strežnika, ki je namenjen spletu.

Prenos podatkov med notranjim in zunanjim strežnikov poteka s pomočjo replikacije podatkov. Na SURS-u uporabljamo replikacijo za prenos podatkov iz notranjega strežnika, ki je dostopen zgolj iz notranjega omrežja, na strežnik, namenjen prikazovanju podatkov na spletu. Odločili smo se za uporabo transakcijske replikacije, ki poteka med dvema podatkovnima strežnikoma Microsoft SQL Server 2014.

Prenos podatkov med različnimi podatkovnimi strežniki je pogosta naloga v poslovnem okolju. Razlogi za prenos podatkov so lahko različne lokacije strežnikov, izdelava varnostnih kopij, kopija podatkov, namenjena pripravi raznih poročil.

Replikacija podatkov je mnogo več kot le premikanje podatkov med različnimi podatkovnimi bazami. Orodja za replikacijo morajo pokrivati širok nabor poslovnih zahtev in zagotavljati prenašanje podatkov brez prevelike obremenitve izvornega podatkovnega strežnika. Replikacija lahko tudi zagotavlja večjo razpoložljivost in izboljšano odzivnost, kadar je osnovna podatkovna zbirka preveč obremenjena.

Podjetja imajo različne razloge za uporabo replikacije. Premikanje podatkov je pogosta naloga v poslovnih okoljih, kopijo podatkov potrebujemo zaradi priprave poročil, podatkovnih skladišč, varnostnih kopij idr.

Pri replikaciji z uporabo podatkovnega strežnika Microsoft SQL Server se uporabljajo trije strežniki (LeBlanc, 2013):

- založnik – podatkovni strežnik z izvornimi podatki.
- Naročnik – podatkovni strežnik, na katerega se kopirajo podatki.
- Distributer – podatkovni strežnik, ki shranjuje spremembe.

Podatkovni strežnik SQL Server omogoča več različnih vrst replikacij. Najenostavnejša je replikacija Snapshot (posnetek). Primerna je za prenos manjših količin podatkov, kadar se podatki ne spreminjajo pogosto in so lahko zastareli. Replikacija Snapshot naredi posnetek vseh podatkov, nato se pošljejo naročniku s pomočjo distributerja. V primeru prenosa velike količine podatkov je proces lahko dolgotrajen. Replikacija se večinoma izvaja periodično (dnevno, tedensko ...), zato je lahko velika zakasnitev pri podatkih. Replikacija Snapshot je

najenostavnejši tip replikacije, primeren za manjše podatkovne baze in za baze, pri katerih se podatki ne spreminjajo pogosto. Naredi posnetek podatkov v podatkovni bazi. Največkrat se uporablja za občasne prenose podatkov, npr. tedenske, mesečne itd.

Druga možnost je uporaba transakcijske replikacije. Ta se uporablja takrat, ko se podatki pogosto spreminjajo in potrebujemo podatke skoraj v realnem času. Transakcijska replikacija je namenjena prenosu podatkov, pri katerem so zakasnitve pri podatkih lahko minimalne in se podatki pogosto spreminjajo. Prenašajo se podatkovne transakcije, shranjene v transakcijskem dnevniku (angl. *transaction log*). Transakcije se preberejo iz dnevnika in se nato s pomočjo distributerja pošljejo naročniku, kjer se izvedejo.

Zakasnitev (angl. *latency*) je pomemben vidik pri izbiri replikacije podatkov. Opredeljena je kot količina časa, ki jo podatki potrebujejo za pot med izvorom in ciljem. Manjša kot je zakasnitev, boljše je za prenos podatkov.

Sistem replikacije mora zadostiti naslednjim poslovnim zahtevam (Lavrič, 2004):

- visoka dostopnost podatkov: replikacija mora biti zanesljiva in ne sme izpostavljati poslovnih operacij računalniškim napakam.
- Konsistentna dostava informacij: distribucijski sistem mora zaščititi podatkovno integriteto.
- Visoka zmogljivost: replikacija ne sme bremeniti podatkovnega vira, izkoristiti mora učinkovitost mreže in omogočati optimizacijske metode pri lokalnem dostopanju do podatkov.
- Enostavno centralizirano skrbništvo: skrbnik mora biti sposoben enostavnega upravljanja z distribuiranimi komponentami replikacijskega sistema iz enega delovnega mesta.
- Heterogen dostop do izvornih podatkov: replikacija naj bi bila sposobna seliti podatke skozi podatkovne vire različnih proizvajalcev.
- Lokalna avtonomija: vsaka stran, ki dobiva podatke prek replikacije, bi se morala sama prosto odločati, kako bo te podatke prikazovala, do njih dostopala in jih spreminjala.

3.4 Uvoz podatkov

Na SURS-u imamo različne sisteme, iz katerih črpamo podatke, potrebne za delovanje aplikacij. Dokumentni sistem uporablja podatke iz podatkovne baze MetaWeb, v kateri so shranjeni metapodatki, namenjeni statističnim objavam. Pri uvozu podatkov in metapodatkov iz drugih strežnikov si pomagamo z orodjem Microsoft Integration Services.

Polnjenje podatkov (angl. *extract transform load,* v nadaljevanju ETL) predstavlja procese, ki pridobivajo podatke iz transakcijskih sistemov in drugih različnih virov, jih preoblikujejo glede na poslovne potrebe in jih shranijo v ciljno podatkovno bazo.

V podjetjih so podatki shranjeni v zelo različnih sistemih. Integracija podatkov je zahteven problem zaradi različnih podatkovnih virov, različnih formatov podatkov in slabe kakovosti

podatkov. Podatki se shranjujejo v različni sistemih in v različne vrste podatkovnih baz. Cena diskovnih polj je vsak dan nižja, zato shranjevanje velike količine podatkov ne predstavlja več tako velikega stroška za podjetje. Tolikšna količina podatkov zahteva dobra integracijska orodja, kajti podatke želimo analizirati čim hitreje.

Podatki se v podjetjih nahajajo v različnih oblikah in so različne kakovosti. Oddelki v podjetju uporabljajo različne aplikacije, te pa uporabljajo različne načine shranjevanja podatkov. V večjih mednarodnih podjetjih so podatki tudi geografsko razpršeni.

Proces pridobivanja podatkov poskrbi za prevzem podatkov iz različnih podatkovnih virov, ki imajo različno shranjene podatke; nekateri podatki so v relacijskih podatkovnih bazah (Excel, tekstovnih datotekah itd.). Različni podatkovni viri zahtevajo različne tehnike za pridobivanje podatkov, proces pridobivanja podatkov poskrbi za pretvorbo v format, ki ga lahko proces transformiranja uporabi.

Proces transformiranja s pomočjo različnih pravil in funkcij poskrbi za pretvorbo izvornih podatkov v obliko, ki je primerna za nalaganje v podatkovna skladišča ali druge podatkovne baze. Proces lahko vsebuje združitev podatkov iz različnih podatkovnih virov, združevanje, spreminjanje strukture podatkov, čiščenje podatkov, odstranjevanje dvojnikov itd.

Pred pojavom orodij ETL je bila izdelava skript v programskih jezikih, kot sta Cobol in SQL, edini način za prenos podatkov iz različnih podatkovnih virov. Prva generacija orodij ETL je s pomočjo ustvarjenega podatkovnega toka generirala skripte SQL za prenos podatkov. Glavna slabost prve generacije je bila v podpori zgolj nekaterih podatkovnih baz (Rodrigues, Coles, & Dye, 2012).

Najnovejša orodja nudijo podporo za podatkovne baze, datoteke in spletne storitve. Delo s podatki in transformacije podatkov potekajo neodvisno od podatkovnega vira, kar jim omogoča relativno enostavno dodajanje novih podatkovnih virov.

Na SURS-u imamo večino enostavnejših skript napisanih v jeziku PL/SQL, ki je razširjen programski jezik SQL in je proceduralni jezik z možnostjo uporabe programskih konstruktov znotraj stavkov SQL. Namenjen je pisanju poslovne logike z uporabo procedur in funkcij.

Glavne slabosti ročno napisanih skript so pomanjkanje metapodatkov, težje vzdrževanje, manjša podpora lovljenju napak in daljši cikel razvoja. Na SURS-u sem opazil, da razvijalci uporabljajo različna programska orodja za pisanje skript za prenos podatkov, kar še poveča zahtevnost vzdrževanja in popravljanja.

Naprednejša orodja ETL so manjšim podjetjem cenovno nedostopna, zato tam največkrat uporabljajo skripte za prenos podatkov.

Učinkovito orodje ETL zamenja nepregledne procedure za prenos podatkov, ki jih razvijamo s pomočjo različnih programskih jezikov znotraj urada. Z uporabo tega orodja standardiziramo postopke za delo s podatki, jih dokumentiramo, lažje je tudi vzdrževanje. Orodja imajo v

večini primerov uporabniške vmesnike, ki poenostavijo gradnjo podatkovnih tokov. Imajo tudi možnost branja metapodatkov iz podatkovne baze in orodij za modeliranje. Metapodatki nam opišejo podatkovne tabele, stolpce, podatkovne tipe in druge lastnosti podatkovne baze.

Na SURS-u uporabljamo ETL-orodje SQL Server Integration Services. Zanj smo se odločili, ker je del podatkovnega strežnika Microsoft SQL Server in ker ni dodatnih stroškov z nakupom licenc. Orodje ETL mora biti robustno, učinkovito, varno in enostavno za vzdrževanje. Robustnost je podprta z uporabo transakcij, ki zagotavljajo konsistentnost podatkov, z možnostjo lovljenja napak, pošiljanja elektronske pošte v primeru težav itd.

SQL Server Integration Services je dodatek k strežniku Microsoft SQL Server in je namenjen (Rodrigues, Coles, & Dye, 2012):

- pridobivanju podatkov iz različnih podatkovnih virov, kot so datoteke XML, preglednice v Excelu, spletne storitve, relacijske in druge podatkovne baze. Če nam vmesnik ne zadošča, si lahko pomagamo s programskim jezikom .Net in razvijemo dodatek za nepodprt vir.
- Preverjanju podatkov glede na podana pravila skozi proces ETL. Podatke lahko preverjamo s pomočjo različnih metod, ki preverjajo ujemanje vzorcev v nizih, ujemanje številskih vrednosti z določeno zalogo vrednosti idr.
- Čiščenju podatkov, prepoznavi nepravilnih podatkov in spreminjanju nepravilnih podatkov z vnaprej določenimi vrednostmi. Na primer spreminjanju negativnih števil v vrednost nič ali odstranitvi odvečnih presledkov iz niza znakov.
- Odstranitvi podvojenih podatkov. V nekaterih primerih se štejejo za podvojene podatke vrstice, ki se ujemajo v celoti, v drugih je dovolj že ujemanje v določenem podatkovnem polju, kot je npr. telefonska številka.
- Nalaganju podatkov v različne podatkovne baze, datoteke, spletne storitve in druge podatkovne shrambe. Ponuja bogat nabor vmesnikov za nalaganje podatkov v večino priljubljenih podatkovnih baz.

3.5 Upravljanje izvorne kode

Pri razvoju in vzdrževanju aplikacij potrebujemo shrambo izvorne kode. Pod pojmom izvorna koda mislimo na programsko kodo, nastavitvene datoteke, dokumentacijo itd. V shrambo je priporočljivo shranjevati vse, kar potrebujemo za prevajanje in nameščanje aplikacije.

Shrambe programske kode so lahko zelo različne. Pri nekaterih manjših projektih, pri katerih ne potrebujemo shranjevanja različic datotek (zgodovine), lahko uporabljamo zgolj datotečni sistem. Pri večjih projektih je priporočljiva uporaba bolj naprednih orodij za verzioniranje, ki omogočajo tudi združevanje istih datotek ter reševanje konfliktov znotraj datotek.

Shramba programske kode se uporablja za deljenje programske kode med različnimi razvijalci. Razvijalci si iz shrambe naložijo zadnjo različico programske kode, dopolnjujejo

programsko kodo in jo nato naložijo nazaj v shrambo. Če so datoteke spremenili drugi razvijalci, lahko pride do konflikta. Enostavnejše konflikte razreši orodje samo, pri bolj zahtevnih moramo ročno določiti, kateri del programske kode naj se uporabi. Tak pristop zagotavlja vsem razvijalcem dostop do zadnje različice programske kode.

Shramba omogoča tudi prikaz zgodovine sprememb pri določeni datoteki. Zgodovina sprememb nam omogoča lažje reševanje napak v programski kodi, kajti pri vsaki spremembi vidimo tudi ime razvijalca, ki je to spremembo izvedel.

Pri večjih razvojnih projektih sodeluje večje število razvijalcev, ki dodajajo in spreminjajo različne datoteke. Popravkom v programski kodi je treba slediti, pri čemer so nam v veliko pomoč orodja za verzioniranje programske kode. Osnovni nalogi orodij za verzioniranje sta shranjevanje sprememb na datotekah in združevanje popravkov različnih razvijalcev na isti datoteki.

Prednosti uporabe orodij so v lažjem shranjevanju varnostnih kopij. Razvijalci morajo redno dodajati izvorno kodo, s čimer se zmanjša možnost izgube programske kode. Orodja tudi povečajo produktivnost, kajti olajšajo integracijo izvorne kode različnih razvijalcev. Razvijalci lažje dodajajo izvorno kodo, brez strahu pred povzročeno škodo. Kadar popravki niso primerni, lahko na dokaj enostaven način dobimo prejšnje delujoče stanje.

Pri razvojnem procesu je pomembno vedeti, kdo in kdaj je dodal kakšen del programske kode. Po daljšem obdobju tako lažje odpravimo napako v kodi.

Na SURS-u razvijamo kode predvsem z orodjem Visual Studio, ki ima možnost integracije s shrambo Microsoft Team Foundation Server. Lahko pa se povežemo tudi na odprtokodne shrambe, med katerimi je priljubljena Subversion – to orodje tudi sami uporabljamo in smo z njim zadovoljni.

3.5.1 Orodje Subversion

Leta 1986 je Dick Grune (Harrison, 2011) predstavil orodje z imenom Concurrent Versioning System, ki je omogočalo razvijalcem delo na različnih datotekah in nato združevanje spremenjenih datotek. Orodje je predhodnik odprtokodnega orodja Subversion.

Na SURS-u smo se odločali med uporabo sistema Microsoft TFS in orodja Subversion. Izbrali smo Subversion, kajti je brezplačno orodje in ima vse, kar potrebujemo za razvoj. Edina večja slabost, ki smo jo opazili, je slabša možnost integracije z razvojnim okoljem Visual Studio.

Subversion uporablja transakcije pri delu s podatkovno bazo. Težave smo imeli z združevanjem binarnih datotek – pri vsakem prevajanju programske kode je prišlo do sprememb, ki jih je bilo nato treba združiti. Binarne datoteke niso berljive, kot so tekstovne datoteke, zato je združevanje skoraj nemogoče. Vse binarne datoteke, ki se spreminjajo, smo odstranili iz shrambe izvorne kode.

Pri razvoju programske opreme pogosto storimo kakšno napako, verzioniranje nam omogoča pregled predhodnih različic, kar nam prihrani veliko časa in dela. Pri nekaterih manjših projektih nismo uporabljali orodja za verzioniranje, vendar se je to pokazalo kot napačen pristop.

3.6 Testiranje podatkovnega sloja in programske kode

V našem oddelku si pogosto zastavimo vprašanje, kako natančno naj preverjamo programsko kodo. V večjih podjetjih imajo zaposlene, katerih naloga je zgolj preverjanje programskih kod. Razvijalci moramo nenehno preverjati, ali naš izdelek ustreza zahtevam.

Razvijalci imamo različna mišljenja o tem, kdaj je bolje pisati teste. Nekateri pišejo teste po že napisani programski kodi, tudi sam jih pišem tako. V zadnjem času je zelo priljubljeno pisanje testov še pred pisanjem programske kode. Tak način dela imenujemo testno voden razvoj.

Za ogrodje .Net je na voljo veliko ogrodij za testiranje, na SURS-u uporabljamo ogrodje nUnit, ki je dokaj enostavno za uporabo in ima veliko koristnih funkcij.

Pri ročnem testiranju programske kode je veliko ponavljajočega dela. Programsko kodo testiramo s pomočjo razhroščevalnika, kar je zelo zamudno. Pri vsakem popravku v programski kodi moramo postopek testiranja ponoviti in preveriti tudi vso programsko kodo, na katero spremembe lahko vplivajo. Pri takem načinu testiranja hitro pozabimo na kakšen pomemben del programske kode in spregledamo nepravilno delovanje. Ročno testiranje programske kode je zelo nepriljubljeno pri razvijalcih.

Kljub napisanim testom moramo vseeno ročno preveriti delovanje programske kode. S testi poskušamo pokriti čim večji del kode, vendar je nemogoče doseči stoodstotno pokritost. Z ročnim testiranjem preverimo pravilnost delovanja mask, njihovo smiselnost in uporabniško izkušnjo. Smiselno je tudi preverjanje uporabe programske opreme z naročnikom. Sami lahko razvijemo sistem brez napak, vendar če je nerazumljiv končnemu uporabniku, nismo dosegli zadanega cilja.

Zaradi časovne stiske pri razvoju programske opreme velikokrat zanemarimo testiranje. Premalo testiranja lahko pripelje do zamude pri predaji programske opreme, kar poveča stroške razvoja, podjetje pa lahko izgubi verodostojnost. Testno voden razvoj pripelje tudi do boljše kakovosti programske kode, kar na dolgi rok zmanjša čas, potreben za spremembe, in število odkritih napak po predaji izdelka naročniku.

Za vsak projekt si želimo zmanjšati tveganja, ena od možnosti je testno voden razvoj. Z naborom različnih testov preverimo skladnost z zahtevami naročnika in lažje spremljamo napredek projekta. Vključitev naročnikov že na začetku razvojnega cikla je zelo pomembna, kajti tako lahko preverimo njihove želje in zahteve. Zadovoljne stranke pa so seveda ključ do uspeha podjetja (Osherove, 2014).

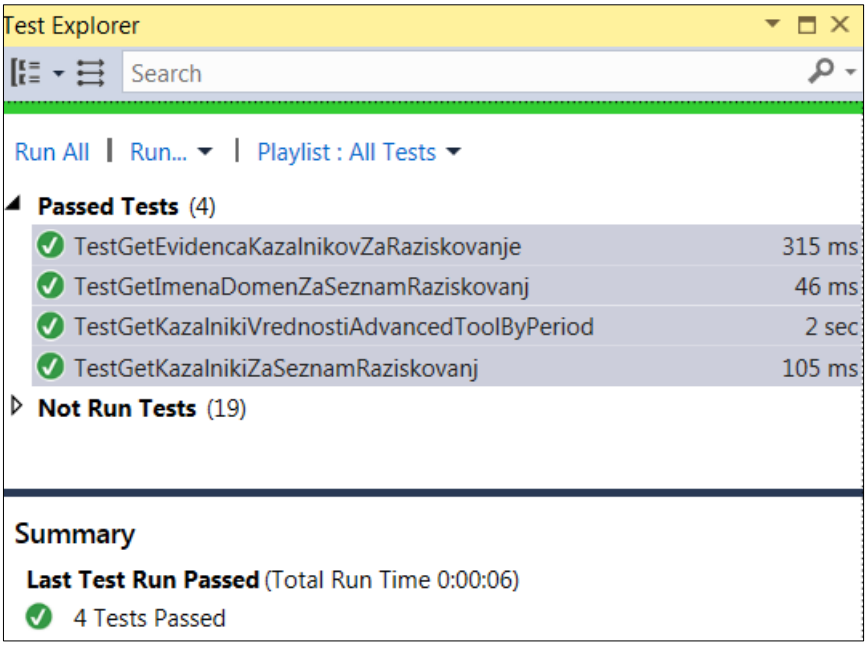
Kadar se testi uspešno izvedejo, se lahko lotimo novih ali preoblikujemo obstoječo kodo z namenom izboljšanja kakovosti, odstranitve podvojenih delov kode itd. Preoblikovanje lahko izvedemo po vsakem testu ali po več napisanih testih. Je dobra praksa pri razvoju programske opreme, kajti poskrbi za boljšo berljivost in lažje vzdrževanje kode. Testno voden razvoj je dober način za izboljšanje kakovosti kode in boljše razumevanje funkcionalnih zahtev metod.

3.6.1 Test enote

Test enote (angl. *unit testing*) ni novost pri razvoju programske opreme, pojavil se je že v sedemdesetih letih prejšnjega stoletja. Je preverjanje pravilnosti delovanja manjšega dela programske kode, največkrat ene metode. Za ogrodje .Net obstaja veliko dobrih ogrodij za testiranje, med njimi so nUnit, MbUnit in xUnit.

Po definiciji (Osherove, 2014) je test enote delček kode, največkrat metode, ki se sklicuje na drug del kode in preveri pravilnost nekaterih predpostavk. Test pade, če predpostavke niso pravilne. Slika 10 prikazuje nUnit.

Slika 10: nUnit



Ogrodja nam olajšajo delo pri pisanju testov. Teste pišemo na enak strukturiran način v vseh aplikacijah .Net. Ogrodja imajo tudi možnost ročnega zaganjanja testov ali zaganjanja testov iz ukazne vrstice. Po izvedenih testih lahko preverimo, koliko se jih je uspešno izvedlo. Ogrodje nUnit je odprtokodna rešitev in se pogosto posodablja. Namestimo ga na enostaven način s pomočjo dodatka Nuget. Uporabniški vmesnik je enostaven in nam pove, kateri testi so se uspešno izvedli in kateri ne. Testi enot se izvajajo zelo hitro, zato jih lahko pogosto izvajamo. Na voljo je zelo veliko ogrodij za testiranje za različne programske jezike. Na SURS-u razvijamo v ogrodju .Net, za katerega je na voljo kar nekaj ogrodij za testiranje. Mi uporabljamo ogrodje nUnit za testiranje programske kode.

Veliko ogrodij za testiranje izhaja iz ogrodja sUnit, ki ga je razvil Kent Beck (Osherove, 2014) leta 1998. Beck je predstavil koncept testov v programskem jeziku SmallTalk in ga prenesel v mnoge druge programske jezike kot zelo dobro prakso pri načrtovanju programske opreme.

Dodajanje testov v dokončano programsko kodo je veliko težje, kajti celoten sistem je treba preučiti in dopolniti s testi. Razvijalci imajo pogosto omejen čas za razvoj, zato velikokrat spregledajo pomembnost testiranja programske opreme oziroma prelagajo izdelavo testov na poznejši čas. Več kot je kode brez pokritosti s testi, težja naloga jih čaka pred predajo programske kode.

Dobri testi enot so (Osherove, 2014):

- vredni zaupanja. Razvijalci morajo zaupati testom in njihovim rezultatom. Zanesljivi testi nimajo hroščev v kodi in testirajo pomemben del kode.
- Enostavni za vzdrževanje. Testi, ki so težavni za vzdrževanje, lahko povzročijo zamudo pri celotnem projektu. Razvijalci bodo prenehali vzdrževati teste, ki so preveč zahtevni za popravljanje ali se preveč pogosto spreminjajo.
- Berljivi. Pri berljivosti gre predvsem zato, da se enostavno vidi, kaj je narobe v programski kodi. Testi naj bodo berljivi tudi zaradi sodelavcev, ki bodo pregledovali teste čez nekaj mesecev.

Razvijalci morajo testom zaupati, sicer niso smiselni. Najbolj primerni so kratki testi, ki so namenjeni zgolj eni stvari. Teste je treba tudi upoštevati in ne dvomiti v njihovo pravilnost.

Kot pri vsaki programski kodi se tudi pri testih dogajajo napake. Odkrite napake je treba čim prej odpraviti, vendar jih je zelo težko odkriti. Pogosto razvijalci popravljajo programsko kodo, ker so prepričani, da s testom ni težav. Tako povzročijo dodatne napake v kodi, namesto da bi odpravili napake v testu.

Zaposleni pogosto niso navdušeni nad spremembami v delovnem procesu in s strahom gledajo na spremembe. Podpora vodstva pri zahtevnih spremembah je zelo dobrodošla. Na SURS-u smo začeli z dodajanjem testov pri manjših projektih, sprva smo testirali zgolj ključne metode v programski kodi. Smiselno je začeti z vpeljavo novih metod dela pri manjših projektih, ki niso ključnega pomena za organizacijo. Zaposleni različno sprejemamo nove načine dela. V naši organizaciji smo imeli predvsem težave s starejšimi zaposlenimi, ki niso bili pripravljeni slediti spremembam v načinu dela. Vodstvo je ocenilo, da ni smiselno vztrajati, zato se testiranje programske kode ni preveč obneslo.

Pokritost nam pove, koliko kode je podprte s testi. Sama pokritost nam ne pomaga veliko, če ne preverimo tudi smiselnosti in pravilnosti izvajanja testov. Orodja za preverjanje pokritosti v .Net ogrodju so DotCover, PartCover, NDepend in NCrunch.

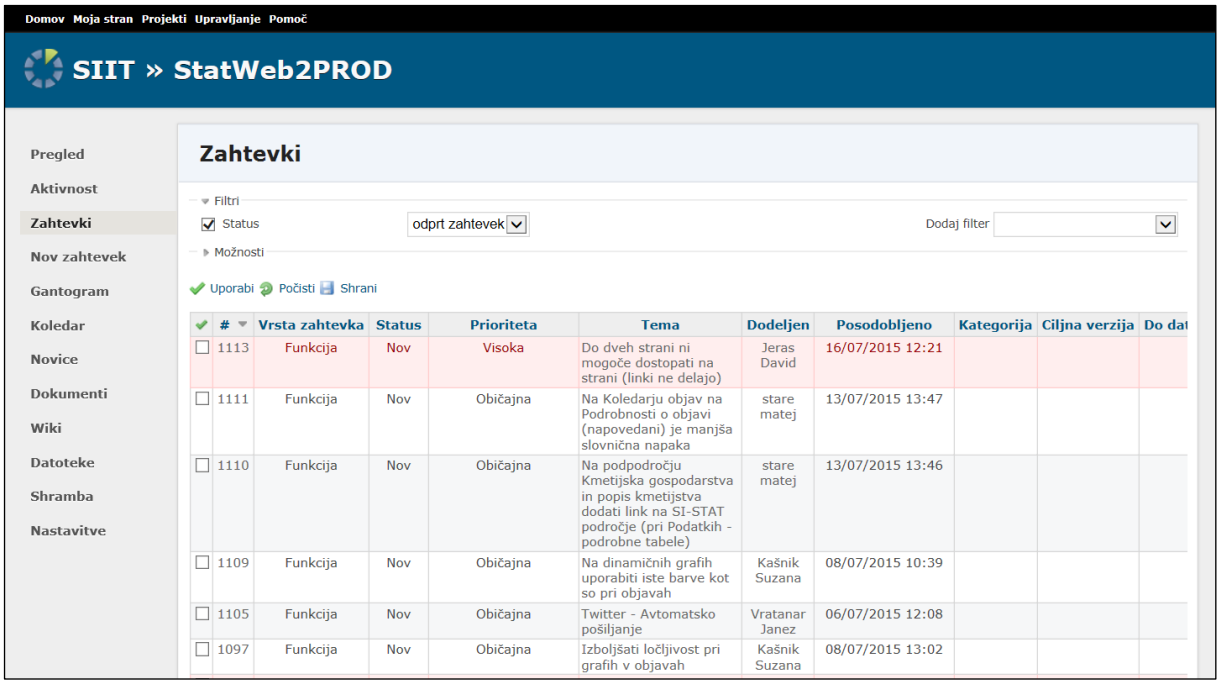
Pokritost kode pod dvajsetimi odstotki nas opozori na pomanjkljivosti pri pisanju testov. Spreminjanje kode v tem primeru lahko privede do napak, ki jih odkrijemo šele pri ročnem testiranju ali pri uporabi aplikacije, kar lahko zelo poveča stroške razvoja in zmanjša zadovoljstvo uporabnikov.

3.6.2 Testiranje podatkovnega sloja

Testiranje podatkovnega sloja je zahtevnejše kot testiranje zgolj programske kode. Uporabimo lahko t. i. navidezno podatkovno bazo, druga možnost je uporaba prave podatkovne baze. Podatkovno bazo moramo pred začetkom vsakega testiranja namestiti znova z začetnim stanjem, nato zaženemo teste in preverimo vrnjene vrednosti. Testiranje pravilnosti delovanja ogrodij ORM ni smiselno, saj jih že dovolj preverijo njihovi razvijalci. Pri testiranju podatkovne baze moramo zagotoviti ponovljivost testov.

Za izvedbo dobrega testa je pomembna dobra kombinacija vhodnih parametrov. Kadar opravljamo več testov, moramo biti pozorni tudi na vrstni red izvajanja. Rezultate testiranj je dobro shranjevati, shranjujemo jih lahko v datoteko ali podatkovno bazo. Teste, ki ne vrnejo pričakovanih rezultatov, moramo zabeležiti in popraviti shranjeno proceduro. Na SURS-u uporabljamo za beleženje hroščev v programski kodi spletno aplikacijo RedMine. Slika 11 prikazuje spletni vmesnik aplikacije RedMine.

Slika 11: Aplikacija RedMine



Testiranje podatkovne baze lahko razdelimo na tri glavne korake (Stall, 2005):

- postavimo podatkovno bazo v začetno stanje,
- izvedemo teste,

- preverimo, ali stanje podatkovne baze po testih ustreza našim pričakovanjem.

Pogosta napaka pri testiranju podatkovne baze je priprava zgolj manjše različice celotne produkcijske baze. To je sicer bolj enostavno, vendar se lahko pojavijo težave pri delovanju produkcijske baze. Uporaba realne produkcijske baze je lahko problematična glede hitrosti. Pri testiranju podatkovne baze moramo preveriti tudi shranjene procedure, ki jih testiramo podobno kot preostale metode – shranjeno proceduro izvedemo in primerjamo izhodne vrednosti s pričakovanimi vrednostmi. Določene teste lahko uporabimo na različnih procedurah. Ključno pri učinkovitem testiranju shranjenih procedur je pravilna nastavitev vstopnih parametrov in testnih podatkov.

3.6.3 Kakovost programske kode

Pri razvoju programske opreme stremimo k čim kakovostnejši kodi. To nam sicer zmanjša hitrost razvoja, vendar se ta čas v veliki meri povrne pri popravkih in nadgradnji. Za preverjanje kakovosti kode obstaja več načinov.

Pri statični analizi kode lahko uporabimo možnost, ki nam jo ponuja orodje Visual Studio. Ta analiza programske kode nudi dodatne informacije pri pregledih programske kode znotraj ekipe. Orodje ima različna pravila, ki jih nato upošteva pri analizi kode. Pravila lahko dodajamo in odstranimo.

Za lažjo oceno kompleksnosti programske kode so nam na voljo metrike programske kode. Glavni elementi metrike so število vrstic programske kode, povezanost med razredi in indeks vzdrževanja. Število vrstic nam lahko pove, da je določena metoda preveč obsežna in jo je smiselno razdeliti na več manjših. Povezanost meri odvisnost med razredi – manjša kot je povezanost, lažje kodo znova uporabimo in vzdržujemo.

Goodliffe (2006) opisuje preglede kode kot način odkrivanja skritih napak, povečanje kakovosti kode, širjenje znanja in uveljavljanje kolektivne odgovornosti za programsko kodo.

3.6.4 Preoblikovanje

Kakovost programske kode izboljša tudi preoblikovanje (angl. *refactoring*). Preoblikovanje je čiščenje programske kode z namenom lažjega razumevanja in lažjega spreminjanja. Programska koda mora ohraniti način delovanja, kakršen je bil pred preoblikovanjem.

Preoblikovanje (Fields, Harvie, Fowler, & Beck, 2010) lahko primerjamo tudi z optimizacijo programske kode. Tudi pri optimizaciji največkrat ne spreminjamo lastnosti, temveč način izvajanja. Pri preoblikovanju končni uporabnik sistema ne opazi sprememb, ki so se zgodile v metodah in funkcijah.

Fowler (1999) definira preoblikovanje kot discipliniran način uvajanja sprememb v izvorno kodo z namenom izboljšanja strukture in lažjega vzdrževanja. Glavna lastnost preoblikovanja

je ohranjanje obnašanja izvorne kode. Preoblikovanje ne dodaja in ne odvzema funkcionalnosti, temveč zgolj izboljšuje kakovost.

Brez uporabe preoblikovanja postane programska koda nepregledna. Razvijalci pogosto spreminjajo kodo, ne da bi pomislili na celotno strukturo. Pri slabo strukturirani programski kodi pogosto naletimo na enake oziroma podobne metode na več različnih mestih, kar oteži popravljanje in spreminjanje. Pomemben vidik preoblikovanja je tudi zmanjšanje podvojene kode. Manj kot imamo programske kode, lažji so popravki in nadgradnje. Pri podvojeni programski kodi hitro naletimo na težavo – popravimo zgolj eno metodo, podvojeno metodo pa spregledamo.

Pri razvoju programske opreme pogosto zaradi časovnih omejitev pozabimo na vzdrževanje in odpravljanje napak. Koda je nepregledna in pogosto tudi podvojena. Razvijalci, ki sami niso bili prisotni pri razvoju, zelo težko razumejo programsko kodo brez ustreznih komentarjev.

Preoblikovanje na dolgi rok izboljša kakovost in tudi zmanjša čas, potreben za razvoj, kajti z izboljšanjem kakovosti se zmanjša tudi število napak.

Med bolj težavnimi je preoblikovanje relacijskih podatkovnih baz. Poslovne aplikacije so tesno sklopljene s podatkovno bazo, v kateri so shranjeni podatki, zato so popravki težavni. Podatkovne baze tudi vsebujejo različne vrste podatkov, kar otežuje spreminjanje strukture. Kot primer naj navedem, da samo odstranitev enega stolpca iz podatkovne tabele lahko onemogoči pravilno delovanje celotne aplikacije. Pri večjih popravkih na podatkovni bazi je treba preseliti podatke iz obstoječe podatkovne baze v novo bazo, kar je pri velikem številu podatkov dolgotrajen proces.

Ambler (2006) ugotavlja, da je preoblikovanje podatkovne sheme težje izvedljivo kot preoblikovanje programske kode. Razlog je v centralni umestitvi podatkovne baze, preoblikovanje podatkovne sheme pa pogosto povzroči tudi preoblikovanje povezanih sistemov.

Ambler (2006) definira naslednje vrste preoblikovanja podatkovne sheme:

- preoblikovanje strukture podatkovne sheme (angl. *structural refactoring*),
- preoblikovanje z namenom izboljšanja kakovosti podatkov (angl. *data quality refactoring*),
- preoblikovanje referenčne integritete,
- preoblikovanje arhitekture dostopa do podatkovne sheme,
- preoblikovanje shranjenih procedur.

Za preoblikovanje shranjenih procedur in poizvedb SQL se največkrat odločimo zaradi slabe zmogljivosti. Z leti se poveča količina podatkov in pri razvoju se pogosto ne predvideva tega povečanja. Določene težave z zmogljivostjo lahko rešimo z boljšo in dražjo strojno opremo, vendar to ne reši težav s slabo napisanimi poizvedbami SQL, temveč jih zgolj malo omili.

Preoblikovanje programske kode je zelo široko področje razvoja programske opreme. Lahko samo preimenujemo spremenljivke, lahko spremenimo večji razred v več manjših itd. Preoblikovanje se nanaša na skoraj vse večje projekte, pri katerih prihaja do sprememb skozi čas.

Na SURS-u razvijamo aplikacije s pomočjo orodja Visual Studio. Orodje omogoča preimenovanje, spreminjanje vrstnega reda parametrov, odstranjevanje parametrov, izvleček metod in vmesnikov. Med bolj pogosto uporabljenimi preoblikovanji je preimenovanje, pri čemer zgolj vpišemo novo ime v programsko kodo in s pomočjo pomočnika v orodju dobimo predogled sprememb. Preimenujemo lahko tudi besedilo v komentarjih in nizih znakov.

Podvojena programska koda je dober razlog za preoblikovanje. Andy Hunt in Dave Thomas sta predstavila princip »ne ponavljaj se« (angl. *Do Not Repeat Yourself*) v knjigi Pragmatic Programmer (Hunt & Thomas, 1999), v kateri trdita, da mora imeti vsaka metoda zgolj eno nedvoumno predstavitev v sistemu.

Obstaja veliko razlogov, zaradi katerih se moramo izogibati podvajanju kode. Kadar imamo enako kodo na različnih mestih, se hitro zgodi, da popravimo zgolj eno metodo ali funkcijo, na druge pa pozabimo. Pri podvajanju naletimo na dve različici; koda je lahko enaka, lahko pa je različna, vendar namenjena isti stvari.

4 PRIMERJAVA RAZVOJA LASTNEGA PODATKOVNEGA SLOJA Z RAZVOJEM PODATKOVNEGA SLOJA S POMOČJO OGRODJA ORM

V nadaljevanju bom primerjal razvoj lastnega podatkovnega sloja in razvoj podatkovnega sloja s pomočjo ogrodja ORM. Analiziral bom prednosti in slabosti obeh pristopov.

Vodja razvoja oziroma arhitekt projekta se odloči, katero vrsto podatkovnega sloja bodo uporabili pri razvoju. Po standardu ISO/IEC 42010 je arhitekt oseba, odgovorna za arhitekturo sistema. Arhitekt sodeluje s poslovnim analitikom in projektnim vodjo, ocenjuje in predlaga možnosti za sistem in usklajuje ekipo razvijalcev. Arhitekt sodeluje v vseh fazah razvojnega procesa, vključno z analizo potreb, razvojem, testiranjem in uvajanjem (Esposito & Saltarello, 2014).

Podatkovni sloj je pomemben del vsake poslovne aplikacije, slaba odločitev glede načina dela lahko povzroči veliko nepotrebnega dela in podaljša razvoj aplikacije. Odločitev je treba sprejeti v začetni fazi projekta, v kateri pogosto niso jasne vse podrobnosti. Poslovna logika aplikacije se tudi navezuje na podatkovni sloj, zato so spremembe v poznejši fazi razvoja težavne in drage.

Arhitekt s pomočjo zbranih zahtev izbere najbolj primerno arhitekturo aplikacije. Strukturo aplikacije razdeli na več slojev in manjših podsistemov. Celotna zasnova aplikacije je

pogojena s cilji in zahtevami naročnika. Podatkovni sloj je eden od slojev v strukturi aplikacije. Dobra arhitekturna zasnova omogoča hiter in učinkovit razvoj ter zmanjša število napak.

Izbrana arhitektura naj se zgleduje po splošnih smernicah: zmanjševanje odvisnosti med sloji, vsaka komponenta naj ima točno določeno vlogo itd. Arhitektura je pogojena tudi z varnostjo, razširljivostjo in z dovoljenimi tehnologijami v podjetju. Razvijalci tudi različno poznajo določene tehnologije. Vsi ti vidiki prinašajo dodatne omejitve in zmanjšajo možnosti izbire (Esposito & Saltarello, 2014).

4.1 Kriteriji primerjave med lastnim razvojem in uporabo ogrodja ORM

Pri izboru kriterijev sem si pomagal s standardoma ISO 9126 in ISO 25010.

Pipan (2007, str. 14) je zapisal: »Standard ISO/IEC 9126-1 danes predstavlja okvir za ocenjevanje kakovosti izdelkov programske opreme. Standard ne predpisuje zahtev za programsko opremo, temveč definira model kakovosti, ki je uporaben in primeren za vse vrste programskih rešitev. Vključuje vseh šest glavnih kategorij kakovosti programske opreme, ki so pomembne tudi pri njenem razvoju: funkcionalnost, zanesljivost, uporabnost, učinkovitost, vzdrževalnost in prenosljivost«.

Po raziskavi podjetja Gartner (Brian, 2014) sta glavna kriterija za izbor ogrodja produktivnost in nabor funkcionalnosti. Pomembna kriterija sta še zmogljivost in testiranje programske kode.

Watson (2014) je zapisal, da so glavni kriteriji pri izboru ogrodja zmogljivost, varnost in vzdrževanje.

Standard ISO 25010 opisuje osem glavnih lastnosti (Bass, Clements, & Kazman, 2013):

- funkcionalna ustreznost. Stopnja, do katere sistem ustreza funkcionalnim zahtevam.
- Zmogljivost. Zmogljivost glede na podane kriterije.
- Združljivost. Stopnja, do katere sistem lahko izmenjuje informacije z drugimi sistemi.
- Uporabnost. Stopnja, do katere sistem dosega natančno določene cilje učinkovito in v zadovoljstvo uporabnikov.
- Zanesljivost. Stopnja, do katere sistem opravlja določene funkcije pod določenimi pogoji v določenem časovnem obdobju.
- Varnost. Stopnja, do katere sistem varuje informacije in podatke, da so na voljo zgolj osebam z ustreznimi pravicami.
- Vzdrževalnost. Stopnja uspešnosti in učinkovitosti, s katero lahko spremenimo sistem s pomočjo razvijalcev.
- Prenosljivost. Stopnja uspešnosti, s katero lahko sistem prenesemo iz enega okolja v drugega.

Napake v programski kodi lahko povzročijo veliko škode. Dino Esposito (Esposito & Saltarello, 2014) navaja, da je napaka v programski kodi povzročila padec rakete Ariane 5 leta 1996 samo 40 sekund po vzletu. Do napake naj bi prišlo zaradi težav pri pretvorbi števil.

Po besedah Esposita (Esposito & Saltarello, 2014) je vzdrževanje programske kode ena pomembnejših lastnosti razvoja programske opreme. Vzdrževanje se nanaša na stopnjo, do katere lahko posodabljamo programsko kodo brez težav. Možnost enostavnega vzdrževanja programske kode je posledica različnih dejavnikov, kot so berljiva programska koda, uporaba vzorcev razvoja in dobrih praks.

Enostavnost vzdrževanja programske kode je posledica različnih dejavnikov, berljivost programske kode je eden od njih. Težko berljiva programska koda je nerazumljiva za druge razvijalce in popravki so veliko bolj zapleteni kot pri dobro napisani programski kodi. Komentarji v kodi in dobra dokumentacija nam tudi olajšajo branje programske kode.

S pomočjo standarda ISO, dostopne literature in izkušenj sem za kriterije izbral:

- varnostni vidik,
- vzdrževanje,
- zmogljivost,
- prenosljivost,
- hitrost razvoja,
- poslovni vidik.

4.2 Varnost

Gaberšček (2007) je zapisal: »Naložbe v informacijsko varnost dosegajo že tudi petnajst odstotkov proračunov služb za informatiko varnostno bolj ozaveščenih organizacij«.

Laskowski (2011, str. 10) navaja, da so bili varnostni strokovnjaki v podjetju IBM v letu 2008 priča več kot milijon napadom na podatkovne strežnike. Ugotovili so tudi, da se zelo povečuje število avtomatiziranih napadov na spletne strežnike. Varnostni strokovnjaki so še ugotovili, da se število zlonamernih uporabnikov (hekerjev) povečuje iz leta v leto.

Po podatkih organizacije Open Web Application Security Project (2015) so napadi s pomočjo zlonamerne kode SQL med desetimi najbolj pogostimi napadi na spletu.

Standard ISO 27001 navaja tri glavne vidike varnosti podatkov in informacij (Vasudevan, 2015):

- razpoložljivost. Podatki so dostopni na zahtevo osebam in računalniškim programom.
- Zaupnost. Podatki niso na voljo nepooblaščenim osebam.
- Integriteta. Podatki so točni in jih nepooblaščene osebe ne spreminjajo.

Organizacija mora za zagotavljanje varnosti zadostiti naslednjim varnostnim zahtevam (Jerman Blažič, Klobučar, Nedeljkovič, & Perše, 2001):

- overjanje – možnost preverjanja identitete subjekta,
- zaupnost – informacije ne smejo biti razkrite nepooblaščenim osebam,
- neokrnjenost – ohranjanje pristnosti podatkov,
- nadzor dostopa – preprečevanje nepooblaščene uporabe.

Varnostni vidik je treba upoštevati pri razvoju aplikacij. Pri uporabi shranjenih procedur lahko različnim uporabnikom podrobno določimo pravice za izvajanje. Onemogočimo lahko dostop do podatkovnih tabel, uporabniki lahko dostopajo do podatkov zgolj prek shranjenih procedur. Zlonamerni uporabniki tako težje spreminjajo podatke, kajti uporabljajo lahko zgolj shranjene procedure z že napisanimi poizvedbami. Razvijalci lahko tudi zlorabijo zaupanje in napišejo škodljive poizvedbe SQL, zato moramo pri prevzemu kode preveriti poizvedbe. Lažje jih preverimo pri uporabi shranjenih procedur, kajti ni nam treba iskati kode SQL po aplikaciji (vse shranjene procedure so na enem mestu).

Vieira (2009) navaja, da lahko s pomočjo shranjenih procedur ali pogledov zagotovimo uporabnikom dostop do podatkov brez možnosti dostopa do podatkovnih tabel.

Sistemi z varnostjo med ključnimi zahtevami pogosto uporabljajo shranjene procedure kot edini način za dostopanje in popravljanje podatkov (Vieira, 2009). Neposreden dostop do podatkovnih tabel ni dovoljen. S tem onemogočimo neposredno popravljanje zapisov v podatkovni bazi. Uporabnik ne more izvajati poizvedb SQL, delo s podatki poteka zgolj s pomočjo shranjenih procedur. Zaposleni lahko vseeno povzročijo škodo podjetju s popravljanjem podatkov, vendar lahko izvajajo zgolj poizvedbe SQL, ki jih vsebujejo shranjene procedure. Shranjene procedure nam zmanjšajo tveganje za nepooblaščen dostop do podatkov.

Shranjene procedure lahko tudi varnostno podpišemo in šifriramo. To nam omogoča zaščito naše programske kode pred morebitno zlorabo, preprečimo tudi vpogled v shranjene procedure.

Z uporabo ogrodij ORM lahko uporabljamo shranjene procedure, vendar nam tak način razvoja podatkovnega sloja vzame veliko časa in je veliko bolj zapleten kot zgolj uporaba ogrodja ORM. Ogrodja brez uporabe shranjenih procedur sama gradijo poizvedbe SQL, nad katerimi nimamo nadzora. Preverimo jih lahko s sledenjem na podatkovni bazi, kar je pri velikem številu poizvedb zelo nepregledno.

Pri gradnji lastnega podatkovnega sloja imamo nadzor nad poizvedbami SQL, kajti napišemo jih sami. Pozorni moramo biti na vhodne parametre poizvedb SQL, dobro je uporabiti poizvedbe z dodanimi parametri. Gradnja SQL stavkov s t. i. lepljenjem vhodnih parametrov lahko privede do napada na podatkovni strežnik (angl. *SQL injection*). Takšni napadi so zelo

priljubljeni pri zlonamernih uporabnikih in nam pri slabo zaščiteni podatkovni bazi lahko povzročijo veliko škode.

4.3 Vzdrževanje

Vzdrževanje programske opreme je lahko bolj zahtevno kot sam razvoj aplikacij, zelo pomembni sta dobra dokumentacija in dobro napisana programska koda. Glavni skrbi arhitektov programske opreme sta oblikovanje in implementacija take programske kode, ki zadovolji vse zahteve naročnika in je tudi enostavna za nadgradnjo in vzdrževanje.

Enega večjih stroškov v celotni življenjski dobi aplikacij predstavljata vzdrževanje in spreminjanje aplikacij. Spremembe nastanejo zaradi določenih zakonskih zahtev, zastarelosti aplikacij, počasnega delovanja itd. Spreminjanje aplikacij je lažje pri spletnih aplikacijah, pri katerih ni potrebno nameščanje aplikacij na odjemalce. Pri vzdrževanju bolj obsežnih aplikacij se pokažejo prednosti večslojne arhitekture, kajti lažje odpravljamo napake zgolj na enem mestu, ne da bi posegali v druge sloje aplikacije.

Kot primer vzdrževanja naj navedem staro SURS-ovo programsko kodo, napisano v programskem jeziku ASP. V žargonu se programski kodi ASP reče tudi »špageti koda«. Ta koda je zelo težavna za vzdrževanje in za zahtevane nadgradnje. Počasi selimo programsko kodo na ogrodje Asp.Net. Prenos aplikacij na novo ogrodje je pogosto zelo zahtevno in težko upravičljivo pri vodstvu. Celotno programsko kodo, razen poizvedb SQL, je treba napisati na novo.

Programska koda je lahko težavna za nadgradnjo zaradi pomanjkanja znanja razvijalcev, pogostih sprememb pri naročnikih in zelo kratkih rokov za dokončanje projekta. Za postavitev dobre arhitekture sistema je treba imeti veliko znanja in izkušenj. Vodje projektov pod pritiskom prodajalcev in naročnikov programske opreme pristanejo na zahteve, ki so težko izvedljive, to pa pripelje do nepreglednih rešitev in slabo zasnovane arhitekture sistema. Pisanje tehnične dokumentacije je tudi delo, ki se ga večina razvijalcev izogiba. Pomanjkanje nadzora nad dokumentacijo pripelje do pomanjkljive in slabo napisane dokumentacije, kar še oteži nadgradnjo.

Na SURS-u imamo primere večjih aplikacij brez ustrezne dokumentacije, programska koda tudi ni shranjena v shrambi programske kode. Te aplikacije je pravzaprav nemogoče nadgrajevati, pojavijo se tudi težave pri nadgradnji strežnikov, ker določene storitve niso več podprte.

Na podlagi razvitih dveh podatkovnih slojev, izkušenj in strokovne literature sem prišel do spoznanja, da pri lastno razvitem podatkovnem sloju potrebujemo že za osnovne operacije branja in popravljanja podatkov veliko število poizvedb SQL oziroma shranjenih procedur. Ogrodja ORM sama poskrbijo za poizvedbe SQL, kar nam olajša vzdrževanje. Takšen pristop

ima tudi nekaj slabosti, izgubimo namreč nadzor nad poizvedbami. Pri uporabi shranjenih procedur je ta razlika manjša, kajti v obeh primerih moramo pri spremembi podatkovnega modela popraviti shranjene procedure.

Pri vzdrževanju podatkovnega sloja v ogrodju Ado.Net je več dela pri popravljanju programske kode kot pri uporabi ogrodij ORM. Pri brisanju enega stolpca iz podatkovne tabele je treba popraviti vse poizvedbe, sicer pride do napak pri izvajanju aplikacije. Težave se pogosto pojavijo tudi pri preimenovanju stolpcev. Največja pomanjkljivost pri vzdrževanju poizvedb SQL je v težavnem odkrivanju napak, te se namreč pokažejo šele pri izvajanju. Odkritje napak pri naročniku lahko povzroči veliko slabe volje in škodi ugledu podjetja.

4.4 Zmogljivost

Zmogljivost je ena od ključnih lastnosti dobro napisane programske kode.

Ogrodja ORM so počasnejša kot dober ročno napisan podatkovni sloj, vendar so razlike pri enostavnih poizvedbah majhne.

Ogrodja pri poizvedbah preberejo vse stolpce, kar ni optimalno, kadar ne potrebujemo vseh stolpcev. Relacijske podatkovne baze uporabljajo za izboljšanje hitrosti indekse na podatkovnih tabelah. Ogrodja prebirajo vse stolpce v podatkovni tabeli, kar onemogoči uporabo določenih indeksov. Pri ogrodjih ORM nimamo dostopa do poizvedb SQL, kar nam oteži njihovo optimizacijo. Če želimo izboljšati poizvedbe, jih moramo napisati v nizu.

Ogrodja omogočajo tudi uporabo lastnih poizvedb SQL, kar je smiselno pri bolj zahtevnih poizvedbah. Prednost je v hitrosti, vendar se je treba zavedati, da je poizvedba SQL napisana v nizu znakov, pri katerih lahko hitro pride do kakšne napake v poizvedbi. Težje tudi uporabimo drugo podatkovno bazo, kajti vse poizvedbe moramo preveriti in po potrebi popraviti.

Ogrodja omogočajo nalaganje povezanih entitet. Te lahko naložimo z enim klicem na podatkovno bazo (angl. *eager loading*), lahko pa jih naložimo z več klici po potrebi (angl. *lazy loading*).

Žontar (2009, str. 60) navaja: »Nalaganje na zahtevo je lastnost ogrodja, da naloži samo zahtevan objekt, ne pa tudi objekt, s katerim ima relacije, s tem zagotovimo boljšo izrabo pomnilnika, manj prometa med odjemalcem in podatkovnim strežnikom ter manjšo obremenitev podatkovnega strežnika«.

Pri nalaganju povezanih entitet moramo biti pozorni na hitrost izvajanja poizvedb. Pri večjem številu povezanih entitet se lahko hitro pojavijo težave z zmogljivostjo. Smiselno je s pomočjo orodja SQL Profiler preveriti poizvedbe SQL in čas njihovega izvajanja. Obstajajo tudi druga orodja za preverjanje zmogljivosti, med bolj znanimi sta Entity Framework Profiler in ORM Profiler.

Pri ročno napisanem podatkovnem sloju sami nadzorujemo poizvedbe. Skrbnik podatkovne baze lahko pomaga razvijalcem pri bolj zapletenih poizvedbah, izboljša uporabo indeksov na podatkovni bazi in preveri pravilnost poizvedb. Lastno napisan podatkovni sloj je pri pravilno napisanih poizvedbah hitrejši kot uporaba ogrodij ORM. Pri enostavnejših poizvedbah so razlike zelo majhne in skoraj zanemarljive.

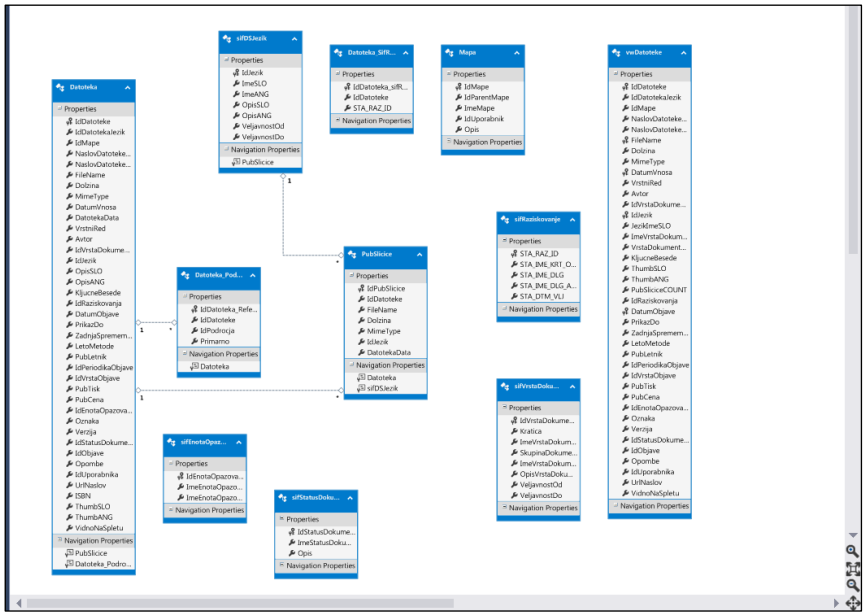
Med bolj izpostavljenimi težavami pri ogrodjih ORM je t. i. problem SELECT N + 1. Ogrodja lahko preberejo vrednosti iz podatkovne baze v povezanih tabelah, šele ko jih potrebujejo (angl. *lazy loading*). Za vsako vrstico v rezultatu poizvedbe se izvede dodatna poizvedba, Za 1.000 vrstic podatkov se izvede 1.001 poizvedba. Poizvedbe se sicer izvedejo zelo hitro, vendar lahko zaradi števila poizvedb zelo upočasnijo sistem. Nepravilna uporaba ogrodij ORM lahko privede do slabe odzivnosti aplikacije.

Pri aplikacijah, pri katerih sta odzivnost in hitrost zelo pomembni, je smiselna uporaba ogrodja Ado.Net. Ogrodja ORM so primerna za uporabo pri večini programskih kod za delo s podatki, kjer pa je hitrost ključnega pomena, je najbolje uporabiti ogrodje Ado.Net.

4.4.1 Primerjava hitrosti dostopanja do podatkov

Za primerjavo hitrosti sem izbral ogrodje Entity Framework različice 6.1.3 in ogrodje Ado.Net različice 4.5. Izbral sem trenutno zadnji stabilni različici obeh ogrodij. Primerjal sem hitrosti ene poizvedbe, hitrosti 100 zaporednih poizvedb ter vstavljanje in popravljanje podatkov. Slika 12 prikazuje podatkovni model aplikacije Dokumentni sistem.

Slika 12: Podatkovni model aplikacije Dokumenti sistem



Pri izvedbi primerjave sem si pomagal s testi, opisanimi v knjigi Programing Entity Framework (Lerman, 2010). Teste sem izvajal na podatkovnem strežniku Microsoft SQL Server 2014. Za podatkovno bazo sem izbral podatkovno bazo, namenjeno aplikaciji

Dokumentni sistem. Podatkovna shema je enostavna, vsebuje deset podatkovnih tabel in pogled (angl. *view*). Izvajal sem jih na računalniku z operacijskim sistemom Windows 7 in s pomočjo orodja Visual Studio.

V prvem primeru sem preveril hitrost poizvedbe, ki vrne zapis v podatkovni bazi s pomočjo primarnega ključa. Najhitrejšo in najpočasnejšo poizvedbo sem odstranil, nato sem izračunal povprečje stotih ponovitev. Poizvedba je enostavna in se navezuje zgolj na eno podatkovno tabelo. Na podatkovnem strežniku se izvede zelo hitro, tako da nima velikega vpliva na celoten čas izvajanja poizvedbe.

Pri ogrodju EF se najpočasneje izvede prva poizvedba, kajti ogrodje potrebuje določene podatke za izvedbo poizvedb. Za prvo poizvedbo ogrodje potrebuje 1344 milisekund, druga poizvedba se izvede v 23 milisekundah. Razlika je očitna, pri večjih podatkovnih modelih se ta razlika še poveča. Povprečni čas stotih poizvedb je 21,9 milisekund. Tabela 1 prikazuje primerjavo hitrosti dostopanja do podatkov.

Tabela 1: Primerjava hitrosti dostopanja do podatkov

Opis	**Ogrodje Ado.Net (v ms)**	**Ogrodje Entity Framework (v ms)**
Poizvedba s pomočjo primarnega ključa	5,3	21,9
Poizvedba, ki vrne 1000 zapisov	28,8	120,6
Posodabljanje enega zapisa	5,3	35
Vstavljanje ene vrstice podatkov	6	10,9
Brisanje 100 zapisov	11	815

Pri ogrodju Ado.Net z uporabo objekta DataReader se prva poizvedba izvede v devetih milisekundah, druga v šestih, povprečje stotih poizvedb je 5,3 milisekunde.

Pri drugi primerjavi sem preveril hitrost poizvedbe, ki vrne 1000 zapisov v podatkovni bazi. Poizvedbo sem zagnal stokrat in znova izračunal povprečje.

Pri ogrodju EF se znova najpočasneje izvede prva poizvedba, ki traja 1452 milisekund, druga poizvedba se izvede v 122 milisekundah. Povprečni čas poizvedbe pri stotih ponovitvah je 120,6 milisekunde.

Ogrodje Ado.Net izvede poizvedbo prvič v 31 milisekundah, druga poizvedba traja 29 milisekund. Povprečni čas poizvedbe pri stotih ponovitvah je 28,8 milisekunde.

Ogrodje Ado.Net je približno štirikrat hitrejše od ogrodja EF, vendar so razlike absolutno gledano manj kot desetinka sekunde. Uporabnik aplikacije ne opazi razlik, razen morda pri prvi izvedeni poizvedbi pri ogrodju EF.

Pri preverjanju hitrosti posodabljanja podatkov sem preverjal, koliko časa traja posodabljanje naslova datoteke. Ogrodje Ado.net izvede posodobitev naslova v povprečju v 5,3 milisekunde

pri stotih ponovitvah. Ogrodje EF posodobitev izvede v povprečju v 35 milisekundah, vendar se tu pred posodabljanjem izvede še branje zapisa.

Hitrost vstavljanja ene vrstice podatkov sem preveril s pomočjo dodajanja novega zapisa v podatkovno tabelo. Povprečni čas vstavljanja enega zapisa pri stotih ponovitvah s pomočjo ogrodja EF je bil 10,9 milisekunde. Ogrodje Ado.Net izvede dodajanje novega zapisa v šestih milisekundah.

Pri primerjavi brisanja zapisov sem se odločil za brisanje stotih zapisov v podatkovni tabeli. Ogrodje Ado.Net je za brisanje potrebovalo 11 milisekund, ogrodje EF pa 815 milisekund.

Ogrodje Ado.Net je pri izvajanju poizvedb hitrejše od ogrodja EF, zelo velika razlika je pri brisanju večjega števila zapisov. Ogrodje EF je počasnejše tudi pri preostalih poizvedbah, vendar je razlika manjša. Pri poizvedovanju je razlika 16 milisekund, kar ne predstavlja ovire pri uporabi ogrodja EF pri razvoju aplikacij.

Ogrodje Ado.Net omogoča največ nadzora nad poizvedbami SQL, z dobrim znanjem jezika SQL napišemo poizvedbe, ki se izvajajo najhitreje. Slaba lastnost takega pristopa je velika količina programske kode, ki jo je treba napisati. Ogrodje Ado.Net je zelo primerno za razvoj aplikacij z zapletenimi poizvedbami in zahtevanim majhnim odzivnim časom. Pri odločitvi za določeno ogrodje moramo premisliti, ali je razlika v hitrosti dovolj tehten razlog, da upravičimo več dela pri razvoju.

4.4.2 Shranjevanje v pomnilnik

S pomočjo shranjevanja rezultatov poizvedb v pomnilnik lahko zelo izboljšamo zmogljivost sistema. V pomnilnik lahko shranjujemo vrednosti za vsakega uporabnika s pomočjo sej. Druga možnost je shranjevanje za vse uporabnike aplikacije, pri čemer pridejo v poštev predvsem razni šifranti. Pri shranjevanju podatkov v pomnilnik je glavni pomislek, ali imamo v njem pravilne in zadnje vrednosti. Pogosto spreminjajoči se podatki niso primerni za shranjevanje v pomnilnik.

Ogrodja ORM omogočajo shranjevanje rezultatov poizvedb v pomnilnik. Pri iskanju s pomočjo primarnega ključa se preveri prisotnost objekta v pomnilniku. Če objekt že obstaja v pomnilniku, ga prebere iz pomnilnika. Tak način nam prihrani dostopanje do podatkovne baze. Podatki, ki se ne spreminjajo veliko, so najbolj primerni za shranjevanje v pomnilnik.

Z uporabo pomnilnika izboljšamo zmogljivosti tudi v ogrodju Ado.Net. V pomnilnik lahko shranjujemo različne objekte .Net, najbolj primerna sta objekta DataSet in DataTable. S podatki v pomnilniku zmanjšamo število poizvedb na podatkovni bazi.

S pomočjo razreda SqlDependency zagotovimo veljavne podatke v pomnilniku. Razred SqlDependency opozori aplikacijo na spremembe v podatkovni bazi. Uporaba razreda SqlDependency je na voljo zgolj v povezavi s podatkovnim strežnikom Microsoft SQL Server od različice 2005 naprej. V pomnilniku je smiselno shranjevati podatke, ki se ne spreminjajo

pogosto in se veliko uporabljajo. Najbolj primerna uporaba je v spletnih aplikacijah, in sicer za podatke, ki se ne spreminjajo pogosto.

4.4.3 Sočasno dostopanje do podatkov

Pri pogosto spreminjajočih se podatkih je treba zagotoviti pravilen način shranjevanja podatkov. Težave s sočasnim dostopanjem do istih podatkov se pojavijo v primeru popravljanja podatkov na vnosni maski, ki jih je v vmesnem času že popravil nek drug uporabnik. To težavo lahko rešujemo na več načinov. V zelo redkih primerih uporabimo metodo zaklepanja podatkov, veliko bolj pogoste so metode, pri katerih preverjamo spremembe podatkov.

Uporabniki lahko vnašajo in spreminjajo podatke in jih ne shranijo takoj. V vmesnem času drug uporabnik spremeni iste podatke in tako nastanejo težave. Težave lahko nastanejo tudi pri večjem številu zaporednih vnosnih mask, pri katerih je zahtevano shranjevanje v enem koraku.

Na SURS-u uporabljamo pri pogosto spreminjajočih se podatkih t. i. optimistični način. Morebitne spremembe lahko ugotovimo s pomočjo časovnega žiga (angl. *timestamp*), ki se pri vsaki spremembi vrednosti podatkov v vrstici posodobi. Preden se podatki v vrstici posodobijo, je treba preveriti vrednosti časovnega žiga. Če se te ne ujemajo, se posodobitev podatkov ne izvede.

Ogrodje Ado.Net uporablja optimistični način za reševanje težav pri sočasnem spreminjanju istih podatkov. Če je podatke, medtem ko smo jih popravljali, spremenil že kdo drug, sta na voljo dva pristopa.

Prvi pristop je uporaba polja v podatkovni tabeli s časovnim žigom. Pri popravljanju podatkov preverimo vrednost časovnega žiga, in če se vrednosti ujemata, shranimo nove vrednosti. Druga možnost je preverjanje vrednosti vseh stolpcev. Objekt DataSet ima shranjene originalne vrednosti za določen zapis in nove spremenjene vrednosti. Pred shranjevanjem novih vrednosti se opravi preverjanje, ali se je v vmesnem času zgodila kakšna sprememba.

4.4.4 Vstavljanje večjih količin podatkov

Vstavljanje večje količine podatkov predstavlja težavo ogrodjem ORM. Ogrodje vstavlja vrstico po vrstico s poizvedbo na podatkovni bazi. Tudi pri brisanju večjega števila vrstic se ogrodja ne izkažejo najbolje. Brisati je treba vrstico za vrstico, kar je časovno zelo zamudno. S pomočjo poizvedbe SQL se lahko izvede brisanje večjega števila vrstic na enostaven in učinkovit način. Vstavljanje večjih tekstovnih datotek (angl. *comma-separated values*) s pomočjo ogrodij je zelo zamudno in ni priporočljivo.

Na SURS-u uporabljamo za večje datoteke orodja, ki jih ponuja podatkovna baza in so se izkazala za bistveno hitrejša od uporabe programskih rešitev. Pri bolj zahtevnih prenosih

podatkov uporabljamo orodje Microsoft Integration Services. Uporabimo lahko tudi ukaz SQLBulkInsert v ogrodju Ado.Net, ki je optimiziran za vstavljanje velikega števila zapisov.

Razlika med nalaganjem podatkov s pomočjo ogrodij ORM ali s pomočjo orodij ETL je lahko zelo velika. Orodja ETL so prilagojena vstavljanju večjih količin podatkov in so učinkovitejša. SURS je uporabljal rešitev vstavljanja večjih količin podatkov s pomočjo orodja ETL, zunanji razvijalci pa so razvili novo rešitev z uporabo spletnih storitev. Ta rešitev je bila veliko počasnejša – pomembna je izbira pravilnega orodja za določeno nalogo.

4.4.5 Poizvedovanje

Pri ogrodju EF lahko poizvedujemo po podatkih na več načinov, najpogostejši način je poizvedovanje s pomočjo poizvedb LINQ. Ko s pomočjo poizvedb LINQ ne dosežemo zastavljenih ciljev, si pomagamo s poizvedbami Entity SQL. Tretja možnost je uporaba poizvedb SQL za določeno podatkovno bazo. Težava pri uporabi poizvedb SQL za točno določeno podatkovno bazo je v težjem prenosu aplikacije na drugo podatkovno bazo, kar je ena od glavnih prednosti ogrodij ORM.

Ogrodje EF samo zgradi poizvedbe SQL in nam tako olajša delo. Uporaba poizvedb SQL je smiselna pri zahtevnejših poizvedbah, pri katerih želimo uporabiti funkcionalnosti podatkovne baze, ki jih na drugačen način ne moremo.

Poizvedovanje pri lastno razvitem podatkovnem sloju zahteva od razvijalcev veliko dela s pisanjem poizvedb SQL oziroma shranjenih procedur. Poizvedbe so napisane v nizu znakov in hitro lahko pride do napak, te pa se pokažejo šele pri izvajanju programske kode. Prednost poizvedovanja s pomočjo ogrodja Ado.Net je v nadzoru nad poizvedbami SQL.

4.4.6 Večslojne aplikacije

Ogrodje ORM nam pri večslojnih aplikacijah nudi več načinov prenašanja vrednosti med različnimi sloji aplikacije. Pri enostavnejših aplikacijah, namenjenih zgolj branju podatkov iz podatkovne baze, prenašamo podatke s pomočjo enostavnih objektov POCO. Pri shranjevanju in popravljanju podatkov je prenos med sloji bolj težaven, podatkovni sloj namreč nima informacij o tem, kateri podatki so se spremenili.

Za prenos entitet pri spletnih aplikacijah je najpogosteje smiselno uporabljati ogrodje Asp.Net Web API.

Klasično ogrodje Ado.Net omogoča razvoj večslojnih aplikacij s pomočjo spletnih storitev Xml. Podatkovni sloj aplikacije in poizvedbe SQL so znotraj spletnih storitev. Predstavitveni sloj pa pridobiva podatke s pomočjo spletnih storitev.

Med predstavitvenim in podatkovnim slojem se uporabljajo objekti tipa DataSet za prenos med slojema. Poslovna logika je znotraj predstavitvenega sloja.

Prednosti uporabe spletnih storitev Xml so (Designing a .NET Application, 2015):

- razvoj aplikacij je lahko zelo hiter. Objekt DataSet omogoča povezovanje podatkov na gradnike znotraj mask. Na ta način hitro zgradimo masko za pregled in vnašanje podatkov.
- Uporabniki lahko uporabljajo aplikacije s pomočjo povezave na internet na različnih lokacijah.
- Predstavitveni sloj je ločen od podatkovnega sloja in ne vsebuje poizvedb SQL.
- Podatkovni sloj se posodablja na enem mestu, ni potrebno nameščanje novih komponent na odjemalcih.

Slabosti uporabe spletnih storitev Xml so (Designing a .NET Application, 2015):

- Vsa imena stolpcev so »zapečena« v programski kodi. Pri spremembah stolpcev v podatkovni bazi so potrebni posegi v programsko kodo v aplikaciji. Napaka se odkrije šele pri izvajanju programske kode.
- Spletne storitve so počasnejše kot neposreden dostop do podatkovne baze.

Za prenos med sloji se lahko uporabijo objekti DataSet oziroma DataTable, vendar ti objekti porabijo veliko računalniškega spomina in niso najbolj primerni za uporabo pri zelo obiskanih spletnih straneh.

4.5 Prenosljivost podatkovnega sloja med različnimi podatkovnimi bazami

Ena od glavnih prednosti razvoja podatkovnega sloja s pomočjo ogrodja ORM je prenosljivost med različnimi bazami podatkov. Na enostaven način lahko začnemo z uporabo druge podatkovne baze. Tako ogrodje Entity Framework kot NHibernate podpirata uporabo različnih baz podatkov. Podprte so vse bolj uporabljane odprtokodne in plačljive relacijske podatkovne baze.

Podatkovni sloj, zgrajen s pomočjo ogrodja Ado.Net, je veliko težje prilagajati različnim bazam podatkov. Pri prenosu na drugo bazo je treba preveriti vse poizvedbe SQL in jih popraviti. Za različne baze so na voljo tudi različni vmesniki za delo z njimi. Pri razvoju podatkovnega sloja, namenjenega različnim bazam, lahko uporabimo splošne vmesnike za delo s podatki, vendar imajo ti okrnjene zmožnosti. Specializirani vmesniki so zmogljivejši in prilagojeni različnim funkcionalnostim točno določene podatkovne baze.

Esposito (Esposito & Saltarello, 2014) pravi, da uporaba različnih podatkovnih baz pri aplikacijah ni tako pogosta, kot se nam zdi na prvi pogled. Obstajajo seveda aplikacije z različnimi podatkovnimi bazami, vendar je po njegovem mnenju bolj pogosta uporaba zgolj ene vrste podatkovne baze.

Korotkevitch (2014) navaja, da je prenosljivost med različnimi podatkovnimi bazami uresničljiva zgolj pri enostavnejših aplikacijah. Podporo za različne podatkovne baze lahko

nudimo zgolj z uporabo funkcij, ki so podprte v različnih podatkovnih bazah. Odreči se moramo naprednim funkcijam in optimizaciji.

Pri uporabi ogrodij ORM se poslovna logika nahaja v programski kodi. Pri večslojni arhitekturi je to v poslovnem sloju. Poslovno logiko aplikacije lahko uporabimo na različnih podatkovnih bazah brez popravkov. Zaradi varnostnih vidikov nekatere aplikacije ne dovoljujejo uporabe poizvedb SQL, omejeni smo zgolj na uporabo shranjenih procedur. Prenos na drugo podatkovno bazo je v tem primeru veliko težji. Shranjene procedure je treba prilagoditi podatkovni bazi.

4.6 Hitrost razvoja

Pri razvoju podatkovnega sloja nam uporaba ogrodij ORM prihrani veliko časa in zmanjša količino programske kode, namenjene delu s podatki. Razvijalci se lahko posvetimo delu s poslovnimi zahtevami aplikacije, ogrodje pa poskrbi za osnovne operacije za delo s podatkovno bazo. Pri spremembah na podatkovni shemi niso potrebni popravki poizvedb SQL, kajti poizvedbe niso shranjene v programski kodi.

Pri ogrodju Ado.Net je treba za vsako poizvedbo napisati kodo SQL oziroma uporabiti t. i. čarovnike za delo s podatki. Vsaka sprememba na podatkovni shemi zahteva popravke poizvedb SQL – to povzroči dodatno delo v primerjavi z ogrodji ORM.

Po mojem prepričanju je pri razvoju novih aplikacij za dostop do podatkovne baze smiselna uporaba ogrodja ORM, izjeme so določene zapletene poizvedbe in vstavljanje velike količine podatkov. Pri zelo zapletenih poizvedbah ter pri aplikacijah z veliko transakcijami je bolj priporočljiva uporaba shranjenih procedur, ki so primernejše tudi za delo z zaupnimi podatki. Odločitev je pogosto pogojena tudi z osebnimi preferencami razvijalca.

Pred odločitvijo o vrsti podatkovnega sloja se je dobro posvetovati s skrbnikom podatkovnih baz. Razvijalci na SURS-u imamo drugačen pogled kot skrbniki podatkovnih baz, kajti hitrost razvoja nam je pomembnejša od optimalnih poizvedb SQL.

4.7 Poslovni kriteriji

Izboljšana produktivnost nam prinese manjše stroške razvoja in poveča dobiček. Ogrodja ORM v primerjavi z lastno razvitim slojem zmanjšajo število tipkarskih napak pri pisanju poizvedb. Aplikacije z manjšim številom napak in kakovostna programska koda prinesejo podjetju ugled in pravočasno končane projekte. Kakovostno in pravočasno končani projekti so ključ do uspeha podjetij, ki se ukvarjajo z razvojem programske opreme.

Ogrodja ORM zmanjšajo količino programske kode, namenjene delu s podatki. Manj kot je programske kode, lažje in cenejše je vzdrževanje.

Na SURS-u težje denarno merimo prihranke, ki nam jih je prinesla uporaba ogrodja EF. Izpostavil bi predvsem manjše število napak in enostavnejše spreminjanje entitetno-relacijskega modela v podatkovni bazi. Pri razvoju aplikacije Dokumentni sistem je bilo približno pol manj kode v primerjavi z lastno razvitim slojem.

Gradišnik (2005, str. 64) navaja, da »pretekle izkušnje, obstoječa vlaganja, predlagana nova tehnologija in ostali faktorji igrajo pomembno vlogo pri končni odločitvi«. Določene zaposlene je težje prepričati v nove tehnologije, predvsem zaradi strahu in navad. Odpor razvijalcev do uporabe nove tehnologije za delo s podatki lahko predstavlja tveganje za podjetje.

Ogrodje EF je trenutno prva izbira podjetja Microsoft za razvoj podatkovnega sloja. Ogrodje se razvija in dopolnjuje. Ogrodje Ado.Net je sicer zrelo in bo v uporabi še dolgo časa, vendar je v podjetju Microsoft v ozadju.

Ogrodja ORM pripomorejo k zniževanju stroškov razvoja programske opreme. To pa je eden od ključnih dejavnikov, zaradi katerega se vodje projektov odločajo za uporabo teh ogrodij. Porter (1998) je zapisal, da naj bi podjetje ustvarilo večjo vrednost in konkurenčno prednost z doseganjem nižjih stroškov, kot jih imajo konkurenti.

Strmec (2009) pravi, da je cikel razvoja programske opreme čedalje krajši. Naročniki pričakujejo vedno bolj kakovostno programsko rešitev z bogatim naborom funkcionalnosti in možnostjo hitrega prilagajanja rešitve.

Hitrejši razvoj programske opreme omogoča tudi lažje prilagajanje spremembam na trgu. Naročniki informacijskih rešitev si želijo hitro implementirati novo rešitev s čim manj stroški. Ogrodja ORM omogočajo razvoj programske opreme z manj napakami, kar omogoča hitrejšo in uspešno uporabo rešitve, kajti ne potrebujemo več toliko časa za testiranje. Možnost hitrega prilagajanja informacijskih rešitev in odzivanje na spremembe pri poslovnih zahtevah je lahko strateškega pomena za razvoj in rast podjetja.

Za uspešno uvedene programske rešitve se štejejo tiste, pri katerih smo dosegli polno načrtovano funkcionalnost rešitve ob časovnih in stroškovnih prekoračitvah, ki ne presegajo 20 odstotkov načrtovanih vrednosti (Gradišar, Jaklič, & Turk, 2007).

Zmanjševanje stroškov, namenjenih za informacijsko tehnologijo, lahko podjetja pripelje do izbire cenejših ponudnikov podatkovnih baz. Zamenjava podatkovne baze pri lastno razvitem podatkovnem sloju prinaša veliko dela in posledično stroškov, ogrodja ORM pa nam olajšajo prehod in zmanjšajo njegove stroške.

4.8 SWOT-analiza ogrodja ORM v primerjavi z lastno razvitim podatkovnim slojem

Analiza SWOT predstavlja prednosti (angl. *Strengths*), slabosti (angl. *Weaknesses*), priložnosti (angl. *Opportunities*) in nevarnosti (angl. *Threats*). Spodnja analiza SWOT nam je v pomoč pri odločitvi za vrsto podatkovnega sloja.

Ta analiza se uporablja za identifikacijo različnih vidikov, ki nam pomagajo pri izbiri vrste podatkovnega sloja. Ena od prednosti analize SWOT je enostaven in berljiv prikaz prednosti in slabosti. Enostaven prikaz je lahko tudi zavajajoč, kajti na prvi pogled se zdi, da imajo vsi podatki isto težo. Določena vrsta podatkovnega sloja ima lahko veliko prednosti in zgolj eno slabost, ki je lahko ključna (Green, 2012).

Analizo SWOT sem naredil na podlagi dveh različnih razvitih podatkovnih slojev in njune evalvacije. Pomagal sem si tudi s strokovno literaturo in izkušnjami, pridobljenimi z delom na SURS-u.

4.8.1 Prednosti ogrodja ORM

- **produktivnost.** Količina programske kode za dostopanje do podatkov se zmanjša z uporabo ogrodij ORM v primerjavi z lastno razvitim podatkovnim slojem. Ogrodja poskrbijo za preslikavo poizvedb v poizvedovalni jezik SQL. Razvijalci se tako izognemo ponavljajočemu pisanju programske kode in se lahko posvetimo poslovnemu delu aplikacije.
- **Hitrejši razvoj programske opreme.** Ogrodja ORM omogočajo hitrejši razvoj programske opreme. Manjša količina programske kode, namenjene delu s podatki, poveča hitrost razvoja programske opreme v primerjavi z lastno razvitim slojem.
- **Hitrejše odkrivanje napak.** Prevajalnik programske kode sporoči napake pri prevajanju in tako jih lahko hitreje odpravimo. Napake pri lastno razvitem podatkovnem sloju se pokažejo šele pri izvajanju aplikacije.
- **Enostavnejše vzdrževanje programske kode.** Manj kot je programske kode, lažje jo vzdržujemo. Spremembe na podatkovnem modelu se s pomočjo ogrodja ORM preslikajo v poizvedbe, zato ni treba popravljati vseh poizvedb, povezanih s spremembami na podatkovnih tabelah.
- **Povezljivost na različne vrste podatkovnih baz.** Ogrodja ORM omogočajo razvoj podatkovnega sloja, namenjenega različnim vrstam podatkovnih baz. Pri lastno razvitem podatkovnem sloju je treba pri menjavi podatkovnega strežnika preveriti in popraviti poizvedbe SQL.

4.8.2 Slabosti ogrodja ORM

- **počasnejše izvajanje poizvedb.** Med najbolj izpostavljenimi slabostmi ogrodij ORM je počasnejše izvajanje poizvedb v primerjavi s poizvedbami SQL. Pisanje poizvedb s pomočjo jezika SQL oziroma shranjenih procedur nam omogoča optimizacijo poizvedb in uporabo naprednih funkcij podatkovne baze.
- **Slabša izraba zmožnosti podatkovne baze.** Zaradi prenosljivosti med različnimi podatkovnimi bazami lahko uporabimo zgolj osnovne funkcije, ki so na voljo v vseh podatkovnih strežnikih, ki jih ogrodje podpira. Odreči se moramo naprednim funkcijam, ki jih določena podatkovna baza omogoča.
- **Težja optimizacija poizvedb.** Pri ogrodjih ORM nimamo neposrednega nadzora nad poizvedbami SQL, zato jih težje izboljšamo. Ne moremo posredovati težavnih poizvedb strokovnjakom za podatkovne baze.
- **Slaba izkoriščenost indeksov na podatkovnih tabelah.** Ogrodja ORM preberejo vse stolpce v podatkovni bazi, z uporabo lastnih poizvedb SQL lažje uporabimo indekse.
- **Vstavljanje večje količine podatkov** (angl. *bulk insert*). Pri vstavljanju večje količine podatkov se ogrodja ORM ne izkažejo najbolje. Ogrodja vstavljajo vrstico po vrstico, kar je zelo zamudno.
- **Uporaba ogrodja ORM.** Ogrodja ORM niso primerna za aplikacije, pri katerih poizvedbe SQL niso dovoljene.

4.8.3 Priložnosti

- **odprtokodno ogrodje.** Manjkajoče funkcionalnosti razvijajo tudi zunanji razvijalci, ne samo zaposleni v podjetju Microsoft. Ogrodje se tako hitreje razvija. Prednost pred klasičnimi odprtokodnimi rešitvami je v tem, da Microsoft preveri dodano programsko kodo, preden jo dobimo v uporabo.
- **Ogrodje EF 7 bo delovalo tudi na drugih platformah.** V letu 2016 naj bi podjetje Microsoft izdalo sedmo različico ogrodja EF, ki bo delovalo tudi v drugih okoljih, kot sta Linux in iOS, ne le v okolju Windows.

4.8.4 Nevarnosti

- **nova tehnologija**. Razvijalci potrebujemo nekaj časa, da se privadimo na nov način dela. Nekateri razvijalci s težavo spremenijo način dela.
- **Uporaba ogrodja ORM, kjer to ni smiselno**. Ogrodja ORM ni smiselno uporabljati za vse vrste aplikacij. Pri aplikacijah s poudarkom na zmogljivosti je bolje uporabljati lastno razvit podatkovni sloj.
- **Slabše poznavanje jezika SQL**. Razvijalci se lahko preveč zanesejo na poizvedovanje s pomočjo ogrodja in zanemarijo učenje poizvedovalnega jezika SQL.

- **Normalizirana podatkovna baza**. Za učinkovito uporabo ogrodja ORM potrebujemo normalizirano podatkovno bazo. Če podatkovna baza ni zgrajena po pravilih, nam to oteži uporabo ogrodja ORM.

SKLEP

Razvoj programske opreme je zahtevno delo, ki se pogosto konča neuspešno. Veliko število informacijskih projektov se ne konča v predvidenem roku, naročniki so pogosto razočarani nad izdelkom. Pri razvoju novih aplikacij poskušamo čim bolj zmanjšati morebitna tveganja in upoštevati primere dobrih praks. Tehnologija in način dela se zelo hitro spreminjata, razvijalci težko sledimo vsem spremembam in se odločimo za najboljši način dela.

Arhitekt programskih rešitev ima ključno vlogo pri izboru metode razvoja podatkovnega sloja. Odločitev je zelo pomembna, kajti posledice napačne odločitve se pokažejo pozneje, ko je za spremembe lahko že prepozno. Pogosta napaka je odločanje na podlagi lastnih preferenc oziroma odločanje za način dela, ki je arhitektu poznan.

V magistrskem delu sem se osredotočil na razvoj podatkovnega sloja s pomočjo tehnologije Microsoft .Net. Primerjal sem razvoj lastnega podatkovnega sloja z razvojem s pomočjo ogrodja ORM.

Po izkušnjah avtorjev knjige Entity Framework 4 in Action (Mostarda et al., 2011) porabimo za pisanje programske kode za dostop do podatkov do 35 odstotkov časa. Uporaba ogrodij ORM lahko v nekaterih primerih zmanjša ta čas na pet odstotkov, v večini primerov pa se ti odstotki gibljejo med 15 in 20 odstotkov.

Moj osnovni avtorski prispevek je v analizi razvoja podatkovnega sloja. V raziskavi sem predstavil prednosti in slabosti obeh načinov razvoja. Razvil sem prototip aplikacije za shranjevanje vsebine dokumentov in njihovih metapodatkov. Podatkovni sloj sem razvil dvakrat, prvič s pomočjo ogrodja ORM, drugič pa na klasičen način s pomočjo ogrodja Ado.Net.

Ogrodja ORM predstavljajo po mojem mnenju velik preskok pri razvoju podatkovnega sloja, vendar niso primerna za vse programske rešitve.

V magistrskem delu sem iskal odgovore na naslednji vprašanji:

1. Ali uporaba ogrodij ORM prispeva k bolj učinkovitemu in kakovostnejšemu razvoju programske opreme?

S pomočjo dostopne literature in prototipa aplikacije sem preveril prednosti in slabosti uporabe ogrodja ORM. Uporaba tega ogrodja izboljša produktivnost, za delo s podatki je potrebne veliko manj programske kode. Prevajalnik nam sporoči večino težav oziroma napak

že pri prevajanju programske kode, kar je zelo velika prednost v primerjavi z lastno razvitim podatkovnim slojem.

2. Kako uporaba ogrodij ORM omogoča hitrejšo menjavo podatkovnega strežnika?

Ogrodja ORM omogočajo lažjo menjavo podatkovnega strežnika. Poizvedbe SQL niso shranjene v programski kodi, ogrodja jih generirajo sproti. Menjava je enostavnejša in lažja, vendar izgubimo možnost uporabe naprednih funkcionalnosti, ki jih omogoča podatkovna baza. Omejiti se moramo na funkcije, ki jih omogočajo vse podprte podatkovne baze. Pri določenih podatkovnih bazah je nujen tudi nakup vmesnika za delo z ogrodjem ORM, kar povzroči dodatne stroške in morebitne težave.

Izbor metodologije pri razvoju podatkovnega sloja predstavlja strateško odločitev. V magistrskem delu sem predstavil pomembnejše kriterije, ki jih je treba upoštevati pri odločitvi. Opisal sem prednosti in slabosti obeh pristopov. Moj avtorski prispevek je v analizi dveh različnih pristopov pri razvoju podatkovnega sloja. Raziskal sem ogrodja ORM v okolju .Net.

Cilj magistrskega dela je dosežen. V njem sem predstavil najpomembnejše kriterije za izbor vrste podatkovnega sloja. Odločitev ni preprosta, treba je upoštevati znanje razvijalcev, čas za učenje nove tehnologije in pripravljenost zaposlenih na spremembe.

Na SURS-u imamo manjše število razvijalcev, zato sta nam zelo pomembna enostavno vzdrževanje in hiter razvoj. Odločili smo se za uporabo ogrodja EF, ki ga hitro usvojimo, na voljo je veliko knjig in druge literature. Z manjšim številom zaposlenih lahko dosežemo enake cilje, preostale zaposlene lahko prerazporedimo na druge naloge. Ogrodje je brezplačno, stroške predstavljajo izobraževanje in licence za orodje Microsoft Visual Studio.

V magistrskem delu sem prišel do spoznanja, da je za gradnjo podatkovnega sloja na Statističnem uradu RS bolj primerna izbira ogrodja ORM. Za večino dela s podatkovno bazo je najbolje uporabiti ogrodje ORM, ogrodje Ado.Net uporabimo zgolj pri določenih izjemah, kot so zapletene poizvedbe in vstavljanje velikega števila podatkov. Ogrodje ORM omogoča hiter in enostaven razvoj aplikacij, kar je zelo pomembno pri zmanjševanju časa in stroškov razvoja aplikacij.

LITERATURA IN VIRI

1. Ambler, S. (2006). *Refactoring Databases: Evolutionary Database Design.* Boston: Addison Wesley.
2. Bass, L., Clements, P., & Kazman, R. (2013). *Software Architecture in Practice*. New Jersey: Pearson.
3. Bauer, C., King, G., & Gregory, G. (2016). *Java Persistence with Hibernate*. Greenwich: Manning.
4. Berardi, N., Katawazi, A., & Bellinaso, M. (2009). *ASP.NET MVC 1.0 Website Programming*. Hoboken: John Wiley & Sons.
5. Brian, D. (2014, 13. maj). Choosing a JavaScript Framework. *Gartner*. Najdeno 15. maja 2016 na spletnem naslovu https://www.gartner.com/doc/2738717/choosing-javascript-framework
6. Celko, J. (2015). *SQL for Smarties* (2nd ed.). Burlington: Morgan Kaufmann Publishers.
7. Cure, A., & Schenker, G. (2011). *NHibernate 3 Beginner's Guide.* Birmingham: Packt publishing.
8. *Designing a .NET Application*. Najdeno 15. junija 2015 na spletnem naslovu http://msdn.microsoft.com/en-us/library/ms973829.aspx
9. Dewson, R. (2012). *Beginning SQL Server 2012 for Developers.* New York: Apress.
10. Downing, L. (2006). Implementing EDMS: Putting people first. *Information management Journal*, *40*(4), 44–50.
11. *Entity Framework.* Najdeno 15. Junija 2015 na spletnem naslovu https://msdn.microsoft.com/en-us/data/ef.aspx
12. Esposito, D. (2011). *Programming Microsoft ASP.NET 4*. Seattle: Microsoft Press.
13. Esposito, D., & Saltarello, A. (2014). *Microsoft .NET: Architecting Applications for the Enterprise*. Seattle: Microsoft Press.
14. Fields, J., Harvie, S., Fowler, M., & Beck, K. (2010). *Refactoring: Ruby Edition*. New Jersey: Pearson.
15. Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Boston: Addison Wesley.
16. Freeman, A. (2013). *Pro ASP.NET MVC 5*. New York: Apress.
17. Gaberščček, S. (2007). *Ekonomika naložb v informacijsko varnost* (magistrsko delo). Ljubljana: Ekonomska fakulteta.
18. Gačnik, M. (2005). *Arhitektura sodobnih rešitev Microsoft.Net*. Ljubljana: Microsoft Corporation.
19. Goodliffe, P. (2006). *Code craft*. San Francisco: No Starch Press.
20. Grad, J., & Jaklič, J. (1996). *Baze podatkov*. Ljubljana: Ekonomska fakulteta.
21. Gradišar, M., Jaklič, J., & Turk, T. (2007). *Osnove poslovne informatike*. Ljubljana: Ekonomska fakulteta.

22. Gradišnik, D. (2005). *Primerjava temeljnih pristopov pri razvoju informacijskih rešitev elektronskega poslovanja* (magistrsko delo). Ljubljana: Ekonomska fakulteta.
23. Green, P. (2012). *An Analysis, Design, And Implementation of a Business Intelligence Solution for Powerplay Mobile*. Najdeno 5. januarja 2016 na spletnem naslovu http://www.csb.uncw.edu/mscsis/complete/pdf/PatrickGreenCapstone.pdf
24. Harrison, L. (2011). *TortoiseSVN 1.7 Beginner's Guide*. Birmingham: Packt publishing.
25. Henderson K., & Celko J. (2000). *The Guru's Guide to Transact-SQL*. Boston: Addison Wesley.
26. Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer*. Boston: Addison Wesley.
27. Jerman Blažič, B., Klobučar, T., Nedeljkovič, D. & Perše, Z. (2001). *Elektronsko poslovanje na internetu*. Ljubljana: Gospodarski vestnik.
28. Korotkevitch, D. (2014). *Pro Sql Server Internals*. New York: Apress.
29. Kuaté, P., & Harris, T. (2009). *NHibernate in Action*. Greenwich: Manning.
30. Laskowski, J. (2011). *Agile IT Security Implementation Methodology*. Birmingham: Packt publishing.
31. Lavrič, R. (2004). *Analiza replikacije na primeru spletnega portala NLB* (magistrsko delo). Ljubljana: Ekonomska fakulteta.
32. LeBlanc, P. (2013). *Microsoft SQL Server 2012 Step by Step*. Seattle: Microsoft Press.
33. Leonard, A. (2013). *Pro Hibernate and MongoDB*. New York: Apress.
34. Lerman, J. (2010). *Programming Entity Framework, (*2nd ed.*)*. Sebastopol: O'Reilly Media.
35. *LinqPad*. Najdeno 5. marca 2014 na spletnem naslovu https://www.linqpad.net/
36. Marguerie, F., Eichert, S., & Wooley, J. (2008). *LINQ in Action*. Greenwich: Manning.
37. Medvešek, S. (2013). *Analiza uvedbe dokumentnega sistema v elektroenergetskem podjetju* (magistrsko delo). Ljubljana: Ekonomska fakulteta.
38. Mohorič, T. (1991). *Podatkovne baze*. Ljubljana: Fakulteta za elektrotehniko in računalništvo.
39. Mostarda, S., Sanctis, M., & Bochicchio, D. (2011). *Entity Framework 4 in Action*. Greenwich: Manning.
40. Novak, I., Velvart, A., Granicz, A., Balássy, G., Hajdrik, A., Sellers, M., Hillar, G., Molnár, A.& Kanjilal, J. (2010). *Visual Studio 2010 and .NET 4*. Hoboken: John Wiley & Sons.
41. Osherove, R. (2014). *The Art of Unit Testing*. Greenwich: Manning.
42. *Open Web Application Security Project* . Najdeno 21. junija 2015 na spletnem naslovu https://www.owasp.org/
43. Pipan, M. (2007). *Metode in tehnike ocenjevanja uporabnosti programskih rešitev* (magistrsko delo). Ljubljana: Ekonomska fakulteta.
44. Porter, M. E. (1998). *Competitive advantage: Creating and sustaining superior perforamance*. London: Free press.

45. Rodrigues, F., Coles, M., & Dye, D. (2012). *Pro SQL Server 2012 Integration Services*. New York: Apress.
46. Rudolf, M., & Zorman, M. (2004). EDMS kot podpora obvladovanja tveganj na področju pogodbenega urejanja medsebojnih razmerij. *Zbornik posvetovanja Dnevi slovenske informatike* (str. 409–414). Portorož: Slovensko društvo informatika.
47. Russo, M., & Pialorsi, P. (2010). *Programming Microsoft LINQ in .NET Framework 4*. Seattle: Microsoft Press.
48. Schenker, G., & Cure, A. (2011). *NHibernate 3*. Birmingham: Packt publishing.
49. Stall, T. (2005, 6. april). Unit Testing the Data Access Layer. Najdeno 10. aprila 2015 na spletnem naslovu http://www.4guysfromrolla.com/articles/040605-1.aspx
50. Stare, M. (2013). Uvajanje orodij ORM, uporaba pri prenovi spletnega portala. *Zbornik posvetovanja Dnevi slovenske informatike* (str. 123–126). Portorož: Slovensko društvo Informatika.
51. Statistični urad Republike Slovenije. (2015). *Vizija Statističnega urada Republike Slovenije* (interno gradivo). Ljubljana: Statistični urad Republike Slovenije.
52. Strmec, J. (2009). *Prenova metode razvoja programskih rešitev v podjetju Hermes Softlab* (magistrsko delo). Ljubljana: Ekonomska fakulteta.
53. Vasudevan, V. (2015). *Application Security in the ISO 27001:2013 Environment*. London: IT Governance.
54. Vieira, R. (2009). *Professional Microsoft SQL Server 2008 Programming*. Indianapolis: Wiley Publishing.
55. Yener, M., & Theedom, A. (2015). *Professional Java EE Design Patterns*. Indianapolis: Wiley.
56. Watson, B. (2014). *Writing High-Performance .NET Code*. Indianapolis: Sams.
57. Umek Luzar, K. (2005). *Dokumentacijski sistem v farmacevtskem podjetju* (magistrsko delo). Ljubljana: Ekonomska fakulteta.
58. Žontar, R. (2009). *Zagotavljanje trajnosti objektov v ogrodju .Net* (diplomsko delo). Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko.