UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Marko Podgoršek

N-NIVOJSKA ARHITEKTURA V OGRODJU .NET VERZIJE 3.5

Diplomska naloga
na univerzitetnem študiju

Mentor: doc. dr. Marko Bajec

Ljubljana, 2008

Zahvala

Zahvalil bi se:

- svojemu mentorju doc. dr. Marku Bajcu za njegovo prijazno pomoč pri izdelavi te diplomske naloge;
- podjetju Adacta d. o. o. in svojemu nadrejenemu Draganu Radoloviču, ki je verjel moji viziji in mi omogočil podrobno pregledati Microsoft Visual Studio 2008. Seveda sem mu hvaležen še za privolitev k prenosu aplikacije, ki jo trenutno razvijamo, iz Visual Studia 2003 v Visual Studio 2008. Sedaj je programiranje veliko lažje;
- vsem ostalim sodelavcem, ki so mi kadarkoli pomagali, še posebno Nejcu Faganelu, Mateju Ahačiču, Tadeju Jančarju in Šandorju Feldiju, ker so to junaki, ki sedijo z mano v pisarni in s katerimi delim večino delovnega časa;
- Anžetu Grosu, ki je bil tako prijazen, da je prebral prvo verzijo diplome. Preživel je. Postal je tudi lektor te diplome.

Kazalo

Seznam uporabljenih kratic in simbolov

AJAX	Asynchronous JavaScript and XML
CRUD	Create, Read, Update and Delete
GPL	General Public License
HTML	HyperText Markup Language
LINQ	Language Integrated Query
MVC	Model-View-Controller
SQL	Structured Query Language
URL	Uniform Resource Locator
WCF	Windows Communication Foundation
WPF	Windows Presentation Foundation
WWF	Windows Workflow Foundation

Povzetek

Dokument vsebuje strnjen opis arhitektur računalniških sistemov, razloge za prehode iz ene arhitekture v drugo, probleme, na katere naletimo pri prehodu, in kako te rešujemo. Opis N-nivojske arhitekture je bolj podroben in vsebuje opise treh nivojev – podatkovni, poslovni in predstavitveni nivo. Namen dokumenta je predstaviti implementacijo N-nivojske arhitekture z uporabo ogrodja .Net verzije 3.5 v razvojnem okolju Microsoft Visual Studio 2008. Pozornost pri implementaciji je osredotočena na pravilno uporabo novih tehnologij, na izogibanje pisanja odvečne izvorne kode, na uporabo generatorjev kode, kjer je to možno, in na „opisno“ programiranje z namenom, da je končna izvorna koda čista, pregledna in enostavna za popravljanje.

Ključne besede:

arhitektura, visual studio, crud, linq, windows communication foundation

Abstract

In this document we discuss several different types of system architecture, including a concise description of several types of acrhictecture, reasons for migrating systems onto a new architecture, common problems we might encounter and possible solutions. We describe in detail the N-tier architecture including the data, business and presentation tier. The goal of this document is to present an implementation of an N-tier architecture using .NET Framework 3.5 and Visual Studio 2008. We focus on the correct use of new technologies, avoiding unnecessary source code, using code generators where possible and on „descriptive“ programming to ensure a concise, clear and readable source code.

Key words:

architecture, visual studio, crud, linq, windows communication foundation

1. Uvod

Skozi leta razvoja računalniških sistemov se je s krajšimi koraki ali daljšimi skoki spreminjala tudi njihova arhitektura. Sprva namizne aplikacije, ki so se v celoti nahajale na enem računalniku, so se zaradi potreb po sočasni uporabi podatkov s strani več uporabnikov spremenile v obliko „odjemalec/strežnik“. Tudi ta revolucionarni model je doživel svojo preobrazbo, saj so se skozi njegovo dolgoletno uporabo v različnih računalniških sistemih njegove pomanjkljivosti dodobra zbistrile, kar je botrovalo nastanku novega modela računalniške arhitekture – 3-nivojske arhitekture. Ta je prinesel ogromno novosti na področju strukture, načrtovanja, razvoja in posredno tudi implementacije računalniških sistemov s tem, da je funkcionalno sorodne dele sistemov združil v logične enote in jih med seboj fizično ločil. Zadnji korak v razvoju arhitekture računalniških sistemov predstavlja N-nivojska arhitektura, ki je nekakšna samoumevna nadgradnja 3-nivojske in je s povečanjem števila nivojev prinesla več „specializacije“ oz. boljšo strukturiranost sistemov.

V procesu evolucije se je na področju računalniških sistemov zgodilo precej sprememb, veliko stvari pa kljub temu ostaja bolj ali manj nespremenjenih, kar gre v večji meri pripisati človeškemu dejavniku. Ljudje imamo po naravi odpor do sprememb, kar je razlog, da so se nekateri principi iz prejšnjih generacij računalniških sistemov ohranili tudi v novejših generacijah. Več o tem v poglavju 2.4.

Nefleksibilnost pri odločanju o spremembah in novostih ter pri vpeljavi le-teh se slej ko prej odrazi tudi v praktičnih problemih. Eden takih je na primer implementacija poslovne logike v nek računalniški sistem. Na vprašanje o tem, kam naj bi bila v N-nivojsko zasnovanem sistemu implementirana poslovna logika, bo vsak razvijalec odgovoril, da na poslovni nivo. Podoben odgovor bi dobili tudi od razvijalcev, ki bi jih vprašali o tem, kje imajo poslovno logiko implementirano v lastnem sistemu. Pravzaprav je tudi laikom popolnoma samoumevno, da se poslovna logika nahaja na nivoju z enakim imenom – poslovni nivo, kljub temu pa se v večini računalniških sistemov lahko prepričamo o tem, kako zelo ohlapno si razvijalci predstavljajo združevanje funkcionalnosti po nivojih. Iz večine primerov bo popolnoma jasno razvidno, da je logika, ki bi jo moral vsebovati poslovni nivo, praviloma raztresena po različnih nivojih. Potrebno se je zavedati, da filozofija, ki se skriva za N-nivojsko arhitekturo, predvideva kar se da neodvisno implementacijo posameznih nivojev. Poslovni nivo tako ni namenjen le določenemu delu „skupne“ poslovne logike, temveč celotni poslovni logiki, ki naj bi jo nek računalniški sistem uporabljal.

V zadnjih letih sem sodeloval in si nabiral izkušnje pri izgradnji oz. optimizaciji nekaterih N-nivojskih računalniških sistemov. Prebiranje literature, neposreden kontakt z ljudmi, ki izdelujejo takšne sisteme, in v končni fazi praktični problemi, s katerimi sem se srečeval, so me pripeljali do ugotovitve, da v večini primerov ni moč govoriti o kaki konkretni abstrakciji med posameznimi nivoji. Veliko poslovne logike se je – kljub temu da ji je namenjen lastni nivo – ponavadi nahajalo na podatkovnem nivoju, praviloma pa je je bilo kar precej moč zaslediti tudi na predstavitvenem nivoju.

Zelo poredko sem zasledil tudi točno obrazložitev izrazov plast in nivo.

Nivo je fizična reprezentacija, definirana s fizičnim strežnikom ali s skupino strežnikov, ki opravljajo isto funkcijo.
Plast je del sistema, ki je vsebovana v enem procesu ali eni programski komponenti. V enem nivoju je lahko vsebovanih več plasti, lahko pa se vsako plast prestavi tudi na svoj nivo, če se pri tem uporabi enega izmed možnih načinov komunikacije preko mrežnih protokolov.
N-nivojsko arhitekturo, njene prednosti in problematiko napačnega oz. pomanjkljivega pristopa k njej sem opisal v drugem poglavju postopoma in po sklopih. Tretje poglavje izpostavlja pomen kvalitetne implementacije poslovne logike in dejstvo, da se je med implementacijo poslovne plasti treba ukvarjati izključno s poslovno logiko, prenos podatkov, transakcije med posameznimi nivoji, pisanje serializacijske kode in ostalo pa se prepusti razvojnemu okolju, čarovnikom in generatorjem kode. S takšnim pristopom pohitrimo in olajšamo skupinski razvoj sistema kakor tudi njegovo vzdrževanje. Poleg tega se skozi N-nivojsko arhitekturo enostavneje in bolje približamo razmišljanju naročnika, katerega želja je le, da bi sistem dobro deloval, notranji mehanizmi, logika delovanja in struktura pa ga ne zanimajo.

2. Opis arhitekture

Poglavji 2.1 in 2.2. se dotikata problemov, ki se pogosto pojavljajo v računalniških sistemih današnjega časa. Evolucijo problema razlaga poglavje 2.1, stanje današnjih sistemov pa poglavje 2.2. Problemi so večinoma povezani z razdrobljenostjo poslovne logike po različnih nivojih N-nivojskih sistemov, čemur navadno botruje človeški odpor do sprememb. Poglavji 2.3. in 2.4. opisujeta prehod na t. i. srednji oz. poslovni nivo, razloge zanj, probleme, na katere lahko naletimo pri prehodu, ter rešitve le-teh. V poglavju 2.5. je podrobneje opisana N-nivojska arhitektura.

2.1. Evolucija problema

Skozi razvoj računalniških sistemov se je razporeditev poslovne logike po nivojih ves čas spreminjala. Načini njenega prerazporejanja in razlogi zanje so lepo razvidni iz opisov sistemov različnih generacij, ki sledijo. Prav ti razlogi so botrovali izoblikovanju za razvoj računalniških sistemov pomembnega vprašanja, kje naj se nahaja poslovna logika [1].

2.1.1. Namizna aplikacija

Vsa logika, ki jo aplikacija uporablja, je s poslovno logiko vred implementirana na enem samem nivoju. Lokacija poslovne logike pri tipični namizni aplikaciji nima ključnega pomena, saj plasti v takšni aplikaciji niso razmejene oz. so med seboj samo pomešane.

2.1.2. Odjemalec/strežnik

Arhitektura odjemalec/strežnik predvideva dva nivoja, na vsakem od njiju pa vsaj eno plast. Prvi sistemi, grajeni po tej arhitekturi, so bili zamišljeni zelo enostavno, saj so na strežniški strani poganjali samo podatkovno bazo. Z vidika nivojev se je takšne sisteme ločevalo na aplikativni (odjemalec) in podatkovni del (strežnik). Poslovna logika se je navadno v celoti nahajala na odjemalcu, kjer je bila pomešana z ostalo logiko (npr. uporabniškim vmesnikom).
Takšen pristop je nekaj časa krojil obliko računalniških sistemov, vendar so se kaj kmalu pokazale njegove očitne pomanjkljivosti.
Ob vsakršni spremembi poslovne logike je bilo z novostmi potrebno posodobiti vse odjemalce. Ročno posodabljanje v večjih okoljih ni prišlo v poštev, zato se je posodabljanje odjemalcev praviloma reševalo z omrežnimi posodobitvami oz. posodobitvami na daljavo, ki pa so ob tedanjih kapacitetah lahko pomenile dodatno obremenitev ali celo preobremenitev omrežja. Optimizacija omrežne komunikacije je bila neizbežna. Kaj hitro je bilo jasno, da bi povečan promet, ki ga je povzročalo nenehno posodabljanje odjemalcev, lahko dodobra zajezili tako, da bi večino poslovne logike z odjemalca premaknili na strežnik.

Strežnik je bil v sistemu vrste odjemalec/strežnik arhitekturno sicer dobra izbira za implementacijo poslovne logike, v praksi pa je takšna implementacija za seboj potegnila nove probleme zaradi specifičnih potreb podatkovne baze, ki je ravno tako tekla na njem. Poleg tega so se podatkovne baze izkazale neprimerne za implementacijo in vršenje poslovne logike, saj so bile v prvi vrsti namenjene shranjevanju in pridobivanju podatkov, njihova razširljivost pa je bila ravno tako osredotočena na doseganje teh dveh ciljev. Dodatne omejitve je predstavljal jezik za poizvedbe v bazi oz. njegov dodatek za pisanje baznih procedur. Ta je bil namenjen osnovnim podatkovnim transformacijam in ni omogočal kompleksnejših rešitev. Njegova funkcionalnost je v osnovi služila čim hitrejšemu izvajanju poizvedb v bazi, ne pa tudi implementaciji poslovne logike, kar pomeni, da je bilo treba vso logiko, preneseno iz odjemalca na strežnik, na novo osmisliti, prilagoditi oz. preoblikovati tako, da je ustrezala omejitvam baznih procedur. Takšna vmesna rešitev je bila med drugim tudi posledica nefleksibilnosti človeškega faktorja oz. njegovega zavrtega prilagajanja spremembam, saj se je popolnoma nov problem skušalo rešiti z obstoječimi in že v začetku ne preveč primernimi mehanizmi.
Vsem odklonilnim argumentom navkljub se je poslovna logika z odjemalca preselila v bazne procedure, izbira strežnika za nosilca večine poslovne logike pa se je kljub njegovi nepopolnosti izkazala za pomemben, morda celo bistven korak proti večplastnosti današnjih sistemov.
V arhitekturi odjemalec/strežnik se je poslovna logika razdrobila na dva dela – vsebovala sta jo oba nivoja. Strežnik je je vseboval okrog 65 %, približno 35 % pa je je ostalo odjemalcu.

2.1.3. Tronivojska arhitektura

Ko so se pomanjkljivosti modela odjemalec/strežnik dodobra razjasnile, se je pričel vzpon tronivjoske arhitekture.
Delitev poslovne logike na dva nivoja je bila že v osnovi zastavljena dokaj ohlapno. Dvonivojska arhitektura je počasi prerasla svoje okvire. Njene pomanjkljivosti so postajale vse bolj očitne, z njimi povezani problemi pa so bili znotraj obstoječih okvirov vse težje rešljivi.
Največji problem sistemov, grajenih po modelu odjemalec/strežnik, je bil na strani strežnika. Današnje podatkovne baze lahko brez problemov upravljajo z več tisoč sočasnimi povezavami, v devetdesetih letih preteklega stoletja, ko je model odjemalec/strežnik dosegel svoj vrhunec, pa so imele podatkovne baze velike probleme že pri obladovanju nekaj sto sočasnih povezav. Dodaten problem je predstavljalo licenciranje podatkovnih baz. Večina ponudnikov je višino računa pogojevala s številom sočasnih povezav na bazo, kar je tako za manjša kot večja podjetja pomenilo precejšen denarni zalogaj. Oba opisana problema sta izoblikovala potrebo po zmanjševanju števila povezav na podatkovno bazo.
Neposredna povezava na podatkovno bazo potrebuje visoko podatkovno prepustnost in nizko latenco, te pa z odjemalci, ki so v omrežje povezani preko podatkovno nizkoprepustnih povezav, ni moč zagotavljati.
Utrdila se je tehnologija upravljanja s povezavami na podatkovno bazo imenovana razdeljevanje povezav (connection pooling). Njena glavna značilnost je to, da lahko obstoječe, trenutno nerabljene povezave, ki so bile ustvarjene zaradi zahtevkov za delo s podatkovno bazo, poda v uporabo neki drugi zahtevi, za katero še ne obstaja vzpostavljena povezava.
Za implementacijo razdeljevanja povezav med veliko različnimi odjemalci, je bil potreben dodaten nivo. Srednji nivo, vrinjen med odjemalce in strežnik, je vsaj sprva v večini sistemov opravljal izključno funkcije, povezane z razdeljevanjem povezav.

Istočasno pa se je ponekod zgodil tudi premik naprej. Programski jeziki, s katerimi je bil implementiran srednji nivo (C++, Visual Basic, Delphi, Java), so bili veliko bolj priročni in fleksibilni kot togi jeziki za izdelavo baznih procedur, kar je snovalce nekaterih sistemov pripeljalo do tega, da so na novi vmesni nivo poleg razdeljevanja povezav počasi začeli vpeljevati tudi poslovno logiko. Kmalu je postalo jasno, da je bil to korak v pravo smer in da je srednji nivo odlična izbira za implementacijo poslovne logike.

2.2. Današnji sistemi

V današnjih sistemih skorajda ne najdemo več samostojne poslovne namizne aplikacije, arhitektura odjemalec/strežnik pa je še prisotna. Stara tronivojska arhitektura je dozorela in se prerodila v N-nivojsko. Uporaba N-nivojske arhitekture je korak v pravo smer, a jo je treba uporabiti na pravi način. Več o tem je napisano v poglavju 2.2.2.

2.2.1. Odjemalec/strežnik

V povprečju se poslovna logika v sistemski arhitekturi odjemalec/strežnik deli po ključu, opisanem v poglavju 2.1.2 (35 % - odjemalec : 65 % - strežnik), v praksi pa se njena količina na eni in drugi strani razlikuje od aplikacije do aplikacije. Večina poslovne logike je implementirane v obliki baznih procedur in pogledov na tabele. Na odjemalcu je praviloma implementiran tisti del poslovne logike, ki mu bazne procedure in jezik SQL ne zagotavljajo zadostne fleksibilnosti ali pa se na odjemalcu izvaja bistveno hitreje kot na strežniku.

2.2.2. N-nivojska arhitektura

Pri gradnji N-nivojskih sistemov iz različnih razlogov prihaja do stanja, ki je z vidika združevanja poslovne logike pogosto slabše od tistega v sistemih, grajenih na osnovi starejših arhitektur. Namesto združevanja se v mnogih primerih zgodi ravno nasprotno – pride do dodatne drobitve poslovne logike kljub novemu nivoju, katerega namen je ravno združevanje celotne poslovne logike na enem mestu. Več o tem je napisano v poglavju 2.4.
Tudi pri N-nivojski arhitekturi se sistemi med seboj razlikujejo po tem, kako je njihova poslovna logika razporejena po nivojih, vseeno pa imajo skupno točko – poslovna logika je sedaj namesto po dveh razdrobljena po treh nivojih. V nadaljevanju so opisani trije najpogostejši primeri njene razporeditve.

2.2.2.1. Primer 1

Eno pogostejših razporeditev poslovne logike v N-nivojskem sistemu prikazuje tabela 1.

Predstavitveni nivo	Poslovni nivo	Podatkovni nivo
35 %	0 %	65 %

Tabela 1. Razporeditev poslovne logike po nivojih za primer 1

Poslovni nivo v tem primeru ne vsebuje nobene poslovne logike, ampak služi le kot pretvorni nivo med prenosno obliko podatkov (npr. XML) in obliko podatkov, ki jo razume podatkovna baza. Takšna pretvorba se imenuje tudi serializacija (serialization). Sistem, ki uporablja takšno delitev, omogoča možnost razdeljevanja povezav, neodvisnost in delno ločitev odjemalcev od podatkovne baze, vendar pa ne vsebuje pravega poslovnega nivoja, saj poslovna logika ni združena na srednjem nivoju, temveč se enako kot pri dvonivojski arhitekturi v celoti nahaja na obeh ostalih nivojih.

2.2.2.2. Primer 2

Pogosta je tudi razporeditev poslovne logike, razvidna iz tabele 2.

Predstavitveni nivo	Poslovni nivo	Podatkovni nivo
15 %	25 %	60 %

Tabela 2. Razporeditev poslovne logike po nivojih za primer 2

V tem primeru se nekaj poslovne logike že nahaja na srednjem nivoju. Tipično se na poslovni nivo premakne (večji) del poslovne logike, ki se je do sedaj nahajala na predstavitvenem nivoju, večina logike, implementirane na podatkovnem nivoju, pa največkrat ostane kar na svojem mestu.
V takšnem primeru je dosežena 25% izolacija poslovne logike, kar ni v skladu s cilji implementacije poslovnega nivoja, hkrati pa so možnosti za napake v tem primeru veliko večje kot v prvem. Razlogi za napake so lahko različni, ker je npr. delež poslovne logike implementiran na srednjem nivoju, razvijalec predstavitvenega nivoja lahko zaradi površnosti pozabi preveriti vsa poslovna pravila oz. jih nekaj celo načrtno ignorira.
V primeru pravilne implementacije do takšnih napak ne prihaja, saj se vsa poslovna pravila preverjajo na poslovnem nivoju.

2.2.2.3. Primer 3

V N-nivojski arhitekturi je predvideno, da naj bi poslovni nivo zajemal celotno poslovno logiko. To prikazuje prva vrstica s podatki v tabeli 3.

Predstavitveni nivo	Poslovni nivo	Podatkovni nivo
0 %	100 %	0 %
3 %	100 %	0.5 %

Tabela 3. Razporeditev poslovne logike po nivojih za primer 3

N-nivojski sistem, ki je pravilno zgrajen (po specifikaciji), ima sledeče prednosti:

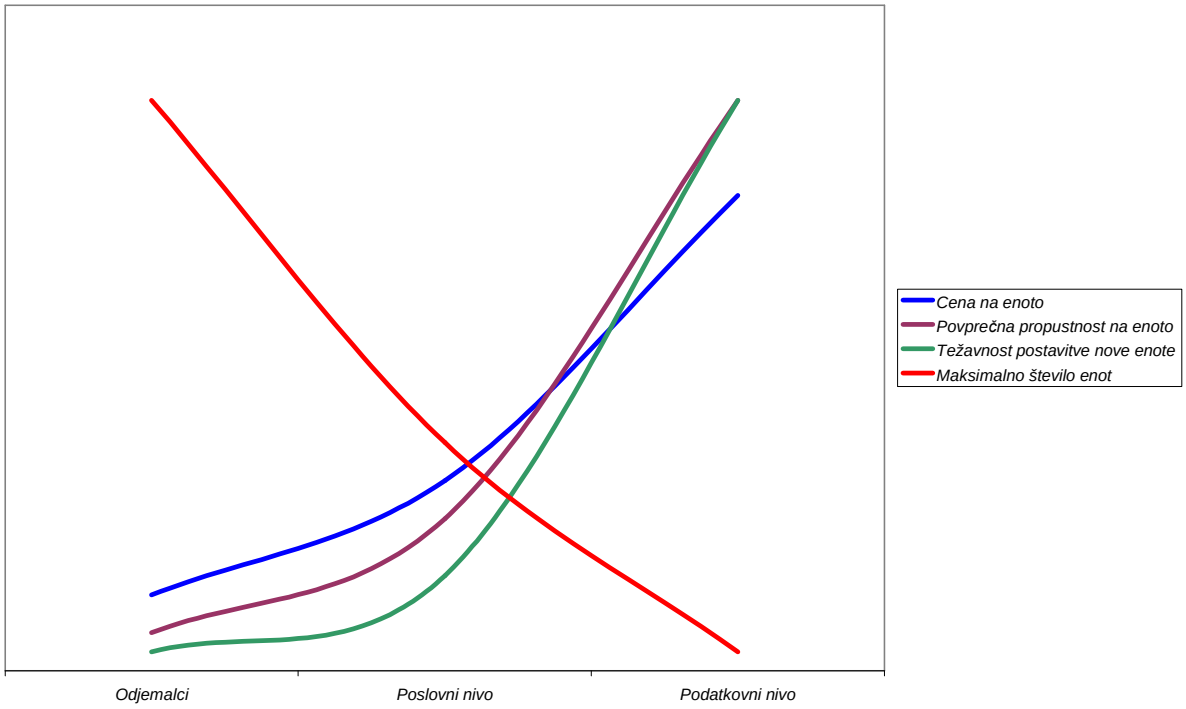
- celotna poslovna logika se nahaja na enem mestu, zato jo je moč enostavno preverjati, razhroščevati, spreminjati,
- za implementacijo poslovnih pravil se lahko uporabi pravi jezik, ki je bolj fleksibilen in bolj priročen kot SQL in bazne procedure,
- baza služi izključno za shranjevanje podatkov in s tem pridobi na učinkovitosti; branje in pisanje podatkov se kot edini pomembnejši nalogi podatkovnega strežnika lahko izvajata optimalno.

Takšen primer predstavlja ideal, h kateremu je pri implementaciji poslovnega nivoja potrebno stremeti, vendar pa brez določenih kompromisov ne gre. Podvajanju določenih poslovnih pravil na različnih nivojih se je tudi v N-nivojskem sistemu težko izogniti.
Druga vrstica v tabeli 3 prikazuje dejansko stanje kvalitetno vpeljanega N-nivojskega sistema. Razvidno je, da se tako na predstavitvenem kot na podatkovnem nivoju določena funkcionalnost, povezana s poslovno logiko, podvaja.
Na predstavitvenem nivoju gre za mehanizem validacije vnosnih podatkov, ki se hkrati nahaja tudi na poslovnem nivoju. Čeprav se validacija izvaja na predstavitvenem nivoju, kjer se podatki tudi vnašajo, jo poslovni nivo naknadno izvede še enkrat. V primeru, da se validacija na predstavitvenem nivoju zaradi takšnih ali drugačnih razlogov ne izvede, zanjo tako poskrbi poslovni nivo.
Pri podatkovnem nivoju do podvojitev poslovne logike navadno prihaja zaradi potreb paketnega ažuriranja, ki ga je zaradi hitrosti izvajanja in manjše obremenitve omrežja smiselno izvesti na podatkovnem strežniku oz. na sami bazi. Kljub prenosu funkcionalnosti

paketnega ažuriranja na podatkovni nivo je pametno, da poslovna plast že pred ažuriranjem podatkovne baze preveri, ali se bo le-to sploh lahko pravilno izvedlo.

2.3. Selitev na srednji nivo

Selitev na srednji nivo je polna pasti, v katere se je v primeru slabega načrtovanja selitve kaj lahko ujeti. Močno vkoreninjena človeška potreba po ohranjanju starih navad lahko kaj hitro botruje temu, da se ob odsotnosti jasnih smernic kak kos poslovne logike podzavestno implementira v bazne procedure. Temu se je potrebno za vsako ceno izogniti, saj v nasprotnem primeru zastavljeni cilj ne bo dosežen. Poslovna logika bo najverjetneje le na malce drugačen način in v drugačnih deležih razdrobljena po razpoložljivih nivojih.
Pri selitvi aplikacije iz starejše na N-nivojsko arhitekturo je zato pomembno čim strožje upoštevati priporočila.
Bazne procedure naj bi bile uporabljene le na podatkovnih bazah, ki so optimizirane za njihovo hitro izvajanje. Danes je takšnih baz v uporabi še precej, vendar se vse njihove novejše inačice – tako kot nekateri od njim konkurenčnih produktov – vse bolj osredotočajo tudi na optimizacijo ad-hoc stavkov SQL oz. stavkov, ki niso ne v obliki procedur ne kako drugače shranjeni v podatkovni bazi.
Če se pri delu s podatki uporabljajo bazne procedure, je potrebno njihovo funkcionalnost strogo omejiti. Tak, na videz konzervativen, pristop predvideva uporabo baznih procedur izključno pri združevanju tabel za potrebe pridobivanja podatkov, pri ažuriranju podatkov v bazi pa je priporočeno, da v eni proceduri ne ažuriramo več kot ene tabele.
Selitev na srednji nivo je le v redkih primerih možna brez dodatne strojne opreme. Nakup le-te v samem začetku sicer poviša stroške vpeljave sistema, v končni fazi pa se N-nivojska arhitektura ob pravilni integraciji lahko pokaže za cenejšo od svojih predhodnic. Večanje zmogljivosti podatkovnega strežnika je cenovno namreč bistveno manj ugodna rešitev kot optimizacija sistema s širitvijo srednjega nivoja. Strojna oprema, ki jo za svoje delovanje potrebuje podatkovni strežnik, spada v višji cenovni razred, srednji nivo pa popolnoma optimalno deluje že na opremi reda namiznih računalnikov. Tudi širitev srednjega nivoja na več fizičnih strežnikov je enostavnejša, hitrejša in zato občutno cenejša, kot pa če bi se iz enega na več strežnikov odločili širiti podatkovno bazo. Pri srednjem nivoju je na nov strežnik, preden ga vključimo v obstoječo infrastrukturo, potrebno naložiti le ustrezno programsko opremo, preusmerjanje odjemalcev na različne strežnike pa prepustiti usmerjevalniku ali kakemu drugemu kosu strojne ali programske opreme. Širitev baze na več fizičnih strežnikov zahteva natančno načrtovanje in uskladitev podatkovnih baz na posameznih strežnikih. Izbira slednjih zaradi narave podatkovnega strežnika in različnih potreb pri različnih podatkovnih bazah ni preprosta. V primeru večje napake ali površnosti v načrtovanju se lahko sekundarni stroški, povezani z reševanjem napačne izbire strojne opreme ali napačne integracije novega podatkovnega strežnika, neobvladljivo povzpnejo.
Razliko v ceni med omenjenima razširitvama povzroča še en, že omenjen, dejavnik – način licenciranja podatkovnih baz. Te se navadno licencirajo po številu procesorjev, na katerih takšna baza deluje. Zastonjske različice podatkovnih baz so že dolgo na voljo, vendar se jim večina klasičnih podjetij iz dostikrat slabo utemeljenih in površno argumentiranih razlogov še vedno izogiba.
Selitev poslovne logike na srednji nivo lahko občutno zmanjša obremenjenost podatkovnega strežnika, hkrati pa odpravi potrebo po njegovi nadgradnji in s tem povezane visoke stroške.
Možnost za razširitev posameznih nivojev je prikazana na sliki 1, kjer so izpostavljeni najbolj pomembni dejavniki, ki jih je treba upoštevati pri odločitvi, kateri nivo bomo razširili (podatki niso v merilu).



Slika 1. Velikosti dejavnikov glede na posamezne nivoje

Pri širitvi odjemalca največji problem predstavljajo nenehne posodobitve, povezane s spremembami v poslovni logiki. Odjemalcev je praviloma veliko, zato je njihovo nenehno posodabljanje z večine pogledov potratna praksa. Pri podatkovnem nivoju se za negativne dejavnike širitve izkažejo cena, propustnost in težavnost postavitve novih enot, kar jasno kaže na to, da je v večini primerov razširitev infrastrukture podatkovnega nivoja velik nesmisel. Širitev sistema je lahko dovolj enostavna, poceni in brez večjih posebnosti le, v kolikor se širitve lotimo na srednjem nivoju.

2.4. Ovire in rešitve selitve

Vse ovire, na katere naletimo pri selitvi poslovne logike na srednji nivo, niso tako očitne, kot je očitna sprememba programskega jezika iz jezika za pisanje baznih procedur v jezik tretje generacije (npr. C#).

Ne zadostuje, da se s pomembnostjo selitve na N-nivojsko arhitekturo, novostmi, ki jih ta prinaša, in ovirami, na katere je pri njej moč naleteti, seznanimo le sami. V smiselnost selitve je potrebno prepričati večino oz. vse ljudi, ki bodo sistem implementirali in ga kasneje vzdrževali (razvijalci, administratorji ...). V fazi argumentacije selitve je potrebno pričakovati začetni odpor, pogojen s človeško nagnjenostjo k ohranjanju starih navad, zaradi katerega prepričevanje širše skupine ljudi v smiselnost selitve ni enostavna naloga.

Idealen pristop k razširitvi poslovnega nivoja predvideva vpletenost sistemskega arhitekta, ki naj bi opravljal redne preglede izvorne kode in sproti odkrival ter označeval vso poslovno logiko, ki je implementirana nepravilno ali na nepravem mestu. Napake, odkrite v zgodnjem stadiju razvoja, so odpravljene mnogo hitreje in ceneje, kot če bi jih npr. odpravljali v končni, produkcijski fazi.

Če v skupini snovalcev in razvijalcev sistema ni sistemskega arhitekta, potem je najbolje, da se razvijalci kar sami preverjajo oz. si na ta način razdelijo naloge sistemskega arhitekta.

Podjetja, ki imajo večino poslovne logike implementirane v baznih procedurah, v dosti primerih na enak način rešujejo tudi problematiko varnosti. Večina razvijalcev rešitev po tem

kopitu je prepričanih, da je podatkovna baza vsemogočna in da bi varnostni mehanizem, vgrajen kamorkoli drugam kot v bazo samo, lahko pomenil resno varnostno grožnjo. Implementacija srednjega nivoja ter s tem prehod na N-nivojsko arhitekturo je v takšnih sistemih skorajda nemogoča.

V sodobnih sistemih, izdelanih s tehnologijami, kakršni sta .NET in Java, je varnost na ravni srednjega nivoja vse bolj poudarjena. .NET tehnologija se že nekaj časa osredotoča na to, da bi v svoje programske jezike (C#, Visual Basic .NET, J#) čim bolje vpeljala kar največ varnostnih mehanizmov. Argumenti za takšno odločitev so podobni tistim, ki v istih jezikih predvidevajo tudi implementacijo poslovne logike (poglavje 2.1.3).

Pred selitvijo iz starejše na N-nivojsko arhitekturo je nanjo potrebno pripraviti različne profile ljudi oz. zaposlenih v podjetju.

Med pomembnejše skupine, ki jih je potrebno izobraziti, sodijo ljudje, ki delajo s podatkovno bazo. Ti so poslovno logiko s pomočjo baznih procedur najprej implementirali na stran podatkovnega strežnika, nato pa jo tam vzdrževali toliko časa, da ločnice med poslovno logiko in hranjenjem podatkov največkrat sploh ne vidijo več, kaj šele da bi jo poznali iz prakse – po dolgih letih združevanja obeh dejavnosti jim je njun skupni obstoj postal samoumeven in oboje vidijo izključno kot nerazdružljivo celoto. Bojazni skrbnikov baz ob vpeljavi N-nivojske arhitekture so nemalokrat povezane tudi s strahom pred izpadom njihovega sistema, ki se ob vpeljavi vseh večjih sprememb lahko zgodi. Poleg dodatne motivacije pri prehodu na N-nivojsko arhitekturo in znanja, potrebnega za miselni preskok, jim je potrebno obrazložiti tudi, da se obseg njihovega dela na podatkovnem nivoju oz. v podatkovni bazi ne bo bistveno zmanjšal in da ne bodo postali odvečna delovna sila. Optimizacija poizvedb, načrtovanje ter vzdrževanje podatkovne baze in še marsikatera druga naloga še naprej ostajajo v domeni skrbnikov baz. Zaradi odsotnosti poslovne logike bodo skrbniki po prehodu na N-nivojsko arhitekturo tem nalogam lahko posvetili celo več časa. Kontrola zahtevkov s srednjega nivoja bo v N-nivojskem sistemu ena njihovih prednostnih nalog. Takšni zahtevki nudijo ključne parametre, pomembne za optimizacijo poizvedb.

Vodstvo je kot najvplivnejša skupina v podjetju za uvajanje sprememb v računalniški sistem podjetja zelo pomemben dejavnik, ki pa spremembam v veliko primerih nasprotuje. Ni mu toliko pomembno, da bodo spremembe delovne procese v podjetju poenostavile, izboljšale in pohitrile. Parametri, ki ključno vplivajo na odločanje vodstva o rešitvah na ravni računalniških sistemov, so stroški rešitve, njen razvojni čas ter poslovne prednosti, ki iz rešitve izhajajo.

Položaju vodstva v podjetju navkljub so pri selitvi na N-nivojsko arhitekturo skrbniki podatkovnih baz najbolj problematična skupina. V njihovo prepričevanje je potrebno investirati daleč največ časa in energije. Kadar je govora o podatkovnih bazah, imajo njihovi skrbniki vedno končno besedo. Izjemoma lahko nad njihovim mnenjem prevlada samo nasprotna odločitev vodstva, ki pa navadno nima tehničnega znanja in zato skorajda brez izjem odločitve o bazah podatkov oz. spremembah, povezanih z njimi prepušča njihovim skrbnikom.

Ravno tako kot vodstvo nima tehničnega znanja s področja baz podatkov, tudi skrbniki baz nimajo znanja o N-nivojski arhitekturi. Vse, kar ne spada neposredno v podatkovni nivo, skrbniki podatkovnih baz obravnavajo izključno kot dodatnega odjemalca ali del nekega odjemalca, saj jim je arhitektura odjemalec/strežnik z njihovega vidika daleč najbližja. Deloma zaradi njej že tako podrejenega ustroja baz podatkov, deloma pa tudi zaradi dolgoletnega dela s tovrstnimi sistemi. Za spremembe – ne glede na to, kakšne so – se skrbniki baz načeloma ne zmenijo preveč, zanima jih izključno upravljanje s podatkovno bazo. Popuščajo praviloma samo takrat, kadar jim spremembe olajšajo obstoječe opravke in hkrati ne nalagajo dodatnega dela.

Večina podjetij v svoji kadrovski evidenci vodi le enega skrbnika podatkovnih baz, ki ga navadno zaposlujejo dlje časa. Nekateri skrbniki v istem podjetju z isto podatkovno bazo delajo že po deset, dvajset ali več let. Dolgoletne izkušnje z isto bazo ter ohranjanje celovitosti in nespremenljivosti njenih podatkov jim z leti dajo občutek moči v podjetju, zato je upor proti spremembam z njihove strani razumljiv. Nečesa, kar že tako dolgo zadovoljivo deluje, ne želijo spreminjati.
Seveda zgoraj našteto ne velja za administratorje v splošnem. Nekateri so manj problematični in se spremembam ne upirajo, če so te postavljene v normalne okvire. Vseeno je najboljša in najhitrejša pot do uspešne selitve na N-nivojsko arhitekturo tista prek vodstva. Vodstvo lažje poskrbi, da selitev sprejmejo tudi administratorji.

2.5. N-nivojska arhitektura

Menim, da bi morala biti vsaka poslovna aplikacija napisana večplastno. Če se na večplastnost osredotoča že med načrtovanjem, bo končni izdelek razbit na smiselne enote. Takšno poslovno aplikacijo bo po potrebi lažje razširiti, njeno vzdrževanje pa bo prav tako hitrejše.
Če se pokaže potreba po tem, da bi neka dobro zgrajena večplastna poslovna aplikacija beležila npr. vse izvedene stavke SQL, je med obstoječe plasti potrebno le vriniti plast za beleženje, preko katere bo speljan vsak zahtevek za dostop do baze.
V vseh primerih trije nivoji ne zadostujejo. Prav tako ni nujno, da je poslovni nivo sestavljen samo iz dveh plasti. Večinoma se v aplikacije dodaja še plast za prijavljanje in preverjanje vlog uporabnikov, plast za beleženje dogodkov aplikacije itd.
Podatkovni nivo obsega izključno podatkovno bazo. Na njem ni dodatnih plasti ali kakršnekoli druge funkcionalnosti.
Poslovni nivo sestavljata najmanj dve plasti. Poslovna plast in plast podatkovnega dostopa. Na tem nivoju definiramo tudi objekte za prenos podatkov.
Predstavitveni nivo je sestavljen iz ene ali dveh plasti. Če je uporabniški vmesnik zelo kompleksen, je njegovo logiko in njegov grafični prikaz pametno ločiti na neodvisna dela. Za posebej priročnega se tak pristop pokaže pri izdelavi sodobnih spletnih aplikacij, kjer del uporabniškega vmesnika z uporabo skriptnega jezika Javascript navadno kontrolira spletni brskalnik, preostali del pa kontrolira spletni strežnik.

2.5.1. Zakaj takšna arhitektura

Poleg prednosti, omenjenih v poglavju 2.2.2.3., obstajajo še drugi razlogi za N-nivojsko arhitekturo.
Programske komponente, ki jih zgradimo na ta način, so prenosljive. Lahko jih uporabljamo tudi drugje, ne le v okolju, za katerega so bile razvite (npr. v drugem sistemu), saj so med seboj neodvisne. Distribucija dela med razvijalci je na ta način dosti lažja.
Pri klasičnih aplikacijah, ki niso razbite na komponente, ni abstrakcije in pravil o tem, kje naj bi se nahajal kak del programske kode. To povzroča, da se določeni deli kode podvajajo (npr. v dveh različnih maskah istega uporabniškega vmesnika). Kadar se spremeni kodo na enem, je potrebno spremeniti tudi podvojeno kodo na drugem mestu. Takšne komplikacije močno povečajo možnost napak in povzročajo obilo dodatnega dela.
Če poslovno logiko ločimo od uporabniškega vmesnika, lahko le-tega po potrebi zamenjamo na enostaven način. Z minimalno truda lahko že v osnovi izdelamo več uporabniških vmesnikov.
Kljub dobrim vtisom pa je potrebno vedeti, da ima N-nivojska arhitektura tudi določene slabosti. Ena takšnih je „podajanje“ podatkov med plastmi tudi, kadar te z njimi ne naredijo

ničesar. Takšno početje je nujno potrebno, da bi komunikacija lahko potekala le med sosednjimi plastmi. Sistem je s tem dodatno obremenjen, kar pa pri sodobni računalniški opremi ni preveč opazno. Druga slabost N-nivojske arhitekture je povečano število projektov, ki jim je potrebno slediti; praviloma po en projekt na programsko komponento. V primeru, da se za vsako tabelo v podatkovni bazi izdela še podatkovni razred (več o tem v poglavju 2.5.3.2), dobimo še zelo veliko razredov. Izdelava številnih podatkovnih razredov pravzaprav niti ne predstavlja kakega ogromnega problema, saj so na voljo avtomatizmi v obliki generatorjev kode (CodeSmith, Visual Studio ...), ki se jih je pri tem moč poslužiti.

2.5.2. Podatkovni nivo

Shranjevanju podatkov rečemo tudi persistenčno shranjevanje, saj se podatki ohranijo tudi po izklopu sistema iz električnega omrežja.
Na voljo je več načinov shranjevanja podatkov. Odločiti se je možno za enostavnejše, kakršni so npr. shranjevanje v tekstovne, binarne ali XML datoteke, ali za kompleksnejše načine, kot je npr. shranjevanje v relacijsko podatkovno bazo.
Relacijske podatkovne baze predstavljajo najbolj priljubljeno metodo shranjevanja podatkov. Pri njihovi izbiri je potrebno upoštevati več faktorjev [2]:

- **Kakšna strojna oprema bo poganjala podatkovno bazo**

Komercialni produkti (Microsoft SQL Server, Oracle) so z vidika strojne opreme ponavadi bolj zahtevni kot zastonjske različice (PostgreSQL, MySQL, SQLite).

- **Hitrost**

Na hitrost podatkovne baze lahko vplivamo, če bazo podrobneje poznamo. Podatkovna baza Microsoft SQL Server je optimizirana za delo z baznimi procedurami, medtem ko je baza Oracle odlična pri uporabi pogledov na tabele. Če se pokaže potreba po povečanju zmogljivosti delovanja baze, je takšne informacije dobro upoštevati.

- **Razširljivost**

V zvezi z razširljivostjo je najpomembnejše vedeti, ali bo pri velikem številu podatkov podatkovna baza še dobro delovala. Oracle je npr. dobra izbira takrat, ko se vnaprej ve, da bo baza shranjevala ogromno količino podatkov.

- **Vzdrževanje**

Z vidika enostavnosti vzdrževanja se odlično obneseta podatkovni bazi Microsoft SQL Server in PostgreSQL. Za izdelavo varnostne kopije podatkovne baze PostgreSQL je potrebno samo klikniti na gumb, ki avtomatično izdela arhivsko datoteko z vsemi podatki. Postopek pri bazi Microsoft SQL Server ni dosti bolj zapleten.

- **Kako obširna je njena funkcionalnost**

Z vidika funkcionalnosti je posebno pozornost potrebno posvetiti podatkovni bazi MySQL, ki nima popolno standardizirane sintakse jezika SQL. V določenih primerih je potrebno stavke za bazo MySQL napisati popolnoma drugače, kot pa bi se jih pisalo v drugih podatkovnih bazah, saj v nasprotnem primeru ne vrnejo želenega rezultata.
Funkcionalnost shranjevanja prostorskih geografskih podatkov v podatkovno bazo ponujajo MySQL, PostgreSQL ter komercialna različica baze Oracle. Microsoft SQL Server podpore za shranjevanje te vrste podatkov nima vgrajene.

- **Cena**

Kar se tiče cene, je potrebno vedeti, da je večina popularnih podatkovnih baz na voljo v brezplačni različici, vendar je funkcionalnost takšnih različic praviloma rahlo okrnjena. Nasploh pa tudi brezplačne podatkovne baze za uporabo v komercialne namene niso brezplačne (ker to omejuje licenca GPL), so pa cenejše od npr. podatkovne baze Oracle.

- **Kompatibilnost z razvijalskim orodjem**

Pri izbiri je pametno preveriti tudi, kako se podatkovna baza ujema z razvijalskim orodjem, ki bo uporabljeno pri razvoju. Microsoft Visual Studio in Microsoft SQL Server (Express Edition) sta zelo kompatibilna izdelka in omogočata hiter razvoj. Aplikacije, pisane v jeziku PHP, večinoma uporabljajo podatkovni bazi PostgreSQL ter MySQL, ASP.NET aplikacije pa Microsoft SQL Server.
Čeprav bo večina funkcionalnosti N-nivojskega poslovnega sistema implementiranih na poslovnem nivoju, bo na podatkovni bazi vseeno potrebno poskrbeti za marsikatero stvar. Tabelam je potrebno zagotoviti primarne ključe in medsebojne relacije s pomočjo teh ključev. Zaradi hitrejšega poizvedovanja za podatki je dobro indeksirati polja v tabelah. Če je nujno potrebno, lahko v bazi implementiramo tudi kakšen prožilec, sicer pa je to že eno izmed opravil, ki se jih izvaja na poslovnem nivoju.

2.5.2.1. CRUD operacije

CRUD je kratica za angleške izraze:

- Create (ustvari)
- Read (preberi)
- Update (posodobi)
- Delete (pobriši)

CRUD predstavlja štiri osnovne funkcionalnosti persistenčnega shranjevanja, ki ga potrebuje vsaka večja aplikacija. Za podatkovne baze lahko te operacije preslikamo v standardne stavke SQL (tabela 4).

operacija CRUD	**stavek SQL**
Create	INSERT
Read	SELECT
Update	UPDATE
Delete	DELETE

Tabela 4. Preslikava operacij CRUD v stavke SQL

V kolikor aplikacija podpira vse štiri operacije, smatramo, da je z vidika shranjevanja podatkov popolna [3], kar je hkrati tudi potrditev tega, da na podatkovnem nivoju ni treba početi ničesar drugega, kot izvajati te operacije.
Uporaba tega pravila nam je torej lahko v oporo, ko se odločamo o tem, kakšne naloge naj podatkovni nivo izvaja.

2.5.3. Poslovni nivo

Poslovni nivo predstavlja najpomembnejši del N-nivojske arhitekture. Poleg ostalih nalog opravlja tudi funkcijo povezovalnega člena med predstavitvenim in podatkovnim nivojem.
Poslovni nivo vsebuje vso poslovno logiko, transformacije podatkov med obliko za prikaz in obliko za shranjevanje ter obratno, validacijo podatkov itd.
Pri poslovnem nivoju je ločnica med nivojem in plastjo najbolj zamegljena. Plast podatkovnega dostopa služi izključno poslovnemu nivoju, objekti za prenos podatkov pa komunikaciji s predstavitvenim nivojem, kar kaže na to, da je poslovni nivo hkrati pravzaprav tudi poslovna plast.
Microsoftova definicija [4] pravi, da je poslovni nivo kombinacija validacije vnesenih podatkov, preverjanja prijav, poizvedb po podatkovni bazi, politike dostopa in algoritemskih transformacij, ki določajo način poslovanja podjetja.

2.5.3.1. Poslovna plast

Poslovna plast je glavna plast N-nivojskega računalniškega sistema. Vanjo je implementirana celotna funkcionalnost omenjena v prejšnji točki (poslovna logika, validacija, transformacije itd.).
Poslovno plast predstavljajo realni poslovni objekti (npr. bančni računi, posojila, inventar) [5], implementirani kot razredi. Nekatere izmed metod teh razredov lahko preko omrežnega protokola ponudimo v uporabo predstavitvenem nivoju. Te metode predstavljajo vstopno točko do poslovne logike. Preostale metode poslovnih objektov so namenjene npr. avtomatski pripravi mesečnih faktur ali računov, računanju letnih ali mesečnih obresti itd. Periodično se jih proži s pomočjo kake sistemske storitve ali druge podobne rešitve.
Načrtovanje poslovne plasti že samo po sebi ni lahko opravilo, kompleksnejša poslovna logika pa celotno načrtovanje še dodatno oteži. Problemi se največkrat pojavijo zaradi kratkih časovnih rokov za izdelavo poslovnega sistema oz. njegovo selitev na novo arhitekturo ali ob izvajanju aktivnosti pred načrtovanjem (tj. analize poslovanja podjetja). Prehitro izdelana in zato netočna analiza je odlična podlaga za slabo implementacijo poslovnega nivoja.
Pri načrtovanju se praviloma nezavedno uporablja enega izmed treh vzorcev [6]:

- podatkovno voden vzorec,
- predstavitveno voden vzorec in
- izolacijski vzorec.

Pri podatkovno vodenem vzorcu je najprej oblikovan podatkovni nivo, okoli njega se v drugem koraku izoblikuje predstavitveni in na koncu še poslovni nivo (konkretno plast – poglavje 2.5.3).
Ta vzorec je pogost, ker je podoben tradicionalnemu pristopu k razvoju aplikacij vrste odjemalec/strežnik. Uporablja se ga tudi pri sistemih, oblikovanih okrog obstoječe podatkovne baze.
Lastnost aplikacij, grajenih po podatkovno vodenem vzorcu, je, da maske v njihovih uporabniških vmesnikih odražajo strukturo podatkovne baze. Intuitivnost je v takih aplikacijah zelo okrnjena, saj so grajene okoli relacijskega modela podatkovne baze in ne okrog uporabnika in njegovega načina razmišljanja.
V ciklu razvoja se je potrebno vračati nazaj, na razvoj podatkovnega nivoja, ker določenih podatkov iz podatkovne baze ne moremo pridobiti v obliki, pravilni za prikaz na predstavitvenem nivoju. V takšnih primerih mora razvijalec uporabniškega vmesnika sprožiti zahtevo za spremembo strukture podatkovne baze v korist predstavitvenemu nivoju, kar gre seveda v škodo podatkovnemu nivoju. Težava je v tem, da takšne spremembe na predstavitveni strani sicer lahko rešijo določen problem, na podatkovni pa povzročijo nove ali pa poglobijo obstoječe probleme. V kolikor se za spremembe podatkovnega nivoja v korist predstavitvenega nivoja le odločimo, se moramo zavedati, da z njimi navadno kršimo splošna pravila normalizacije podatkovne baze, saj takšne spremembe nemalokrat vključujejo podvajanje podatkov v bazi.
Nekoliko drugačen je pristop pri predstavitveno vodenem vzorcu. Podatkovni nivo je v tem primeru namreč oblikovan okrog predstavitvenega. Poslovni nivo v tem primeru vsebuje le manjše transformacije. Podatkovna baza je slabo oblikovana, ni normalizirana in zato z vidika zmogljivosti počasna, saj je celotno shranjevanje podatkov podrejeno prikazu in čim lažjemu dostopu do podatkov s strani predstavitvenega nivoja.
Pri izolacijskem vzorcu se predstavitveni in podatkovni nivo razvijata paralelno, neodvisno drug od drugega. Tak pristop izloči vse škodljive dejavnike prejšnjih dveh vzorcev (podatkovni in predstavitveni nivo se ne prilagajata drug drugemu). Po končanem razvoju podatkovnega in predstavitvenega nivoja je poslovni nivo potrebno oblikovati tako, da le-ta vsebuje vse transformacije, s katerimi zagotovi podporo obema preostalima nivojema. Ker sta

predstavitveni in podatkovni nivo med seboj popolnoma ločena, lahko kateregakoli od njiju spreminjamo, ne da bi s tem vplivali na drugega – za potrebe združljivosti nivojev je ob takšnih spremembah treba le še popraviti oz. dodelati transformacijo podatkov na poslovnem nivoju.
Izolacijski vzorec je za razvoj dobro strukturirane aplikacije najboljši pristop. Podatkovna baza se lahko oblikuje po priporočilih in standardih, sistem se podreja poslovnim zahtevam in je enostaven za nadgrajevanje, hkrati pa v kar največji meri zadovolji pričakovanja uporabnika.
Pri izgradnji poslovne plasti je potrebno upoštevati, da se do podatkovnega vira nikdar ne dostopa neposredno. Vsak dostop do podatkov mora opraviti plast podatkovnega dostopa. S takšnim pristopom zagotovimo neodvisnost poslovnega nivoja od podatkovnega vira. Ob spremembi le-tega (npr. menjava podatkovne baze iz Oracle v MySQL) nam tako ni potrebno popravljati poslovne plasti.

2.5.3.2. Plast podatkovnega dostopa

Namen te plasti je prevzem celotne funkcionalnosti dostopa do podatkovnega vira. Namesto da bi stavke SQL za poizvedbe, ažuriranje, ustvarjanje ali brisanje zapisov neprestano pisali v poslovni plasti, z nje raje kličemo funkcije plasti podatkovnega dostopa. Takšno grupiranje funkcionalnosti tudi marsikaj poenostavi, saj so vsi stavki SQL in celotna logika dostopa do baze na enem, dosti bolj preglednem mestu.
Plast podatkovnega dostopa opravlja sledeče naloge:

- abstrakcijo klicev operacij CRUD,
- zaklepanje tabel in nadzor nad hkratnim ažuriranjem zapisov,
- nadzor nad avtentikacijo,
- upravljanje s transakcijami,
- pomnjenje že prebranih podatkov in preverjanje tega, kdaj jih je treba ponovno prebrati itd.

Vgraditev mehanizmov za opravljanje opisanih nalog je lahko sila utrujajoča naloga. Za poenostavitev tega zamudnega dela obstajajo generatorji kode (Linq to SQL, NHibernate, CodeSmith). Še pred kakima dvema letoma so razvijalci to plast implementirali ročno. Eden izmed razlogov za obilo ročnega dela je bilo nezaupanje generatorjem kode, mnogi razvijalci so namreč verjeli, da sami lahko pišejo lepšo in boljšo kodo, nekateri pa za generatorje kode preprosto niso vedeli.
Podobno se je dogajalo tudi v časih programiranja v zbirnem jeziku, saj razvijalci niso verjeli, da lahko prevajalnik jezika C izdela boljšo kodo, kot jo sami napišejo ročno. O tem, koliko ljudi dandanes programira v zbirniku, verjetno ni potrebno izgubljati besed.
Da bi se ohranila čim globlja abstrakcija podatkovnega vira, se iz njega ob poizvedbah ne vrača vrstic in stolpcev tabel, temveč razrede, v katere so preslikane tabele, izdelati pa jih zna kak generator strojne kode. Tipizirane lastnosti razreda so preslikave polj, razred sam predstavlja eno vrstico v tabeli, seznam določenega razreda pa več vrstic.
Takšni razredi so uporabni v več pogledih. Omogočajo preverjanje napak v času prevajanja programa in v tistih razvojnih okoljih, ki vključujejo mehanizem „pomoči pri programiranju“ (Intellisense), nudijo razvijalcu nekakšno oporo pri vizualizaciji tabel in polj v njih. Pri predstavitveno vodenem vzorcu razvoja se te razrede navadno uporabi tudi kot objekte za prenos podatkov (opisani so v naslednjem poglavju), saj podatkovna baza pri tem vzorcu odraža zahteve uporabniškega vmesnika (poglavje 2.5.3.1).

2.5.3.3. Objekti za prenos podatkov

Objekti za prenos podatkov so navidez identični razredom, opisanim v prejšnjem poglavju. Razlikujejo se po tem, da so lahko slika ne samo ene tabele ampak tudi neke splošne poizvedbe SQL. Njihova glavna lastnost je, da so ponavadi v celoti sestavljeni izključno iz tipiziranih lastnosti. Nekateri bolj kompleksni imajo za potrebe obveščanja o spremembi podatkov vgrajene še prožilce dogodkov (events), ki se izvedejo ob spremembi katere od lastnosti.
Tudi pri implementaciji objektov za prenos podatkov se uporablja generatorje, saj je njihova ročna izdelava zelo obširna, rutinska in dolgočasna naloga. Ob spremembah poizvedb SQL je sicer potrebno spremeniti tudi odgovarjajoče objekte za prenos podatkov, kar predstavlja precej dela, vendar se izdelave objektov za prenos podatkov splača lotiti, saj se z njimi doseže tipiziranost podatkov.
V poglavju 2.5.3.1. je omenjena transformacija podatkov in izolacijski vzorec. Z izrazom transformacija je opisana preslikava podatkov iz oblike, primerne za podatkovno bazo, v obliko, primerno za uporabniški vmesnik, in obratno (tj. iz razredov, ki so slika tabel (poglavje 2.5.3.2.), naredimo objekte za prenos podatkov in obratno).
Odločitev o tem, kakšne objekte potrebujemo, je pri izolacijskem vzorcu enostavna. Razvijalec uporabniškega vmesnika oblikuje maske tako, da te zadovoljijo uporabnika. Pri tem mu relacijski model baze ni znan [7]. Če v maski potrebuje npr. podatke določene stranke in vsa njena naročila, potem mora razvijalec poslovne plasti izdelati objekt s tipiziranimi lastnostmi, ki predstavljajo informacije o stranki, ena od lastnosti pa predstavlja seznam naročil.

2.5.4. Predstavitveni nivo

Gledano na poenostavljen način sta predstavitveni nivo in uporabniški vmesnik eno in isto. Uporabniški vmesnik mora odražati želje uporabnika in ne arhitekture računalniškega sistema. Z uporabo izolacijskega vzorca takšnemu pogledu tudi sledimo.
Uporabniški vmesnik je lahko namizna ali spletna aplikacija, lahko pa tudi aplikacija kake druge vrste.
Če se osredotočimo na namizno in spletno aplikacijo, lahko med njima opazimo določene razlike.
Pri namizni aplikaciji lahko urejamo več objektov za prenos podatkov naenkrat, nato pa vse spremembe posredujemo poslovnemu nivoju, njen grafični uporabniški vmesnik pa je lahko bogat in odziven. Vsa ta moč pa ima tudi slabo stran – razvoj takšnih uporabniških vmesnikov je dolgotrajen in zapleten proces, za katerega je potrebno ogromno programerskega znanja in dobro poznavanje knjižnic za razvoj grafičnega vmesnika.
Spletna aplikacija je po navadi napisana tako, da z njo lahko urejamo le en objekt za prenos podatkov naenkrat, saj po interpretaciji strani v spletnem brskalniku v pomnilniku ne ostane nobena podatkovna struktura, ki bi predstavljala naše podatke. Objekti za prenos podatkov se nahajajo na spletnem strežniku, ta pa do brskalnika pošilja le upodobitev oz. grafično podobo strani v obliki HTML. Vsaka sprememba strani povzroči ponovno obdelavo na spletnem strežniku, ki brskalniku vsakič pošlje nov rezultat. Neprestano osveževanje strani se do neke mere lahko omejuje npr. z uporabo tehnologije AJAX, a tudi ta vsak nov podatek pridobi od strežnika, le da ob tem ne osvežuje celotne strani.
V sistemu s spletno aplikacijo je predstavitveni nivo pravzaprav spletni strežnik. Če pride do sprememb na uporabniškem vmesniku, je le-tega potrebno posodobiti samo na strežniku. V primeru namizne aplikacije je to nemogoče, saj moramo vsakemu od odjemalcev, ki so neodvisnega značaja, posredovati novo verzijo. Pri večjih sistemih lahko takšno posodabljanje hitro pripelje do nepreglednosti, saj je včasih potrebno posodobiti tudi po več tisoč

računalnikov. Spletna aplikacija je v tem primeru boljša izbira, saj jo je mnogo lažje implementirati in vzdrževati.

Dodatna prednost spletne aplikacije je tudi njena neodvisnost od operacijskih sistemov, saj deluje na vseh sistemih (edini pogoj za njeno delovanje je obstoj spletnega brskalnika v sistemu), česar pa za namizno aplikacijo ne moremo trditi.

Sodobni spletni uporabniški vmesniki, kakršen je npr. Google Mail, so tako zmogljivi, da se lahko kosajo z namiznimi aplikacijami. Trenutni trend v razvoju spletnih aplikacij so za domačo rabo dovolj zmogljiva spletna pisarniška orodja.

Občasno se dogaja tudi, da se pokaže potreba tako po spletni kot tudi po namizni aplikaciji. Razvoj dveh tako različnih uporabniških vmesnikov ni ravno prijetno delo. Microsoft je pri tem naredil korak naprej, saj s pomočjo tehnologij WPF (Windows Presentation Foundation) in Silverlight omogoča razvoj enega samega uporabniškega vmesnika za obe platformi, namizno (operacijski sistem) in spletno (spletni strežnik). Več informacij o obeh tehnologijah je na voljo na Microsoftovi domači strani.

Pri obeh vrstah uporabniškega vmesnika je potrebno zagotoviti dostop do poslovnega nivoja. Da se razvijalcem ne bi bilo potrebno ukvarjati z načinom, kako ga bodo zagotovili, imajo na voljo generatorje, ki ovijejo kompleksnost dostopa do podatkov preko omrežnih protokolov v enostavne razrede (proxy).

Najpomembneje pri implementaciji predstavitvenega nivoja je, da se v njem ne uporablja oz. se vanj ne implementira nikakršna poslovna logika. Prostor zanjo je na poslovnem nivoju. Z namenom izdelave boljšega uporabniškega vmesnika (trenutna validacija, omogočanje oz. onemogočanje določenih polj, prikrivanje oz. skrivanje delov maske itd.) se lahko na predstavitvenem nivoju uporablja izključno kopija poslovnih pravil.

3. Praktični primer implementacije

Z dobrim poznavanjem orodij in tehnologij lahko N-nivojski sistem implementiramo na dokaj enostaven način. Najnovejša orodja lahko proces razvoja poenostavijo do te mere, da mora razvijalec razmišljati le še o tem, kako bo implementiral poslovno logiko.
Znanje, potrebno za tak pristop, je kar obširno, krivulja učenja strma, vendar pa se vložen trud na koncu obrestuje.
Z znanjem, ki sem ga pridobil med raziskovanjem N-nivojske arhitekture, bi lahko nekemu nepoznavalcu zelo enostavno razložil, kje so njene prednosti. Zaradi jasno razvidnih prednosti te arhitekture je vpeljava novih razvijalcev v njeno uporabo enostavna, zato le-ti hitro postanejo produktivni.
Da bi se izognil dodatnim komplikacijam, sem se v primeru osredotočil izključno na programsko ogrodje Microsoft .NET. Ostalih ogrodij pri razvoju nisem uporabljal, s čimer sem se ognil dodatnemu učenju uporabe teh ogrodij, ki bi jih v najslabšem primeru lahko uporabil le pri tem projektu. Poleg omenjenega prinašajo „tretja“ aplikativna ogrodja tudi problem odvisnosti od njihovih razvijalcev, ti pa pri novih verzijah marsikdaj pozabijo na kompatibilnost s starejšimi verzijami ogrodij. Sam se jim danes iz praktičnih razlogov izogibam, ker sem takšne probleme na lastni koži že večkrat izkusil, npr. pri uporabi ogrodja Autodesk Object Arx.
Pri implementaciji primera sem se želel izogniti tudi pisanju odvečne programske kode. Z odvečno programsko kodo mislim na tisti del kode, ki programerjem pri razvoju N-nivojskih aplikaciji predstavlja nujno zlo – „proxy“ razredi, serializacijska koda itd.
Pri razvoju nam je lahko v veliko pomoč tudi aplikacija za vodenje projektov, s katero je možno na enostaven način dodeljevati opravila delovni skupini oz. njenim posameznim članom, hkrati pa se da z njo učinkovito nadzirati stanje projektov.
Delo v skupini zahteva tudi distribucijo posameznih delov programske kode ustreznim članom skupine ter nadzor nad spremembami v njej. V ta namen je pametno uporabiti enega izmed sistemov za revizijo kode.

3.1. Vodenje projekta

Razvoj vsake aplikacije je projekt, ki ga je priporočljivo nadzorovati. Tak nadzor mora med drugim obsegati čim bolj natančno evidenco že opravljenih nalog in tudi evidenco nalog, ki jih je še treba postoriti. Za vsako nalogo je dobro vedeti tudi, kdo jo je opravljal in koliko časa je zanjo porabil.
V kolikor je takšna evidenca dobro vzdrževana, jo lahko uporabimo za izdelavo kvalitetnih analiz, vizualizacij, statistik, lahko služi kot osnova za istavo računa naročniku itd.
Obstaja precej aplikacij, ki omogočajo nadzor nad projekti in vodenje projektov na različne načine, vsaka pa zagovarja svoj pristop k problemu. Sam sem izbral izdelek JIRA, ki ga ponuja podjetje Atlassian (http://www.atlassian.com/software/jira/). Aplikacija sicer ni

brezplačna, omogoča pa mnogo več kot običajni produkti za projektno vodenje. Poleg projektov in nalog, vezanih nanje, aplikacija obvladuje tudi vodenje obstranskih nalog. Prednost rešitve JIRA je tudi ta, da zna naloge povezati s sistemom za revizijo kode, kar razvijalcem omogoča tudi pregled nad tem, katera naloga je povzročila spremembo v kodi.
JIRA je spletna aplikacija, ki jo po svetu uporablja več kot 9000 podjetij, med katerimi je nemalo takih, ki tako po velikosti kot po prepoznavnosti spadajo v sam svetovni vrh. Kratka navodila za uporabo aplikacije so na strani http://www.atlassian.com/software/jira/docs/latest/.

3.1.1. Delitev dela

Delo, ki ga zahteva nek projekt, je potrebno v obliki nalog razdeliti med razvijalce v projektni skupini.
V splošnem lahko projektne naloge razdelimo na tri področja:

- razvoj uporabniškega vmesnika,
- razvoj poslovne logike in
- modeliranje podatkovne baze.

Menjavanje vlog posameznih razvijalcev mora potekati tako, da en razvijalec ne razvija ves čas samo uporabniškega vmesnika ali pa poslovne logike. S pomočjo raznolikosti nalog sistem spozna bolj podrobno in z več vidikov. Poleg tega menjavanje vlog zagotavlja, da bo v primeru razvijalčevega odhoda delo na projektu nemoteno potekalo naprej, saj njegovi kolegi problematiko poznajo tudi z njegovega zornega kota [6].
Menjavanje vlog razvijalcev ne sme potekati brez nadzora. Najboljši pristop k nadzorovanju je ta, da en razvijalec pri razvoju določenega modula nima več kot ene vloge, ko pa svoje delo tam zaključi, se razvijalcu dodeli ena od vlog iz drugega modula. V kolikor dovolimo, da en razvijalec v enem modulu opravi več kot dve vlogi, se možnost kršitev pravil abstrakcije poveča.
Ista pravila se morajo upoštevati tudi, če ima določeno vlogo v razvoju modula več razvijalcev hkrati. Pravila menjave vloge v istem modulu se ne sme kršiti pri nobenem od vpletenih razvijalcev.
V zelo majhnih skupinah ali pa v primeru, ko aplikacijo razvija posameznik, možnosti deljenja dela ni. V takem primeru mora razvijalec hkrati delati le na eni vlogi (npr. med razvijanjem poslovne logike naj ne bi popravljal še uporabniškega vmesnika). Priporočljiv vrstni red vlog je:

- oblikovalec podatkovne baze,
- razvijalec uporabniškega vmesnika in na koncu
- razvijalec poslovne logike.

Možen je tudi drugačen pristop. Razvijalec lahko celotno delo, ki ga zahteva njegova vloga, opravi preko vseh modulov in nato preide na drugo vlogo. Takšen pristop kvalitetno izolira funkcionalnost posameznih plasti, kljub temu pa ima tudi slabosti: nenehno preskakovanje med moduli lahko razvijalca zmede, zaradi česar je za takšen pristop potrebno celoten načrt aplikacije izdelati vnaprej, pred začetkom samega razvoja.

3.2. Revizija izvorne kode

Upravljanje z izvorno kodo je že dolgo časa kritično početje za programerje. Posamezni razvijalci si s sistemi za revizijo izvorne kode pomagajo npr., kadar v izvorni kodi napravijo spremembe, ki jih nato želijo razveljaviti. V razvijalskih skupinah je normalno, da več razvijalcev hkrati popravlja določeno datoteko, ob koncu dneva pa je potrebno ugotoviti čigave spremembe so najboljše – sistemi za revizijo kode lahko pomagajo tudi pri tem.

Sam sem izbral odprtokodno rešitev Subversion (http://subversion.tigris.org/), ki je trenutno najbolj uporabljan sistem revizije.
Da bi delo v skupini potekalo kar najbolj optimalno, je pametno postaviti Subversion strežnik, na katerega je mogoče dostopati preko interneta, saj v tem primeru delo lahko poteka tudi od doma ali s kake druge oddaljene lokacije. Strežnik pa ni nujno potreben za uporabo sistema Subversion, saj le ta podpira sintakso URL za dostop do repozitorija (http://, https://, svn://, svn+ssh://, file:/// itd.) in s tem omogoča, da je centralni repozitorij izvorne kode na praktično kateremkoli računalniku povezanem v lokalno omrežje. V tem primeru je do repozitorija potrebno dostopati z URL predpono „file:///“.
Subversion je sistem, ki deluje v ukazni vrstici, kar pogosto ni najbolj praktično. Obstaja veliko grafičnih vmesnikov, ki v ozadju izvajajo ukaze za Subversion. Eden izmed njih je odprtokodni TortoiseSVN (http://tortoisesvn.tigris.org/), katerega dobra lastnost je, da se integrira v program za delo z datotekami Windows Explorer, zaradi česar se uporabniku ni treba učiti dela z novim uporabniškim vmesnikom.
Dokumentacija za delo s programom Subversion je na voljo na spletni strani http://svnbook.red-bean.com/en/1.4/index.html. Dokumentacija za TortoiseSVN se distribuira skupaj s programom (v obliki pomoči).

3.3. Razvojno okolje

Razvojno okolje, ki sem ga izbral za izdelavo aplikacije, je Microsoft Visual Studio 2008, ki je eno izmed glavnih orodij za razvoj v operacijskem sistemu Microsoft Windows. Na voljo je več različic. Visual Studio Express je le malo okrnjena in na več delov razbita inačica licenčnega produkta Visual Studio Standard, tega je možno nadgraditi še v različici Visual Studio Professional in Visual Studio Team System. Pregledna primerjalna tabela funkcionalnosti, ki jo ponuja posamezen produkt, je na strani http://en.wikipedia.org/wiki/Microsoft_Visual_Studio.
Prevajalniki za okolje .NET so na voljo v paketu izvajalnih knjižnic (.NET Framework Runtime), tako da lahko razvijalec za pisanje izvorne kode, uporabi katerikoli tekstovni urejevalnik, vseeno pa je priporočljiva uporaba razvojnega okolja.
Dobre lastnosti uporabe razvojnega okolja so:

- barvanje izvorne kode,
- IntelliSense pomoč pri programiranju,
- podpora refaktoriranju izvorne kode,
- vgrajen razhroščevalnik,
- grafični oblikovalec pogovornih oken,
- grafični oblikovalec razredov,
- čarovniki,
- različne predloge za projekte,
- vgrajena podpora dostopa do podatkovnih baz in podpora jeziku SQL ...

Visual Studio 2008 podpira knjižnico .NET verzije 3.5, ki vsebuje:

- .NET knjižnico 2.0,
- jezik C# verzije 3.0,
- jezik Visual Basic.NET 9.0,
- Windows Communication Fundation (WCF),
- Windows Presentation Foundation (WPF),
- Windows Worklow Foundation (WWF),
- CardSpace.

Glavni novosti, ki sta pomembni pri implementaciji spodaj opisanega N-nivojskega sistema, sta nova verzija jezika C# (poglavje 3.4) in WCF (poglavje 3.5).

Za podatkovno bazo sem izbral Microsoft Sql Server 2005 (Express), ki je trenutno edina, ki jo popolnoma podpira tehnologija LINQ (Language Integrated Query). V prvem paketu popravkov za Visual Studio 2008 bo dodana podpora LINQ še za ostale podatkovne baze v nekoliko izboljšani različici. Več o tem, kaj je LINQ, zakaj je tako praktičen in kaj pridobimo če ga uporabimo je napisano v poglavju 3.4.8.
Več podatkov o razvojnem okolju Visual Studio je na voljo na http://msdn2.microsoft.com/en-us/vstudio/default.aspx.
Brezplačni različici razvojnega okolja Visual Studio in podatkovne baze SQL Server sta na voljo na http://www.microsoft.com/express/.

3.4. Novosti v jeziku C# verzije 3.0

Verzija 3.0 v programski jezik C# ni prinesla veliko novosti. Največ sprememb je bilo potrebnih za vpeljavo poglavitne novosti – jezika LINQ, nekatere od ostalih novosti pa ponujajo pomoč pri izdelavi bolj kompaktne programske kode [8].

3.4.1. Avtomatska implementacija lastnosti

Implementacija lastnosti razreda, ki ne vsebuje nobene logike, je bila v prejšnji verziji dolga več vrstic. Naslednji primer prikazuje razliko med tem, kako lastnosti razreda implementiramo v verziji 3.0 in kako ga je bilo treba implementirati v prejšnjih verzijah jezika C#.

- Prejšnje verzije

```
private string ime;

public string Ime
{
    get { return ime; }
    set { ime = value; }
}
```

- Verzija 3.0

```
public string Ime { get; set; }
```

Lastnosti služijo razredom zato, da ne bi po nepotrebnem izpostavljali svojih spremenljivk. Z njimi je možno tudi eksplicitno definirati, ali neko spremenljivko lahko samo beremo ali vanjo lahko tudi pišemo, omogočimo podatkovno spajanje (databinding), serializacijo in lažje dodajanje logike (npr. pri nastavljanju vrednosti).

3.4.2. Inicializacija objektov in seznamov

V prejšnjih verzijah jezika C# smo nek razred najprej inicializirali in šele nato definirali njegove lastnosti. Za zelo praktično se je izkazala možnost, da se lastnosti definira s pomočjo konstruktorja, ki pa je seveda moral biti na voljo. Konstruktor vseeno ni bil najboljša možnost za inicializacijo (npr. kadar je potrebno nastaviti le dve lastnosti, nam konstruktor ponuja le možnost nastavitve desetih lastnosti hkrati), kar je pomenilo, da se je določen delež objektov še vedno najprej inicializiral, nakar se je njegove lastnosti naknadno opredelilo. V novi verziji jezika C# so ta problem zaobšli na naslednji način:

- Prejšnje verzije

```
Oseba o = new Oseba();
o.Ime = "Marko";
o.Priimek = "Podgoršek";
o.Starost = 29;
```

- Verzija 3.0

```
Oseba o = new Oseba {
    Ime = "Marko",
    Priimek = "Podgoršek",
    Starost = 29 };
```

Razred je sedaj možno inicializirati in mu nato opredeliti le tiste lastnosti, ki so potrebne.

3.4.3. Razširitvene metode

Da bi se katerega od razredov, ki so priloženi v knjižnici .NET (npr. *System.String*) razširilo, je bilo v prejšnjih verzijah jezika C# potrebno razred dedovati in ga razširiti v novega, kar pa ni ravno praktično. Tak pristop je posebej problematičen pri razredih, ki se v aplikaciji uporabljajo pogosto – takšen je npr. *System.String*. Sprememba tega razreda zahteva od razvijalca, da v celotnem projektu nadomesti besedo *String* z imenom novega razreda, npr. *mojString*. V novi verziji jezika C# je možno napisati metodo, ki nadgrajuje nek razred. To povemo z besedo *this* na naslednji način:

```
public static bool isValidEmail(this string s)
{
    Regex r = new Regex(@"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$");
    return r.IsMatch(s);
}
```

Uporaba:

```
string email = "marko.podgorsek@email.si";
if(email.isValidEmail()) { }
```

3.4.4. Definicije delnih metod

C# 2.0 je predstavil nov koncept delnih razredov. Ti omogočajo razbitje programske kode razreda na dve ali več izvornih datotek. Pri generiranju izvorne kode je tak koncept zelo uporaben. Generator naredi eno datoteko, razvijalec pa lahko doda svoje metode v drugo datoteko. Logika obeh se zato ne prepleta, poleg tega pa razvijalčeva koda ostane popolnoma nedotaknjena s strani generatorja.
Delne metode so zelo podobne delnim razredom. Omogočajo, da v eni od izvornih datotek napovemo obstoj metode, ki jo implementiramo v drugi izvorni datoteki. Eden glavnih namenov delnih metod je implementacija enostavnih dogodkov kot npr.:

```
partial void OnImeChanged();
```

Takšno metodo generira generator LINQ to SQL, ki za vsako polje v tabeli naredi svojo definicijo delne metode. Razvijalec lahko nato v drugi datoteki to definicijo implementira, npr. za preverjanje dolžine imena, ki ne sme preseči dolžine polja v podatkovni tabeli.

3.4.5. Lambda izrazi

Lambda izraz ni enostaven koncept, ki bi ga bilo mogoče razumeti v trenutku. O lambda izrazih se najlaže govori kot o nečem, kar deluje na nekem seznamu, tako da na vsakem elementu v tem seznamu izvede neko operacijo.

```
osebe.Where(o => o.Priimek == "Podgoršek");
```

Izraz v primeru nam vrne vse osebe iz seznama oseb, katerih priimek je Podgoršek. Takšno funkcionalnost bi brez uporabe lambda izrazov lahko izvedli tako, da bi želene osebe iskali s sprehodom skozi celoten seznam s pomočjo ene od zank.

3.4.6. Implicitno tipizirane lokalne spremenljivke

Implicitno tipizirane lokalne spremenljivke so definirane s ključno besedo *var*, ki v tem primeru ne predstavlja podatkovnega tipa *variant*, kakršnega poznata npr. Visual Basic in JavaScript.
Visual Basic in JavaScript tip spremenljivke določata v času izvajanja, pri implicitno tipizirani spremenljivki pa njen podatkovni tip določi prevajalnik. Takšne vrste spremenljivka je enakovredna npr. podatkovnemu tipu *System.Int32*, le da razvijalec tega ne rabi eksplicitno opredeljevati.
Poudariti je treba, da te spremenljivke delujejo samo lokalno (npr. v eni metodi). Izven lokalnega obsega se takšne spremenljivke ne da uporabljati.
Primer:

```
var i = 5;
var s = "Tekst";
i = s; //Napaka
```

3.4.7. Anonimni tipi

Anonimni tipi omogočajo definicijo razredov brez potrebe po njihovi formalni deklaraciji. Anonimni tipi so močno povezani z implicitno tipiziranimi lokalnimi spremenljivkami in s tehnologijo LINQ.
Primer:

```
var defrazreda = new {
    Ime = "Marko",
    Priimek = "Podgoršek",
    Starost = 29 };
```

Kot je razvidno iz primera, za ključno besedo *new* ni napisan noben podatkovni tip. Na njegovem mestu se nahaja sintaksa za inicializacijo lastnosti iz poglavja 3.4.2. Spremenljivka *defrazreda* je anonimnega tipa (nima formalne deklaracije), a ker je enake oblike kot tip *Oseba*, se jo v ta tip lahko pretvori (cast).

3.4.8. LINQ

LINQ nam omogoča, da poizvedbe nad podatki zapišemo neposredno v izvorno kodo. Poizvedujemo lahko po objektih, ki temeljijo na vmesniku *IEnumerable*. Takih objektov je veliko. Določene objekte, ki ne temeljijo na tem vmesniku, še vedno lahko pretvorimo v pravo obliko s pomočjo vgrajenih razširitvenih metod.

Obstajajo različni ponudniki LINQ-ja (LINQ to System search, LINQ to Entities, LINQ to Google [9]), ki omogočajo njegovo uporabo nad podatki, pridobljenimi z različnimi tehnologijami. Z njimi se lahko na enak način izvaja poizvedbe na strukturah v pomnilniku, datotekah ali pa nad podatki v podatkovni bazi. Ponudnik lahko pretvori poizvedbo LINQ v stavek SQL za poizvedbo nad podatkovno bazo, v stavek XPath za poizvedbo po datoteki XML ali v kaj drugega.
Možnosti uporabe LINQ-ja so skorajda neomejene. Možna je naprimer združitev (join) dveh tabel, četudi se ena nahaja v pomnilniku, druga pa v podatkovni bazi – pri tem razvijalec tabeli vidi, kot bi obe prihajali iz istega vira.
Za pomoč pri učenju tehnologije LINQ je na voljo aplikacija LINQPad, ki se nahaja na naslovu http://www.linqpad.net/.
Spodnji primer prikazuje, kako se s pomočjo tehnologije LINQ iz tabele *Purchases* prebere 3 vrstice, potem ko se prvih 5 preskoči. Vrstni red je postavljen glede na ceno (*Price*).

```
(from p in Purchases
orderby p.Price
select p).Skip(5).Take(3);
```

Zgornji LINQ izraz se prevede v lambda izraz

```
Purchases
   .OrderBy (p => p.Price)
   .Skip (5)
   .Take (3)
```

Lambda izraz pa se na koncu prevede v SQL stavek (za Microsoft SQL Server 2005):

```
SELECT [t1].[ID],
       [t1].[CustomerID],
       [t1].[Date],
       [t1].[Description],
       [t1].[Price]
FROM (
    SELECT ROW_NUMBER()
           OVER ( ORDER BY [t0].[Price] ) AS [ROW_NUMBER],
           [t0].[ID],
           [t0].[CustomerID],
           [t0].[Date],
           [t0].[Description],
           [t0].[Price]
    FROM [Purchase] AS [t0]
    ) AS [t1]
WHERE [t1].[ROW_NUMBER] BETWEEN @p0 + 1 AND @p0 + @p1
ORDER BY [t1].[ROW_NUMBER]
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [5]
-- @p1: Input Int (Size = 0; Prec = 0; Scale = 0) [3]
```

Primer zelo lepo pokaže, da lahko z uporabo LINQ-ja izvorna koda postane bistveno bolj čitljiva in odporna na napake, saj pravilnost izrazov LINQ preverja prevajalnik. Če se tabela *Purchases* nahaja v neki drugi podatkovni bazi, je za dostop do nje treba sestaviti le drugačen stavek SQL v ponudniku, če pa je tabela *Purchases* zapisana v datoteki XML, se izraz LINQ pretvori v stavek XPath. Veliko prednost pri tej tehnologiji predstavlja prav dejstvo, da mora

za dostop do podatkov kateregakoli vira razvijalec poznati le sintakso jezika LINQ, ne potrebuje pa znanja o sintaksi jezika SQL, Xpath ali katerihkoli drugih jezikov za dostop do podatkov.

3.5. Windows Comunication Foundation

Windows Communication Foundation (WCF) je programsko ogrodje za gradnjo komunikacijskega dela N-nivojskih sistemov. WCF je bil dodan ogrodju .NET verzije 3.0.
V prejšnih verzijah je bilo na razpolago več različnih komunikacijskih programskih modelov (Web Services, .NET Remoting, Distributed Transactions, Message Queues ...), ki pa jih je WCF združil v enega samega. Takšna celovitost je tudi ena izmed največjih prednosti, ki jih prinaša uporaba ogrodja WCF. Pred tem je bilo za spletne storitve, ki bazirajo na protokolu HTTP, potrebno napisati popolnoma drugačno izvorno kodo kot za .NET Remoting, ki za komunikacijske potrebe uporablja TCP/IP protokol. Dandanes je mogoče za obe tehnologiji uporabljati enako izvorno kodo – za pravilno delovanje ene in druge je potrebno prilagoditi le konfiguracijsko datoteko (poglavje 3.5.3).

3.5.1. Definicija podatkovnih in operacijskih dogovorov

Razrede in razredne metode, ki jih želimo ponuditi odjemalcu (predstavitvenem nivoju), moramo nekako označiti. Najlažje je to prikazati na enostavnem primeru.
Naredili bomo komponento za delo z osebami. V Visual Studiu naredimo nov projekt – knjižnico razredov (Class Library). Izhodna datoteka projekta te vrste je dinamična knjižnica (datoteka s končnico .dll).
V projekt bomo dodali dva objekta za prenos podatkov (*DtoOseba* in *DtoNaslov*), dva vmesnika (*IOseba* in *INaslov*) in dva razreda (*Oseba* in *Naslov*). Objekta za prenos podatkov bomo označili z atributom, ki predstavlja podatkovni dogovor, vmesnika z atributom, ki predstavlja operacijski dogovor, razreda pa bosta implementirala vsak svoj vmesnik. Načeloma ta postopek predstavlja vse, kar mora razvijalec vedeti, da svoj projekt pripravi za komunikacijo preko omrežnih protokolov.
Datoteka DtoNaslov.cs:

```
using System;
using System.Runtime.Serialization;

namespace WCFClassLib
{
    [DataContract]
    public class DtoNaslov
    {
        [DataMember]
        public int Id { get; set; }
        [DataMember]
        public string Ulica { get; set; }
        [DataMember]
        public int HisnaStevilka { get; set; }
        [DataMember]
        public string Kraj { get; set; }
    }
}
```

Datoteka DtoOseba.cs:

```
using System;
using System.Runtime.Serialization;

namespace WCFClassLib
{
    [DataContract]
    public class DtoOseba
    {
        [DataMember]
        public string EMSO { get; set; }
        [DataMember]
        public string Ime { get; set; }
        [DataMember]
        public string Priimek { get; set; }
        [DataMember]
        public DateTime DatumRojstva { get; set; }
        [DataMember]
        public DtoNaslov Naslov { get; set; }
    }
}
```

V gornjih datotekah se lepo vidi uporaba atributov *DataContract* in *DataMember*. Z atributom *DataContract* označimo razrede, ki jih bomo uporabili za prenos podatkov, z atributom *DataMember* pa povemo, katere lastnosti razreda želimo serializirati. Serializacijo izvede razred *DataContractSerializer*, in sicer le za tiste lastnosti, ki so označene z atributom *DataMember*. Razred *XmlSerializer*, ki se je uporabljal v prejšnjih verzijah ogrodja .NET, je s privzetimi nastavitvami serializiral vse javne lastnosti razreda, kar pa dostikrat ni zaželeno.
V *DtoOseba* smo definirali lastnost *Naslov*, ki je tipa *DtoNaslov*. Razred *DtoNaslov* je označen za serializacijo, kar pomeni da je za serializacijo možno označiti tudi lastnost *Naslov*. *DataContractSerializer* privzeto serializira osnovne tipe, vse ostale tipe je za serializacijo potrebno eksplicitno označiti z uporabo atributov *DataContract* in *DataMember*. Serializacija je mogoča le, če po razredni hierarhiji (v smeri od zgoraj navzdol) lahko pridemo do posameznih osnovnih tipov.
Datoteka INaslov.cs:

```
using System.Collections.Generic;
using System.ServiceModel;

namespace WCFClassLib
{
    [ServiceContract]
    public interface INaslov
    {
        [OperationContract]
        List<DtoNaslov> GetNaslovByKraj(string kraj);
    }
}
```

Datoteka IOseba.cs:

```
using System.Collections.Generic;
using System.ServiceModel;

namespace WCFClassLib
{
    [ServiceContract]
    public interface IOseba
    {
        [OperationContract]
        List<DtoOseba> GetVseOsebe();
        [OperationContract]
        DtoOseba GetOsebaByEmso(string emso);
        [OperationContract]
        void AddOseba(DtoOseba oseba);
        [OperationContract]
        void EditOseba(DtoOseba oseba, DtoOseba old_oseba);
        [OperationContract]
        void DeleteOseba(DtoOseba oseba);
    }
}
```

V obeh vmesnikih iz zgornjih primerov smo uporabili atribute tipov *ServiceContract* in *OperationContract*. Z atributom *ServiceContract* označimo vmesnik, ki ga želimo objaviti preko komunikacijskega protokola, z atributom *OperationContract* pa označimo, katere metode vmesnika bomo ponudili odjemalcu. Vmesnik je definicija metod in lastnosti, ki jih mora implementirati vsak iz tega vmesnika izpeljani razred.
Poglejmo si implementacijo razreda Naslov:

```
using System.Collections.Generic;
using System.Linq;

namespace WCFClassLib
{
    public class Naslov : INaslov
    {
        private List<DtoNaslov> data =
            new List<DtoNaslov>
            {
                new DtoNaslov
                {
                    Kraj = "Kranj",
                    Ulica = "Jaka Platiše",
                    HisnaStevilka = 5
                },
                new DtoNaslov
                {
                    Kraj = "Kranj",
                    Ulica = "Juleta Gabrovška",
                    HisnaStevilka = 13
                },
```

```
                new DtoNaslov
                {
                    Kraj = "Ljubljana",
                    Ulica = "Letališka cesta",
                    HisnaStevilka = 1
                }
            };

        public List<DtoNaslov> GetNaslovByKraj(string kraj)
        {
            return (from o in data
                     where o.Kraj == kraj
                     select o).ToList();
        }
    }
}
```

V zgornjem primeru smo sicer izdelali seznam treh naslovov, vendar pa se je potrebno zavedati, da bomo kasneje vse naslove shranjevali v podatkovno bazo.
Razred Oseba je bolj zanimiv.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace WCFClassLib
{
    public class Oseba : IOseba
    {
        private List<DtoOseba> data =
            new List<DtoOseba>
            {
                new DtoOseba
                {
                    EMSO = "1312978500102",
                    Ime = "Marko",
                    Priimek = "Podgoršek",
                    DatumRojstva = new DateTime(1978, 12, 13),
                    Naslov =
                        new DtoNaslov
                        {
                            Kraj = "Kranj",
                            Ulica = "Jaka Platiše",
                            HisnaStevilka = 5
                        }
                },
                //Ostale osebe pobrisane...
                new DtoOseba
                {
                    EMSO = "0111997500152",
                    Ime = "Tadej",
```

```
                    Priimek = "Kovač",
                    DatumRojstva = new DateTime(1997, 11, 1),
                    Naslov =
                        new DtoNaslov
                        {
                            Kraj = "Ljubljana",
                            Ulica = "Letališka cesta",
                            HisnaStevilka = 1
                        }
                }
            };

        public List<DtoOseba> GetVseOsebe()
        {
            return data;
        }

        public DtoOseba GetOsebaByEmso(string emso)
        {
            var rv = from o in data
                     where o.EMSO == emso
                     select o;

            return rv.SingleOrDefault();
        }

        public void AddOseba(DtoOseba oseba)
        {
            data.Add(oseba);
        }

        public void EditOseba(DtoOseba oseba,
                              DtoOseba old_oseba)
        {
            var osebaForEdit = (from o in data
                                where o.EMSO == oseba.EMSO
                                select o).Single();

            osebaForEdit.Ime = oseba.Ime;
            osebaForEdit.Priimek = oseba.Priimek;
            osebaForEdit.DatumRojstva = oseba.DatumRojstva;
        }

        public void DeleteOseba(DtoOseba oseba)
        {
            data.Remove(oseba);
        }
    }
```

Zgornji razred je primer implementacije CRUD operacij, ki jih bomo uporabili v primerih poglavja 3.7.2. V poglavju 3.8.1 bomo isti razred uporabili za branje in ažuriranje podatkov v

podatkovni bazi. Podrobnosti o metodi *EditOseba*, ki vsebuje 2 parametra, so navedene v poglavju 3.8.3.

3.5.2. Sporočanje napak

Mehanizme za ustvarjanje in prestrezanje napak mora vsebovati vsaka bolj zahtevna aplikacija. Obvladovanje napak v okviru posameznih nivojev samo po sebi ni problematično, saj večina programskih jezikov vsebuje programske konstrukte, ki nam pri delu z napakami pomagajo. Večji problem predstavlja napaka, ki se zgodi na enem, prestreže pa na drugem nivoju. V WCF imamo v ta namen na voljo atribut *FaultContract*.
Obogatimo metodo v vmesniku *INaslov* s tem atributom:

```
[OperationContract]
[FaultContract(typeof(string))]
List<DtoNaslov> GetNaslovByKraj(string kraj);
```

Pri uporabi atributa *FaultContract* je potrebno vnaprej napovedati, kakšnega tipa bo naša napaka. Tip, ki ga definiramo, mora biti serializiran. Tip *string* je osnovni tip, zato je njegova serializacija možna.
V razredu *Naslov* spremenimo vsebino funkcije na naslednji način:

```
public List<DtoNaslov> GetNaslovByKraj(string kraj)
{
    var rv = (from o in data
              where o.Kraj == kraj
              select o).ToList();

    if (rv.Count == 0)
        throw new FaultException("Ni naslovov v tem kraju");

    return rv;
}
```

Zgornja programska koda vrne napako, če ne najde nobenega naslova za določen kraj. To napako lahko odjemalec prestreže, kot bi se zgodila lokalno (na istem nivoju – poglavje 3.7.2).
Na enak način, na kakršnega smo z atributom *FaultContract* označili metodo *GetNaslovByKraj* (zgornji primer), označimo še metode *AddOseba*, *EditOseba* in *DeleteOseba* v vmesniku *IOseba*.

3.5.3. Konfiguracijska datoteka

Posledica razbremenitve razvijalca s tehnologijo WCF je obremenitev sistemskega administratorja pri namestitvi programskih komponent in izdelavi konfiguracijske datoteke. Pisanje te datoteke spada v težji in bolj zapleten del tehnologije WCF.
WCF Service Configuration Editor, aplikacija, priložena razvijalskemu paketu Visual Studio 2008, olajša izdelavo konfiguracijskih datotek. Ponuja ogromno možnosti.
V spodnjem primeru je naveden primer konfiguracijske datoteke, ki objavi komponento iz primera v prejšnjem poglavju (poglavje 3.5.2) preko strežnika IIS na HTTP protokol. Uporabljena je tudi Windows avtentikacija.

To konfiguracijo smo izbrali zato, ker se je veliko uporabljala v sistemih, izdelanih s predhodnimi verzijami ogrodja .NET, skupaj s klasičnimi spletnimi storitvami. Ker bomo našo komponento objavili na tak način, jo bodo lahko uporabljali tudi starejši odjemalci.
Če želimo našo komponento nato objaviti še preko protokola TCP, nam ni treba ničesar na novo programirati. Spremeniti je potrebno le konfiguracijsko datoteko. Lahko se odločimo tudi, da komponento z isto konfiguracijsko datoteko objavimo na več različnih načinov (trenutno jih je na voljo okoli 20).
Primer konfiguracijske datoteke (web.config), ki upošteva WS-I Basic Profile 1.1 [10] in Windows avtentikacijo:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authentication mode="Windows"/>
    <authorization>
      <deny users="?"/>
    </authorization>
  </system.web>

  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="WCFClassLib.Behavior">
          <!--To avoid disclosing metadata information, set
              the value below to false before deployment-->
          <serviceMetadata httpGetEnabled="true" />
          <!--To receive exception details in faults for
              debugging purposes, set the value below to
              true. Set to false before deployment to avoid
              disclosing exception information-->
          <serviceDebug includeExceptionDetailInFaults="true"
          />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <bindings>
      <!--WCF basicHttpBinding with Windows Authentication-->
      <basicHttpBinding>
        <binding name="BindWinAuth">
          <security mode="TransportCredentialOnly">
            <transport clientCredentialType="Windows" />
          </security>
        </binding>
      </basicHttpBinding>
    </bindings>
    <services>
      <service behaviorConfiguration="WCFClassLib.Behavior"
               name="WCFClassLib.Oseba">
        <host>
          <baseAddresses>
            <!-- Change to the server address virtual
```

```
                directory -->
            <add baseAddress="http://localhost/Diploma/" />
          </baseAddresses>
        </host>
        <!-- Unless fully qualified, address is relative to
             base address supplied above -->
        <endpoint address=""
                  binding="basicHttpBinding"
                  bindingConfiguration="BindWinAuth"
                  contract="WCFClassLib.IOseba" />
      </service>
      <service behaviorConfiguration="WCFClassLib.Behavior"
               name="WCFClassLib.Naslov">
        <host>
          <baseAddresses>
            <!-- Change to the server address virtual
                 directory -->
            <add baseAddress="http://localhost/Diploma/" />
          </baseAddresses>
        </host>
        <!-- Unless fully qualified, address is relative to
             base address supplied above -->
        <endpoint address=""
                  binding="basicHttpBinding"
                  bindingConfiguration="BindWinAuth"
                  contract="WCFClassLib.INaslov" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Na vrhu datoteke, v elementu *system.web* za avtentikacijski mehanizem nastavimo Windows avtentikacijo (<*authentication mode="Windows"/*>) in onemogočimo dostop vsem anonimnežem (<*deny users="?"/*>). Ta del konfiguracije ni v povezavi z WCF.
Element *system.serviceModel* je konfiguracijski element, potreben za delovanje WCF.
Sledi definicija tega, kako naj se celotna storitev obnaša (element *behaviors/serviceBehaviors/behavior*). Element *behavior* poimenujemo za kasnejšo referenco (<*behavior name="WCFClassLib.Behavior"*>). Ime je lahko poljubno. Storitvi nato omogočimo še, da svoje podatke (metadata) oglašuje preko protokola HTTP (<*serviceMetadata httpGetEnabled="true" /*>). V produkcijski verziji spletne stortve je to vrednost potrebno spremeniti na *"false"* (kar je omenjeno tudi v komentarju). Druga vrednost, ki jo potrebujemo samo v fazi razvoja, ne pa tudi v produkciji, in je zapisana v naslednjem elementu (<*serviceDebug includeExceptionDetailInFaults="true" /*>), nam olajša iskanje ter odpravljanje napak.
Pod elementom bindings definiramo *basicHttpBinding*. Ta upošteva zgoraj omenjeni WS-I Basic Profile 1.1, tako da ga tudi starejši odjemalci razpoznajo kot klasičo spletno storitev. Uporaba Windows avtentikacije je definirana v vrsticah, ki sledijo in sovpadajo z vrhnjim delom konfiguracijske datoteke (system.web). Elementu binding podamo tudi ime za kasnejšo referenco.
Sledi element *services* – element, ki definira, kaj naša spletna storitev ponuja. V našem primeru sta to dva razreda, ki smo ju definirali s pomočjo vmesnikov (*Oseba* in *IOseba*,

Naslov in *INaslov*). Elementu service določimo atribut *behaviorConfiguration*, ki pove, katero konfiguracijo „obnašanja“ naj spletna storitev uporabi. Le-to smo definirali nekaj vrstic pred elementom *service*. Atribut *name* mora obvezno vsebovati celotno ime razreda (z vsemi imenskimi prostori), ki ga želimo objaviti.
V element *baseAddresses* dodamo osnovni URL, ki kaže na našo storitev (pot do navideznega direktorija našega spletnega strežnika).
Element *endpoint*, je točka, na katero se odjemalci lahko povežejo preko komunikacijskih protokolov. Definiramo atribut *binding* na *basicHttpBinding* in z atributom *bindingConfiguration* navedemo ime zgoraj definirane avtentikacijske konfiguracije (za uporabo Windows avtentikacije). Atribut contract je zelo pomemben. Nastavimo ga na celotno ime vmesnika *IOseba* (<*endpoint contract="WCFClassLib.IOseba" />*), ki ga implementira razred *Oseba* (<*service name=" WCFClassLib.Oseba"*>).
Postopek nato ponovimo še za razred *Naslov* oz. njegovo konfiguracijo elementa *service*.
Zgornjo datoteko se lahko uporabi kot „recept“ za izdelavo drugih konfiguracijskih datotek, ki morajo upoštevati Windows avtentikacijo in objavo storitve preko spletnega strežnika IIS.

3.5.4. Objava na spletnem strežniku IIS

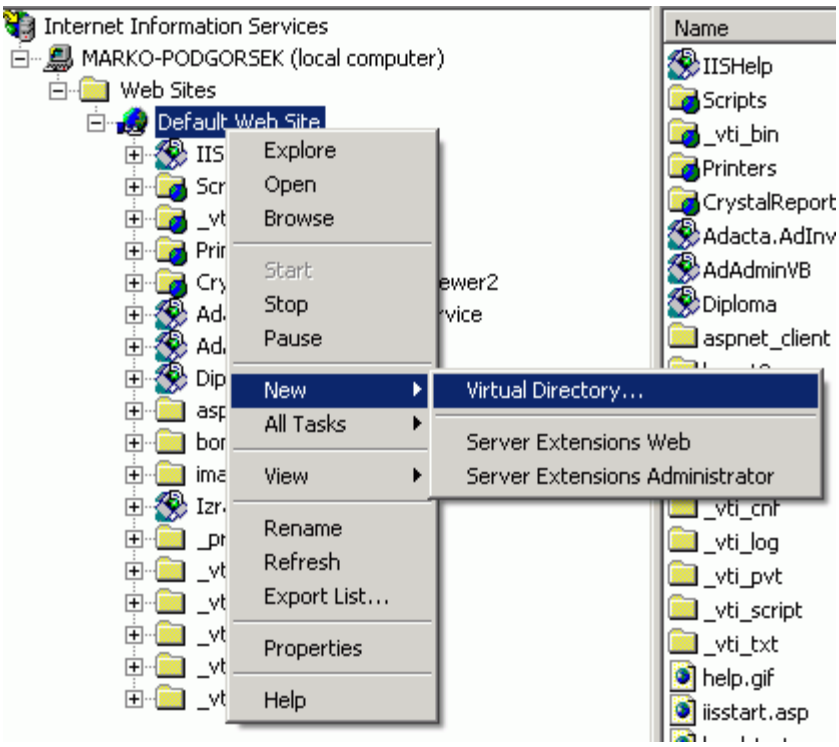
Postopek objave storitve na spletnem strežniku začnemo tako, da v Visual Studiu 2008 ustvarimo nov projekt (File/New/Web Site...) in za vrsto projekta izberemo WCF Service. Iz projekta pobrišemo mapi App_Code in App_Data, datoteko web.config pa prepišemo z novimi nastavitvami (poglavje 3.5.3). Naredimo tudi kopijo datoteke Service.svc. Prvo preimenujemo v Oseba.svc in jo opremimo z vsebino:

```
<%@ ServiceHost Language="C#"
                Debug="true"
                Service="WCFClassLib.Oseba" %>
```

Drugo preimenujemo v Naslov.svc in vanjo vpišemo:

```
<%@ ServiceHost Language="C#"
                Debug="true"
                Service="WCFClassLib.Naslov" %>
```

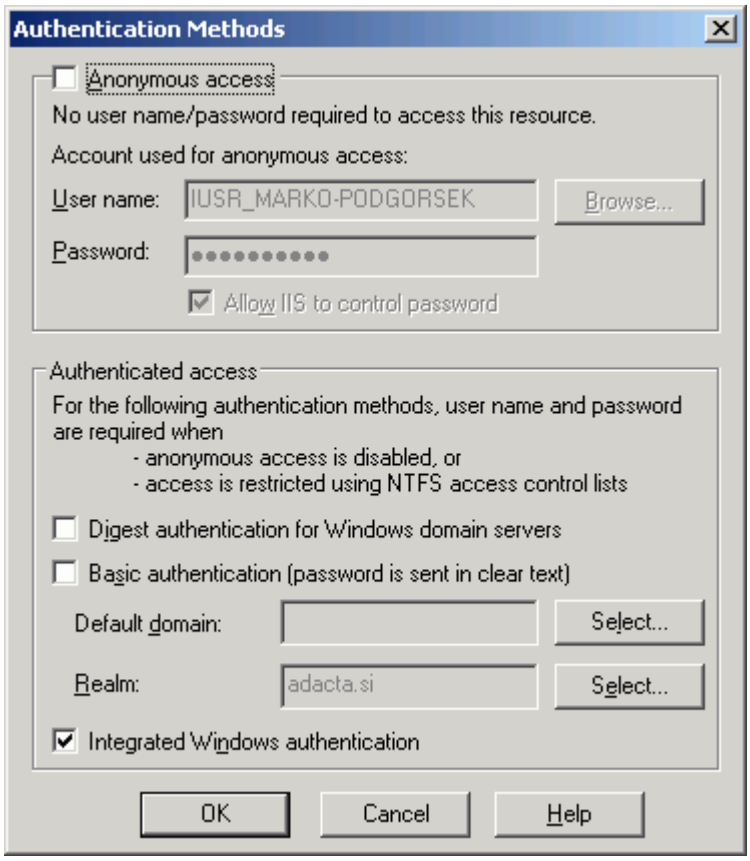
Vrednost atributa *Service* je ime, ki smo ga definirali v konfiguracijski datoteki iz prejšnjega poglavja (<*service name="WCFClassLib.Naslov"*>). Vrednost atributa *Debug* v produkcijski verziji seveda spremenimo na *"false"*.
Novonastalemu projektu pod reference dodamo izhodno datoteko projekta, ki smo ga izdelali v poglavju 3.5.1 WCFClassLib.dll (v kontekstnem meniju projekta izberemo Add Reference in v pogovornem oknu, ki se odpre, na zavihku Browse poiščemo želeno datoteko).
V konfiguracijskem dialogu spletnega strežnika IIS naredimo nov navidezni direktorij (slika 2).



Slika 2. Ustvarjanje novega navideznega direktorija

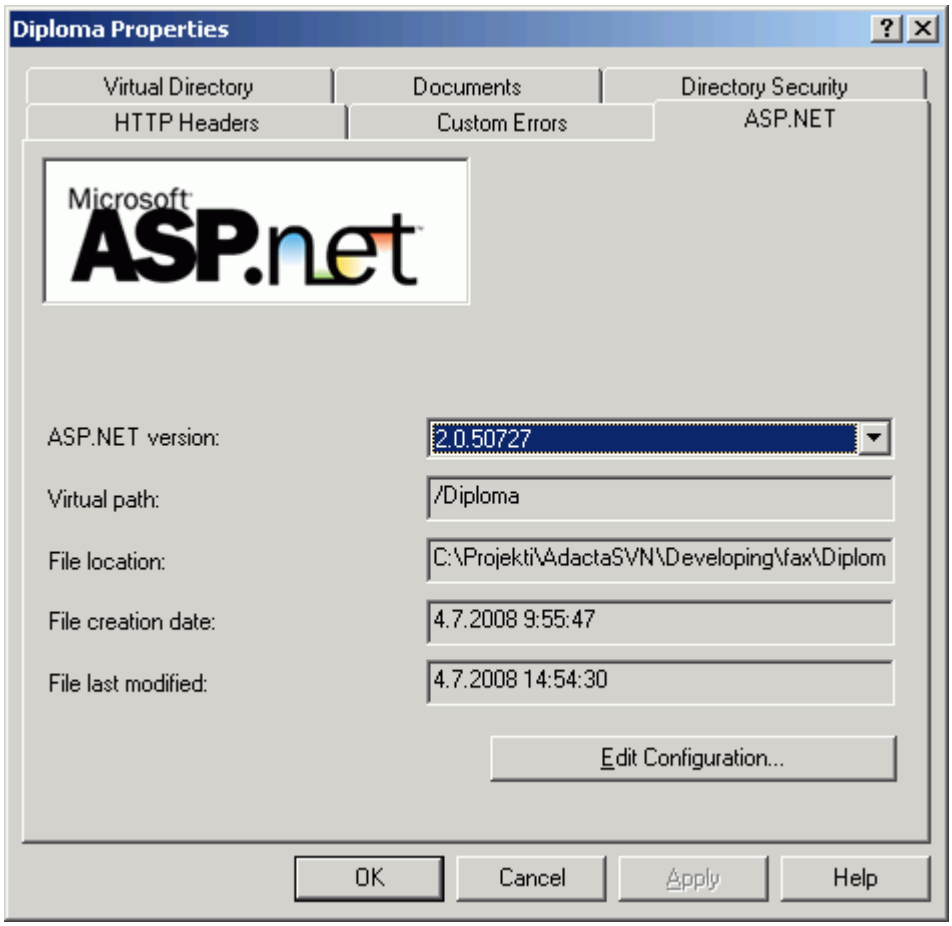
V čarovniku, ki se prikaže, pod Alias vpišemo želeno ime direktorija. V zgornjem primeru konfiguracijske datoteke smo v ta namen uporabil ime Diploma, kar pomeni, da je direktorij preko HTTP protokola dostopen na URL http://localhost/Diploma. Na naslednji strani izberemo pot do mape, kjer se nahaja naš spletni projekt. Na zadnji strani pustimo privzeti prvi dve kljukici v spisku in končamo.

Nato iz kontekstnega menija novonastalega virtualnega direktorija izberemo opcijo Properties, v pogovornem oknu, ki se odpre na zavihku Directory Security, kliknemo na prvi gumb Edit... Odpre se pogovorno okno, prikazano na sliki 3.



Slika 3. Pogovorno okno za nastavitve varnosti

Onemogočiti moramo anonimni dostop in omogočiti Integrated Windows authentication, nato izbiro potrdimo.
Vrnemo se v prejšnje pogovorno okno, kjer izberemo zavihek ASP.NET (slika 4) in na njem verzijo ASP.NET tehnologije, ki jo uporabljamo - 2.0.50727 (možno je, da se zadnja številka nekoliko razlikuje od tiste na sliki).



Slika 4. Zavihek ASP.NET prikazuje pravilne nastavitve

Z nastavitvami spletnega strežnika smo končali. Nastavljene spremembe je pametno preizkusiti, kar najhitreje naredimo tako, da v enega od spletnih brskalnikov vpišemo URL service-a http://localhost/Diploma/Oseba.svc (ali Naslov.svc). Najbolje je, da pri tem uporabimo Microsoft Internet Explorer, ki za razliko od npr. Mozilla Firefox-a, uporabniško ime in geslo, s katerima smo prijavljeni v operacijski sistem (Windows), v HTTP zahtevku avtomatsko pošlje na ciljni URL.

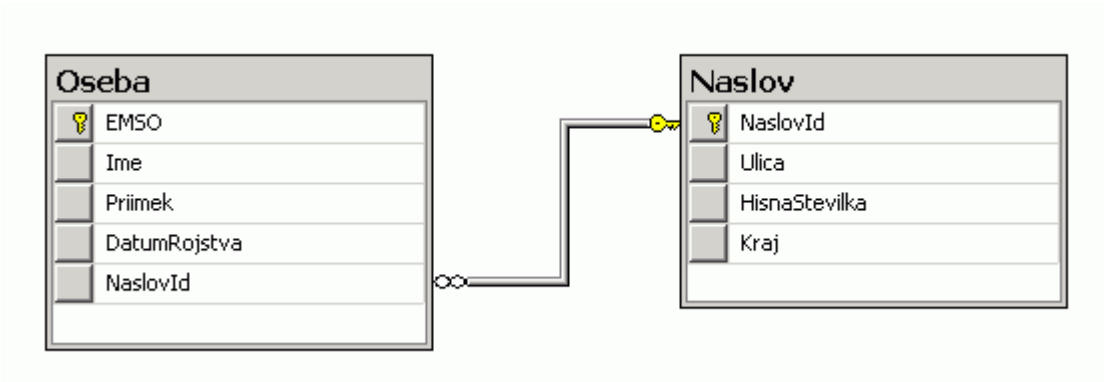
3.6. Implementacija podatkovnega nivoja

Pri implementaciji podatkovnega nivoja se je potrebno, kot je bilo že večkrat omenjeno, osredotočiti izključno na izdelavo tabel ter njihovih primarnih in tujih ključev (tj. relacij med tabelami).

Pri oblikovanju tabel pazimo, da imamo podatkovno bazo normalizirano [11]. Baznih procedur ne implementiramo. Prav tako je priporočljivo, da se v večini primerov izognemo implementaciji pogledov na tabele. Bazne procedure in pogledi na tabele so odvisni od podatkovne baze. Če se odločimo za menjavo obstoječe podatkovne baze s podatkovno bazo drugega ponudnika, moramo vse poglede in procedure prepisati iz ene podatkovne baze v drugo, pri čemer je potrebno računati na to, da si bazi po vsej verjetnosti sintaktično ne bosta najbolj podobni. Prepisovanje lahko v nekaterih primerih predstavlja ogromen zalogaj za administratorje podatkovnih baz in velik časovni zamik ob menjavi.

Poglede na tabele je priporočljivo uporabljati le za zelo zahtevne poizvedbe, seveda izključno takrat, kadar takšen pristop pomeni veliko performančno prednost. Podobno priporočilo velja tudi za bazne procedure (poglavje 2.3).

Podatkovni model našega primera je zelo enostaven. Prikazuje ga slika 5.



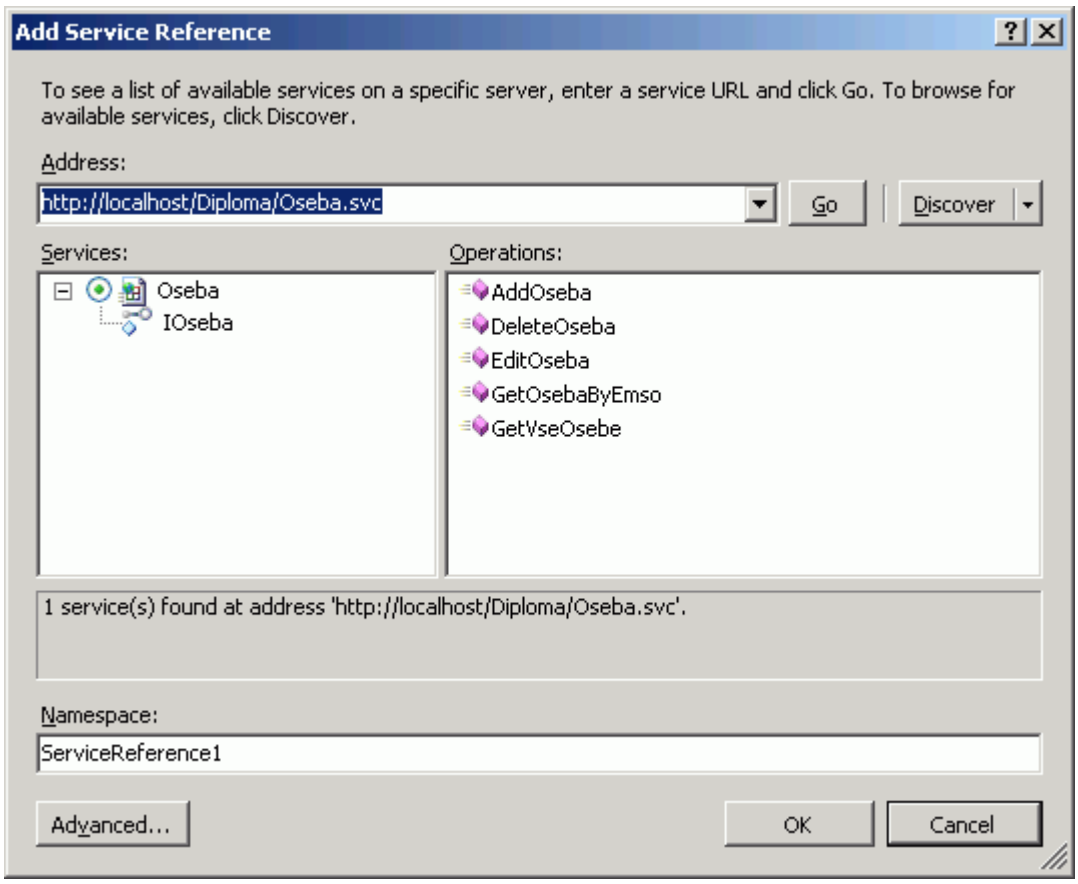
Slika 5. Podatkovni model za naš primer implementacije N-nivojskega sistema

3.7. Implementacija predstavitvenega nivoja

Implementacije predstavitvenega nivoja se lotimo šele po tem, ko izdelamo podatkovni nivo, o čemer smo govorili že v poglavju 3.1.1.
V našem primeru predstavitveni nivo predstavlja ASP.NET spletna aplikacija. Pri programiranju uporabniškega vmesnika se osredotočimo na to, da ga v čim večji meri izdelamo opisno – z uporabo ASP.NET kontrol, in se, kolikor le je to mogoče, izogibamo ročnemu pisanju kode.
Težiti moramo tudi k temu, da v predstavitveni nivo ne implementiramo poslovne logike, saj se celotna poslovna logika nahaja na našem strežniku in je dostopna preko tehnologije WCF (poglavje 3.5.4).
Začnemo tako, da v Visual Studiu 2008 ustvarimo novo spletno aplikacijo (File/New/Web Site...) in izberemo ASP.NET Web Site.

3.7.1. Povezava na WCF

Za komunikacijo s spletno storitvijo potrebujemo „proxy“ razrede. Generiramo jih lahko na dva načina.
Prvi način (slika 6) predvideva uporabo funkcionalnosti, vgrajene v Visual Studio 2008 (Add Service Reference).

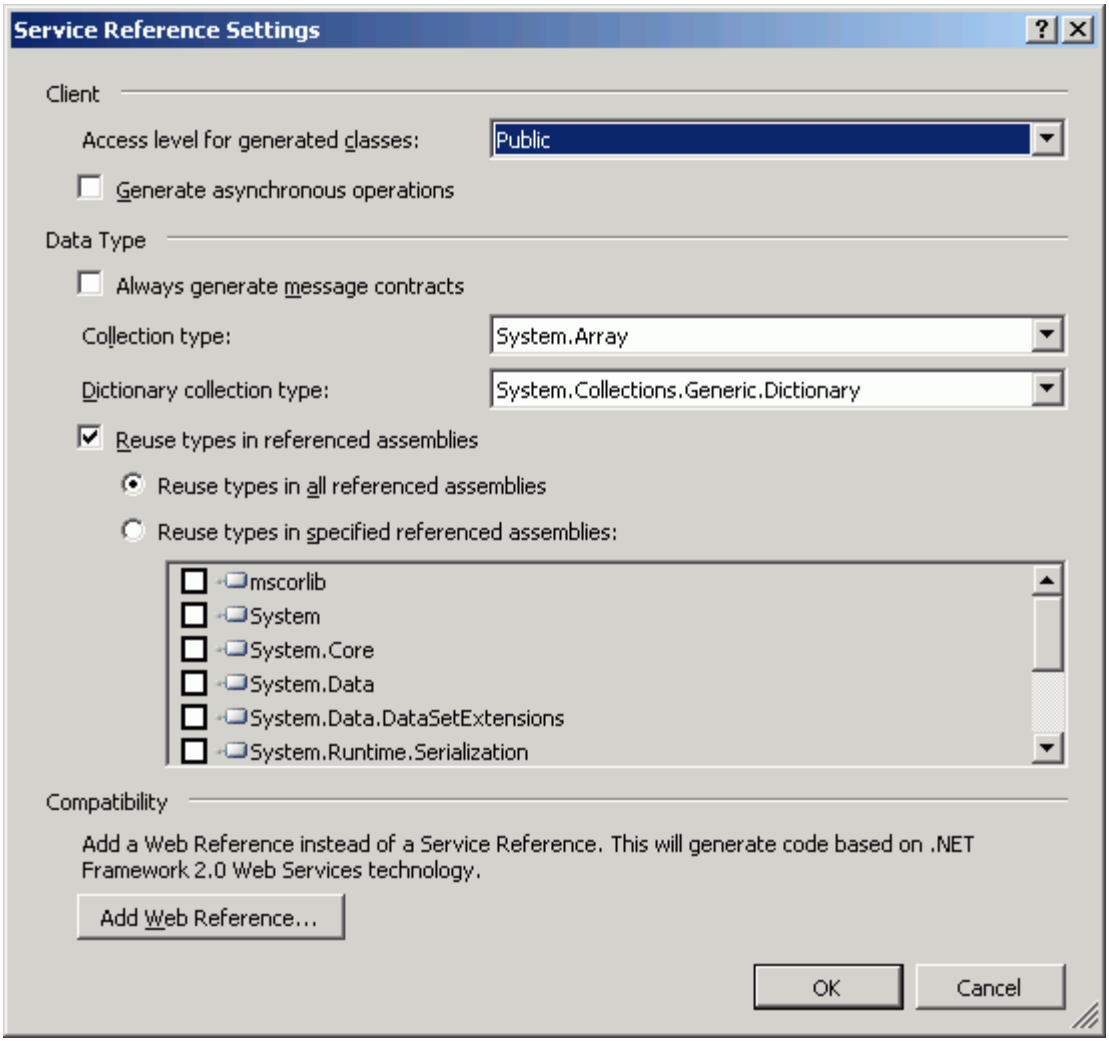


Slika 6. Pogovorno okno za generiranje „proxy“ razreda

V spodnjem delu lahko napišemo, v kateri imenski prostor želimo umestiti generirano kodo („proxy“ razred).

To pogovorno okno ima precejšnjo pomanjkljivost, saj lahko generira kodo le za eno spletno storitev naenkrat. Tak pristop lahko pripelje do nepotrebnih problemov. V poglavju 3.5.1 smo definirali dva operacijska dogovora, do katerih dostopamo preko .svc datotek, objavljenih na IIS strežniku (poglavje 3.5.4.). Operacijski dogovor *Oseba* uporablja oba razreda za prenos podatkov (*DtoOseba* in tudi *DtoNaslov*), zato se tudi definiciji obeh preneseta v izbrani imenski prostor. Če želimo dodati še referenco na operacijski dogovor *Naslov*, moramo izbrati imenski prostor, ki ni enak prvemu. To pa nas privede do problema. Imamo namreč še eno definicijo razreda za prenos podatkov – *DtoNaslov*, ki se nahaja v drugem imenskem prostoru. Prevajalnik ta dva razreda napačno zazna kot različna, čeprav to nista, kar oteži „podajanje“ enega razreda iz enega operacijskega dogovora v drugega.

Omenjenemu problemu se lahko izognemo, če imamo vse razrede za prenos podatkov v eni programski komponenti (tj. v eni knjižnici .dll). Referenco na to knjižnico dodamo v projekt WCF in nato še v projekt predstavitvenega nivoja. S klikom na gumb Advanced... (slika 6) prikličemo pogovorno okno naprednih nastavitev za generiranje „proxy“ razredov, prikazano na sliki 7.

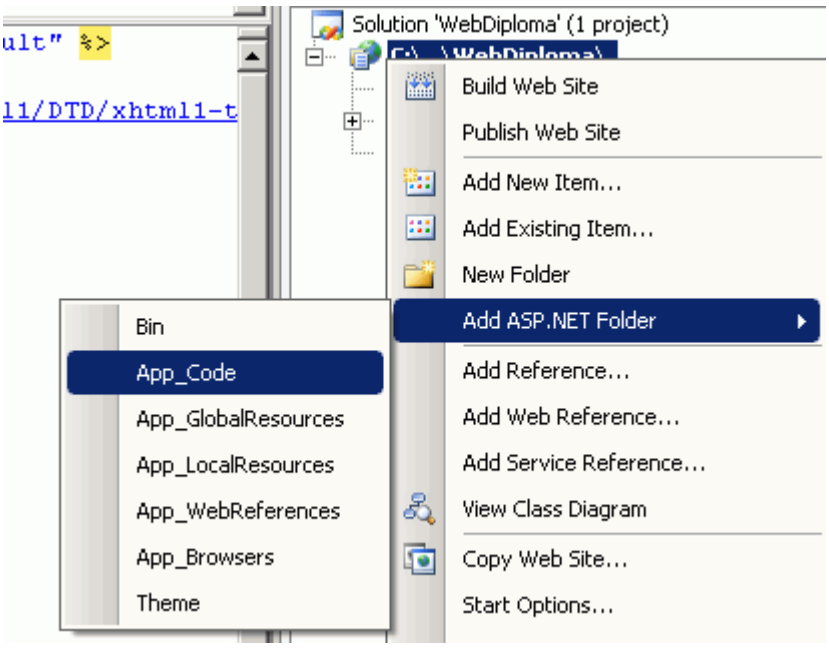


Slika 7. Pogovorno okno naprednih nastavitev za generiranje „proxy“ razreda

V tem pogovornem oknu je za nas najbolj zanimiva opcija „Reuse types in referenced assemblies“. Označena opcija sporoči generatorju, naj upošteva že referencirane programske komponente in za vsak nov tip, ki ga uporablja spletna storitev, preveri, če ni bil isti tip definiran že v kaki od prejšnjih komponent – v takšnem primeru novega tipa ne uvozi, temveč raje uporabi že definiranega.
Drugi način je uporaba aplikacije svcutil.exe v ukazni vrstici. Ta nam omogoča hkratno referenco na vse spletne storitve, ki jih potrebujemo. Končni rezultat je ena sama datoteka z imenskim prostorom, ki vsebuje več „proxy“ razredov in vse definicije razredov za prenos podatkov.
V našem primeru je ukaz, ki ga moramo izvesti:

```
svcutil.exe /out:DiplomaWS.cs
            /targetClientVersion:Version35
            http://localhost/Diploma/Naslov.svc
            http://localhost/Diploma/Oseba.svc
```

Rezultat ukaza je datoteka DiplomaWS.cs, ki jo dodamo v naš projekt predstavitvenega nivoja. To naredimo tako, da najprej izdelamo mapo v projektu, v katero bomo našo programsko kodo postavili. V kontekstnem meniju projekta izberemo „Add ASP.NET Folder“ in nato „App_Code“ (slika 8).

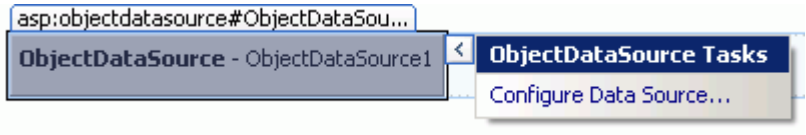


Slika 8. Dodajanje mape App_Code

Izvorna koda, ki se nahaja v tej mapi (npr. naša datoteka DiplomaWS.cs), se bo avtomatsko prevedla v izvajalno kodo ob prvem obisku naše spletne strani in tudi ob vsaki njeni spremembi.
Pri generiranju „proxy“ razredov dobimo še datoteko output.config, katere vsebino je potrebno prekopirati v že obstoječo konfiguracijsko datoteko našega projekta. Konkretno to naredimo tako, da v element *configuration* datoteke web.config prekopiramo celoten element *system.serviceModel* iz datoteke output.config.
V slednji datoteki so nastavitve za odjemalca, ki se ujemajo z nastavitami naše spletne storitve.

3.7.2. Kontrola ObjectDataSource

ObjectDataSource je podatkovna kontrola za ASP.NET strani, ki ponuja dostop do podatkov preko metod razredov, za razliko od podatkovne kontrole *SqlDataSource*, ki deluje direktno nad podatkovno bazo. Razredi, ki jih bomo uporabili, so bili zgenerirani v poglavju 3.7.1.
Trenutno imamo v projektu datoteko Default.aspx. To datoteko odpremo v načinu za oblikovanje (design) ter vanjo iz dialoga, ki je dostopen preko menija View/Toolbox potegnemo kontrolo *ObjectDataSource*, ki se nahaja v razdelku Data. Ko je ta kontrola na strani, lahko kliknemo puščico desno ob njej in izberemo Configure Data Source... (slika 9).



Slika 9. Kontrola *ObjectDataSource* s prikazanim konfiguracijskim menijem

Na prvi strani čarovnika, ki se prikaže, izberemo razred *OsebaClient*. Ta implementira operacije CRUD za objekte tipa *DtoOseba*. Nadaljujemo z izbiro teh operacij:

- za SELECT izberemo metodo *GetVseOsebe*,
- za UPDATE izberemo metodo *EditOseba*,
- za INSERT izberemo metodo *AddOseba* in
- za DELETE izberemo metodo *DeleteOseba*.

Po potrditvi nam čarovnik zgenerira kodo za *ObjectDataSource*. Vsebino slednje lahko preverimo s klikom na gumb „Source“.

```
<asp:ObjectDataSource ID="ObjectDataSource1"
                      runat="server"
                      TypeName="OsebaClient"
                      DataObjectTypeName="WCFClassLib.DtoOseba"
                      DeleteMethod="DeleteOseba"
                      InsertMethod="AddOseba"
                      SelectMethod="GetVseOsebe"
                      UpdateMethod="EditOseba">
</asp:ObjectDataSource>
```

Atribut *TypeName* pove, kateri razred uporabljamo za dostop do podatkov, *DataObjectTypeName* pa pove, katerega uporabljamo za prenos podatkov. Ostali atributi definirajo metode operacij CRUD.
Da bi kasneje lahko preverjali, ali je prišlo do hkratnega ažuriranja zapisov (poglavje 3.8.3), dodamo dva atributa:

```
OldValuesParameterFormatString="old_{0}"
ConflictDetection="CompareAllValues"
```

Dodamo še funkcionalnost sporočanja informacij uporabniku po končanih operacijah dodajanja, brisanja in urejanja, tako da kontroli *ObjectDataSource* dodamo tri atribute

```
OnDeleted="ObjectDataSource1_AfterEvent"
OnInserted="ObjectDataSource1_AfterEvent"
OnUpdated="ObjectDataSource1_AfterEvent"
```

in v datoteko Default.aspx.cs dopišemo

```
protected void ObjectDataSource1_AfterEvent(object sender,
                            ObjectDataSourceStatusEventArgs e)
{
    if (e.Exception != null)
    {
        InfoText.Text = e.Exception.InnerException.Message;
        InfoText.ForeColor = System.Drawing.Color.Red;
        e.ExceptionHandled = true;
    }
    else
    {
        InfoText.Text = "Transakcija uspešna";
        InfoText.ForeColor = System.Drawing.Color.Green;
    }
}
```

ter v datoteko Default.aspx dodamo

```
<asp:Label ID="InfoText" runat="server"></asp:Label>
```

Zaradi manjše malomarnosti z regijskimi nastavitvami, ki so si jo privoščili pri Microsoftu, je za to, da dosežemo želeno funkcionalnost naše kontrole, potrebnega malce več dela. Način

zapisovanja datuma, števil s plavajočo vejico in valut je v Sloveniji drugačen kot v Ameriki, česar kontrola *ObjectDataSource* ne upošteva. Da bi se obnašala pravilno, jo je v to potrebno prisiliti.
Vse tipe, ki uporabljajo plavajočo vejico (razlika med Slovenijo in Ameriko je, da sta pika in vejca zamenjani) in datume, je potrebno dodatno definirati v elementu *asp:ObjectDataSource* [12]:

```
<UpdateParameters>
    <asp:Parameter Name="DatumRojstva" Type="DateTime" />
</UpdateParameters>
```

Pomembno je, da nastavimo atributa *Name* in *Type*, kjer je *Name* ime, *Type* pa tip lastnosti razreda.
Podobno velja za element *InsertParameters* in ostale, kjer je potreben pravilen prikaz podatkov, odvisnih od regionalnih nastavitev.
Problem z valuto (Microsoft še ni ugotovil, da Slovenija sedaj uporablja euro – € in ne več tolarja – SIT) najlaže rešimo tako, da konfiguraciji v datoteki web.config v element *system.web* dodamo sledeči zapis:

```
<globalization culture="de-DE" />
```

Primer zgoraj sicer izbere nemške regionalne nastavitve, vendar se te od slovenskih razlikujejo le v privzeti valuti. Za Nemčijo operacijski sistem Windows privzame euro, kar v našem primeru ne predstavlja problema, saj ga danes uporabljamo tudi sami.

3.7.3. Pogledi na podatke

Spletne strani, napisane z ogrodjem .NET, podpirajo opisno dvosmerno povezovanje (declarative two way databinding) podatkov z elementi maske uporabniškega vmesnika, kar pomeni, da lahko lastnosti razreda za prenos podatkov urejamo npr. v tekstovnih poljih, ne da bi v ta namen pisali dodatno kodo.
Primer enosmernega povezovanja:

```
<asp:TextBox ID="ImeTextBox"
             runat="server"
             Text='<%# Eval("Ime") %>' />
```

Enosmerno povezovanje podpira tudi gnezdene lastnosti (v našem primeru lastnost *Naslov*):

```
<asp:TextBox ID="NaslovUlicaTextBox"
             runat="server"
             Text='<%# Eval("Naslov.Ulica") %>' />
```

Problem se pojavi pri dvosmernem povezovanju, ki drugega primera ne podpira (dvosmerno povezovanje usposobimo, če namesto funkcije *Eval* uporabimo funkcijo *Bind*). Pri delu s podatki iz podatkovne baze na tak problem ne naletimo, saj stavek SELECT naredi pogled na podatke. Pogled skrije relacijsko strukturo baze in nam kot rezultat vrne samo tabelo pogleda brez hierarhije, ki se skriva za njo. Kadar delamo z razredi za prenos podatkov, moramo nanje gledati podobno kot na relacijsko strukturo v bazi [13].
V našem primeru to pomeni, da lastnost *Naslov* razreda *DtoOseba* pretvorimo v tri nove lastnosti. V mapi App_Code naredimo novo datoteko z imenom ClientViews.cs s sledečo vsebino:

```
namespace WCFClassLib
{
    public partial class DtoOseba
    {
        public string Ulica
        {
            get
            {
                if (Naslov != null)
                    return Naslov.Ulica;
                else
                    return null;
            }
            set
            {
                if (Naslov == null)
                    Naslov = new DtoNaslov();

                Naslov.Ulica = value;
            }
        }

        public int? HisnaStevilka
        {
            get
            {
                if (Naslov != null)
                    return Naslov.HisnaStevilka;
                else
                    return null;
            }
            set
            {
                if (Naslov == null)
                    Naslov = new DtoNaslov();

                if(value != null)
                    Naslov.HisnaStevilka = (int)value;
            }
        }

        public string Kraj
        {
            get
            {
                if (Naslov != null)
                    return Naslov.Kraj;
                else
                    return null;
            }
            set
```

```
            {
                if (Naslov == null)
                    Naslov = new DtoNaslov();

                Naslov.Kraj = value;
            }
        }
    }
}
```

Generirani razred *DtoOseba* je delni razred (poglavje 3.4.4), ki ga v datoteki ClientViews.cs dopolnimo tako, da ustvarimo tri nove lastnosti, ki podatke berejo iz lastnosti *Naslov* in jih vanjo zapisujejo. Dopolnjeni razred je pripravljen za dvosmerno povezovanje.

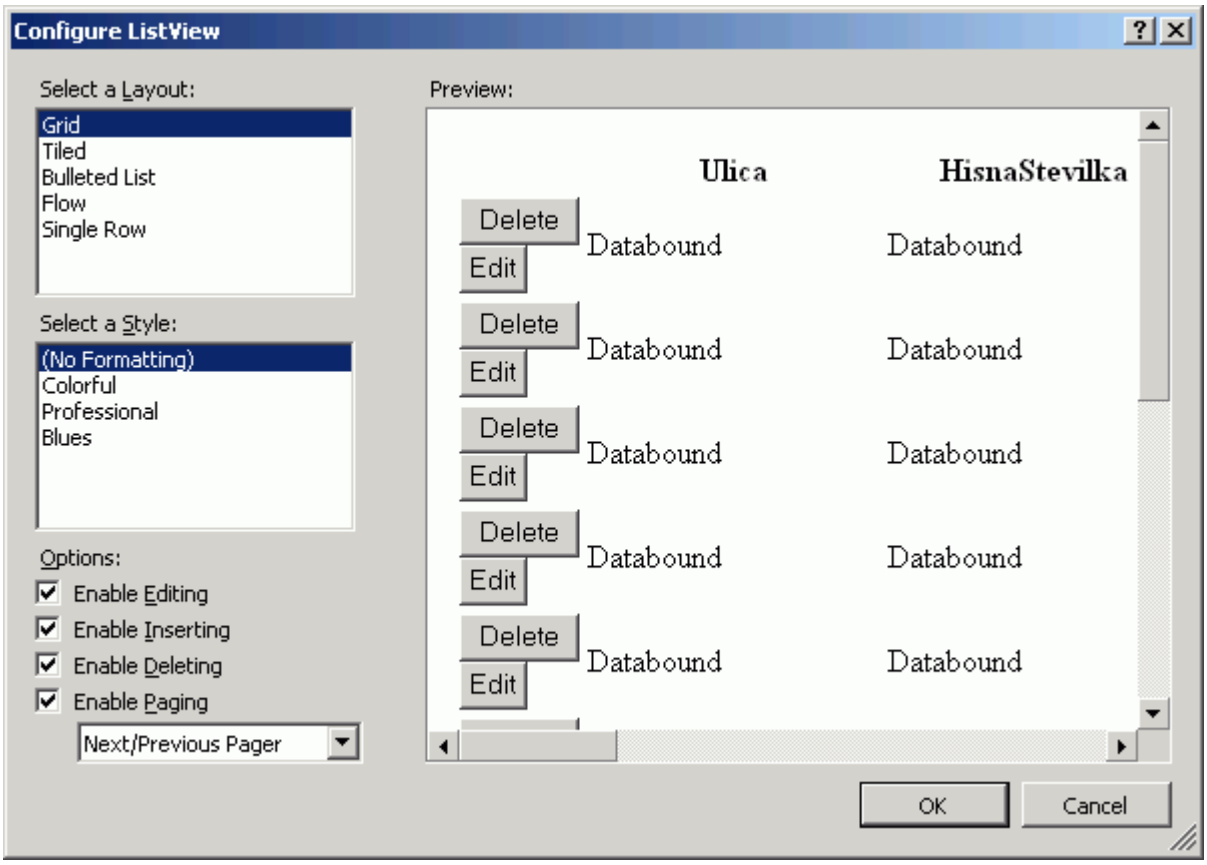
3.7.4. Kontrola ListView

Kontrola *ListView*, ki je bila predstavljena v ogrodju .NET verzije 3.5, je zelo fleksibilna, saj omogoča prikazovanje podatkov z možnostjo sortiranja, dodajana, brisanja in urejanja.
Pred ogrodjem .NET verzije 3.5 so razvijalci za prikaz podatkov lahko uporabili kontrole *GridView*, *DataList* in *Repeater*, za urejanje podatkov pa *GridView*, *DetailsView* in *FormView*. Lahko bi rekli, da je kontrola *GridView* najmočnejša od vseh, saj podpira tako prikazovanje kot urejanje. Kljub svoji moči ima določene pomanjkljivosti, saj ne podpira dodajanja in ima zelo omejene možnosti preoblikovanja. Vsi podatki v njej so prikazani tabelarično [14].
ListView združuje vse dobre lastnosti prej omenjenih kontrol. Omogoča nam oblikovanje s pomočjo predlog za vsak tip funkcionalnosti (prikaz, dodajanje, urejanje).
Na voljo imamo naslednje predloge:

- *EmptyDataTemplate*,
- *EmptyItemTemplate*,
- *ItemTemplate*,
- *AlternatingItemTemplate*,
- *ItemSeparatorTemplate*,
- *GroupTemplate*,
- *GroupSeparatorTemplate*,
- *InsertItemTemplate*,
- *EditItemTemplate*,
- *LayoutTemplate* in
- *SelectedItemTemplate*.

Prazna kontrola *ListView* ne zgenerira kode HTML, zato mora za celoten prikaz poskrbeti razvijalec. Pri ostalih kontrolah se HTML deloma vseeno zgenerira.
Najbolj uporabni predlogi za prikaz sta *LayoutTemplate* in *ItemTemplate*. Predloga *LayoutTemplate* definira HTML kodo, v katero se bodo izpisovali podatki. Lahko bi ji rekli tudi okvir prikaza. *ItemTemplate* definira, kako se bo prikazal posamezen zapis.
V predlogo *LayoutTemplate* moramo dodati element HTML, katerega lastnost *ID* nastavimo na *"itemPlaceholder"*. Vanj bodo izpisani podatki.
V predlogi *InsertTemplate* za prikaz podatkov uporabljamo funkcijo *Eval* (poglavje 3.7.3), pri *EditItemTemplate* in *InsertItemTemplate* pa funkcijo *Bind*.
Za lažje razumevanje kontrole *ListView* oz. njene uporabe je najbolje, da čarovniku pustimo, da nam zgenerira del kode, ki jo nato prilagodimo našim potrebam.
V datoteko Default.aspx podobno kot kontrolo *ObjectDataSource* (poglavje 3.7.2) dodamo kontrolo *ListView*. Za njen podatkovni vir izberemo prej kreirani *ObjectDataSource1* (z

opcijo Choose Data Source...). Datotetko shranimo in konfiguracijo kontrole še enkrat prikličemo s klikom na puščico desno od nje. Odpre se pogovorno okno, v katerem izberemo opcijo Configure ListView.
V novem pogovornem oknu izberemo, kar prikazuje slika 10.



Slika 10. Konfiguracijsko pogovorno okno kontrole *ListView*

Čarovnik nam zgenerira kodo, ki prikazuje podatke v tabelarični obliki in omogoča njihovo urejanje, brisanje in dodajanje novih zapisov. Zgenerirano je toliko stolpcev, kot je lastnosti v razredu *DtoOseba*. Lastnosti *Naslov* in *ExtensionData* sta odveč, zato popravimo kodo in ju odstranimo. Predlogi *AlternatingItemTemplate* in *SelectedItemTemplate* sta za naš primer ravno tako odveč, zato tudi njiju lahko pobrišemo.
Spremenimo tudi izpis datuma.

```
Bind("DatumRojstva", "{0:dd.MM.yyyy}")
Eval("DatumRojstva", "{0:dd.MM.yyyy}")
```

V predloge *InsertItemTemplate* in *EditItemTemplate* na vnosna polja dodamo še validatorje, s katerimi zagotovimo pravilnost vnosov.

```
<asp:CompareValidator ID="CompareValidator2"
                      runat="server"
                      ErrorMessage="Vpišite številko"
                      ControlToValidate="HisnaStevilkaTextBox"
                      Operator="DataTypeCheck"
                      Type="Integer">
</asp:CompareValidator>
<asp:CompareValidator ID="CompareValidator1"
```

```
                        runat="server"
                        ErrorMessage="Vpišite datum"
                        ControlToValidate="DatumRojstvaTextBox"
                        Operator="DataTypeCheck"
                        Type="Date">
</asp:CompareValidator>
```

Validatorje postavimo v okolico polj, ki jih bodo preverjali. Oba validatorja in oba gumba v predlogi označimo z atributom *ValidationGroup*, ki ga nastavimo na enako vrednost. Enak postopek ponovimo za drugo predlogo, vendar atributu *ValidationGroup* podamo drugo vrednost, tako da so validatorji za urejanje in dodajanje med seboj neodvisni.
Kontroli *ListView* dodamo atribut *DataKeyNames="EMSO"*, s katerim povemo, katero polje vsebuje primarni ključ naših podatkov.
V kolikor smo z izgledom strani zadovoljni, je to vse, kar je treba narediti na predstavitvenem nivoju. Podatke o osebah lahko s pomočjo te maske sedaj pregledujemo, urejamo, izdelujemo nove zapise in brišemo stare. Lahko rečemo, da imamo podprte vse operacije CRUD, kar nam omogoča popolno kontrolo nad podatki (poglavje 2.5.2.1.).

3.8. Implementacija poslovnega nivoja

Poslovni nivo smo začeli pisati že v poglavju (poglavje 3.5), potrebna je le še implementacija dostopa do podatkovne baze s pomočjo tehnologije LINQ to SQL, s katerim bomo omogočili transakcije pri ažuriranju podatkov in preverjali hkratna ažuriranja.
Implementirati želimo logiko, ki bo ob dodajanju nove osebe preverila, ali vnešeni naslov že obstaja. V kolikor obstaja, bo uporabljen že obstoječi *NaslovId*, sicer naredimo nov zapis naslova in uporabimo nov *NaslovId*. Ravno tako bo ob brisanju osebe preverila, ali na njenem naslovu prebiva tudi kaka druga oseba. Če takšna oseba ne obstaja, potem naslov pobrišemo. Pri urejanju že obstoječe osebe bo programska logika preverila, če ni zapisa spremenil že kak uporabnik pred nami.

3.8.1. LINQ to SQL

V projekt, ki smo ga začeli pisati v poglavju 3.5., dodamo nov element (v kontekstnem meniju projekta izberemo Add/New Item... LINQ to SQL Classes).
V dialogu Server Explorer (meni View/Server Explorer) naredimo novo povezavo na našo podatkovno bazo, ki smo jo izdelali v poglavju 3.6. Pod ustvarjeno povezavo se pod mapo Tables nahajata tabeli *Naslov* in *Oseba*. Ti dve tabeli z miško potegnemo iz Server Explorerja v prikazovalnik datoteke DataClasses1.dbml (to je datoteka, ki je nastala, ko smo dodali nov element LINQ to SQL Classes). Na vprašanje, ali povezovalni niz (connection string) želimo shraniti v konfiguracijsko datoteko, odgovorimo pritrdilno.
Spremeniti moramo še nekaj nastavitev v datoteki DataClasses1.dbml. Z desnim gumbom miške kliknemo na prazno površino v prikazovalniku in izberemo Properties). Imenska prostora *Context* in *Entity* nastavimo na *WCFClassLib.DAL*, *Name* pa nastavimo na *DiplomaDataContext*. Imenski prostor smo spremenili zato, da bi se izognili možnim konfliktom z našimi že obstoječimi istoimenskimi razredi.
Da bi naš modul za delo z osebami lahko našel povezovalni niz tudi v spletnem projektu (ki modul referencira), ga moramo prepisati še v web.config (iz app.config je potrebno prekopirati celoten element *connectionStrings* v web.config).
Spodnji primer prikazuje spremembe, ki smo jih naredili v funkciji *GetOsebaByEmso*.

```
public DtoOseba GetOsebaByEmso(string emso)
{
    DiplomaDataContext dc = new DiplomaDataContext();

    var rv = from o in dc.Osebas
             where o.EMSO == emso
             select new DtoOseba
             {
                 EMSO = o.EMSO,
                 Ime = o.Ime,
                 Priimek = o.Priimek,
                 DatumRojstva = o.DatumRojstva,
                 Naslov = new DtoNaslov
                 {
                     Id = o.Naslov.NaslovId,
                     Ulica = o.Naslov.Ulica,
                     HisnaStevilka = o.Naslov.HisnaStevilka,
                     Kraj = o.Naslov.Kraj
                 }
             };

    return rv.SingleOrDefault();
}
```

V *DiplomaDataContext* se nahajajo vsi potrebni razredi za dostop do našega podatkovnega modela. Tabela *Oseba* je dobila ime *Osebas*, kar sovpada z angleško slovnico, kjer končnica „-s“ na koncu samostalnika pomeni množino. Ena oseba (v LINQ izrazu je to spremenljivka *o*) je preko relacije povezana z enim naslovom (*o.Naslov*). Preko relacij lahko dostopamo do potrebnih podatkov. Relacije nam generator LINQ to SQL zgenerira avtomatsko, seveda le, če smo prej pravilno izdelali vse primarne in tuje ključe.
Dodajanje in brisanje sta predstavljena v naslednjem poglavju, urejanje pa še eno poglavje kasneje.

3.8.2. Transakcije

Transakcije so uporabne, kadar v podatkovno bazo zapisujemo več kot en zapis hkrati. V našem primeru se to zgodi pri vsakem dodajanju in brisanju osebe. Kadar dodajamo novo osebo in naslov njenega bivališča, ki v bazi še ne obstaja, se v bazo istočasno zapiše tudi naslov. Kadar osebo brišemo iz baze, hkrati preverimo, ali na njenem naslovu morda ne živi še kaka druga oseba in v primeru, kadar na njem ne živi nihče, le-tega pobrišemo. Se pravi: z enim klicem metode (*AddOseba, DeleteOseba*) pravzaprav ažuriramo dve tabeli.
Izvorna koda za metodo *AddOseba*:

```
public void AddOseba(DtoOseba oseba)
{
     DiplomaDataContext dc = new DiplomaDataContext();
     using (TransactionScope ts = new TransactionScope())
     {
          try
          {
               var dcOseba = ConvertDtoOsebaToDCOseba(oseba);
```

```
                dc.Osebas.InsertOnSubmit(dcOseba);
                dc.SubmitChanges();
                ts.Complete();
           }
           catch (Exception ex)
           {
                throw new FaultException(ex.Message);
           }
     }
}
```

In še koda metode *ConvertDtoOsebaToDCOseba*, ki pretvori naš objekt za prenos podatkov v objekt, ki je potreben za delo s podatkovno bazo:

```
private WCFClassLib.DAL.Oseba
ConvertDtoOsebaToDCOseba(DtoOseba oseba)
{
     Naslov nas = new Naslov();

     return new WCFClassLib.DAL.Oseba
     {
          EMSO = oseba.EMSO,
          Ime = oseba.Ime,
          Priimek = oseba.Priimek,
          DatumRojstva = oseba.DatumRojstva,
          NaslovId = nas.GetIdByData(oseba.Naslov.Ulica,
                                     oseba.Naslov.HisnaStevilka,
                                     oseba.Naslov.Kraj)
     };
}
```

Metoda *GetIdByData* v razredu *Naslov*, ki preveri, če nek naslov že obstaja v bazi. V primeru da naslov obstaja, metoda vrne njegov *NaslovId*, sicer pa ustvari novega in vrne novi *NaslovId*.

```
public int GetIdByData(string ulica, int hisnaSt, string kraj)
{
     DiplomaDataContext dc = new DiplomaDataContext();
     WCFClassLib.DAL.Naslov rv;
     using (TransactionScope ts = new TransactionScope())
     {
          try
          {
               rv = (from o in dc.Naslovs
                     where o.Ulica == ulica &&
                     o.HisnaStevilka == hisnaSt &&
                     o.Kraj == kraj
                     select o).Single();
          }
          catch
          {
```

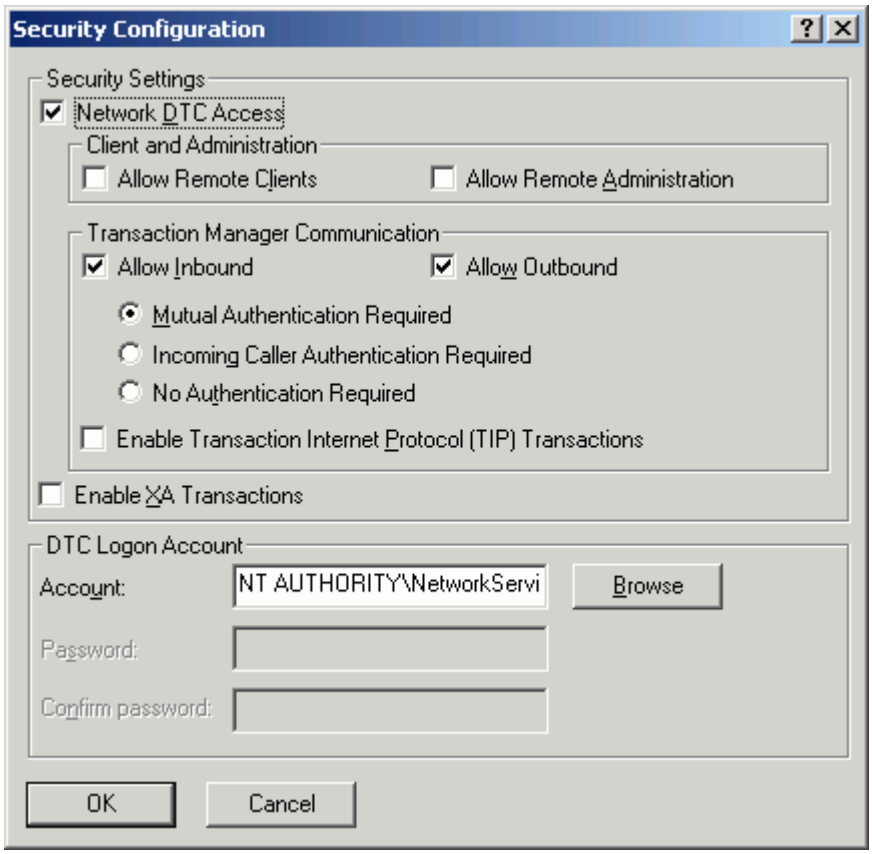
```
                rv = new WCFClassLib.DAL.Naslov
                {
                     Ulica = ulica,
                     HisnaStevilka = hisnaSt,
                     Kraj = kraj
                };

                dc.Naslovs.InsertOnSubmit(rv);
                dc.SubmitChanges();
           }
           ts.Complete();
      }
      return rv.NaslovId;
}
```

Metoda *AddOseba* torej kliče metodo *ConvertDtoOsebaToDCOseba* in ta naprej metodo *GetIdByData* razreda *Naslov*. Kadar se katera od metod *InsertOnSubmit* ali *SubmitChanges*, ki ju kliče metoda *AddOseba*, ne izvede, moramo zapis v tabeli *Naslov* pobrisati (če je bil ustvarjen nov zapis) in s tem podatkovno bazo vrniti nazaj na stanje pred ažuriranjem.
V prihodnosti se lahko zgodi, da bomo klicali metodo *GetIdByData* direktno iz odjemalca ali pa bo ta metoda posredno klicala še kako drugo metodo – v vsakem primeru morajo transakcije delovati pravilno.
Včasih je bila rešitev tega problema bistveno bolj kompleksna, kot je danes. Ponavadi so razvijalci izdelali dodatno plast, ki je bila namenjena samo in točno temu, da ustvari in konča transakcijo. V primeru, da bi obstajala boljša rešitev, bi dodatno plast lahko uvrstili med odvečno kodo. V našem primeru takšne kode nočemo pisati, saj vsaka odvečna koda poveča možnost napak.
Rešitev v obliki razreda za upravljanje s transakcijami, *TransactionScope*, je prišla z ogrodjem .NET verzije 2.0. Uporaba razreda je vidna v zgornjem primeru pri metodi *AddOseba* in *GetIdByData*. Če vsako metodo ažuriranja (dodajanje, urejanje, brisanje) ovijemo v konstrukt,

```
using(TransactionScope ts = new TransactionScope())
{
    //operacije ažuriranj
    ts.Complete();
}
```

bo operacijski sistem sam poskrbel, da bo prva inicializirana transakcija popolnoma nova (*new TransactionScope* v metodi *AddOseba*), vse nadaljne (*new TransactionScope* v metodi *GetIdByData*) pa bodo preverile, ali transakcija že obstaja in v tem primeru uporabile že obstoječo. Vsak *TransactionScope* je dolžan poklicati lastno metodo *Complete*. Če se to ne zgodi, bo celotna hierarhija transakcij izvedena neuspešno, kar bo povzročilo razveljavitev. Z uporabo razreda *TransactionScope* pridobimo konsistenten model programiranja, pri katerem nam ni treba skrbeti, katera metoda mora inicializirati novo transakcijo, ampak jo lahko inicializira vsaka, sistem pa zagotovi, da bo glavna tista, ki je inicializirana prva.
Da *TransactionScope* začne delovati [15], moramo v oknu Control Panel/Administrative Tools/Component Services z desnim klikom na Console Root/Component Services/Computers/My Computer izbrati opcijo Properties. V pogovornem oknu, ki se odpre, izberemo zavihek MSDTC in kliknemo na gumb Security Configuration... Prikaže se pogovorno okno, na katerem nastavimo opcije tako, kot jih prikazuje slika 11.



Slika 11. Pravilne nastavitve v pogovornem oknu Security Configuration

Oglejmo si še operacijo brisanja:

```
public void DeleteOseba(DtoOseba oseba)
{
     DiplomaDataContext dc = new DiplomaDataContext();
     using (TransactionScope ts = new TransactionScope())
     {
          try
          {
               var dcOseba = (from o in dc.Osebas
                              where o.EMSO == oseba.EMSO
                              select o).Single();

               dc.Osebas.DeleteOnSubmit(dcOseba);
               dc.SubmitChanges();
               Naslov nas = new Naslov();
               nas.TryDelete(dcOseba.Naslov);
               ts.Complete();
          }
          catch (Exception ex)
          {
               throw new FaultException(ex.Message);
          }
     }
}
```

V parametru *oseba* zgornje metode pridobimo ključ (*EMSO*) osebe, ki jo želimo pobrisati. Metoda *DeleteOnSubmit* potrebuje kot parameter objekt, ki predstavlja osebo s tem ključem in ne samo vrednosti ključa, zato s pomočjo stavka LINQ to osebo preberemo iz podatkovne baze. S klicem metode *TryDelete* razreda *Naslov* zagotovimo, da se pobriše naslov pobrisane osebe, če je bila le-ta zadnja živeča na tem naslovu.

```
public void TryDelete(WCFClassLib.DAL.Naslov naslov)
{
     if (naslov.Osebas.Count == 0)
     {
          DiplomaDataContext dc = new DiplomaDataContext();
          using (TransactionScope ts = new TransactionScope())
          {
               dc.Naslovs.Attach(naslov, false);
               dc.Naslovs.DeleteOnSubmit(naslov);
               dc.SubmitChanges();
               ts.Complete();
          }
     }
}
```

V tej metodi smo uporabili drugačen pristop k brisanju podatkov kot v prejšnji metodi. Naslov osebe smo pridobili v drugem podatkovnem kontekstu (*DataContext*), zato ga v kontekstu, inicializiranem v zgornji metodi, ni. V ta podatkovni kontekst ga moramo pripeti (*Attach*) ročno. Prvi parameter metode *Attach* je objekt tipa *Naslov*, drugi parameter pa pove, da podatki naslova niso bili spremenjeni. Po tem, ko naslov pripnemo, ga lahko pobrišemo (seveda, če je število oseb, živečih na tem naslovu, enako nič).

3.8.3. Preverjanje hkratnega ažuriranja zapisov

Pri ažuriranju starih zapisov moramo preveriti, ali ni kdo po našem branju in pred našim ažuriranjem obstoječega podatka le-tega že spremenil. Takšnemu preverjanju pravimo tudi nadzor nad sočasnim izvajanjem transakcij.
Omenjeni nadzor je možno implementirati na več načinov [16]. Dodamo lahko polje s časovnim žigom (timestamp) in preverjamo tega ali pa preverjamo kar vsa polja, kar je precej bolj zamudno, če stavke SQL pišemo ročno. LINQ to SQL nam to delo olajša.

```
public void EditOseba(DtoOseba oseba, DtoOseba old_oseba)
{
   DiplomaDataContext dc = new DiplomaDataContext();
   using (TransactionScope ts = new TransactionScope())
   {
      try
      {
         var dcOseba = ConvertDtoOsebaToDCOseba(oseba);
         var dcOldOseba = onvertDtoOsebaToDCOseba(old_oseba);

         dc.Osebas.Attach(dcOseba, dcOldOseba);
         dc.SubmitChanges(ConflictMode.FailOnFirstConflict);
         ts.Complete();
      }
```

```
      catch (Exception ex)
      {
         throw new FaultException(ex.Message);
      }
   }
}
```

Tako kot pri metodi za brisanje tudi metodi za preverjanje pripnemo podatke o osebah v trenutni podatkovni kontekst (le da za to uporabimo drugo metodo - *Attach*). Prvi parameter metode *Attach* predstavlja nove podatke, drugi pa originalne podatke. LINQ to SQL iz obojih zgenerira UPDATE stavek SQL tako, da primerja vse stare vrednosti z zapisi v bazi, spremeni pa samo tiste, ki so res spremenjeni. Primer takega stavka:

```
UPDATE Oseba
SET DatumRojstva = @DatumRojstva
WHERE EMSO = @old_EMSO
  AND Ime = @old_Ime
  AND Priimek = @old_Priimek
  AND DatumRojstva = @old_DatumRojstva
  AND NaslovId = @old_NaslovId
```

Parametri s predpono old_ pridobijo vrednosti s spremenljivke *dcOldOseba*, vsi ostali pa iz *dcOseba*.

4. Zaključek

Pri izdelavi primera N-nivojske arhitekture sem se osredotočil na čim enostavnejšo implementacijo. Za razvoj nisem uporabil nobene dodatne knjižnice, temveč izključno tiste, ki so integrirane v Microsoftov razvijalski paket Visual Studio 2008. Zavedam se, da Microsoft ne dela najboljših programov in da moram zaradi tega na tem mestu predstaviti tudi drugačne pristope k implementaciji N-nivojske arhitekture.

Microsoft se v jeziku .NET osredotoča na podatkovno usmerjeno implemetacijo (data-centric design), za razliko od npr. Jave, ki že dolgo uporablja domensko implementacijo (domain-driven design). Razlika med njima je ta, da se pri domenski implementaciji osredotočamo na celoten problem, ki ga moramo rešiti, pri podatkovno usmerjeni implementaciji pa se osredotočamo na podatke in problem rešujemo s podatkovnega vidika. Seveda to ni edina razlika, je pa ena izmed takšnih, ki jasno prikaže, da Microsoft nima najboljše rešitve.

V času zbiranja gradiva za to diplomo sem čakal na izid ogrodja ADO.NET Entity Framework, ki pa žal ni izšlo pravočasno. V diplomi sem zato uporabil tehnologijo LINQ to SQL. Obe tehnologiji sta si med seboj podobni, razlika med njima je v tem, da LINQ to SQL deluje samo nad podatkovno bazo Microsoft SQL Server, ADO.NET Entity Framework pa naj bi podpiral vse podatkovne baze (s takim pristopom dosežemo neodvisnost od podatkovnega vira). Žal so beta preizkuševalci te nove Microsoftove tehnologije o njej podali zelo negativno mnenje [17]. Pričakovali so, da bo tehnologija osredotočena bolj domensko in ne toliko podatkovno, a se jim želje niso uresničile. Microsoft se je odzval z odgovorom, da se bodo na domensko implementacijo osredotočili šele v prihodnjih verzijah Visual Studia. ADO.NET Entity Framework naj bi predstavljal direktno konkurenco odprtokodni tehnologiji NHibernate, a naj bi njene kvalitete ne dosegal. Izdaja ogrodja je bila zaradi tega premaknjena v prihodnost.

Zadnje čase se veliko govori o implementaciji programskih sistemov s „Test-driven design“ pristopom. Pri takšnem pristopu moramo težiti k čim večji neodvisnosti programskih komponent. To je potrebno zato, da je mogoče neodvisno testirati čim manjše koščke programske kode. Pri tem si pomagamo z vzorcem „dependency injection“ z drugim imenom „inversion of control“. Za takšen način programiranja nam ASP.NET WebForms, del ogrodja .NET, ki smo ga uporabili pri implementaciji predstavitvenega nivoja, ne pride ravno v poštev, saj testiranje uporabniškega vmesnika ne more potekati neodvisno od ostalih komponent. Boljši pristop pri razvoju predstavitvenega nivoja predstavlja uporaba vzorca MVC (Model-View-Controller). Tehnologija, ki podpira takšen način razvoja spletnih strani, se imenuje ASP.NET MVC in je trenutno še v „testnih vodah“. Implementira jo Microsoft.

Za nadaljne raziskovanje priporočam pregled rešitve Sharp Architecture [18], ki uporablja Microsoftove tehnologije in nekaj zelo kvalitetnih odprtokodnih rešitev. Uporablja princip „test-driven“ pristopa s tehnologijami NHibernate in Dependancy Injection, ter ASP.NET MVC Preview.

Nato pa priporočam tudi pregled ostalih tehnologij, ki bodo vključene v prvi paket popravkov za Visual Studio 2008.

Microsoft se – resda s precejšnjim zaostankom – očitno obrača v pravo smer. Kaj nam bo ponudil, bo morda bolj jasno čez kako leto ali dve. Trenutno imamo od najnovejših pristopov k razvoju programskih rešitev večinoma na razpolago le odprtokodne knjižnice (NHibernate, Spring.NET, Castle Project). Sam lahko potrdim, da je knjižnica NHibernate že zelo stabilna in zrela knjižnica. Z malce dodatnega zaupanja se nam ni treba bati, da bi z njeno uporabo lahko prišlo do problemov kompatibilnosti z novejšimi verzijami.

Slike

Tabele

Literatura

[1] (2005) Dude, where's my business logic? Dostopno na: http://www.codeproject.com/KB/architecture/DudeWheresMyBusinessLogic.aspx
[2] (2007) Choosing the Right Database. Dostopno na: http://www.paragoncorporation.com/ITConsumerGuide.aspx?ArticleID=1
[3] (2008) Persistence (computer science). Dostopno na: http://en.wikipedia.org/wiki/Persistence_(computer_science)
[4] R. Lhotka, *Expert C# 2005 Business Objects, Second Edition*, Apress, 2006
[5] (2008) Business logic. Dostopno na: http://en.wikipedia.org/wiki/Business_logic
[6] (2005) Tier Pressure and Isolationism. Dostopno na: http://www.codeproject.com/KB/architecture/TierPressure.aspx
[7] L. Bolognese, M. Pizzo, K. Short, ..., *Designing Data Tier Components and Passing Data Through Tiers*, Microsoft, 2002
[8] D. Vidmar, „Kaj je novega v C# 3.0“, *Monitor*, December 2007
[9] (2008) Language Integrated Query. Dostopno na: http://en.wikipedia.org/wiki/Linq
[10] (2004) Basic Profile Version 1.1. Dostopno na: http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html
[11] (2008) Database normalization. Dostopno na: http://en.wikipedia.org/wiki/Database_normalization
[12] (2007) UpdateParameters Collection Usage. Dostopno na: http://www.webswapp.com/codesamples/aspnet20/updateparameters/default.aspx
[13] (2006) Using Bind with nested properties. Dostopno na: http://mikeoff.blogspot.com/2006/07/using-bind-with-nested-properties.html
[14] (2007) Using ASP.NET 3.5's ListView and DataPager Controls: Displaying Data with the ListView. Dostopno na: http://aspnet.4guysfromrolla.com/articles/122607-1.aspx
[15] (2008) How To Enable TransactionScope commands for unit tests. Dostopno na: http://www.adverseconditionals.com/2008/04/how-to-enable-transactionscope-commands.html
[16] (2007) Data Retrieval and CUD Operations in N-Tier Applications (LINQ to SQL). Dostopno na: http://msdn.microsoft.com/en-us/library/bb546187.aspx
[17] (2008) Testers give Microsoft’s Entity Framework a no-confidence vote. Dostopno na: http://blogs.zdnet.com/microsoft/?p=1457
[18] (2008) S#arp Architecture: ASP.NET MVC with NHibernate. Dostopno na: http://code.google.com/p/sharp-architecture/

Izjava

Izjavljam, da sem diplomsko nalogo izdelal samostojno pod vodstvom mentorja doc. dr. Marka Bajca. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.

Ljubljana, 18.8.2008 Marko Podgoršek