Primjena uzoraka dizajna u izradi sloja pristupa podacima

Kulenović, Dario

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dario Kulenović

Primjena uzoraka dizajna u izradi sloja pristupa podacima

ZAVRŠNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dario Kulenović

Matični broj: 45144

Studij: Informacijski sustavi

Primjena uzoraka dizajna u izradi sloja pristupa podacima

ZAVRŠNI RAD

Mentor/Mentorica:

Dr. sc. Marko Mijač

Varaždin, studeni 2022.

Dario Kulenović

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Dohvaćanje i spremanje podataka su jedne od najvažnijih operacija kod današnjih računalnih programa bili oni stolni, na Web-u ili mobilni. Stoga, primjena različitih uzoraka dizajna nužna je s obzirom na repetitivnost problema kod izgradnje arhitekture programa, ali i kod poslovne logike koja se obrađuje u samim programima. Uzorci dizajna nam olakšavaju i poboljšavaju efikasnost pisanja samog koda te otklanjaju dileme koje mogu nastati kod kreiranja programa. Također pružaju strukturirani pristup samoj izgradnji arhitekture programa, odnosno slojeva programa. Konkretno, sloj za pristup podataka, u moderno vrijeme kada se aplikacije mogu rapidno mijenjati česta praksa je odvojiti kao zaseban sloj aplikacije čime se omogućuje modularnost i neovisnost samog sloja kao i rada nad njime. U ovom završnom radu obraditi će se najznačajniji uzorci dizajna sloja za pristup podatcima koji se trenutno koriste. Nakon obrade najznačajnijih uzoraka, za postojeću aplikaciju će se odrediti da li se ona može unaprijediti refaktoriranjem uz pomoć nekog od obrađenih uzoraka dizajna ili kombinacijom više njih te će se analizirati posljedicu u performansama koji je izazvala promjena arhitekture sloja za pristup podatcima. Kod refaktoriranja će se koristiti Microsoft Visual Studio 2022. i Microsoft SQL Server Management Studio 18 te će aplikacija biti dostupna online na platformi GitHub. Koristiti će se programski jezik C# te .NET Framework. Za izradu dijagrama koristiti će se Visual Paradigm Community Edition.

Ključne riječi: uzorci dizajna; sloj; uzorak; baza podataka; sloj za pristup podatcima; računalni programi; podatci;

Sadržaj

1. Uvod

U samim početcima softverskog inženjerstva računalni programi nisu imali vizualno sučelje i nisu se koristila spremišta podataka kakvih ih danas poznajemo u obliku baza podataka pa je s time arhitektura aplikacije bila jednostavnija nego danas. Komunikacija između korisnika računala i samog računala se vršila preko komandnog sučelja gdje bi korisnik pisao tekst, a računalo bi također u obliku teksta vratilo povratnu poruku na temelju izračuna ili neke druge radnje koja se obavljala u samom kodu. Možemo reći da se nije vodilo računa o tome da li će programi biti odvojeni na slojeve. Računalni programi bili su jednoslojni jer se programski kôd sastojao od jedne cjeline. Njegova svrha je bila da pokrije spremanje podataka u memoriju, obradu i ispis podataka korisniku natrag u komandno sučelje.

S razvitkom samih računala i poboljšanjem hardverskog dijela računala, s vremenom su se počela uvoditi grafička sučelja (engl. Graphical User Interface). Ona su omogućila prikaz podataka na monitorima računala na vizualno potpuniji način za samog korisnika. To bi značilo da je korisnicima bilo intuitivnije komunicirati sa programom bez nekog većeg znanja pozadinske programske logike odnosno putem formi ili prozora sa gumbima. U usporedbi s ranijim računalnim programima koji su koristili komandno sučelje i zahtijevali određeno predznanje za upravljanje samim sučeljem. Ono je u velikoj mjeri bilo slično samom programiranju i vrlo neintuitivno za korištenje krajnjem korisniku.

Spremanje podataka je također napredovalo, počele su se koristiti baze podataka. Danas poznajemo relacijske baze podataka za koje možemo reći da su trenutno najzastupljenije od svih vrsta spremišta podataka i većina današnjih aplikacija ih koristi. [1] Poslovna logika je s vremenom postala kompleksnija i količina podataka koja se obrađuje postaje sve veća i veća i stoga su performanse samih aplikacija postale ključne kako bi se osiguralo što brže izvođenje procesa unutar samih organizacija. S obzirom na napredak prethodno navedenih komponenata samih računalnih programa stvorila se potreba za odvajanjem svakog od njih u poseban sloj zbog toga što se time osiguralo nesmetano odvijanje svake od komponenata, modularnost, skalabilnost, bolje performanse i standardiziranu komunikaciju među slojevima. Za potrebe ovoga rada koristiti ćemo se troslojnom arhitekturom aplikacije, iako je bitno istaknuti da to nije jedina vrsta arhitekture računalnih programa. Tri sloja u troslojnoj arhitekturi su prezentacijski, sloj poslovne logike i sloj za pristup podatcima. Primarni fokus ovoga rada biti će na sloju za pristup podatcima te najznačajnijim uzorcima dizajna koji su postali uobičajena praksa u izgradnji arhitekture samog sloja za pristup podatcima. Cilj ovoga rada je obraditi temu samih uzoraka dizajna podatkovnog sloja podataka

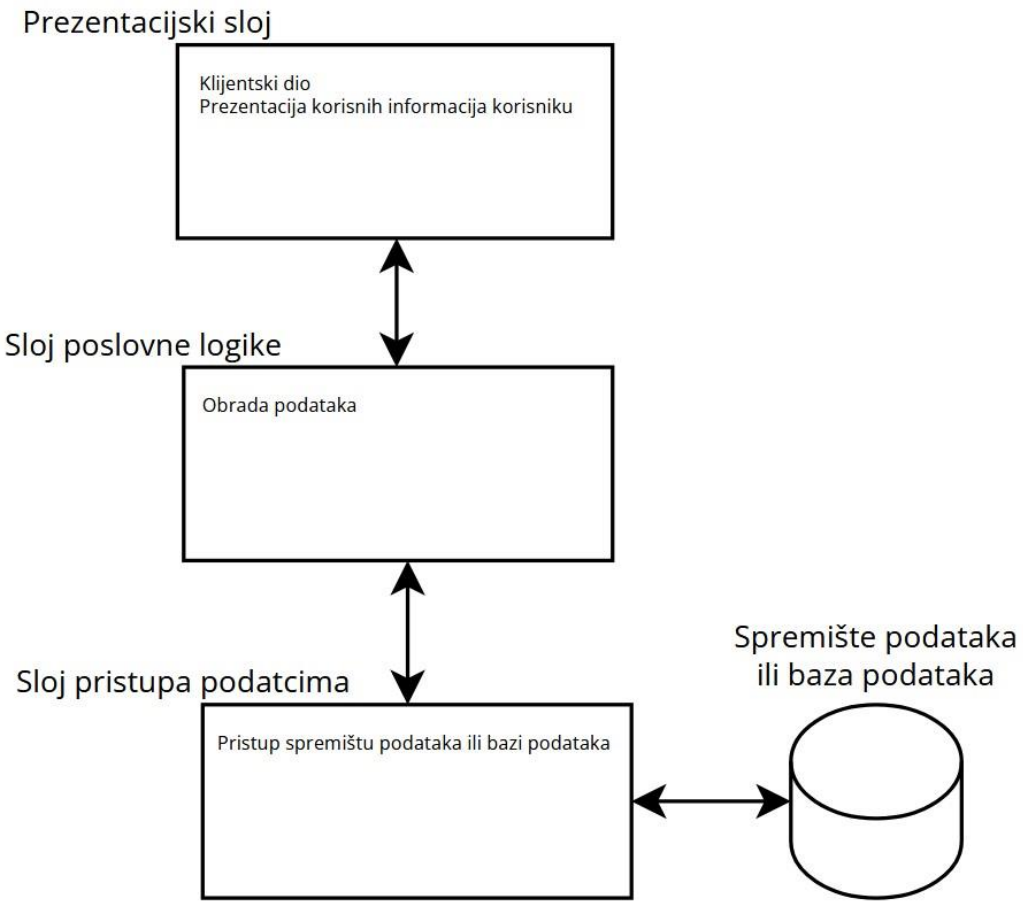
i prema prikupljenim znanjima o njima praktično ih primijeniti na način da će se refaktorirati sloj za pristup podataka prethodno izrađene aplikacije. [2]

2. Sloj za pristup podatcima

Sloj za pristup podatcima je sloj u troslojnoj arhitekturi računalnih programa koji je zadužen za pristup podatcima i bazama podataka. To bi u praksi značilo pristup samim bazama podataka slanjem upita pomoću strukturiranog upitnog jezika (engl. Structured Query Language ili kraće SQL) ili pohranjivanje podataka u obliku datoteka u datotečnom sustavu. Mi ćemo se s obzirom da radimo na aplikaciji koja će koristiti bazu podataka, konkretno relacijsku, fokusirati na baze podataka i koristiti ih za pohranu podataka unutar samog podatkovnog sloja. Kod baza podataka je važno istaknuti da su one naspram ostalih dijelova aplikacije jedine statične, trajne i cilj im je da očuvaju podatke u dužem vremenskom periodu bez rizika od njihovog uništavanja ili neželjenih promjena. Ovo poglavlje je napisano na temelju izvora [3].

Podatkovni sloj komunicira sa slojem poslovne logike i prezentacijskim slojem na način da sloj poslovne logike od podatkovnog sloja upitom traži određenu količinu podataka i nakon toga logički sloj obrađuje te podatke, odnosno vrši nad njima određene operacije u obliku matematičkih izračuna i drugih operacija kako bi se došlo do željenog rezultata. Rezultat obrade podataka sa sloja poslovne logike se dalje prosljeđuje prezentacijskom sloju koji te podatke koristi kako bi stvorio prikladan prikaz tih podataka takvih kakvi su proslijeđeni, u obliku grafikona, grafika i slično, ovisno o domeni i namjeni samog računalnog programa. Kada govorimo izvan okvira troslojne arhitekture, sloj za pristup podatcima može biti smješten i u samom sloju poslovne logike umrežen u kôd same poslovne logike tako da zajedno s kôdom poslovne logike tvori neodvojivu cjelinu. Obično je kôd za pristupanje podatcima u tom slučaju rađen prema poslovnoj logici i ovisan je o njenim promjenama. Na taj način se onemogućava lakoća izvršavanja promjena i održavanje koda koji se koristi pristup podatcima jer je vrlo vjerojatno da će u slučaju promjene poslovne logike to posljedično zahtijevati i promjenu podatkovnog modela i također promjene nad načinom pristupa podataka unutar sloja. Ponovno će se morati napraviti cijeli dio za pristup podatcima i dio programske logike što zahtjeva mnogo vremena i teže je takav računalni program održavati jer nema jasne razlike i odijeljenosti između tih dvaju dijelova. Kod troslojne arhitekture je mnogo lakše upravljati promjenama nad samim slojem zbog toga što se u slučaju promjene poslovne logike sloj za pristup podatcima može lako modificirati kako bi služio novoj programskoj logici.

Kod programiranja koristeći troslojnu arhitekturu ili općenito višeslojnu arhitekturu pomaže to što je sloj za pristup podatcima i sam podijeljen na zasebne dijelove koji se mogu po potrebi mijenjati i dodavati ih. Na sljedećoj slici možemo vidjeti kako bi izgledao koncept troslojne arhitekture:



Slika 1. Prikaz troslojne arhitekture

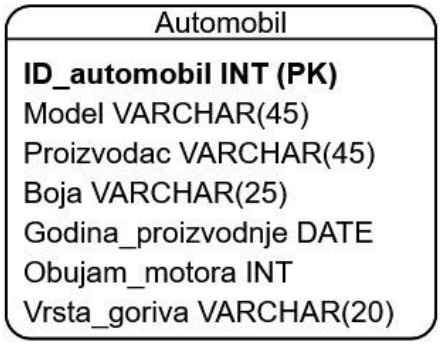
Za samu izradu podatkovnog sloja potrebno nam je više dijelova koji ujedno predstavljaju komponente za pristup podatcima, izvore podataka i njihove modele, a to su:

- Tehnologija za pristup podatcima
- Modeli podataka
- Pohranjene procedure ili pogledi
- Tablice u bazi podataka
- XSLT, XML (pretežito kod Web aplikacija)

Što se tiče samih alata i tehnika za modeliranje sloja za pristup podatcima pretežito koristimo modeliranje pomoću ERA modela i modela/dijagrama podataka o čemu će biti riječi u sljedećem poglavlju.

2.1. ERA modeliranje

ERA model predstavlja statični model podataka koji nam služi za opis podataka koji se koriste u poslovnom svijetu, a to radimo uz pomoć entiteta, njihovih atributa i definiranih relacija (veza) među njima. Na slici ispod možemo vidjeti jedan primjer entiteta automobila.



Slika 2. Primjer entiteta u ERA modelu (vlastita izrada)

Entitet ima svoj naziv koji je u ovom slučaju „Automobil“ te pripadajuće atribute koji označavaju osobine samog automobila koje su jednoznačne i trebaju imati stvarnu primjenu na taj automobil. Kada bismo atributima pridružili stvarne vrijednosti, odnosno sirove podatke, dobili bismo jedan redak u tablici baze podataka Automobil čiji nazivi stupaca odgovaraju nazivima atributa, konkretno: Model, Proizvođač, Boja, Godina proizvodnje, Obujam motora i Vrsta goriva. Primarni ključ ovog entiteta je ID_automobil i izražen je brojem odnosno tipom podatka INT (engl. integer). Od tipova podataka još se spominju VARCHAR koji predstavlja određen broj slova koji taj atribut može primiti. DATE koji prima datum u formatu „YYYY-MM-DD“ gdje su „Y“ znamenke godine, „M“ znamenke mjeseca u godini, „DD“ znamenke dana u godini.

ERA modeliranje također definira pristup gradnji samog modela i akcije koje trebamo poduzeti da bi sagradili kvalitetan model, a to su:

- Definiranje atributa
- Definiranje primarnog ključa
- Definiranje relacija
- Provjera ispravnosti modela

Kako bismo napravili dobar ERA model potrebno je prvo definirati atribute za pojedine entitete i definirati primarni ključ za svaki entitet. Primarni ključ mora sadržavati određena svojstva kako bi bio validan kandidat za odabir. On mora biti jedinstven zbog toga što je primarni ključ sam po sebi ono što razlikuje jedan zapis u tablici u samoj bazi podataka od drugih zapisa i čini taj zapis jedinstvenim te ga je kao takvog jednostavno pozvati kada se za to ukaže potreba bez

brige o tome da ćemo dobiti neki drugi zapis umjesto toga. On ne smije imati vrijednost NULL odnosno na njegovom mjestu ne smije biti prazna vrijednost. Treći uvjet je da za svaki entitet mora postojati jedan i samo jedan primarni ključ. Kada smo definirali atribute entiteta i njihov primarni ključ onda možemo krenuti definirati veze između entiteta. Pri tome treba obratiti pozornost na brojnost veze i poziciju vanjskog ključa. Brojnost veze označava brojčani odnos između entiteta u smislu koliko jednog entiteta ima s obzirom na drugi. Na primjer, jedna država može imati samo jedan glavni grad pa kažemo da je to veza jedan naprema jedan. Postoje još i druge veze kao što su više na više, jedan na više kojima se grade relacije između entiteta. Prilikom građenja ERA modela, ako pratimo prethodno navedena pravila, trebali bi dobiti model koji nije redundantan i sadrži valjanu logiku između entiteta. Ovaj cijeli postupak se provodi rekurzivno što znači da se provjeravanje validnosti pravila mora ponavljati za cijeli model svaki puta kada se izvrši neka promjena na modelu odnosno doda novi entitet kako bi se izbjegle neželjene redundancije i mane u samoj logici modela. [4]

2.2. Relacijske baze podataka

Relacijska baza podataka je jedna od vrsta baza podataka koja je temeljena matematičkom pojmu relacije, a kod koje se podatci i veze između njih prikazuju tablicama koje imaju retke i stupce. Ova vrsta baze podataka je napravljena kako bi se korisniku podatci prezentirali u čitljivijoj formi u obliku tablica (relacija) i da bi se omogućila manipulacija s podatcima pomoću relacijskih operatora.[5] S obzirom na veze među podatcima i samim entitetima, ovaj način pohrane podataka nam omogućava jednostavnije izvršavanje kompleksnih pretraga nad podatcima po određenim parametrima i uočavanje odnosa među podatcima.

Kod ovog pristupa podatci se spremaju u tablice podataka koje sadrže određene informacije specifične za neki entitet. Dijele se na stupce i retke gdje redci predstavljaju jedan zapis u tablici koji se odnosi na neki specifičan objekt ili stvar kojeg opisujemo iz realnog svijeta. Za njega su specificirane njegove posebne osobine koje ga opisuju. Njih nazivamo atributima. Stupci predstavljaju same atribute te možemo reći da svaki stupac predstavlja jedan atribut ili u stvarnom svijetu jednu osobinu predmeta kojeg pokušavamo opisati putem ovog načina zapisa. Možemo još reći da se jedan skup zapisa u tablici može nazvati n-torkom (engl. tuple) jer sadrži n zapisa odnosno određen broj zapisa. Na sljedećoj slici možemo vidjeti prikaz kako bi izgledala jedna tablica u bazi podataka za entitet „Automobil“.

Automobil						
ID	Model	Proizvođač	Boja	Godina proizvodnje	Vrsta goriva	Obujam motora
1	Serija 3	BMW	Crna	2011	Benzin	1999
2	Octavia	Škoda	Crvena	2018	Diesel	2449
3	Renault	Clio	Bijela	2015	Diesel	1449
...	...	...	...	...	...	...

Slika 3. Prikaz zapisa u tablici baze podataka (vlastita izrada)

Kao što je prethodno navedeno u tekstu, zapisi su raspoređeni u retke, a atributi u stupce tablice. Možemo primijetiti da se u prvom stupcu nalazi atribut „ID“. On predstavlja primarni ključ tablice koji je jedan od najvažnijih elemenata u relacijskim bazama podataka. On služi kao jedinstveni identifikator po kojemu ćemo zapis kasnije moći pronaći kod same pretrage podataka. Osigurava da ne dođe do pojave redundantnosti, odnosno dupliciranih zapisa što bi bilo nepoželjno s obzirom da je cilj ovog pristupa i općenito cilj efikasne pohrane podataka da se redundantnost eliminira. Ne želimo bespotrebno spremanje podataka koji već postoje zbog toga što to stvara dodatan stres na samu memoriju podataka.

Kod ove vrste baza podataka, dodatno još postoje dva ograničenja u smislu pravila koja je važno poštovati pri samoj izgradnji kako bi se osigurao integritet podataka i pravilno postavili primarni i vanjski ključevi, a to su:

- Integritet entiteta
- Referencijalni integritet

Integritet entiteta nam nalaže da primarni ključ uvijek mora biti jedinstvena vrijednost i ne smije biti prazna vrijednost ili NULL, a referencijalni integritet nam nalaže da svaki zapis s pozicije vanjskog ključa mora postojati i u tablici primarnog ključa tablice iz koje vanjski ključ dolazi. To su dva važna ograničenja koja nas obvezuju da vodimo o njima brigu dok gradimo samu bazu podataka, ali nam u konačnici olakšavaju rad s ovakvom vrstom baze podataka jer eliminiraju redundanciju i greške u logici kod povezivanja tablica relacijama. Možemo primijetiti da pristup izgradnji ERA modela koji je objašnjen u prethodnom poglavlju također korelira sa ovim ograničenjima. [6]

2.3. Sloj za pristup podatcima u .NET tehnologiji

U praktičnom dijelu ovoga rada i kod razrade uzoraka dizajna ćemo se koristiti Microsoftovom platformom besplatnom za korištenje i otvorenog kôda naziva .NET i programskim jezikom C# razvijenim od strane iste kompanije. Ona omogućuje razvoj nativnih

aplikacija za Web, mobilne uređaje, desktop, izradu mikro servisa koji se pokreću na Docker kontenjerima (engl. Docker container), omogućuje strojno učenje, izradu igara i Internet of things. Mi ćemo se fokusirati na desktop aplikacije. Iako je izrada samog sloja za pristup podatcima vrlo slična za većinu platformi, mijenjaju se samo neke specifične komponente vezane za platformu na kojoj se radi. Možemo reći da ako smo obradili jednu, imamo dosta dobro polazište znanja za izgradnju samog sloja za pristup podatcima i na drugim platformama. To je jako korisno ukoliko programer treba u sklopu svoje karijere u nekom trenutku promijeniti tehnologiju na kojoj radi. To je vrlo korisna i moćna osobina ove platforme. U ovom poglavlju će se obraditi bazična arhitektura sloja za pristup podatcima te tehnike koje se koriste u samoj izgradnji sloja u okviru platforme .NET i mogućnosti u obliku tehnologija za pristup podatcima koje sama platforma pruža.

2.3.1. Tehnologije za pristup podatcima

U .NET okruženju pruža nam se mnoštvo različitih tehnologija za pristup podatcima, koje imaju svoje prednosti i mane ovisno o namjeni aplikacije i samoj platformi za koju se aplikacija razvija. U sljedećoj tablici se nalazi popis tehnologija koje se koriste unutar .NET-a i opis njihovih specifičnosti i namjene:

Tablica 1. Popis tehnologija za pristup podatcima unutar .NET-a

Naziv tehnologije	Opis tehnologije
ADO.NET Core	Pruža osnovne CRUD operacije nad bazom podataka, a podržava sljedeće poslužitelje: SQL Server, OLE DB, Open Database Connectivity (ODBC), SQL Server Compact Edition, i Oracle baze podataka
ADO.NET Data Services Framework	Ovaj razvojni okvir (engl. framework) nam daje na korištenje Entity Data Model putem RESTful servisa koji mogu biti pristupani putem HTTP-a. Podatci se mogu biti pristupani direktno putem URI-a, odnosno linka na web mjesto. Može biti konfiguriran

	da vraća podatke u JSON formatu i Atom formatu.
ADO.NET Entity Framework	Ovaj razvojni okvir nam pruža strukturu pristupa podatcima uz čvrsto tipizirani pristup podatcima što znači da je naglasak na samim tipovima podataka te da se kod samog pristupa mora voditi računa o tipovima podataka koji su određeni u relacijskoj bazi podataka iz koje se oni transportiraju. Ključna prednost ovog pristupa je da omogućava lakoću mapiranja objekata poslovne logike sa samom strukturom podataka u relacijskoj bazi podataka i pomaže u brzini reflektiranja promjena između sloja za pristup podatcima i samog sloja gdje su zapisani podatci, odnosno relacijskim bazama podataka. Može se kombinirati sa jezikom LINQ koji dodatno olakšava izvršavanje upita na samu bazu podataka jer pojednostavljuje SQL jezik.
ADO.NET Sync Services	ADO.NET servisi za sinkronizaciju (engl. Sync Services) predstavljaju poslužitelj koji je dio Microsoft Sync programskog okvira (engl. framework) i koristi se kod implementacije sinkronizacije za baze podataka kompatibilne sa ADO.NET-om. Omogućava da se u aplikacije ugradi sinkronizacija što znači da je moguće periodički dohvaćati podatke kako bi se oni osvježavali u između servera i klijentske baze podataka.
Language Integrated Query (kraće LINQ)	LINQ pruža biblioteke koje nadograđuju programski jezik C# u smislu

	pojednostavljene sintakse za upite ne samo kod zamjene za SQL nego i kod upravljanja sa entitetima, XML-om, objektima u memoriji, DataSet, Data Services kao komponentama .NET-a.
LINQ to SQL	Ova tehnologija nam pojednostavljuje pisanje SQL upita prema bazi podataka na samom Microsoft SQL Serveru u smislu da omogućava korištenje čvrstog tipiziranja podataka koji se u pozadini prevode u čisti SQL te se šalju bazama podataka u njima razumljivom obliku.

(Izvor: Microsoft Application Architecture Guide, 2nd Edition, 2009)

Na temelju ovih informacija možemo zaključiti da .NET pruža raznovrsne alate za sam pristup podatcima. To se posebice ističe kod LINQ jezika i njegove kombinacije sa Entity okvirom. Pomoću njih možemo pristupati podatcima i oblikovati sam pristup puno efikasnije nego da to radimo manualno. Olakšavaju pisanje upita prema bazi podataka pretvaranjem samog LINQ kôda u upit razumljiv bazi podataka i stvaranjem poslovnih objekata (entiteta) direktno prema strukturi koja je zapisana u bazi podataka. Zbog toga znatno ubrzavaju sam proces programiranja. [7]

2.4. ADO.NET

ADO.NET je tehnologija unutar Microsoftovog .NET okruženja koja je zadužena za pružanje pristupa bazi podataka kao što smo vidjeli u prethodnom poglavlju. Ona pokriva pristup bazama smještenim na SQL Serveru, putem XML datoteka i izvora podataka (engl. data sources) stavljenih na raspolaganje od strane OLE DB i ODBC. Za OLE DB je važno napomenuti da OLE DB nije više održavan od strane Microsofta te ga se više neće razvijati i nadograđivati stoga se preporuča zaobići njegovu upotrebu u izradi novih aplikacije ako je to moguće. Također je važno naglasiti da Microsoft ActiveX Data Objects (kraće ADO) nije isto što i ADO.NET već su to dvije različite tehnologije. [8]

Ova tehnologija razdvaja pristup podatcima i samu manipulaciju podatcima u dvije cjeline koje se mogu koristiti odvojeno ili zajedno. Programerski okvir .NET Framework se koristi zajedno sa ovom tehnologijom kako bi se dohvaćali podatci spajanjem na bazu

podataka, izvršavanjem upita i dohvaćanjem podataka sa baze. Kada se podatci dohvate oni se mogu direktno obrađivati ili spremati u DataSet objekt da bi poslužili onda kada ih je potrebno prikazati korisniku. DataSet nam također može poslužiti da privremeno spremamo podatke generirane od strane same aplikacije ili čitamo iz XML datoteke. O DataSet objektu i ostalim objektima koji se koriste u ovom pristupu biti će detaljnije obrađeni u zasebnom poglavlju. [9]

2.4.1. Povezani i nepovezani pristup izvorima podataka

Možemo razlikovati povezani (engl. connected approach) i nepovezani pristup o kojima će više riječi biti u sljedećem pod poglavlju. Nadalje, nabrojati i opisati će se klase koje se koriste za povezani i nepovezani pristup. U poglavlju iza toga će se opisati tipovi podataka koji se najčešće koriste u manipulaciji podataka.

Kod povezanog pristupa posebnosti su da on pruža pristup izvorima podataka na način da se može samo čitati (engl. read-only) i da se može samo ići naprijed ili dalje kroz podatke (engl. forward-only). Prednosti mogućnosti isključivo čitanja iz baze podataka su da se osigurava konzistentan upit bazi bez omogućavanja neželjenih promjena podataka u bazi što znači i bolju sigurnost jer u slučaju da netko dobije kontrolu nad našom bazom, može samo čitati, a ne i mijenjati ili brisati podatke. Čitanjem samo sljedećih podataka ili čitanjem naprijed dobivamo bolje performanse jer se ono što je već pročitano ne mora spremati u memoriju i zbog toga je dosta pogodno kod aplikacija koje moraju paziti da efikasno koriste memoriju. Ove dvije osobine predstavljaju prednost u odnosu na nepovezani pristup podatcima jer je sam pristup brži, efikasniji po memoriju i konzistentniji te sigurniji za korištenje. Klase koje su nam na raspolaganju za korištenje kod ovog pristupa su Connection, Command, DataReader, Transaction, ParameterCollection i Parameter. U sljedećoj tablici ćemo opisati funkcionalnost pojedine klase:

Tablica 2. Klase povezanog pristupa podatcima u ADO.NET tehnologiji

Connection	U njoj su sadržane informacije o konekciji na izvor podataka pomoću niza povezivanja (engl. connection string). Putem nje se inicira konekcija te kada se obave određene traksakcije, ona se zatvara.
Command	Sadrži upite koji će se izvršiti prema izvoru podataka. Također u sebi sadrži

	ParameterConnection klasu u kojoj se specificiraju parametri što omogućava parametrizirane upite, odnosno unos novih i ažuriranje zapisa podataka.
DataReader	Ova klasa nam omogućava pristup isključivog čitanja i čitanja isključivo naprijed. Samo izvođenje ove klase provodi se kroz objekt Command.
Transaction	Ova klasa omogućava izvođenje transakcija koje predstavljaju skup upita koje se putem transakcije mogu izvoditi u jednom slanju prema izvoru podataka tako da se za svaki upit prema bazi ne otvara ponovno konekcija i šalje taj isti upit. To također smanjuje vrijeme izvršavanja samih upita i u konačnici je efikasnije.
Parameter	Ova klasa sadržava parametre koji će se proslijediti izvoru podataka unutar upita koji Command klasa izvršava.
ParameterCollection	Ima istu funkciju kao i klasa Parameter, samo što u ovom slučaju ona sadrži kolekciju Parametara što omogućava slanje još više parametara od jednom vezanih za više upita.

(Vlastita izrada, prema: ADO.NET in a Nutshell, Matthew MacDonald, Bill Hamilton, 2003.)

Kod nepovezanog pristupa glavna razlika naspram povezanog pristupa je da se podatci dohvaćaju iz baze podataka, ali se u ovom slučaju oni spremaju u memoriju tako da se mogu kasnije koristiti. Odmah možemo istaknuti kako je ovo po memoriju neefikasan pristup stoga treba obratiti pozornost pri odabiru pristupa te na kakvoj će se opremi i okruženju ta aplikacija smjestiti i po tome odabrati da li je takvo okruženje pogodno, odnosno da li će moći izdržati taj napor, a efikasno provoditi sve potrebne akcije. Klase koje se koriste u ovom pristupu su DataAdapter, DataSet, DataTable, DataColumn, DataRow, Constraint, DataRelation, DataView. U sljedećoj tablici ćemo objasniti funkcionalnost svake od klasa:

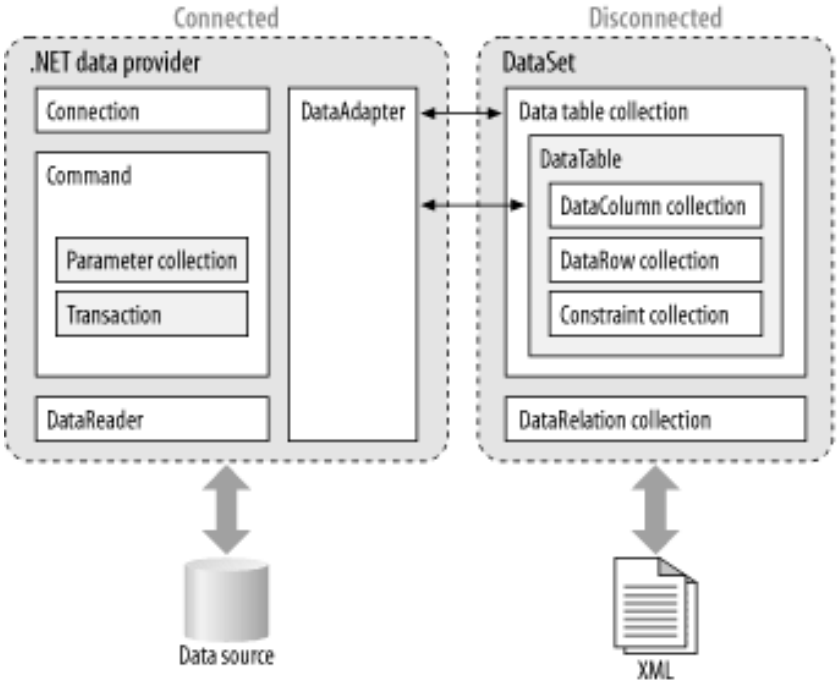
Tablica 3. Klase nepovezanog pristupa podatcima u ADO.NET tehnologiji

DataAdapter	Služi za pristup izvoru podataka i izvlačenje podataka sa izvora kako bi se ti podatci spremili. Podatci se dobivaju slanjem upita prema izvoru podataka.
DataSet	Omogućava pristup podatcima u nepovezanom pristupu što znači da nije potrebna konekcija s izvorom podataka. Možemo ga gledati kao relacijsku bazu podataka koja se nalazi lokalno u memoriji računala. On služi kao spremnik (engl. container) za ostale objekte kao što su: DataTable, DataColumn, DataRow, Constraint, DataRelation. Isto tako može se koristiti za čitanje XML dokumenata koji se mogu obrađivati na način da se čitaju u obliku samog XML-a ili da se koristi DataSet i njegove funkcionalnosti.
DataTable	Ova klasa nam omogućava gledanje podataka u obliku tablice, pa tako u sebi sadrži objekte za pristup stupcima i redcima tablice DataColumn i DataRow. Može se također mijenjati ograničenja tablice putem Constraint klase.
DataColumn	Ova klasa pohranjuje meta podatke o strukturi stupca, ograničenjima i shemom same tablice. Također omogućava stvaranje novog stupca koji će prikazivati vrijednosti izračunate iz drugih stupaca pomoću Expression svojstva.
DataRow	Odgovara retku tablice u bazi podataka. Pomoću DataRow objekta možemo pristupati podatcima u obliku redaka te vršiti nad njima brisanje ili ažuriranje. Posebnost

	je da on sprema promjene nad redcima te se one kasnije mogu ažurirati sa samom bazom podataka.
Constraint	Ova klasa omogućava kreiranje ograničenja nad tablicama koje su spremljene u okviru DataTable klase.
DataRelation	U ovu klasu se spremaju svi podatci o relacijama između tablica u obliku roditeljske i tablice djeteta te se omogućava referencijalni integritet i kaskadno brisanje i ažuriranje tablica.
DataView	Omogućava kreiranje pogleda, odnosno omogućava sortiranje tablica ili filtriranje podataka prema određenim kriterijima.

(Vlastita izrada, prema: ADO.NET in a Nutshell, Matthew MacDonald, Bill Hamilton, 2003.)

Kako bismo rezimirali povezani i nepovezani pristup, na sljedećoj slici možemo vidjeti strukturu jednog i drugog načina pristupa:



Slika 4. Klase kod povezanog i nepovezanog pristupa (preuzeto iz [10])

Na temelju opisanog povezanog i nepovezanog pristupa možemo zaključiti da nam ova tehnologija daje korisne načine za pristup izvorima podataka koje možemo kasnije odabrati ovisno o zahtjevima naše aplikacije. S vremenom kako su se aplikacije s više slojeva razvijale, programeri prelaze na nepovezani pristup (engl. disconnected approach) bazama podataka kako bi se poboljšala skalabilnost aplikacija i smanjio utjecaj na performanse memorije. [10]

3. Uzorci dizajna kod sloja za pristup podatcima

U ovom poglavlju će se obraditi pojam uzoraka dizajna te će se izdvojiti i opisati one uzorke koji su relevantni u izradi sloja za pristup podatcima danas. Za svaki uzorak dizajna će se napraviti njihov opis, objasniti prednosti i mane te ukoliko je primjenjivo priložiti grafički prikaz izgleda samog dizajna u obliku njegove strukture. Struktura opisivanja uzoraka dizajna će biti napisana na temelju knjige Gamma, Helm, Johnson i Vlissides [11]. Uzorci dizajna će biti prvenstveno obrađeni prema knjizi [12], posebno će biti napomenuto ako je sadržaj prema drugim izvorima.

Uzorci dizajna predstavljaju rješenja problema koji se jako često pojavljuju kod razvijanja aplikacija pa ih je stoga moguće rješavati na već poznate, predefinirane načine. Važno je napomenuti da uzorci dizajna nisu algoritmi niti detaljni koncepti koji daju konkretan plan izrade rješenja od početka do kraja za neki problem nego više kao nacrt ili skica koji može služiti kao ideja ili podloga za rješavanje nekog od problema. Uzorci općenito ne postoje samo u programskom inženjerstvu, nego i kod drugih znanosti kao što je arhitektura, matematika, grafički dizajn, psihologija i ostale znanosti. Također ih možemo naći i u prirodi. Povijesno, utemeljiteljem discipline smatra se Christopher Alexander koji je u svojoj knjizi „A Pattern Language“ 1977. godine započeo ideju korištenja uzoraka dizajna u arhitekturi građevina. Njegova definicija uzoraka dizajna je bila: „Svaki uzorak opisuje problem koji se pojavljuje ponovno u našem okruženju, pa tako opisuje srž rješenja tog problema, na takav način da se može ponovno iskoristiti milijun puta, bez da se problem riješi dva puta na isti način.“ Iako ne postoji definitivna niti stroga definicija samog uzorka dizajna, za ovu definiciju se smatra da opisuje ideju uzoraka dizajna jako dobro.

Za strukturu opisivanja uzoraka dizajna uzeti ćemo strukturu:

- Ime uzorka – daje naziv uzorku dizajna koji mora biti prikladan na način da sama srž problema koju uzorak rješava bude odmah uočljiva iz njega
- Svrha – kratko objašnjenje koje odgovara na pitanja: Što ovaj uzorak dizajna radi? Koje je obrazloženje i svrha ovog uzorka? Koju manu u dizajnu ili problem ovaj uzorak rješava?
- Drugi nazivi – jedan uzorak može imati više naziva pa je zbog toga važno navesti i druge nazive kako bi se uzorak bolje prepoznao i kako bi se standardiziralo nazivlje u praksi

- Motivacija – scenarij koji pobliže opisuje problem i rješenje problema s pomoću interakcije između klasa i struktura objekata kako bi se smanjila apstrakcija pojmova i bolje opisao sam uzorak
- Primjena - koje su situacije gdje se koristi ovaj uzorak dizajna, primjeri lošeg načina uporabe i kako prepoznati takve situacije
- Struktura – grafička reprezentacija klasa koje se koriste u uzorku korištenjem tehnika modeliranja, s pomoću jezika UML (engl. Unified Modeling Language)
- Sudionici – klase i/ili objekti koji sudjeluju u dizajnu uzorka, pobliže opisane njihove odgovornosti
- Kolaboracije – kako sudionici zajedno sudjeluju kako bi izvršavali svoje odgovornosti
- Važnost – na koji način uzorak omogućava da se dođe do željenih rješenja problema i koje su mane ili kompromisi kod korištenja konkretnog uzorka
- Implementacija – koji bi se problemi mogli naći pri implementaciji ovog uzorka dizajna, da li ima nekih specifičnih problema vezano uz pojedini programski jezik
- Primjena u praksi – upotreba u praksi u stvarnom svijetu
- Srodni uzorci – koji su uzorci povezani s ovim uzorkom, koje su sličnosti ili razlike koje su najbitnije, sa kojim bi se uzorkom ovaj uzorak dizajna mogao zajedno koristiti

Kako bismo dokumentirali najpoznatije uzorke dizajna, koristiti ćemo se ovakvom strukturom kao vodiljom kada ona bude moguća, ali ćemo ovisno o potrebi izbaciti neke od dijelova ukoliko dođemo do zaključka da nisu potrebni.

3.1. Singleton

Singleton je jedan od najpopularnijih uzoraka dizajna koji pripada grupi kreacijskih (engl. creational) uzoraka dizajna što znači da se koristi pri kreiranju objekata te tako omogućavaju fleksibilnost i ponovnu uporabu kôda. Kao što vidimo ovime se rješava dva problema.

Svrha

Ovaj uzorak dizajna nam omogućava da za objekt onemogućimo stvaranje više instanci tog objekta, a pruža globalni pristup toj instanci objekta.

Motivacija

Prvi problem je stvaranje novih instanci objekata. Konstruktori klasa pri njihovom pozivanju uvijek imaju za zadatak stvoriti novu instancu objekta, oni su na taj način zamišljeni. Ovdje će to biti ograničenje te je potrebno promijeniti logiku u samom konstruktoru kako bi se izbjeglo stvaranje nove instance ako ona već postoji. Vidjet ćemo kako se ovo ostvaruje u praksi kod same strukture klase koju ćemo uzeti za primjer. Drugi problem je omogućavanje globalnog pristupa instanci tog objekta. To ćemo postići tako da ćemo napraviti privatnu statičnu varijablu koja će poslužiti kao instanca tog objekta. Moći ćemo dohvatiti njegovu instancu iz bilo kojeg dijela programskog kôda.

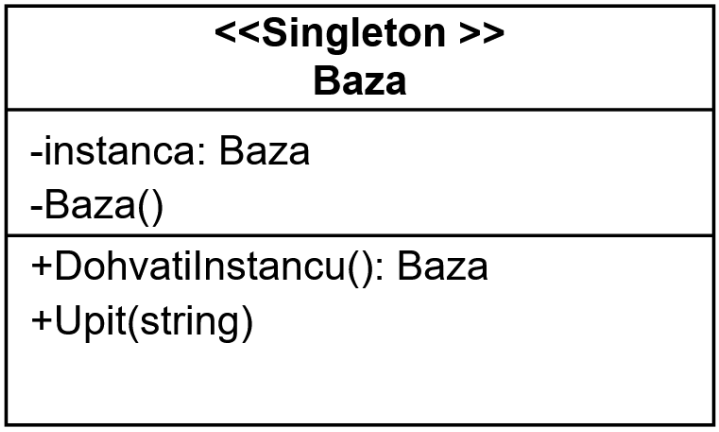
Primjena

Uzorak Singleton koristiti ćemo kada:

- Mora postojati jedna i samo jedna instanca klase i mora biti omogućen pristup klijentu klase sa jasno definiranog mjesta.
- Kada jedina instanca može biti proširena sa stvaranjem podklasa i klijenti klase bi trebali koristiti napravljenu instancu bez mijenjanja programskog kôda.

Struktura

Struktura klase prema ovom uzorku dizajna izgleda kao na sljedećoj slici:



Slika 5: Struktura klase prema uzorka dizajna Singleton (vlastita izrada)

Možemo primijetiti da jedina komponenta ove klase kojoj možemo pristupati je metoda DohvatiInstancu koja nam vraća instancu objekta Baza. Time osiguravamo da svaki puta kada pozivamo tu metodu ne stvaramo novu instancu te klase. To je bitno zbog toga što kod svakog stvaranja nove instance objekta Baza moramo provjeriti da li je taj objekt već prethodno instanciran. U metodu DohvatiInstancu klase ćemo za tu potrebu postaviti uvjet koji kaže: Ukoliko instanca ne postoji, stvori novu instancu, a ukoliko instanca već postoji, vrati već stvorenu instancu. Na taj način osiguravamo da se unutar aplikacije uvijek koristi jedna instanca što rješava prvi problem koji smo prethodno objasnili. Također možemo primijetiti još jednu dodatnu metodu naziva Upit koja će nam služiti za slanje upita u obliku SQL jezika prema bazi podataka što će nam biti bitno kod izgradnje sloja za pristup podatcima kako bi slali upite prema bazi podataka. Za ovaj primjer smo koristili klasu baza da bismo približili uporabu ovog uzorka dizajna samom sloju za pristup podatcima. Ovaj pristup vrijedi i općenito kada imamo potrebu za klasom čija instanca mora biti dostupna u isto vrijeme više mjesta u našem programu i moramo imati veću kontrolu nad njom te kad moramo biti sigurni da se objekt inicijalizira po prvi puta samo onda kada ga prvi puta zatražimo. [12]

Sudionici

Singleton definira metodu kojom klijentima klase daje pristup njezinoj instanci. Također je odgovoran za stvaranje svoje jedinstvene instance.

Kolaboracije

Klijenti pristupaju instanci Singleton-a jedino kroz njegovu metodu gdje se stvara instanca.

Važnost

Uzorak Singleton nam pruža više mogućnosti:

- Kontroliran pristup instanci klase. Zato što klasa enkapsulira svoju instancu, možemo imati strogu kontrolu kako i kada će ju klijenti koristiti.
- Smanjena količina kôda. Ovaj uzorak nam omogućava izostavljanje uporabe globalnih varijabli što znači da nećemo imati mnoštvo globalnih varijabli koje će spremati instance što će posljedično rezultirati sa manje programskog kôda i boljoj čitljivosti kôda.
- Dopušta korištenje i više instanci Singleton klase. Ovaj uzorak nam pruža lakoću promjene programskog kôda jer se kasnije možemo odlučiti i dodati nove instance klase i kontrolirati koliko ih ukupno ima. Da bi to ostvarili moramo samo promijeniti metodu koja stvara instancu.
- Fleksibilnije je koristiti uzorak Singleton nego statičke metode. Funkcionalnost uzorka Singleton možemo izvesti i pomoću metoda sa statičnim članovima kao što je to izvedeno u jeziku C++, ali je teško omogućiti da se kasnije promijeni dizajn i dopusti kreiranje više instanci klase. Statički članovi kod C++-a nisu virtualni, pa podklase nećemo moći pregaziti na polimorfan način.

Negativne strane ovog uzorka dizajna su da on krši pravilo da jedna klasa mora implementirati samo jednu funkcionalnost ili odgovornost zbog toga što istovremeno rješava dva problema. Kod korištenja više dretava, mora se paziti da istovremeno više dretava ne kreira više instanci Singleton objekta. Još jedna negativna strana je da se zbog toga što je klasa u ovom obliku privatna i ne može ju se nadjačati (engl. to override) pa je stoga jako teško ili nemoguće vršiti jedinične testove (engl. unit testing) nad ovakvom klasom.

Implementacija

Kada implementiramo Singleton trebamo osigurati da se instanca klase stvara samo jednom i da joj se kasnije može pristupiti po potrebi. Slijedi implementacija Singleton klase:

```
class Singleton    {
   private static Singleton instance = null;

   private Singleton()
   {
   }
```

```
        public static Singleton Instance
        {
            get
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
                return instance;
            }
        }
    }
```

Prvo smo napravili jednu privatni i statični objekt koji smo nazvali „instance“, a on će nam služiti za stvaranje instance. Privatan je da se njegova vrijednost ne može mijenjati od strane klijenta kako bi smo osigurali stvaranje samo jedne instance, a statičan (engl. static) je zbog toga da se vrijednosti zapisane u njemu sačuvaju kroz instance toga tipa. Nadalje, metoda koja je konstruktor ove klase je privatna tako da se ona ne može instancirati bez uvjeta koji ćemo postaviti u javnom konstruktoru. Taj uvjet zapravo kontrolira da li je objekt instance već stvoren te u slučaju da nije će se stvoriti, a u slučaju da jest, vratiti će se već postojeća instanca naše klase Singleton. Na taj način osiguravamo da se stvori samo jedna instanca i s time smo ispunili svrhu uzorka dizajna Singleton.

Važno je napomenuti da je ovo verzija Singleton uzorka bez podrške za rad s dretvama pa se kao takva ne preporuča za korištenje zbog loših performansi i naknadnih problema koji mogu nastati. Ti problemi nastaju kada imamo više dretvi i recimo da u isto vrijeme dvije dretve pokušavaju instancirati objekt Singleton. Kada se to dogodi ne možemo osigurati da se nije stvorila samo jedna instanca pa stoga to onemogućava ispunjavanje svrhe Singleton-a da treba postojati samo jedna instanca.

Primjena u praksi

Primjene u praksi s obzirom na pristup podatcima su:

- Stvaranje uzorka Facade – koristan je kod kreiranja konekcija s bazom podataka i može u jako velikoj mjeri ubrzati performanse aplikacije
- Dijeljenje podataka – korisno je kad imamo konstante vrijednosti ili konfiguracijske vrijednosti koje se trebaju negdje pohraniti da bi bile čitane od nekih drugih komponenata aplikacije

- Kao cache memorija – kada jednom dohvatimo podatke iz baze podataka ili nekog drugog izvora, efikasno je pohraniti ih u obliku cache-a da bi se mogli brzo poslužiti s tim podatcima jer su zapisani u memoriji našeg računala i ne moramo ih stalno dohvaćati iz baze podataka. Ovdje još proces ubrzavaju i dretve kojima se ubrzava sam proces dohvaćanja podataka.

Srodni uzorci

Možemo reći da se pomoću ovog uzorka mogu implementirati još i uzorci: Abstract factory, Builder, Prototype, Facade.

3.2. Active Record

Svrha

Active Record je uzorak kojemu je svrha kreiranje objekta koji će opisati redak u tablici baze podataka ili pogleda i uz to omogućiti metode za pristup bazi podataka i primjenu poslovne logike nad tim podatcima.

Motivacija

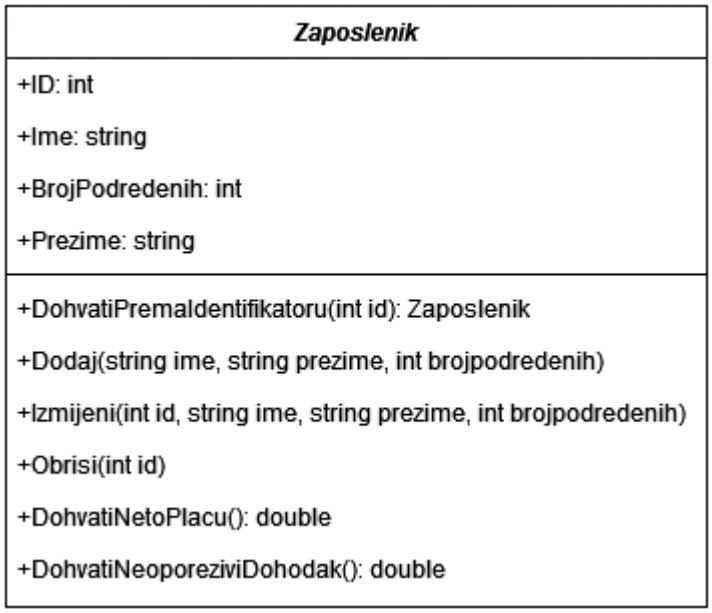
Kod ove klase imati ćemo svojstva klase koji su iste strukture kao što su oni u samoj tablici baze podataka i naglasak je na tome da moraju biti identični. Nadalje, omogućiti ćemo CRUD operacije pomoću metoda u koje ćemo ih smjestiti. Također možemo dodati i metode s kojima ćemo tražiti zapis u tablici pomoću određenih parametara i s time spremiti neke najčešće upite. Važno je naglasiti da će metode traženja biti statične kako bi se očuvali podatci koji se dohvaćaju. I na kraju slijedi dodavanje metoda koje će obrađivati podatke dohvaćene iz baze podataka.

Primjena

Active Record ćemo koristiti kada:

- Poslovna logika nije kompleksna te sadrži većinom CRUD operacije i operacije nad samo jednim zapisom u tablici baze podataka

Struktura



Slika 6: Struktura klase prema uzorku dizajna Active Record

Kod ove klase možemo primijetiti da sadrži i podatke o zaposleniku kao što je ime, prezime i broj podređenih i samu poslovnu logiku u obliku izračunavanja neoporezivog dohotka i dohvaćanja neto plaće. Svaki Active Record objekt je zadužen za čitanje i spremanje podataka u bazu podataka i za poslovnu logiku kojom se dohvaćaju neki drugi podatci na temelju postojećih podataka. Važno je da kod podatkovnog dijela pazimo da on odgovara atributima (stupcima) tablica u bazi podataka jer moramo biti u mogućnosti izvršiti SQL upite prema bazi na temelju tih atributa, odnosno da oni odgovaraju parametrima upita. Metode kod ovog uzorka dizajna mogu izvoditi i druge stvari kao što je pretvaranje tipova pogodnih SQL-u u tipove podataka koji će kasnije biti pogodni za upisivanje u memoriju.

Najčešća uporaba metoda kod ovog uzorka dizajna jest:

- Stvaranje instance Active Record objekta prema retku dobivenom SQL upitom iz baze podataka
- Stvaranje nove instance za kasnije spremanje podataka u tablicu
- Statičke metode traženja u koje možemo pohraniti SQL upite i koja vraća Active Record objekte

- Ažuriranje i ubacivanje podataka u bazu podataka iz podataka koje sadrži Active Record objekt
- Get i set dijelovi
- Implementacija dijelova poslovne logike

Sudionici

Active Record nam pruža pristup svome konstruktoru, CRUD metodama i metodama koje izvršavaju poslovnu logiku.

Kolaboracije

- Klijenti klase koju gradimo prema Active Record uzorku mogu pristupati njenom konstruktoru te stvarati njene instance.
- Pomoću javnih statičnih metoda klijenti mogu dohvaćati podatke iz baze podataka i pohraniti ih u objekt ove klase.
- Pomoću metoda poslovne logike, klijenti mogu dobiti obrađene podatke koje metode vraćaju na temelju odrađenih operacija nad prethodno dohvaćenom objektu.

Važnost

Ovaj uzorak dizajna koristi se kad logika domene ili poslovna logika nije previše kompleksna nego se bazira na CRUD operacijama. Možemo ga još koristiti u slučaju da već koristimo Transaction Script uzorak dizajna, a uvidimo da nam se kôd dosta ponavlja za istu tablicu u bazi podataka. U tom slučaju kreiramo jedan Active Record objekt u koji možemo smjestiti metode koje će odgovarati domenskoj logici i neće biti potrebno raditi više Transaction Script objekata ili metoda. Kod odabira uzorka dizajna u većini slučajeva se bira između ovog uzorka i Data Mapper uzorka te odabir ovisi o kriteriju složenosti same domenske logike. Ako imamo složeniju domensku logiku, odnosno ako ona sadrži mnogo relacija, kolekcija, nasljeđivanja i slično, onda je ćemo u većini slučajeva odabrati Data Mapper. U suprotnom ćemo izabrati Active Record. Još jedna mana ovog uzorka dizajna, koju smo već spomenuli jest da njegova struktura mora strogo pratiti stanje atributa (stupaca) u bazi podataka što znači da će biti teže refaktorirati kod aplikacije u kasnijim stupnjevima samog razvoja.

Implementacija

U kôdu ispod možemo vidjeti implementaciju ovog uzorka dizajna, kod nje smo izostavili implementaciju svih CRUD operacija zbog smanjenja količine koda jer se kôd razlikuje samo u upitima koji se šalju prema bazi podataka pa stoga ne treba posebno pisati za svaku operaciju.

```
    public class Zaposlenik
    {
        private const string CONNECTION_STRING = "data
source=ActiveRecord.db";
        public int ID { get; set;}
        public string Ime { get; set;}
        public string Prezime { get; set;}
        public int BrojPodredenih { get; set;}

        public Zaposlenik(string ime, string prezime, int
brojpodredenih)
        {
            Ime = ime;
            Prezime = prezime;
            BrojPodredenih = brojpodredenih;
        }

        public static object DohvatiPremaImenu(int id)
        {
            using (SqlConnection connection = new
SqlConnection(CONNECTION_STRING))
            {
                connection.Open();
                using(SqlCommand command = connection.CreateCommand())
                {
                    command.CommandType = CommandType.Text;
                    command.CommandText = "SELECT TOP 1 * FROM
[Zaposlenik] WHERE [ID] = ID"; //upit koji dohvaća zaposlenika prema
identifikatoru
                    command.Parameters.AddWithValue("ID", id);
```

```
                    SqlDataReader reader = command.ExecuteReader();

                    reader.Read(); //izvršavanje upita prema bazi
podataka

                    string ime = reader.GetString("Ime");
                    string prezime = reader.GetString("Prezime");
                    int brojpodredenih = (int)reader["BrojPodredenih"];

                    reader.Close();

                    return new Zaposlenik(ime, prezime, brojpodredenih);
                }
            }
        }

        public static object Obrisi(int id)
        {
            // kôd za brisanje zaposlenika
        }
        public static object DodajNovog(string ime, string prezime, int
brojpodredenih)
        {
            // kôd za dodavanje novog zaposlenika
        }

        public static object Izmijeni(int id, string ime, string
prezime, int brojpodredenih)
        {
            // kôd za izmjenu postojećeg zaposlenika
        }
    }
```

Primjena u praksi

Ovaj uzorak možemo koristiti za:

- Pristupanje zapisima u relacijskim bazama podataka bez obzira na tehnologiju.

- Smanjenje količine kôda u slučaju da već koristimo uzorak Transaction Script jer se on može refaktorirati u ovaj uzorak korištenjem Gateway uzorka pa onda njegovog pretvaranja u Active Record.
- Kada poslovna logika nije kompleksna, ovaj uzorak je dobar kandidat za korištenje.

Srodni uzorci

Ovaj uzorak je najsličniji Data Mapper uzorku dizajna.

3.3. Data Access Object

Svrha

Kod ovog uzorka dizajna svrha je razdvajanje pristupa podatcima i domenske/poslovne logike te će nam za ovaj uzorak trebati više klasa kako bismo postigli njegovu strukturu.

Motivacija

Kod ovog uzorka dizajna trebamo tri klase. Prva klasa koju ćemo trebati je klasa koja će biti sučelje prema bazi podataka ili nekom drugom spremištu podataka. Ona će sadržavati CRUD metode za komunikaciju s bazom. Druga klasa će implementirati prethodnu klasu. Biti će zadužena za dohvaćanje podataka iz nekog od izvora podataka. Treća klasa je klasa koja sadržava svojstva koja odgovaraju tablici iz baze podataka i u nju spremamo podatke koji se dohvaćaju pomoću prethodne klase koja dohvaća te podatke.

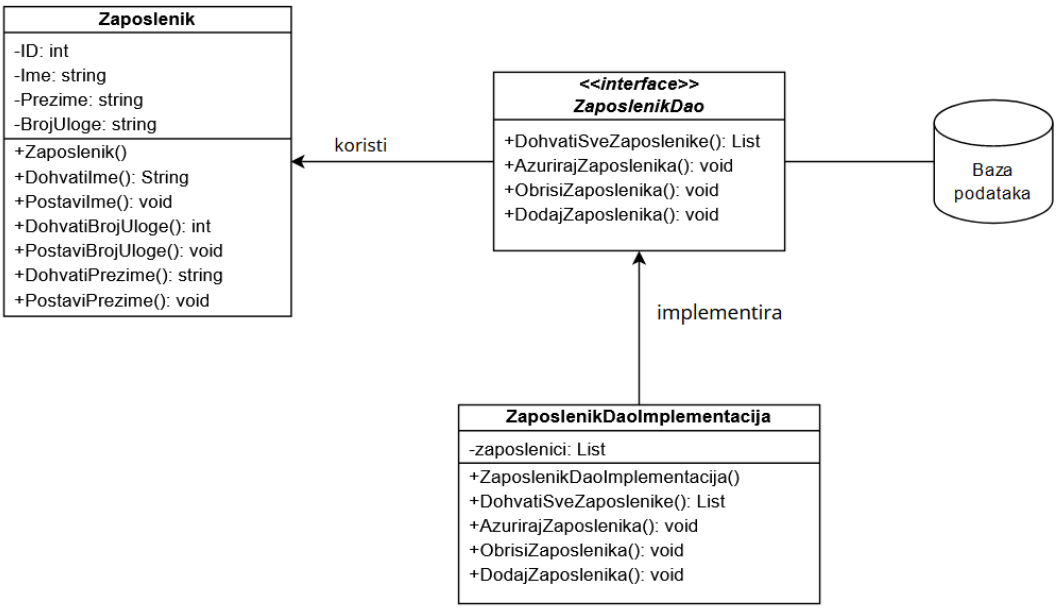
Primjena

Data Access Object ćemo koristiti kada:

- Kada trebamo odvojiti logiku pristupa samim podatcima u poseban sloj kako bi se sloj poslovne logike mogao razvijati zasebno.

Struktura

Na sljedećoj slici možemo vidjeti strukturu samog uzorka dizajna:



Slika 7: Struktura uzorka dizajna Data Access Object (vlastita izrada prema [13])

Sudionici

- Klasa poslovne logike
  - ona treba pristup podatcima kako bi te podatke spremila u svoj objekt
- Klasa sučelja za pristup podatcima
  - klasa koja predstavlja sučelje prema bazi podataka i sadrži metode za pristup podatcima
- Izvor podataka
  - ovaj dio predstavlja samu bazu podataka kao izvor podataka, ali to može biti bilo koji drugi izvor
- Klasa prijenosa/implementacije
  - ovaj objekt služi kao nositelj podataka

Kolaboracije

Objekt klase implementacije se koristiti da vrati podatke natrag klijentu. Objekt klase za pristup podatcima može koristiti objekt implementacije da bi vratio podatke natrag klijentu i obrnuto da bi se ažurirala baza podataka.

Važnost

Glavna prednost ovog uzorka dizajna je u tome da omogućava sloju poslovne logike i sloju za pristup podatcima da se razvijaju nevezano jedan za drugog i također onemogućena je vidljivost između slojeva. Ako očekujemo da će nam se poslovna logika učestalo i brzo razvijati onda je ovo odgovarajući uzorak dizajna. Mane ovog pristupa su u tome da može doći do duplikacije kôda, propuštajuće apstrakcije (engl. leaky abstraction) i inverzije apstrakcije (engl. abstraction inversion). Propuštajuća apstrakcija znači da se korištenjem ovog uzorka može dogoditi da apstrakcijom otkrijemo detalje poslovne logike gdje to inače ne želimo. Inverzija apstrakcije znači da objekti kojima su otkrivene metode od strane sučelja moraju implementirati druge metode koje nisu otkrivene od strane sučelja te se time gubi na performansama i stvara nepotrebna kompleksnost implementacije.

Implementacija

Prvo ćemo prikazati poslovni objekt:

```
public class ZaposlenikDAO
{
    private int ID { get; set; }
    private string Ime { get; set; }
    private string Prezime { get; set; }
    private int BrojUloge { get; set; }

    public ZaposlenikDAO(string ime, string prezime, int brojuloge)
    {
        this.Ime = ime;
        this.Prezime = prezime;
        this.BrojUloge = brojuloge;
    }

    public int DohvatiIdentifikator()
    {
        return ID;
    }
```

```
        public string DohvatiIme()
        {
            return Ime;
        }

        public void PostaviIme(string ime)
        {
            this.Ime = ime;
        }

        public string DohvatiPrezime()
        {
            return Prezime;
        }

        public void PostaviPrezime(string prezime)
        {
            this.Prezime = prezime;
        }

        public int DohvatiBrojUloge()
        {
            return BrojUloge;
        }

        public void PostaviBrojUloge(int brojuloge)
        {
            this.BrojUloge = brojuloge;
        }
 }
```

Nakon toga slijedi sučelje sa metodama za pristup podatcima:

```
public interface  IZaposlenikDAO
    {
        public List<ZaposlenikDAO> DohvatiSveZaposlenike();
        public ZaposlenikDAO DohvatiZaposlenika(int id);
```

```
        public void AzurirajZaposlenika(ZaposlenikDAO zaposlenik);
        public void ObrisiZaposlenika(ZaposlenikDAO zaposlenik);
    }
```

I na kraju ćemo prikazati klasu implementacije:

```
    public class ZaposlenikDAOImplementacija : IZaposlenikDAO
    {
        //lista zaposlenici će predstavljati bazu podataka zbog
jednostavnosti implementacije
        List<ZaposlenikDAO> zaposlenici;

        public ZaposlenikDAOImplementacija()
        {
            zaposlenici = new List<ZaposlenikDAO>();
            ZaposlenikDAO zaposlenik1 = new ZaposlenikDAO("Marko",
"Peric", 0);
            ZaposlenikDAO zaposlenik2 = new ZaposlenikDAO("John",
"Smith", 2);

            zaposlenici.Add(zaposlenik1);
            zaposlenici.Add(zaposlenik2);
        }

        public List<ZaposlenikDAO> DohvatiSveZaposlenike()
        {
            return zaposlenici;
        }

        public ZaposlenikDAO DohvatiZaposlenika(int id)
        {
            return zaposlenici.Find(x => x.DohvatiIdentifikator() ==
id);
        }

        public void AzurirajZaposlenika(ZaposlenikDAO zaposlenik)
        {
```

```
            zaposlenici.Find(x => x ==
zaposlenik).PostaviIme(zaposlenik.DohvatiIme());
        }

        public void ObrisiZaposlenika(ZaposlenikDAO zaposlenik)
        {
            zaposlenici.RemoveAll(x => x == zaposlenik);
        }
    }
```

Primjena u praksi

Primjena ovog uzorka u praksi:

- Kada očekujemo frekventne promjene u sloju poslovne logike
- Ako migriramo bazu podataka na drugu tehnologiju.

Srodni uzorci

Data Access Object (skraćeno DAO) je uzorak dizajna koji je specifičan za Microsoftove tehnologije, a sličan je Gateway i Table Data Gateway uzorku od kojih ćemo kasnije u tekstu obraditi Table Data Gateway jer se nadograđuje na Gateway i vrlo su slični pa nema potrebe obrađivati oba. [14]

3.4. Record Set

Svrha

Svrha ovog uzorka dizajna je omogućiti pristup podatcima koji su spremljeni u obliku tablica, u memoriji računala.

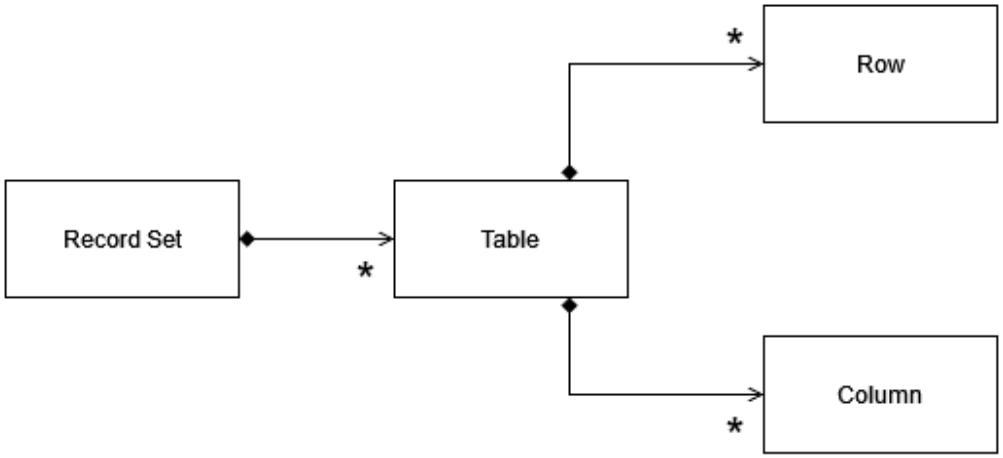
Motivacija

Ovaj uzorak dizajna je uzorak koji mi kao programeri u većini slučajeva nećemo implementirati u praksi jer on predstavlja prezentaciju tabličnih podataka u memoriji same tehnologije koju koristimo pa je taj dio već određen od strane proizvođača, ali proizvođač nam pruža strukture podataka kojima možemo koristiti taj mehanizam te je važno da znamo pozadinu implementacije samog uzorka da bismo što bolje i efikasnije koristili tehnologiju.

Primjena

Ovaj uzorak dizajna je preporučljivo koristiti kada tehnologija i okruženje u kojem radimo pruža dobar i pouzdan način pristupu podataka koji se bazira na ovom uzorku. Možemo ga iskoristiti da bismo brzo dohvatili podatke iz baze podataka i da ih nastavimo koristiti u takvom obliku što je jako pogodno ako imamo grafičko sučelje gdje također moramo prikazati podatke u takvom obliku stoga su modifikacije minimalne i brži smo u startu od drugih načina dohvaćanja podataka.

Struktura



Slika 8. Struktura Record Set uzorka dizajna (vlastita izrada prema [12])

Struktura uzorka dizajna Record Set odgovara strukturi tablice relacijske baze podataka te ona ima stupce i retke. Cilj ove strukture je omogućiti tipove podataka koji će odgovarati takvoj strukturi pa tako će Record Set biti kolekcija tablica u relacijskoj shemi te će se pomoću objekta Table moći pristupati pojedinim tablicama i nadalje kolekcijama redaka (Row) i stupaca (Column).

Sudionici

- Record Set
    - Sadrži kolekciju Table objekata.
- Table
    - Sadrži kolekcije objekata redaka Row i stupaca Column.

- Row
    - Sadrži kolekciju redaka.
- Column
    - Sadrži kolekciju stupaca.

Kolaboracije

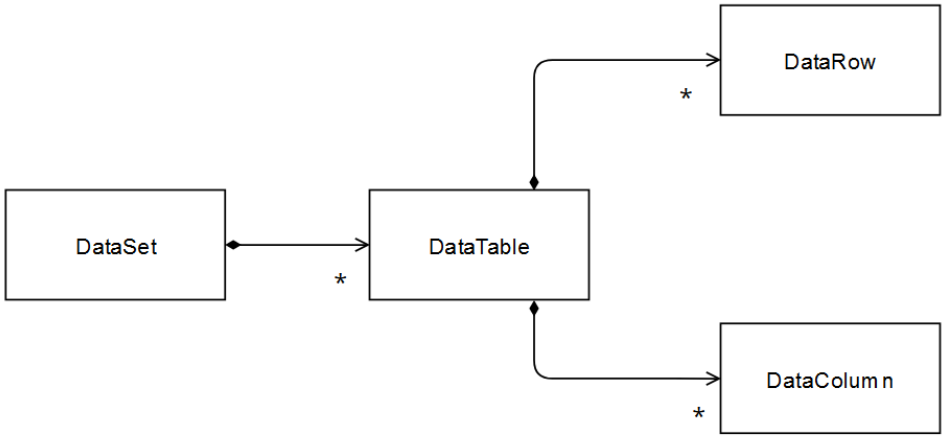
Klijent pomoću Record Set-a pristupa tablicama pohranjenim u tom objektu u obliku kolekcije te nadalje za pojedinu tablicu može vidjeti njezine stupce i retke isto tako u obliku kolekcije objekata.

Važnost

Ovaj uzorak dizajna nam pruža prednosti korištenja postojeće tehnologije okruženja kako bi podatcima pristupali na intuitivan i dobro poznat način, u obliku strukture tablice relacijske baze podataka. Može se koristiti za spremanje podataka iz baze podataka i s time olakšati pristup podatcima. Također ga nalazimo kod grafičkih sučelja, odnosno kod okruženja koja pružaju alate za dizajn grafičkih sučelja gdje se komponente organiziraju po strukturi ovog uzorka i omogućavaju strukturiran i učinkovit pristup svojstvima tih komponenata. Takvu strukturu na primjer možemo pronaći kod Windows Forms-a u okviru Microsoftove tehnologije.

Primjena u praksi

Na sljedećoj slici je prikazana arhitektura DataSet-a i objekata koji on sadrži:



Slika 9: Struktura DataSet-a kod ADO.NET tehnologije (vlastita izrada prema [9])

Ova struktura odgovara strukturi Record Set uzorka dizajna pa je pogodna za objašnjavanje uzorka na konkretnom primjeru. Možemo vidjeti da objekti koje DataSet klasa sadrži odgovaraju tablici u bazi podataka te njezinom retku i stupcu kojima možemo preko ove tehnologije pristupati. Ako oduzmemo iz nazivlja riječ „Data“ možemo vidjeti da se radi o strukturi Record Set uzorka dizajna, gdje imamo Record Set, Table, Row i Column. Koji predstavljaju pristup podataka u obliku tablice koja ima retke i stupce, kao što je to u relacijskim bazama podataka.

Srodni uzorci

Upotreba ovog uzorka može biti u kombinaciji sa Table Module uzorkom podatka koji ćemo obraditi u sljedećem poglavlju, a izvršava se na način da se poslovna logika organizira oko Table Module uzorka gdje će biti smještena manipulacija nad podatcima i onda se koristi Record Set da bi se izvuklo podatke iz baze podataka te na kraju pohranilo promjene. [12]

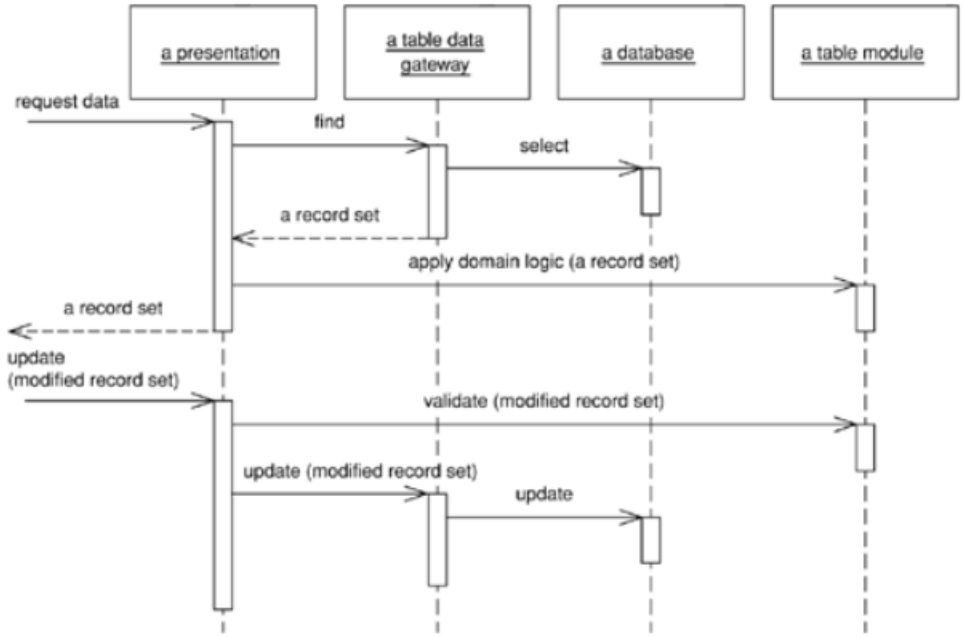
3.5. Table Module

Svrha

Table Module organizira poslovnu logiku oko jedne klase za svaku tablicu u bazi podataka i jedna instanca klase sadrži metode koje će vršiti manipulaciju nad podatcima.

Motivacija

Ključna zadaća klase koju ćemo stvoriti prema ovom uzorku dizajna jest primjena poslovne logike na podatke koji će biti dohvaćeni iz baze podataka. Jedna klasa odgovara jednoj tablici u bazi podataka. Na slici ispod vidimo dijagram slijeda koji opisuje kako klasa Table Module uzorka komunicira sa prezentacijskim slojem, Table Data Gateway klasom i slojem pristupa podatcima.



Slika 10.: Interakcije između slojeva i klase Table Module (vlastita izrada prema [12])

Možemo vidjeti da Table Data Gateway služi za dohvaćanje zapisa iz baze podataka, a prezentacijski sloj to prikazuje korisniku. Nakon toga kada je potrebna primjena poslovne logike nad zapisima koji su dohvaćeni, pozivaju se metode Table Module klase. Kasnije, kod ažuriranja podataka, se zapisi vraćaju natrag Table Module klasi na validaciju da li su dobro uneseni.

Primjena

Ovaj uzorak dizajna koristiti ćemo kada trebamo klasu koja će sadržavati metode poslovne logike vezane za zapise za jednu tablicu u bazi podataka.

Struktura

Struktura ovog uzorka je jednostavna, imamo jednu klasu koja sadrži metode poslovne logike.

Sudionici

- Table Module
    - sadrži metode poslovne logike

Kolaboracije

Kolaboracija ovog uzorka s klijentskim klasama je u takvom odnosu da klase klijenti šalju zapise instanci Table Module klase koja svojim metodama te podatke obrađuje u okviru poslovne logike.

Važnost

Table module radi u kombinaciji sa Record Set uzorkom daje korisnu kombinaciju za približavanje samih struktura podataka poslovnoj logici. Kako bismo dobili podatke, koristiti ćemo neki od pristupa bazi podataka i napuniti ćemo RecordSet, odnosno u ovom slučaju DataSet ako pričamo o .NET tehnologiji te ćemo ga koristiti u kombinaciji sa Table Module-om da bismo prosljeđivali podatke na obradu. Ovaj pristup se također može ulančati tako da više Table Module uzoraka prosljeđuju jedan drugom DataSet te pojedinačno vrše svaki svoju operaciju koja je već u pojedinom slučaju potrebna.

No negativne strane ovog uzorka su da ne daje potpunu fleksibilnost kad se poslovna logika zakomplicira. Što znači da ne podržava direktne odnose između instanci i ne radi dobro s polimorfizmima. Kada su objekti u poslovnoj logici i tablice u bazi podataka dosta slični, bolje je koristiti Domain Model uzorak dizajna koji koristi Active Record uzorak za podatke, a Table Module radi bolje kada su dijelovi aplikacije bazirani na tablično orijentiranoj strukturi podataka kao što je Record Set ili DataSet kod .NET-a. Recimo kod Jave će više prevladavati prva kombinacija, a kod .NET-a druga.

Primjena u praksi

Primjena ovog uzorka u praksi:

- Odvajanje poslovne logike u zasebnu klasu
- Izrada jednostavne poslovne logike za podatke koji bi odgovarali jednoj tablici u bazi podataka
- Može se dobro kombinirati s drugim uzorcima koji dohvaćaju zapise iz baze podataka ili nekog drugo izvora

Srodni uzorci

Transaction Script, Domain Model.

3.6. Table Data Gateway

Svrha

Ovaj uzorak rješava problem pristupa jednoj tablici u bazi podataka i vršenja svih upita koje želimo nad tom tablicom.

Motivacija

Table Data Gateway sadrži sav programski kôd za pristup jednoj baze podataka ili pogledu. Upiti se spremaju u metode koji se šalju u SQL obliku prema bazi podataka. Može sadržavati CRUD operacije i još neke složenije upite filtriranja.

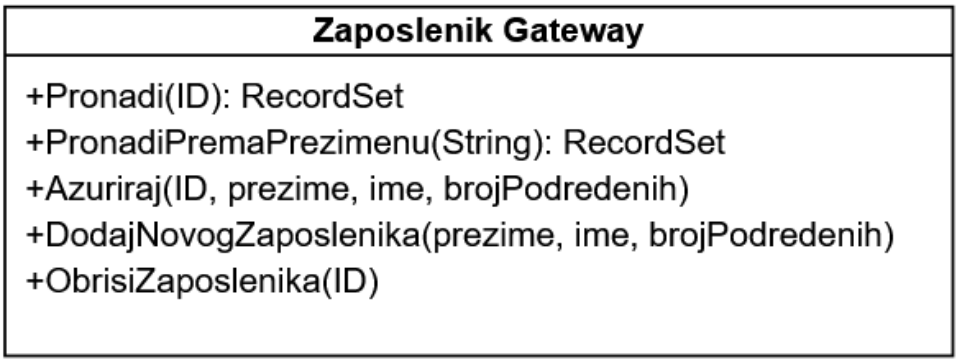
Primjena

Bitno je za svaku metodu kod objekta kojeg radimo prema ovom uzorku da one sadrže sve parametre koji će se slati na bazu podataka.

Ovo je jedan od najjednostavnijih pristupa bazi podataka pa ga je stoga dobro koristiti jer je pristup podatcima jako lagan i čitljiv.

Struktura

Na sljedećoj slici možemo vidjeti strukturu objekta prema Table Data Gateway uzorku dizajna:



Slika 11: Struktura klase prema Table Data Gateway uzorku dizajna (vlastita izrada)

Kao što vidimo ova skica objekta se ne razlikuje sa sučeljem kod Data Access Object uzorka dizajna te je ovaj uzorak, možemo reći, dio tog uzorka nadograđen s još nekim mehanizmima. Ti mehanizmi zapravo nadograđuju ono što nedostaje ovom uzorku dizajna, a

to je da dodaju strukturu podatka u koju će se spremati redci dohvaćeni iz tablice u bazi zbog toga što to u ovom slučaju nije skroz definirano. Mi smo ovdje stavili za primjer RecordSet što je dobro kod korištenja ADO.NET-a jer imamo DataSet na raspolaganju. Kod ostalih tehnologija osim to može stvoriti problem jer nije idealno da objekt koji je pohranjen u memoriji zna mnogo o sučelju prema bazi podataka.

Sudionici

- Table Data Gateway
    - Ova klasa služi za komunikaciju s bazom podataka.

Kolaboracije

Klasa Table Data Gateway služi klijentima tako da svojim metodama pruža pristup bazi podataka odnosno dohvaća zapise i po potrebi ažurirane zapise sprema te također briše zapise iz baze podataka.

Važnost

Glavna prednost je stvaranje metoda koje primaju parametre koji će se iskoristiti u upitima prema bazi podataka. Ovaj uzorak dizajna nam omogućuje izostavljanje SQL upita u drugim dijelovima aplikacije što pridonosi smanjenju kompleksnosti aplikacije odnosno boljoj strukturi i smanjenju količine programskog kôda. Dobro je što možemo napraviti klasu prema ovome uzorku za svaku tablicu u bazi podataka, a možemo i napraviti samo jednu klasu koja će raditi na svim tablicama što je jako korisno u smanjenju programskog koda i samog vremena razvoja aplikacije.

Primjena u praksi

Kad pričamo o korištenju ovih uzoraka dizajna, generalno ako uvidimo potrebu za Gateway uzorkom dizajna, koji predstavlja najjednostavniji pristup bazi pomoću sučelja, onda trebamo odabrati točno onu vrstu koja nam treba. Ako trebamo direktan pristup redcima, onda uzimamo Row Data Gateway, ako trebamo cijele tablice i izvlačenje većeg broja redaka, onda biramo Table Data Gateway. Table Data Gateway se također može koristiti sa Table Module uzorkom dizajna gdje podatci koji se vraćaju iz baze mogu biti iz pogleda što smanjuje odvojenost našeg kôda od baze podataka. Ako koristimo Domain Model onda možemo postaviti Table Data Gateway da nam vraća objekt prigodan objektu u poslovnom logici. Ovdje zbog toga imamo visoku povezanost između objekata domene i samog Gateway-a što nam može stvoriti probleme kod skaliranja, ali ovu opciju ne možemo isključiti jer može biti korisna.

Kod najjednostavnijih aplikacija možemo imati jedan Table Data Gateway koji će sadržavati upite za sve tablice u bazi podataka, to se jako brzo može zakomplicirati, ali ako je predvidivo da će nam aplikacija cijelo vrijeme ostati jednostavna, nije loša opcija.

Srodni uzorci

Ovaj uzorak dizajna vuče svoju ideju iz Gateway uzorka dizajna, a sličan je Data Access Object uzorku. Postoji još i Row Data Gateway za kojeg po samom imenu možemo reći da se radi o direktnom pristupu retcima tablice u bazi podataka pojedinačno. Koncept je jako sličan pa ga nećemo posebno obrađivati.

3.7. Transaction Script

Ovo je uzorak dizajna koji pripada vrsti uzoraka dizajna koji se bave tematikom poslovne logike, ali ga je važno spomenuti jer se ovdje spominje pristup podatcima i kontakt između prezentacijskog sloja i sloja za pristup podatcima. Aplikacije se kod većine poslovnih aplikacija mogu svesti na niz transakcija koje se obavljaju unutar nekih procesa. Primjer za to može biti upit na bazu podataka u svrhu obavljanja prikaza sadržaja direktno na ekran korisniku. Važno je napomenuti da se u jednoj transakciji obavlja samo jedna procedura tako da će svaka zasebna transakcija imati svoj Transaction Script uzorak dizajna posebno. Naglasak je na jednostavnosti i brzini izvođenja.

Svrha

Ovaj uzorak dizajna organizira poslovnu logiku na način da postoji jedna procedura za svaki zahtjev iz prezentacijskog dijela.

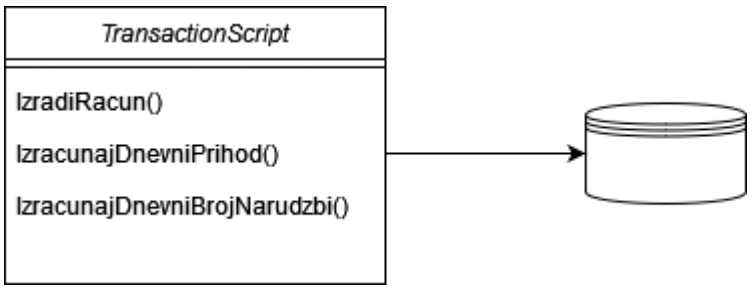
Motivacija

Kod ovog uzorka naglasak je na transakcijama koje izvršavaju poslovnu logiku. Svaka transakcija i njena poslovna logika će biti smještena u pojedinu metodu. One mogu biti smještene zajedno u jednu klasu na temelju domene koju obrađuju te se pozivati po potrebi.

Primjena

- Najčešća primjena je da imamo više transakcija u jednoj klasi gdje se svaka klasa bavi domenom koju transakcije u njoj obrađuju.
- Kada imamo više-dretveni pristup ovo će olakšati da možemo paziti na vrijeme odvijanja unutar aplikacije i pravovremeno pokrenuti određenu transakcijsku skriptu.
- U okviru sloja za pristup podatcima ćemo imati jednu transakciju za svaki kontakt s bazom podataka ili za svaku transakciju koja se izvršava u samoj bazi podataka.

Struktura



Slika 12. Struktura Transaction Script uzorka dizajna

Struktura ovog uzorka dizajna se temelji na tome da se napravi transakcija u obliku metode ili više njih u zasebnoj klasi. Metoda će izvoditi neku poslovnu logiku nad podatcima koje dohvaća iz baze podataka i na kraju će vratiti rezultat obrade podataka. Trebamo ju staviti na prikladno mjesto pa je stoga pogodno razdvojiti transakcijske skripte prezentacijskog sloja i one podatkovnog sloja.

Sudionici

- Transaction Script
  - Klasa koja sadržava metodu/e koja sadrži određenu transakciju.

Kolaboracije

Klasa Transaction Script daje klijentima mogućnost izvođenja transakcija pomoću svojih javnih metoda.

Važnost

Kod ovog uzorka dizajna trebamo paziti na mogućnost duplikacije koda jer je lako moguće da će širenjem aplikacije nastajati skripte koje obavljaju identičan posao, a samo su potrebne na nekom drugom mjestu pa treba razmisliti da li je ovo pogodan način za rješenje ovog problema kod samog planiranja. Prednost ovog uzorka dizajna jest u brzini izvođenja naspram uvođenja sloja uzorka programske logike (odnosno Domain Model-a), ali kad se poslovna logika zakomplicira, poželjno je uvesti zaseban sloj koji će se brinuti da sve funkcionira zbog toga što je jednostavno moćniji alat u tome slučaju jer ne zahtjeva da logika bude jednostavna naspram ovog uzorka dizajna. Negativna strana ovog pristupa je u tome što je jako teško kasnije ponovno iskoristiti kod i vrlo vjerojatno ćemo ga trebati refaktorirati u Domain Model u trenutku kad poslovna logika postane previše kompleksna i teško održiva.

Implementacija

Implementacija ovog uzorka će izgledati ovako:

```
public class TransactionScript
{
    public void IzradiRacun()
    {
        //  Dohvati narudžbe iz baze podataka
        //  Iteriraj kroz narudžbe
        //  Izračunaj konačnu cijenu i uračunaj PDV
        //  Izradi račun
        //  Pohrani račun u bazu podataka
        //  Označi narudžbu kao spremnu za isporuku
    }
}
```

Klasa TransactionScript će imati jednu metodu koja će odrađivati zadaću izrade računa za sve pristigle narudžbe. Možemo vidjeti slijed kojim se odvijaju pojedine akcije nad podatcima i što je na kraju finalni produkt, a to je izrada računa i oznaka da je narudžba spremna za isporuku.

Primjena u praksi

- Stvaranje transakcija koje obrađuju jednostavniju poslovnu logiku.

Srodni uzorci

Domain Model, Table Module, Service Layer.

3.8. Repository

Svrha

Ovaj uzorak stvara vezu između sloja poslovne logike i sloja za pristup podatcima, predstavljajući kolekciju objekata u memoriji.

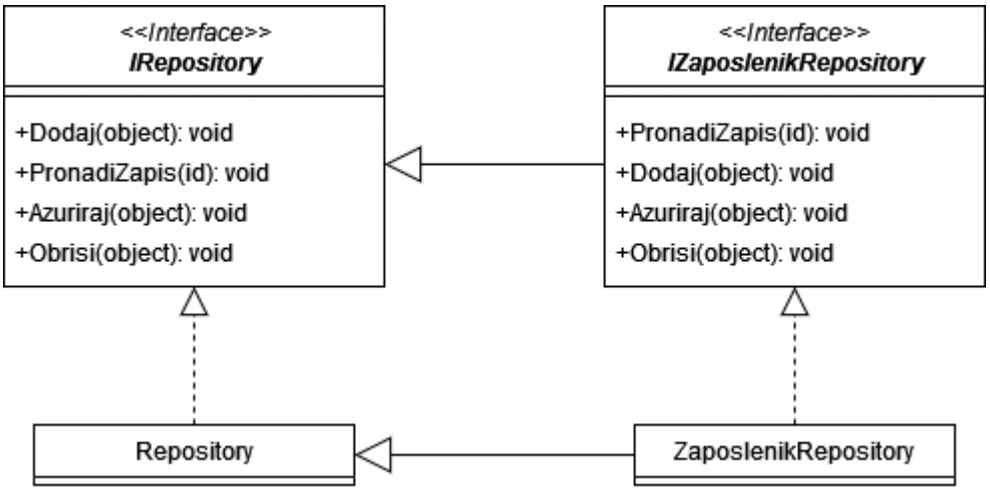
Motivacija

Aplikacija sa kompleksnom poslovnom logikom će zahtijevati da se odvoji sloj za pristup podataka i sloj poslovne logike. Kod ovog uzorka ćemo imati sučelje koje će sadržavati CRUD aplikacije za koje će se slati upiti na bazu podataka. To sučelje možemo još i proširiti s metodama specifičnim za neki objekt, tako što će novo sučelje vezano za konkretni objekt naslijediti generičko sučelje. Na taj način osiguravamo da se prethodne metode prenesu i njihove funkcionalnosti prenesu i mogu dalje koristiti. Ovaj uzorak dizajna nam omogućava da prethodno dohvaćene zapise iz baze podataka uređujemo, dodajemo nove zapise, brišemo zapise i također vršimo neke specifičnije upite koje ćemo definirati u zasebnim metodama.

Primjena

Što se tiče negativnih strana primjene ovog uzorka, često se kod Entity Framework-a misli da je Repository uzorak dizajna već implementiran u samoj tehnologiji u obliku DbSet-a. Oni odgovaraju strukturi samog Repository uzorka, ali problem je u tome što ovaj uzorak mora biti neovisan o tehnologiji. Kada se koristi samo DbSet, kada bi se u budućnosti promijenila tehnologija morala bi se ponovno programirati cijela logika toga dijela jer bi se programski kôd odnosio specifično za DbSet tehnologiju. Zato je dobra praksa koristiti DbSet za dohvaćanje zapisa i općenito komunikaciju s bazom u okviru samog Repository uzorka, a ne da on služi kao zamjena za njega.

Struktura



Slika 13. Struktura Repository uzorka (vlastita izrada)

Repository uzorak dizajna se koristi sa idejom više uzoraka dizajna, od kojih je najsličniji Query Object. Query Object se sastoji od objekta koji kreira upit na bazu podataka i sprema ga u metodu i takva predefinirana metoda u sebe prima parametre upita te se tako stvara SQL upit koji se šalje na bazu. Repository uzorak dizajna je kompleksnija verzija tog uzorka dizajna. Ovaj uzorak dizajna je zapravo jednostavno sučelje. Klase klijenti kreiraju objekt s kriterijima koje žele da se vrati pomoću upita, a Repository im pruža metode upita u koje će ulaziti ti parametri i vraća rezultat u obliku istog objekta kao što je zatraženo u upitu. Nakon toga pokrećemo metodu unutar same klase repozitorija koja će usporediti objekte u memoriji sa samim kriterijem i vratiti objekte koji odgovaraju upitu. Ovdje je važno naglasiti da se vraćaju isti objekti pa je dalje s njime lako raditi u sferi poslovne logike.

Kod repozitorija imamo sučelje IRepository koje sadrži CRUD metode koje se koriste nad objektima koji su pohranjeni u memoriji, a prethodno dohvaćeni iz baze podataka. Klasa Repository predstavlja implementaciju sučelja u kojoj se realizira dohvaćanje konteksta pristupa prema bazi podataka, dohvaćaju se zapisi iz baze koji se pohranjuju u kolekciju objekata. Sučelje klase ZaposlenikRepository će naslijediti naše generičko IRepository sučelje i time omogućiti dodavanje dodatnih metoda sa specifičnijim upitima vezanim za tu klasu. ZaposlenikRepository klasa nasljeđuje od Repository te implementira logiku pristupa bazi i pohranjivanju zapisa u kolekciju koja će se dalje koristiti.

Sudionici

- IRepository sučelje
    - Sučelje koje u sebi sadrži generičke metode za rad s kolekcijama podataka.

- Repository klasa
  - Zadaća joj je da implementira IRepository sučelje, dodaje pristup zapisima u bazi podataka i pohranjuje ih da bi se mogli koristiti u obliku kolekcije objekata.
- IZaposlenikRepository sučelje
  - Služi za iskorištavanje naslijeđenih metoda od sučelja i nadopunjavanje sučelja IRepository.
- ZaposlenikRepository klasa
  - Implementira IZaposlenikRepository sučelje i nasljeđuje Repository klasu.

Kolaboracije

Sučelje omogućava nasljeđivanje svojih metoda klasama koje ga trebaju koristiti, u ovom slučaju je to Repository klasa. Klasa Repository te metode nasljeđuje te nadopunjuje svojim, za objekt specifičnim metodama, koje kasnije klijenti mogu koristiti za vršenje odgovarajućih operacija nad objektima.

Važnost

Korištenjem sučelja omogućava se lako nasljeđivanje metoda za manipulaciju nad objektima. Neovisan je o tehnologiji pristupa baze podataka pa je stoga pogodan za korištenje u slučaju čestih promjena pristupa bazama podataka jer se ne mora ponovno mijenjati cijela logika, nego samo konkretne komponente koje su zadužene za pristup podatcima. Glavna prednost korištenja ovog uzorka je smanjenje duplikacije kôda u aplikaciji zbog toga što svaka klasa kojoj je potrebna manipulacija s objektima može naslijediti Repository sučelje koje će mu pružiti već napravljene metode. One se neće morati izrađivati ponovno za svaki objekt. To jako olakšava čitljivost koda i pozitivno utječe na brzinu izrade aplikacije.

Implementacija

Prvo ćemo napraviti implementaciju Repository uzorka dizajna u obliku sučelja:

```
public interface IRepository<T> where T : EntityBase
{
    T GetById(int id);
    IEnumerable<T> List();
```

```
        IEnumerable<T> List(Expression<Func<T, bool>> predicate);
        void Add(T entity);
        void Delete(T entity);
        void Edit(T entity);
    }
    public abstract class EntityBase
    {
       public int Id { get; protected set; }
    }
```

Slijedi implementacija sučelja Repository:

```
public class Repository<T> : IRepository<T> where T : EntityBase
{
    private readonly ApplicationDbContext _dbContext;

    public Repository(ApplicationDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public virtual T GetById(int id)
    {
        return _dbContext.Set<T>().Find(id);
    }

    public virtual IEnumerable<T> List()
    {
        return _dbContext.Set<T>().AsEnumerable();
    }

    public virtual IEnumerable<T>
List(System.Linq.Expressions.Expression<Func<T, bool>> predicate)
    {
        return _dbContext.Set<T>()
```

```
                .Where(predicate)
                .AsEnumerable();
        }

        public void Insert(T entity)
        {
            _dbContext.Set<T>().Add(entity);
            _dbContext.SaveChanges();
        }

        public void Update(T entity)
        {
            _dbContext.Entry(entity).State = EntityState.Modified;
            _dbContext.SaveChanges();
        }

        public void Delete(T entity)
        {
            _dbContext.Set<T>().Remove(entity);
            _dbContext.SaveChanges();
        }
    }
```

Oba primjera implementacije su preuzeta sa [15]. Možemo primijetiti da smo spremanje promjena implementirali kod svake metode zasebno, a mogli smo to napraviti i u odvojenoj metodi koja će nakon što napravimo sve promjene biti pozvana da ih spremi. Možemo koristiti oba načina ovisno o zahtjevima naše aplikacije. Također dobro je za tu svrhu koristiti Unit of Work uzorak dizajna kako bismo još bolje razvojili slojeve radi bolje održivosti koda i veće fleksibilnosti kod izvršavanja promjena u budućnosti.

Primjena u praksi

Ovaj pristup može u velikoj mjeri uvećati čitljivost koda i lakoću slaganja kompleksnijih upita prema bazi podataka. Najveća korist ovog pristupa u praksi će biti kada imamo jako kompleksnu programsku logiku i puno upita pa ih je u tom slučaju dobro sve staviti u jedan repozitorij kako kôd ne bismo pisali ponovno za svaku klasu kojoj je potreban pristup bazi.

Srodni uzorci

Ovaj uzorak se može koristiti zajedno sa Unit of Work uzorkom na način da se logika kreiranja objekta, praćenja promjena i spremanja objekta implementira pomoću njega umjesto spremanja promjena kao što smo naveli kod Repository uzorka.

3.9. Unit of Work

Kada dohvaćamo i unosimo podatke u baze podataka, bitno je paziti da se promjene prate da bi se kasnije mogle izvršiti nad samom bazom podataka. Recimo da mijenjamo i dodajemo neke retke, također brišemo i ažuriramo retke. Na kraju kada želimo završiti s našim proizvoljnim procesom koji već trebamo odraditi, možemo pohraniti promjene u bazu podataka.

Svrha

Omogućiti kolekciju objekata nad kojima će se vršiti poslovne transakcije i koordinirati s evidentiranjem i pohranom promjena nad tom kolekcijom.

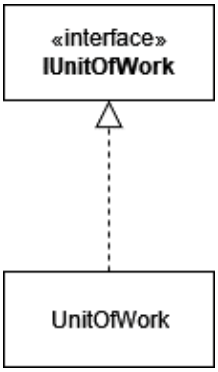
Motivacija

Kod ovog uzorka dizajna potrebno nam je slično kao i kod Repository uzorka dizajna jedno sučelje i jedna klasa koja će implementirati to sučelje. Važno je napomenuti da se uzorak dizajna Unit of Work koristi u kombinaciji sa Repository sučeljem pa ćemo ovdje koristiti neke od komponenata koje smo imali kod tog uzorka dizajna. Kod Unit of Work sučelja, za njegova svojstva (engl. properties) ćemo staviti sva ona konkretna sučelja koja smo napravili kod Repository patterna odnosno koja ga nasljeđuju. Tako bismo ovdje imali IZaposlenikRepository. Preko tog svojstva ćemo moći pristupiti metodama tog sučelja i vršiti operacije nad objektima kolekcije koju ćemo dohvatiti kasnije kod implementacije ovog sučelja. Sučelje će također sadržavati metodu koja će pohranjivati promjene u bazu podataka. Druga komponenta ovog uzorka je klasa koja implementira prethodno objašnjeno sučelje. Ona će implementirati sučelje pa je stoga u njoj implementiran pristup s bazom te u njen konstruktor možemo staviti dohvaćanje podataka iz baze koji će se koristiti u suradnji s Repository uzorkom. Na ovaj način smo osigurali odvojen pristup bazi od same poslovne logike.

Primjena

Ovaj uzorak dizajna se koristi u kombinaciji sa Repository uzorkom dizajna da bi se osiguralo spremanje i praćenje promjena nad kolekcijom objekata koju Repository uzorak modificira.

Struktura



Slika 14: Skica strukture Unit of Work uzorka (vlastita izrada)

U pozadini struktura ovog uzorka dizajna se sastoji od objekta koji prati izmjene koje mi radimo i zapisuje ih u privremenu memoriju te kad je potrebno primijeniti promjene, čitaju se svi zapisi promjena i primjenjuju se na bazi podataka. Kada je potrebno primijeniti promjene, otvara se transakcija za koju Unit of Work također brine da se izvede u potpunosti. Programeri nikada explicitno ne zovu ovakve metode nego se cijela procedura odvija u pozadini same tehnologije u kojoj radimo. Kako bi Unit of Work mogao znati koje objekte treba pratiti, postoji više pristupa koji to omogućavaju:

- Registracija kod pozivatelja (engl. Caller registration)
- Registracija kod objekta (engl. Object registration)
- Unit of work controller

Registracija kod pozivatelja prati promjene koje se događaju kod pristupa klijenta objektu, klijent mora znati i sjetiti se da registrira da su se vršile promjene nad objektom.

Registracija kod objekta podrazumijeva da se logika same registracije smjesti direktno na objekt i kada se pristupa objektu okida se pojedina registracija ovisno o akciji koja se radi. Recimo učitavanje objekta iz baze znači da je objekt čist i da nema promjena na njemu. Kada se set metode pokrenu, možemo reći da je objekt prljav i tako dalje. Kod ovog slučaja Unit of Work uzorak dizajna se treba smjestiti ili direktno u sam objekt, ili ga se treba proslijediti objektu s kojim radimo i onda pozivati registraciju određenih radnji. Treća opcija je da programer stavlja manualno pozivanje metoda registracije na određena mjesta gdje je to pogodno.

Kod Unit of Work kontrolera, svako čitanje prolazi kroz kontroler i za svaki objekt se postavlja registracija da je čist. Svaki put kada se dohvaća objekt njegovo svježe stanje se

sprema u privremenu memoriju i uspoređuje se sa stanjem u bazi pri samoj primjeni promjena. Na taj način se omogućuje ažuriranje promjena prilikom same primjene promjena, odnosno prije samog čina primjene što je jako korisno jer možemo odabrati koje promjene želimo primijeniti.

Sudionici

- IUnitOfWork
  - Sučelje koje implementira metodu pohrane podataka i svojstva tipa Repository sučelja.
- UnitOfWork
  - Implementira IUnitOfWork sučelje.

Kolaboracije

UnitOfWork implementira sučelje IUnitOfWork.

Važnost

- Podiže stupanj apstrakcije između poslovne logike i sloja za pristup podatcima.
- Poboljšava održivost, fleksibilnost i pogodan je za testiranje.
- Smanjuje duplikaciju koda unatoč korištenju više sučelja i klasa.

Implementacija

Slijedi programski kôd IUnitOfWork sučelja:

```
interface IUnitOfWork : IDisposable
{
    INekiRepozitorij Repozitorij { get; }
    INekiRepozitorij2 Repozitorij2 { get; }

    int SpremiPromjene();
}
```

Klasa koja implementira sučelje:

```
class UnitOfWork : IUnitOfWork
```

```
{
    private KontekstBaze kontekst;
    public UnitOfWork(KontekstBaze kontekst) //dohvaća se kontekst
    {
        this.kontekst = kontekst;
        Repozitorij = new INekiRepozitorij(this.kontekst);
        Repozitorij2 = new INekiRepozitorij2(this.kontekst);
    }

    public INekiRepozitorij Repozitorij
    {
        get;
        private set;
    }

    public INekiRepozitorij2 Repozitorij2
    {
        get;
        private set;
    }

    public void Dispose()
    {
        kontekst.Dispose();
    }

    public int SpremiPromjene()
    {
        kontekst.SaveChanges(); // spremanje promjena
    }
}
```

Primjena u praksi

Možemo vidjeti da je ovaj uzorak dizajna jako koristan u bilježenju promjena nad bazom podataka i općenito promjena. Važna stvar za istaknuti je da se ovim pristupom sve bilježenje promjena centralizira na jednom mjestu pa je lakše upravljati samim bilježenjem i vidjeti sve promjene na jednom mjestu, a ne trebamo previše brinuti da li će se promjene zabilježiti jer sustav to radi sam za nas u većini slučajeva.

Srodni uzorci

Ovaj pristup je također dobra podloga za građenje Optimistic Offline Lock i Optimistic Offline Lock uzoraka dizajna. [12]

4. Usporedba uzoraka dizajna

Tablica 4. Usporedba obrađenih uzoraka dizajna

Naziv uzorka	Prednosti uzorka	Nedostatci uzorka	Područje primjene
Singleton	Kontroliran pristup instanci klase. Smanjena količina kôda. Bolja čitljivost kôda.	Krši pravilo da jedna klasa mora implementirati samo jednu funkcionalnost. Nespretno korištenje kod dretava. Treba paziti da se ne kreira više instanci. Nemoguće je vršiti jedinične testove nad ovim uzorkom.	Stvaranje uzorka Facade. Dijeljenje podataka (konfiguracije). Za spremanje podataka u obliku sličnom cache memoriji.
Active Record	Pristup bazama podataka bez obzira na tehnologiju. Smanjena količina kôda.	Nije dobar za korištenje kod kompleksne poslovne logike.	Kada poslovna logika nije kompleksna te sadrži većinom CRUD operacije i operacije nad samo jednim zapisom u tablici baze podataka. Pogodan kod refaktoriranja Transaction Script uzorka.
Data Access Object	Omogućava slojevitost aplikacije. Dobra skalabilnost.	Moguća posljedica je propuštajuća apstrakcija i inverzija apstrakcije. Može doći do duplikacije kôda.	Pogodan kod kompleksne poslovne logike i kod njenih frekventnih promjena. Kada migriramo bazu podataka na neku drugu tehnologiju spremanja.
Record Set	Intuitivan pristup podatcima u obliku tablice u bazi podataka.	Ovisimo o tome da li je ovakav pristup implementiran od strane proizvođača tehnologije u kojoj radimo i o samim metodama tehnologije.	Može se koristiti u kombinaciji sa Table Module. Kada je prikladan pristup podatcima u obliku tablice u bazi podataka.

Table Module	Smanjenje količine kôda. Olakšana manipulacija nad podatcima vezano za jednu tablicu u bazi podatka.	Nije fleksibilan niti skalabilan ako se poslovna logika zakomplicira. Ne podržava direktne odnose između instanci i ne radi dobro s polimorfizmima.	Ovaj uzorak dizajna koristiti ćemo kada trebamo klasu koja će sadržavati metode poslovne logike vezane za zapise za jednu tablicu u bazi podataka. Dobar za korištenje u kombinaciji sa Record Set uzorkom.
Table Data Gateway	Smanjenje količine kôda. Jedan od najjednostavnijih pristupa bazama podatka. Pogodan kod CRUD operacija i filtriranja zapisa iz baze podatka. Omogućuje vršenje upita pomoću parametara.	Nije za samostalno korištenje kod kompleksne programske logike. Problemi kod skaliranja aplikacije.	Za izvlačenje podataka u obliku redaka ili stupaca. Može se koristiti sa Domain Model-om kada je programska logika kompleksna.
Transaction Script	Jednostavnost i brzina izvođenja su glavna prednost.	Mogućnost duplikacije kôda. Nije skalabilan. Loše odgovara na kompliciranje programske logike.	Najčešća primjena je da imamo više transakcija u jednoj klasi gdje se svaka klasa bavi domenom koju transakcije u njoj obrađuju. Koristi se kod jednostavne programske logike.
Repository	Omogućava slojevitost. Poveznica između sloja za pristup podatcima i sloja poslovne logike. Lakoća nasljeđivanja metoda. Neovisan o tehnologiji. Dobra čitljivost i brzina izvođenja kôda.	Uvođenje dodatne razine apstrakcije može biti nepotrebno kod jednostavnijih aplikacija i ovaj uzorak u tom slučaju stvara nepotrebno veliku količinu kôda.	Najveća korist ovog pristupa u praksi će biti kada imamo jako kompleksnu programsku logiku i puno upita pa ih je u tom slučaju dobro sve staviti u jedan repozitorij kako kôd ne bismo pisali ponovno za svaku klasu kojoj je potreban pristup bazi.

Unit of Work	Omogućava rad s kolekcijom objekata nad kojima možemo izvršavati i pohranjivati promjene. Poboljšava održivost, fleksibilnost. Pogodan za testiranje. Smanjuje duplikaciju kôda.	Nije pogodan za korištenje u jednostavnijim aplikacijama.	Bilježenje promjena nad kolekcijama kod Repository uzorka i općenito za bilježenje i izvršavanje promjena nad podatcima.

(vlastita izrada)

5. Refaktoriranje postojeće aplikacije

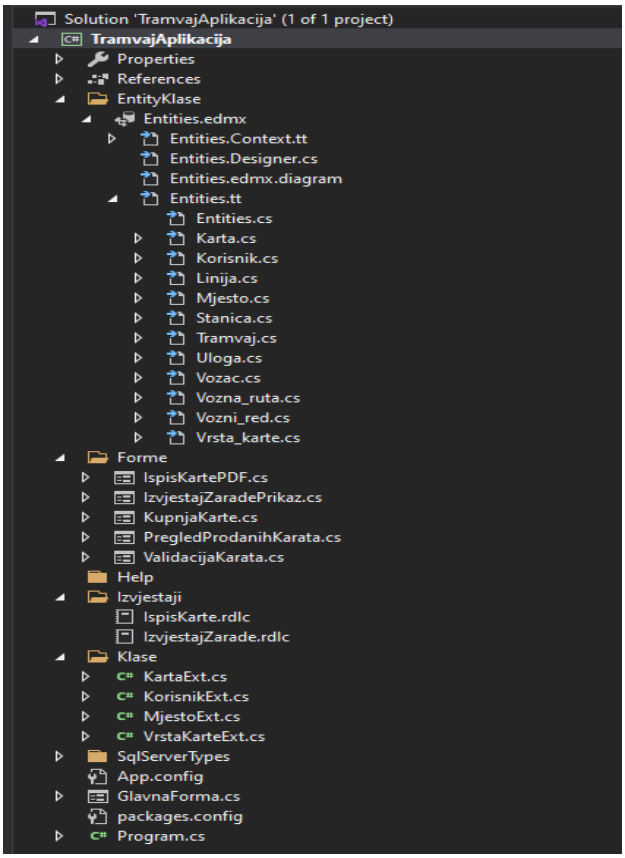
Refaktoriranje kôda je disciplinirana tehnika restrukturiranja postojećeg tijela kôda, a da se pritom ne mijenja njegovo vanjsko ponašanje. [16] Koristimo ju kako bismo poboljšali kvalitetu samog kôda i izbacili nepotrebne ponavljajuće dijelove kako bismo postigli lakše održavanje.

4.1. Opis aplikacije

Aplikacija na kojoj ćemo demonstrirati prethodno obrađene uzorke dizajna izrađena je u sklopu kolegija Programsko inženjerstvo te je dostupna na Git Hub platformi [17]. Aplikacija služi za potporu upravljanju i potporu pružanju usluga tramvajskog prijevoza te je izrađena u Microsoft .NET tehnologiji. WinForms se koristi za grafičko sučelje aplikacije i Entity Framework za pristup bazi. Za bazu podataka se koristi lokalna baza smještena na Microsoft SQL Server-u te je korišten Microsoft SQL Management Studio 2019. u kreiranju same baze i upravljanju serverom na koji je smještena. Zbog jednostavnosti demonstracije uzoraka dizajna uzeti ćemo dijelove aplikacije koji obrađuju sljedeće funkcionalnosti: kupnja i ispis karte u PDF formatu, izvještaj zarade, pregled prodanih karata i validacija karata.

4.2. Stanje aplikacije prije refaktoriranja

Prije refaktoriranja aplikacija se koristi Entity Framework-om za pristup bazi podataka te on predstavlja zaseban sloj pristupa podatcima. Prema strukturi datoteka (klasa) u aplikaciji možemo vidjeti njezinu strukturu na sljedećoj slici:



Slika 15. Struktura projekta stare aplikacije

U mapi EntityKlase su smješteni elementi Entity Frameworka koji se sastoje od generiranih klasa prema tablicama koje su definirane u bazi podataka. Taj dio nam već u velikoj mjeri olakšava samo programiranje jer ne moramo pisati poseban kôd za klase svih tablica u bazi podataka. Mapa forme sadrži sve WinForms forme koje se koriste za u opisu navedene funkcionalnosti. Sam kôd poslovne logike je smješten zajedno sa kôdom grafičkog sučelja i možemo reći da trenutno poslovna logika nije odvojena u zaseban sloj jer nije bilo potrebe kod ovakve jednostavnije aplikacije. Još je preostao jedan dio aplikacije, a to su parcijalne klase u mapi Klase. One proširuju postojeće klase generirane od strane Entity Frameworka te im dodaju određena svojstva kako bi olakšali prikaz podatcima u grafičkom sučelju. Opisati ćemo pristup podatcima na klasi KupnjaKarte jer ona sadrži najviše operacija koje se odnose na bazu podataka. Nećemo obrađivati sve klase zbog toga što je pristup bazi jednak. Slijedi kôd klase KupnjaKarte koji je zadužen za inicijalizaciju konteksta:

```
public partial class KupnjaKarte : Form
{
    Entities entities = new Entities();
```

„Entities“ klasa služi za stvaranje konteksta kojim se izvršava komunikacija s bazom te promjene nad kolekcijama podataka dohvaćenim iz baze. Ona nasljeđuje klasu DbContext koja je sastavna klasa Entity Frameworka te ona pruža sve potrebne mehanizme za spajanje na bazu podataka i izvršavanje upita.

Trenutno se u aplikaciji kontekst „entities“ direktno u kôdu grafičkog sučelja i nema strukture koja bi dala potporu za elegantnije zvanje samog konteksta. Također nemamo kontrolu nad time koliko će se instanci samog konteksta moći napraviti što može dovesti do potencijalnih problema pri spremanju podataka za različite entitete ukoliko je otvoreno više od jednog konteksta. Kako bismo omogućili elegantniji pristup podatcima i regulirali količinu stvorenih konteksta, odnosno osigurali da se stvara samo jedan, trebamo pronaći neki od uzoraka dizajna koji će nam dati te mogućnosti.

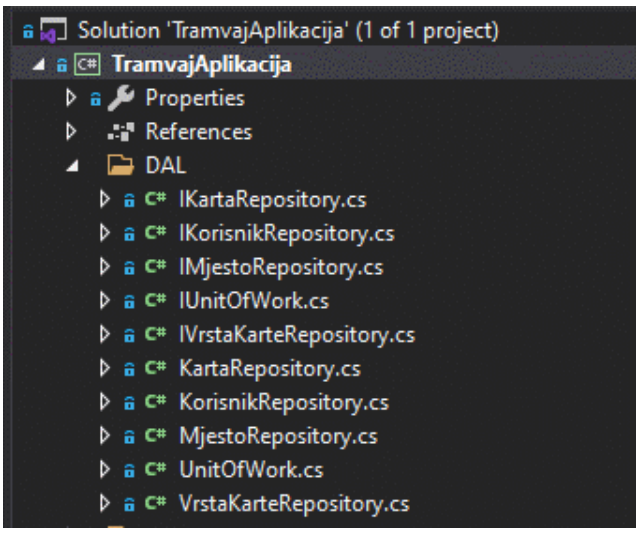
Kod klase KupnjaKarte podatci se učitavaju lokalno što možemo vidjeti u sljedećem kôdu:

```
private void UcitajPodatkeVrsteKarte()
{
    entities.Vrsta_karte.Load();
    var query = from v in entities.Vrsta_karte.Local select v.naziv;
    query.ToList().ForEach(v => uxVrstaKarte.Items.Add(v));
}

private void UcitajPodatkeMjesto()
{
    entities.Mjesto.Load();
    var query = from m in entities.Mjesto.Local select m.naziv;
    query.ToList().ForEach(m => uxMjesto.Items.Add(m));

   }
```

Funkcija Load koja se poziva nam omogućuje da podatke spremimo lokalno i koristimo ih bez slanja upita za svaku promjenu te sam kontakt s bazom podataka odgodimo do trenutka kada je potrebno pohraniti promjene. Taj pristup nam konkretno kod ove klase neće biti toliko bitan jer se radi o manjem broju podataka, samo jednom stupcu tablice. Kod prve funkcije je to naziv svih vrsta karata koje možemo kupiti, a kod druge funkcije je to naziv svih mjesta. Možemo reći da smo ovdje dohvatili sve podatke samo da bismo izvršili jedan upit što u budućnosti može biti dosta zahtjevno po memoriju ako bude jako puno zapisa u respektivnim tablicama.

4.3. Refaktoriranje aplikacije

Kod refaktoriranja aplikacije kako bismo riješili prethodno navedene probleme napraviti ćemo Repository uzorak dizajna u kombinaciji sa Unit of Work uzorkom. Repository uzorak će nam poslužiti za pospremanje svih CRUD operacija za kolekcije koje se dohvaćaju pomoću konteksta, a Unit of Work će nam poslužiti za omogućavanje stvaranja samo jednog konteksta i pohranjivanje svih promjena nad kontekstom kada odradimo određena ažuriranja, brisanja ili dodavanja novih zapisa. Izostaviti ćemo lokalno spremanje pomoću funkcije Load zbog potencijalnog povećanja broja zapisa u budućnosti. Posljedično ovaj pristup će omogućiti elegantniji pristup samom kontekstu i spremanju podataka u bazu podataka i dodatnom odvajanju pristupa podatcima u zaseban sloj.



Slika 16. Repository i Unit of Work klase

Na slici iznad možemo vidjeti sve klase koje smo dodali u projekt, a implementiraju sučelja Repository uzorka i klase koje implementiraju sučelje te na kraju Unit of Work klasu. Slijedi kôd klase sučelja IKartaRepository, klase KartaRepository i UnitOfWork klase. Nećemo ovdje obraditi kôd sviju klasa jer je sama struktura uzoraka ista za svaku klasu.

Slijedi kôd klase sučelja IKartaRepository:

```
public interface IKartaRepository
{
    IEnumerable<Karta> DohvatiSve();
    Karta DohvatiPremaId(int KartaID);
    void Dodaj(Karta karta);
```

```
        void Azuriraj(Karta karta);
        void Obrisi(Karta karta);
        void Save();
       }
```

Nadalje, kôd klase KartaRepository:

```
    public class KartaRepository : IKartaRepository
    {
        private readonly Entities _context;


        public KartaRepository(Entities entities)
        {
            _context = entities;
        }

        public Karta DohvatiPrazniObjekt()
        {
            return _context.Set<Karta>().Create(); //vraća novi objekt s kreiranim
proxy navigacijskim svojstvima
        }

        public Karta DohvatiPremaId(int KartaID)
        {
            return _context.Karta.Find(KartaID);
        }

        public IEnumerable<Karta> DohvatiSve()
        {
            return _context.Karta;
        }

        public void Azuriraj(Karta karta)
        {
            _context.Entry(karta).State = EntityState.Modified;
        }

        public void Dodaj(Karta karta)
        {
            _context.Karta.Add(karta);
        }

        public void Obrisi(Karta karta)
        {
            _context.Karta.Remove(karta);
        }

        public void Save()
        {
            _context.SaveChanges();
        }
       }
```

I kôd klase IUnitOfWork:

```
    public interface IUnitOfWork : IDisposable
```

```
{
    void CreateTransaction();
    void Commit();
    void Rollback();
    void Save();
       }
```

I na kraju kôd klase UnitOfWork:

```
  public class UnitOfWork : IUnitOfWork
{
    private KartaRepository _kartaRepo;
    private KorisnikRepository _korisnikRepo;
    private VrstaKarteRepository _vrstaKarteRepo;
    private MjestoRepository _mjestoRepo;

    private Entities _entities;
    private DbContextTransaction _objTran;
    private string _errorMessage = string.Empty;
    private bool disposedValue;

    public UnitOfWork(Entities entities)
    {
        _entities = entities;
    }

    public KartaRepository KartaRepo
    {
        get
        {
            if(_kartaRepo == null)
            {
                _kartaRepo = new KartaRepository(_entities);
            }
            return _kartaRepo;
        }
    }

    public KorisnikRepository KorisnikRepo
    {
        get
        {
            if (_korisnikRepo == null)
            {
                _korisnikRepo = new KorisnikRepository(_entities);
            }
            return _korisnikRepo;
        }
    }

    public VrstaKarteRepository VrstaKarteRepo
    {
        get
        {
            if (_vrstaKarteRepo == null)
            {
                _vrstaKarteRepo = new VrstaKarteRepository(_entities);
            }
            return _vrstaKarteRepo;
```

```
            }
        }

        public MjestoRepository MjestoRepo
        {
            get
            {
                if (_mjestoRepo == null)
                {
                    _mjestoRepo = new MjestoRepository(_entities);
                }
                return _mjestoRepo;
            }
        }

        public void Commit()
        {
            _objTran.Commit();
        }

        public void CreateTransaction()
        {
            _objTran = _entities.Database.BeginTransaction();
        }

        public void Rollback()
        {
            _objTran.Rollback();
            _objTran.Dispose();
        }

        public void Save()
        {
            try
            {
                _entities.SaveChanges();
            }
            catch (DbEntityValidationException dbEx)
            {
                foreach (var validationErrors in dbEx.EntityValidationErrors)
                    foreach (var validationError in validationErrors.ValidationErrors)
                        _errorMessage += string.Format("Property: {0} Error: {1}",
validationError.PropertyName, validationError.ErrorMessage) + Environment.NewLine;
                throw new Exception(_errorMessage, dbEx);
            }
        }

        protected virtual void Dispose(bool disposing)
        {
            if (!disposedValue)
            {
                if (disposing)
                {
                    _entities.Dispose();
                }
                disposedValue = true;
            }
        }

        public void Dispose()
        {
```

```
            Dispose(disposing: true);
            GC.SuppressFinalize(this);
        }
    }
```

U programskom kôdu možemo vidjeti da smo pomoću Repository uzorka omogućili CRUD operacije nad entitetom Karta te smo također omogućili metodu stvaranja praznog objekta. Kreiranje praznog objekta smo omogućili da nam Entity Framework generira proxy klase za svako navigacijsko svojstvo koje je kreirano od strane samog Entity-a jer nam klasa karta sadrži tri takva svojstva. Inače se navigacijska svojstva prilikom ručnog kreiranja objekata ne bi također vezala na kontekst i automatski generirala. To je jedna specifičnost koja je vezana za tehnologiju pa ju je važno napomenuti.

Prethodno spomenute dvije funkcije smo izbacili iz upotrebe zbog jednostavnosti kôda i sada možemo vidjeti na primjeru korištenje uzoraka u funkciji koja se izvršava kod učitavanja forme KupnjaKarata:

```
        private void KupnjaKarata_Load(object sender, EventArgs e)
        {

            //ucitavanje svih vrsta karata u padajuci izbornik
            _unitOfWork.VrstaKarteRepo.DohvatiSve().Select(v =>
v.naziv).ToList().ForEach(m => uxVrstaKarte.Items.Add(m));
            //ucitavanje svih mjesta u padajuci izbornik
            _unitOfWork.MjestoRepo.DohvatiSve().Select(v =>
v.naziv).ToList().ForEach(m => uxMjesto.Items.Add(m));

            uxDatum.Text = DateTime.Now.Date.ToShortDateString();
            uxVrijeme.Text = DateTime.Now.ToString("HH:mm");
               }
```

Možemo primijetiti da smo izostavili funkciju Load i da se upit izvršava u jednoj liniji kôda te se precizno vide pozivi objekta Unit of Work uzorka i konkretnog repozitorija za vrstu karte te se oni dohvaćaju i na njima se vrše upiti.

U sljedećem kôdu možemo vidjeti kreiranje transakcije kod UnitOfWork uzorka dizajna i odrađivanje spremanja nove karte u repozitorij te izvršavanje pohrane na bazu podataka metodom Commit. U budućnosti kada poslovna logika bude kompleksnija možemo dodati u Rollback funkciju u slučaju da želimo da se promjene ne spreme uslijed greške ili nekog specifičnog uvjeta. Slijedi kôd:

```
        private void uxKupiKartu_Click(object sender, EventArgs e)
        {
            if (uxMjesto.Text == "")
            {
                MessageBox.Show("Odaberite mjesto!");
                return;
            }
            if (uxVrstaKarte.Text == "")
```

```
            {
                MessageBox.Show("Odaberite vrstu karte!");
                return;
            }
            Vrsta_karte _vrsta = _unitOfWork.VrstaKarteRepo.DohvatiSve().Where(v =>
v.naziv == uxVrstaKarte.Text).First();
            int _mjestoID = _unitOfWork.MjestoRepo.DohvatiSve().Where(m => m.naziv ==
uxMjesto.Text).Select(m => m.id_mjesto).FirstOrDefault();
            _unitOfWork.CreateTransaction();
            var _karta = _unitOfWork.KartaRepo.DohvatiPrazniObjekt();

            _karta.valjanost = 1;
            _karta.id_vrsta_karte = _vrsta.id_vrsta_karte;
            _karta.id_mjesto = _mjestoID;
            _karta.id_korisnik = 1; // hardkodiran korisnik ID-a 1 jer nemamo funkciju
login-a, za lakše testiranje
            _karta.datum_kupnje = DateTime.Now;

            _unitOfWork.KartaRepo.Dodaj(_karta);
            _unitOfWork.KartaRepo.Save();
            _unitOfWork.Commit();

            //ispis karte u PDF format
            IspisKartePDF ispis = new
IspisKartePDF(_unitOfWork.KartaRepo.DohvatiPremaId(_karta.id_karta));
            ispis.ShowDialog();
        }
```

Ovim pristupom smo sakrili podatkovni sloj od ostalih slojeva i tako postigli dodatnu slojevitost. Iako se Entity Framework može gledati kao sloj za pristup podatcima, ovo je dodatan način kako možemo slojevitost podići na višu razinu. Uz pomoć Unit of Work uzorka smo osigurali kreiranje samo jedne instance objekta konteksta da bismo izbjegli potencijalne probleme kod spremanja podataka ako oni dolaze iz više instanci konteksta. Kako će se aplikacija dalje razvijati, biti će korisno imati dodatan sloj jer ćemo samim time moći lagano vršiti promjene nad njime.

5. Zaključak

S napretkom tehnologije, dolaskom relacijskih baza podataka i potrebom za kompleksnom poslovnom logikom te slojevitim aplikacijama bilo je potrebno izraditi odgovarajuće alate koji će biti dobar temelj i vodilja u kreiranju sloja pristupa podatcima. Potreba za pohranom znanja kojeg su programeri naučili pri izgradnji aplikacija se odrazila na nastanak uzoraka dizajna. Uzorci dizajna nam pružaju dobar i koristan izvor informacija o rješenjima problema ne samo kod podatkovnog sloja nego i kod drugih slojeva aplikacije i drugih konkretnih problema. Omogućava se ponovno iskorištavanje programskog kôda i time se skraćuje sam proces programiranja i iskorištavaju se već prethodno ispitane prakse koje će se koristiti još dugi niz godina.

U ovom završnom radu je obrađena slojevita arhitektura aplikacije uz glavni fokus na troslojnu arhitekturu i sloj za pristup podatcima. Ponuđeni su alati kojim se kreira sloj za pohranu podataka kao što je ERA model i objašnjena je važnost relacijskih baza podataka te njihova trenutna popularnost i opseg korištenja u praksi. Dotaknuli smo se Microsoftovih tehnologija, konkretno platforme .NET i programskog jezika C# te njihovih specifičnih alata kao što je ADO.NET-a koji služi kao alat za pristup bazama podataka te su objašnjeni njegovi mehanizmi te različiti pristupi samim izvorima podataka. Nadalje, objasnili smo najpopularnije uzorke dizajna koji se koriste za izgradnju sloja za pristup podatcima i detaljno specificirali njihove mehanizme, svrhu te prednosti i s napomenama o specifičnostima za tehnologije u kojima se oni koriste. Također su objašnjene mane pojedinih uzoraka koje nam mogu prouzročiti probleme tijekom programiranja vezano za samu tehnologiju, ali i općenito. Na kraju se napravila obrada postojeće aplikacije, odnosno njeno refaktoriranje s pomoću dvaju odabranih uzoraka Repository i Unit of Work koji su se pokazali kao dobar kandidat za odvajanje sloja za pristup podatcima i pripremu istog za buduće promjene u smislu skalabilnosti. Repository uzorak smo iskoristili kako bismo u njega pospremili sve CRUD operacije koje Entity Framework izvršava nad bazom podataka i pomoću Unit of Work uzorka smo postigli kontrolu spremanja podataka i upravljanje nad stvaranjem konteksta koje će biti korisno u budućnosti.

Mislim da su uzorci dizajna jako korisni u izradi aplikacija jer nam pružaju strukturu, stabilnost i već provjeren „kostur“ kojeg možemo oblikovati kako bismo postigli potrebe za izgradnju naše aplikacije, ne samo kod sloja za pristup podatcima nego i općenito. Također mislim da omogućavaju standardizaciju pristupa te lakše korištenje unutar timova i efikasniju komunikaciju pri izgradnji pojedinih dijelova aplikacija što je jako bitan faktor kod programiranja.

Popis literature

[1] “DB-Engines Ranking - Complete Ranking.” https://db-engines.com/en/ranking. (pristupano: 11.07.2022.)

[2] Herberto Graca, “Layered Architecture”

https://herbertograca.com/2017/08/03/layered-architecture/ (pristupano 09.06.2022.).

[3] “NET Application Architecture: the Data Access Layer - Simple Talk.” https://www.red-gate.com/simple-talk/development/dotnet-development/net-application-architecture-the-data-access-layer/ (pristupano 14.06.2022.).

[4] Z. Stapić, „Arhitekture i pristupi u radu s podacima“, nastavni materijal na kolegiju Programsko inženjerstvo dostupan na platformi Moodle, Sveučilište u Zagrebu, Fakultet organizacije i informatike, Varaždin

[5] M. Maleković, K. Rabuzin, Uvod u baze podataka. Varaždin: Fakultet organizacije i informatike, 2016.

[6] B. Lutkevich, “What Is a Relational Database,” , 2021,

https://www.techtarget.com/searchdatamanagement/definition/relational-database (pristupano 16.06.2022.).

[7] Microsoft Application Architecture Guide, 2nd Edition, “Chapter 15: Designing Data Components”, 2009,

https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658119(v=pandp.10) (pristupano 16.06.2022.).

[8] Microsoft Docs, “Overview - ADO.NET.”,

https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview (pristupano 19.06.2022.).

[9] Microsoft Docs, “Architecture - ADO.NET”,

https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-architecture (pristupano 19.06.2022.).

[10] B. Hamilton, M. MacDonald, “ADO.NET in a Nutshell,” 2003.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[12] M. Fowler and D. Rice, Patterns of Enterprise Application Architecture. Addison-Wesley, 2003.

[13] Tutorials Point, „Data Access Object Pattern“,

https://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm (pristupano 25.06.2022.)

[14] GeeksforGeeks, “Data Access Object Pattern - GeeksforGeeks.”

https://www.geeksforgeeks.org/data-access-object-pattern/ (pristupano 13.07.2022.).

[15] DevIQ, „Repository Pattern“,
https://deviq.com/design-patterns/repository-pattern (pristupano 01.08.2022.)

[16] Martin Fowler „Refactoring“,

https://www.refactoring.com/ (pristupano 12.09.2022.)

[17] Dario Kulenović, Ante Novak, Vanja Kostanjevac, Niki Hercigonja, „Desktop aplikacija za potporu upravljanju i pružanju prijevoznih usluga“,

https://github.com/foivz/pi21-dkulenovi-vkostanje-nhercigon-anovak (pristupano 29.08.2022.)

Popis slika

Popis slika treba biti izrađen po uzoru na indeksirani sadržaj, te upućivati na broj stranice na kojoj se slika može pronaći.

Popis tablica

Popis tablica treba biti izrađen po uzoru na indeksirani sadržaj, te upućivati na broj stranice na kojoj se tablica može pronaći.

Prilozi

[1] Projekt praktičnog dijela završnog rada dostupan na GitHub platformi:

https://github.com/dkulenovic/Zavrsni-rad