



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ
САДУ



Ненад Зеленовић

**Иплементација комуникације између
клијента и сервера користећи Data Access
Layer**

ДИПЛОМСКИ РАД
- Основне академске студије –

Нови Сад, 2019.



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА


Редни број, РБР:			
Идентификациони број, ИБР:			
Тип документације, ТД:	Монографска публикација		
Тип записа, ТЗ:	Текстуални штампани документ/ ЦД		
Врста рада, ВР:	Дипломски рад		
Аутор, АУ:	Ненад Зеленовић		
Ментор, МН:	др Бранислав Атлагић		
Наслов рада, НР:	Имплементација комуникације између клијента и сервера користећи Data Access Layer		
Језик публикације, ЈП:	Српски (латиница)		
Језик извода, ЈИ:	Српски/енглески		
Земља публикавања, ЗП:	Србија		
Уже географско подручје, УГП:	Војводина		
Година, ГО:	2019.		
Издавач, ИЗ:	Ауторски репринт		
Место и адреса, МА:	Факултет техничких наука (ФТН), Д. Обрадовића 6, 21000 Нови Сад		
Физички опис рада, ФО: (поглавља/страна/ цитата/табела/слика/графика/прилога)			
Научна област, НО:	Електротехника и рачунарство		
Научна дисциплина, НД:	Примењене рачунарске науке и информатика		
Предметна одредница/Кључне речи, ПО:	Трослојна архитектура, апстракција, скалабилност		
УДК			
Чува се, ЧУ:	Библиотека ФТН, Д. Обрадовића 6, 21000 Нови Сад		
Важна напомена, ВН:			
Извод, ИЗ:			
Датум прихватања теме, ДП:			
Датум одбране, ДО:			
Чланови комисије, КО:	Члан:		Потпис ментора
	Члан:		
	Члан, ментор:	Др. Бранислав Атлагић, доцент	



UNIVERSITY OF NOVI SAD • FACULTY OF TECHNICAL SCIENCES
21000 NOVI SAD, Dositej Obradović Square 6

KEY WORDS DOCUMENTATION

Accession number, ANO :			
Identification number, INO :			
Document type, DT :	Monographic publication		
Type of record, TR :	Textual material, printed/CD		
Contents code, CC :	Bachelor thesis		
Author, AU :	Nenad Zelenović		
Mentor, MN :	Branislav Atlagić, Ph. D.		
Title, TI :	Implementing communication between client and server using Data Access Layer		
Language of text, LT :	Serbian		
Language of abstract, LA :	Serbian/English		
Country of publication, CP :	Serbia		
Locality of publication, LP :	Vojvodina		
Publication year, PY :	2019.		
Publisher, PB :	Author's reprint		
Publication place, PP :	Faculty of Technical Sciences, D. Obradovića 6, 21000 Novi Sad		
Physical description, PD : (chapters/pages/ref./tables/pictures/graphs/appendixes)			
Scientific field, SF :	Electrical and computer engineering		
Scientific discipline, SD :	Applied computer science and informatics		
Subject/Keywords, S/KW :	3-tier architecture, abstraction, scalability		
UC			
Holding data, HD :	Library of Faculty of Technical Sciences, D. Obradovića 6, 21000 Novi Sad		
Note, N :			
Abstract, AB :			
Accepted by the Scientific Board on, ASB :			
Defended on, DE :			
Defended Board, DB :	Member:		Menthor's sign
	Member:		
	Member, Mentor:	Ph. D. Branislav Atlagić	

	УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Датум:
	ЗАДАТАК ЗА ИЗРАДУ ДИПЛОМСКОГ (BACHELOR) РАДА	Лист/Листова:

(Податке уноси предметни наставник - ментор)

Врста студија:	<input checked="" type="checkbox"/> Основне академске студије <input type="checkbox"/> Основне струковне студије
Студијски програм:	Електроенергетски софтверски инжењеринг
Руководилац студијског програма:	Др Драган Поповић, ред. проф.

Студент:	Ненад Зеленовић	Број индекса:	Е338/2014
Област:	Електротехника и рачунарство		
Ментор:	др Бранислав Атлагић		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ДИПЛОМСКИ (Bachelor) РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;
- литература

НАСЛОВ ДИПЛОМСКОГ (BACHELOR) РАДА:

IMPLEMENTASIJA KOMUNIKACIJE IZMEĐU KLIJENTA I SERVERA KORISTEĆI DATA ACCESS LAYER
--

ТЕКСТ ЗАДАТКА:

--

Руководилац студијског програма:	Ментор рада:

Примерак за: ☐ - Студента; ☐ - Ментора

SADRŽAJ

1.	UVOD	6
2.	OPIS REŠAVANOG PROBLEMA	7
3.	TEHNOLOGIJA I ALATI	10
3.1.	NET Framework	10
3.2.	Microsoft Visual Studio	10
3.3.	C#	10
3.4.	Windows Presentation Form (WPF)	10
3.5.	Windows Communication Foundation (WCF)	11
3.6.	NUnit	11
3.7.	OpenCover	11
3.8.	Model-View-ViewModel (MVVM)	11
3.9.	Dizajn paterni	12
3.9.1.	Command	12
3.9.2.	Strategy	13
3.9.3.	Flyweight	14
3.9.4.	Singleton	15
4.	OPIS REŠENJA PROBLEMA	16
4.1.	Klijent	16
4.2.	DAL	17
4.3.	Drajveri	22
4.4.	NUnit testiranje	24
5.	ZAKLJUČAK	26
6.	LITERATURA	27
7.	LISTA KORIŠĆENIH SKRAĆENICA	28
	Podaci o kandidatu	29

1. UVOD

Sistemske sistemi u prošlosti nisu imali odgovarajuću formalnu arhitekturu. Kod je bio grupisan u module, sa nejasno definisanim ulogama i obavezama između slojeva. Posledica ovakvih sistema je bila teška izmena i poprilično je teško bilo odrediti karakteristike bez potpunog poznavanja svake komponentne sistema.

Jedno od rešenja ovog problema se javlja u obliku slojevite arhitekture [1]. Komponente u okviru slojevite arhitekture su organizovane u formi horizontalnih slojeva, gde svaki sloj ima specifičnu ulogu. Nije jasno definisan broj slojeva koje sistem mora da implemenira, tako da postoje sistemi sa tri, četiri, pet... slojeva. Najčešće se javlja arhitektura sa četiri sloja (prezentacioni, sloj poslovne logike, sloj perzistencije podataka i sloj baze podataka). Konkretno za implementaciju ovog rada korišćena je troslojna arihitektura sa fokusom na razvijanju srednjeg sloja, gde su sloj poslovne logike i perzistencije podataka smešteni u jedan sloj.

Jedna od osnovnih karakteristika ovakve arhitekture je odvajanje brige (Separation of Concerns), gde svaka komponentna ima odgovarajući zadatak i ne vodi brigu o dešavanju i načinu rada druge komponentne. Tako na primer sloj poslovne logike ne sme da vodi brigu kako će sloj baze podataka neki podatak da skladišti ili kako će prezentacioni sloj te podatke da prezentuje.

Prednosti ovakvog sistema su mnogobrojne:

- *Fleksibilnost*
- *Apstrakcija*
- *Nezavisnost*
- *Lako testiranje i održavanje*
- *Skalabilnost*

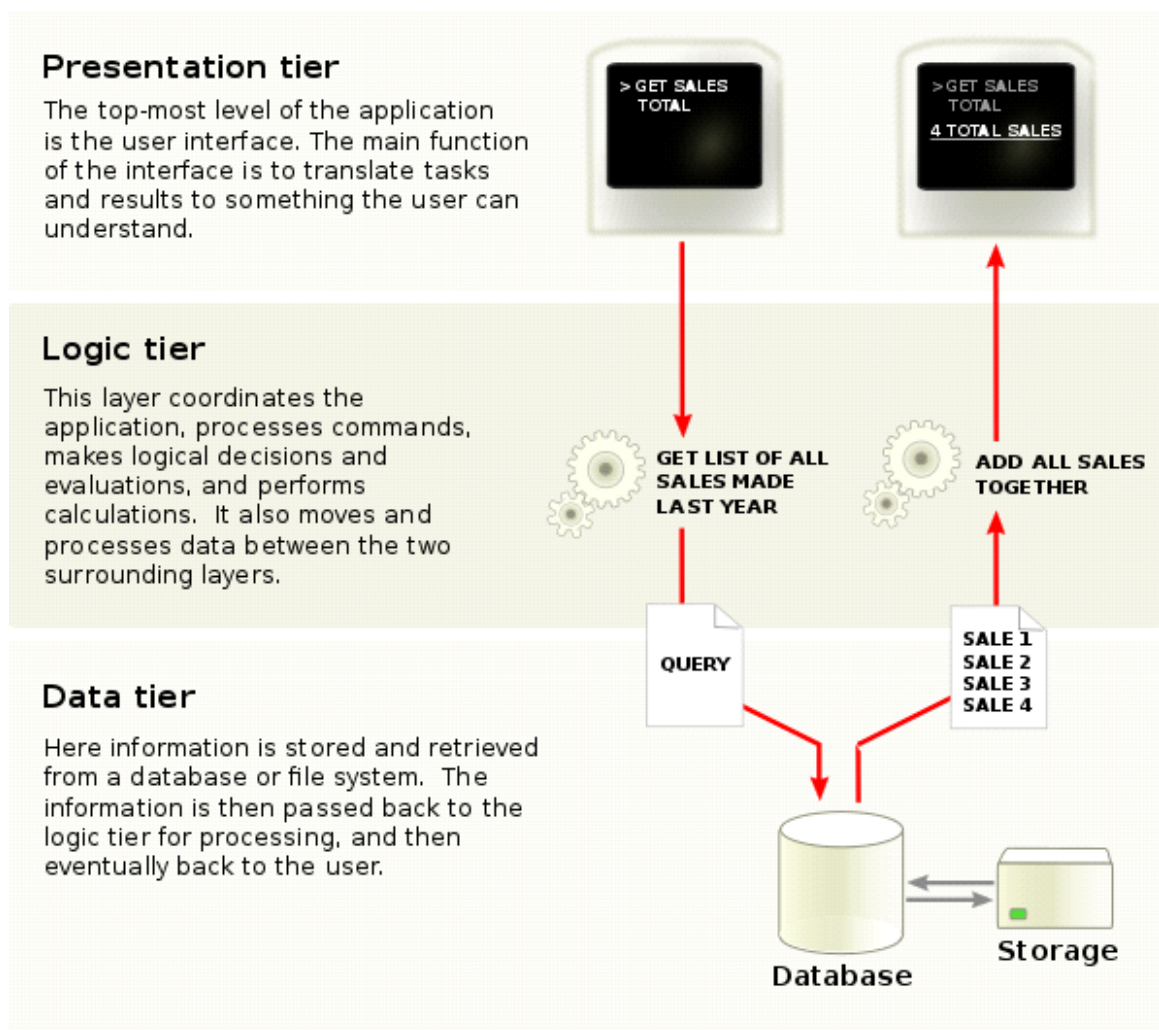
Neki nedostaci slojevite arhitekture su:

- *Kaskadne promene*
- *Povećanjem slojeva, smanjuju se performanse*
- *Nije namenjena jednostavnim sistemima*

2. OPIS REŠAVANOG PROBLEMA

Svrha rada je bila razvijanje aplikacije koja implementira troslojnu arhitekturu. Slojevi te arhitekture su:

- *Prezentacioni sloj (klijent)* - bavi se ispisom podataka korisniku, korišćenje odgovarajuće tehnologije za korisnički interfejs (WPF). To je ono što korisnik vidi.
- *Biznis logika (srednji sloj)* - on se bavi implementiranjem poslovne logike. Poslovnu logiku čine poslovni procesi i poslovne komponente. Na ovom sloju se vrše razne validacije podataka koje stižu od prezentacionog sloja, rukovanje izuzetcima, logovanje...
- *Sloj baze podataka* - zadnji sloj troslojne arhitekture, služi za izvršavanje operacije koja je zadata na klijentskoj strani. Ako se radi o bazi podataka, ti podaci se mogu i skladištiti.



Slika 2.1. Troslojna arhitektura

Najprostiji primer ovakve organizacije bi bilo sabiranje dva broja. *Prezentacioni sloj* bi morao da implementira odgovarajući grafički interfejs, odgovarajuća polja gde bi ti brojevi mogli biti uneti. Zatim *srednji sloj* treba da primi ta dva broja od *prezentacionog sloja*, da izvrši validaciju, i da te brojeve redirektuje ka odgovarajućem servisu (*sloj baze podataka*) koji bi te brojeve sabrao i vratio rezultat *srednjem sloju* i od *srednjeg sloja* kao *prezentacionom sloju*.

Glavni fokus je bio na razvijanju *biznis logike*. Implementiranje što više mogućeg nivoa apstrakcije, davanje mogućnost korisniku da kroz *prezentacioni sloj* pošalje podatak neograničavajućeg tipa kroz odgovarajući interfejs i da se taj podatak validira i redirektuje ka odgovarajućim servisima ili bazi podataka (*sloj baze podataka*). Podatak može biti tipa int, string, objekat klase, SQL upit... Tip ne sme da ograniči klijenta.

Biznis logika se sastoji od dva podsloja:

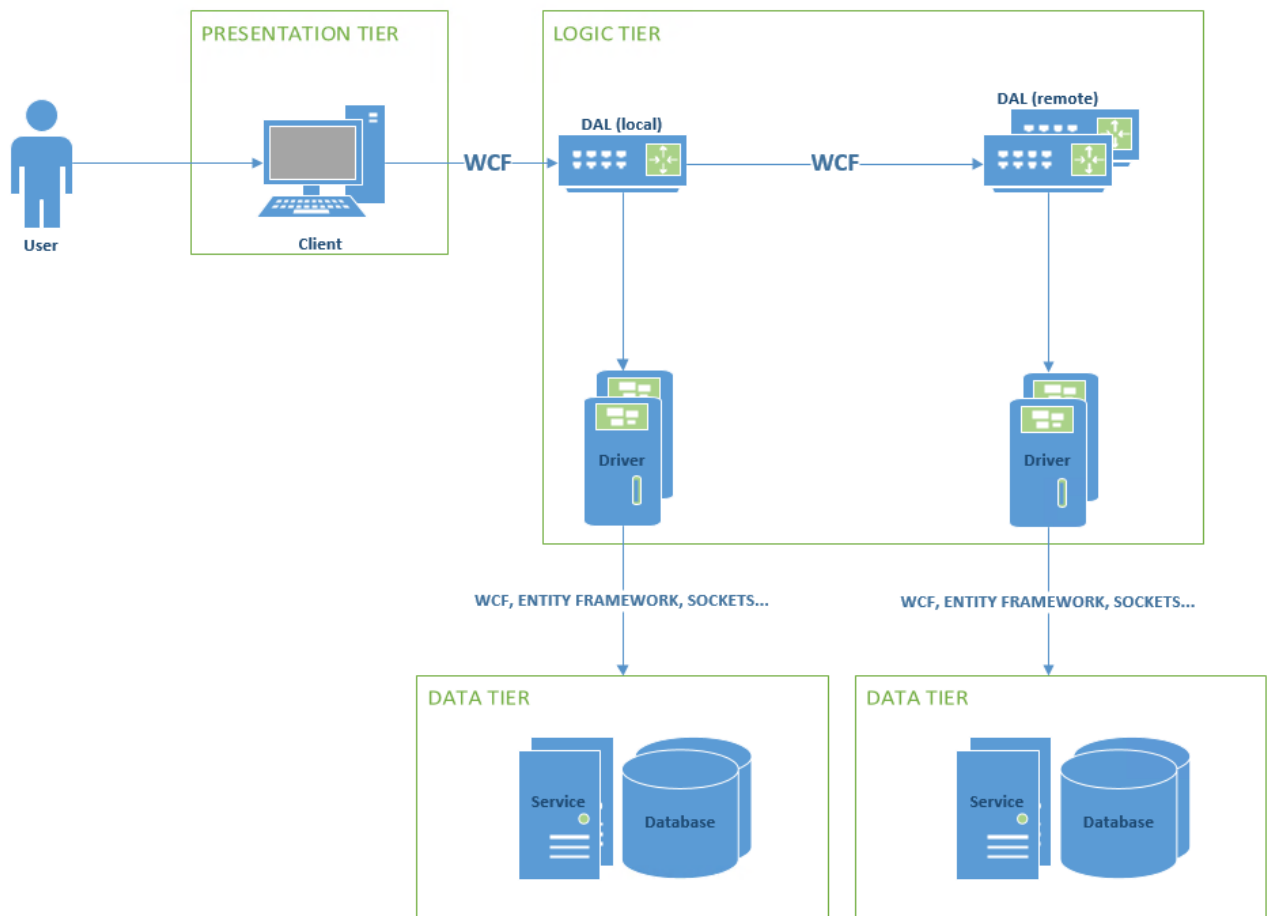
- *DAL*
- *Drivers*

DAL služi za primanje podataka od klijenta i njegov je zadatak da te podatke prosledi ka odgovarajućem *drajveru*. *DAL* se pokreće kao posebna aplikacija i nakon pokretanja biraju se *drajveri*, sa kojima *DAL* upravlja tj. može da poziva *drajvere* koje je izabrao i njima šalje podatke. *DAL* može da bude tipa: *local* ili *remote*. *Local DAL* je jedinstven i klijent sa njim direktno komunicira, dok *remote DAL* može da ima *n* instanci. Preko nego što klijent zatraži izvršenje operacije, *local DAL* mora da bude aktivan i moraju da budu izabrani *drajveri* tj. mora da se zada *local DAL*-u sa kojim *drajverima* rukuje, dok *remote DAL*-ovi su opcioni i njih ne moramo pokretati. Nakon podizanja *local DAL*-a tek tada klijent može da zatraži izvršenje operacije. Ako zadatu operaciju *local DAL* ne podržava tj. ne rukuje sa odgovarajućim *drajverom* on će pokušati da razgovara sa *remote DAL*-ovima i da vidim jel neko od njih podržava zadatu operaciju. Ako neki od *remote DAL*-ova podržava operaciju ona će biti izvršena i rezultat te operacije će biti vraćen klijentu, dok ako ni jedan *remote DAL* ne podržava operaciju tj. ne rukuje sa odgovarajućim *drajverom* onda će se klijentu ispisati greška sa porukom da treba se otvoriti novi *remote DAL*.

Drajver će primiti podatke od *DAL*-a i odraditi validaciju podataka i pozvati odgovarajući servis/bazu podataka da bi se ti podaci izvršili/skladištili. Za svaku operaciju koju klijentu pružimo, moramo imati i *drajver* za tu operaciju. Jer *drajver* zna na koji način da raspakuje i validira podatke i zna koji servis/bazu podataka da pozove.

Pozivi u sistemu rade na osnovu sihrone komunikacije.

Izgled sistema, možemo videti na slici 2.2.



Слика 2.2. - Дизајн система

3. TEHNOLOGIJA I ALATI

U ovom poglavlju pričaćemo o svim tehnologija i alatima koji su korišćeni za implementaciju rada.

3.1. NET Framework- Okruženje za razvoj sofvera, razvijano od strane Microsoft-a za Windows platforme. Uključuje veliku biblioteku klasa (Framework Class Library). Microsoft je sa razvojem .NET-a počeo ranih 1990-tih, pod nazivom Next Generation Windows Services. Početkom 2000-tih prva beta verzija .NET 1.0 je objavljena, a u avgustu 2000. u saradnji sa Intel-om i HP-om, Microsoft je počeo sa standardizacijom CLI-ja, koja će omogućiti izvršavanje različitih programskih jezika na različitim arhitekturama-platformama.

Programi se izvršavaju kroz softversko okruženje CLR, virtualnu mašinu koja sadrži: memory managment, exception handling, garbage collector... Omogućeno je korišćenje 25 programskih jezika od kojih su najpolularniji C#, C++ i VisualBasic. Jezici se, svaki preko svog kompajlera, kompajliraju u CIL među-jezik. Zatim, u zavisnosti od toga na kojoj se platformi izvršava, CLR kompajlira CIL u mašinski kod. Glavni razvojni alat je Visual Studio. [2]

3.2. Microsoft Visual Studio- predstavlja integrirano razvojno okruženje. Koristi se za razvoj računarskih programa za Windows, veb-stranica, aplikacija i usluga. Koristi Microsoftove platforme za razvoj raznih API-ja za Windows, Windows Forms, WPF. Program takođe sadrži alate poput dizajnera oblika koji se koristi za pravljenje aplikacija s grafičkim korisničkim interfejsom, veb-dizajnera, dizajnera klasa i dizajnera shema baza podataka. Visual Studio podržava različite programske jezike i dozvoljava uređivaču koda i debuggeru da podržava gotovo bilo koji programski jezik. Ugrađeni jezici su C, C++, VB.NET, C# i F#. Također podržava XML, HTML, JavaScript i CSS. [3]

3.3. C#- je objektno orijentisan programski jezik koji je razvio Microsoft početkom 21og veka. Reč je o jeziku opšte namene koji služi za pravljenje aplikacija u okviru .NET okruženja. Iako ne postoji dugo kao neki drugi programski jezici, C# je jedan od najpopularnijih jezika. C# se odlikuje velikim mogućnostima, jednostavnošću upotrebe i lakoćom usvajanja, zbog čega je danas jedan od najpopularnijih programskih jezika koji svoju primenu nalazi u velikim i malim kompanijama i u različitim oblastima. [4]

3.4. Windows Presentation Form (WPF)- je grafički podsistem za renderovanje korisničkog interfejsa u aplikacijama zasnovanim na Windows operativnim sistemima. Razvijen je od strane Microsoft-a. WPF koristi XAML, izveden od XML-a da definiše i poveže različite UI elemente. WPF aplikacije mogu biti razvijene kao samostalni desktop programi ili kao ugrađeni objekti u website stranicama. Ima za cilj da objedini niz zajedničkih interfejs elemenata, kao što su 2D/3D renderovanja, fiskirana i adaptivna dokumenta, tipografiju, vektorsku grafiku... [5]

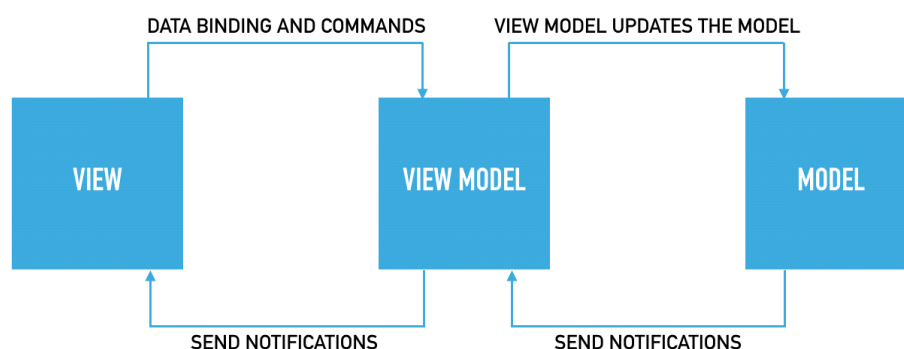
3.5. Windows Communication Foundation (WCF)- je servisno orijentisani model razmene poruka, koji omogućava programima da komuniciraju preko računarske mreže ili lokalno. *WCF* je alat koji u sebi uključuje set biblioteka razvijenih za distribuirano programiranje. [6]

3.6. NUnit- je framework za testiranje otvorenog koda. Zasniva se na unit testiranju, gde se zasebno testiraju pojedinačne celine koda programa kako bi se utvrdilo da li su te celine u potpunosti bez grešaka i spremne za upotrebu. Ovo okruženje pruža bogat skup tvrdnji kao statičkih metoda klase *Assert*. Ako tvrdnja ne uspe, pozvana metoda neće vratiti nikakvu vrednost i desiće se greška što dalje dovodi do toga da test u kome se nalazi ta tvrdnja neće biti uspešan. Unit test treba da ima jedan poziv određene metode klase *Assert* da bi bio što jednostavniji i da bi jedan unit test testirao tačno jedan scenario. [7]

3.7. OpenCover- je besplatan alat otvorenog izvornog koda koji služi za određivanje procenta pokrivenosti koda za .NET 2.0 i novije verzije koji se pokreće na .NETplatformama. Podržava pokrivenost sekvenci, pokrivenost grana kao i celokupnu pokrivenost koda. Bogat HTML korisnički interfejs rezultata *OpenCover*-a može da se vizualizuje pomoću *ReportGenerator*-a. [8]

3.8. Model-View-ViewModel (MVVM)- je patern koji razdvaja aplikaciju na više komponenti tako da svaka komponenta ima svoje specifične odgovornosti [9]. *MVVM* arhitektura je preporučena od strane Google-a kao jedan od najboljih načina strukture koda Android aplikacija. Pri korišćenju *MVVM* paterna kod aplikacije je razdvojen na tri dela:

- *View* - Ova sekcija sadrži klase (Aktivnosti i Fragmenti) koje su zadužene za prikaz interfejsa i prihvatanje akcija korisnika, nakon čega o tome obaveštava *ViewModel*
- *ViewModel* - Ova sekcija sadrži klase koje su zadužene za pristup podacima (Repository) i da obaveste *View* ukoliko dolazi do promena.
- *Model* - Ova sekcija sadrži klase zadužene za pristup raznim vrstama podataka (baza, webservice...) i da absraktuje takve izvore podataka kroz jedan API.



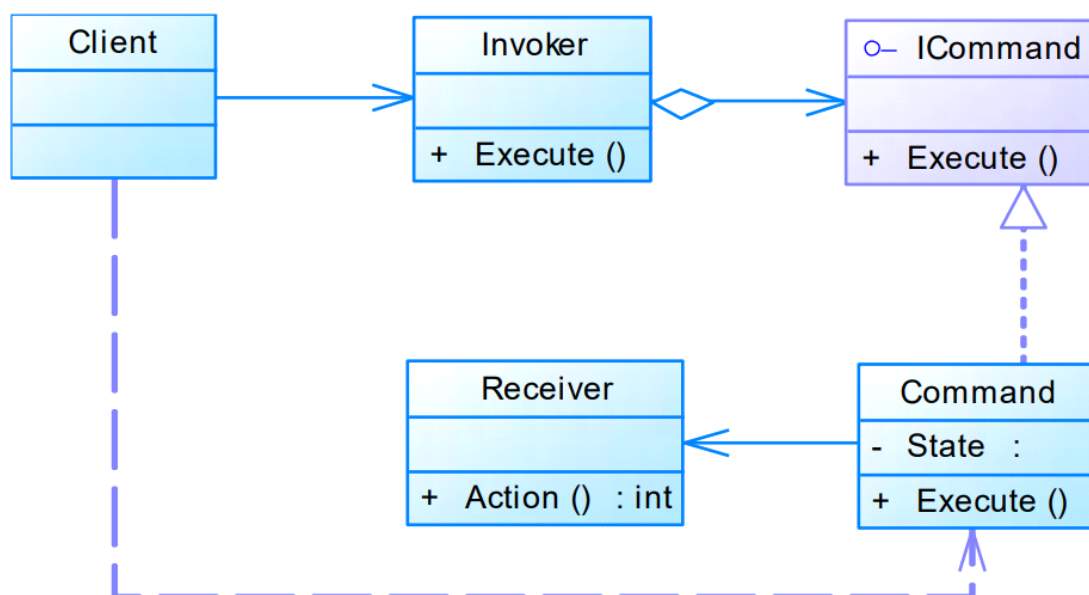
Slika 3.8.1. *MVVM* patern

3.9. Dizajn paterni - u softverskom inženjerstvu, *dizajn patern* ili *šablon* ili *obrasac* je opšte, ponovo upotrebljivo rešenje za česte probleme koji se sreću prilikom projektovanja softvera. *Dizajn patern* nije gotov dizajn koji se može direktno pretvoriti u izvorni kod. On služi samo kao opis ili šablon prilagođen da reši neki opširniji problem u posebnom kontekstu [10]. Postoje tri vrste *dizajn paterna*:

- *Strukturalni paterni* - bave se kompozicijom i obično predstavljaju različite načine za definisanje odnosa među objektima. Oni obezbeđuju da kada je neophodna promena u jednom delu sistema, ostatak sistema ne mora da se menja. Takođe pomažu da svaki deo sistema radi ono čemu je najbolje prilagođen. Neki od *strukturalnih paterna* su: Decorator, Facade, Flyweight, Adapter i Proxy.
- *Kreacioni paterni* - ovi paterni bave se kreacijom objekata, na način prilagođen određenoj primeni. Posebno su važni u situacijama u kojima bi uobičajen pristup kreiranju objekata doveo do povećanja kompleksnosti projekta. Neki od paterna koji spadaju u ovu grupu su: Constructor, Factory, Prototype, Singleton i Builder.
- *Bihevioralni paterni* - Ova grupa paterna tiče se poboljšanja komunikacije između različitih objekata u sistemu. Poznati primeri su: Strategy, Iterator, Mediator, Observer i Visitor.

Dizajn paterni koji su korišćeni u implementaciji rada biće nabrojavi i objašnjeni u tekstu ispod.

3.9.1. Command - *Command* patern kreira distancu između klijenata koji zahtevaju operacije i objekata koji ih izvršavaju. Patern je izrazito višestran. On može da podrži: slanje zahteva ka različitim objektima, smeštanje zahteva u redove, logovanje i odbijanje zahteva. [11]



Slika 3.9.1.1. UML dijagram za *Command patern*

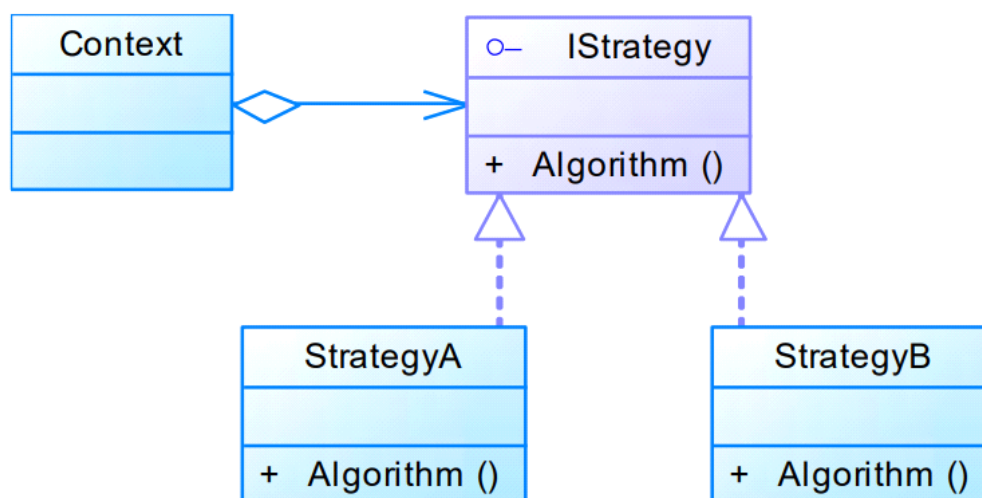
U okviru UML dijagrama možemo videti više učesnika čije su uloge sledeće:

- *Client* - kreira i izvršava komande
- *ICommand* - interfejs koji navodi operacije Execute koje se mogu izvršiti
- *Invoker* - poziva klasu Command da izvrši određenu akciju
- *Command* - klasa koja implementira Execute operaciju tako što uključuje operacije iz klasa Receiver
- *Receiver* - klasa koja može da izvrši zahtevanu akciju
- *Action* - operaciju koju je potrebno izvršiti

Command patern na prvi pogled ima puno učesnika, ali se neki od njih odbacuju kada se koriste delegati.

Razlog uvođenja paternā: Ako želimo da implementiramo pravi *MVVM* patern, potrebno je odvojiti svu poslovnu logiku iz *View*-a i smestiti u *ViewModel*. Kada se doda neko dugme u *View*-u za to dugme je potrebno uraditi *binding* (vezivanje). Sa tim *binding*-om kažemo da će sva poslovna logika premestiti u *ViewModel* i tamo će se pozvati. Takođe mora postojati implementirana metoda na *ViewModel*-u, koja će biti pozvana na kliktanja na dugme. Ta metoda predstavlja komandu. Sam taj *binding* prestavlja akciju na koju se trigeruje komanda.

3.9.2. Strategy- *Strategy* patern uključuje uklanjanje algoritma iz klase u kojoj se nalazio i njegovo prebacivanje u posebne klase. Mogu postojati različiti algoritmi (strategije) koji se mogu primeniti za posmatrani problem. Ukoliko se algoritmi nalazu u jednom fajlu odakle se i pozivaju, dobićemo teško čitljiv kod sa dosta uslovnih iskaza. *Strategy* patern omogućava klijentu da odabere koji algoritam želi da koristi iz familije postojećih algoritama i pruža jednostavan način da se pristupi tom algoritmu. [12]



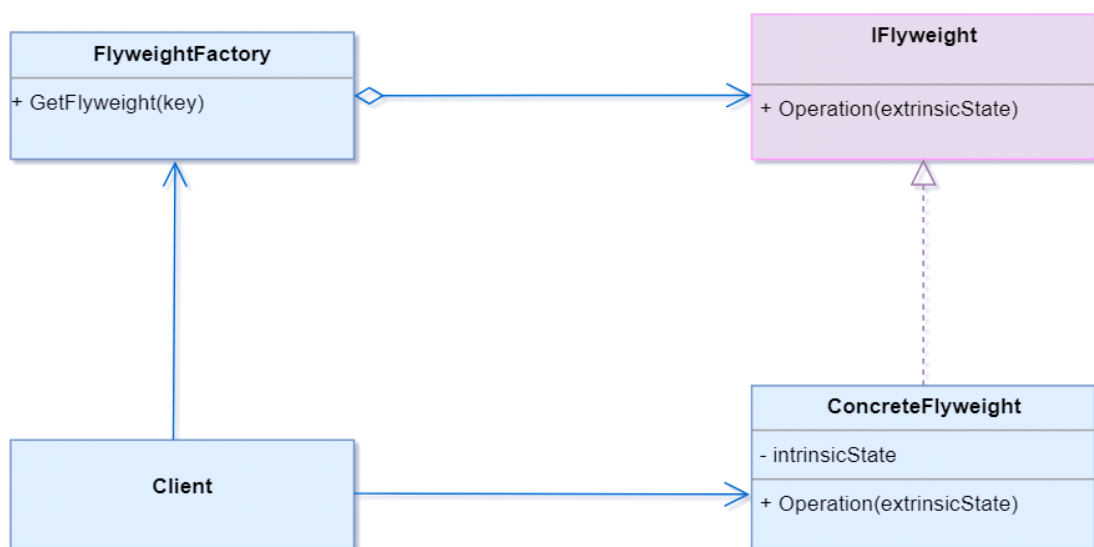
Slika 3.9.2.1. UML dijagram za *Strategy* patern

U okviru UML dijagram vidimo nekoliko učesnika čije su uloge sledeće:

- *Context* - klasa koja poseduje osnovne informacije koji algoritam treba da se izvrši
- *IStrategy* - definiše zajednički interfejs za sve strategije
- *StrategyA*, *StrategyB* - klase koje uključuju algoritme koji implementiraju *IStrategy*

Razlog uvođenja paterna: S obzirom da svi *drajveri* implementiraju isti interfejs na *DAL*-u se implementira ovaj patern i preko njega se bira koji *drajver* želimo pozvati i proslediti mu odgovarajuće podatke.

3.9.3. Flyweight- *Flyweight* patern koristi deljenje za rukovanje velikog broja sitnijih objekata. Prednost je ovog paterna je smanjenje broja objekata, štedi se memorija. Objedinjuje stanje većeg broja objekata, unutrašnje stanje u jednom objektu.



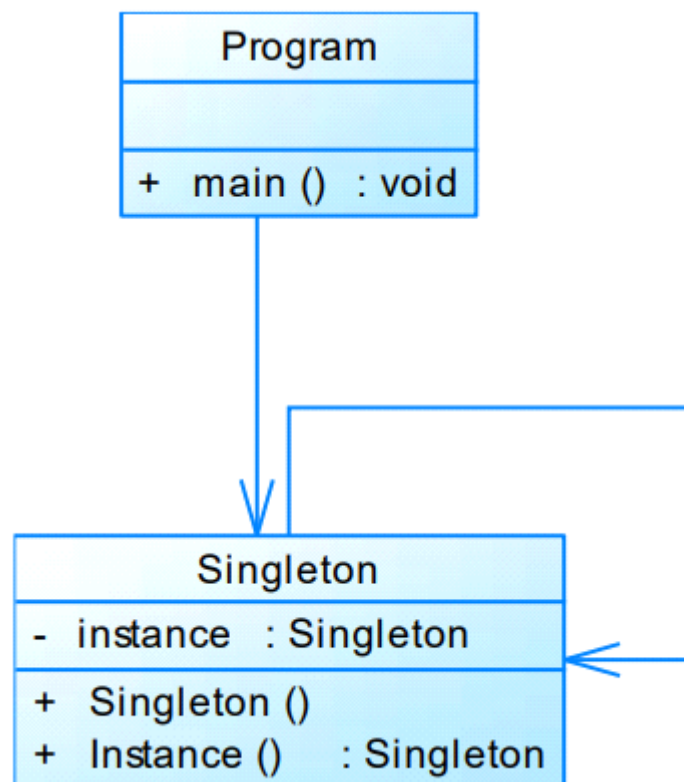
Slika 3.9.3.1. UML dijagram za *Flyweight* patern

U okviru UML dijagram vidimo nekoliko učesnika čije su uloge sledeće:

- *IFlyweight* - deklarise interfejs za spoljašnje stanje
- *ConcreteFlyweight* - definiše unutrašnje stanje, nezavisno od svog konteksta, mora biti deljiv
- *FlyweightFactory* - pravi objekte, upravlja njima, obezbeđuje ih klijentima
- *Client* - odžava referencu na objekat *FlyweightFactory*

Razlog uvođenja paterna: Svaki *drajver* kao njegov poslednji korak svih operacija je poziv servisa/baze podataka. Taj poziv se obezbeđuje preko *WCF* konekcije. I sad da se ne pravi svaki put novi poziv, te pozive čuvamo i skladištimo. *Drajver* poziva *FlyweightFactory* i od njega traži da mu vrati poziv ka servisu/bazi podataka. Ključ preko kojeg se identifikuju pozivi jeste adresa na kojoj service/baza podataka sluša.

3.9.4. Singleton - *Singleton* pattern ograničava instanciranje klase i osigurava da samo jedna instanca date klase postoji i pruža globalnu tačku pristupa ka toj instanci. Patern osigurava da je klasa instancirana samo jednom i da su svi zahtevi upućeni ka tom jednom i samo jednom objektu. [13]



Slika 3.9.4.1. UML dijagram za *Singleton* patern

UML dijagram klasa koji predstavlja *singleton* patern je prikazan na slici 3.5. Sa slike se može videti da kreiramo samo klasu *Singleton* koja obezbeđuje da će biti kreirana samo jedna njena instanca.

Razlog uvođenja paterna: U radu paterni *Singleton* i *Flyweight* su usko povezani. S obrizom da se ka klasi *FlyweightFactory* koja se nalazi u *Flyweight* paternu pristupa iz više različitih klijenata, i svaki put se instancira. Rešenje tog problema se javlja u vidu uvođenja *Singleton* paterna.

4. OPIS REŠENJA PROBLEMA

U ovom poglavlju će biti objašnjena implementacija i rad sledećih komponenti aplikacije:

4.1. Klijent

Implementacija klijenta se zasnima na tome da se korisniku obezbedi odgovarajuća polja za unošenje podataka, da se rezultat operacije ispiše i da aplikacija ima odgovarajući estetski izgled. Jedna stvar o kojoj klijent ne treba da vodi računa jeste kakvog tipa su podaci. Dali su tipa int, string, bool... Klijent treba da samo pošalje unete podatke i da čeka odgovor. Rešenje tog problema se javlja u implementiranju klase *InputObjects* (listing 4.1.1.).

```
[DataContract]
[KnownType(typeof(object[]))]
public class InputObjects
{
    private object inputs;

    public InputObjects()
    {
    }

    public InputObjects(object inputs)
    {
        this.inputs = inputs;
    }

    [DataMember]
    public object Inputs
    {
        get { return inputs; }
        set { inputs = value; }
    }
}
```

Listing 4.1.1. Izgled klase *InputObjects*

Input Objects ima samo jedno polje *inputs* koje je tipa *object*. Kada klijent zatraži izvršenje neke operacije, pravi se instanca klase *Input Objects*, a podatke koje je klijent uneo "pakuju" se u *inputs* polje u odgovorajućem redosledu i takav upakovan objekat se šalje dalje.

Kao što se podaci neke operacije koje klijent unosi smeštaju u klasu tipa *InputObjects*, tako i rezultat te operacije se smešta u posebnu klasu. Ta klase je tipa *OutputMessage* (listing 4.1.2.) i ima dva polja *output* i *type*. U *output* je smešta rezultat operacije, dok *type* predstavlja kod tipa je rezultat (uspešno ili neuspešno). Na osnovu *type* polja klijent zna na koji način da ispiše rezultat operacije.


```
[DataContract]
public class OutputMessage
{
    private string output;
    private MessageTypes type;

    public OutputMessage()
    {
    }

    public OutputMessage(string output, MessageTypes type)
    {
        this.output = output;
        this.type = type;
    }

    [DataMember]
    public string Output
    {
        get { return output; }
        set { output = value; }
    }

    [DataMember]
    public MessageTypes Type
    {
        get { return type; }
        set { type = value; }
    }
}
```

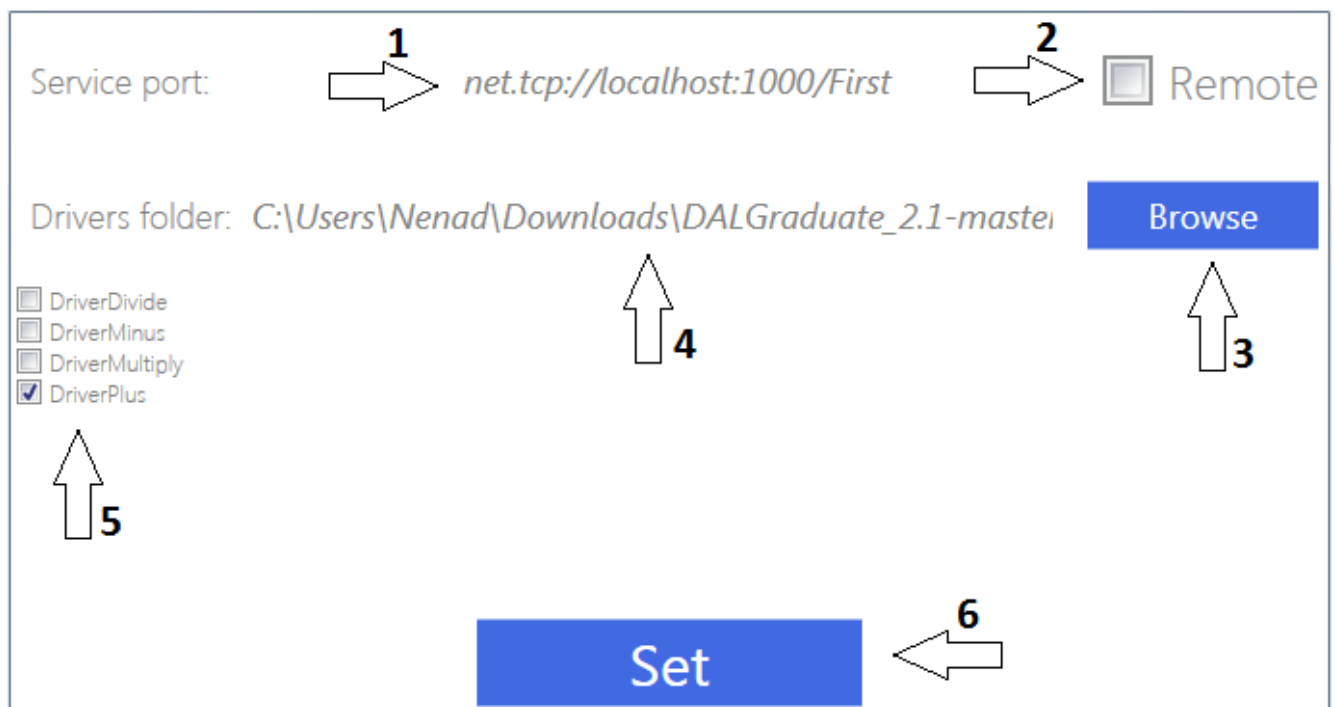
Listing 4.1.2. Izgled klase *OutputMessage*

Obe klase *InputObjects* i *OutputMessage* imaju atribut *[DataContract]*, dok njihova polja imaju atribut *[DataMember]*, ti atributi su obavezni jer se ove klase dele kroz WCF konekcije i preko njih WCF zna dali su te klase uključene u proces serijalizacije.

4.2. DAL

DAL se podiže kao zasebna aplikacija, nezavisno od klijentske aplikacije. Tek nakon postavke svih opcija koje DAL podržava, tek tada klijent može da zatraži izvršenje operacije tj. tek tada može da prosledi podatke.

Na slici 4.2.1. je prikazan izgled DAL aplikacije. Inače za izgled aplikacije je korišćena WPF tehnologija.



Slika 4.2.1. Izgled *local* DAL aplikacije

Osnovni pojmovi u DAL aplikaciji:

1. Predstavlja adresu na kojoj DAL sluša, tj. na tu adresu klijent šalje svoj zahtev
2. Checkbox *Remote*, sa kojim podešavamo hoće li nam DAL biti *local* ili *remote* tipa. Ako se ne štiklira ovo polje, DAL će biti *local*.
3. Nakon kliktanja dugmeta *Browse*, otvara se novi prozor i dobija se mogućnost da se izabere folder gde se nalaze svi drajveri
4. Lokalna adresa izabranih drajvera
5. Listbox svih izabranih drajvera
6. Konačno dugme (*Set*) sa kojim se potvrđuju sve prethodno podešene postavke. Pre kliktanja ovog dugmeta prvo moramo izabrati minimum bar jedan drajver od ponuđenih.

Kako se veza između klijenta i DAL-a odvija preko WCF konekcije tako mora i da postoji neki interfejs, preko koga će ta veza biti izvršena (listing 4.2.1.).

```
[ServiceContract]
public interface IClientConnectToDAL
{
    [OperationContract]
    OutputMessage ReturnOperationValue(
        InputObjects inputs,
        OperationTypes type,
        bool remote);

    List<string> ReturnDriverDlls(string path);

    bool AddDrivers(List<string> selectedDrivers);
}
```

Listing 4.2.1. Interfejs *IClientConnectToDAL*

Interfejs *IClientConnectToDAL* ne koristi samo klijent da bi se povezao sa *local* DAL-om, nego taj isti interfejs *local* DAL koristi da bi se povezao sa *remote* DAL-om. Atributi *[ServiceContract]* i *[OperationContract]* su obavezni i oni označavaju da se radi o interfejsu koji se koristi za WCF svrhe. Da bi se klijent povezao sa DAL-om, prvo DAL mora da implementira ovaj interfejs da bi ga klijent uopšte pozvao. Klijent može da pozove samo jednu metodu tj. metodu *ReturnOperationValue*. Razlog ove pojave jeste atribut koji se nalazi iznad te metode *[OperationContract]*, bez tog atributa WCF ne može da podrži poziv te metode sa klijentske strane. Ostale dve metode, njih WCF ne podrža jer one nisu dostupne klijentu. Naime njih koristi samo DAL, i one služe da se izvuku drajveri iz nekog foldera i da se ti drajveri kasnije ispišu u DAL aplikaciji.

Metoda koja je namenjena za klijenta (*ReturnOperationValue*) ima tri parametra:

1. *inputs* koji je tipa *InputObjects*, ovaj parametar predstavlja podatke koje je klijent uneo. Tipa podaci su upakovani u ovaj parametar i poslani dalje
2. *type* koji je tipa *OperationTypes*, predstavlja kojeg je tipa operacija. Može biti operacija sabiranja, množenja, upis novog objekta u bazu, SQL upit. Na osnovu ovog parametra, DAL zna koji drajver da pozove
3. *remote* koji je tipa *bool*, označava da li se radi o *local* ili *remote* DAL-u. Pošto ovaj isti interfejs i ovu metodu koristi i *local* DAL da bi se povezao sa nekim *remote* DAL-om, onda se uviđa potreba za ovim parametrom.

Implementacija ove metode se nalazi na DAL-u i njena implementacija se može videti na listingu 4.2.2.

```
public OutputMessage ReturnOperationValue(InputObjects inputs, OperationTypes type, bool
remote)
{
    if (!canOperate)
    {
        return
        driverObjectContainer.ReturnErrorMessage(StringMessegas.errorSelectDrivers,
        MessageType.ErrorPopUp);
    }

    IOperationMessenger driver = null;
    string convertedDriver = driverMapper.ConvertTypeToDriver(type);

    if (drivers.TryGetValue(convertedDriver, out driver))
    {
        dalController.SetStrategy(driver);
        return dalController.DoOperation(inputs);
    }
    else
    {
        if (!remote)
        {
            List<string> addresses;
            try
            {
                addresses =
                txtFileWorker.ReadAllLines(StringMessegas.endPointTXT);
            }
            catch
            {
                addresses = new List<string>(0);
            }

            return driverWorker.CreateNewDAL(inputs, type, addresses);
        }
        else
        {
            return null;
        }
    }
}
```

Listing 4.2.2. implementacija metode *ReturnOperationValue*

Prvo se nailazi na boolean *canOperate*. Ta vrednost se postavlja na true u metodi *AddDrivers*, tek kada se na DAL-u izabere barem jedan drajver i time se klijentu omogućuje dalji rad. U suprotnom, ako još nijedan drajver nije izabran, klijentu će biti ispisana greška da prvo mora biti izabran drajver. Da se ne pravi svaki put novi objekat povrate poruke, pozivamo pomoćnu klasu i metodu *ReturnErrorMessage* u kojoj se čuvaju sve povratne poruke sistema u obliku greške. Time optimizujemo korišćenje memorije (kombinacija *Singleton* i *Flyweight patterna*).

Na osnovu parametra *type* znamo o kojoj se operaciji radi, tako da znamo i koji drajver treba da pozovemo. Pozivamo pomoćnu klasu i metode *ConvertTypeToDriver* šaljemo parametar *type*, a povratna vrednost je naziv drajvera koji implementira odgovarajuću operaciju.

Nakon toga, pokušavamo da pronađemo drajver tražene operacije u privatnom dictionary *drivers*. Ako pronađe drajver, postavlja se njegova instanca uz korišćenje pomoćne klase i metode *SetStrategy* (ova klasa implementira *Strategy patern*). Zatim nakon postavljanja, tom drajveru se šalju podaci koje je korisnik uneo.

U suprotnom slučaju, ako se ne pronađe drajver iz dictionary pokušavamo da ostvarimo vezu sa nekim od *remote* DAL-ova.

Proveravamo koju vrednost ima parametar *remote*. Ako je *false*, radi se o *local* DAL-u i onda on pokušava da otvori *txt* fajl, u kome se nalaze adrese svih *remote* DAL-ova. Inače nakon pokretanja *remote* DAL-a, on svoju adresu upisuje na kraj ovog fajla. Otvaramo fajl i sve adrese stavljamo u lokalnu listu *addresses*. Try, catch blok se tu nalazi jer u slučaju da ne postoji fajl onda ne želimo da nam aplikacija pukne. Nakon punjenja liste sa adresama, pozivamo pomoćnu klasu i metodu *CreateNewDAL*, prosleđujemo podatke koje je korisnik uneo, tip operacije i adrese. U toj metodi se pronalazi odgovarajući *remote* DAL sa odgovarajućim drajverom koji podržava traženu operaciju. Vrš se linearni upit od prve adrese pa do poslednje i staje se na prvom DAL-u koji ima odgovarajući drajver. Ako ne postoji nijedan *remote* DAL koji rukuje sa drajverom tražene operacije, klijentu će biti ispisana greška da mora otvoriti novi *remote* DAL.

Ako je parametar *remote* pozitivne vrednosti, tj *true* onda metoda samo vraća *null* kao povratnu vrednost.

Izgled DAL aplikacije, kad se radi o *remote* DAL-u je identičan kao i *local*, samo što se checkbox kojim se daje mogućnost da se bira između *local* ili *remote*, štiklira. Izgled se može videti na slici 4.2.2.

The screenshot shows a configuration window for the DAL application. At the top, there is a label 'Service port:' followed by the text 'net.tcp://localhost:1361/'. To the right of this text is a checked checkbox labeled 'Remote'. Below this, there is a label 'Drivers folder:' followed by the path 'C:\Users\admin\Source\Repos\DALGraduate_2.1\DAL'. To the right of this path is a blue button labeled 'Browse'. Below the 'Drivers folder' section, there are four checkboxes: 'DriverDivide' (unchecked), 'DriverMinus' (checked), 'DriverMultiply' (checked), and 'DriverPlus' (unchecked). At the bottom center of the window is a large blue button labeled 'Set'.

Slika 4.2.2. Izgled *remote* DAL aplikacije

4.3. Drajveri

Nakon pokretanja DAL-a i klijenta, nakon odabira drajvera i unošenja podataka i biranja odgovarajuće operacije ti podaci će biti prosleđeni kao određenom drajveru koji će te podatke da validira, konvertuje...

Pošto se klijentu obezbeđuje visok nivo apstrakcije, tako je bilo potrebno napraviti jedan interfejs koji će svi drajveri da nasleđuju i da implementiraju. Naziv takvog interfejsa je *IOperationMessenger*, izgled interfejsa možemo videti na listingu 4.3.1.

```
public interface IOperationMessenger
{
    OutputMessage DoOperation(InputObjects inputs);
}
```

Listing 4.3.1. Izgled interfejsa *IOperationMessenger*

Interfejs imao samo jednu metodu *DoOperation*, sa jednim parametrom *inputs* tipa *InputObjects*. Pošto smo na DAL-u saznali o kom koji drajver implementira traženu operaciju, onda nam samo trebaju podaci koje je korisnik uneo. Drajver zna kako te podatke da otpakuje i validira, jer zna koju operaciju podržava.

Na listingu 4.3.2. je dat primer implementacije metode *DoOperation*. Radi se o drajveru koji je namenjen za deljenje dva broja.

Prvo se otpakuju podaci, pa zatim se pokušava izvršiti parsiranje tih podataka iz tipa *object* u tipa *decimal*. Ako se parsiranje izvrši neuspešno, to znači da je korisnik uneo pogrešan tip podatka tj. da podatak koji je uneo ne sačinjava samo brojeve nego i neke druge karaktere i zato je parsiranje prošlo neuspešno. Druga provera gleda da li je drugi broj *nula*, ako jeste opet se korisniku ispisuje greška koja kaže da se prvi broj ne može deliti sa drugim ako je vrednost drugog broja jednaka *nuli*.

Tu se završava sva moguća validacija, zatim se ti podaci salju ka odgovarajućem servisu/bazi podataka koji će te podatke da izvrši, skladišti... U ovom konkretnom slučaju, poziva se servis koji deli dva broja i vraća rezultat operacije nazad ka drajveru zatim se taj rezultat šalje ka DAL-u pa konačno do klijenta. Konekcija izmedju drajvera i servisa, u ovoj slučaju se odvija preko WCF konekcije.

Ovde je ubačen *try, catch* blok jer postoji mogućnost od neočekivanih grešaka. Tipa može se desiti da servis uopšte nije ni podignut tj. da ne postoji ili da se ugasio u toku slanja podataka pa se time i veza prekida. Za svaki *catch* blok pokušavamo da uhvatimo *exception* koji je uzrokovao aktiviranje *catch* bloka i tu grešku šaljemo ka klijentu. Kao i za metodu *ReturnOperationValue* koja poziva pomoćnu klasu i metodu *ReturnErrorMessage* u kojoj se čuvaju sve povratne poruke sistema u obliku greške, tako se i ovde koristi isti princip.

```
public OutputMessage DoOperation(InputObjects objects)
{
    object[] parameters = objects.Inputs as object[];
    decimal first;
    decimal second;

    if (!decimal.TryParse(parameters[0] as string, out first) ||
        !decimal.TryParse(parameters[1] as string, out second))
    {
        return
            driverObjectContainer.ReturnErrorMessage(StringMessegas.errorInputDecimal, MessageTypes.ErrorPopUp);
    }

    if (second == 0)
    {
        return
            driverObjectContainer.ReturnErrorMessage(StringMessegas.errorDivideByZero, MessageTypes.ErrorPopUp);
    }

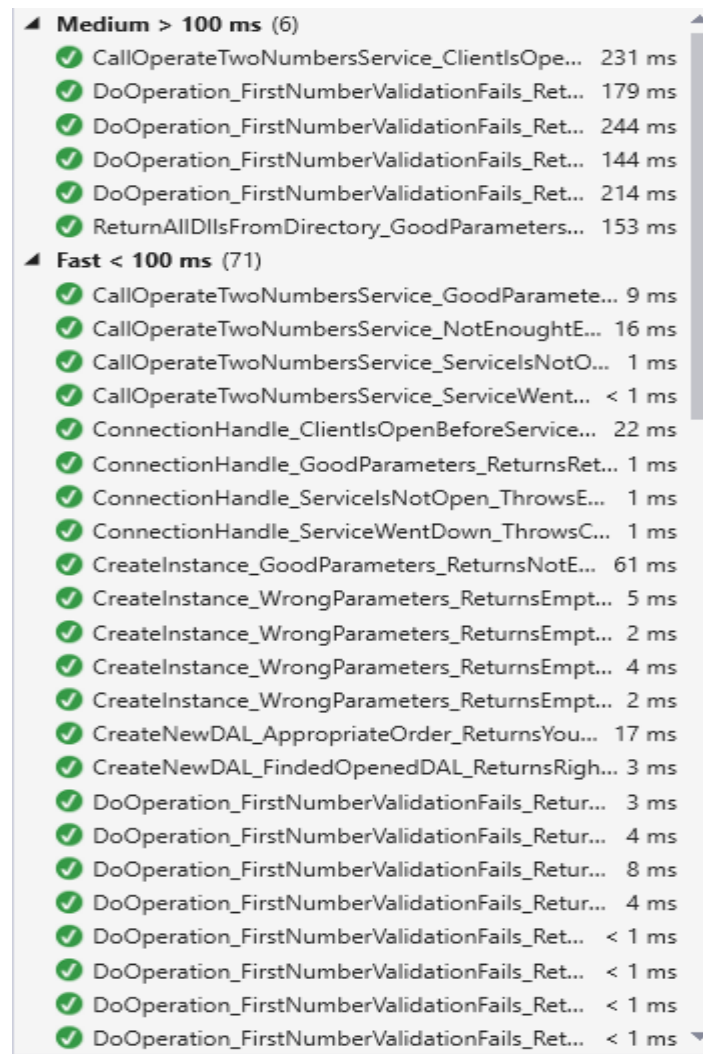
    try
    {
        string result =
            driverObjectContainer.ReturnDriverProxy(address).DoOperation
                (first, second).ToString();

        return new OutputMessage(result, MessageTypes.Success);
    }
    catch (EndpointNotFoundException)
    {
        return
            driverObjectContainer.ReturnErrorMessage(StringMessegas.errorDivideServiceNotOpen, MessageTypes.ErrorPopUp);
    }
    catch (CommunicationObjectFaultedException)
    {
        driverObjectContainer.OverrideChannel(address);
        return
            driverObjectContainer.ReturnErrorMessage(StringMessegas.errorDivideServiceThenClient, MessageTypes.ErrorPopUp);
    }
    catch (CommunicationException)
    {
        return
            driverObjectContainer.ReturnErrorMessage(StringMessegas.errorDivideServiceShutDown, MessageTypes.ErrorPopUp);
    }
}
```

Listing 4.3.2. Implementacija metode *DoOperation* za deljenje dva broja

4.4. NUnit testiranje

Aplikacija je testirana uz pomoć *NUnit* testova prethodno opisanih u tekstu. Testiraju se sve implementirane metode i testiraju se svi mogući slučajevi te metode. Na slici 4.4.1. se mogu videti svi testkovi koji su se uspešno izvršili.



Slika 4.4.1. Uspešni *unit* testovi

Primer jednog testa (listing 4.4.1.) za testiranje drajvera koji služi za deljenje dva broja. Primer testa pokriva slučaj kada je prvi broj pogrešno unet, tj. uz brojeve uneti su i još neki ne željeni karakteri. Pošto želimo da testiramo metodu *DoOperation*, a ona prima kao parametar objekat tipa *InputObjects* tako i moramo da napravimo instancu tog objekta sa test vrednostima. Zatim se poziva ta testirana metoda i prosleđuju joj se parametar. Pošto metoda *DoOperation* u svojoj implementaciji kada ne prođe validiranje brojeva pozove pomoćnu klasu i metodu *ReturnErrorMessage* sa odgovarajućim parametrima tako i mi moramo da proverimo tj. testiramo da li je uopšte pozvana ta metoda kad dođe do neuspešne validacije.


```
[Test]
[TestCase("154pp", "60")]
[TestCase("", "60")]
[TestCase("-.45454544g", "60")]
[TestCase("asd1f", "60")]
public void DoOperation_FirstNumberValidationFails_ReturnsErrorMessage( string
numberOne, string numberTwo)
{
    InputObjects inputObject = new InputObjects(new object[] { numberOne,
numberTwo });
    var actual = divideNumbersDriver.DoOperation(inputObject);

    driverObjectContainer
        .Received()
        .ReturnErrorMessage(StringMessegas.errorInputDecimal,
        MessageType.ErrorPopUp);
}
```

Listing 4.4.1. Primer NUnit testa za metodu *DoOperation* koja deli dva broja

5. ZAKLJUČAK

Prednosti korišćenja slojevite arhitekture su mnogobrojne i takve prednosti su implementirane u radu:

- **Nezavisna implementacija svakog elementa sistema** – svaki element sistema je implementiran nezavisno od ostalih delova. Klijent može zasebno da se razvija od DAL-a, a DAL može zasebno da se razvija od drajvera. Ova osobina je obezbeđena korišćenjem odgovarajućih interfejsa koji svaka komponenta
- **Skalabilnost** – veoma lako se može dodati novi prozor na klijentu koji će raditi odgovarajuću operaciju, samo za tu operaciju je potrebno dodati i drajver koji će je podržavati
- **Apstrakcija** – klijent svoj zahtev sa podacima pakuje u objekte, koje se raspakuju kada dođu do drajvera. Time je omogućeno da se veliki broj različitih vrsta objekata pošalje (int, string, bool, instanca neke klase...)

Mane ovakve implementacije i korišćenja slojevite arhitekture generalno jesu kaskadne promene. Primer takve mane se odgleda u tome da drajver zna na koji način klijent pakuje podatke. Greška se može desiti ako klijent na jedan način upakuje podatke, a drajver na drugi način ih otpakuje. Ako odredimo na koji se način pakuju podaci na klijentu, na taj isti način moramo raspakovati podatke na drajveru. Ako promenimo način na klijentu, moramo i na drajveru. Pod ovom osobinom se odnose kaskadne promene.

Alternativa pakovanja podataka, bi bila da se na postojećem interfejsu dodaju metode za svaku operaciju. Takva primena ima dve očigledne mane:

- Ako postoji jedan drajver za jednu operaciju i svi implementiraju isti interfejs, a taj interfejs ima puno metoda to znači da bi svaki drajver imao implementiranu samo jednu metodu
- Ako se odlučimo da imamo samo jedan drajver koji implementira interfejs, onda bi sve te metode bile implementirane na jednom mestu pa se time krši jedan od S.O.L.I.D. principa

Takođe uvek postoji prostora za napredak i usavršavanje što je upravo slučaj i sa ovim radom. Određeni segmenti aplikacije bi mogli biti rešeni na bolji i efikasniji način. Jedan od mogućih unapređenja ili izmena je da klijenta aplikacija bude urađena u nekoj drugoj tehnologiji koja nije WPF npr. Windows Form ili ASP.NET. Komunikacija između klijenta i DAL može se odvijati preko socketa, umesto WCF-a.

Kada sistem krene da se ponaša kako ne treba, mogla bi se implementirati logging ideja praćenja event-ova. U logove bi se upisivali greške sistema za koje korisnik ne bi trebao da zna. Npr. zašto nema više slobodnih portova na koji bi se podigli novi *remote* DAL-ovi.

6. LITERATURA

- [1] Stackify, *What is N-Tier Architecture? How It Works, Examples, Tutorials, and More*, datum pristupa: 17/09/2019 <<https://stackify.com/n-tier-architecture/>>
- [2] Guru99, *What is Microsoft .Net Framework?*, datum pristupa: 17/09/2019 <<https://www.guru99.com/net-framework.html>>
- [3] GeeksforGeeks, *Introduction to Visual Studio*, datum pristupa: 17/09/2019 <<https://www.geeksforgeeks.org/introduction-to-visual-studio/>>
- [4] Jon Skeet, *C# in Depth, 3rd Edition*, ISBN-13: 978-1617291340
- [5] WPF Tutorial, *What is WPF?*, datum pristupa: 17/09/2019 <<https://www.wpf-tutorial.com/about-wpf/what-is-wpf/>>
- [6] Microsoft, *What Is Windows Communication Foundation*, datum pristupa: 17/09/2019 <<https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf>>
- [7] NUnit, datum pristupa: 17/09/2019 <<https://nunit.org/>>
- [6] Automation Rhapsody, *Code coverage of manual or automated tests with OpenCover for .NET applications*, datum pristupa: 17/09/2019 <<https://automationrhapsody.com/code-coverage-manual-automated-tests-opencover-net-applications/>>
- [9] Wintellect, *Model-View-ViewModel (MVVM) Explained*, datum pristupa: 17/09/2019 <<https://www.wintellect.com/model-view-viewmodel-mvvm-explained/>>
- [10] Source Making, *Design Patterns*, datum pristupa: 17/09/2019 <https://sourcemaking.com/design_patterns>
- [11] Source Making, *Command Design Pattern*, datum pristupa: 17/09/2019 <https://sourcemaking.com/design_patterns/command>
- [12] Source Making, *Strategy Design Pattern*, datum pristupa: 17/09/2019 <https://sourcemaking.com/design_patterns/strategy>
- [13] Source Making, *Singleton Design Pattern*, datum pristupa: 17/09/2019 <https://sourcemaking.com/design_patterns/singleton>
- [14] Source Making, *Flyweight Design Pattern*, datum pristupa: 17/09/2019 <https://sourcemaking.com/design_patterns/flyweight>

7. LISTA KORIŠĆENIH SKRAĆENICA

- WPF - Windows Presentation Foundation
- WCF - Windows Communication Foundation
- SQL - Structured Query Language
- HP - Hewlett-Packard
- CLI - Common Language Infrastructure
- CLR - Common Language Runtime
- CIL - Common Intermediate Language
- API - Application Programming Interface
- XML - Extensible Markup Language
- HTML - Hypertext Markup Language
- CSS - Cascading Style Sheets
- XAML - Extensible Application Markup Language
- UI - User Interface
- MVVM - Model-View-ViewModel
- UML - Unified Modeling Language

PODACI O KANDIDATU

Kandidat Nenad Zelenović je rođen 01.04.1995. godine u Somboru. Završio je srednju ekonomsku školu u Somboru 2014. godine. Fakultet tehničkih nauka u Novom Sadu je upisao 2014. godine. Ispunio je sve obaveze i položio sve ispite predviđene studijskim programom.