БЕОГРАДСКА АКАДЕМИЈА ПОСЛОВНИХ И УМЕТНИЧКИХ СТРУКОВНИХ СТУДИЈА

СЕМИНАРСКИ РАД

ОСНОВНЕ СТРУКОВНЕ СТУДИЈЕ

Професор: Проф. Др. Горан Аритоновић

Студент: Бранислав Пауновић

Београд
2025.

БАПУСС

Београдска академија
пословних и уметничких
струковних студија

Студијски програм : Информациони системи и технологије

СЕМИНАРСКИ РАД

ОСНОВНЕ СТРУКОВНЕ СТУДИЈЕ

(Тема:) **ПРИМЕНА ПРИНЦИПА ОБЈЕКТНО ОРИЈЕНТИСАНОГ ПРОГРАМИРАЊА КРОЗ РАЗВОЈ ДЕСКТОП АПЛИКАЦИЈЕ ЗА ЕВИДЕНЦИЈУ СТУДЕНАТА**

Професор:

1. Др. Горан Аритоновић _____,

(потпис)

Датум одбране рада: 27.06.2025

Оцена: _____

Студент: Бранислав Пауновић

Број индекса: 2I1/0054/24

ПРЕДМЕТ: ИЗЈАВА

Овај рад сам оригинално написао/ла, те тврдим на основу њега раније нико није имао корист у некој другој институцији.

Потпис студента:

_____

Садржај :

1. **Увод**
   a) Циљ и обим рада
   b) Кратак опис WinForms апликације
   c) Korišćene tehnologije (C#, WinForms, SQL Server, ADO.NET)

2. **Објектно оријентисана структура апликације**
   a) Класа Student (ID, Ime, Prezime, Email, DatumRođenja)
   b) Остале класе (по потреби): User, Adresa
   c) Наслеђивање, енкапсулација, OOP концепти у коду
   d) Веза ентитета и класа са базом
   e) ER дијаграм студената и повезаних ентитета (mini dijagram)

3. **ADO.NET – основни концепти**
   a) Шта је ADO.NET и .NET data provider
   b) Кључни простори имена (System.Data, System.Data.SqlClient)
   c) Разлика између повезаног и неповезаног начина рада

4. **Управљање конекцијом ка бази**
   a) Класа SqlConnection
      – *Својства: State, ConnectionTimeout*
      – *Методе: Open(), Close(), BeginTransaction()*
      – *Догађаји: StateChange*
   b) ConnectionString (App.config / Settings)
   c) Windows vs SQL Server аутентификација

5. **Извршавање SQL команди**
   a) Класа SqlCommand
      – *Параметри: CommandText, CommandType, Connection, Parameters*
      – *Методе: ExecuteNonQuery(), ExecuteScalar(), ExecuteReader()*
   b) Параметризовани упити – Input и Output параметри
   c) Рад са ускладиштеним процедурама
      – *Креирање у SSMS*
      – *Подешавање у C# (CommandType.StoredProcedure)*
      – *Читање Output параметара*

6. **Читање података: SqlDataReader**
   a) Концепт read-only, forward-only курсора
   b) Методе: Read(), IsDBNull(), GetString(), GetInt32() итд.
   c) Мапирање података у објекат класе Student

7. Data Access Layer (DAL)

a) Инстанца класе SmepDal
b) CRUD методе:
– *List<Smer> VratiSmer()*
– *int UbaciSmer(Smer s)*
– *int PromeniSmer(Smer s)*
– *int ObrisiSmer(int id)*
c) Try-Catch обрада изузетака у DAL слоју

---

8. WinForms кориснички интерфејс

a) Основне форме: frm_Student, frm_DataGreedView
b) Приказ података:
b.1) ***DataGridView***
– *Подешавање DataSource*
– *Конфигурација колона, избор редова, догађаји*
b.2) ***ListView***
– *Режими приказа, својства:* `Columns`, `Items`, `GridLines`
– *Догађај:* `SelectedIndexChanged`
c) Повезивање UI и DAL метода

---

9. Валидација и обрада грешака

a) Валидација уноса: string.IsNullOrWhiteSpace()
b) Try-Catch блокови при раду са базом
c) Приказ грешака кориснику (MessageBox, error provider...)

---

10. Закључак

a) Шта је постигнуто у раду
b) Кључне лекције (ADO.NET, WinForms, SQL)
c) Могућности проширења (претрага, логин, више ентитета)

---

11. Литература

a)Лекције Проф. Др. Горана Аритоновића (ООП2022_10, ООП2022_11)
b)Лекције из средње ИТХС школе
c)ChatGpt
d)Kopilot

1. Увод

a) Циљ и обим рада

Циљ овог семинарског рада је креирање једноставне али функционалне WinForms апликације која омогућава унос, измену, брисање и приказ података о студентима, чиме се практично демонстрирају принципи објектно оријентисаног програмирања (OOP) и рад са базама података преко ADO.NET технологије. Обим рада обухвата дизајн базе у SQL Server-у, развој програмске логике у C# језику, као и реализацију графичког корисничког интерфејса у WinForms окружењу. Посебан акценат је стављен на израду Data Access Layer-а (DAL), обраду грешака, и валидацију уноса.

b) Кратак опис WinForms апликације

Апликација представља класичан десктоп WinForms софтвер намењен евиденцији студената, са циљем да омогући лако управљање подацима у оквиру образовне институције.

Основне функционалности обухватају:

- логовање корисника,
- додавање нових студената,
- преглед и измену постојећих података,
- као и брисање записа.

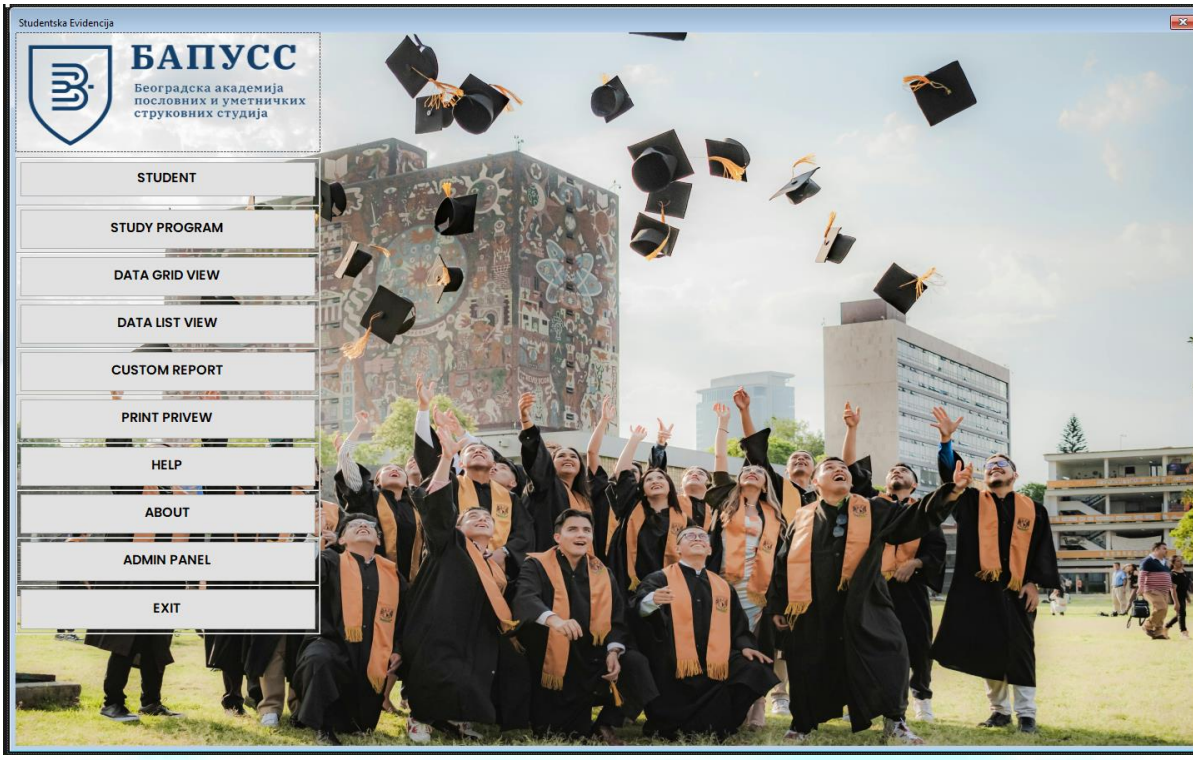
Почетна форма је **`frm_Login`**, која омогућава приступ систему само овлашћеним корисницима.

Након успешне пријаве, кориснику се отвара **`frm_MainForm`**, која представља централни мени апликације. Са ове форме се, једноставним кликом на одговарајуће дугме, позивају остале форме задужене за конкретне функционалности:

- **`STUDENT`** – отвара **`frm_Student`**, за унос, измену и брисање студената.
- **`STUDY PROGRAM`** – отвара форму за рад са подацима о студијским програмима.
- **`DATA GRID VIEW`** – отвара **`frm_DataGrid`**, са табеларним приказом података о студентима користећи DataGridView.
- **`DATA LIST VIEW`** – отвара **`frm_ListView`**, са прегледом података у виду листе.
- **`CUSTOM REPORT`** – отвара **`frm_CustomReport`**, која приказује прилагођене извештаје.
- Остала дугмад воде ка формама: `HELP`, `ABOUT`, `ADMIN PANEL`, `PRINT PREVIEW` и `EXIT`.

Свака форма функционише као независна целина, али је у потпуности интегрисана у апликацију преко `frm_MainForm`. Комуникација са базом података реализована је преко **DAL (Data Access Layer)** слоја, што омогућава раздвајање логике пословања од корисничког интерфејса и доприноси бољој организацији кода, већој сигурности и лакшем одржавању.

Оваква архитектура – логовање, модуларне форме, одвојен слој за приступ подацима – обезбеђује јасну навигацију, стабилан рад, једноставну употребу и могућност проширења система у будућности.

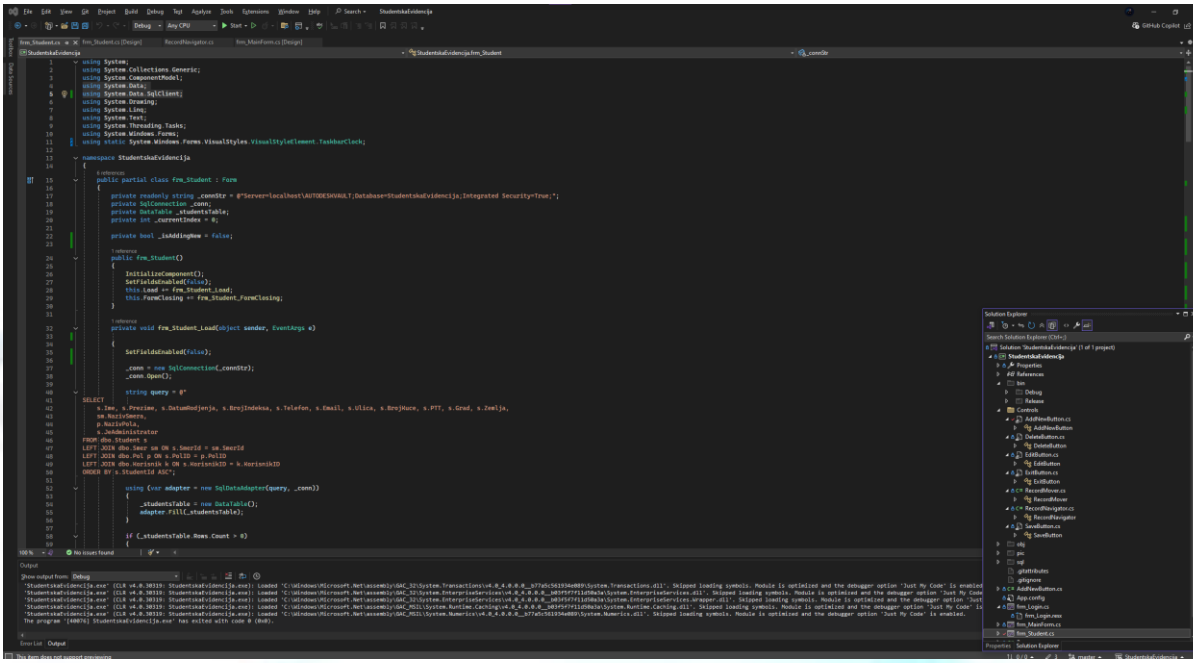


Форма frm_MainForm

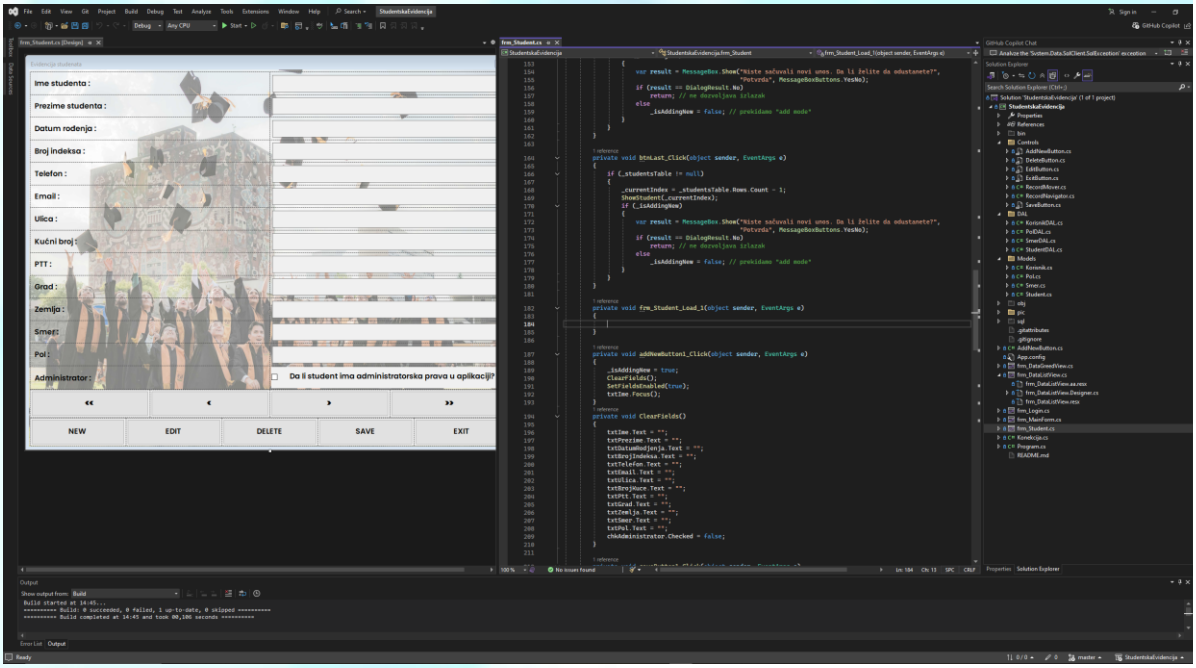
---

c) Коришћене технологије

- **C#** – као главни програмски језик коришћен за имплементацију класа, логике, управљање догађајима и креирање објеката у WinForms окружењу.
- **WinForms** – као технологија за изградњу десктоп графичког корисничког интерфејса.
- **SQL Server 2021** – као систем за управљање релационим базама података (RDBMS), у којем су смештене све табеле, подаци, релације и евентуалне процедуре које апликација користи.
- **ADO.NET** – као .NET библиотека за повезивање апликације са базом података, преко објеката као што су: `SqlConnection`, `SqlCommand`, `SqlDataReader`, `SqlParameter`, итд.
- **Visual Studio Community 2022** – као главно интегрисано развојно окружење (IDE) у којем је креиран и тестирани цео пројекат, укључујући форме, класе, кôд и дизајн.

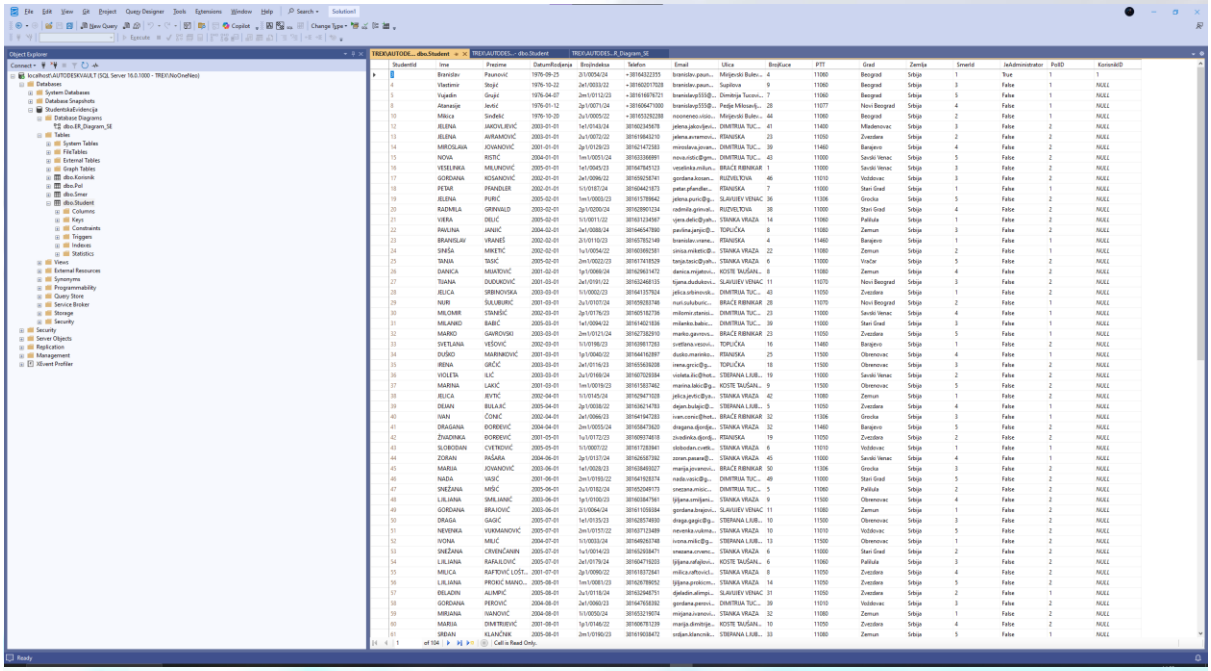


Visual Studio Community 2022 бесплатно Microsoftovo развојно окружење за C#



Развојно окружење пројекта StudentskaEvidencija са приказом форме frm_Student, дела њеног кода, као и SolutionExplorera са свим класама, објектима и формама пројекта.

- **SQL Server Management Studio 21 (SSMS)** – као графички алат за креирање и одржавање базе података, извршавање SQL упита, преглед табела и података.



SQL Server Menagment Studio 21 бесплатно Microsoftovo развојно окружење за SQL Server

- **GitHub** – као удаљени репозиторијум (remote repository) за контролу верзија, сигурносно чување кода и сарадњу. Све фазе развоја апликације, од иницијалног креирања до завршне верзије, праћене су кроз GitHub.
- **.NET Framework 4.x** – као радни оквир потребан за извршавање WinForms апликације у Windows окружењу.

2. Објектно оријентисана структура апликације

а) Класа Student (ID, Ime, Prezime, Email, DatumRođenja)

Класа `Student` представља основни модел података у апликацији. Садржи приватне атрибуте као што су `StudentId`, `Ime`, `Prezime`, `Email` и `DatumRodjenja`, који представљају колоне из истоимене табеле у бази. Сви атрибути су доступни преко јавних својстава (properties) чиме се примењује енкапсулација. Ова класа служи као мост између базе података и логике апликације, а користи се у већини форми ради приказа, уноса или измене студената.

б) Остале класе (User, Adresa, RecordNavigator)

Поред класе `Student`, апликација користи и класу `Korisnik` за моделовање овлашћених корисника система, као и класу `Smer` (у форми својстава унутар студента: смера који студент похађа). Посебно значајна је класа `RecordNavigator`, која инкапсулира логику навигације кроз редове података у DataTable-у. Уместо да свака форма садржи своју логику за

`btnFirst`, `btnPrevious`, `btnNext`, `btnLast`, све је централизовано у `RecordNavigator`, што је чист пример добре ООП праксе.

в) Наслеђивање, енкапсулација, ООР концепти у коду

Апликација користи основне принципе објектно оријентисаног програмирања: **енкапсулацију** (сви подаци у класама су приватни и приступа им се преко гетера и сетера), **апстракцију** (корисник не зна како се подаци дохватају, већ само користи методе), као и **наследђивање** у одређеним деловима кода – на пример, форма `frm_AdminPanel` може да проширује функционалност `frmLogin`. Додатно, `RecordNavigator` изоловано управља навигацијом, што омогућава поновну употребу кода без дуплирања.

Дугмад померања по слоговима направљена уз помоћ – **record navigator clase**

Класа **RecordNavigator**, која омогућава навигацију кроз редове података из базе *(први, претходни, следећи, последњи)*. Ова класа чува унутарњи **DataTable** и индекс тренутног реда, а **методе** попут *MoveFirst(), MovePrevious(), MoveNext() и MoveLast()* омогућавају прецизно кретање.

На формама се налазе четири дугмета за навигацију, која преко догађаја позивају методе из класе. Овим се постиже енкапсулација логике *(скривање унутрашње логике класе и излагање само оних метода које су потребне спољa.)* у једном месту, што омогућава поновну потребу кода на више форми и смањује количину понављања у програму – типичан пример **ООП концепта у пракси.** Циљ енкапсулације је мања могућност грешке, лакше одржавање, јасан интерфејс.

Пример – дугме `Move Last`

```
private void btnLast_Click(object sender, EventArgs e)
{
   navigator.MoveLast();
}
```

Исти принцип се користи и за дугмад `btnFirst`, `btnPrevious` и `btnNext`, која позивају методе `MoveFirst()`, `MovePrevious()` и `MoveNext()` у оквиру класе `RecordNavigator`.

г) Веза ентитета и класа са базом

Свака класа одговара конкретној табели у бази. На пример, класа `Student` мапира се на табелу `Student` у SQL Server-у. Повезивање се врши путем ADO.NET компоненти (`SqlConnection`, `SqlCommand`, `SqlDataReader`), кроз DAL слој. Методе за `Insert`, `Update`, `Delete`, и `Select` су дефинисане у **DAL класама** и користе се из UI-а, чиме се раздваја логика пословања од презентационог слоја.

DAL (Data Access Layer)

- Садржи класе као што су `StudentDAL.cs` или `SmerDAL.cs`.
- Ту се налазе методе као што су:

    - `List<Student> GetAll()`
    - `Student GetById(int id)`
    - `void Insert(Student s)`
    - `void Update(Student s)`
    - `void Delete(int id)`
- Ове методе користе **ADO.NET** за комуникацију са **SQL Server-ом.**

Како сарађују:

- **Forma (UI)** → позива метод из `StudentDAL`
- `StudentDAL` → враћа или прихвата објекат `Student` (који долази из модела)
- Податак се приказује у форми (нпр. у `DataGridView`), или се шаље назад у базу

Пример:

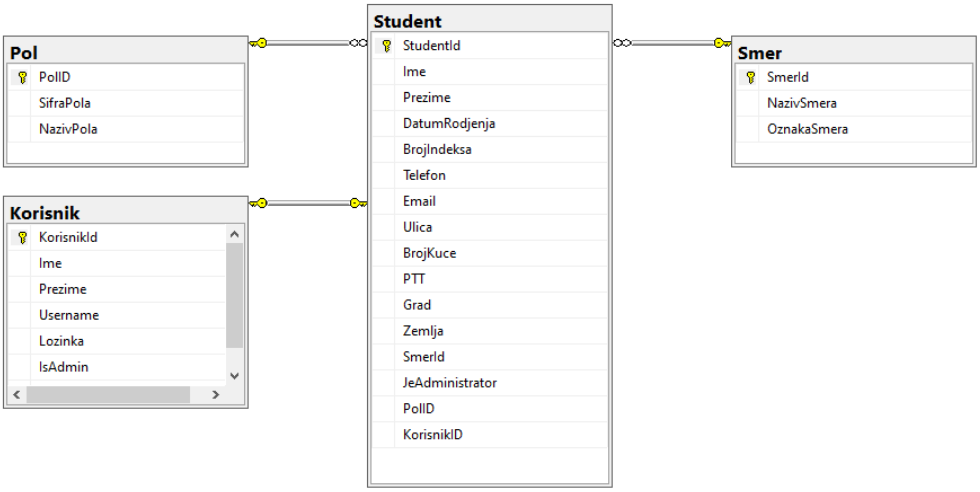
```
// Student.cs (Model)
public class Student {
   public int StudentId { get; set; }
   public string Ime { get; set; }
   public string Prezime { get; set; }
   public string Email { get; set; }
   public DateTime DatumRodjenja { get; set; }
}

// StudentDAL.cs
public class StudentDAL {
   public List<Student> GetAll() {
      // otvori konekciju, procitaj iz baze, vrati List<Student>
   }
}
```

д) ER дијаграм студената и повезаних ентитета (mini dijagram)

ER дијаграм (Entity-Relationship дијаграм) графички приказује ентитете као што су Студент, Смер, Пол и Корисник, и њихове међусобне релације у бази података, омогућавајући лакше разумевање структуре и повезаности података.

На **ER дијаграму** видимо да табела `Student` има везе ка табелама `Pol`, `Smer`, и `Korisnik` преко страних кључева (`PolID`, `SmerId`, `KorisnikID`). Ово омогућава нормализацију података и смањење дуплирања. Сваки студент има пол, припада једном смеру и везан је за једног корисника који га је унео или ажурирао. ER дијаграм визуелно представља структуру базе и њену повезаност са објектним моделом.



Ер дијаграм **SQL Server** базе података **СтудентскаЕвиденција**

3. ADO.NET – основни концепти

а) Шта је ADO.NET и .NET data provider

ADO.NET представља флексибилан **фрејмворк** у **.NET** екосистему за приступ подацима, где се користе два основна модела: **повезани (connected)** и **неповезани (disconnected)**. У оба случаја кључне компоненте су **интерфејси** :

- `IdbConnection` – успоставља и затвара везу,
- `IdbCommand` – извршава SQL команде и
- `IdataReader` – чита резултате упита

које конкретни провајдери (.NET data providers) реализују за различите СУБД-ове. Data provider за SQL Server укључује класе :

- `SqlConnection` – успоставља конекцију са сервером,
- `SqlCommand` – извршава SQL команде
- `SqlDataReader` – чита резултате упита и
- `SqlDataAdapter` – попуњава **DataTable и DataSet**

које обрађују све аспекте комуникације са сервером. Да би се користио **SQL Server .NET** снабдевач података, потребно је у врху кода укључити простор имена:

using System.Data – Укључава основне ADO.NET типове и интерфејсе ;
using System.Data.SqlClient – Овим простором имена приступамо свим специфичним класама за рад са Microsoft SQL Server-ом и његовим T-SQL механизмом.

У нашој апликацији о студентској евиденцији, уместо да свака WinForms форма директно „зна“ за **ADO.NET класе**, у нашем слоју за приступ подацима (`StudentDal.cs`) уводимо неопходне просторе имена и инкапсулирамо комплетну логику конекције и извршавања упита:

```
using System.Configuration;     // читање connectionString-а из App.config
using System.Data;              // DataTable, DataSet и опште ADO.NET типови
using System.Data.SqlClient;    // SqlConnection, SqlCommand, SqlDataReader…
```

```
namespace StudentskaEvidencija.DAL
{
   public static class StudentDal
   {
      private static readonly string Cnn =
         ConfigurationManager
           .ConnectionStrings["StudentskaEvidencija"]
           .ConnectionString;

      // Пример методе за читање свих студената
      public static List<Student> VratiSve()
      {
         var lista = new List<Student>();
         using(var con = new SqlConnection(Cnn))
         using(var cmd = new SqlCommand(
              "SELECT StudentId,Ime,Prezime,Email,DatumRodjenja FROM Student", con))
         {
            con.Open();
            using(var dr = cmd.ExecuteReader())
            {
               while (dr.Read())
               {
                  lista.Add(new Student {
                     StudentId    = dr.GetInt32(0),
                     Ime          = dr.GetString(1),
                     Prezime      = dr.GetString(2),
                     Email        = dr.GetString(3),
                     DatumRodjenja= dr.GetDateTime(4)
                  });
               }
            }
         }
         return lista;
      }
      // … INSERT, UPDATE, DELETE методе …
   }
}
```

Форма не уводи ниједан ADO.NET простран имена – довољно је :

```
using StudentskaEvidencija.DAL;
```

У `frm_Student_Load` позивамо само :

```
var lista = StudentDal.VratiSve();
dgvStudents.DataSource = lista;
```

b) Кључни простори имена (System.Data, System.Data.SqlClient)

У ADO.NET-у користе се **два главна простора** имена:

Простор имена `System.Data` садржи основне ADO.NET типове попут :

- `DataSet` – меморијски контејнер табела,
- `DataTable` – меморијска табела података,
- `DataRelation` веза између табела

и дефиниције за рад са подацима на апстрактном нивоу.

Уколико желимо да радимо конкретно са SQL Server-ом, кључујемо `System.Data.SqlClient`, где налазимо класе као што су :

- `SqlConnection` (за успостављање везе),
- `SqlCommand` (за извршавање упита) и
- `SqlDataReader` (за прослеђивање резултата од сервера).

Без овог другог простора имена **не можемо** да користимо ниједну **SQL Server**–специфичну функционалност у **C#** коду. Ова подела омогућава јасну разграничавање апстрактних и реализационих компоненти у **ADO.NET** слоју.

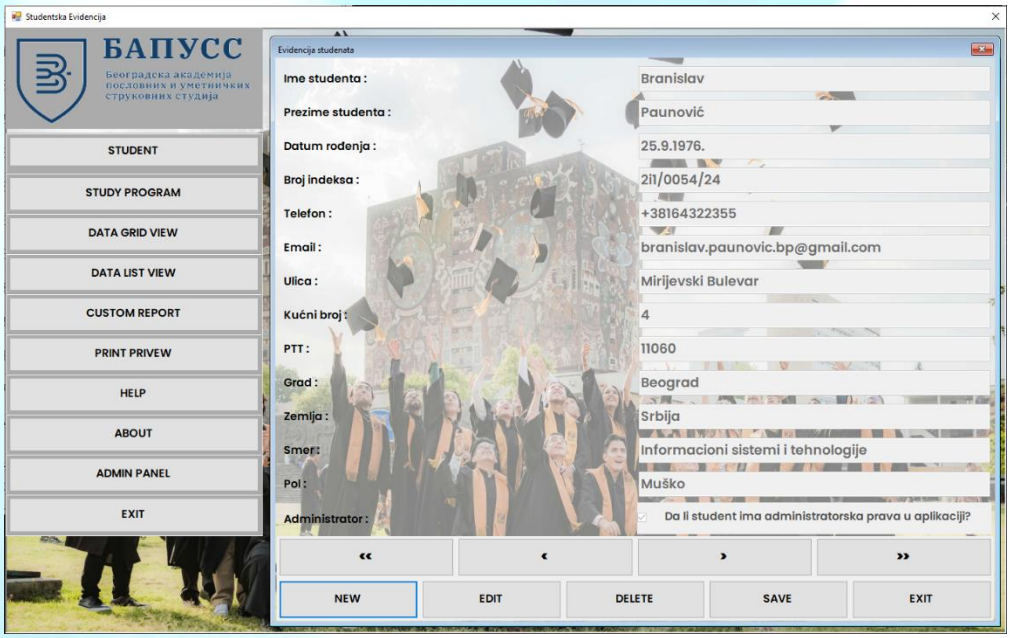
Да би апликација могла да користи SQL Server Data Provider, мора се укључити простор имена using System.Data.SqlClient;.

c) Разлика између повезаног и неповезаног начина рада

У **повезаном (connected)** начину рада апликација одржава активну везу са базом докле год обрађује податке. На пример, `SqlDataReader` чита ред по ред из базе директно док је конекција отворена.

У **неповезаном (disconnected)** начину рада подаци се учитавају у `DataSet` или `DataTable`, након чега се веза са базом затвара, а подаци се обрађују у меморији без активне везе. Ово је корисно када желимо да смањимо оптерећење сервера или да податке накнадно обрађујемо.

Форма `frm_Student` је класичан пример **повезаног (connected) сценарија.** Када корисник отвори ову форму, апликација успоставља везу са базом података преко `SqlConnection` и одмах користи `SqlCommand` и `SqlDataReader` да чита податке о студентима. Докле год је активна конекција, могуће је читати ред по ред из базе и приказивати их у формулару. Веза са базом се затвара тек након што се заврши читање. Ово омогућава високу тачност података у реалном времену.



Форма frm_Student

Пример кода који се користи у форми :

```
// Креирање објекта за повезивање са базом, користећи connection string
SqlConnection conn = new SqlConnection(connectionString);

// Креирање SQL команде која враћа све редове из табеле Student
SqlCommand cmd = new SqlCommand("SELECT * FROM Student", conn);

// Отварање конекције ка бази података
conn.Open();

// Извршавање команде и добијање SqlDataReader-а за читање ред по ред
SqlDataReader reader = cmd.ExecuteReader();

// Читање података из резултата упита, ред по ред
while (reader.Read())
{
   // Учитавамо поља из текућег реда и попуњавамо текст боксеве на форми
   txtIme.Text = reader["Ime"].ToString();
   txtPrezime.Text = reader["Prezime"].ToString();
   // ... овде се наставља са осталим пољима као што су email, datum рођења итд.
}

// Затварање reader-а пошто је читање завршено
reader.Close();

// Затварање конекције ка бази
conn.Close();
```

У овом примеру смо приказали како се може реализовати повезани (connected) начин рада са базом **података без коришћења DAL (Data Access Layer)**. Сав код за приступ бази налази се директно у форми frm_Student, што је карактеристично за мање апликације или почетне фазе развоја.

4. Управљање конекцијом ка бази

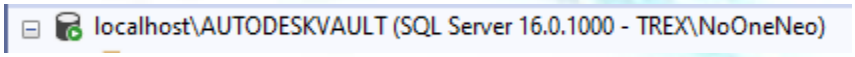
a) Класа SqlConnection

`SqlConnection` је основна класа **ADO.NET-а** која се користи за успостављање везе са **SQL Server** базом података. Најчешће се користи у комбинацији са `SqlCommand`, `SqlDataReader`, `SqlDataAdapter`, итд. Поседује важна **својства** као што су :

Najvažnija svojstva (`Properties`) klase `SqlConnection`:

Својсто	Тип	Објашњење
`ConnectionString`	`string`	Текстуални стринг који дефинише како се апликација повезује са базом (садржи име сервера, базе, лозинку, итд).
`Database`	`string`	Назив базе података на коју је тренутно отворена конекција.
`DataSource`	`string`	Назив SQL Server инстанце (на пример: `localhost\SQLEXPRESS`).
`ServerVersion`	`string`	Верзија SQL Server-а на који је апликација повезана.

Својство	Тип	Објашњење
`State`	`ConnectionState` enum	Стање конекције: може бити `Open`, `Closed`, `Connecting`, `Executing`, `Fetching`.
`ConnectionTimeout`	`int`	Време у секундама које апликација чека да се конекција успостави пре него што јави грешку.
`WorkstationId`	`string`	Име рачунара са кога се успоставља конекција према серверу.
`ClientConnectionId`	`Guid`	Јединствени идентификатор везан за ту конекцију (корисно за праћење и дијагностику)
`PacketSize`	`int`	Величина пакета у бајтовима који се шаљу између клијента и сервера (обично 8000 бајтова).

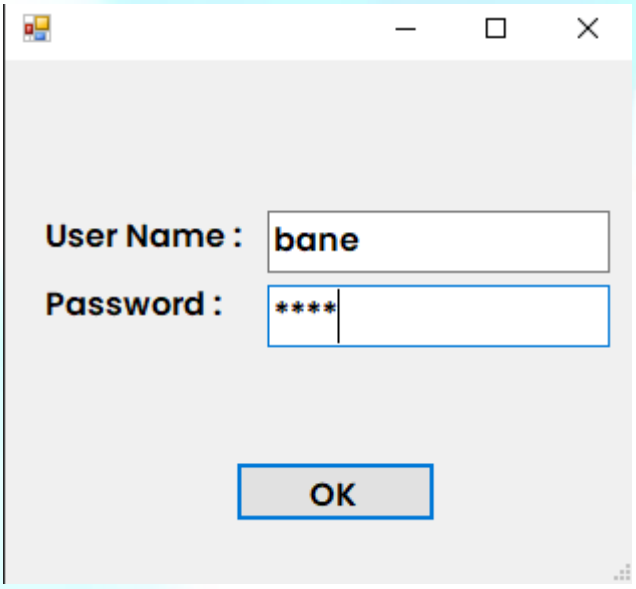


Слика својства `DataSource : localhost\AUTODESKVAULT` са својством `WorkstationId TREX\NoOneNeo` назив наше `SQL Server` инстанцне са називом наше радне станице на којој се налази наша `SQL Server` база података `StudentskaEvidencija`

Од **метода**, најзначајнији су:

- `Open()` – отвара везу са базом.
- `Close()` – затвара везу и ослобађа ресурсе.
- `BeginTransaction()` – покреће трансакцију за групу SQL операција.

Класа такође има **догађај** `StateChange`, који се активира када се промени стање конекције, што омогућава реакцију програма у реалном времену (нпр. приказ обавештења).



На слици је приказана `frm_Login` форма, служи као почетна тачка приступа апликацији, где се уноси **корисничко име** и **лозинка**. Њена основна сврха је **аутентификација корисника** – односно провера да ли постоји налог у бази који има дозволу да покрене апликацију.

Кораци које апликација извршава:

- Унос података: Корисник уноси username и password у два текстуална поља.
- Прво се проверава да ли је унета фиксна лозинка за администратора (корисно у случају да база није доступна).
- Ако није, форма затим успоставља везу са базом података преко SqlConnection, користећи ConnectionString из App.config.
- SQL упит проверава да ли у табели dbo.Korisnik постоји ред са тим подацима:

  ```
  SELECT COUNT(*) FROM dbo.Korisnik WHERE Username = @username AND Lozinka = @password
  ```

- Ако је резултат већи од 0, сматрамо да је корисник валидан и омогућавамо приступ.
- У супротном, приказује се порука о грешци и нуди се нови покушај.

Форма `frm_Login` представља прву тачку комуникације апликације са базом, где се путем класа `SqlConnection` и `SqlCommand` реализује провера унетих података корисника на основу табеле `Korisnik`. Кроз отворену конекцију и `ExecuteScalar()` метод проверавамо да ли у бази постоји корисник са датим корисничким именом и лозинком, при чему користимо параметре ради заштите од SQL injection напада. Овим демонстрирамо повезан (connected) начин рада са базом, као и сигурно руковање конекцијом кроз `try-catch-finally` блок.

b) ConnectionString (App.config / Settings)

`ConnectionString` је стринг у којем су дефинисани сви подаци неопходни за повезивање са базом (локација сервера, назив базе, начин аутентификације, timeout, итд).

Обично изгледа овако:

```
<connectionStrings>

 <add name="StudentskaEvidencija" connectionString="Data Source=localhost;Initial Catalog=StudentskaEvidencija;Integrated Security=True;" providerName="System.Data.SqlClient" />

</connectionStrings>
```

Уместо да се хардкодира у коду, најбоља пракса је да се ConnectionString чува у App.config фајлу (или Settings.settings), како би се могао лако мењати без потребе за рекомпајлирањем програма.

Из кода се затим чита преко:

```
string cs = ConfigurationManager.ConnectionStrings["StudentskaEvidencija "].ConnectionString;
```

Овај приступ доприноси безбедности, преносивости и лакшем одржавању апликације.

c) Windows vs SQL Server аутентификација

Windows Authentication користи налог тренутно улогованог Windows корисника за приступ бази. Најбезбеднији је у оквиру домена и користи `Integrated Security=True` у **ConnectionString**-у.

Пример:

Data Source=localhost;Initial Catalog=StudentskaEvidencija;Integrated Security=True;

SQL Server Authentication користи ручно задат `username` и `password`. Ово је чешћи избор на серверима ван домена или на хостингима који не подржавају интегрисану безбедност.

Пример:

Data Source=localhost;Initial Catalog=StudentskaEvidencija;User ID=sa;Password=1234;

У пракси, **Windows Authentication** је безбеднији, јер не захтева да лозинка буде експлицитно написана у фајловима.

Међутим, **SQL аутентификација** је флексибилнија и често се користи у cloud/remote кружењима.

5. Извршавање SQL команди

а) Класа SqlCommand

Класа `SqlCommand` омогућава извршавање SQL команди над базом података. Она користи својства попут :

`CommandText` (садржи SQL упит или име процедуре),
`CommandType` (обично `Text` или `StoredProcedure`),
`Connection` (SqlConnection објекат), и
`Parameters` (списак улазних/излазних параметара).

У форми `frm_Login` користимо `SqlCommand` за проверу да ли корисник постоји у табели `Korisnik`. Метода `ExecuteScalar()` враћа једну вредност – број редова који одговарају параметрима. Алтернативно, за INSERT/UPDATE/DELETE користимо `ExecuteNonQuery()`, а за читање више редова `ExecuteReader()`

– Параметри: CommandText, CommandType, Connection, Parameters
– Методе: ExecuteNonQuery(), ExecuteScalar(), ExecuteReader()

CommandType типови и објашњења:

1. **`CommandType.`**Text – (подразумевани тип)
   `SQL` команда је обичан текстуални `SQL` упит.
   Пример: `"SELECT * FROM Student"`
2. **`CommandType.StoredProcedure`**
   `SQL` команда је назив процедуре из базе.
   Пример: `"sp_GetAllStudents"` (а не `SQL` код директно)
3. **`CommandType.TableDirect`** (ретко се користи)
   `SQL` команда је име табеле.
   Пример: `"Student"`
   Овим се селектују сви подаци из табеле.

b) Параметризовани упити – Input и Output параметри

Параметризовани упити омогућавају сигурнију комуникацију са базом и спречавају SQL injection нападе.

У `frm_Login`, користимо `cmd.Parameters.AddWithValue()` да проследимо `@username` и `@password` без директног уметања у query string. Input параметри су вредности које шаљемо у базу, а Output параметри се користе када желимо да добијемо неку вредност из процедуре. Иако наш пример користи само input параметре, слично се додају и output параметри уз `SqlParameter.Direction = ParameterDirection.Output`.

Output параметри су параметри који враћају вредност из SQL Server-а назад у C# апликацију, без SELECT-а.

Уместо да податке враћаш као резултат упита (SELECT), користиш параметар који SQL пуни, а ти га у C# касније прочиташ.

Користе се код успешног додавања студента, па желимо да нам SQL врати ID новог студента као output. Пример једног output параметра на упрошћеној stored процедури за додавање студента :

```
CREATE PROCEDURE sp_DodajStudenta
   @Ime NVARCHAR(50),
   @Prezime NVARCHAR(50),
   @NoviStudentID INT OUTPUT
AS
BEGIN
   INSERT INTO Student (Ime, Prezime)
   VALUES (@Ime, @Prezime)

   SET @NoviStudentID = SCOPE_IDENTITY() -- враћа ID последње убачене ставке
END
```

C# Пример који позива ту процедуру :

```
using (SqlConnection conn = new SqlConnection(connString))
{
   SqlCommand cmd = new SqlCommand("sp_DodajStudenta", conn);
   cmd.CommandType = CommandType.StoredProcedure;

   cmd.Parameters.AddWithValue("@Ime", "Marko");
   cmd.Parameters.AddWithValue("@Prezime", "Marković");

   SqlParameter outputId = new SqlParameter("@NoviStudentID", SqlDbType.Int)
   {
      Direction = ParameterDirection.Output
   };
   cmd.Parameters.Add(outputId);

   conn.Open();
   cmd.ExecuteNonQuery();
   conn.Close();

   int noviId = (int)outputId.Value; // Ево ти output вредност!
   MessageBox.Show($"Успешно додат студент са ID: {noviId}");
}
```

c) Рад са ускладиштеним процедурама

Ускладиштене процедуре се креирају у SSMS (SQL Server Management Studio) под табом „Programmability > Stored Procedures“. У C#-у, постављамо `CommandType` на `StoredProcedure` и позивамо је преко `SqlCommand`. Параметре дефинишемо као и код класичних упита. Ако процедура враћа излазне вредности, читамо их након `ExecuteNonQuery()` преко `cmd.Parameters["@param"].Value`. Иако их у нашем пројекту нисмо користили, овај механизам се лако може применити у сложенијим операцијама (нпр. додавање студента преко procedure `sp_DodajStudenta`).

```
string query = "SELECT COUNT(*) FROM dbo.Korisnik WHERE Username = @username AND Lozinka = @password";
using (var cmd = new SqlCommand(query, conn))
{
   cmd.Parameters.AddWithValue("@username", username);
   cmd.Parameters.AddWithValue("@password", password);
   int count = (int)cmd.ExecuteScalar(); // враћа број редова
}
```

Овим примером смо приказали употребу класе `SqlCommand`, инпут параметара и методе `ExecuteScalar()` – директна примена теорије у нашем пројекту.

6. Читање података: SqlDataReader

a) Концепт read-only, forward-only курсора

`SqlDataReader` представља тип курсора који чита податке из базе у само једном смеру (forward-only) и не може се враћати назад нити мењати податке. То га чини изузетно брзим и ефикасним за читање великих количина података ред по ред.

У нашем примеру, у форми `frm_Student`, не користимо `SqlDataReader` већ `SqlDataAdapter` и `DataTable`, јер нам је потребна навигација напред-назад кроз све студенте, што `DataTable` омогућава лакше и флексибилније.

SqlDataAdapter је мост између базе података и објекта **DataTable** у меморији. Он извршава упит, попуњава податке у меморију **(DataTable)** и омогућава да радиш са тим подацима без сталне везе са базом – то се зове **неповезан (disconnected)** приступ. Подаци се једном учитају, па се форма (као што је frm_Student) може слободно кретати напред-назад кроз редове користећи **_currentIndex**, без додатног читања са сервера. Идеално је када ти треба комплетна табела (или више слогова) за приказ, претрагу, навигацију или уређивање, јер чува податке локално и не оптерећује базу.

b) Методе: Read(), IsDBNull(), GetString(), GetInt32() итд.

Метода `Read()` чита следећи ред из резултата – враћа `true` ако постоји ред. `IsDBNull()` проверава да ли је вредност у пољу `NULL`, да бисмо избегли грешке приликом конверзије. Методи као `GetString()`, `GetInt32()` или `GetDateTime()` служе за прецизно преузимање вредности одговарајућих типова. Пример: `reader.GetString(1)` узима вредност из друге колоне као стринг. У нашој форми `frm_Student` користимо `reader["Ime"].ToString()` или `reader.GetString(0)` за приказ имена студента у `txtIme.Text`.

c) Мапирање података у објекат класе Student

Када читамо више редова из табеле `Student`, логично је да сваки ред мапирамо у посебан објекат класе `Student`. То радимо тако што за сваки ред креирамо нови објекат: `Student s = new Student();`, па затим сетујемо својства као: `s.Ime = reader["Ime"].ToString();`. Тако добијамо листу студената коју можемо користити за приказ у `ListView`, `DataGridView`, или за даљу обраду. У нашем случају, ми их учитавамо редом и попуњавамо форму, што је једноставнији али практичан начин за почетак.

Пример мапирања у форми `frm_Student`
Унутар методе `ShowStudent(int index)`: :

```
var row = _studentsTable.Rows[index];
txtIme.Text = row["Ime"].ToString();
txtPrezime.Text = row["Prezime"].ToString();
txtDatumRodjenja.Text = Convert.ToDateTime(row["DatumRodjenja"]).ToShortDateString();
// ... остала поља
```

`row` представља један ред из `DataTable`.

За свако поље (нпр. `Ime`, `Prezime`, `DatumRodjenja`...) вредност из реда се **мапира у одговарајући TextBox** (нпр. `txtIme`, `txtPrezime`...).
Закључак:

Форма `frm_Student` **не користи SqlDataReader**, али **користи SqlDataAdapter да попуни DataTable**, а затим се кроз `ShowStudent()` **мапира сваки ред у форме** – што је чист пример **мапирања података из табеле у објекат (форму)**.

7. Data Access Layer (DAL)

Data Access Layer (DAL) је слој у апликацији који служи као посредник између базе података и остатка програма. У њему се налазе методе које шаљу упите ка бази и враћају резултате у виду објеката. На пример, у класи `SmerDAL`, метод `GetAllSmerovi()` чита податке из табеле `Smer` и враћа листу објеката класе `Smer`. На тај начин се логика приступа бази измешта из главног кода и омогућава лакше одржавање. DAL обезбеђује централно место за све упите ка бази.

a) Инстанцна класа SmerDAL за приступ бази (DAL)

Класа `SmerDAL` у нашем примеру је инстанцна класа која прима connection string кроз конструктор. Коришћење инстанцне класе омогућава већу флексибилност, као што је преношење различитих connection string-ова. У нашем примеру, `SmerDAL` је решен као обичан објекат класе.

```
public class SmerDAL
{
    private readonly string connectionString;

    public SmerDAL(string connString)
    {
        connectionString = connString;
    }

    public List<Smer> GetAllSmerovi()
    {
```

```
        List<Smer> smerovi = new List<Smer>();

        using (SqlConnection conn = new SqlConnection(connectionString))
        {
            string query = "SELECT * FROM Smer";
            SqlCommand cmd = new SqlCommand(query, conn);
            conn.Open();

            SqlDataReader reader = cmd.ExecuteReader();
            while (reader.Read())
            {
                Smer s = new Smer
                {
                    SmerId = Convert.ToInt32(reader["SmerId"]),
                    NazivSmera = reader["NazivSmera"].ToString(),
                    OznakaSmera = reader["OznakaSmera"].ToString()
                };

                smerovi.Add(s);
            }

            reader.Close();
        }

        return smerovi;
    }
    }
```

b) CRUD методе:

– List<Student> VratiStudente()
– int UbaciStudenta(Student s)
– int PromeniStudenta(Student s)
– int ObrisiStudenta(int id)

CRUD је скраћеница за **Create, Read, Update, Delete** – четири основне операције над подацима:

- `List<Student> VratiStudente()` → чита податке из базе (`Read`).
- `int UbaciStudenta(Student s)` → убацује новог студента у базу (`Create`).
- `int PromeniStudenta(Student s)` → ажурира постојећег студента (`Update`).
- `int ObrisiStudenta(int id)` → брише студента из базе по ID-ју (`Delete`).

У нашој апликацији прилком учитавања форме frm_DataGredView, ми отварамо конекциони стриг, читамо податке из табеле ред по ред и пунимо у објекте Student креирамо `List<Student>` за сваки нови обејкат типа Student који се додаје у `List<Student>`. Та листа се враћа UI слоју (форми) и ту листу прримамо преко LINQ пројекције креирамо нови анонимни објекат који се поставља као `DataSource` у `dgv_Student`. Након овога конекција се затвара. О овоме ће бити речи у следећем поглављу.

c) Try-Catch обрада изузетака у DAL слоју

У `DAL` слоју је важно обрадити потенцијалне грешке као што су проблеми са везом ка бази или грешке у упитима. То се ради помоћу `try-catch` блока. Унутар try блока иде логика рада са базом, а `catch` хвата и обрађује изузетке. У пракси, у `catch`-у можеш исписати грешку, логовати је или бацити нови изузетак вишем слоју. На пример:

```
try
{
    conn.Open();
    SqlDataReader reader = cmd.ExecuteReader();
}
catch (Exception ex)
{
    MessageBox.Show("Грешка при раду са базом: " + ex.Message);
}
```

8. WinForms кориснички интерфејс

WinForms, или *Windows Forms*, представља *Microsoft*-ову технологију за развој десктоп апликација у *.NET* окружењу. Кориснички интерфејс у *WinForms* апликацијама гради се око појма форме, која служи као главни прозор у оквиру ког се додају визуелне компоненте, попут **дугмади**, **текстуалних поља**, **табела**, **листа** и других контрола. Свака контрола има своја својства као што су **текст**, **боја**, **фонт** и **положај**, као и **догађаје** који реагују на корисничке акције, попут клика или уноса текста.
Форме и контроле могу се креирати визуелно, превлачењем елемената у **Visual Studio**-у, или програмски, кроз **C#** код. **WinForms** омогућава једноставно повезивање са базом података преко **ADO.NET** механизма, што омогућава рад са подацима у реалном времену. Податке можемо приказати у табеларном облику користећи `DataGridView`, или у виду листе користећи `ListView`, при чему се подаци повезују са класама из **DAL** (**Data Access Layer**) слоја.
Сваку корисничку акцију, као што је клик на дугме за чување или брисање, прати обрада у коду која користи **CRUD** методе за рад са подацима. Кориснички интерфејс у **WinForms** је прегледан, једноставан за коришћење и интуитиван, па се често користи за интерне пословне апликације и образовне пројекте. Иако се данас све више користе новији оквири попут **WPF**-а или **web** решења, **WinForms** остаје стабилан и поуздан избор за многе класичне десктоп системе.

a) Основне форме: frm_Student, frm_DataGreedView

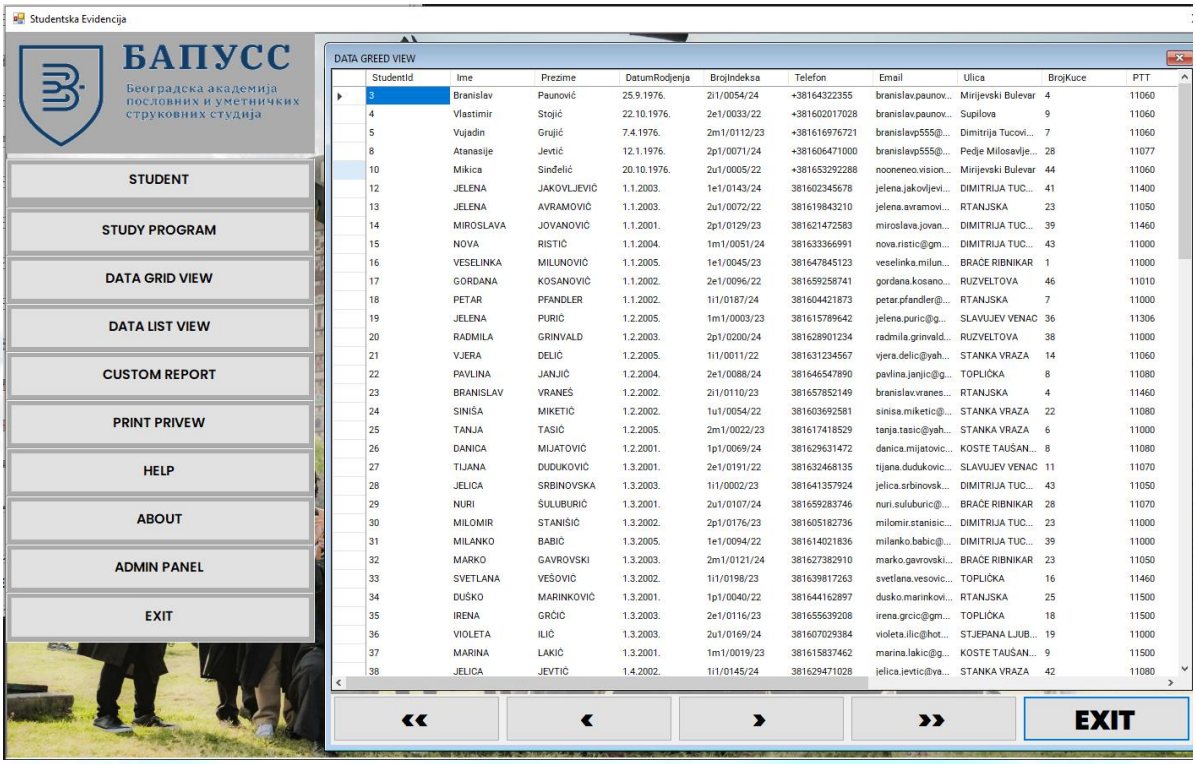
- `frm_Student` је главна форма за унос, измену и брисање података о студентима. Користи контроле `TextBox` и `CheckBox` за унос података, као и класе дугмета `btnAdd`, `btnEdit`, `btnDelete`, `btnSave` и `btnExit`. Њу смо већ видели у поглављу 3/c) Разлика између повезаног и неповезаног начина рада.

- `frm_DataGreedView` служи за приказ свих студената у форми DataGridView. Повлачи податке преко `StudentDAL` и користи LINQ пројекцију за додатне информације о смеру и полу.

- `frm_DataListView` приказује све студенте кроз ListView контролу. Садржи логику за попуњавање ListView преко DAL класа.

b) Приказ података:

b.1) ***DataGridView***
– Подешавање DataSource
– Конфигурација колона, избор редова, догађаји



Форма frm_DataGreedView

- `DataSource` се поставља на `LINQ` пројекцију `List`.
- Подешавање **DataSource** везе у нашем примеру пројекта StudentskaEvidencija ка localhost\AUTODESKVAULT :
    - Приликом иницијализације DAL класе (нпр. `StudentDAL`) у конструктору прослеђујемо connection string.
    - За повезивање на MSSQL Server на локалној машини користимо:
    - `string connString = "Data Source=localhost\autodeskvault;Initial Catalog=StudentskaEvidencija;Integrated Security=True;";`
    - Овај connection string поставља `DataSource` на инстанцу MSSQL Server-а `localhost\AUTODESKVAULT`, базу података `StudentskaEvidencija` и омогућава Windows аутентификацију преко `Integrated Security=True`
- Колоне се конфигуришу аутоматски или ручно.
- Избор редова се врши мишем, а навигација преко дугмади (`btnFirst`, `btnPrevious`, `btnNext`, `btnLast`).
- Подржава догађаје као што су `SelectionChanged`, `CellClick` и други
- Својство `Columns` дефинише појединачне колоне у `DataGridView` контроли. Оне могу бити аутоматски генерисане на основу `DataSource`, али уколико желимо већу контролу, можемо ручно додати колоне, подесити им заглавља (`HeaderText`), ширину, поравнање текста и видљивост. У нашем примеру, неке колоне попут `StudentId` се често сакривају јер нису релевантне за крајњег корисника.
- Својство `SelectionMode` контролише како се редови или ћелије бирају у `DataGridView`. У нашој форми `frm_DataGreedView` ово својство је постављено на `FullRowSelect`, што значи да се приликом клика на било коју ћелију бира цео ред. Ово је корисно јер желимо да корисник ради са целим студентским записом, а не са појединачним пољем.

Селектованом реду у DataGridView контроли (`frm_DataGreedView`) **приступамо** тако штоп приликом учитавања форме `frm_DataGreedView`, преко DAL слоја се повлаче подаци из базе и смештају у листу објеката типа `Student`, која се затим приказује у `DataGridView`

контроли `dgv_Student`. Да бисмо омогућили приступ и управљање селектованим редовима у овој табели, имплементирана је метода `SelectRow(int index)` која омогућава програмски избор конкретног реда на основу његовог индекса.

Ова метода прво проверава да ли је индекс валидан (у оквиру граница броја редова), затим чисти све претходне селекције, бира тражени ред, поставља тај ред као први видљиви на екрану и ажурира вредност промењиве `currentRowIndex`, која прати тренутно активни ред. Навигација кроз редове омогућена је дугмадима (`btnFirst`, `btnPrevious`, `btnNext`, `btnLast`) која позивају методу `SelectRow()` са одговарајућим индексом. На тај начин, корисник може прецизно да се креће кроз податке у табели, а апликација у сваком тренутку "зна" који ред је селектован и може на основу тога да изврши додатне радње, попут измене или брисања.

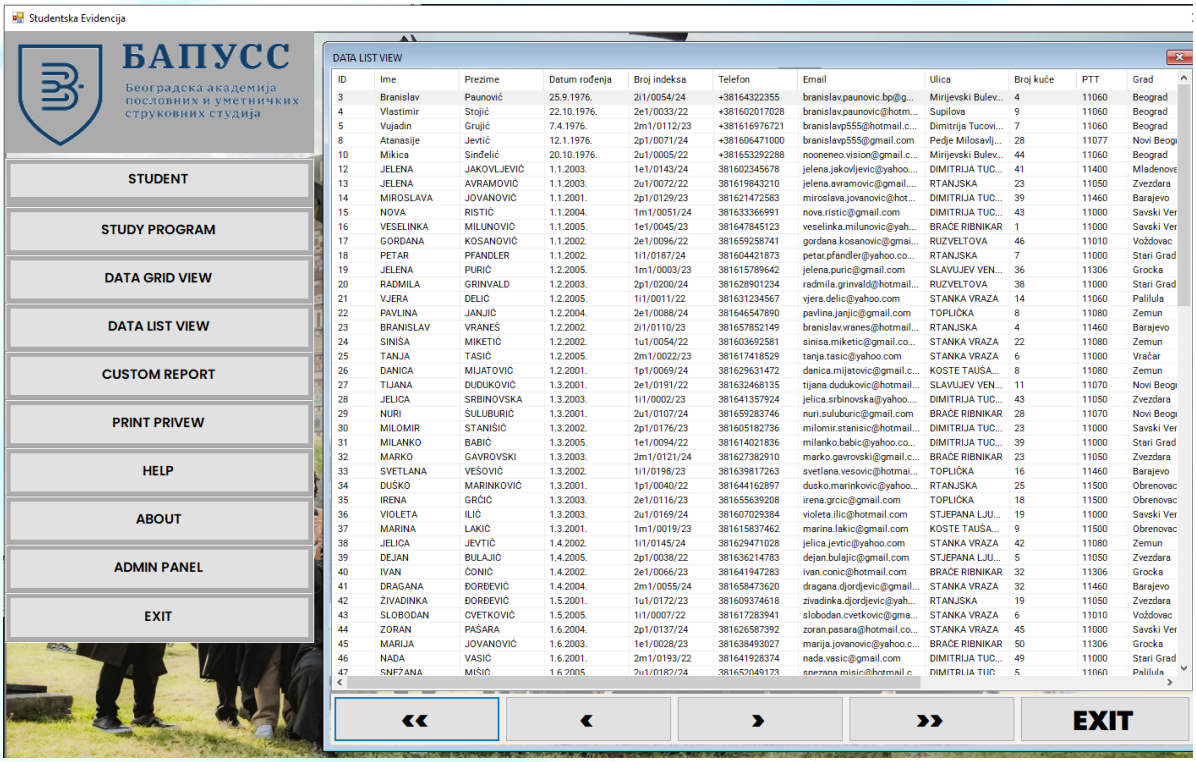
У нашој апликацији Studnetska Evidencija, цео DataGreed попуњавамо ручном логиком, програмским путем, користећи `DAL` и класу `Student`, а не преко визуелног дизајнера.

DataGreed контрола се такође може повезивати и попуњавати и преко визуелног дизајнера WinForm property-ja.

b.2) ***ListView***

– Режими приказа, својства: `Columns`, `Items`, `GridLines`

– Догађај: `SelectedIndexChanged`



Форма frm_DataListView

- Користи режим `Details` за приказ табеле.
- Својства:
  - `Columns` – дефинишу се имена колона
  - `Items` – додају се редови као ListViewItem
  - `GridLines` – приказује мрежу између редова и колона
  - `View` – својство одређује како ће ти подаци бити приказани :
    - `Details` – табеларни приказ са колонама (најчешће коришћено),

        - `LargeIcon`, `SmallIcon` – приказ са великим или малим иконама,
        - `List` – једноставна вертикална листа,
        - `Tile` – плочице са више информација по ставци.
    - `MultiSelect` у `ListView` контроли дефинише да ли корисник може изабрати више ставки истовремено. Ако је `MultiSelect` = `true`, корисник може држати `Ctrl` или `Shift` и кликати више редова. Ако је `false`, може бити изабрана само једна ставка у исто време.

- Догађај `SelectedIndexChanged` омогућава праћење избора корисника.

Сваки ред у `ListView` контроли представљен је преко објекта класе `ListViewItem`. Ова класа омогућава да додаш основни текст (тзв. `caption`), као и податке за додатне колоне преко `SubItems`. Такође можеш доделити слику преко `ImageIndex` или `ImageKey`, као и означити ставку као селектовану (`Selected` = `true`).

c) Повезивање UI и DAL метода

- UI слој (форме) иницира позиве ка DAL методама (`GetAllStudents()`, `UbaciStudenta()`, `ObrisiStudenta()`, итд.).
- DAL класе (нпр. StudentDAL, SmerDAL, PolDAL) враћају објекте који се даље приказују на формама.
- Комуникација се заснива на преносу података кроз објекте модела (нпр. Student, Smer).

9. Валидација и обрада грешака

a) Валидација уноса: string.IsNullOrWhiteSpace()

Валидација представља процес провере да ли су подаци које корисник уноси у апликацију исправни и у дозвољеном формату пре него што се обраде или пошаљу у базу. Метода `string.IsNullOrWhiteSpace()` у .NET-у служи да проверимо да ли је неки текстуални унос празан, садржи само размаке или није унет уопште. Ово је нарочито важно код уноса имена, презимена, броја индекса и других података који не смеју остати празни. Ако би се овакви подаци унели у базу без валидирања, могли би нарушити интегритет података или изазвати грешке. У пракси, ову методу користимо пре чувања података — ако врати `true`, приказујемо поруку кориснику да поље мора бити попуњено.

b) Try-Catch блокови при раду са базом

`try-catch` блок у **C#**-у омогућава да се ухвате и обраде изузеци (грешке) које се могу десити током извршавања програма, нарочито при раду са базом података. У блоку `try` се постављају операције које могу бацити изузетак, као што су отварање конекције, слање упита или читање података. Ако дође до грешке (на пример, база није доступна или је погрешан упит), програм скаче у `catch` блок, где можемо исписати грешку, логовати је или обавестити корисника. Ово спречава да апликација падне и пружа контролисан начин за реаговање. У озбиљним апликацијама, `try-catch` је обавезан код сваког рада са `SqlConnection`, `SqlCommand` и сличним објектима.

c) Приказ грешака кориснику (MessageBox, error provider...)

Када се открије грешка или невалидан унос, важно је да апликација обавести корисника на јасан и ненаметљив начин. За то користимо различите механизме — најједноставнији је `MessageBox.Show()`, који отвара дијалог са поруком. За визуелно означавање грешке директно на форми користимо `ErrorProvider`, који приказује црвени кружић поред поља са

неважећим подацима, уз објашњење на догађај `MouseOver`. Ово омогућава боље корисничко искуство, јер корисник одмах види шта није у реду и где. Поред тога, могуће је користити и `Label` контроле за динамичко приказивање порука о грешкама на самој форми. Кључно је да порука буде разумљива, кратка и тачна, како би корисник могао брзо да исправи унос.

10. Закључак

a) Шта је постигнуто у раду

У овом раду успешно је реализована **WinForms** апликација за евиденцију студената, која омогућава све основне **CRUD** операције – унос, измену, брисање и преглед података – уз логин систем и централни мени за лаку навигацију. Апликација је заснована на стабилној и добро организованој објектно оријентисаној архитектури која укључује више ентитета као што су `Student`, `User`, `Adresa`, као и помоћну класу `RecordNavigator` која омогућава навигацију кроз податке. Посебан акценат стављен је на раздвајање логике по слојевима – посебно се издваја **Data Access Layer** (**DAL**) који енкапсулира све **SQL** операције, што омогућава лакше тестирање, одржавање и проширење апликације у будућности. У оквиру рада успешно су примењене **ADO.NET** компоненте као што су `SqlConnection`, `SqlCommand`, `SqlDataReader` и `SqlDataAdapter`, а кориснички интерфејс је реализован у више форма са визуелно прегледним контролама (`DataGridView`, `ListView`) и пријатним **UX**-ом. Захваљујући овим техникама и структурама, добијен је један компактан, функционалан и стабилан систем.

b) Кључне лекције (ADO.NET, WinForms, SQL)

Током рада на пројекту, стекнуте су бројне техничке и архитектонске лекције. **ADO.NET** је показао колико је моћан алат за рад са базом, како у `connected`, тако и у `disconnected` режиму, док је коришћење параметаризованих упита показало своју важност у заштити од **SQL injection**-а. Конекцијски низови су постављени у конфигурациони фајл (`App.config`), што је допринело безбедности и лакшем преношењу апликације. **WinForms** интерфејс се показао као добар избор за почетничке и средње комплексне апликације, али захтева пажљиво повезивање **UI** компоненти са логиком у позадини. Кроз рад су усвојени важни принципи доброг софтверског инжењеринга, попут **SOLID** принципа и раздвајања одговорности између **UI** и **DAL** слоја. Такође, посебна пажња је посвећена валидацији уноса коришћењем `string.IsNullOrWhiteSpace()` и обради изузетака кроз `try-catch` блокове, уз коришћење `MessageBox` и `ErrorProvider` компоненти за информисање корисника.

c) Могућности проширења (претрага, логин, више ентитета)

И ако је основна функционалност успешно реализована, структура апликације отворена је за бројна проширења. Једна од првих надоградњи могла би бити претрага и филтрирање података на формама, коришћењем **LINQ** или **SQL WHERE** клаузула за ефикасније проналажење записа. Даље, имплементација улога корисника и система ауторизације би омогућила контролу приступа – на пример, само администратори би имали дозволу за брисање или измену података. Апликација се такође може проширити додавањем нових ентитета у базу података и **ER дијаграм**, као што су „Предмет“, „Оцена“, „Испит“, што би систем трансформисало у комплетан академски информациони систем. Поред тога, додатак модула за генерисање извештаја у **PDF** или **Excel** формату, као и опција штампања докумената, значајно би подигли корисничку вредност апликације. На крају, уколико се покаже потреба за модернијим интерфејсом и бољом интеграцијом са веб технологијама, постоји простор за миграцију ка **WPF** платформи или чак веб апликацији заснованој на **ASP.NET MVC** или **Blazor** технологијама.