Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота
за освітнім ступенем «магістр»

на тему: РОЗРОБКА НАСТІЛЬНОГО ДОДАТКУ «ОСОБИСТА КАРТКА СТУДЕНТА» НА ОСНОВІ ТЕХНОЛОГІЇ WINDOWS PRESENTATION FOUNDATION

Виконала: Магістрантка 2 курсу
групи М-КН-21
спеціальності 122 Комп’ютерні науки
Бурка Ю. М.
Науковий керівник: к.ф.-м.н, професор
Шахрайчук М. І.

Рівне - 2022

ЗМІСТ

ВСТУП

Актуальність роботи. Миттєвий розвиток інформаційних технологій дав великий поштовх для вдосконалення усіх сфер суспільства.

Величезні обсяги інформації, що обробляються в державних та приватних установах, від офісу малого підприємства до офісу великої корпорації, переважно спрямовані на створення управлінських документів. А управлінські документи, в свою чергу, спрямовані на прийняття управлінських рішень, що є цільовою функцією кожної установи.

Впровадження автоматизованих інформаційних систем (ІС) у діяльність різних установ створює великі можливості для підвищення якості документообігу та надає змогу покращити продуктивність та якість управлінської праці.

Автоматизація будь-якого елементу навчального процесу, навіть у вищих навчальних закладах, є дуже актуальною.

Однією з найвідповідальніших ділянок діяльності у вищому навчальному закладі є деканат.

Працівникам деканату доводиться опрацьовувати величезну кількість інформації щодо реєстрації студентів, забезпечуючи комфортний навчальний процес та надаючи інформацію іншим підрозділам вищого навчального закладу. Крім того, вся інформація повинна надаватися в різних формах. Очевидною є потреба у впровадженні інформаційної системи, яка автоматизує основні функції навчального процесу. Але перш ніж почати впровадження автоматизованої інформаційної системи для опрацювання даних про студента (АІС) в деканаті, як і в будь-якому іншому підрозділі ВНЗ або будь-якій іншій установі, необхідно визначити головні вимоги до її функціонування. Основою для визначення цих вимог є узагальнення, які будуть отримані за результатами вивчення предметної області діяльності студентів у межах своїх деканатів.

Метою роботи є: побудова та створення єдиного програмного середовища, яке повинно автоматизувати управлінську діяльність деканату, замінити паперовий документообіг на електронний та зводить до мінімуму кількість помилок під час роботи і дає змогу сконцентрувати більше уваги на виконанні інших функціональних обов'язків.

Для досягнення сформульованої мети були поставлені та вирішені наступні **завдання**:

- аналіз джерельної бази даних студентів;
- на основі опрацьованих даних, виділити найнеобхіднішу інформацію для створення бази даних та подальшої можливості роботи з попередньо згаданою БД;
- засобами WPF та мовою C# створити настільний додаток, з можливістю додавати дані про студентів, опрацьовувати їх, здійснювати пошук за ім'ям та прізвищем або за групою, в якій навчається певний студент.

Об'єкт дослідження: предметна область і проблемне середовище деканатів.

Предмет дослідження: моделювання предметної області та керування базою даних АІС за допомогою програмного додатку «Особиста картка студента».

Для реалізації поставлених задач було проведено розбір необхідних та ефективних областей застосування об'єктно-орієнтованих мов програмування та технологій, що на них базуються. Для розробки додатку були використані наступні мови та засоби програмування: C#, WPF, XAML, JSON та з допомогою реляційних баз даних в середовищі SQL Server Management Studio (SSMS).

Структура роботи. Дипломна робота складається зі вступу, чотирьох розділів, висновків, переліку використаних джерел та додатків. У першому розділі розглянуто призначення бази даних, її основні елементи та можливості роботи з інформацією в межах інформаційної системи. Другий розділ містить огляд усіх необхідних засобів: мов програмування, підсистем, платформ та

середовищ для програмної реалізації. Третій розділ висвітлює основну інформацію (функції та базові компоненти розмітки), яку було застосовано під час написання коду на мові XAML. Інформація про розробку програми, створення елементів WPF та дизайну для кнопок у вікнах застосунку, а також налагодження програмного коду містяться у четвертому розділі.

Загальний обсяг роботи становить 65 сторінок. Вона містить 15 рисунків. Список використаних джерел включає 17 найменувань. Обсяг додатків – 10 сторінок.

РОЗДІЛ 1. РЕЛЯЦІЙНА БАЗА ДАНИХ

У перші роки існування баз даних кожна програма зберігала дані в унікальній структурі. Коли необхідно було створювати додатки для використання цих даних, розробникам доводилося знати багато деталей про конкретну структуру даних для того, щоб знайти необхідні дані. Ці структури даних були неефективними, ними було важко керувати і їх було важко оптимізувати для забезпечення належної продуктивності додатків. Для вирішення проблеми множинних довільних структур даних була розроблена реляційна модель бази даних. Але, перш ніж описати вище зазначену модель, варто розглянути, що ж являють собою бази даних в цілому.

Отож, база даних - це інтегрована сукупність добре структурованих і взаємопов'язаних даних, організованих за певними нормами, що містять загальні принципи опису, зберігання та обробки даних [15].

Бази даних (БД) використовуються в інформаційних системах, що автоматизують діяльність предметної області.

Інформаційна система - це сукупність технічних та програмних методів, засобів та людей, які забезпечують збір, зберігання, опрацювання та пошук необхідної інформації для вирішення завдань.

Проектування бази даних передбачає декілька етапів:

1. Аналіз вимог. Враховуються функціональні можливості, необхідні системі, збираються та опрацьовуються потреби користувачів, на основі яких екстраполюються вимоги, яким повинна відповідати система.

2. Концептуальне проектування. Опис дійсної ситуації, для якої розробляється система, яка здійснюється з використанням структур сутностей і зв'язків. Результатом є концептуальна схема.

3. Логічний дизайн. Це полягає в розробці бази даних таким чином, щоб вона абсолютно і ефективно відповідала раніше реалізованій

концептуальній схемі. Результатом є певна логічна схема і, відповідно, вибір СУБД - системи управління базами даних, яка буде використовуватися.

4. Фізичний дизайн. На цьому етапі визначаються структури зберігання таблиць та допоміжні структури доступу до даних - індекси. Логічна схема даних переводиться у фізичну схему.

5. Реалізація бази даних. Створюються таблиці та впроваджуються додатки.

Кожна фаза передбачає послідовне проектування і включає в себе набір необхідних призначень.

На етапі аналізу та визначення вимог спочатку необхідно провести, так зване, досудове розслідування, в ході якого повинні бути дотримані вимоги користувачів, аналізуючи вихідну ситуацію і, нарешті, виробляючи вказівки щодо того, як моделювати еталон реальності. На основі спостережень та оцінок робиться більш детальне визначення, яке може вказувати, наприклад, на різні види користувачів, залучених до системи, умови, які повинні бути виконані для того, щоб дані були значущими та дійсними (обмеження цілісності) тощо.

Завдання концептуального проектування полягає в тому, щоб взяти на вхід неформальні специфікації еталонної реальності, щоб створити формальний і повний опис, який, однак, не залежить від представлення, що використовується в СУБД. Іншими словами, концептуальний дизайн спрямований на представлення інформаційного змісту бази даних, не турбуючись про фактичну реалізацію. Результатом цього етапу є так звана концептуальна схема або E/R діаграма.

Логічне проектування бере за основу концептуальну схему, створену на попередньому етапі, і переводить її в логічну схему, яка позначає "сімейство" баз даних, що використовуються. Логічний дизайн, таким чином, залежить від моделі представлення даних, що використовуються. Зазвичай посилаються на реляційну логічну модель, але не можна не згадати, що є й інші моделі, більш чи

менш вживані. До них відносяться ретикулярна модель, об'єктна модель, XML модель та модель NoSQL.

Фізичне проектування приймає на вхід логічну схему, визначену на етапі логічного проектування, і доповнює її специфікацією параметрів для зберігання даних, які в основному стосуються організації файлів та індексів. Фізична модель сильно залежить від використовуваної СУБД, але може відрізнятися і в межах СУБД одного сімейства.

Етап впровадження включає компіляцію та виконання команд для створення бази даних мови визначення даних, автоматичне або ручне заповнення бази даних. Дані можуть бути конвертовані з існуючого формату та впровадженими в прикладні програми, щоб користувачі в подальшому могли отримати доступ до даних та маніпулювати ними у контрольований спосіб [2].

1.1. Складові елементи бази даних

Розуміння призначення бази даних характеризуватиме селекція в процесі проектування. Обов'язково необхідно першочергово розглянути базу даних з усіх точок зору. Наприклад, якщо створюється інформаційна база даних для публічної бібліотеки, слід подумати про те, як користувачі та бібліотекари можуть отримати доступ до даних [4]. Також, необхідно зібрати усі дані та визначити до якого типу даних буде належати той чи інший елемент кожної таблиці бази даних. Після того, як визначено типи даних, які буде включати база даних, звідки вони будуть надходити і як вони будуть використовуватися, можете приступити до планування самої бази даних.

У базі даних відповідні дані згруповані в таблиці, кожна з яких складається з рядків (також званих кортежами) і стовпців, подібно до електронної таблиці.

Щоб перетворити списки даних у таблиці, необхідно почати зі створення таблиці для кожного типу об'єктів, таких як студент, група, успішність та додаткова інформація (Рис. 1, 2).

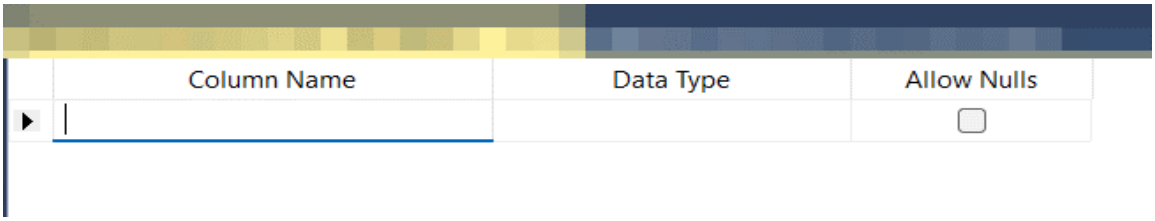


Рис. 1. Створення нової таблиці даних

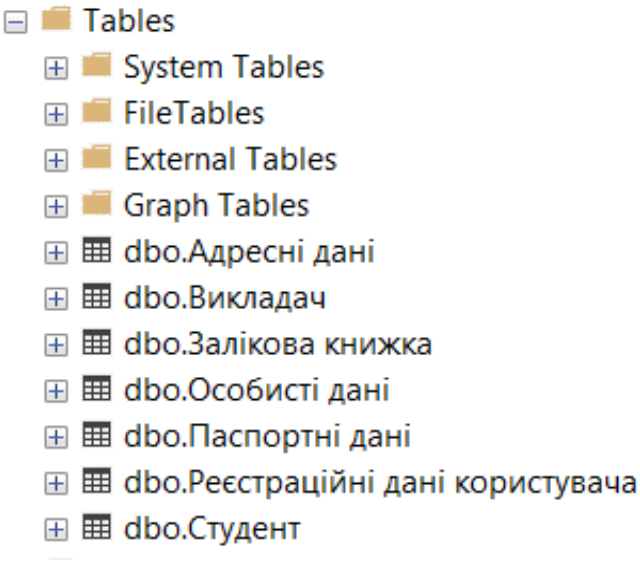


Рис. 2. Перелік таблиць необхідних для роботи з даними застосунку

Кожен рядок таблиці називається записом. Записи містять дані про щось або когось, наприклад, про конкретного студента. На відміну від цього, стовпці (також відомі як поля або атрибути) містять єдиний тип інформації, яка з'являється в кожному записі, наприклад, факультет, назви яких перелічено у таблиці.

Для прикладу, проілюструємо таблицю даних з інформацією про студента, яку необхідно віднести до таблиці з загальними даними пов’язаними з освітнім процесом (Рис.3):

Column Name	Data Type	Allow Nulls
Прізвище	nchar(10)	☐
[Ім'я]	nchar(10)	☐
[По-батькові]	nchar(20)	☑
Група	nchar(10)	☐
Курс	int	☐
Спеціальність	nchar(10)	☑
[Залікова книга]	uniqueidentifier	☐
Куратор	nchar(40)	☑
Факультет	nchar(40)	☑
[Навчальний ступінь]	nchar(10)	☑
[ID студента]	uniqueidentifier	☐
		☐

Рис. 3. Таблиця, яка містить освітні дані про студента

Щоб забезпечити узгодженість даних від одного запису до іншого, необхідно призначити кожному стовпчику відповідний тип даних. До поширених типів даних відносяться (Рис.4):

- char - текст певної довжини
- varchar - текст різної довжини
- text - великі обсяги тексту
- int - додатне або від'ємне ціле число
- float, double - також можуть зберігати числа з плаваючою комою
- blob - двійкові дані

Деякі системи управління базами даних, наприклад, Dynamics 365, також пропонують тип даних Autonumber, який автоматично генерує унікальний номер у кожному рядку.

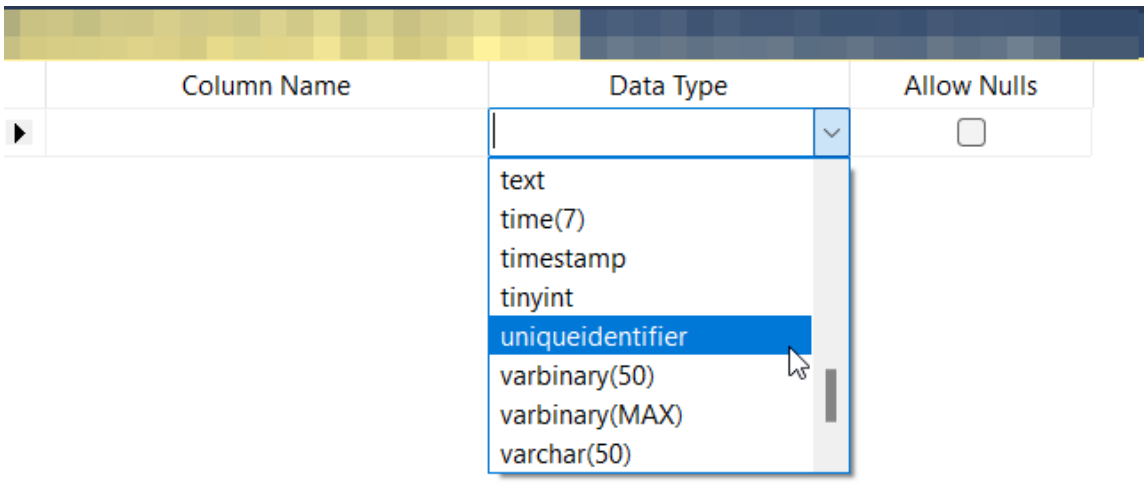


Рис. 4. Тип даних «Ідентифікатор» в MSQLS

З метою створення візуального огляду бази даних, відомого як діаграма "сутність-зв'язок", фактичні таблиці не будуть включені. Натомість кожна таблиця стане квадратиком на діаграмі. Назва кожної графи повинна вказувати на те, що описують дані в цій таблиці, а атрибути перераховані нижче, наприклад, таким чином (Рис.5):

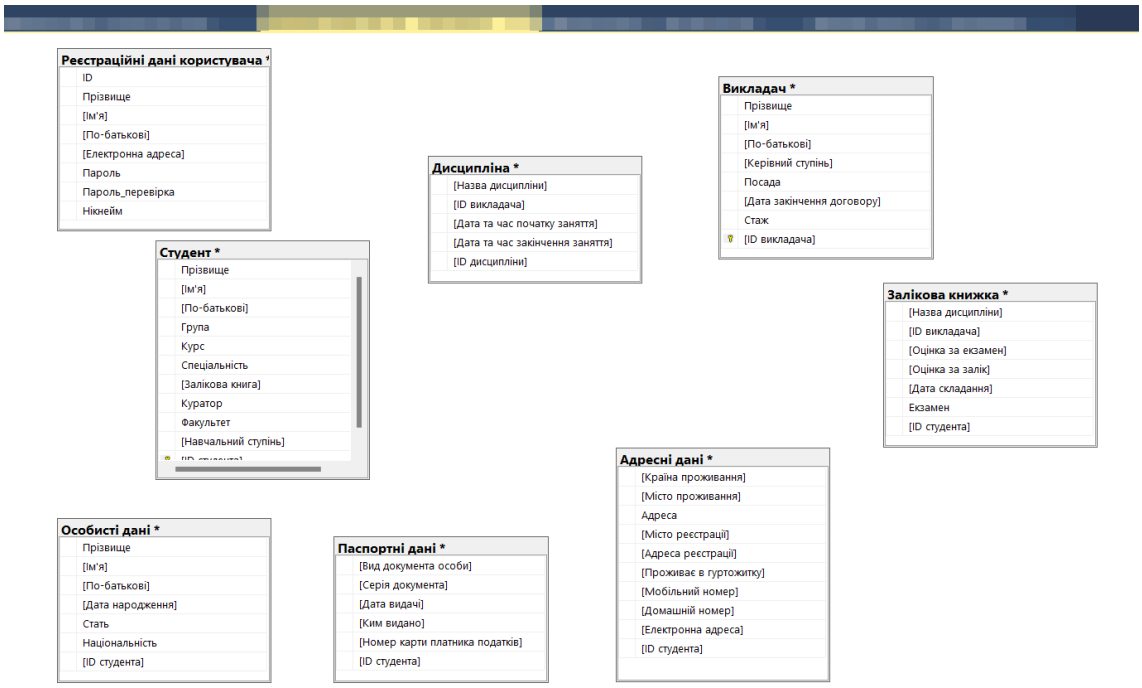


Рис. 5. Діаграма БД проекту без зв’язків

Найголовніше, що необхідно вирішити, який атрибут або атрибути будуть служити первинним ключем для кожної таблиці, якщо такі є. Первинний ключ (ПК) - це унікальний ідентифікатор для даної сутності, що дає можливість вибрати точного студента з усієї таблиці даних, знаючи тільки це єдине значення.

Атрибути, обрані в якості первинних ключів, повинні бути унікальними, незмінними та завжди присутніми (ніколи не повинні мати значення NULL або пустими). З цієї причини номери залікових книжок або повні імена студентів є ефективними первинними ключами, тоді як позначення групи або домашньої адреси не є такими (Рис.6). У БД також є можливість використовувати кілька полів разом як первинний ключ (так званий складений ключ).

Column Name	Data Type	Allow Nulls
Прізвище	nchar(10)	☐
[Ім'я]	nchar(10)	☐
[По-батькові]	nchar(20)	☑
Група	nchar(10)	☐
Курс	int	☐
Спеціальність	nchar(10)	☑
[Залікова книга]	uniqueidentifier	☐
Куратор	nchar(40)	☑
Факультет	nchar(40)	☑
[Навчальний ступінь]	nchar(10)	☑
[ID студента]	uniqueidentifier	☐
		☐

Рис. 6. Поле «Залікова книга», відображене в якості первинного ключа

Коли прийде час створювати фактичну базу даних, ви введете як логічну структуру даних, так і фізичну структуру даних на мові визначення даних, що підтримується вашою системою управління базами даних. На цьому етапі слід

також оцінити розмір бази даних, щоб бути впевненим, що ви зможете досягти необхідного рівня продуктивності та обсягу пам'яті.

На моменті необхідності створення фактичної бази даних, необхідно буде ввести як логічну структуру даних, так і фізичну структуру даних на мові визначення даних, що підтримується системою управління базами даних, якою користується програміст. На цьому етапі слід також оцінити розмір бази даних, щоб бути впевненим в можливості досягти необхідного рівня продуктивності та обсягу пам'яті.

1.2. Взаємовідносини між суб’єктами бази даних

На етапі, коли таблиці бази даних є створеними і добре організованими, можна приступити до аналізу зв'язків між ними. Кардинальність відноситься до кількості елементів, які взаємодіють між двома пов'язаними таблицями. Визначення кардинальності дозволяє більш ефективно розбивати дані на таблиці.

Кожен суб'єкт БД (entity) потенційно може мати відношення до іншого, але ці відношення, як правило, бувають трьох типів:

Відношення "один до одного". Коли існує лише один екземпляр Об'єкта А на кожен екземпляр Об'єкта Б, вважається, що вони мають відношення "один до одного" (часто пишуть 1:1). Даний тип зв'язку в ER-моделі можна позначити лінією з дефісом на кожному кінці. Якщо немає вагомих причин не робити цього, співвідношення 1:1 зазвичай вказує на те, що було б краще об'єднати дані з двох таблиць в одну таблицю.

Однак, за певних обставин можна створити таблиці зі співвідношенням 1:1. Для прикладу, таблиця містить поле з необов'язковими даними "опис", яке є порожнім для багатьох записів. Таким чином, можна перемістити всі описи в окрему таблицю, усунувши порожній простір і підвищивши продуктивність бази даних.

Для того, щоб забезпечити коректну відповідність даних, слід додати принаймні один ідентичний стовпець у кожну таблицю, швидше за все, первинний ключ.

Відношення "один до багатьох". Ці зв'язки виникають, коли запис, що міститься в одній таблиці, пов'язаний з декількома записами в іншій. Наприклад, один студент може мати багато іспитів або студент міг взяти в бібліотеці кілька книг одночасно. Зв'язки "один до багатьох" (1:M) позначаються так званою "нотацією воронячих лапок".

Щоб реалізувати відношення 1:M при конфігуруванні бази даних, просто необхідно додати первинний ключ з "однієї" сторони відношення як атрибут в іншу таблицю. Коли первинний ключ вказаний таким чином в іншій таблиці, він називається зовнішнім ключем. Таблиця на "одній" стороні зв'язку вважається батьківською по відношенню до дочірньої таблиці на іншій стороні.

Відношення "багато до багатьох". Коли декілька сутностей в одній таблиці можуть бути пов'язані з декількома сутностями в іншій таблиці, кажуть, що вони мають зв'язок "багато до багатьох" (M:N). Це може статися у випадку студентів та класів, оскільки студент може відвідувати багато курсів, а клас може мати багато студентів. На жаль, реалізувати такий тип зв'язку безпосередньо в базі даних неможливо. Натомість, необхідно розбити його на два відношення "один до багатьох". Для цього необхідно створити нову сутність між цими двома таблицями. Якщо між студентами та іспитами існує зв'язок M:N, можна назвати цю нову сутність "студенти_складеніІспити", оскільки вона буде показувати зміст кожного продажу. Обидві таблиці студентів та іспитів будуть мати зв'язок 1:M з студенти_складеніІспити. Цей тип проміжної сутності в різних моделях називається таблицею зв'язків, асоціативною сутністю або таблицею приєднання. Кожен запис у таблиці зв'язків відповідатиме двом об'єктам у сусідніх таблицях (він також може містити додаткову інформацію).

Іноді таблиця вказує сама на себе. Наприклад, таблиця викладачі може мати атрибут "доцент", який посилається на іншу особу в тій самій таблиці. Це називається рекурсивним відношенням.

Надлишкове відношення - це відношення, виражене більше одного разу. Зазвичай можна видалити один із зв'язків без втрати важливої інформації. Наприклад, якщо сутність "студенти" має прямий зв'язок з іншою сутністю під назвою "викладачі", але також має зв'язок з викладачами опосередковано через "пари", у програміста є можливість видалити прямий зв'язок між "студентами" та "викладачами". Найкраще рішення буде усунути цей зв'язок та закріпити учнів за вчителями через таблицю пари (Рис.7).

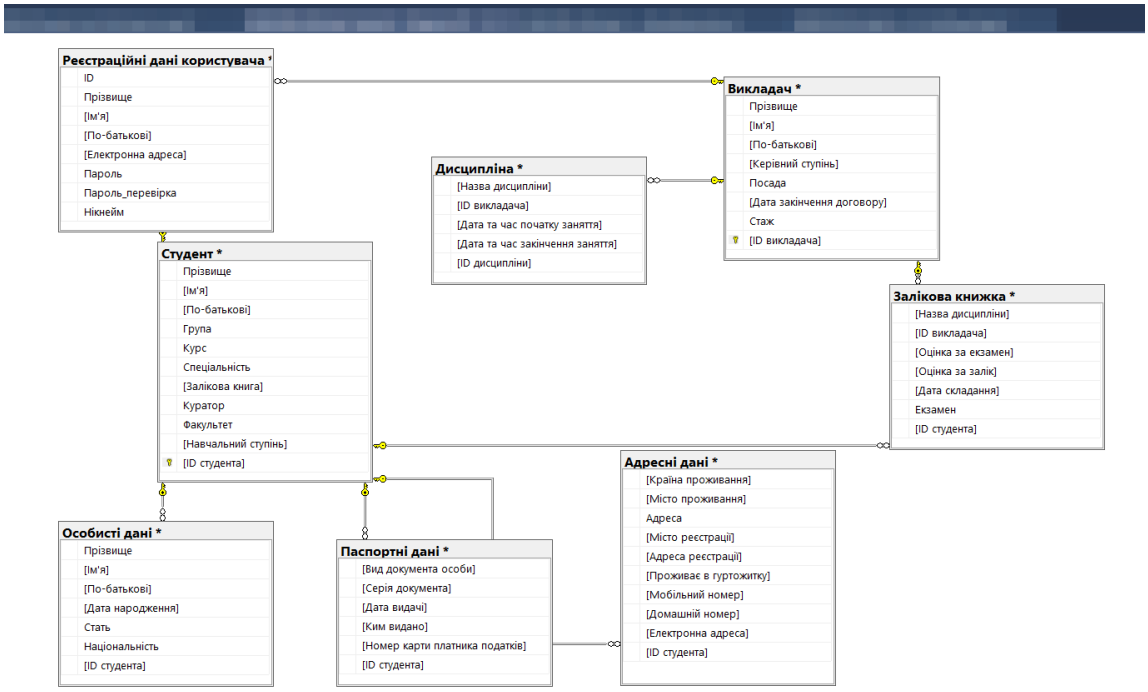


Рис. 7. Діаграма БД проекту зі зв’язками

Після дослідження та вибору необхідних відношень між таблицями, наступним етапом повинна виконуватися нормалізація бази даних, щоб переконатися, що створені таблиці були структуровані правильно.

1.3. Нормалізація бази даних та її форми

Можливість усунення непотрібної надмірності в реляційній базі даних піддаючи саму БД певному процесу називається процесом нормалізації. Останнє дозволяє декомпонувати реляційну схему на декілька нормалізованих таблиць, таких, що задовольняють певним нормальним формам.

Загалом, бази даних, які використовують для обробки транзакцій в режимі реального часу (скорочено OLTP, від англ. Online Transaction Processing), де користувачі відповідають за створення, читання, оновлення та видалення записів, повинні бути нормалізовані.

Бази даних аналітичної обробки в режимі реального часу (OLAP), які полегшують аналіз і звітність, можуть бути більш ефективними за умови певної денормалізації, оскільки їхньою переважною рисою є швидкість обчислень. До них відносяться програми підтримки прийняття рішень, де дані повинні бути швидко проаналізовані, але не повинні бути зміненими.

Кожна форма, або рівень нормалізації, включає правила, пов'язані з нижчими формами.

Перша нормальна форма (скорочено 1NF) визначає, що кожна комірка в таблиці може мати лише одне значення, а не їх список.

У користувача може виникнути спокуса обійти це, розбивши дані на додаткові стовпці, але це також суперечить правилам: таблиця з повторюваними або тісно пов'язаними групами атрибутів не задовольняє першій нормальній формі.

Замість цього необхідно розбивати дані на меншу кількість таблиць або записів до тих пір, поки кожна комірка не буде містити тільки одне значення і не залишиться зайвих стовпців. На цьому етапі дані вважаються атомарними, або розбитими на найменший корисний вимір.

Друга нормальна форма (2NF) вимагає, щоб кожен з атрибутів повністю залежав від первинного ключа. Це означає, що кожен атрибут повинен залежати

безпосередньо від первинного ключа, а не опосередковано через якийсь інший атрибут.

Наприклад, вважається, що атрибут "вік", який залежить від атрибуту "дата народження", який, у свою чергу, залежить від атрибуту "ID студента", має лише часткову функціональну залежність, і таблиця, що містить ці атрибути, не буде задовольняти другій нормальній формі.

Крім того, таблиця з первинним ключем, що складається з декількох полів, порушує другу нормальну форму, якщо одне з інших полів або декілька полів не залежать від кожної частини ключа.

Третя нормальна форма (3NF) додає до цих правил вимогу, щоб кожен неключовий стовпець був незалежним від будь-якого іншого стовпця. Якщо зміна значення в неключовому стовпці призводить до зміни іншого значення, така таблиця не задовольняє третій нормальній формі.

Були запропоновані додаткові форми нормалізації, включаючи нормальну форму Бойса-Кодда, четверту-шосту нормальні форми та нормальну форму доменного ключа, але перші три є найбільш поширеними та найбільш вживаними.

Хоча ці форми пояснюють найкращі практики, яких слід дотримуватися в цілому, ступінь нормалізації залежить від контексту бази даних.

Пізніше, розглянемо усі типи даних, зв’язки та форми нормалізації, які були застосовані в даній кваліфікаційній роботі, а поки перейдемо до наступного, основного етапу – розглянемо технології та мови програмування, які були використані під час реалізації проекту.

1.4. Реляційна база даних

Реляційна модель даних забезпечує стандартний спосіб представлення та запиту даних, який міг бути використаний будь-яким додатком. З самого початку розробники визнали, що сильна сторона реляційної моделі бази даних полягає у

використанні таблиць, тобто інтуїтивно зрозумілого, ефективного та гнучкого способу зберігання та доступу до структурованої інформації [5].

Реляційна модель передбачає, що логічні структури даних (таблиці даних, представлення та індекси) відокремлені від фізичних структур зберігання. Завдяки такому поділу адміністратори баз даних можуть керувати фізичним сховищем даних, не ставлячи під загрозу доступ до цих даних як до логічної структури. Наприклад, перейменування файлу бази даних не означає перейменування таблиць, що зберігаються в ньому.

Розмежування між логічними та фізичними зв’язками відноситься і до операцій над базами даних, які є чітко визначеними діями, що дозволяють додаткам маніпулювати даними та структурами баз даних. Логічні операції дозволяють додатку визначати необхідний йому вміст, тоді як фізичні операції визначають, як отримати доступ до цих даних, а потім виконати завдання.

Для того, щоб гарантувати, що дані завжди точні та доступні, реляційні бази даних дотримуються певних правил цілісності. Наприклад, правило цілісності може визначати, що в таблиці не допускаються дублікати рядків, щоб запобігти потраплянню в базу даних неправильної інформації.

З часом з'явилася ще одна сильна сторона реляційної моделі - розробники почали використовувати мову SQL (Structured Query Language) для запису даних в базу даних і виконання запитів до неї. Протягом багатьох років SQL широко використовується як мова запитів до баз даних. Заснована на реляційній алгебрі, SQL пропонує внутрішньо узгоджену математичну мову, яка дозволяє легко підвищити продуктивність всіх запитів до бази даних. Для порівняння, інші підходи передбачають визначення окремих запитів.

Програмне забезпечення, що використовується для запитів, зберігання, управління та отримання даних, що зберігаються в реляційній базі даних, називається системою управління реляційними базами даних (СУБД). СУБД забезпечує інтерфейс між користувачами і додатками та базою даних, а також

адміністративні функції для управління зберіганням, доступом і продуктивністю даних.

Існує кілька факторів, які слід враховувати при виборі між різними типами баз даних та продуктами реляційних баз даних. Тому, при виборі користувач повинен поставити низку необхідних запитань. Для прикладу: Які вимоги до точності даних? Чи зберігання та точність даних ґрунтується на бізнес-логіці? Чи потрібна масштабованість? Яким є масштаб даних, якими потрібно управляти, і яке очікується зростання? Наскільки важливий паралелізм? Чи буде потрібен одночасний доступ до даних декільком користувачам і додаткам? Чи підтримує програмне забезпечення бази даних паралельність при захисті даних? Які вимоги з точки зору продуктивності та надійності? Які вимоги до якості надання відповідей на запити? А також ряд інших важливих запитань.

РОЗДІЛ 2. ЗАСОБИ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

2.1 Середовище розробки Microsoft Visual Studio 2022

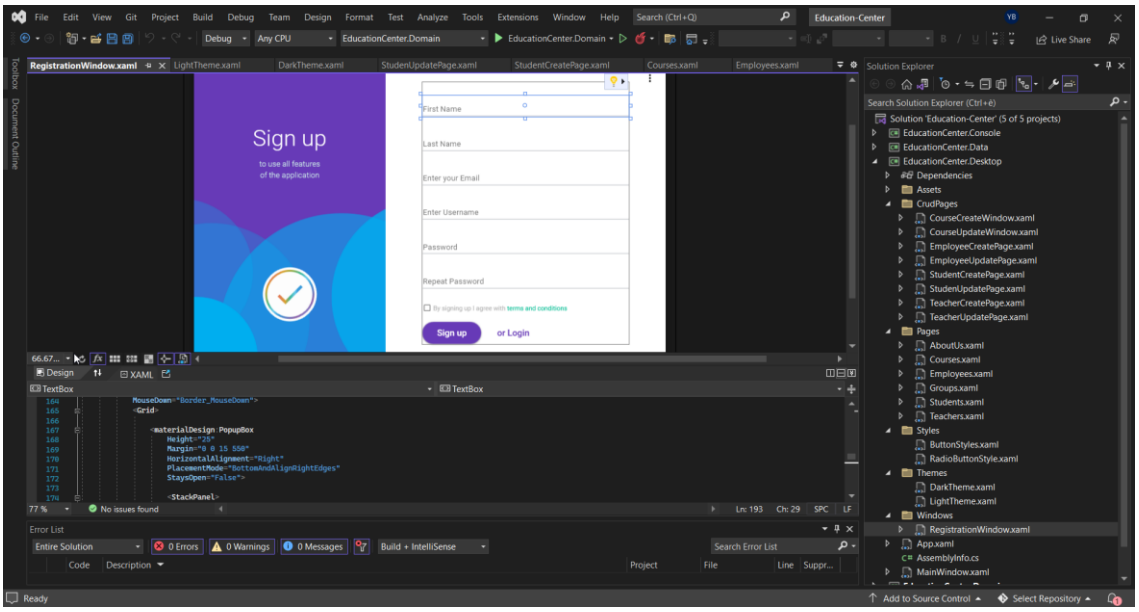


Рис. 8. Візуалізація середовища розробки кваліфікаційного проекту

Microsoft Visual Studio 2022 Professional - найкраща версія Visual Studio (Рис.8). Це перше 64-розрядне середовище розробки, яке з легкістю справляється з величезними за обсягом проектами та складними робочими навантаженнями. Нова система більш чуйна і плавна при виконанні повсякденних дій, таких як написання коду чи синхронізація даних між гілками (branches) проекту.

Середовище розробки Microsoft Visual Studio - серія програмних продуктів від Microsoft, спеціально розроблених як інтегроване середовище для розробки програмного забезпечення. З допомогою програмних компонентів можна розробляти консольні додатки, програми з графічним інтерфейсом, включаючи підтримку технології Windows Forms, Windows Presentation Foundation, веб-сайти, -програми та веб-служби для всіх платформ, які підтримує компанія. Visual Studio містить редактор вихідного коду з вбудованою технологією IntelliSense, яка значно спрощує процес написання коду.

IntelliSense допомагає підвищити продуктивність, забезпечуючи контекстно-залежні доповнення коду, скеровуючи розробників слідувати певним шаблонам і стилям команд, знаходити проблеми з кодом, які важко знайти програмісту та інше. IntelliCode працює як потужний набір інструментів для завершення коду. Інструменти можуть розпізнавати контекст коду: імена змінних, функції та тип згенерованого коду. Це дає змогу IntelliCode завершувати цілі рядки одночасно та допомагати писати код з більшою впевненістю і точністю.

CodeLens дає змогу легко знаходити необхідну та важливу інформацію, для прикладу, зміни які були внесені, що вони викликали ти чи потребують ваші методи модульного тестування. Основна інформація, як-от автори, посилання, тести та історія комітів, завжди доступна, щоб допомогти в ухваленні обґрунтованих робочих рішень.

Також, ще одним, досить важливий і необхідним інструментом даного середовища є можливість проведення сеансів спільної роботи в режимі реального часу (в Live Share), які прискорюють цикли налагодження та редагування для команд на будь-яких мовах і платформах. Персоналізовані сеанси з елементами управління доступом і налаштованими параметрами редактора забезпечують узгодженість коду між кількома розробниками.

В середовищі Visual Studio розробник має можливість використовувати сучасні інструменти, за допомогою яких можна:

- створювати кросплатформні мобільні та класичні додатки з допомогою .NET MAUI;
- створювати веб-інтерфейси на мові C# з використанням Blazor;
- збирати, налагоджувати та тестувати додатки .NET та C++ у середовищах Linux;
- використовувати можливості гарячого перезавантаження в додатках .NET і C++ та багато іншого.

2.2 .NET Framework

Для втілення архітектури певної системи використовують новітні бібліотеки, або фреймворки, які спрощують процес написання коду. Програмний фреймворк - це набір готових до використання програмних рішень, які містять логіку та базову функціональність, архітектуру проекту, або підсистеми та, навіть, дизайн. В середині фреймворка знаходиться набір бібліотек, які пропонують готовий комплекс рішень, також може містити програми, скрипти та в загальному все те, що може полегшити роботу розробника під час написання та поєднання різних елементів об’ємного програмного забезпечення (ПЗ). Основна перевага у використанні фреймворку - це стандартизованість архітектури застосунку.

.NET Framework - кросплатформенна технологія, запропонована корпорацією Microsoft як платформа для створення простих програм та веб-додатків [8].

Всі бібліотеки в .NET мають дані про свою версію. Це дає змогу обійти можливі конфліктні ситуації між версіями збірок, які відмінні між собою.

Технологію .NET поділяють на дві ключові частини - середовище виконання та інструментарій розробки.

Середовища розробки .NET-програм включають в себе використання таких застосунків, як: Visual Studio (яке було згадано вище), Borland Developer Studio, SharpDevelop, тощо. Програми також можна створювати в текстовому редакторі, а для виконання та дебагу використовувати консольний компілятор.

Основоположною ідеєю Microsoft .NET є сумісність різних служб, написаних на різних мовах. Для прикладу, певна служба, що написана на мові C++ для Microsoft .NET, може звернутися до будь-якого методу класу з бібліотеки, що була написана на Delphi. Або, на мові C# можна написати клас, успадкований від класу, що написаний на Visual Basic .NET, а вийняток, створений C# методом, може бути перехоплено і оброблено на Delphi.

Технологія .NET є технологією запатентованою корпорацією Microsoft. Проте, після підписання певних домовленостей з компанією Novell, технологія Mono була визнана реалізацією .NET на Unix-подібних системах. Mono дозволяє реалізовувати ASP.NET, ADO.NET та Windows Forms, але в той же момент рекомендує використовувати відмінні API від згаданих вище.

Фреймворк Microsoft .NET охоплює всі аспекти розробки програмного забезпечення, повністю захищаючи операційну систему, на якій воно працює. Фреймворк пропонує розробнику повністю кероване середовище для розробки додатків, піклуючись про кожен аспект, від виконання і управління ресурсами і пам'яттю до доступу до даних і управління інтерфейсом. Фреймворк .NET складається з трьох основних частин:

- компілятори для основних мов, що підтримуються Microsoft;
- середовище виконання Common Language Runtime;
- бібліотека класів.

Основною концепцією фреймворку є Common Language Runtime: це

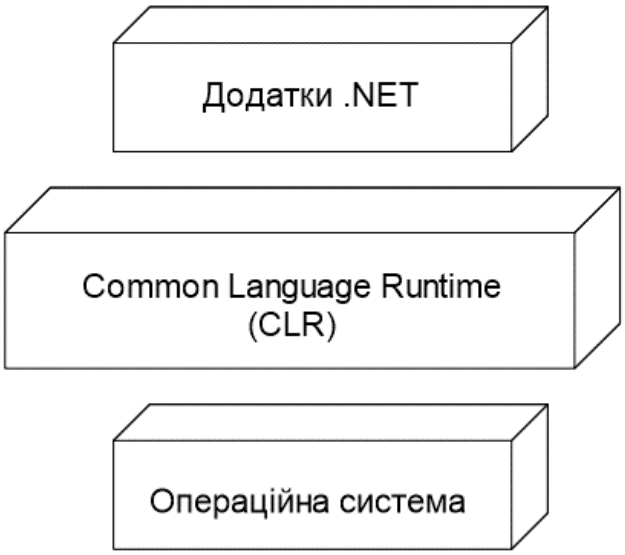
рівень фреймворку, який знаходиться поверх операційної системи і керує виконання .NET додатків. Як згадувалося раніше, програми, написані на мові .NET, не взаємодіють безпосередньо з операційною системою, але взаємодіють через CLR.

Фреймворк відповідає за фактичне виконання заявок, забезпечує дотримання всіх залежностей, управляє пам'яттю, безпекою, мовною інтеграцією тощо. Середовище виконання забезпечує численні сервіси, які спрощують

написання програмного коду, а також розповсюдження додатку та підвищення його надійності.

Common Language Runtime являє собою високопродуктивний виконавчий механізм. Код, на який посилається середовище виконання і виконання якого є керований ним, називається керованим кодом. Машинні коди небезпечні (оскільки обходять час виконання) можуть генеруватися компіляторами, і в даному випадку мова йде про некерований код.

Відповідальність за такі види діяльності, як розробка об'єктів, виконання викликів методів і т.д. також делегується середовищу виконання Common Language Runtime, що дозволяє надавати додаткові сервіси виконуваному коду. Керований код має доступ до середовища виконання CLR, за допомогою якого він може скористатися можливостями платформи (багатомовна інтеграція, управління винятками безпеки, управління версіями тощо).



Common Language Runtime складається з п'яти компонентів:

- CTS - Common Type System.

Середовище виконання використовує уніфіковану систему типів, здатну виражати семантику сучасних мов програмування. Ця система визначає стандартний набір типів даних і правил, необхідних для реалізації нових типів.

Середовище виконання здатне розуміти, як створювати та виконувати такі типи. Компілятори .NET Framework використовують сервіси середовища виконання для визначення типів даних, управління об'єктами та здійснення викликів методів замість того, щоб використовувати специфічні інструменти або мови. Використання єдиної системи шрифтів призводить до глибокої інтеграції між мовами. Код, написаний однією мовою, може успадковувати реалізацію від класів, написаних іншою мовою. Виключенням може бути викликаний код написаний однією мовою і який обробляється кодом, написаним іншою, а такі операції, як налагодження і профілювання, працюють прозоро незалежно від мови, якою написаний код. Це означає, що розробникам більше не потрібно створювати різні версії своїх бібліотек для кожної програми мовою програмування або компілятором, і що, наскільки це стосується класу, вони вже не обмежуються бібліотеками, розробленими для мови програмування, що використовується.

- CLS - загальна мовна специфікація:

CLS - це набір правил, які застосовуються для формування асамблеї. Якщо компонент, написаний мовою (наприклад C#) має використовуватися іншою мовою (наприклад VB .NET), то автор компонента повинен дотримуватись типів та структури, визначені CLS.

Структури CLS повинні відповідати наступним вимогам набір правил, в тому числі:

- уникати використання імен, які зазвичай використовуються як ключові слова на мовах програмування;
- забороняти користувачу контролювати вкладені типи;
- приймати до уваги, що реалізації методів з однаковими іменами та сигнатурами в різних інтерфейсах вважаються незалежними.

- CIL - Common Intermediate Language.

Реалізація в середовищі .NET Framework спільної мови-посередника називається Microsoft Intermediate Language (MSIL). Всі укладачі, які

дотримуються структури CLR повинні генерувати процесорно-незалежне представлення проміжного коду, під назвою "спільна мова-посередник". У середовищі виконується "на льоту" за допомогою компіляції Just In Time. CIL стоїть на позиціях на набагато вищому рівні, ніж більшість машинних мов, маючи інструкції по завантаженню, зберіганню та ініціалізації даних, для виклику методів з об'єктів та багато інших інструкцій для арифметичних і логічних операцій, управління потоками та прямий доступ до пам'яті. CIL також має інструкції щодо підняття та перехоплення винятків для обробка помилок, подібна до інструкцій try/catch в C#. Цей проміжний формат має як споріднені, так і основні риси відмінності від традиційної мови Асамблеї, такі які можуть оперувати даними за допомогою інструкцій типу "push" та "pop" та перенести їх до реєстрів; але, на відміну від Асамблеї, вона не посилається до будь-якої конкретної апаратної платформи.

- JIT - компілятор Just In Time.

Перш ніж проміжна мова може бути виконана, вона повинна бути перетворена компілятором Just In Time .NET Framework в нативний код, який залежить від процесора і працює на тій же архітектурі, на якій працює сам JIT-компілятор. Через обмеження дизайну деяких мов програмування, таких як C, компілятори цих мов можуть бути не в змозі створити безпечний і перевірений код, тому цей код може виконуватися тільки в безпечних областях. Існує два типи компіляторів JIT: звичайний компілятор і скорочена економічна версія. Звичайний компілятор досліджує IL методу і перетворює його в оптимізований для платформи нативний код так само, як це робить традиційний компілятор C/C++. Економний компілятор, з іншого боку, призначений для використання на тих машинах, де вартість використання пам'яті та тактів процесора є високою (наприклад, вбудовані системи на базі Windows CE).

- VES - Virtual Execution System.

Віртуальна система виконання, яка являє собою еквівалент віртуальної машини для середовища Sun/Oracle. VES завантажує, встановлює з'єднання та

виконує програми, написані для Common Language Runtime. ESR виконує свої функції завантажувача, використовуючи інформацію, що міститься в метаданих, і використовує пізнє зв'язування для інтеграції окремо скомпільованих модулів, які також можуть бути написані на різних мовах.

Тому, як вже згадувалося, потрібно ще раз зауважити, що при компіляції програми, написаної на будь-якій мові .NET, вихідний код не транслюється безпосередньо в двійковий виконуваний код, а в проміжний код, який називається MSIL (Microsoft Intermediate Language), який потім інтерпретується CLR. Ця проміжна мова не залежить від апаратного забезпечення та операційної системи, а лише тільки під час виконання CLR піклується про переклад MSIL у двійковий виконуваний код.

2.3 Мова програмування C#

C# - це об'єктно-орієнтована мова програмування (мова ООП), яка розроблена та підтримувана компанією Microsoft, подібна до мов C++ та Java. Це одна з найпотужніших, на даний час, мов для маніпулювання об'єктами у фреймворку .NET.

Можна вважати, що C# є висококласною мовою програмування на .NET Framework. На відміну від інших мов, таких як Visual Basic або C++, вона була створена спеціально для нової платформи. Тому не дивно, що сама корпорація Microsoft використовує мову C# для написання значної частини бібліотек .NET.

C# та Visual Basic .NET є двома з багатьох мов програмування, які є доступними для платформи .NET. В загальному, Common Language Runtime (CLR) платформи .NET Framework є спільною платформою виконання для знайної кількості мов програмування, включаючи C++, F#, Cobol .NET .

Мова C# розроблялась як мова прикладного рівня CLR і з цієї причини вона залежить, переважною мірою, безпосередньо від CLR. Наявність чи відсутність певних функціональностей мови програмування обумовлено

можливістю особливості мови бути трансльованою у відповідні конструкції CLR. CLR надає C#, як і іншим орієнтованим на .NET мовам, багато функціональних можливостей, яких не мають класичні мови програмування. Для прикладу, збірку сміття в самому C# не було реалізовано, але, незважаючи на це, вона працює завдяки CLR для усіх програм, які розроблені на мові C# [1].

Потрібно також зазначити, що C# є новою мовою, в той час як VB .NET несе в собі спадщину усіх попередніх версій Visual Basic (де синтаксис VB .NET, хоча і оновлений для можливостей підтримки об'єктів, практично такий же, як синтаксис Visual Basic 4.0, випущений в далекому 1996 році).

2.3.1 Реалізація шаблону "узагальненого та асинхронного сховища"

Управління доступом до даних безпосередньо в рамках бізнес-логіки не є вдалою ідеєю, оскільки пряма реалізація різних методів CRUD (Create, Read, Update, Delete) неминуче створює сильний зв'язок між двома різними за своєю природою функціональними можливостями. Такий підхід спричиняє ряд недоліків, серед яких:

1. ускладнена тестованість бізнес-логіки;
2. сильна залежність бізнес-логіки від зовнішніх компонентів, таких як база даних;
3. дублювання коду доступу до даних всередині кодової бази;
4. складна реалізація механізмів багаторівневого кешування;

Спочатку потрібно визначити загальний шаблон [14], який згодом будуть використовувати всі наші конкретні типи, тому ми створюємо новий клас BaseAudit наступним чином:

```
using StudentManagment.Domain.Enums;
using System;

namespace StudentManagment.Domain.Commons
{
```

```
public class BaseAudit
{
   public long Id { get; set; }
   public DateTime CreatedAt { get; set; }
   public DateTime UpdatedAt { get; set; }
   public long? UpdatedBy { get; set; }
   public ItemState State { get; set; }
}}
```

Так як в програмі використовується EF Code First, властивість ID буде представляти ключове поле ідентичності студента.

Далі необхідно визначити через інтерфейс різні методи, які будуть реалізовані об'єктом репозиторію. Тому було створено загальний інтерфейс під назвою IGenericRepository наступним чином:

```
using System.Linq.Expressions;
using System.Threading.Tasks;

namespace TestStudentManagmentUoW.Data.IRepositories
{
   public interface IGenericRepository<T> where T : class
   {
       Task<T> CreateAsync(T entity);
       Task<T> UpdateAsync(T entity);
       Task<T> GetAsync(Expression<Func<T, bool>> expression);
       Task<IQueryable<T>> GetAllAsync(Expression<Func<T, bool>>
expression = null);
       Task<bool> DeleteAsync(Expression<Func<T, bool>> expression);
      Task SaveAsync();
   }
}
```

В класі вище використовуємо концепцію узагальнень, які по суті є "моделлю коду", що дозволяє мені, як розробнику, визначати незалежні від типу структури даних.

Налаштувавши різні CRUD-методи з типом повернення Task або Task<T> (тут, для визначення узагальненого типу використовується параметр

узагальненого типу T) таким чином, щоб можна було легко використовувати асинхронні версії методів запитів, що пропонуються EF.

Тепер створимо новий клас, який в ідеалі буде представляти сутність в базі даних, щоб зробити його "consumable" через клас репозиторію, який незабаром буде створено.

Шаблон BaseAudit було використано як базовий клас для класів Студент, Викладач та інших. Для прикладу проілюструємо клас Студент наступним чином:

```
using StudentManagment.Domain.Commons;
using System.ComponentModel.DataAnnotations;

namespace StudentManagment.Domain.Entities.Students
{
   public class Student : BaseAudit
   {
      [Key]
      public string? ID { get; set; }
      public string FirstName { get; set; }
      public string LastName { get; set; }
      public string Phone { get; set; }
      public string Email { get; set; }
      public string Password { get; set; }
      public string Group { get; set; }
      public string Faculty { get; set; }
      public int Year { get; set; }
      public string? CourseId { get; set; }
   }
}
```

, який надалі використовується як базовий інтерфейс для наступного репозиторію

```
using StudentManagment.Domain.Entities.Students;

namespace TestStudentManagmentUoW.Data.IRepositories{
```

```
public interface IStudentRepository : GenericRepository<Student>
        {

        }
}
```

Далі, необхідно створити інтерфейс сервісу методи якого ми будемо використовувати для менеджменту даними.

Але, спершу, необхідно визначити, що являє собою поняття інтерфейсу. Отож, інтерфейс має класоподібну структуру і містить визначення абстрактних, тобто таких, що не потребують реалізації, взаємопов'язаних методів, які можуть бути реалізовані класом. Ми можемо думати про інтерфейси як про контракт; класи, які реалізують один або декілька інтерфейсів, повинні визначати методи, оголошені в інтерфейсах.

Інтерфейс є аналогом класу, але на відміну від класу, він являє собою чисту специфікацію [13]. Використовуючи інтерфейси, можна, наприклад, включити в клас поведінку з декількох джерел. Ця функціональність є важливою в C#, оскільки мова не підтримує множинну спадковість класів.

Нижче представлено інтерфейс сервісу для моделі Студент:

```
using StudentManagment.Domain.Entities.Students;
using System;
using System.Collections.Generic;
using System.Linq.Expressions;
using System.Threading.Tasks;
using TestStudentManagmentUoW.Service.DTOs.Students;

namespace TestStudentManagmentUoW.Service.Interfaces
{
   public interface IStudentService
   {
        Task<Student> CreateAsync(StudentForCreationDto studentDto);
        Task<Student> GetAsync(Expression<Func<Student, bool>>
expression);
        Task<IEnumerable<Student>> GetAllAsync(Expression<Func<Student,
bool>> expression = null);
```

```
            Task<bool> DeleteAsync(Expression<Func<Student, bool>>
expression);
            Task<Student> UpdateAsync(long id, StudentForCreationDto
studentDto);
        }
    }
```

Тепер, коли визначено загальну модель, що представляє наші дані, ми переходимо до визначення класу, який буде запитувати нашу базу даних, повертаючи дані, які наша бізнес-логіка буде певним чином обробляти:

```
using Mapster;
using StudentManagment.Domain.Entities.Students;
using System;
using System.Collections.Generic;
using System.Linq.Expressions;
using System.Threading.Tasks;
using TestStudentManagmentUoW.Data.IRepositories;
using TestStudentManagmentUoW.Data.Repositories;
using TestStudentManagmentUoW.Service.DTOs.Students;
using TestStudentManagmentUoW.Service.Interfaces;
namespace TestStudentManagmentUoW.Service.Services
{
    public class StudentService : IStudentService
    {
        private readonly IStudentRepository studentRepository;

        public StudentService()
        {
            this.studentRepository = new StudentRepository();
        }

        public async Task<Student> CreateAsync(StudentForCreationDto
studentDto)
        {
            var result = await
studentRepository.CreateAsync(studentDto.Adapt<Student>());
            await studentRepository.SaveAsync();
            return result;
        }
```

```
            public async Task<bool> DeleteAsync(Expression<Func<Student,
bool>> expression)
            {
                var isDeleted = await studentRepository.DeleteAsync(expression);
                await studentRepository.SaveAsync();
                return isDeleted;
            }

            public async Task<IEnumerable<Student>>
GetAllAsync(Expression<Func<Student, bool>> expression = null)
                => await studentRepository.GetAllAsync(expression);

            public async Task<Student> GetAsync(Expression<Func<Student,
bool>> expression)
            {
                var student = await studentRepository.GetAsync(expression);
                return student;
            }

            public async Task<Student> UpdateAsync(long id,
StudentForCreationDto studentDto)
            {
                var student = await studentRepository.GetAsync(p => p.Id == id);

                student = await
studentRepository.UpdateAsync(studentDto.Adapt(student));

                await studentRepository.SaveAsync();
                return student;
            }
        }
    }
```

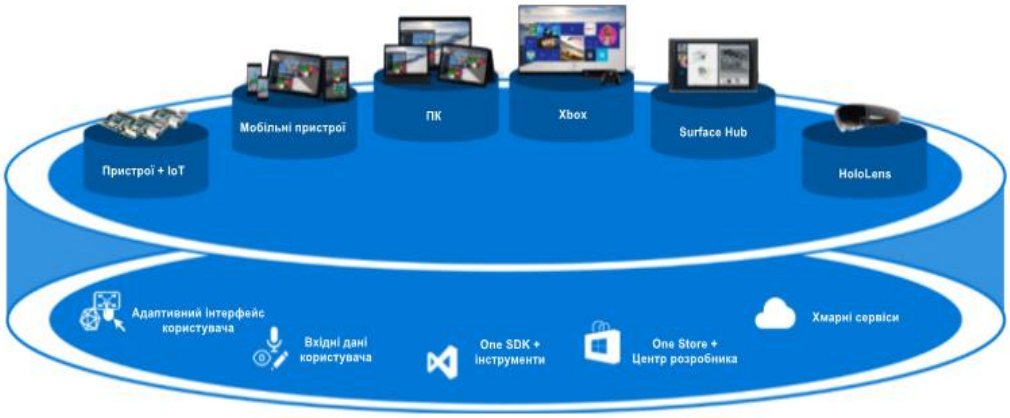
Враховуючи простоту такого підходу та повну незалежність класу від бізнес-логіки, подальша реалізація кешу в пам'яті тепер є простою.

2.4 Графічна підсистема Windows Presentation Foundation (WPF)

Універсальна платформа Windows (UWP) - це однорідна платформа додатків, створена корпорацією Майкрософт і вперше представлена в Windows

[17]. Мета цієї програмної платформи - допомогти розробляти універсальні застосунки, які працюють на Windows , Windows Mobile, Xbox One та Hololens без необхідності переписування під кожну з них. Підтримує розробку Windows-додатків на мовах C++, C#, VB.NET та XAML. API реалізовано на мові C++ та підтримується мовами C++, VB.NET, C#, F# та JavaScript. Дану платформу було розроблено як розширення платформи Windows Runtime, вперше представленої в Windows Server 2012 і Windows 8.

UWP дозволяє розробникам створювати застосунки, які потенційно здатні працювати на різних видах пристроїв. Додатки UWP не запускаються в попередні версії Windows. Застосунки, які здатні реалізувати цю платформу, розробляються нативно за допомогою Visual Studio. Старіші програми для Windows, Windows Phone або інших потребують певних змін.



Windows Presentation Foundation (WPF) - бібліотека класів платформи .NET Framework [6]. Фреймворк представлений для розробки графічного інтерфейсу користувача додатків в середовищах Windows. WPF базується на системі векторної графіки, яка спирається на DirectX для використання переваг апаратного прискорення сучасних відеокарт.

WPF також може використовуватися для реалізації додатків, які можуть виконуватися в браузері Microsoft Internet Explorer або інших сучасних просунутих браузерах, доки існує Framework.

Мова, що використовується для створення користувальницького інтерфейсу в WPF - XAML (eXtensible Application Markup Language), що дозволяє визначати об'єкти в робочій області за допомогою XML-тегів, таким чином можна відокремити логічну частину від графічної. Оскільки UWP-додатки працюють тільки в Windows 10 і вище, необхідно розглянути можливість розробки WPF-додатку для більш ранніх версій Windows.

Слід також зазначити, що Windows як і раніше підтримує традиційні додатки, засновані на фреймворку .NET і таких технологіях, як Windows Forms та WPF (Рис.9).

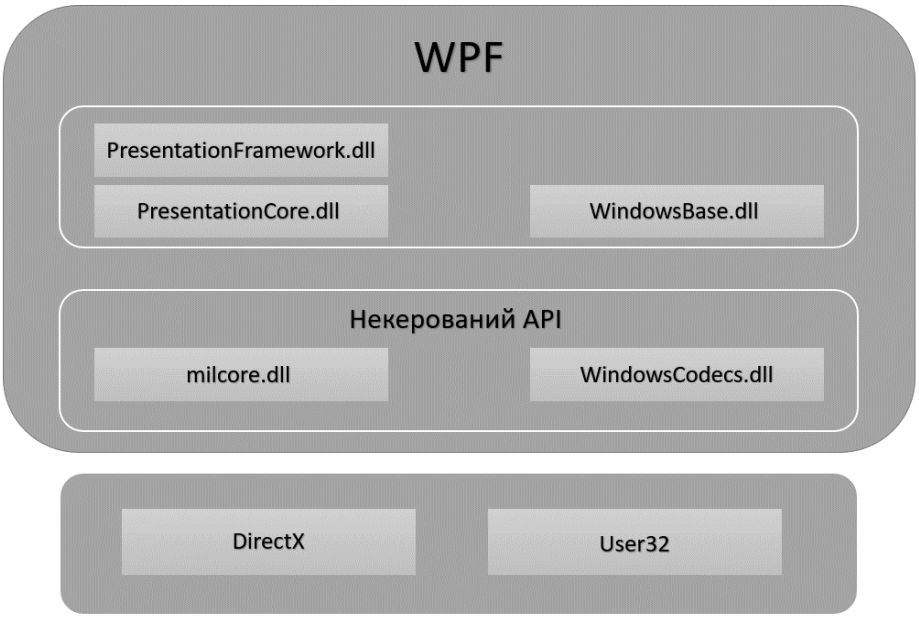


Рис. 9. Схематичне відображення архітектури WPF

WPF надає необхідні візуальні засоби для побудови інтерфейсів в середовищі Windows [12]. Елементи можна перетягувати з панелі інструментів (оновленої відповідно до обраного типу дизайну) і розташовувати за бажанням за допомогою системи автоматичного позиціонування і "стикування" елементів.

Під час написання додатку на WPF відчувалося значне покращення останньої версії бібліотеки, саме у виконанні анімації [7], прив'язці даних та багаторівневих вікон.

Дизайнер додатків WPF представляє кілька цікавих можливостей для роботи над інтерфейсами:

- подвійне відображення розмітки XAML і багатий візуальний редактор, в якому зміни синхронізуються по всьому вікну;
- підтримка графічного відображення користувацьких елементів управління;
- додавання повзунка для збільшення та зменшення масштабу на графічному інтерфейсі вашого додатку;
- можливість зупинити завантаження графічного відображення великих .xaml файлів;
- підтримка розробки XAML коду через intellisense;
- підтримка кастомізації через панель інструментів управління та вікно властивостей;
- наявність опорних ліній для позиціонування органів управління всередині вікна;
- навігатор тегів - інструмент, який відображає позицію відповідного тегу XAML в дереві розмітки, коли певний елемент керування вибрано в поданні проекту.

Крім того, конструктор WPF пропонує підтримку співпраці з Expression Blend: завдяки спільному формату роботи (XAML) і спільному використанню відповідного рішення можна розділити роботу програмістів і дизайнерів.

РОЗДІЛ 3. МОВА КОРИСТУВАЛЬНИЦЬКОГО ІНТЕРФЕЙСУ XAML

Як значилося вище, XAML - це мова, заснована на XML, яка слідує або розширює структурні правила XML. Деяка термінологія є спільною або базується на термінології, що зазвичай використовується для опису мови XML або об'єктної моделі XML-документа.

XAML сама по собі не є однією з поширених мов, що використовуються безпосередньо середовищем виконання CLR. Замість цього, можна думати про XAML як про підтримку власної системи типів. Спеціальна система аналізу XAML, що використовується WPF, заснована на CLR і системі типів CLR [9]. Типи XAML зіставляються з типами CLR для створення екземпляра представлення під час виконання, коли XAML для WPF навпаки.

Залежно від рівня специфікації мови XAML, можна зіставити типи XAML з будь-якою іншою системою типів [10], яка не обов'язково повинна бути CLR, але яка вимагатиме створення та використання іншого синтаксичного аналізатора XAML.

При використанні для забезпечення значення атрибуту синтаксисом, що відрізняє порядок розширень розмітки для XAML-процесора, є присутність відкриваючої та закриваючої фігурних дужок ({ та }). Тип розширення розмітки далі ідентифікується за допомогою маркеру рядка, що розташований безпосередньо за відкриваючою фігурною дужкою.

При використанні в синтаксисі елемента властивості розширення розмітки візуально таке ж, як і будь-який інший елемент, що використовується для надання значення елемента властивості: оголошення елемента XAML, яке посилається на клас розширення розмітки як на елемент, укладений в кутові дужки (< >).

Існує кілька розширень розмітки [16], які не є специфічними для реалізації XAML у WPF, а є реалізаціями внутрішніх або функціональних можливостей XAML як мови. Ці розширення розмітки реалізовані в збірці System.Xaml як

частина загальних служб XAML платформи .NET Framework і включені в простір імен мови XAML.

З точки зору загального використання розмітки, ці розширення розмітки, як правило, ідентифікуються за допомогою префікса x: в синтаксисі. Базовий клас MarkupExtension (також визначений в System.XAML) забезпечує модель, яку повинні використовувати всі розширення розмітки для підтримки в XAML-читачах і XAML-записувачах, в тому числі в XAML WPF. Для прикладу:

- **x:Type** надає об'єкт Type для іменованого типу. Цей функціонал найчастіше використовується в стилях і шаблонах.
- **x:Static** створює статичні значення. Значення надходять від сутностей коду типу значення, які не є безпосередньо типом значення цільової властивості, але можуть повертати цей тип.
- **x:Null** визначає нуль як значення властивості і може використовуватися для атрибутів або значень елементів властивості.
- **x:Array** підтримує створення загальних масивів у синтаксисі XAML у випадках, коли ви навмисно вирішили не використовувати підтримку колекцій, що надається базовими елементами та шаблонами елементів керування WPF.

Тут варто зазначити, що префікс x: використовується для відображення простору імен XAML, характерного для мови XAML, в кореневому елементі XAML-файлу або продукції. Наприклад, шаблони Visual Studio для WPF-застосунків запускають XAML-файл, використовуючи x: відображення.

У користувацькому відображенні простору імен XAML можна вибрати інший префіксний токен. Однак, для ідентифікації об'єктів, які представляють певну частину простору імен XAML для мови XAML, використовується відображення x: замість простору імен за замовчуванням WPF або інших просторів імен XAML, не пов'язаних з конкретним фреймворком (Рис.10):

```
DataTrigger                                                    Binding
1   <Window
2       x:Class="StudentManagment.Desktop.MainWindow"
3       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5       xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
6       xmlns:local="clr-namespace:StudentManagment.Desktop"
7       xmlns:materialDesign="http://materialdesigninxaml.net/winfx/xaml/themes"
8       xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
9       x:Name="home"
10      Title="MainWindow"
11      Width="1100"
12      Height="720"
13      AllowsTransparency="True"
14      Background="Transparent"
15      FontFamily="Poppins"
16      ResizeMode="NoResize"
17      WindowStartupLocation="CenterScreen"
18      WindowState="Normal"
19      WindowStyle="None">
20      <Border Background="{DynamicResource PrimaryBackgroundColor}" CornerRadius="10">
21          <Grid>
22              <Grid.ColumnDefinitions>
23                  <ColumnDefinition Width="250" />
24                  <ColumnDefinition Width="*" />
25              </Grid.ColumnDefinitions>
26
27              <!--  Start: GridColumn Left  -->
28              <Border
29                  Padding="10"
30                  Background="{DynamicResource SecundaryBackgroundColor}"
31                  CornerRadius="10 0 0 10">
32                  <Grid x:Name="gridMenu">
70 %    No issues found
```

Рис. 10. Приклад префіксного токену в програмному коді

У розширюваній мові розмітки додатків (Extensible Application Markup Language, XAML) властивості стилю і шаблону можуть бути технічно встановлені одним з двох способів. Синтаксис атрибутів може використовуватися для посилання на стиль, визначений в межах ресурсу, наприклад, <Style="{StaticResource}"…/>. Крім того, для визначення вбудованого стилю можна використовувати синтаксис елемента властивості (Рис.11), наприклад

```
<Button
    x:Name="btnRestore"
    Margin="5,0"
    Click="btnRestore_Click">
    <Button.Style>
        <Style BasedOn="{StaticResource IconButtonsStyle}" TargetType="{x:Type Button}">
            <Setter Property="Content" Value="{DynamicResource maximize}" />
            <Style.Triggers>
                <DataTrigger Binding="{Binding ElementName=home, Path=WindowState}" Value="Maximized">
                    <Setter Property="Content" Value="{DynamicResource restore}" />
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </Button.Style>

</Button>
```

Рис. 11. Подання синтаксису стилю відображення кнопки

Використання атрибутів є набагато більш поширеним. Стиль, визначений вбудовано і не визначений в ресурсах, обов'язково знаходиться в межах області дії елемента, якого він містить, і не може бути легко використаний повторно, оскільки він не має ключа ресурсу. Стиль, визначений ресурсами, є більш універсальним та корисним і більше відповідає загальному принципу програмування WPF [3], який полягає у відокремленні логіки програми в коді від дизайну в розмітці.

Загалом, немає причин встановлювати вбудований стиль або шаблон, навіть якщо ви маєте намір використовувати цей стиль або шаблон лише в цій позиції. Більшість елементів, які можуть приймати стиль або шаблон, також підтримують властивість вмісту та шаблон вмісту. Якщо ви використовуєте будь-яке логічне дерево, створене за допомогою стилів або шаблонів, лише один раз, також буде простіше заповнити властивість Content еквівалентними дочірніми елементами в прямій розмітці, таким чином, повністю ігноруються стилі та механізми шаблонування.

Залежно від рівня знань основних концепцій XAML, можна використовувати поведінку перетворення типів в XAML-коді базової програми, не усвідомлюючи цього.

Наприклад, WPF визначає буквально сотні властивостей, які приймають значення типу Point. Точкове значення описує координату в двовимірному координатному просторі і фактично має дві важливі властивості: X і Y. При вказівці точки в XAML вказуйте її у вигляді рядка з роздільником (зазвичай комою) між зазначеним значенням X і значенням Y. Наприклад: <LinearGradientBrush StartPoint="0,1" EndPoint="0,0"/> (Рис.12)

```
<materialDesign:PackIcon.Foreground>
    <LinearGradientBrush StartPoint="0,1">
        <GradientStop Offset="0.1" Color="#E27C53" />
        <GradientStop Offset="0.3" Color="#DCA530" />
        <GradientStop Offset="0.5" Color="#3BB799" />
        <GradientStop Offset="0.7" Color="#67CBEE" />
        <GradientStop Offset="0.8" Color="#3699DB" />
    </LinearGradientBrush>
</materialDesign:PackIcon.Foreground>
```

Рис. 12. Подання синтаксису стилю відображення градієнта

Цей простий тип і його просте використання в XAML також включає в себе конвертер типу Point. У даному випадку це клас PointConverter.

Взагалі, в WPF та .NET Framework реалізації парсеру XAML існують певні типи, які передбачають власну обробку перетворення типів, навіть якщо ці типи умовно не вважаються примітивами. Прикладом типів, про які йде мова, є DateTime. Причина цього полягає в тому, як працює архітектура .NET Framework: тип DateTime визначається в mscorlib, найпростішій бібліотеці в .NET. DateTime не допускається з атрибутом з іншої збірки, який вводить залежність (атрибут TypeConverterAttribute з системи), так що звичайний механізм визначення конвертера типів не може бути підтриманий. Натомість парсер XAML має перелік типів, які потребують такої нативної обробки і обробляє ці типи аналогічно до обробки фактичних примітивів.

У прикладі, наведеному вище, згадувався клас Point PointConverter. Для .NET реалізацій XAML всі перетворювачі типів, що використовуються для цілей XAML, є класами, похідними від класу TypeConverterbase. Клас TypeConverter існує у версіях .NET Framework, які передують існуванню XAML. Одне з його початкових застосувань полягало в забезпеченні перетворення рядків для діалогових вікон властивостей у вікнах візуального проектування. Для XAML роль розширюється до базового класу для перетворення рядка в рядок для розбору рядкового значення атрибуту TypeConverter та обробки значення властивості об'єкта під час виконання в рядок для серіалізації як атрибуту.

РОЗДІЛ 4. СТВОРЕННЯ ЕЛЕМЕНТІВ WPF ТА КРОКИ ВИКОНАННЯ ЗАСТОСУНКУ

4.1 Клас Window (WPF .NET)

У WPF вікно визначається класом Window, який використовують для виконання наступних операцій з вікнами:

- відображення
- налаштування розміру, положення та зовнішнього вигляду
- розміщення контенту, специфічного для конкретного додатку.
- керування тривалістю («життям») вікна.

В загальному, вікно розділене на дві зони: неклієнтську та клієнтську. Неклієнтська область вікна втілена засобами WPF і містить в собі частини вікна, спільні для більшості вікон, в тому числі наступні: рядок заголовка, значок, назву, кнопки "Зменшити", "Збільшити" та "Закрити" та інше.

Клієнтська область вікна - це область в межах неклієнтської області, яка використовується розробниками для додавання специфічного для програми вмісту, наприклад, рядків меню, панелей інструментів та елементів керування. Переважно це клієнтська область, вертикальна лінійка зміни розміру та елементи управління, доданий в клієнтську частину розробниками.

У WPF є можливість реалізувати зовнішній вигляд і поведінку вікна за допомогою коду або XAML-розмітки. Однак, як правило, зовнішній вигляд вікна програми реалізується за допомогою XAML-розмітки, тоді як поведінка реалізується за допомогою code-behind, як показано в наступному прикладі (часткове зображення коду вікна “Student”):

```
<Page
   x:Class="StudentManagment.Desktop.Pages.Students"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

```
xmlns:local="clr-namespace:StudentManagment.Desktop.Pages"
xmlns:materialDesign="http://materialdesigninxaml.net/winfx/xaml/themes"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
Title="Students"
d:DesignHeight="450"
d:DesignWidth="800"
FontFamily="Poppins"
Loaded="Page_Loaded"
mc:Ignorable="d">

<Border CornerRadius="8">
   <Grid>
      <DataGrid
         x:Name="dtGrid"
         AutoGenerateColumns="False"
         IsReadOnly="True"
         SelectionChanged="dtGrid_SelectionChanged">
         <DataGrid.Columns>

          <DataGridTemplateColumn Header="Actions">
             <DataGridTemplateColumn.CellTemplate>
                <DataTemplate>
                   <DockPanel>
                      <Button Click="DeleteBtn" Style="{StaticResource
deleteButton}" />
                      <Button x:Name="UpdateStudentBtn" Click="UpdateBtn"
Style="{StaticResource editButton}" />
                   </DockPanel>
                </DataTemplate>
             </DataGridTemplateColumn.CellTemplate>
          </DataGridTemplateColumn>
         </DataGrid.Columns>
      </DataGrid>
      <Button
         x:Name="CreateStudentBtn"
         Width="80"
         Height="32"
         Margin="3"
         HorizontalAlignment="Right"
         VerticalAlignment="Top"
         Click="CreateCourseBtn_Click">
         <ContentControl Content="Create" />
      </Button>
      <Frame x:Name="MainFrame" Margin="40" />
```

```
        </Grid>
    </Border>
</Page>
```

Наступний код є вихідним кодом для XAML-розмітки (для прикладу наведу лише невелику частину класу з причин, що розмір класу містить майже 200 рядків програмного коду):

```
using StudentManagment.Desktop.CrudPages;
// та інші 8 бібліотек

namespace StudentManagment.Desktop.Pages
{
    public partial class Students : Page
    {
        StudenUpdatePage studenUpdatePage;
        StudentCreatePage studentCreatePage;
        IStudentService studentService;
        public Students()
        {
            studenUpdatePage = new StudenUpdatePage();
            studentCreatePage = new StudentCreatePage();
            InitializeComponent();
            studentService = new StudentService();
        }

        public async void Page_Loaded(object sender, RoutedEventArgs e)
        {
            dtGrid.ItemsSource = (await studentService.GetAllAsync()).ToList();
        }

        private async void CreateCourseButton_Click(object sender, RoutedEventArgs
e)
        {
            if (string.IsNullOrEmpty(studentCreatePage.StudentName.Text) ||
string.IsNullOrEmpty(studentCreatePage.StudentLast.Text) ||
string.IsNullOrEmpty(studentCreatePage.StudentPhone.Text) ||
string.IsNullOrEmpty(studentCreatePage.CourseId.Text))
            {
                return;
            }
            var course = new StudentForCreationDto
```

```
            { //  };

            await studentService.CreateAsync(course);

            MessageBox.Show("Successfully created!");

            dtGrid.Visibility = Visibility.Visible;
            CreateStudentBtn.Visibility = Visibility.Visible;
            studentCreatePage.Visibility = Visibility.Collapsed;
            Page_Loaded(sender, e);

        }
    }
}
```

4.2 Створення елементів WPF

При розробці елемента управління WPF, як правило, практичніше вставляти XAML-файл і DLL, на який посилаються, окремо. Це полегшує обмін та налагодження DLL. Також можна обмінюватися DLL, що належать до WPF-елементу також під час роботи Редактора. Посилання здійснюється за допомогою посилання у файлі XAML.

Щоб створити WPF-елемент необхідно:

1. На панелі інструментів вибрати символ WPF-елемента або вибрати відповідний елемент в меню Елементи;

2. У головному вікні обрати точку відліку;

3. Намалювати елемент за допомогою миші;

4. У групі Darstellung властивостей елемента вибрати властивість XAML-Datei, відкриється діалог вибору файлу, після чого необхідно вибрати потрібний файл.

Допустимі файли у форматі

*.xaml: розширювана мова розмітки додатків та *.cdwpf: колективний файл WPF, також показує попередній перегляд зображення.

Файл повинен вже існувати у вузлі Файл/Графіка проекту, або повинен бути створеного у діалоговому вікні. В моєму випадку файл було створено вручну та додано схему, імплементовано корпорацією Microsoft. Проілюструю частину файлу Icons.xaml, додану для коректної мінімізації, максимізації, перезавантаження та здатності закривати піктограми (значки):

```
<PathGeometry x:Key="minimize" Figures="M19 13H5a1 1 0 0 1 0-2h14a1 1 0 0 1
0 2z" />
<PathGeometry x:Key="maximize" Figures="M18 21H6a3 3 0 0 1-3-3V6a3 3 0 0 1
3-3h12a3 3 0 0 1 3 3v12a3 3 0 0 1-3 3zM6 5a1 1 0 0 0-1 1v12a1 1 0 0 0 1 1h12a1 1 0
0 0 1-1V6a1 1 0 0 0-1-1z" />
<PathGeometry x:Key="restore" Figures="M18 3H6a3 3 0 0 0-3 3v12a3 3 0 0 0 3
3h12a3 3 0 0 0 3-3V6a3 3 0 0 0-3-3zm1 15a1 1 0 0 1-1 1H6a1 1 0 0 1-1-1V6a1 1 0 0
1 1-1h12a1 1 0 0 1 1 1z M15 11H9a1 1 0 0 0 0 2h6a1 1 0 0 0 0-2z" />
<PathGeometry x:Key="close" Figures="M13.41 12l4.3-4.29a1 1 0 1 0-1.42-1.42L12
10.59l-4.29-4.3a1 1 0 0 0-1.42 1.42l4.3 4.29-4.3 4.29a1 1 0 0 0 0 1.42 1 1 0 0 0 1.42
0l4.29-4.3 4.29 4.3a1 1 0 0 0 1.42 0 1 1 0 0 0 0-1.42z" />
```

4.3 Створення дизайну кнопки за допомогою XAML

Представлення кнопки складається з різних об'єктів, в тому числі прямокутників та інших елементів для надання кнопці унікального зовнішнього вигляду [11].

До певного моменту, керування зовнішнім виглядом кнопок обмежувалося зміною властивостей кнопок. Що буде, якщо у вас виникне бажання реалізувати більш радикальні зміни у зовнішньому вигляді кнопки? Шаблони дозволяють контролювати представлення об'єкта. Оскільки шаблони можуть використовуватися в межах стилів, є можливість застосувати шаблон до всіх об'єктів, до яких застосовується стиль (в даному випадку - кнопка).

Оскільки такі елементи управління, як кнопка, мають властивість Template, можна визначити значення властивості Template точно так само, як і значення інших властивостей, встановлених в об'єкті Style за допомогою Setter (Рис.13)

```
<Style x:Key="deleteButton" TargetType="{x:Type Button}">
    <Setter Property="Cursor" Value="Hand" />
    <Setter Property="Height" Value="20" />
    <Setter Property="Width" Value="28" />
    <Setter Property="Background" Value="White" />
    <Setter Property="Foreground" Value="#F5564A" />
    <Setter Property="Margin" Value="2,0" />
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type Button}">
                <Border x:Name="BtnBorder">
                    <materialDesign:PackIcon
                        HorizontalAlignment="Center"
                        VerticalAlignment="Center"
                        Kind="Delete" />
                </Border>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Рис. 13. Приклад написання стилю кнопки «Delete»

На наступних етапах необхідно визначити шаблон та ефекти за допомогою яких буде зображено певну кнопку або набір кнопок.

Ще однією цікавою функціональною можливістю є створення тригерів властивостей та тригерів подій для зміни значень властивостей та виконання анімації у відповідь на дії користувача, для прикладу, такі як наведення вказівника миші на кнопку та натискання.

Простий спосіб додати інтерактивність (наведення, відпускання миші, клацання та інше) - визначити тригери в межах моделі або стилю. Для створення тригера визначається властивість "умова", наприклад: значення властивості кнопки IsMouseOver дорівнює true. Потім ви визначаєте сеттери (дії), які відбуваються, коли умова тригера істинна. Для прикладу:

```
<ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
        <Setter TargetName="BtnAddBorder" Property="Background"
    Value="#218838" />
```

```
        </Trigger>
        <Trigger Property="IsMouseOver" Value="False">
            <Setter TargetName="BtnAddBorder" Property="Background"
        Value="#28A745" />
         </Trigger>
</ControlTemplate.Triggers>
```

При наведенні курсору на кнопку генерується подія IsMouseOver і активується дія одного із тригерів. Коли курсор зникає з області кнопки активується другий тригер і він просто зупиняє перший. Тому, коли користувач виводить вказівник миші за межі кнопки, кнопка повертається до того початкового стану, в якому вона була до наведення вказівника миші на кнопку.

4.4 Налагодження управління WPF на етапі виконання

Виходячи з практичного досвіду, який я отримала під час налагодження WPF UserControl, потрібно виділити основні етапи роботи:

1. Переконайтеся, що Runtime запущено і що зображення відкрито за допомогою WPF під керуванням користувача.

Далі переконайтеся, що використовувана в даний момент DLL відповідає поточній збірці (версії) проекту User Control (WPFUserControlLibrary).

2. У проекті Visual Studio встановіть точку переривання в місці

кліку на кнопці. Такі точки вказують на рядки коду, де програма буде призупинена для того, щоб ви могли дослідити її стан (Рис.14).

```
29
                3 references
30              public async void Page_Loaded(object sender, RoutedEventArgs e)
31              {
32                  dtGrid.ItemsSource = (await studentService.GetAllAsync()).ToList();
33              }
34
```

Рис. 14. Приклад встановлення точки переривання програми

3. У Visual Studio в розділі Debug виберіть пункт меню Attach to Process.

4. Виберіть процес виконання Runtime.

5. У розділі Приєднати до виберіть Автоматично або відповідну версію .NET Framework.

6. Натисніть на кнопку "Attach".

7. Тепер активуйте точку зупинки, ввівши значення в елементі управління WPF в полі запуску і натисніть на кнопку запуску (Рис.15).

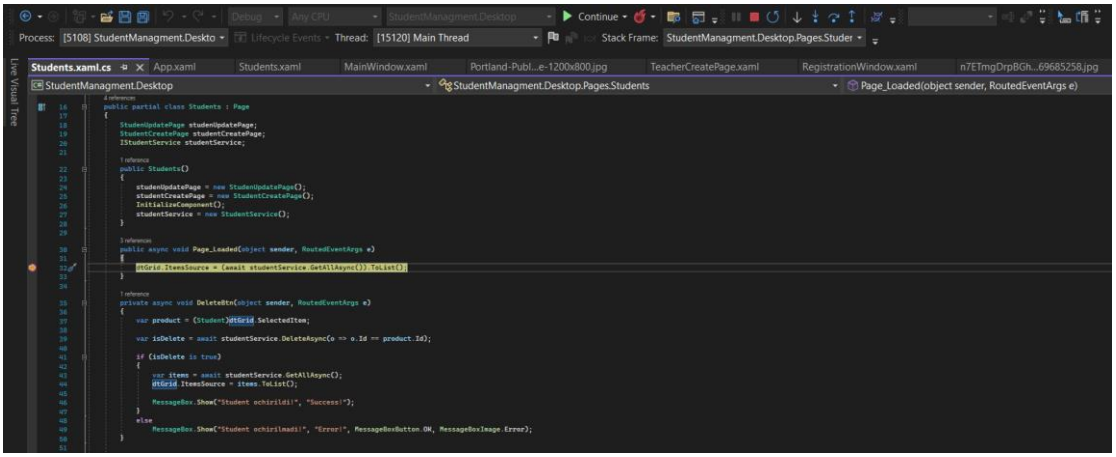


Рис. 15. Точка зупинки програми під час налагодження

Після запуску сеансу відладчика (debugger) програма виконується в звичайному режимі до тих пір, поки не буде досягнута точка зупинки. Коли це відбувається, рядок, на якому програма зупинилася, підсвічується (як можна побачити на рисунку вище (рис.13)) і з'являється вікно інструменту налагодження. Тут виділений рядок ще не виконувався. На даному етапі, програма чекає подальших інструкцій від розробника, а призупинений стан дозволяє досліджувати змінні, які зберігають стан програми.

Якщо розробник впевнений в правильності даних, які отримує та чи інша змінна, можна змінити точку зупинки та перейти до наступного блоку програмного коду.

Варто зауважити, що при запуску Runtime збірки (DLL), на які посилаються в елементі управління користувача WPF в папці \FILES\zenon\custom\additional, або збірки файлів CDWPF копіюються в папку \FILES\zenon\custom\wpfcache. У разі, якщо версія файлу DLL в папці \wpfache дорівнює або більше версії "оригінального" файлу, вона не буде замінена! Тому для цілей налагодження достатньо замінити тільки файл безпосередньо в папці \wpfache. А також попередити, що якщо DLL в папці \additional або в CDWPF оновлюється і ви бачите ці "оновлення", але номер версії не змінюється, DLL в папці \wpcache потрібно попередньо видалити вручну, інакше його не буде оновлено.

Після того, як програму було налагоджено і протестовано, її можна запускати в звичайному режимі та приступати до роботи.

ВИСНОВКИ

У процесі дослідження, було розглянуто найбільш необхідні види програмного забезпечення, які потрібні для якісної та швидкої роботи додатку та, використовуючи які, користувач (а саме персонал деканату) буде здатний пришвидшити роботу обігу паперової роботи, яка пов’язана з даними студентів, дисциплін, викладачів та працівників в загальному, а також даними про оплату навчання.

Створений за допомогою C# .NET, WPF, XML та елементів SQL Server Management Studio (SSMS) настільний застосунок дозволяє працювати в автоматизованому режимі. Знаючи лише індивідуальні та унікальні дані користувача, можна дуже швидко отримати необхідні дані та виконувати над ними дії, маючи на це повноваження. Адміністратори додатку здатні шукати, додавати, видаляти, редагувати дані, в цей час звичайний користувач може лише шукати доступні для нього дані та додавати інформацію про студентів, після цього буде надіслано запит до адміністратора з проханням додати введені дані в базу.

Даний додаток повинен надавати користувачам стандартизований і, разом з тим, зручний інтерфейс, що дуже важливо при роботі з великим об'ємом даних, які будуть зберігатися в базі даних і які тісно пов'язані між собою.

Також, на даний момент підтримується автоматичний контроль перевірки коректності даних та їх нормалізація, можливість без участі розробників формувати нові поля в таблицях баз даних, а також можливість задавати допустимі для них значення.

Хоча неможливо довести абсолютну валідність розробленого інструмента за допомогою єдиної оцінки, яка була проведена мною раніше, оскільки необхідно було б протестувати його на декількох відмінних системах, комп’ютерах різних типів, а можливо і на різних пристроях (наприклад, планшеті).

За час проведення дослідження та розробки даного настільного застосунку було досягнуто усіх поставлених завдань, а саме:

- проведено аналіз бази даних студентів і в додаток до того, бази даних викладачів;
- опрацьовано і зосереджено увагу на лише найбільш необхідну інформацію для створення бази даних та подальшої роботи;
- засобами, згаданими вище, створено комп’ютерний додаток «Особиста картка студента».

В перспективі, планується розширення та модернізація додатку в наступних напрямках: обновлення бібліотек всередині додатку або їх заміна на інші, залежно від функціональних можливостей бібліотеки; розширення роботи для можливості зв’язування даних доданих в застосунку не лише з існуючою базою даних, але й з системою Dynamics365 за допомогою так званого “mapping”, в якій у розробника буде більше функцій та можливостей підтримки потреб користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бублик В. В. Об’єктно-орієнтоване програмування: навч. посібник. Київ, 2015. 624 с

2. Пасічник В. В. Організація баз даних і знань / Пасічник В. В., Резніченко В. А. – BHV, Київ, 2006. – 384 с.

3. Адам Фріман, Windows 8 apps revealed using xaml and c#: using XAML and C#.– 20 листопада 2012

4. Валід Р. Багатоцільова реалізація обов'язкового контролю доступу в реляційних системах управління базами даних / Р. Валід, П. Берд. - Торонто, 2004. - 1010-1020.

5. Пол Бейнон-Девіс, Системи бази даних: навч. посібник. Лондон: palgrave macmillan, 2003. 273 с

6. Ріхтер Джеффрі, CLR VIA C#. Програмування на платформі Microsoft .NET Framework 4.5 мовою C#. 4-е вид.

7. Стівенс Р. WPF 3D: тривимірна графіка за допомогою WPF та C# Paperback - 8 лютого 2018 року. - 426 ст.

8. Троелсен, Джепікс. Мова програмування C# 7 і платформи .NET та .NET Core

9. Чарльз Петцольд, Writing windows 8 apps with C# and XAML, 2013

10. Чаудхурі К. Кулінарна книга з розробки Windows Presentation Foundation: 100 рецептів для створення багатих клієнтських додатків для настільних комп'ютерів на Windows Paperback - 23 лютого 2018 року. - 524 ст.

11. Button In C# (c-sharpcorner.com) [електронний ресурс]. – режим доступу: https://www.c-sharpcorner.com/uploadfile/mahesh/button-in-c-sharp/

12. Hello World app with WPF in C# - Visual Studio (Windows) | Microsoft Learn [електронний ресурс]. – режим доступу: https://learn.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-wpf?view=vs-2022

13. Object-Oriented Programming (C#) | Microsoft Learn [електронний ресурс]. – режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop

14. Repository Design Pattern in C# with Examples - Dot Net Tutorials [електронний ресурс]. – режим доступу: https://dotnettutorials.net/lesson/repository-design-pattern-csharp/

15. SQL Server technical documentation - SQL Server | Microsoft Learn [електронний ресурс]. – режим доступу: https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver16

16. x:Type Markup Extension - XAML | Microsoft Learn [електронний ресурс]. – режим доступу: https://learn.microsoft.com/en-us/dotnet/desktop/xaml-services/xtype-markup-extension

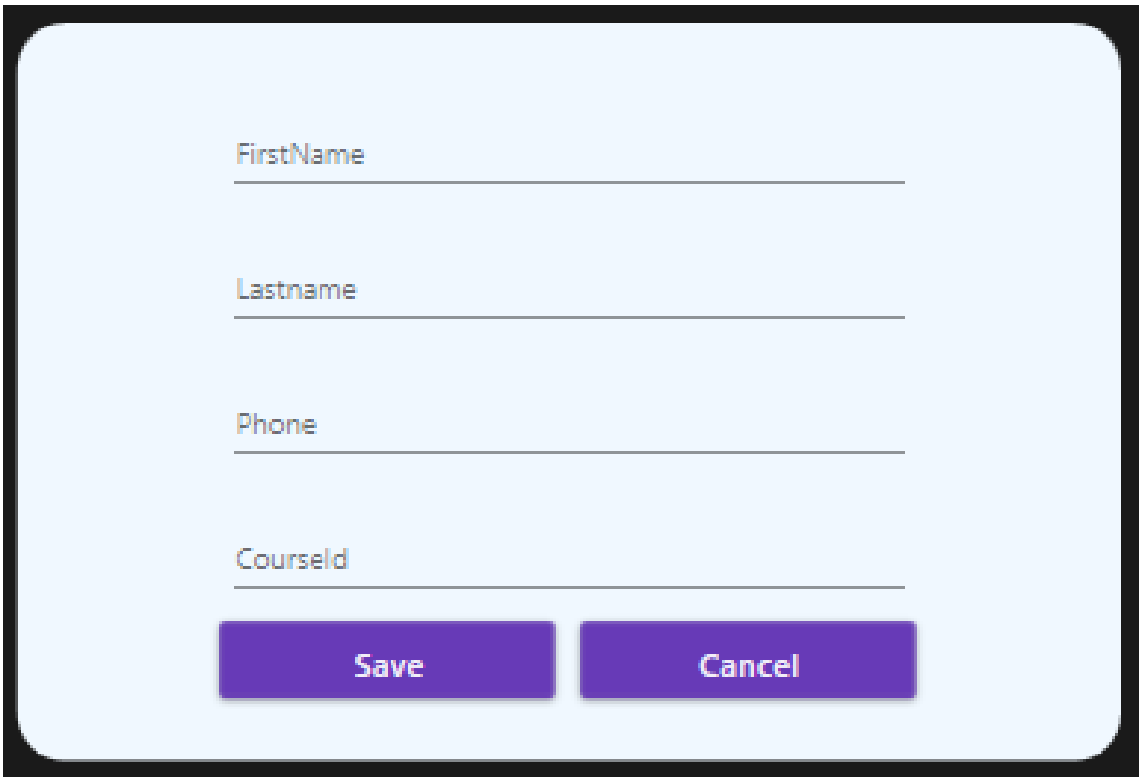
17. What's a Universal Windows Platform (UWP) app? - UWP applications | Microsoft Learn [електронний ресурс]. – режим доступу: https://learn.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide

ДОДАТКИ

ДОДАТОК 1. CRUD СТОРІНКА ДЛЯ СТВОРЕННЯ СТУДЕНТА

```
<Page x:Class="StudentManagment.Desktop.CrudPages.StudentCreatePage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:materialDesign="http://materialdesigninxaml.net/winfx/xaml/themes"
    xmlns:local="clr-namespace:StudentManagment.Desktop.CrudPages"
    mc:Ignorable="d"
    Width="500"
    Height="350"
    Title="StudentCreatePage">

  <Page.Resources>
    <Style BasedOn="{StaticResource MaterialDesignFloatingHintTextBox}"
TargetType="TextBox">
      <Setter Property="Margin" Value="10" />
    </Style>
  </Page.Resources>
  <Grid>
    <Border
      Margin="20"
      Background="AliceBlue"
      CornerRadius="20"
      Cursor="Arrow">
      <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">

        <TextBox x:Name="StudentName"
materialDesign:HintAssist.Hint="FirstName" />
        <TextBox x:Name="StudentLast"
materialDesign:HintAssist.Hint="Lastname" />

        <TextBox x:Name="StudentPhone"
materialDesign:HintAssist.Hint="Phone" />
        <TextBox x:Name="CourseId"
materialDesign:HintAssist.Hint="CourseId" />

        <DockPanel>
          <Button
```

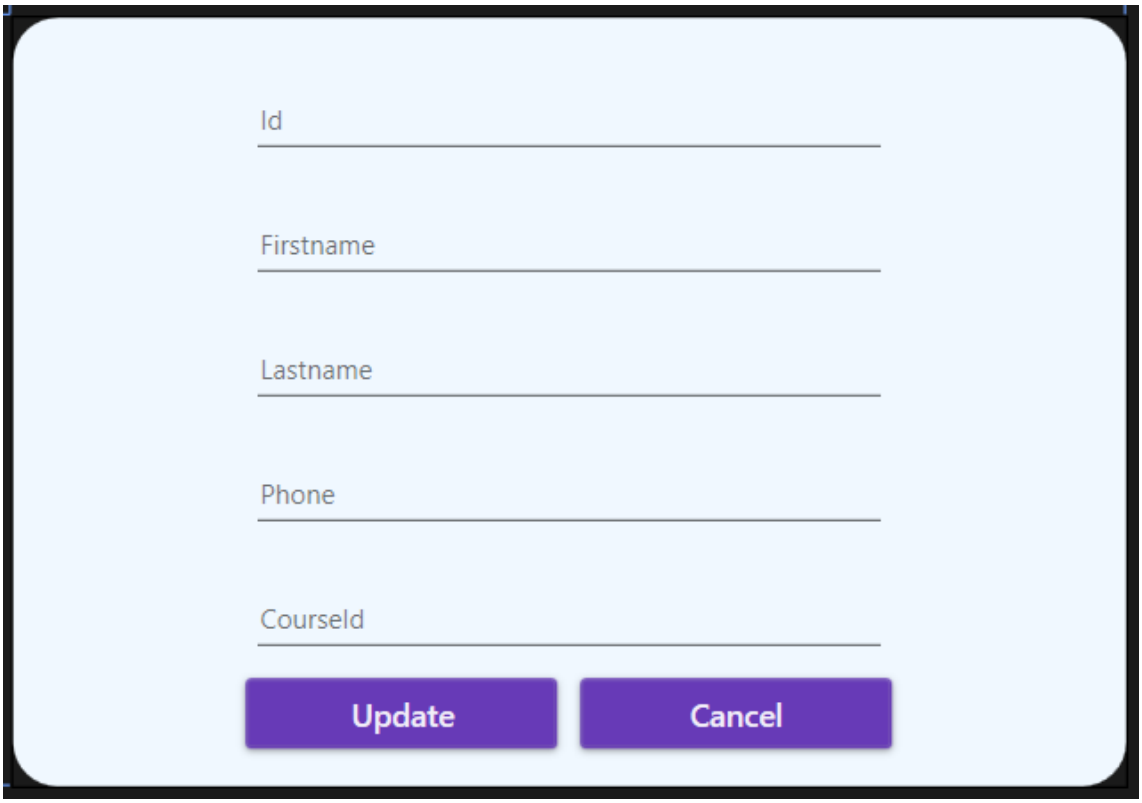
```
            x:Name="CreateStudentButton"
            Width="140"
            Margin="5">
            Save
        </Button>

        <Button
            x:Name="CancelButton"
            Width="140"
            Margin="5">
            Cancel
        </Button>
      </DockPanel>
    </StackPanel>
  </Border>
</Grid>
</Page>
```



ДОДАТОК 2. CRUD СТОРІНКА ДЛЯ ОБНОВЛЕННЯ СТУДЕНТА

```
<Page x:Class="StudentManagment.Desktop.CrudPages.StudenUpdatePage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:materialDesign="http://materialdesigninxaml.net/winfx/xaml/themes"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:StudentManagment.Desktop.CrudPages"
    mc:Ignorable="d"
    Width="500"
    Height="350"
    Title="StudenUpdatePage">

  <Page.Resources>
    <Style BasedOn="{StaticResource MaterialDesignFloatingHintTextBox}"
TargetType="TextBox">
      <Setter Property="Margin" Value="10" />
    </Style>
  </Page.Resources>
  <Grid>
    <Border
      Background="AliceBlue"
      CornerRadius="20"
      Cursor="Arrow">
      <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
        <TextBox x:Name="studentId" materialDesign:HintAssist.Hint="Id" />
        <TextBox x:Name="StudentName"
materialDesign:HintAssist.Hint="Firstname" />
        <TextBox x:Name="StudentLast"
materialDesign:HintAssist.Hint="Lastname" />

        <TextBox x:Name="Phone" materialDesign:HintAssist.Hint="Phone" />
        <TextBox x:Name="CourseId"
materialDesign:HintAssist.Hint="CourseId" />

        <DockPanel>
          <Button Width="140" Margin="5" x:Name="UpdateStudentButton">

            Update
          </Button>

          <Button Width="140" Margin="5" x:Name="CancelButton">
```

```
                Cancel
            </Button>
        </DockPanel>
    </StackPanel>
  </Border>
</Grid>
</Page>
```

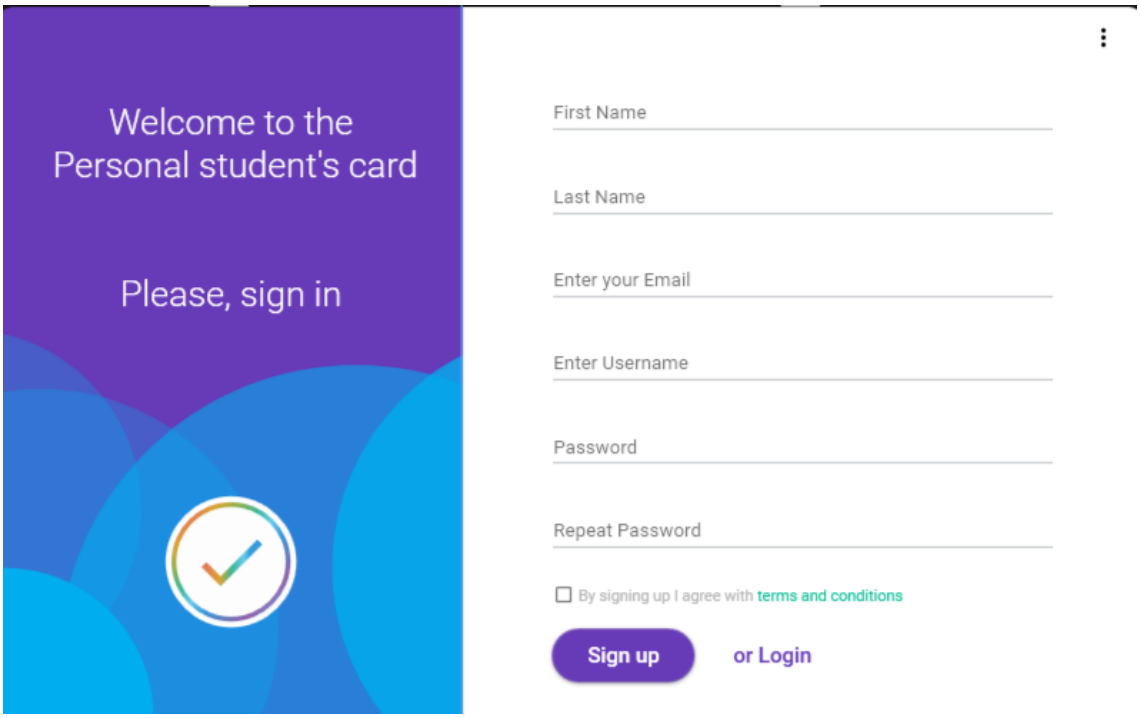


ДОДАТОК 3. СТОРІНКА ДЛЯ ВІДОБРАЖЕННЯ СТУДЕНТІВ З БАЗИ ДАНИХ (SIMPLIFIED)

```
<Page
   x:Class="StudentManagment.Desktop.Pages.Students"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
   xmlns:local="clr-namespace:StudentManagment.Desktop.Pages"
   xmlns:materialDesign="http://materialdesigninxaml.net/winfx/xaml/themes"
   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
   Title="Students"
   d:DesignHeight="450"
   d:DesignWidth="800"
   FontFamily="Poppins"
   Loaded="Page_Loaded"
   mc:Ignorable="d">

   <Border CornerRadius="8">
      <Grid>
         <DataGrid
            x:Name="dtGrid"
            AutoGenerateColumns="False"
            IsReadOnly="True"
            SelectionChanged="dtGrid_SelectionChanged">
            <DataGrid.Columns>
               <DataGridTextColumn Binding="{Binding Id}" Header="Id" />
               <DataGridTextColumn Binding="{Binding FirstName}"
Header="Firstname"   />
               <DataGridTextColumn Binding="{Binding LastName}"
Header="Lastname"     />
               <DataGridTextColumn Binding="{Binding Phone}" Header="Phone" />
               <DataGridTextColumn Binding="{Binding CourseId}" Header="
CourseId   " />

               <DataGridTemplateColumn Header="Actions">
                  <DataGridTemplateColumn.CellTemplate>
                     <DataTemplate>
                        <DockPanel>
                           <Button Click="DeleteBtn" Style="{StaticResource
deleteButton}" />
```

```
                        <Button x:Name="UpdateStudentBtn" Click="UpdateBtn"
Style="{StaticResource editButton}" />
                    </DockPanel>
                  </DataTemplate>
                </DataGridTemplateColumn.CellTemplate>
              </DataGridTemplateColumn>
            </DataGrid.Columns>
          </DataGrid>
          <Button
            x:Name="CreateStudentBtn"
            Width="80"
            Height="32"
            Margin="3"
            HorizontalAlignment="Right"
            VerticalAlignment="Top"
            Click="CreateCourseBtn_Click">
            <ContentControl Content="Create" />
          </Button>
          <Frame x:Name="MainFrame" Margin="40" />
        </Grid>
      </Border>
</Page>
```

ДОДАТОК 4. СТОРІНКА РЕЄСТРАЦІЇ/АВТОРИЗАЦІЇ КОРИСТУВАЧА




```
<Border
        Width="110"
        Height="110"
        Margin="0,0,0,80"
        HorizontalAlignment="Center"
        VerticalAlignment="Bottom"
        Background="#FDFDFD"
        CornerRadius="100">
                <Grid>
          <!--  Colored Ellipse  -->
                <Ellipse
                        Width="100"
                                Height="100"
                        StrokeThickness="4">
                      <Ellipse.Stroke>
                        <LinearGradientBrush>
                      <GradientStop Offset="0.15" Color="#E27C53"/>
                      <GradientStop Offset="0.2" Color="#DCA530" />
                      <GradientStop Offset="0.3" Color="#8BB356" />
                                <GradientStop Offset="0.4" Color="#3BB799" />
                                <GradientStop Offset="0.5" Color="#67CBEE" />
                                <GradientStop Offset="0.6" Color="#3699DB" />
```

```
                    <GradientStop Offset="0.8" Color="#9264AA" />
                    <GradientStop Offset="0.9" Color="#6E94DE" />
                </LinearGradientBrush>
            </Ellipse.Stroke>
        </Ellipse>
        <!--  Colored Tick Icon  -->
        <materialDesign:PackIcon
            Width="70"
            Height="70"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Kind="Tick">
            <materialDesign:PackIcon.Foreground>
             <LinearGradientBrush StartPoint="0,1">
              <GradientStop Offset="0.1" Color="#E27C53"/>
              <GradientStop Offset="0.3" Color="#DCA530"/>
              <GradientStop Offset="0.5" Color="#3BB799"/>
              <GradientStop Offset="0.7" Color="#67CBEE"/>
              <GradientStop Offset="0.8" Color="#3699DB"/>
             </LinearGradientBrush>
            </materialDesign:PackIcon.Foreground>
        </materialDesign:PackIcon>
    </Grid>
</Border>
```

СТВОРЕННЯ ТЕКСТ-БОКСІВ ДЛЯ ВВОДУ ДАНИХ

```
<Border
    Grid.Column="1"
    Margin="0,0,0,10"
    Background="#ffffff"
    CornerRadius="0 10 10 0"
    MouseDown="Border_MouseDown">
    <Grid>

        <materialDesign:PopupBox
            Height="25"
            Margin="0 0 15 550"
            HorizontalAlignment="Right"
            PlacementMode="BottomAndAlignRightEdges"
            StaysOpen="False">
```

```
    <StackPanel>

        <Button
            Margin="0,8,0,0"
            Content="Help Me"
            ToolTip="Having Trouble Logging ?" />

        <Button
            x:Name="btn_exit"
            Click="exitApp"
            Content="Exit"
            ToolTip="Close Application" />

    </StackPanel>
</materialDesign:PopupBox>

<!--  Inputs  -->
<StackPanel Margin="75,0" VerticalAlignment="Center">

    <TextBox
        Margin="0,27"
        FontSize="16"
        materialDesign:HintAssist.FloatingOffset="0,-20"
        materialDesign:HintAssist.Hint="First Name"
        BorderBrush="#C5C8CC"
        BorderThickness="0,0,0,1.5"
        Style="{StaticResource MaterialDesignFloatingHintTextBox}" />

    <TextBox
        Margin="0"
        FontSize="16"
        materialDesign:HintAssist.FloatingOffset="0,-20"
        materialDesign:HintAssist.Hint="Last Name "
        BorderBrush="#C5C8CC"
        BorderThickness="0,0,0,1.5"
        Style="{StaticResource MaterialDesignFloatingHintTextBox}" />

    <TextBox
        Margin="0,27"
        FontSize="16"
        materialDesign:HintAssist.FloatingOffset="0,-20"
        materialDesign:HintAssist.Hint="Enter your Email"
        BorderBrush="#C5C8CC"
        BorderThickness="0,0,0,1.5"
```

```
            Style="{StaticResource MaterialDesignFloatingHintTextBox}" />

        <TextBox
            FontSize="16"
            materialDesign:HintAssist.FloatingOffset="0,-20"
            materialDesign:HintAssist.Hint="Enter Username"
            BorderBrush="#C5C8CC"
            BorderThickness="0,0,0,1.5"
            Style="{StaticResource MaterialDesignFloatingHintTextBox}" />

        <PasswordBox
            Margin="0,27"
            FontSize="16"
            materialDesign:HintAssist.FloatingOffset="0,-18"
            materialDesign:HintAssist.Hint="Password"
            BorderBrush="#C5C8CC"
            BorderThickness="0,0,0,1.5"
            Style="{StaticResource MaterialDesignFloatingHintPasswordBox}"
/>

        <PasswordBox
            FontSize="16"
            materialDesign:HintAssist.FloatingOffset="0,-18"
            materialDesign:HintAssist.Hint="Repeat Password"
            BorderBrush="#C5C8CC"
            BorderThickness="0,0,0,1.5"
            Style="{StaticResource MaterialDesignFloatingHintPasswordBox}"
/>

        <CheckBox Margin="0,30,0,20" FontSize="13">
            <TextBlock>
                <Run Foreground="#b6b6b6">By signing up I agree with</Run>
                <Run Foreground="#07BF96">terms and conditions</Run>
            </TextBlock>
        </CheckBox>

        <StackPanel Orientation="Horizontal">
            <Button
                Width="120"
                Height="45"
                materialDesign:ButtonAssist.CornerRadius="22"
                Background="#673AB7"
                Content="Sign up"
                FontSize="18"
```

```
                Foreground="#ffffff" />
            <Button
                Width="120"
                Height="45"
                Margin="5,0,0,0"
                materialDesign:ButtonAssist.CornerRadius="22"
                Content="or Login"
                FontSize="18"
                Click="LoginBtn"
                Style="{StaticResource MaterialDesignFlatButton}" />
        </StackPanel>
    </StackPanel>
  </Grid>
</Border>
```