МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота бакалавра

ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ КЕРУВАННЯ ЗАМОВЛЕННЯМИ ФОТОЛАБОРАТОРІЇ

Здобувач освіти гр. ІНз-81с — Сергій ПУГАЧ

Науковий керівник — Дмитро ПРИЛЕПА

Завідувач кафедри
доктор технічних наук, професор. — Анатолій ДОВБИШ

СУМИ 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Центр заочної, дистанційної і вечірньої форм навчання

Кафедра комп'ютерних наук

Затверджую___________

Зав. кафедрою Довбиш А.С.

"_____" ____________2022 р.

ЗАВДАННЯ

до кваліфікаційної роботи

Здобувача вищої освіти четвертого курсу, групи ІНз-81с спеціальності "Інформатика" заочної форми навчання Пугача Сергія Анатолійовича.

Тема: "**Інформаційне та програмне забезпечення системи керування замовленнями фотолабораторії** "

Затверджена наказом по СумДУ

№________ від ___________2022 р.

Зміст пояснювальної записки: 1) аналітичний огляд роботи фотолабораторії та систем керування замовленнями; 2) постановка завдання й формування завдань дослідження; 3) проектування бази даних та програмного забезпечення.

Дата видачі завдання "_____"____________2022 р.

Керівник кваліфікаційної роботи ____________ Прилепа Д. В.

Завдання прийняв до виконання ____________ Пугач С. А.

РЕФЕРАТ

Записка: 50 стор., 13 рис., 3 додаток, 15 джерел.

Об’єкт дослідження — процес проектування та реалізації інформаційного та програмного забезпечення системи керування замовленнями фотолабораторії.

Мета роботи — розробка та програмна реалізація системи керування замовленнями фотолабораторії.

Методи дослідження — методи проектування ігрових систем, методи процедурної генерації.

Результати — проаналізовано роботу фотолабораторії та узгоджено основні вимоги замовника (керівника фотолабораторії), проаналізовані інструменти і умови для розробки бази даних та програмного додатку для вирішення поставленої задачі. Згідно вимог розроблено базу даних, спроектовано інтерфейс та реалізовано систему керування послугами (внесення змін у базу даних). Практична реалізація програмного продукту проводиться на мові C# у середовищі розробки Visual Studio.

СЕРЕДОВИЩЕ РОЗРОБКИ, C#, MYSQL WORKBENCH, CODE FIRST, MVVM, UNIT OF WORK, ФОТОЛАБОРАТОРІЯ

ЗМІСТ

ВСТУП.. 5
1 ІНФОРМАЦІЙНИЙ ОГЛЯД.. 7
1.1 Керування замовленнями.. 7
1.2 Особливості роботи фотолабораторії.. 10
1.3 Постановка задачі.. 11
2 ВИБІР ІНСТРУМЕНТІВ ДЛЯ РІШЕННЯ... 13
2.1 Діаграма прецедентів 13
2.2 MySQL сервер.. 18
2.3 MySQL Workbench 19
2.4 Microsoft Visual Studio 2022 19
3 ПРОЕКТУВАННЯ БАЗИ ДАНИХ ТА РОЗРОБКА ПРОГРАМНОГО ДОДАТКУ 21
3.1 Побудова ERD діаграми 21
3.2 Code First 24
3.3 Інтерфейс.. 26
3.4 Unit Of Work 31
ВИСНОВКИ... 33
СПИСОК ЛІТЕРАТУРИ.. 34
ДОДАТКИ.. 36
Додаток А.. 36
Додаток Б 42
Додаток В 48

ВСТУП

Фотолабораторії з‘явились одразу ж після винаходу дагеротипії та призначались для сенсибілізації срібних пластин, їх проявлення парами ртуті й покриття золотом. Розповсюдження мокрого колодієвого процесу значно видозмінило технологію, зробивши фотолабораторії мобільними. Необхідність експонування й обробки фотопластин протягом кількох хвилин після поливу емульсії змушувала перемістити все лабораторне обладнання якомога ближче до місця зйомки. З‘явились фотолабораторії в наметах і пересувних вагончиках. Фотодрук в ту епоху виконувався контактним способом на так званому «денному» фотопапері, що потребувало лише фіксації в розчині віраж-фіксажу. Світлочутливість хлорсрібного фотопаперу такого типу була низькою і знаходилась переважно в ультрафіолетовому діапазоні випромінення. Експонування знімка в спеціальній контактній рамці відбувалось на сонячному світлі у дворі фотоательє. Тому основним призначенням фотолабораторії довгий час залишалась перезарядка касет та фотографічна обробка негативів [1].

Звичну роль фотолабораторія набуває з появою желатиносрібного фотопаперу з проявленням, придатного до проекційного друку. Висока світлочутливість до видимого світла змушувала обробляти такий фотопапір в затемненому приміщенні з червоним або жовто-зеленим з неактиничним освітленням. Поступово основним призначенням фотолабораторії стає фотодрук та обробка фотопаперу. Це наклало відбиток на її структуру, де значну площу стали займати проявочні столи, а також обладнання для промивки та сушки фотовідбитків. Незважаючи на трансформації, фотолабораторія протягом всієї історії аналогової фотографії вважалась її невід‘ємною частиною, і до приходу цифрових технологій була розповсюджена як в професійній, так і любительській практиці. В даний час класична «мокра» лабораторія стала екзотикою, поступившись місцем компактним міні-фотолабораторіям та струменевим принтерам в фотоцентрах [1].

Сучасний споживач послуг не розрізняє поняття «фотоцентр» і «фотолабораторія», тож підприємства використовують класичну назву своєї діяльності, незважаючи на широкий спектр надаваних послуг і торгівлю супутніми товарами. Цифрові технології значно підвищили швидкість друку фотографій та зменшили вартість, а також надали можливості цифрової обробки, художнього оформлення і колажування фотографій. «Фото на документи», друк зображень і документів, виготовлення поліграфічної та рекламної продукції – неповний список послуг сучасних фотолабораторій, а популярна практика онлайн замовлень розширює «територію» обслуговування до меж країни, а іноді й далі. Таким чином з‘являється необхідність контролювати велику кількість замовлень, швидко реагувати на запити клієнтів, що вимагає розробки та впровадження системи керування замовленнями фотолабораторії.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Керування замовленнями

У сучасному бізнесі логістичні активності, пов‘язані з керуванням замовленнями, стають критичними з погляду споживчого обслуговування. Численними дослідженнями встановлено, що час виконання таких процедур як прийом, підготовка, передача, обробка, моніторинг замовлень становить від 50 до 70% загального логістичного циклу його виконання для більшості галузей промисловості західних фірм. Тому для підвищення якості обслуговування споживачів та якнайшвидшого задоволення їх очікувань необхідно скорочувати час та кількість складових циклу за рахунок більш ефективного логістичного менеджменту [2].

Керування замовленнями – це відслідковування замовлень протягом всього життєвого циклу – від отримання до виконання замовлення та керування пов‘язаними з ним даними, процесами, людьми [3].

Керування замовленнями починається одразу після розміщення замовлення клієнтом і закінчується тільки тоді, коли він отримує свій товар чи послугу. Це дозволяє підприємству повністю координувати процес виконання замовлення: від його комплектування, інвентаризації та доставки до відслідковування готовності послуг. Робочий процес може залежати від потреб компанії, але, як правило, складається з трьох етапів:

- Розміщення. Клієнт розміщує замовлення через автоматизовану форму. Співробітник відділу продажу перевіряє відомості та підтверджує замовлення.
- Виконання. Співробітник складу підтверджує інформацію про поставку, створює рахунок і виконує замовлення: відвантаження, упаковка та поставка.
- Керування реєстром. Наявність запасів відстежується в міру їх коливань залежно від попиту [3].

Необхідно відмітити, що добре скоординовані й швидкі поставки продукції споживачам у відповідності до їх замовлень забезпечує компаніям

конкурентні переваги. Здатність швидко і надійно задовольняти потреби замовників не менш важлива, як висока якість надаваних послуг і товарів. Фокусування менеджменту на виконані замовлень приводить до необхідності обліку кожного окремого замовлення та визначення відповідних активностей для його виконання. Ключовим актором успіху керування замовленнями має бути знаходження ефективних шляхів задоволення багатьох різноманітних типів потреб споживачів з їх специфічними очікуваннями та вимогами до якості продукції. У зв‘язку з цим багато компаній замінюють прості процедури замовлень комплексними, що включають усі необхідні логістичні активності. Фокусуючись на вхідних вимогах замовника, менеджмент замовлень визначає шляхи й джерела їх найкращого виконання з позиції якості сервісу та мінімізації витрат. Інтегрований підхід до керування замовленнями висуває певні вимоги до складових циклу замовлення, що набуває більш розкритого вигляду:

- Прийом та попередня обробка замовлення. Менеджмент замовлень отримує всі замовлення споживачів із різних потоків, включаючи пункти роздрібної торгівлі, електронні телекомунікаційні мережі та системи та інших джерел.
- Конфігурування. Кожне замовлення складається з низки продуктових та сервісних атрибутів, які необхідно врахувати під час його виконання. Менеджмент замовлень ідентифікує ці вимоги, навіть коли вони не є чітко специфікованими документально.
- Передача замовлень. Для передачі замовлень від однієї ланки до іншої мають бути використані всі можливі засоби зв‘язку та передачі інформації з агрегуванням даних у сучасних надійних та швидкодіючих телекомунікаційних каналах.
- Визначення джерел виконання замовлень. На основі агрегованих даних про замовлення, менеджмент повинен визначити конкретні джерела їх задоволення з урахуванням усіх необхідних продуктових та сервісних атрибутів. Ці джерела можуть бути розміщені як у дистрибутивній мережі, так і в самому виробництві продукції.

- Планування. Для визначених на попередньому етапі джерел розробляються плани виконання замовлень за укрупненими та специфікованими групами товарів з розрахунком часу циклів виконання та доставки замовлень у кожну торгову точку.
- Моніторинг та контроль. Процедури виконання та доставки замовлених обсягів продукції споживачам супроводжуються безперервним контролем термінів, обсягів та якості поставок за допомогою ефективної системи моніторингу, забезпечуючи цим реалізацію плану виконання замовлень.

Система керування замовленнями (OMS) – це цифровий підхід до керування життєвим циклом замовлення. За її допомоги можна відстежувати всю інформацію та процеси, включно з надходженням замовлення, керуванням запасами, виконання замовлення та післяпродажне обслуговування. OMS забезпечує прозорість як для бізнесу, так і для покупця. Організації можуть практично в режимі реального часу отримувати уявлення про запаси, а замовники можуть відстежувати доставку свого замовлення [3].

Основні функції ефективного керування замовленнями [3]:

- Прозорість. Наочне уявлення всього ланцюга поставок та ізоляція подій для прогнозування проблем та розробки більш ефективних процесів.
- Аналітика. Налаштування процесів керування замовленнями у відповідності до бізнес-правил організації та намічених показників продуктивності.
- Гнучкість. Поділ замовлень або подій на окремі унікальні завдання, які можна передати до відповідних систем або відповідних фахівців.
- Система обліку запасів у реальному часі забезпечує цілісне уявлення про запаси, включаючи товари в наявності, товари, що доставляються, та поточні рівні попиту, скорочуючи потребу в прискореному виконанні замовлень або зберіганні надлишкових запасів.

- Планування доставки та операцій. Узгодження зобов‘язань щодо постачання запасів, ресурсів і навичок, а також ефективніше виконання запитів на обслуговування.
- Технології залучення клієнтів. Надання співробітникам, що працюють з клієнтами, єдине уявлення про клієнтів, запаси та ресурси, щоб підвищити ефективність транзакцій.
- Оптимізація виконання. Аналіз даних та видача рекомендацій з урахуванням вибору способу та місця, термінів та вартості доставки замовлень.

1.2 Особливості роботи фотолабораторії

У сучасному світі використовується безліч методів передачі цифрової інформації в тому числі вихідних зображень стосовно замовлень на друк у фотолабораторії. Це можуть буду файлообмінники на кшталт Fex.net і dropmefiles.com, спеціалізовані сервіси як Google Фото, електронна пошта, соціальні мережі, меседжери як Viber або Telegram, завантаження на власний сервер фотолабораторії, передача на фізичному носії та інші. Саме замовлення може бути оформленим на сайті через спеціальну форму, тому документованим і нести вичерпну інформацію, а може бути поданим у вільній формі через будь-які засоби зв‘язку. Все це ускладнює роботу менеджменту, але є необхідним з точки зору клієнтоорієнтованості.

Працівник, що опрацьовує замовлення повинен:

- Отримати вихідні файли і перевірити їх на придатність до друку;
- Конфігурувати замовлення в залежності від специфікацій та за необхідності зробити уточнення у клієнта;
- Підготувати рахунок;
- Сформувати задачу на виробництво;

На виробництво замовлення беруться з черги задач, виконуються і доставляються клієнту обраним способом. При малій кількості замовлень були можливими ручна підготовка специфікацій, рахунків та решти документації, але

з ростом популярності фотолабораторії стали займати забагато часу, що призводить до необхідності автоматизувати хоча б частину роботи.

Отримання файлів єдиним методом неможливе. Більшість клієнтів прагнуть використати найпростіший та найзручніший саме для них, навіть, якщо він призводить до відносної втрати якості фото (меседжери стискають зображення для зменшення трафіку). Таким чином обмеження способів передачі файлів призведе до втрати клієнтів.

Отже автоматизація документування замовлення є головним завданням даної роботи. Програмне забезпечення повинне мати зручний інтерфейс, що дозволить легко формувати замовлення, додаючи послуги зі списку надаваних фотолабораторією, контролювати додаткові опції як стосовно виробництва, так стосовно доставки чи дисконтних програм. Також ПЗ має формувати рахунок для клієнта та зберігати замовлення як задачу на виробництво в базі даних.

1.3 Постановка задачі.

Метою даної роботи є розробка та програмна реалізація системи керування замовленнями з урахуванням специфіки організації роботи сучасної фотолабораторії. Для досягнення поставленої мети необхідно виконати такі завдання:

1) Розробити інформаційну модель системи керування замовленнями;
2) Визначити та оптимізувати структуру баз даних системи керування замовленнями;
3) Розробити інтерфейс користувача системою керування замовленнями;
4) Розробити бізнес логіку системи;
5) Обрати засоби для програмної реалізації системи;
6) Виконати програмну реалізацію та тестування системи.

Також ПЗ повинно задовольняти наступним вимогам:

1) Формування відповіді як російською та українською мовами
2) Можливість розбирати стандартизований запит клієнта з буфера обміну для заповнення деяких параметрів замовлення: Спосіб

одержання замовлення, ПІБ отримувача, необхідність і спосіб корекції фотографій.

3) Поле «кількість» повинно приймати формулу типу x + y*a + z*b +..., де x,y,… - це кількість файлів, a,b,… - тираж, тобто кількість копій кожного файлу.
4) Параметри, що рідко змінюються (залежність часу виконання замовлення від його розміру, вартість доставки та ін.), текстові роз‘яснення до деяких параметрів замовлення та шаблони відповіді мають зберігатись в окремих файлах.
5) Програма при запуску будує дерево - список послуг розділений на категорії на основі бази даних.
6) Сформовані замовлення мають зберігатись в базі даних з можливістю внести зміни і повторно виставити рахунок за вимогою клієнта.
7) Адміністратор повинен мати змогу редагувати базу даних.

2 ВИБІР ІНСТРУМЕНТІВ ДЛЯ РІШЕННЯ

2.1 Діаграма прецедентів

UML (англ. Unified Modeling Language) — уніфікована мова моделювання, використовується у парадигмі об'єктно-орієнтованого програмування. Є невід'ємною частиною уніфікованого процесу розробки програмного забезпечення. UML є мовою широкого профілю, це відкритий стандарт, що використовує графічні позначення для створення абстрактної моделі системи, яка називається UML-моделлю. UML був створений для визначення, візуалізації, проєктування й документування в основному програмних систем [4].

UML може бути застосовано на всіх етапах життєвого циклу аналізу бізнес-систем і розробки прикладних програм. Різні види діаграм які підтримуються UML, і найбагатший набір можливостей представлення певних аспектів системи робить UML універсальним засобом опису як програмних, так і ділових систем.

Діаграми дають можливість представити систему (як ділову, так і програмну) у такому вигляді, щоб її можна було легко перевести в програмний код.

Основною причиною використання мови UML є спілкування розробників між собою.

Крім того, UML спеціально створювалася для оптимізації процесу розробки програмних систем, що дозволяє збільшити ефективність їх реалізації у кілька разів і помітно поліпшити якість кінцевого продукту [4].

Діаграма прецедентів (або діаграма варіантів використання) — в UML, діаграма, на якій зображено відношення між акторами та прецедентами в системі. Суть діаграми прецедентів полягає в тому, що проєктована система подається у вигляді множини сутностей чи акторів, що взаємодіють із системою за допомогою так званих варіантів використання. Варіант використання (англ. use case) використовують для описання послуг, які система надає актору. Іншими словами, кожен варіант використання визначає деякий набір дій, який виконує

система під час діалогу з актором. При цьому нічого не говориться про те, яким чином буде реалізовано взаємодію акторів із системою [5].

Для побудови діаграми прецедентів використано програму UMLet — це інструмент UML з відкритим вихідним кодом на основі Java, призначений для навчання уніфікованій мові моделювання та швидкого створення діаграм UML. Це інструмент малювання, а не інструмент моделювання, оскільки не існує базового словника чи каталогу об'єктів дизайну, які можна повторно використовувати. UMLet поширюється під Загальною публічною ліцензією GNU [6].

UMLet має простий користувальницький інтерфейс, який використовує коди форматування тексту для модифікації основних фігур за допомогою прикрас та анотацій, тому на шляху користувача немає лісу піктограм або діалогових вікон списку параметрів. Це вимагає від користувача вивчення ще однієї мови текстової розмітки, але зусилля невеликі, а розмітка очевидна для досвідченого дизайнера UML. Інтерфейс зображено на рисунку 2.1.

UMLet може експортувати діаграми до зображень (eps, jpg), форматів малюнків (SVG), форматів документів (PDF). Буфер обміну можна використовувати для копіювання та вставки діаграм як зображень в інші програми. Є можливість створювати власні елементи UML.

Основні об'єкти малювання можна змінювати та використовувати як шаблони, що дозволяє користувачам налаштовувати програму відповідно до своїх потреб. Це вимагає програмування елементів на Java.

Підтримуються найважливіші типи діаграм UML: клас, варіант використання, послідовність, стан, розгортання, діяльність.

Власним форматом файлу програми є UXF, розширення XML, призначене для обміну моделями UML.

UMLet працює окремо або як плагін Eclipse в Windows, OS X і Linux. [6]

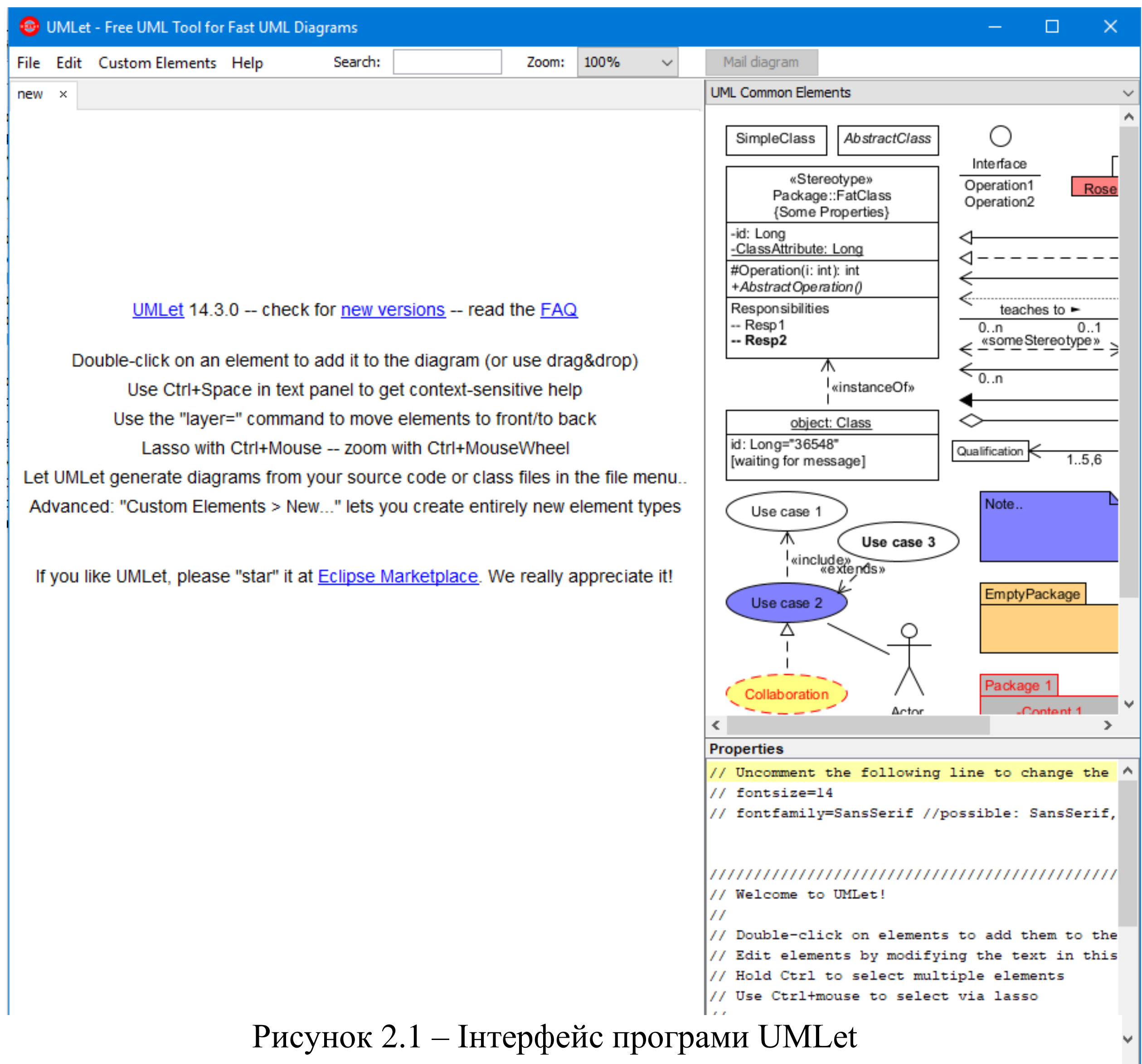

Рисунок 2.1 – Інтерфейс програми UMLet

У результаті аналізу особливостей роботи фотолабораторії виділяються такі актори:

1. «Замовник»:
   – Замовляє друк фотографії чи інші послуги.
2. «Менеджер»:
   – Перевіряє параметри замовлення;
   – Формує рахунок на оплату;
   – Поміщає замовлення в чергу замовлень.
3. «Виконавець»:
   – Надає замовлені послуги;
   – Передає виконане замовлення клієнту або в службу доставки.

4. «Служба доставки»:
   – Доставляє замовлення.
5. «Адміністратор»:
   – Корегує асортимент. Керує актуальним списком послуг.

Виходячи з потреб дійових осіб, виділяються наступні варіанти використання з переліком специфікацій:

1. «Черга замовлень»:
   – Короткий опис – список невиконаних замовлень
     - Основний потік подій – починає виконуватись, коли менеджер додає замовлення до списку;
     - Альтернативні потоки відсутні;
     - Передумови – передбачає контроль виконання замовлення;
     - Постумови – після виконання замовлення видаляється з черги.
2. «Замовлення»:
   – Короткий опис – замовлення послуг фотолабораторії;
     - Основний потік подій – починає виконуватись, коли замовник надає запит на одержання бажаних послуг;
     - Альтернативні потоки – відсутні;
     - Передумови – «прихід» замовника (повідомлення зі списком бажаних послуг з асортименту);
     - Постумови – замовник отримує бажані послуги;
3. «Асортимент»:
   – Короткий опис – список послуг, що можуть бути надані;
     - Основний потік – Починає виконуватись, коли замовник формує замовлення;
     - Альтернативні потоки – відсутні;
     - Передумови – адміністратор формує актуальний список послуг фотолабораторії;
     - Постумови – замовник отримує список доступних послуг.
4. «Бухгалтерія»:

- Короткий опис - містить дані про рахунки клієнтів, оплату за послуги;
  - Основний потік подій – формується довідник про грошові операції на підприємстві;
  - Альтернативні потоки – відсутні;
  - Передумови – передбачає наявність замовлення;
  - Постумови – немає.

5. «Знижки»:
   - Короткий опис – містить дані про знижки постійних клієнтів;
     - Основний потік – формується довідник про знижки, які можуть бути надані постійним клієнтам;
     - Альтернативні потоки – відсутні;
     - Передумови – немає;
     - Постумови – при успішному виконанні замовник отримує знижку.
6. «Виконання»:
   - Короткий опис – надання замовнику вказаних у замовленні послуг;
     - Основний потік – виконання замовлення (друк фотографій та ін.) та передача результатів замовнику;
     - Альтернативні потоки – передача результатів виконання замовлення службі доставки;
     - Передумови - передбачає існування замовлення в черзі замовлень;
     - Постумови – замовник отримує виконане замовлення.
7. «Доставка»:
   - Короткий опис – результати виконання замовлення відправляються замовнику;
     - Основний потік подій – відсутній;
     - Альтернативні потоки - результати виконання замовлення відправляються замовнику;

- Передумови – замовлення сформовано зі вказаним способом і адресою доставки;
- Постумови – при успішному виконанні замовлення доставляється замовнику.

Поведінка акторів і прецедентів системи відображаються їх відносинами, які показані на рисунку 2.2

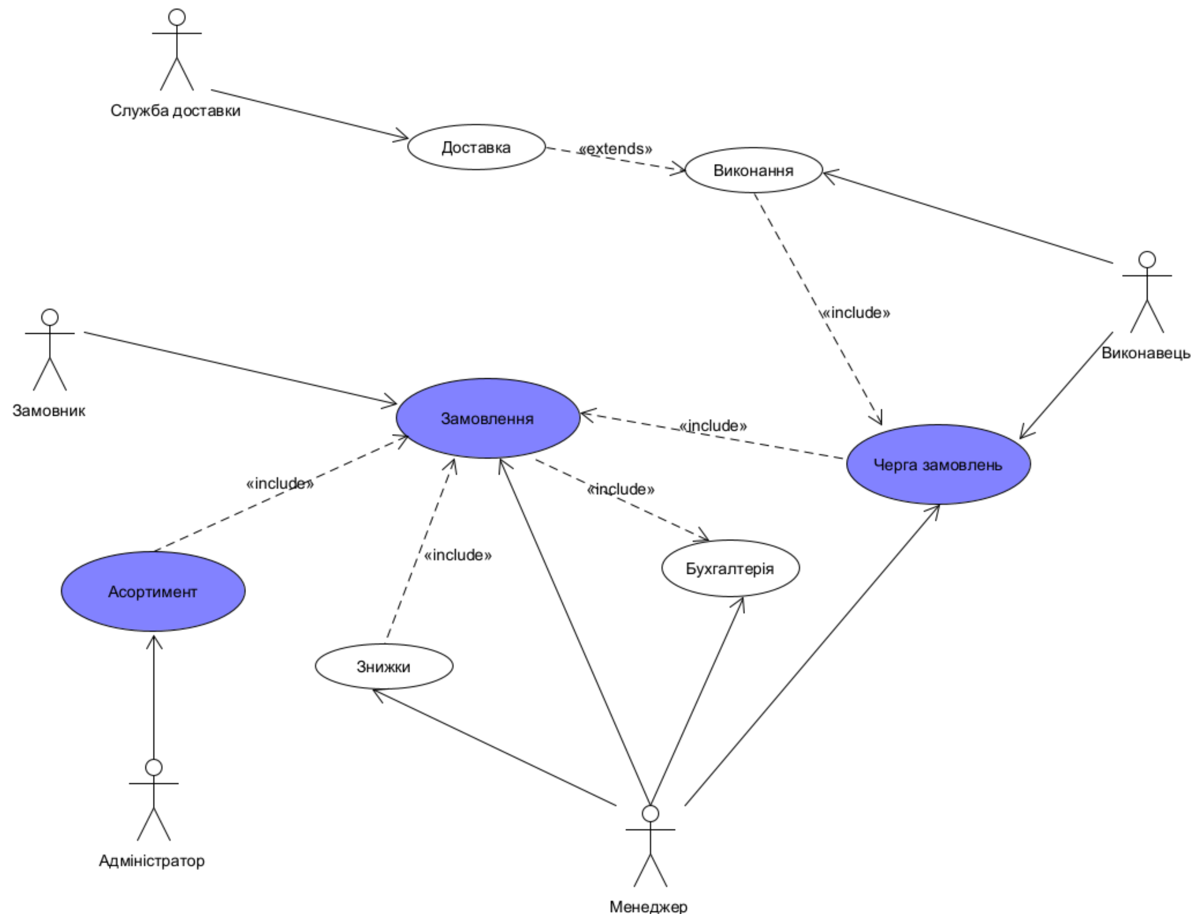

Рисунок 2.2 - Діаграма прецедентів

2.2 MySQL сервер

Можливості сервера MySQL:

1. простота у встановленні та використанні;
2. підтримується необмежена кількість користувачів, що одночасно працюють із БД;
3. кількість рядків у таблицях може досягати 50 млн.;

4. висока швидкість виконання команд;
5. наявність простої і ефективної системи безпеки.

2.3 MySQL Workbench

MySQL Workbench — інструмент для візуального проектування баз даних, що інтегрує проектування, моделювання, створення й експлуатацію БД в єдине безкоштовне оточення для системи баз даних MySQL.

MySQL Workbench спрощує розробку та обслуговування бази даних, автоматизує трудомісткі та схильні до помилок завдання, а також покращує обмін даними між командами розробників. Утиліти для перевірки моделей та схем використовують стандарти найкращої практики моделювання даних, а також застосовують специфічні для MySQL стандарти проектування, щоб не було помилок під час створення нових діаграм або створення фізичних баз даних MySQL.

Можливості програми:

- дозволяє уявити модель бази даних в графічному вигляді;
- функціональний механізм установки зв'язків між таблицями, в тому числі «багато до багатьох» зі створенням таблиці зв'язків;
- відновлення структури таблиць з уже існуючою на сервері БД;
- зручний редактор запитів, що дозволяє відразу ж відправляти їх із сервером і отримати відповідь у вигляді таблиці;

2.4 Microsoft Visual Studio 2022

Microsoft Visual Studio — серія продуктів фірми Майкрософт, які містять інтегроване середовище розробки програмного забезпечення та низку інших інструментальних засобів. Ці продукти дають змогу розробляти як консольні програми, так і програми з графічним інтерфейсом, включно з підтримкою технології Windows Forms та WPF, а також вебсайти, вебзастосунки, вебслужби

як у рідному, так і в керованому кодах для всіх платформ, що підтримуються Microsoft Windows, Windows Mobile, Windows Phone, Windows CE, .NET Framework, .NET Compact Framework та Microsoft Silverlight.

Visual Studio включає в себе редактор початкового коду з підтримкою технології IntelliSense і можливості найпростішого рефакторингу коду. Вбудований відлагоджувач (зневаджувач, англ. debugger, також зустр. англіцизм: деба́гер) може відлагоджувати як початковий код, так і код машинного рівня. Решта вбудованих інструментів містить редактор форм для спрощення створення графічного інтерфейсу програмного додатка, веб-редактор, дизайнер класів та дизайнер схеми бази даних. Visual Studio дозволяє створювати й підключати сторонні додатки (модулі чи плагіни) для розширення функціональності практично на кожному рівні, включно з додаванням підтримки контролю версій початкового коду (як наприклад Subversion, Visual SourceSafe чи Git), додавання нових наборів інструментів (наприклад, для редагування та візуального проектування коду на предметно-орієнтованих мовах програмування) чи інструментів для інших аспектів процесу розробки програмного забезпечення.

3 ПРОЕКТУВАННЯ БАЗИ ДАНИХ ТА РОЗРОБКА ПРОГРАМНОГО ДОДАТКУ

3.1 Побудова ERD діаграми

Для побудови ERD діаграми буде використовуватись MySQL Workbench оскільки в цій програмі є можливість будувати ERD діаграми, генерувати скрипт побудови таблиць згідно діаграми в обраній базі даних з представленням відповідного скрипта, а інтерфейс програми досить зрозумілий та зручний (див. Рис 3.1).

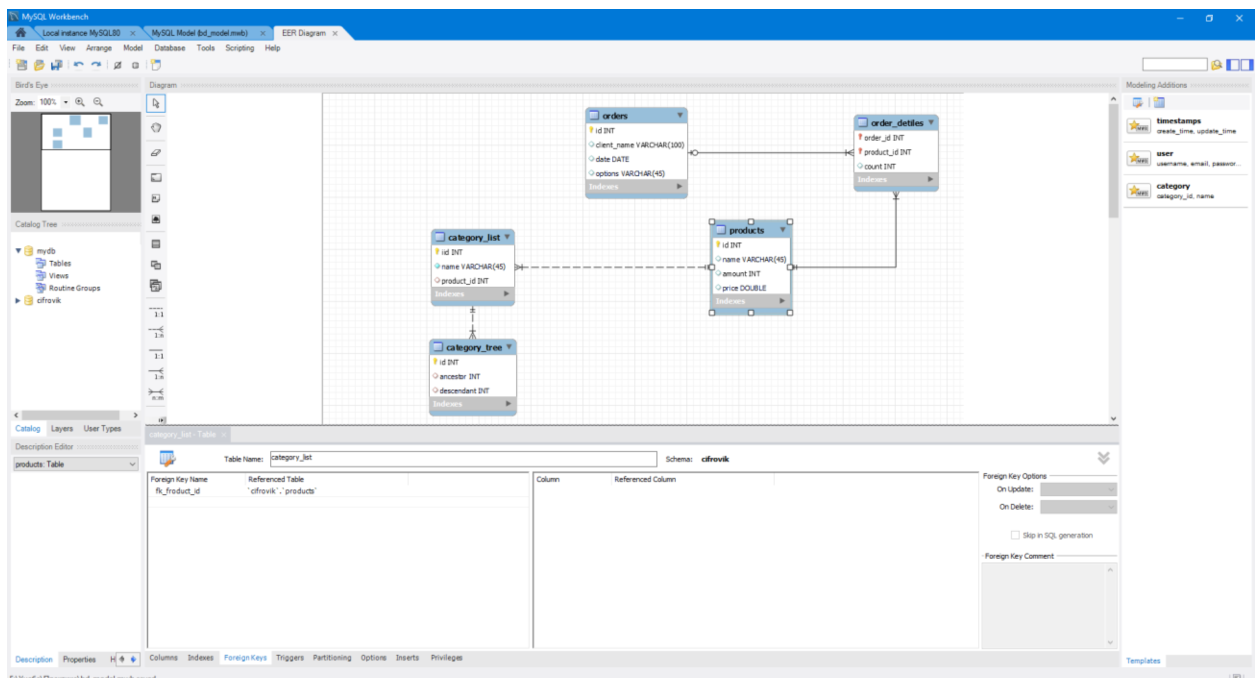

Рисунок 3.1 – Інтерфейс MySQL Workbench

На ERD діаграмі відображені сутності системи та зв'язок між ними (див. Рис 3.2).

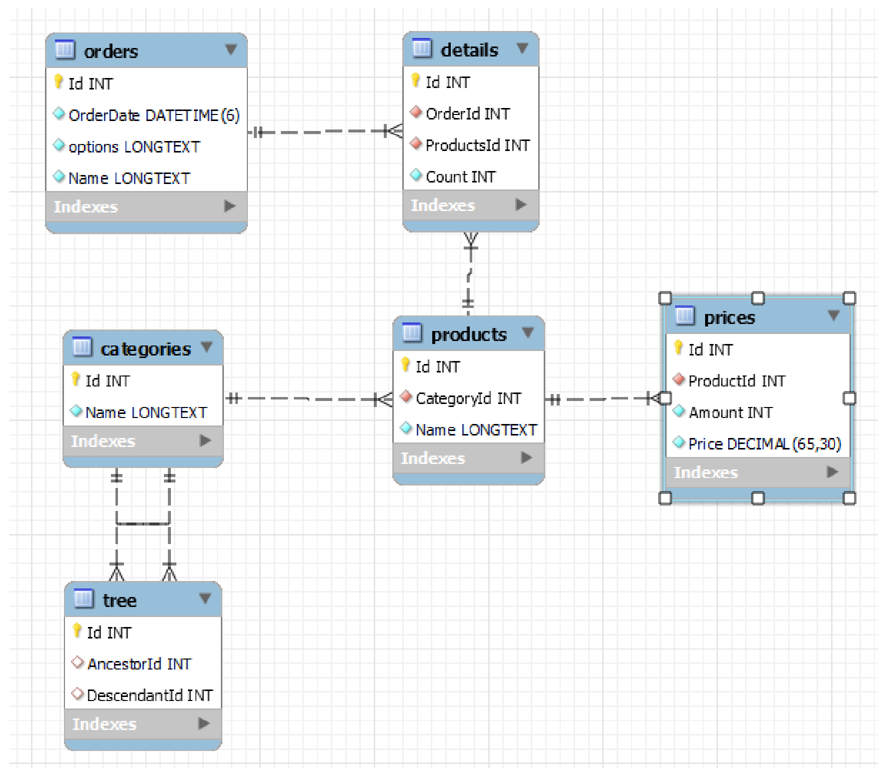

Рисунок 3.2 - ERD діаграма

Цієї діаграми достатньо для побудови структури бази даних. В процесі користувач має можливість згенерувати скрипт обравши потрібну йому базу даних (див. Рис 3.3– 3.4), проте в нашому випадку це не потрібно.

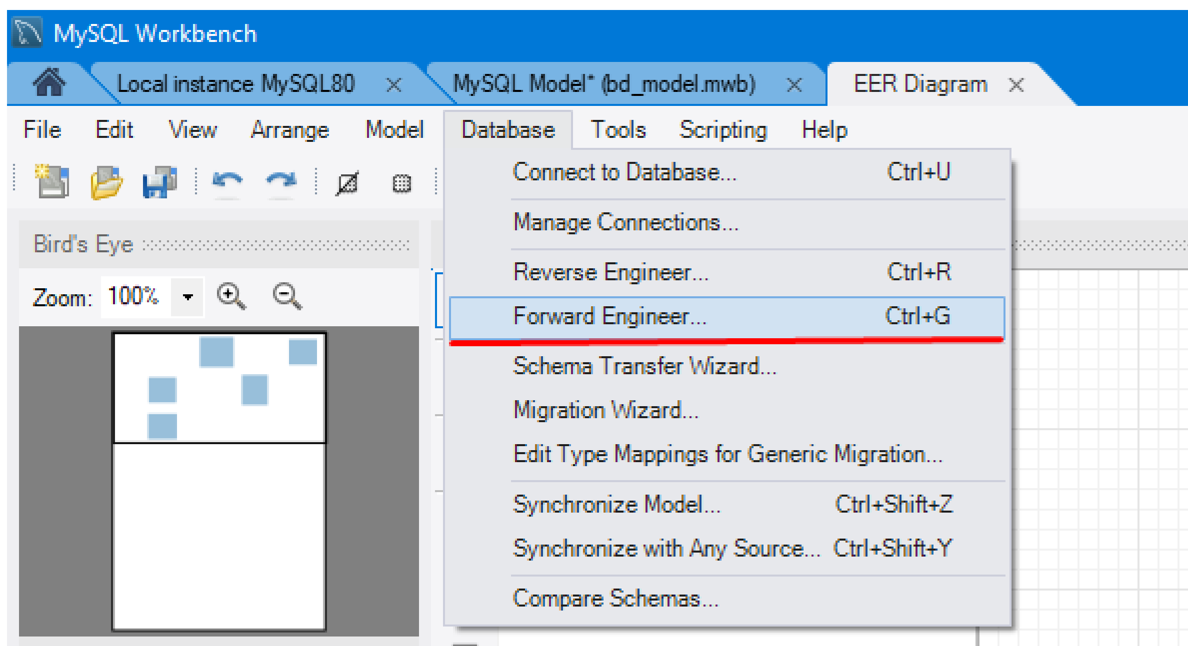

Рисунок 3.3 - Перехід до побудови бази

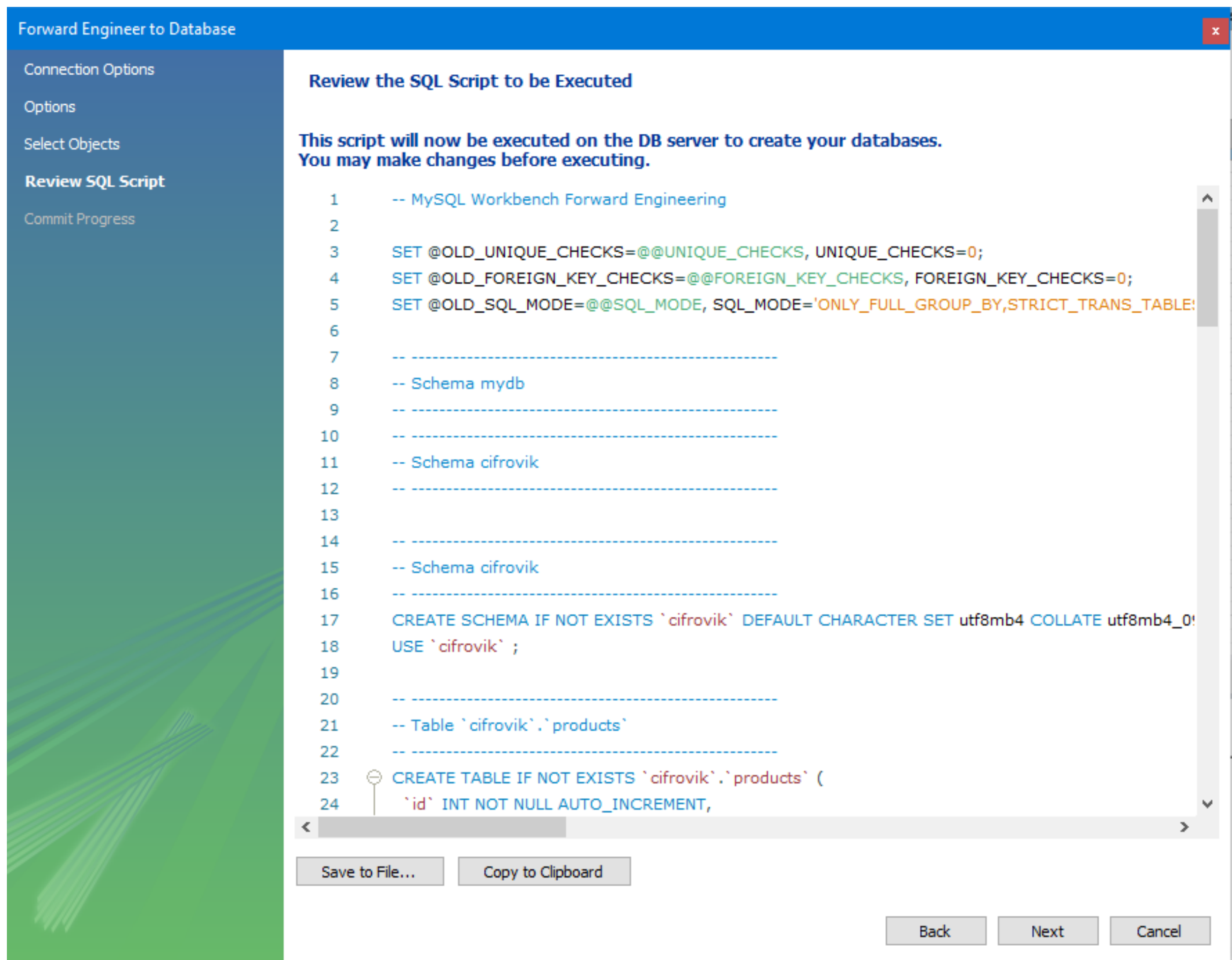

Рисунок 3.4 - Генерація скрипта у процесі побудови бази

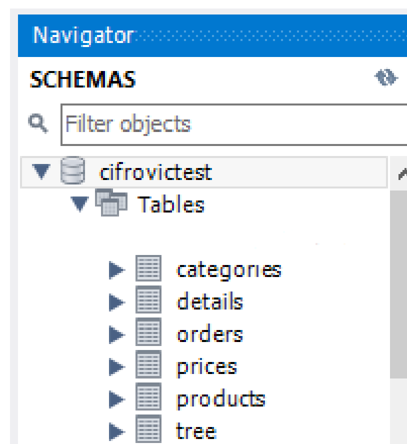

Рисунок 3.5 - Таблиці створеної схеми

3.2 Code First

Існує кілька підходів до розробки програмного забезпечення, що взаємодіє з базою даних, основними з яких є Data Base First і Code first. З назви очевидно, що при першому підході спочатку розробляється база даних і на її основі створюються класи-сутності в програмі, а при другому – спочатку створюються класи сутності, після чого виконується так звана «міграція», за допомогою бібліотек Entity Framework в Visual Studio.

ADO.NET Entity Framework – це об‘єктно-орієнтована технологія доступу до даних і надає можливість взаємодії з об‘єктами як за допомогою LINQ у вигляді LINQ to Entities, так і з використанням Entity SQL.

Language Integrated Query (LINQ) – бібліотека методів розширення для багатьох класів, що наслідують інтерфейс IEnumerable, а це колекції та списки. Ці розширення додають синтаксис, схожий на SQL.

ADO.NET Entity Framework і LINQ дуже спрощують роботу програміста, позбавляючи необхідності формувати складні рядки SQL запитів.

На етапі проектування даного проекту було обрано підхід Data Base First, але оновлення Visual Studio 2022 призвело до несумісності студії з MySQL для Visual Studio. Компанія Oracle не поспішає оновлювати деякі свої продукти. Без прямої взаємодії студії з MySQL завдання ускладнилось і було вирішено перейти до підходу Code First, а також додати до проекту пакет Pomelo.EntityFrameworkCore.MySql, котрий включає необхідні для роботи бібліотеки.

Відповідно до побудованої раніше ER-діаграми було створено бібліотеки Cifrovik.Interfaces та CifrovikDEL, склад яких представлено на рис. 3.6. Бібліотека Cifrovik.Interfaces містить інтерфейси IEntity – основа для сутностей, та IRepository<T> – основа репозиторію сутності. Бібліотека CifrovikDEL містить контекст бази даних, опис сутностей (таблиць бази даних), реалізацію репозиторіїв (у множині – деякі можуть мати власну реалізацію методів). Вихідний код інтерфейсів і класів описаних бібліотек буде розміщено у Додатку А. Також до бібліотеки CifrovikDEL включено сгенеровану, так звану,

міграцію - код для створення бази даних на сервері. Вихідний код міграції розміщено у Додатку Б.

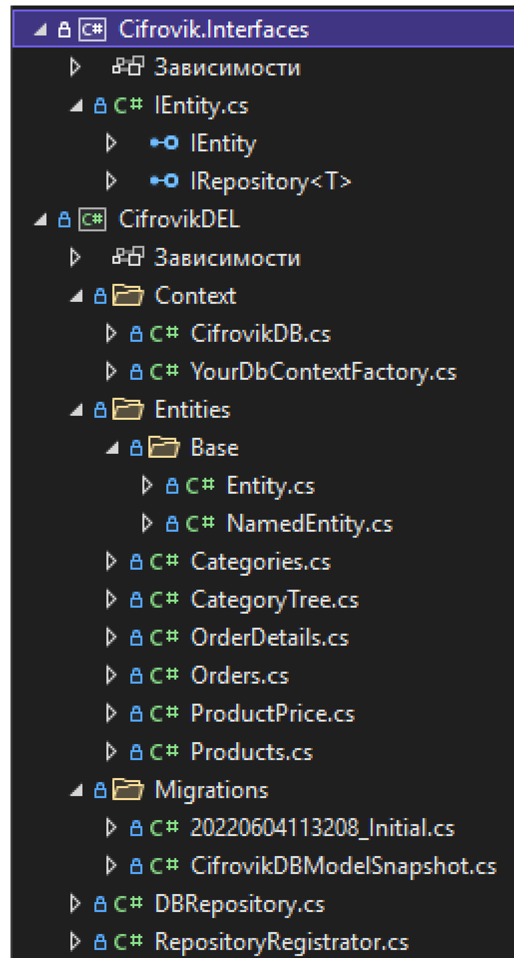

Рисунок 3.6 - Склад бібліотек Cifrovik.Interfaces та CifrovikDEL

Клас YourDbContextFactory.cs було створена для вирішення проблеми зі створенням міграції в режимі проектування (Design Time).

RepositoryRegistrator.cs – частина реалізації інвесії керування, важливого принципу об‘єктно-орієнтованого програмування, що використовується для зменшення зв‘язаності частин програми, що будуть викликатись із загальної «бібліотеки». Якщо об'єкт x (класу X) викликає методи об'єкту y (класу Y), то X залежить від Y. Залежність може бути звернена створенням третього класу, а саме інтерфейсного класу I, який повинен містити в собі усі методи, які x може викликати у об'єкта y. Крім того, Y повинен реалізовувати інтерфейс I. X та Y наразі обидва залежать від I, і клас X більш не залежить від класу Y;

передбачається, що X не реалізує I. Це виключення залежності класу X від Y шляхом створення інтерфейсу I і називається Inversion of Control. Слід сказати, що Y може залежати від інших класів. До внесення змін X залежав від Y, тоді X побічно залежав від усіх класів, від яких залежить Y. За допомогою застосування Inversion of Control всі побічні залежності були розірвані — не тільки залежність X від Y. [7]

Таким чином репозиторії сутностей будуть зареєстровані у списку сервісів і викликатись із нього за необхідності.

3.3 Інтерфейс

На рис. 3.7 зображено заплановане головне вікно програми. Оскільки процес розробки досить складний і довгий, то фактичний вигляд вікон на різних етапах розробки буде відрізнятись.

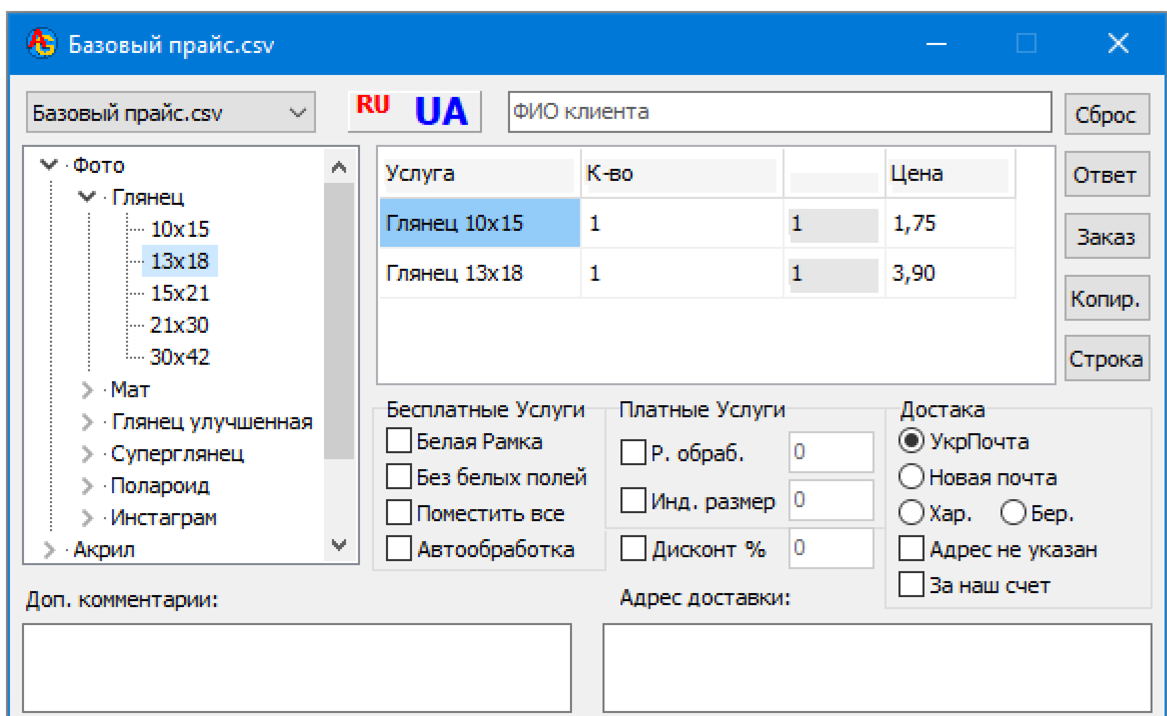

Рисунок 3.7 - Запланований вигляд головного вікна програми

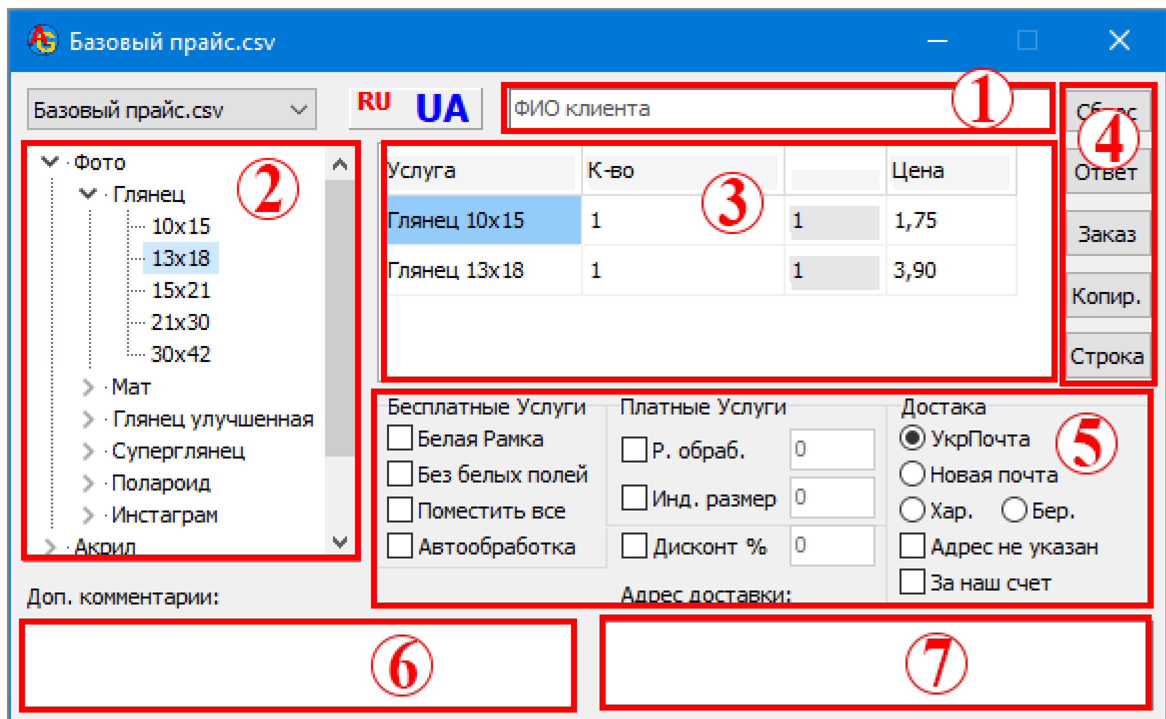

Рисунок 3.8 – Основні частини інтерфейсу головного вікна програми

На рис. 3.8 зображено основні частини інтерфейсу головного вікна програми:

1. Поле ідентифікації замовника, що міститиме ПІБ, електронну пошту, чи інший ідентифікатор;
2. Список надаваних фотолабораторією послуг, розділених по категоріям, на основі яких сформовано деревовидну структуру;
3. Таблиця доданих у замовлення послуг з підрахунком вартості, при чому враховується зміна ціни в залежності від кількості реалізацій кожної послуги;
4. Панель кнопок;
5. Панель вибору додаткових параметрів замовлення та способу одержання готового замовлення клієнтом;
6. Поле вводу коментаря до замовлення, наприклад необхідність включення непередбачених послуг чи підвищення пріоритету замовлення.
7. Поле вводу адреси доставки замовлення за необхідності.

В результаті роботи програми отримуємо відповідь для клієнта, розміщену у вікні відповіді (Рис. 3.9).

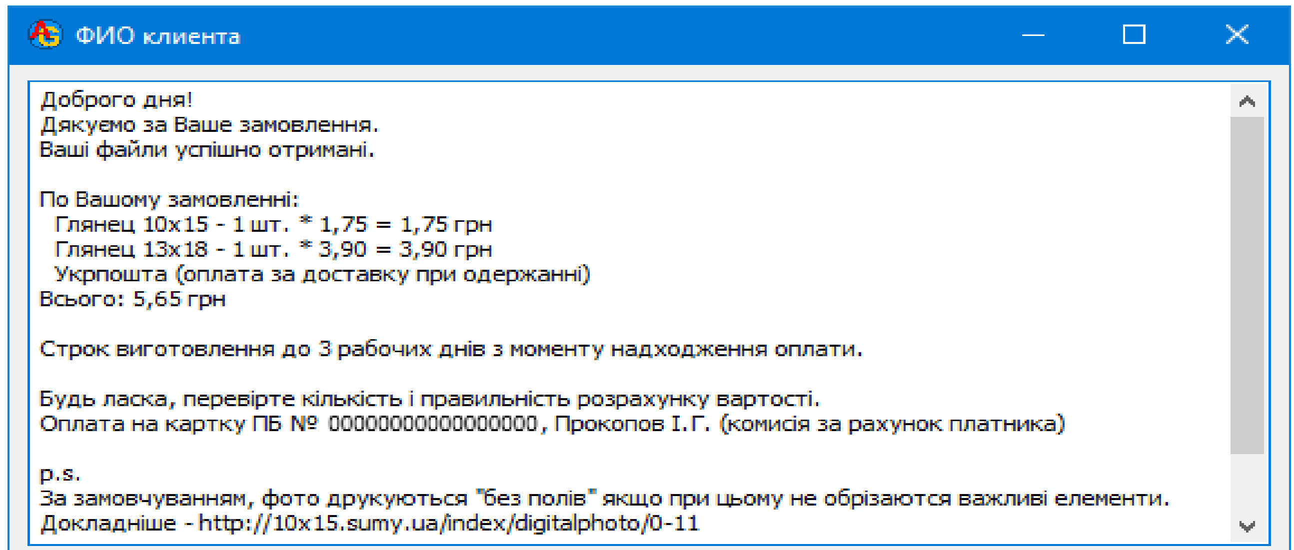

Рисунок 3.9 - Вікно відповіді клієнту

Вікно керування послугами (Рис. 3.10) дозволяє додавати, видаляти чи корегувати категорії та послуги в них:

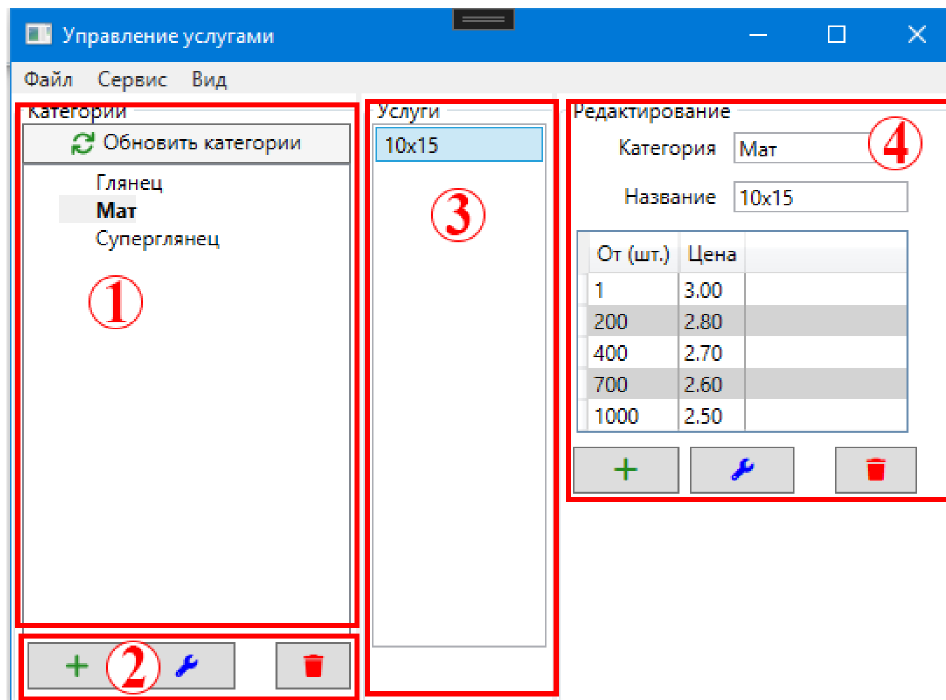

Рисунок 3.10 - Вікно керування послугами

1. Список категорій послуг (в кінцевому варіанті – дерево категорій)
2. Кнопки керування категоріями.
3. Список послуг в обраній категорії.
4. Багатофункціональна частина, що змінює свій вигляд в залежності від виконуваної функції (Рис. 3.11). 1 – відображення інформації про послугу, 2 – редагування чи додавання нової категорії, 3 – редагування існуючої чи додавання нової послуги.

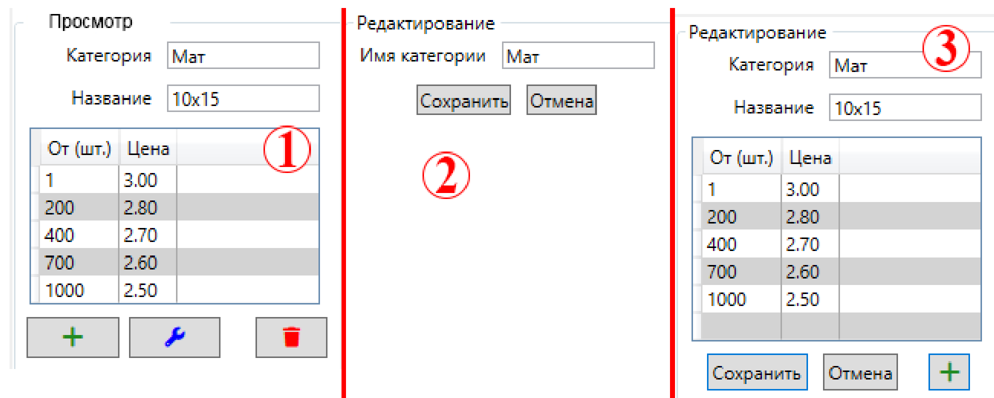

Рисунок 3.11 – Варіанти вигляду багатофункціональної частини вікна керування послугами

Для розробки інтерфейсу використовується Windows Presentation Foundation (WPF, кодова назва — Avalon) — графічна (презентаційна) підсистема (аналог WinForms), яка починаючи з .NET Framework 3.0 в складі цієї платформи. Має пряме відношення до XAML.

WPF є високорівневим об'єктно-орієнтованим функціональним шаром (англ. framework), що дозволяє створювати двовимірні та тривимірні інтерфейси.

XAML (Extensible Application Markup Language) є мовою розмітки, яку використовують для створення екземплярів об'єктів .NET. Хоча мова XAML — це технологія, що може бути застосовна до багатьох різних предметних областей, її головне призначення — конструювання інтерфейсів користувачів WPF. Інакше кажучи, документи XAML визначають розташування панелей, кнопок та інших елементів керування, що становлять вікна в застосунку WPF [8].

Саме даний підхід проектування інтерфейсу надає можливість перемикати частину вікна (4 на Рис. 3.10) між варіантами, зображеними на Рис. 3.11. Код прив‘язки складається з трьох частин.

По перше в XAML код вікна керування послугами включаються рядки

```
<GroupBox Grid.Column="2" Header="Редактирование">
                <ContentControl Content="{Binding CurrentModel}"/>
</GroupBox>
```

Це прив‘язка вмісту в залежності від змінної CurrentModel.

По друге у процесі виконання програми ця змінна міститиме посилання на необхідну View-Model

І третє – це прив‘язка вью-моделі і представлення в підключеному до проекту файлі-ресурсів:

```
<DataTemplate DataType="{x:Type vm:CategoryEditViewModel}">
        <view:CategoryEditView DataContext="{Binding}"/>
</DataTemplate>

<DataTemplate DataType="{x:Type vm:ProductEditViewModel}">
        <view:ProductEditView DataContext="{Binding}"/>
</DataTemplate>

<DataTemplate DataType="{x:Type vm:ProductDetailsViewModel}">
        <view:ProductDetailsView DataContext="{Binding}"/>
</DataTemplate>
```

Model-View-ViewModel — шаблон проектування, що застосовується під час проектування архітектури застосунків (додатків). MVVM орієнтований на такі сучасні платформи розробки, як Windows Presentation Foundation та Silverlight від компанії Microsoft.

MVVM полегшує відокремлення розробки графічного інтерфейсу від розробки бізнес логіки (бек-енд логіки), відомої як модель (можна також сказати, що це відокремлення представлення від моделі). Модель представлення є частиною, яка відповідає за перетворення даних для їх подальшої підтримки і використання. З цієї точки зору, модель представлення більше схожа на модель, ніж на представлення і обробляє більшість, якщо не всю, логіку відображення даних. Модель представлення може також реалізовувати патерн медіатор,

організовуючи доступ до бек-енд логіки навколо множини правил використання, які підтримуються представленням.

MVVM зручно використовувати замість класичного MVC та йому подібних у тих випадках, коли на платформі, де ведеться розробка, присутнє «зв'язування даних».

В MVC/MVP зміни у користувацькому інтерфейсі не впливають безпосередньо на модель, а йдуть через Контролер/Presenter. У таких технологіях, як WPF та Silverlight, присутня концепція «зв'язування даних», що дозволяє зв'язувати дані із візуальними елементами в обидві сторони.

Архітектура MVVM вирішує цю проблему ясним поділом відповідальності:

- Розробка користувацького інтерфейсу здійснюється дизайнером інтерфейсів за допомогою технології, більш-менш природної для такої роботи (XML)
- Логіка користувацького інтерфейсу реалізується розробником як компонент ViewModel
- Функціональні зв'язки між користувацьким інтерфейсом та ViewModel реалізуються через біндинги (bindings), які, по суті, є правилами типу «якщо кнопка А була натиснута, повинен бути викликаний метод onButtonAClick() з ViewModel». Біндинги можуть бути написані в коді або визначені декларативним шляхом.

Архітектура MVVM використовується в тому чи іншому вигляді усіма сучасними технологіями, наприклад Microsoft WPF і Silverlight, Oracle JavaFX, Adobe Flex, AJAX [9].

3.4 Unit Of Work

Unit Of Work - патерн об'єктно-реляційної поведінки, мета якого полягає у відстежуванні зміни об'єктів під час транзакції.

Під час роботи із базою даних важливо відстежувати зміни в об'єктах, в іншому випадку дані не будуть оновлені. Це також вірно для операцій додавання та видалення.

Можна змінювати дані в сховищі при кожній взаємодії з об'єктом, але це призведе до багатьох викликів у базу даних. Очевидно це потребує підтримування транзакції відкритою, що впливає на продуктивність роботи.

Даний шаблон пропонує відстежувати всі зміни над об'єктами та вносити їх у вигляді єдиної транзакції [10].

В даному проекті для керування послугами створено клас ProductsManager.cs, що виконує роль Unit Of Work. Завантажує дані з БД, забезпечує доступ до них іншим класам та зберігає результати роботи в БД. Вихідний код ProductsManager.cs розміщено у Додатку В.

ВИСНОВКИ

У кваліфікаційній роботі проаналізовано роботу фотолабораторії та узгоджено основні вимоги замовника (керівника фотолабораторії), проаналізовані інструменти і умови для розробки бази даних та програмного додатку для вирішення поставленої задачі. Згідно вимог розроблено базу даних, спроектовано інтерфейс та реалізовано систему керування послугами (внесення змін у базу даних).

При цьому, під час виконання кваліфікаційної роботи були виконані наступні завдання:

1) Розроблено інформаційну модель системи керування замовленнями;
2) Визначено та оптимізовано структуру баз даних системи керування замовленнями;
3) Розроблено інтерфейс користувача системою керування замовленнями;
4) Розроблено бізнес логіку системи;
5) Обрано засоби для програмної реалізації системи;
6) Виконано програмну реалізацію та тестування системи.

Практична реалізація програмного продукту виконана на мові C# у середовищі розробки Visual Studio. Повний вихідний код проекту доступний в репозиторії за посиланням https://github.com/Hedeus/Bachelor_Net60

СПИСОК ЛІТЕРАТУРИ

1. Фотолабораторія. – https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%82%D0%BE%D0%BB%D0%B0%D0%B1%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8%D1%8F.
2. Finance-Credit.News. – https://finance-credit.news/ekonomika-logistika/tema-upravlenie-zakazami-69134.html.
3. IBM. Керування замовленнями. – https://www.ibm.com/ru-ru/topics/order-management.
4. Вікіпедія. Unified Modeling Language. – https://uk.wikipedia.org/wiki/Unified_Modeling_Language.
5. Вікіпедія. Діаграма прецедентів. – https://uk.wikipedia.org/wiki/%D0%94%D1%96%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%B0_%D0%BF%D1%80%D0%B5%D1%86%D0%B5%D0%B4%D0%B5%D0%BD%D1%82%D1%96%D0%B2.
6. Вікіпедія. UMLet. – https://en.wikipedia.org/wiki/UMLet.
7. Вікіпедія. Інверсія управління. – https://uk.wikipedia.org/wiki/%D0%86%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D1%96%D1%8F_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%96%D0%BD%D0%BD%D1%8F.
8. Вікіпедія. WPF. – https://uk.wikipedia.org/wiki/Windows_Presentation_Foundation.
9. Вікіпедія. MVVM. – https://uk.wikipedia.org/wiki/Model-View-ViewModel.
10. Вікіпедія. Unit Of Work. – https://uk.wikipedia.org/wiki/Unit_Of_Work.
11. Arnaud, Weil, «Learn WPF MVVM - XAML, C# and the MVVM pattern», 2016.
12. Driscoll B., «Entity Framework 6 Recipes», 2013.
13. Alan, Beaulie, «Learning SQL: Master SQL Fundamentals, 3rd Edition», 2021.

14. Dirk., Strauss, «Started with Visual Studio 2019: Learning and Implementing New Features», 2019.

15. Taurius Litvinavicius, «Exploring Windows Presentation Foundation: With Practical Applications in .NET 5», 2021.

ДОДАТКИ

Додаток А

Бібліотека Cifrovik.Interfaces:

Файл IEntity.cs

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

namespace Cifrovik.Interfaces
{
    public interface IEntity
    {
        int Id { get; set; }
    }

    public interface IRepository<T> where T : class, IEntity, new()
    {
        bool AutosaveChanges { get; set; }
        IQueryable<T> Items { get; }
        T Get(int id);
        Task<T> GetAsync(int id, CancellationToken Cancel = default);
        T Add(T item);
        Task<T> AddAsync(T item, CancellationToken Cancel = default);
        void Update(T item);
        Task UpdateAsync(T item, CancellationToken Cancel = default);
        void Remove(int id);
        Task RemoveAsync(int id, CancellationToken Cancel = default);
        void AutosaveOnOff(bool autosave);
        void Save();
        Task SaveAsync(CancellationToken Cancel = default);
    }
}
```

Бібліотека CifrovikDEL:

Файл CifrovikDB.cs

```
using CifrovikDEL.Entities;
using Microsoft.EntityFrameworkCore;

namespace CifrovikDEL.Context
{
    public class CifrovikDB : DbContext
    {
        public DbSet<Categories> Categories { get; set; }
        public DbSet<Products> Products { get; set; }
        public DbSet<Orders> Orders { get; set; }
        public DbSet<ProductPrice> Prices { get; set; }
        public DbSet<OrderDetails> Details { get; set; }
        public DbSet<CategoryTree> Tree { get; set; }
        public CifrovikDB(DbContextOptions<CifrovikDB> options) : base(options) { }

        // Обмеження унікальності
        protected override void OnModelCreating(ModelBuilder modelBuilder)
```

```
        {
            modelBuilder.Entity<CategoryTree>().HasIndex(i =>
i.DescendantId).IsUnique();
            modelBuilder.Entity<ProductPrice>().HasIndex(i => new { i.ProductId,
i.Amount, i.Price }).IsUnique()
                                                .HasName("IX_UnProdAmoPrice");
        }
    }
}
```

Файл YourDbContextFactory.cs

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;

namespace CifrovikDEL.Context
{
    public class YourDbContextFactory : IDesignTimeDbContextFactory<CifrovikDB>
    {
        public CifrovikDB CreateDbContext(string[] args)
        {
            var optionsBuilder = new DbContextOptionsBuilder<CifrovikDB>();

optionsBuilder.UseMySql("server=localhost;uid=root;pwd=1h9e8d7;database=cifrovictest;"
, new MySqlServerVersion(new Version(8, 0, 27)));

            return new CifrovikDB(optionsBuilder.Options);
        }
    }
}
```

Файл Entity.cs

```
using Cifrovik.Interfaces;

namespace CifrovikDEL.Entities.Base
{
    public abstract class Entity : IEntity
    {
        public int Id { get; set; }
    }
}
```

Файл NamedEntity.cs

```
using System.ComponentModel.DataAnnotations;

namespace CifrovikDEL.Entities.Base
{
    public abstract class NamedEntity : Entity
    {
        [Required]
        public string Name { get; set; }
    }
}
```

Файл Categories.cs

```
using CifrovikDEL.Entities.Base;

namespace CifrovikDEL.Entities
```

```
{
    public class Categories : NamedEntity
    {
        public override string ToString() => $"{Name}";
    }
}
```

Файл CategoryTree.cs

```
using CifrovikDEL.Entities.Base;

namespace CifrovikDEL.Entities
{
    public class CategoryTree : Entity
    {
        virtual public Categories Ancestor { get; set; }
        public int? AncestorId { get; set; }
        virtual public Categories Descendant { get; set; }
        public int? DescendantId { get; set; }

        public override string ToString() => $"Ancestor {AncestorId}, Descendant
{DescendantId}";
    }
}
```

Файл OrderDetails.cs

```
using CifrovikDEL.Entities.Base;

namespace CifrovikDEL.Entities
{
    public class OrderDetails : Entity
    {
        public virtual Orders Order { get; set; }
        public virtual Products Products { get; set; }
        public int Count { get; set; }
    }
}
```

Файл Orders.cs

```
using CifrovikDEL.Entities.Base;

namespace CifrovikDEL.Entities
{
    public class Orders : NamedEntity
    {
        public DateTime OrderDate { get; set; }
        public string options { get; set; }
    }
}
```

Файл ProductPrice.cs

```
using CifrovikDEL.Entities.Base;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace CifrovikDEL.Entities
{
```

```
    public class ProductPrice : Entity
    {
        [Required]
        public virtual Products Product { get; set; }
        public int ProductId { get; set; }
        public int Amount { get; set; }
        [Column(TypeName = "decimal(18,2")]
        public decimal Price { get; set; }
    }
}
```

Файл Products.cs

```
using CifrovikDEL.Entities.Base;

namespace CifrovikDEL.Entities
{
    public class Products : NamedEntity
    {
        public virtual Categories Category { get; set; }
        public int CategoryId { get; set; }
        public override string ToString() => $"{Name}";
    }
}
```

Файл DBRepository.cs

```
using Cifrovik.Interfaces;
using CifrovikDEL.Context;
using CifrovikDEL.Entities;
using CifrovikDEL.Entities.Base;
using Microsoft.EntityFrameworkCore;

namespace CifrovikDEL
{
    internal class DBRepository<T> : IRepository<T> where T : Entity, new()
    {
        private readonly CifrovikDB _db;
        private readonly DbSet<T> _Set;

        public bool AutosaveChanges { get; set; }

        public DBRepository(CifrovikDB db)
        {
            _db = db;
            _Set = db.Set<T>();
        }
        public virtual IQueryable<T> Items => _Set;

        public T Get(int id) => Items.SingleOrDefault(item => item.Id == id);

        public async Task<T> GetAsync(int id, CancellationToken Cancel = default) =>
await Items
            .SingleOrDefaultAsync(item => item.Id == id, Cancel)
            .ConfigureAwait(false);

        public T Add(T item)
        {
            if (item is null) throw new ArgumentNullException(nameof(item));
            _db.Entry(item).State = EntityState.Added;
            if (AutosaveChanges)
                _db.SaveChanges();
```

```
        return item;
    }

    public async Task<T> AddAsync(T item, CancellationToken Cancel = default)
    {
        if (item is null) throw new ArgumentNullException(nameof(item));
        _db.Entry(item).State = EntityState.Added;
        if (AutosaveChanges)
            await _db.SaveChangesAsync(Cancel).ConfigureAwait(false);
        return item;
    }

    public void Update(T item)
    {
        if (item is null) throw new ArgumentNullException(nameof(item));
        _db.Entry(item).State = EntityState.Modified;
        if (AutosaveChanges)
            _db.SaveChanges();
    }

    public async Task UpdateAsync(T item, CancellationToken Cancel = default)
    {
        if (item is null) throw new ArgumentNullException(nameof(item));
        _db.Entry(item).State = EntityState.Modified;
        if (AutosaveChanges)
            await _db.SaveChangesAsync(Cancel).ConfigureAwait(false);
    }

    public void Remove(int id)
    {
        var item = Get(id);
        if (item is null) return;
        _db.Entry(item).State = EntityState.Deleted;

        //_db.Remove(new T { Id = id });
        if (AutosaveChanges)
            _db.SaveChanges();
    }

    public async Task RemoveAsync(int id, CancellationToken Cancel = default)
    {
        _db.Remove(new T { Id = id });
        if (AutosaveChanges)
            await _db.SaveChangesAsync(Cancel).ConfigureAwait(false);
    }

    public void AutosaveOnOff(bool autosave)
    {
        AutosaveChanges = autosave;
    }

    public void Save()
    {
        if (!AutosaveChanges)
            _db.SaveChanges();
    }
    public async Task SaveAsync(CancellationToken Cancel = default)
    {
        if (!AutosaveChanges)
            await _db.SaveChangesAsync(Cancel).ConfigureAwait(false);
    }
}
```

```
    class ProductRepository : DBRepository<Products>
    {
        public override IQueryable<Products> Items => base.Items.Include(item => item.Category);
        public ProductRepository(CifrovikDB db) : base(db) { }
    }

    class PriceRepository : DBRepository<ProductPrice>
    {
        public override IQueryable<ProductPrice> Items => base.Items.Include(item => item.Product);
        public PriceRepository(CifrovikDB db) : base(db) { }
    }
}
```

Файл RepositoryRegistrator.cs

```
using Cifrovik.Interfaces;
using CifrovikDEL.Entities;
using Microsoft.Extensions.DependencyInjection;

namespace CifrovikDEL
{
    public static class RepositoryRegistrator
    {
        public static IServiceCollection AddRepositoriesInDB(this IServiceCollection services) => services
            .AddTransient<IRepository<Categories>, DBRepository<Categories>>()
            .AddTransient<IRepository<CategoryTree>, DBRepository<CategoryTree>>()
            .AddTransient<IRepository<OrderDetails>, DBRepository<OrderDetails>>()
            .AddTransient<IRepository<Orders>, DBRepository<Orders>>()
            .AddTransient<IRepository<ProductPrice>, PriceRepository>()
            .AddTransient<IRepository<Products>, ProductRepository>()
            ;
    }
}
```

Додаток Б

Migrations

Файл 20220604113208_Initial.cs

```
using System;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace CifrovikDEL.Migrations
{
    public partial class Initial : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AlterDatabase()
                .Annotation("MySql:CharSet", "utf8mb4");

            migrationBuilder.CreateTable(
                name: "Categories",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("MySql:ValueGenerationStrategy", MySqlValueGenerationStrategy.IdentityColumn),
                    Name = table.Column<string>(type: "longtext", nullable: false)
                        .Annotation("MySql:CharSet", "utf8mb4")
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Categories", x => x.Id);
                })
                .Annotation("MySql:CharSet", "utf8mb4");

            migrationBuilder.CreateTable(
                name: "Orders",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("MySql:ValueGenerationStrategy", MySqlValueGenerationStrategy.IdentityColumn),
                    OrderDate = table.Column<DateTime>(type: "datetime(6)", nullable: false),
                    options = table.Column<string>(type: "longtext", nullable: false)
                        .Annotation("MySql:CharSet", "utf8mb4"),
                    Name = table.Column<string>(type: "longtext", nullable: false)
                        .Annotation("MySql:CharSet", "utf8mb4")
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Orders", x => x.Id);
                })
                .Annotation("MySql:CharSet", "utf8mb4");

            migrationBuilder.CreateTable(
                name: "Products",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("MySql:ValueGenerationStrategy", MySqlValueGenerationStrategy.IdentityColumn),
                    CategoryId = table.Column<int>(type: "int", nullable: false),
```

```
        Name = table.Column<string>(type: "longtext", nullable: false)
            .Annotation("MySql:CharSet", "utf8mb4")
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Products", x => x.Id);
        table.ForeignKey(
            name: "FK_Products_Categories_CategoryId",
            column: x => x.CategoryId,
            principalTable: "Categories",
            principalColumn: "Id",
            onDelete: ReferentialAction.Cascade);
    })
    .Annotation("MySql:CharSet", "utf8mb4");

migrationBuilder.CreateTable(
    name: "Tree",
    columns: table => new
    {
        Id = table.Column<int>(type: "int", nullable: false)
            .Annotation("MySql:ValueGenerationStrategy", MySqlValueGenerationStrategy.IdentityColumn),
        AncestorId = table.Column<int>(type: "int", nullable: true),
        DescendantId = table.Column<int>(type: "int", nullable: true)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Tree", x => x.Id);
        table.ForeignKey(
            name: "FK_Tree_Categories_AncestorId",
            column: x => x.AncestorId,
            principalTable: "Categories",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_Tree_Categories_DescendantId",
            column: x => x.DescendantId,
            principalTable: "Categories",
            principalColumn: "Id");
    })
    .Annotation("MySql:CharSet", "utf8mb4");

migrationBuilder.CreateTable(
    name: "Details",
    columns: table => new
    {
        Id = table.Column<int>(type: "int", nullable: false)
            .Annotation("MySql:ValueGenerationStrategy", MySqlValueGenerationStrategy.IdentityColumn),
        OrderId = table.Column<int>(type: "int", nullable: false),
        ProductsId = table.Column<int>(type: "int", nullable: false),
        Count = table.Column<int>(type: "int", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Details", x => x.Id);
        table.ForeignKey(
            name: "FK_Details_Orders_OrderId",
            column: x => x.OrderId,
            principalTable: "Orders",
            principalColumn: "Id",
            onDelete: ReferentialAction.Cascade);
        table.ForeignKey(
            name: "FK_Details_Products_ProductsId",
            column: x => x.ProductsId,
```

```
                principalTable: "Products",
                principalColumn: "Id",
                onDelete: ReferentialAction.Cascade);
        })
        .Annotation("MySql:CharSet", "utf8mb4");

    migrationBuilder.CreateTable(
        name: "Prices",
        columns: table => new
        {
            Id = table.Column<int>(type: "int", nullable: false)
                .Annotation("MySql:ValueGenerationStrategy", MySqlValueGenerationStrategy.IdentityColumn),
            ProductId = table.Column<int>(type: "int", nullable: false),
            Amount = table.Column<int>(type: "int", nullable: false),
            Price = table.Column<decimal>(type: "decimal(65,30)", nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Prices", x => x.Id);
            table.ForeignKey(
                name: "FK_Prices_Products_ProductId",
                column: x => x.ProductId,
                principalTable: "Products",
                principalColumn: "Id",
                onDelete: ReferentialAction.Cascade);
        })
        .Annotation("MySql:CharSet", "utf8mb4");

    migrationBuilder.CreateIndex(
        name: "IX_Details_OrderId",
        table: "Details",
        column: "OrderId");

    migrationBuilder.CreateIndex(
        name: "IX_Details_ProductsId",
        table: "Details",
        column: "ProductsId");

    migrationBuilder.CreateIndex(
        name: "IX_UnProdAmoPrice",
        table: "Prices",
        columns: new[] { "ProductId", "Amount", "Price" },
        unique: true);

    migrationBuilder.CreateIndex(
        name: "IX_Products_CategoryId",
        table: "Products",
        column: "CategoryId");

    migrationBuilder.CreateIndex(
        name: "IX_Tree_AncestorId",
        table: "Tree",
        column: "AncestorId");

    migrationBuilder.CreateIndex(
        name: "IX_Tree_DescendantId",
        table: "Tree",
        column: "DescendantId",
        unique: true);
}

protected override void Down(MigrationBuilder migrationBuilder)
```

```
        {
            migrationBuilder.DropTable(
                name: "Details");

            migrationBuilder.DropTable(
                name: "Prices");

            migrationBuilder.DropTable(
                name: "Tree");

            migrationBuilder.DropTable(
                name: "Orders");

            migrationBuilder.DropTable(
                name: "Products");

            migrationBuilder.DropTable(
                name: "Categories");
        }
    }
}
```

Файл CifrovikDBModelSnapshot.cs

```
// <auto-generated />
using System;
using CifrovikDEL.Context;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;

#nullable disable

namespace CifrovikDEL.Migrations
{
    [DbContext(typeof(CifrovikDB))]
    partial class CifrovikDBModelSnapshot : ModelSnapshot
    {
        protected override void BuildModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("ProductVersion", "6.0.5")
                .HasAnnotation("Relational:MaxIdentifierLength", 64);

            modelBuilder.Entity("CifrovikDEL.Entities.Categories", b =>
                {
                    b.Property<int>("Id")
                        .ValueGeneratedOnAdd()
                        .HasColumnType("int");

                    b.Property<string>("Name")
                        .IsRequired()
                        .HasColumnType("longtext");

                    b.HasKey("Id");

                    b.ToTable("Categories");
                });

            modelBuilder.Entity("CifrovikDEL.Entities.CategoryTree", b =>
```

```
    {
        b.Property<int>("Id")
            .ValueGeneratedOnAdd()
            .HasColumnType("int");

        b.Property<int?>("AncestorId")
            .HasColumnType("int");

        b.Property<int?>("DescendantId")
            .HasColumnType("int");

        b.HasKey("Id");

        b.HasIndex("AncestorId");

        b.HasIndex("DescendantId")
            .IsUnique();

        b.ToTable("Tree");
    });

modelBuilder.Entity("CifrovikDEL.Entities.OrderDetails", b =>
    {
        b.Property<int>("Id")
            .ValueGeneratedOnAdd()
            .HasColumnType("int");

        b.Property<int>("Count")
            .HasColumnType("int");

        b.Property<int>("OrderId")
            .HasColumnType("int");

        b.Property<int>("ProductsId")
            .HasColumnType("int");

        b.HasKey("Id");

        b.HasIndex("OrderId");

        b.HasIndex("ProductsId");

        b.ToTable("Details");
    });

modelBuilder.Entity("CifrovikDEL.Entities.Orders", b =>
    {
        b.Property<int>("Id")
            .ValueGeneratedOnAdd()
            .HasColumnType("int");

        b.Property<string>("Name")
            .IsRequired()
            .HasColumnType("longtext");

        b.Property<DateTime>("OrderDate")
            .HasColumnType("datetime(6)");

        b.Property<string>("options")
            .IsRequired()
            .HasColumnType("longtext");
```

```
            b.HasKey("Id");

            b.ToTable("Orders");
        });

    modelBuilder.Entity("CifrovikDEL.Entities.ProductPrice", b =>
        {
            b.Property<int>("Id")
                .ValueGeneratedOnAdd()
                .HasColumnType("int");

            b.Property<int>("Amount")
                .HasColumnType("int");

            b.Property<decimal>("Price")
                .HasColumnType("decimal(65,30)");

            b.Property<int>("ProductId")
                .HasColumnType("int");

            b.HasKey("Id");

            b.HasIndex("ProductId", "Amount", "Price")
                .IsUnique()
                .HasDatabaseName("IX_UnProdAmoPrice");

            b.ToTable("Prices");
        });

    modelBuilder.Entity("CifrovikDEL.Entities.Products", b =>
        {
            b.Property<int>("Id")
                .ValueGeneratedOnAdd()
                .HasColumnType("int");

            b.Property<int>("CategoryId")
                .HasColumnType("int");

            b.Property<string>("Name")
                .IsRequired()
                .HasColumnType("longtext");

            b.HasKey("Id");

            b.HasIndex("CategoryId");

            b.ToTable("Products");
        });

    modelBuilder.Entity("CifrovikDEL.Entities.CategoryTree", b =>
        {
            b.HasOne("CifrovikDEL.Entities.Categories", "Ancestor")
                .WithMany()
                .HasForeignKey("AncestorId");

            b.HasOne("CifrovikDEL.Entities.Categories", "Descendant")
                .WithMany()
                .HasForeignKey("DescendantId");

            b.Navigation("Ancestor");

            b.Navigation("Descendant");
```

```
            });

        modelBuilder.Entity("CifrovikDEL.Entities.OrderDetails", b =>
            {
                b.HasOne("CifrovikDEL.Entities.Orders", "Order")
                    .WithMany()
                    .HasForeignKey("OrderId")
                    .OnDelete(DeleteBehavior.Cascade)
                    .IsRequired();

                b.HasOne("CifrovikDEL.Entities.Products", "Products")
                    .WithMany()
                    .HasForeignKey("ProductsId")
                    .OnDelete(DeleteBehavior.Cascade)
                    .IsRequired();

                b.Navigation("Order");

                b.Navigation("Products");
            });

        modelBuilder.Entity("CifrovikDEL.Entities.ProductPrice", b =>
            {
                b.HasOne("CifrovikDEL.Entities.Products", "Product")
                    .WithMany()
                    .HasForeignKey("ProductId")
                    .OnDelete(DeleteBehavior.Cascade)
                    .IsRequired();

                b.Navigation("Product");
            });

        modelBuilder.Entity("CifrovikDEL.Entities.Products", b =>
            {
                b.HasOne("CifrovikDEL.Entities.Categories", "Category")
                    .WithMany()
                    .HasForeignKey("CategoryId")
                    .OnDelete(DeleteBehavior.Cascade)
                    .IsRequired();

                b.Navigation("Category");
            });
#pragma warning restore 612, 618
        }
    }
}
```

Додаток В

Файл ProductsManager.cs

```
using Bachelor_Net60.ViewModels.Base;
using Cifrovik.Interfaces;
using CifrovikDEL.Entities;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;

namespace Bachelor_Net60.Services.Management
{
```

```
class ProductsManager : ViewModel
{
    private readonly IRepository<Products> _Products;
    private readonly IRepository<Categories> _Categories;
    private readonly IRepository<ProductPrice> _ProductPrice;
    private readonly IRepository<CategoryTree> _CatsTree;

    public IEnumerable<Products> Prods => _Products.Items.ToList();
    public IEnumerable<Categories> Cats => _Categories.Items.ToList();
    public IEnumerable<ProductPrice> Prices => _ProductPrice.Items.ToList();
    public IEnumerable<CategoryTree> Tree => _CatsTree.Items.ToList();

    #region Общие свойства
    private Categories _SelectedCategory;
    public Categories SelectedCategory { get => _SelectedCategory; set => Set(ref _SelectedCategory, val

    public Products _SelectedProduct;
    public Products SelectedProduct { get => _SelectedProduct; set => Set(ref _SelectedProduct, value); 

    private ViewModel _CurrentModel;
    public ViewModel CurrentModel { get => _CurrentModel; set => Set(ref _CurrentModel, value); }
    #endregion


    public ProductsManager(IRepository<Categories> categories,
                    IRepository<Products> products,
                    IRepository<ProductPrice> productPrice,
                    IRepository<CategoryTree> catsTree)
    {
        _Products = products;
        _Categories = categories;
        _ProductPrice = productPrice;
        _CatsTree = catsTree;
    }

    /*-------------------------------Методы--------------------------------*/
    #region Update методы - Обновление существующих в базе записей
    public void ProductsUpdate(Products product)
    {
        _Products.Update(product);
        _Products.Save();
    }

    public void CategoriesUpdate(Categories category)
    {
        _Categories.Update(category);
        _Categories.Save();
    }

    public void ProductPriceUpdate(ObservableCollection<ProductPrice> productPrice)
    {
        //_ProductPrice.Update(productPrice);
        int producctIdToRemove = productPrice.First().ProductId;
        ProductPriceRemove(producctIdToRemove);
        ProductPriceAdd(productPrice);
        _ProductPrice.Save();
    }

    public void CategoryTreeUpdate(CategoryTree categorisTree) => _CatsTree.Update(categorisTree);
    #endregion

    #region Add методы - добавление новых записей в базу
```

```
        public Products ProductsAdd(Products product)
        {
            var p = _Products.Add(product);
            _Products.Save();
            return p;
        }

        public void CategoriesAdd(Categories category)
        {
            _Categories.Add(category);
            _Categories.Save();
        }

        public void ProductPriceAdd(ObservableCollection<ProductPrice> productPrice)
        {
            //_ProductPrice.AutosaveChanges = false;
            foreach (var price in productPrice)
                _ProductPrice.Add(price);
            _ProductPrice.Save();
        }

        public void CategoryTreeAdd(CategoryTree categorisTree) => _CatsTree.Add(categorisTree);
        public async void CategoryTreeAddAsync(CategoryTree categorisTree) => await _CatsTree.AddAsync(categorisTree);
        #endregion

        #region Remove методы - удаление новых записей в базу
        public void ProductsRemove(Products product)
        {
            ProductPriceRemove(product.Id);
            _Products.Remove(product.Id);
            _Products.Save();
        }

        public void CategoriesRemove(int Id)
        {
            _Categories.Remove(Id);
            _Categories.Save();
        }

        public void ProductPriceRemove(int productId)
        {
            IEnumerable<ProductPrice> pricesToRemove = (Prices.Where(price => price.ProductId == productId));
            foreach (var price in pricesToRemove)
                _ProductPrice.Remove(price.Id);
            _ProductPrice.Save();
        }

        public void CategoryTreeRemove(CategoryTree categorisTree) => _CatsTree.Remove(categorisTree.Id);
        #endregion

        #region Autosave
        public void ProductsAutosave(bool autosave) => _Products.AutosaveChanges = autosave;


        #endregion

        #region Save
        public void ProductsSave() => _Products.Save();
        #endregion

    }
}
```