Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська дипломна робота

на тему: «Розробка програмного забезпечення для кас самообслуговування з використанням фреймворків .Net і Angular»

Виконав: студент _IV___ курсу
групи ____2ПІ-18б_____
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Рибак А.Ю.
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. ПЗ Романюк О.В.
(прізвище та ініціали)
Рецензент:. к.т.н., доц. каф КН Колодний В.В.
(прізвище та ініціали)

Допущено до захисту
Зав. кафедри _____ Романюк О.Н.
«____» _____ 2022 р.

ВНТУ – 2022

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший бакалаврський
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
д.т.н., проф. Романюк О. Н.
_____
25 березня 2022 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Рибаку Анатолію Юрійовичу

1. Тема роботи – «Розробка програмного забезпечення для кас самообслуговування з використанням фреймворків .Net і Angular».

Керівник роботи: Романюк Оксана Володимирівна, к.т.н., доцент кафедри ПЗ, затверджені наказом вищого навчального закладу від 24 березня 2022 року №66.

2. Строк подання студентом роботи 13 червня 2022 року.

3. Вихідні дані до роботи: модель розробки – водоспадна; система управління базами даних – Microsoft SQL Server; мова запитів – C#; вхідні дані – база даних із товарами та інформацією про них конкретного закладу, обліковими записами клієнтів та історіями покупок, інформація про знижки для кожного покупця, особиста інформація клієнтів; середовище розробки – Microsoft Visual Studio; мова програмування – C#.

4. Зміст розрахунково–пояснювальної записки: вступ; обґрунтування вибору методу розробки та постановка задачі дослідження; розробка структури та алгоритмів роботи програмного продукту; розробка програмних компонент для каси самообслуговування; тестування програмного забезпечення; висновки; список використаних джерел; додатки.

5. Перелік графічного матеріалу: титульний лист; актуальність теми; мета, об'єкт та предмет дослідження; задачі дослідження; новизна одержаних результатів; практична цінність одержаних результатів; використані фреймворки; використання архітектури Onion; використання Microsoft SQL Server; схема бази даних; блок-схема загального алгоритму; блок-схема алгоритму генерування

знижок; тестування веб-додатку; апробація та публікація результатів бакалаврської дипломної роботи; фінальний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Романюк О.В., к.т.н., доцент кафедри ПЗ	25.03.2022	10.06.2022

7. Дата видачі завдання 25 березня 2022 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз завдання і вибір методу вирішення поставленої задачі дослідження	26.03.2022 - 03.04.2022	Вик.
2	Розробка загальної архітектури веб-додатку та бази даних	04.04.2022 - 14.04.2022	Вик.
3	Розробка загального алгоритму роботи веб-додаткку та алгоритму нарахування персональних знижок для користувачів	15.04.2022 - 28.04.2022	Вик.
4	Вибір середовища та мови розробки	29.04.2022 - 03.05.2022	Вик.
5	Програмна реалізація додатку для кас самообслуговування	04.05.2022 - 24.05.2022	Вик.
6	Тестування програми	25.05.2022 - 30.05.2022	Вик.
7	Оформлення матеріалів до захисту БДР	31.05.2022 - 10.06.2022	Вик.

Студент _____ **Рибак А.Ю.**
(підпис) (прізвище та ініціали)

Керівник бакалаврської дипломної роботи _____ **Романюк О. В.**
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська дипломна робота складається зі 108 сторінок формату А4, на яких є 41 рисунок, 5 таблиць, список використаних джерел містить 26 найменувань.

У бакалаврській дипломній роботі проведено детальний аналіз технологій для розробки кас самообслуговування. Встановлено об’єкт, предмет, завдання та методи дослідження. Сформульовано мету дослідження – підвищення швидкості та зручності процесу купівлі товарів у різних магазинах шляхом розробки програмного забезпечення для каси самообслуговування, а також підвищення попиту на більш широкий асортимент товарів роздрібної торгівлі за рахунок удосконалення алгоритму нарахування знижок. Для реалізації мети було розроблено веб-додаток для каси самообслуговування.

Запропоновано алгоритм нарахування знижок для кожного зареєстрованого покупця, який представляє собою генератор знижок на певну продукцію в залежності від частоти її придбання покупцем. Удоскнално графічний інтерфейс додатку для кас-самооблуговування.

Створено веб-додаток з використанням мови програмування C# та фреймворку .Net, а також з використанням фреймворку Angular для розробки графічного інтерфейсу та авторизації зареєстрованих користувачів у середовищах для розробки MS Visual Studio та MS Visual Code. Для розробки бази даних використана мова програмування C# з платформою для розробки веб-додатків Web Api та система управління базами даних MS SQL Sever. В результаті виконання бакалаврської дипломної роботи розроблено веб-додаток, працездатність і правильність роботи якого перевірено, підготовлена інструкція користувача.

Ключові слова: .Net, Angular, каси самообслуговування, MS SQL Sever, веб-додаток, C#, алгоритм.

ABSTRACT

The bachelor's thesis consists of 108 A4 pages, which have 41 figures, 5 tables, a list of sources used contains 26 items.

In the bachelor's thesis a detailed analysis of technologies for the development of self-service cash registers. The object, subject, tasks and research methods are established. The purpose of the study is to increase the speed and convenience of the process of buying goods in different stores by developing software for self-service cash registers, as well as increasing demand for a wider range of retail goods by improving the algorithm for calculating discounts. To achieve this goal, a web application for self-service checkouts was developed.

An algorithm for calculating discounts for each registered buyer is proposed, which is a generator of discounts for certain products depending on the frequency of its purchase by the buyer. The graphical interface of the application for self-service cash registers has been improved.

A web application was created using the C # programming language and the .Net framework, as well as using the Angular framework to develop a graphical interface and authorize registered users in MS Visual Studio and MS Visual Code development environments. The C # programming language with the Web Api web application development platform and the MS SQL Sever database management system were used to develop the database. As a result of the bachelor's thesis, a web application was developed, the efficiency and correctness of which was checked, and a user manual was prepared.

Keywords: .Net, Angular, self-service cash register, MS SQL Sever, web application, C #.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. Незважаючи на те, що сервіси самообслуговування не є чимось новим для західних держав, для України подібний тренд тільки починає імплементуватися у різноманітних варіаціях. Для українських споживачів на сучасному етапі стає дедалі більш звичайною справою надавати послуги самим собі без необхідності залучення інших осіб. Глобальна тенденція людей по всьому світу розбиратися у повсякденних речах самостійно проявляється у статистичних показниках: опитування, проведене SOTI серед споживачів демонструє, що 66% покупців віддають перевагу технологіям самообслуговування замість того, щоб звертатися безпосередньо до продавців та консультантів.

Основними причинами такого підвищення попиту на апарати самообслуговування є, по-перше, завантаженість та активний спосіб життя покупців, для яких час є надзвичайно цінним ресурсом. У сучасних умовах постінформаційного суспільства люди звикли до миттєвого та самостійного отримання послуги у зручному для них темпі. За даними опитування Box Technologies та Intel, 90% британців уникають довгих черг. По-друге, прилади самообслуговування значно скорочують скупчення людей і, відповідно, простір у магазинах, що робить їх вигідними не лише з точки зору залучення потенційних клієнтів, але й з погляду самих власників одиниць роздрібної торгівлі.

Серед варіантів самостійного здійснення послуг особливо виділяються каси самообслуговування (КСО). Дана форма обслуговування є актуальним та зручним рішенням через надмірно великий потік клієнтів, що допомагає боротися з перевантаженнями звичайних кас. Одним із прикладів найпопулярнішого використання кас самообслуговування в Україні є проект «Самокаса» у «Сільпо». Механізм роботи кас самообслуговування простий: магазин проводить самостійне сканування певних одиниць продуктів і здійснює оплату у зручній для покупця формі.

Таким чином, вибір теми дослідження обґрунтовується її актуальністю в сучасних умовах для України з огляду на зростання попиту на каси самообслуговування. Фреймворки .NET і Angular були використані через зручність

написання бази даних та логіки запитів веб-додатку та через їхню потужність роботи з візуальною частиною веб-додатку та можливості розробки функцій з реєстрації та авторизації користувачів.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно з планом виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження. Метою дослідження є підвищення швидкості та зручності процесу купівлі товарів у різних магазинах шляхом розробки програмного забезпечення для каси самообслуговування, а також підвищення попиту на більш широкий асортимент товарів роздрібної торгівлі за рахунок удосконалення алгоритму нарахування знижок.

Відповідно до поставленої мети в бакалаврській дипломній роботі потрібно вирішити наступні **завдання:**

- проаналізувати стан імплементації та методів розробки програмного забезпечення для кас самообслуговування;
- розробити архітектуру програмного додатку;
- розробити базу даних із товарами магазину, акаунтами клієнтів з їх особистими даними та історією замовлень;
- розробити алгоритм генерації знижок для кожного клієнта та збереження їх у базі даних;
- розробити графічний інтерфейс веб-додатку;
- розробити програмне забезпечення для кас самообслуговування на основі створеної архітектури та алгоритмів;
- провести тестування веб-додатку;
- розробити інструкцію користувача.

Об'єктом дослідження є процес розробки програмного забезпечення для кас самообслуговування.

Предметом дослідження виступає методи та засоби розробки програмного забезпечення для каси самообслуговування.

Методи дослідження. У бакалаврській дипломній роботі

використовувалися: теорія алгоритмів для розробки та вдосконалення алгоритмів програмного забечпечення; теорія баз даних для розробки бази даних із товарами магазину, акаунтами клієнтів з їх особистими даними та історією замовлень; комп'ютерне моделювання для аналізу та перевірки отриманих теоретичних положень; методи тестування програмного забезпечення для перевірки працездатності розробленого програмного продукту.

Новизна одержаних результатів.

1. Удосконалено алгоритм нарахування знижок на товари, у якому, на відміну від відомих алгоритмів, нараховуються персональні знижки на кожну одиницю товару для кожного зареєстрованого клієна в залежності від частоти купівлі різних товарів, що дозволило підвищити попит на більш широкий асортимент товарів роздрібної торгівлі.

2. Удосконалено графічний інтерфейс каси самообслуговування, у якому, на відміну від аналогів, присутній мінімально необхідний набір функціональних елементів, що дозволило підвищити швидкість та покращити зручність роботи з касою самообслуговування.

Практична цінність отриманих результатів.

Практична цінність одержаних результатів полягає в тому, що на основі отриманих в бакалаврській дипломній роботі теоретичних положень запропоновано алгоритми та розроблено веб-додаток для підвищення швидкості та зручності здійснення покупок у різноманітних магазинах.

Особистий внесок здобувача. Усі наукові результати, що викладені у бакалаврській дипломній роботі, отримано автором самостійно. У наукових працях, опублікованих у співавторстві, автору належать такі результати: особливості застосуванн фреймворку Angular для створення графічної частини веб-додатку [1].

Апробація результатів роботи. Результати роботи доповідалися на:

– LI Науково-технічній конференції факультету інформаційних технологій та комп'ютерної інженерії (2022 р., м. Вінниця).

Публікації. За тематикою дослідження опубліковано одну наукову працю у збірниках матеріалів конференцій.

1 ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДУ РОЗРОБКИ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Аналіз стану імплементації програмного забезпечення для кас самообслуговування

Каси самообслуговування та передбачене для них програмне забезпечення є ефективним ринковим рішенням у сучасних умовах. В Україні тенденція впровадження у процес самообслуговування нових технологій з метою отримання конкурентних переваг зростає з кожним роком. Це пов'язано з тим, що ця практика вже давно активно використовується в мережах роздрібної торгівлі на Заході та сприяє збільшенню попиту на товари певної одиниці роздрібної торгівлі. Процес розпочався у 2013 році із встановлення каси самообслуговування в одній з філій мережі “Велика кишеня” у Києві. Для українських ринків характерні два варіанти стратегій щодо впровадження кас самообслуговування: з використанням світових брендів із програмного забезпечення та використання власних систем. У контексті досвіду українських мереж роздрібної торгівлі щодо впровадження веб-додатків можна навести такі дані: новаторством для України стало запровадження системи автоматизованої торгівлі для супермаркету “Велика кишеня” з використанням холдингу Retail Group. Дана ініціатива дозволила клієнтам розраховуватися за товари самостійно за допомогою банківських карток та готівкового розрахунку. Подальшими прикладами досвіду встановлення веб-додатку для кас самообслуговування можна навести мережу Velmart в Україні, супермаркети Novus та Сільпо [2].

Незважаючи на наявність великого діапазону послуг на ринках за кордоном, український рітейл не може похвалитися швидкою імплементацією тенденції щодо оснащення магазинів необхідним обладнанням та програмним забезпеченням для здійснення самообслуговування клієнтами. В Україні існує проблема надмірної затратності подібних рішень і неясності ситуації щодо попиту на каси самообслуговування серед покупців. У зв'язку з цим імплементація проектів щодо

встановлення кас самообслуговування та програмного забезпечення до сих пір залишається низькою.

На поточному етапі існує широкий вибір пропозицій серед веб-додатків самообслуговування. Світові тенденції постачання до магазинів новітнього програмного забезпечення концентруються у Сполучених Штатах Америки, Європі та Китаї. Перший досвід імплементації кас самообслуговування було здійснено у Нью-Йорку у 1992 році. У контексті прикладів досвіду закордонного рітейлу серед програмного забезпечення, передбаченого для кас самообслуговування, виділяється пропозиція від компанії StrongPoint. Компанія створює веб-додатки для всіх видів самообслуговування для продуктового рітейлу. StrongPoint займає лідерські позиції на ринку програмного забезпечення за рахунок високої швидкості здійснення покупки, сканування та оплати. Система StrongPoint відрізняється тим, що має базу даних на основі ваги, яка гарантує, що всі продукти повинні бути просканові перед безпосередньою упаковкою. Для власників бізнесу є можливість комбінувати програмне забезпечення з обладнанням для кас самообслуговування StrongPoint або будь-яким іншим обладнанням, залежно від уподобань замовника [3].

Отже, для України питання розповсюдження кас самообслуговування в мережах роздрібної торгівлі є надзвичайно актуальним попри недостатнє фінансове забезпечення та відповідну низьку реалізацію проектів серед більшості міні-маркетів. При цьому світових досвід імплементації проектів з встановлення кас самообслуговування з веб-додатками може стати в нагоді при вдосконаленні власне українського підходу до проблематики.

1.2 Порівняльний аналіз аналогів

Не дивлячись на всі переваги реалізації каси самообслуговування у вигляді веб-додатку, на сьогоднішній день найбільшою популярністю користуються спеціально облаштовані каси самообслуговування на базі десктопної програми. Серед найбільш популярних аналогів програмного забезпечення для спеціальних облаштованих кас можна відзначити наступні:

– TEMABIT;

– Erply;

– SystemGroup / NCR;

TemaBit є розробником IT-продуктів для бізнесу однієї з найбільших комерційних і промислових груп України Fozzy Group. Команда компанії розробила понад 80% програмного забезпечення, яким користуються понад 65 000 співробітників, тисячі партнерів і мільйони гостей Сільпо, Фора, Fozzy, Thrash! і служби доставки Justin.

Однією з переваг компанії є наявні розробки кас самообслуговування в магазинах лінійки Сільпо. Проект носить назву «Самокаса».

Каса має приємний дизайн, простий та зрозумілий інтерфейс. Покупець сканує товар, перекладає його на контрольну платформу та обирає зручний для нього тип оплати, завершуючи покупку. На всіх контрольних платформах присутні спеціальні датчики для порівняння відсканованого і покладеного на платформу товару. Автоматичні каси можуть забезпечувати оплату як банківськими картками, так і готівкою. Для цього вони обладнані купюроприймачами та можуть видавати решту покупцеві[4]. Але вона має свої недоліки, а саме: немає можливості авторизації та реєстрації клієнтів, відсутня можливість змінити мову інтерфейсу та немає реалізації алгоритму нарахування знижок для клієнтів. На рисунку 1.1 зображено інтерфейс каси самообслуговування в магазині Сільпо.

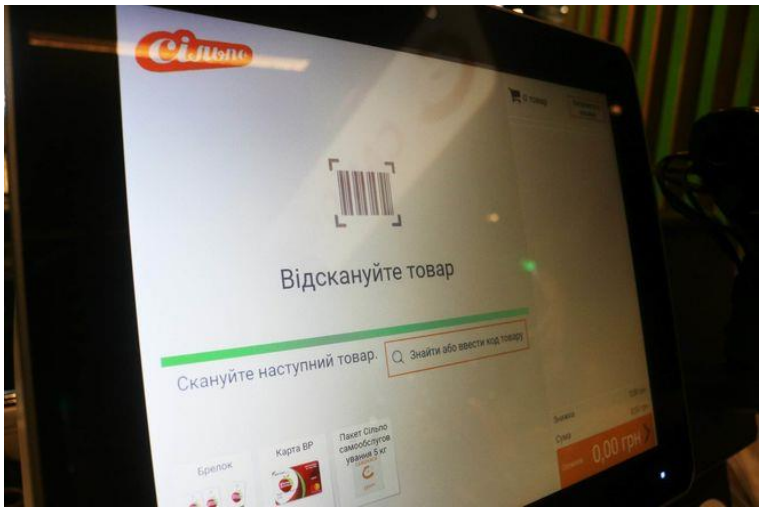


Рисунок 1.1 – Користувацький інтерфейс каси у магазині Сільпо

Ще одним прикладом використання кас самообслуговування є система компанії Erply. Програмне забезпечення, що пропонує компанія, дозволяє отримати доступ до інвентарю магазину в режимі реального часу і в будь-якому місці, зберігати дані магазину, клієнтів, інвентар і т. д. зручно у хмарі. Перевагами є зручний інтерфейс та функціонал, проте немає можливості зміни мови інтерфейсу[5]. На рисунку 1.2 наведено інтерфейс веб-програми компанії Erply.

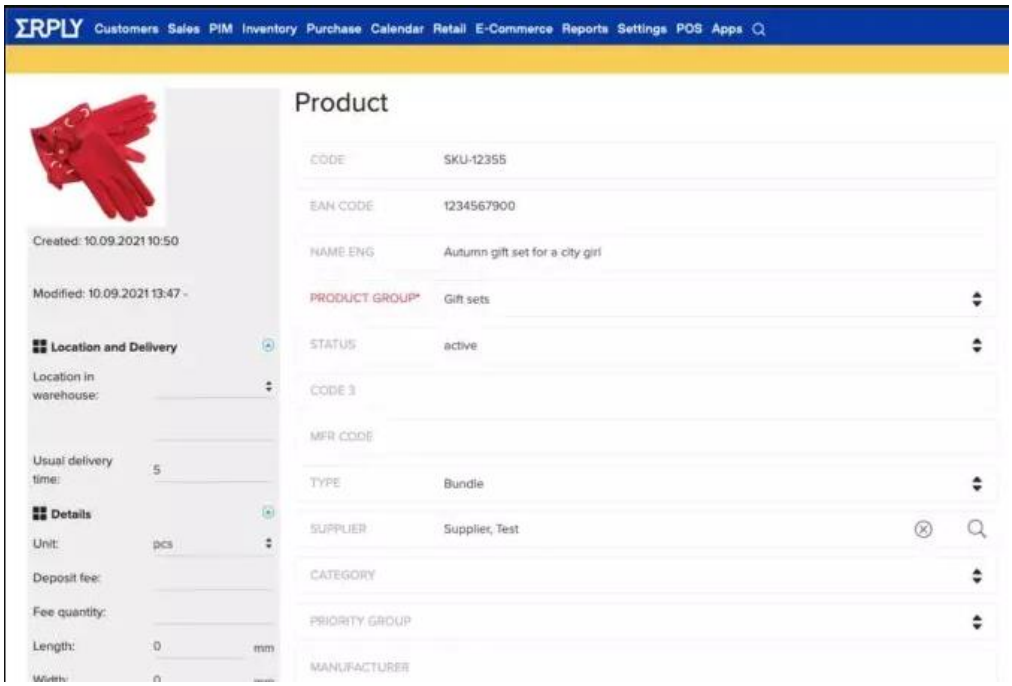


Рисунок 1.2 – Користувацький інтерфейс програмного забезпечення Erply

SystemGroup має розробки в області кас самообслуговування. Їх найбільш успішним проектом є використання своїх кас в таких популярних закладах як Велика Кишеня, McDonalds, Small & SKIF.

У закладах McDonald's відвідувачам надалась можливість замовляти свої замовлення без касира. Покупці можуть скористатися новою касою самообслуговування та зробити замовлення оплативши його банківською карткою. Каса володіє інтерактивним доступом до меню, вибіром типу замовлення (забрати з собою або отримати на місці), типом оплати, так як замовлення можна оплатити готівкою після його отримання, а також передає інформацію про нове замовлення робітникам самого закладу, які одразу розпочнуть його приготування. Клієнт може

отримати своє замовлення у зоні видачі замовлень. Перевагами даної каси є великий дисплей, який підкреслюється адаптивним інтерфейсом, проте немає можливості авторизуватися та зареєструватися для користувача[6]. На рисунку 1.3 наведено інтерфейс каси у McDonald's.

Рисунок 1.3 – Користувацький інтерфейс каси у McDonald's

Проаналізувавши аналоги визначено їх сильні та слабкі місця та проведено порівняння із розроблюваним веб-додатком для каси самообслуговування “Express pay”. Результати порівняння показано в таблиці 1.1.

Таблиця 1.1 – Порівняльні характеристики програмних продуктів

Критерій	Сільпо	Erply	McDonald's	Express Pay
Реєстрація та авторизація користувачів	0	1	0	1
Зміна мови інтерфейсу	0	0	1	1
Алгоритм генерування знижок для клієнтів	0	0	0	1

В результаті порівняння існуючих аналогів було дійдено до висновку, що розробка власного веб-додатку для каси самообслуговування є більш ніж доцільною. Внаслідок розробки отримаємо продукт, який покриває недоліки існуючих аналогів та забезпечує порівняно вищу ефективність та більший обсяг функціоналу.

1.3 Аналіз методів розв’язання поставленої задачі

Існує декілька способів вирішення задачі як зберігати інформацію про всіх користувачів, про всі товари та інформацію про нараховані знижки клієнтам. Першим є спосіб збереження на хмарних базах даних таких як Amazon Web Service, Microsoft Azure, MongoDB Atlas, база даних Oracle. Вони, звісно, дуже зручні у використанні та мають зручний та налаштований інтерфейс. На даний момент є дві поширені можелі розгортання: клієнти можуть придбати послугу до бази даних, що обслуговується постачальником хмарного сервісу, або використовуючи хмарні сховища, запустити базу даних безпосередньо на них за допомогою віртуальної машини. Серед хмарних баз даних є як SQL-орієнтовані так і ті, що використовують модель даних NoSQL. Але такий підхід має і свої недоліки, а саме:

1. Залежність від обраного провайдера.

Підприємства переміщують робочі навантаження в хмару з різних причин, чи то ліквідація центрів обробки даних, міграція успадкованих робочих навантажень або створення та запуск програм з високою продуктивністю в більш гнучкому середовищі. Робота традиційних та спочатку хмарних робочих додатків у хмарі вимагає стабільно високої продуктивності та надійності в усьому стеку.

2. Вимоги до кваліфікації з будівництва Cloud Native архітектур.

При перенесенні систем у постійному вигляді споживання ресурсів хмар може бути невиправдано дорогим. У зв'язку з цим крім факту фізичного перенесення даних при перенесенні враховується також підхід до роботи з даними. Великі компанії вводять посаду CDO, в обов'язки якого входить управління оптимізацією зберігання та використання даних.

Другим способом збереження інформації є створення бази даних власноруч за допомогою фреймворку .Net, Entity Framework, та мови програмування C#. Такий підхід менш затратний та набагато гнучкіший у розробці бази даних, тому саме таке рішення було прийнято для розробки бази даних для каси самообслуговування.

Алгоритм генерування знижок для покупців було вирішено розробити таким чином, щоб в залежності від типів товарів і кількості, яку придбав споживач, йому нараховувалась знижка на топ-3 найбільш популярних товари, на які він здійснив покупку. Даний алгоритм слідкує за активністю кожного зареєстрованого покупця та генерує йому знижку в залежності від товару та їхньої кількості, яку він придбав протягом тижня. Через тиждень всі знижки оновлюються та є змога отримати її на будь-який інший товар, таким чином кількість покупців та продажі будуть збільшуватись, заохочуючи нових клієнтів.

Реалізація даної ідеї у вигляді веб-додатку є найбільш вигідним варіантом для клієнтів та розробників, адже його можна відкрити у більшості браузерів, він не є вимогливим до ресурсів самої каси самообслуговування. Все, що потребуватиметься, – це налаштувати касу та провести декілька операцій для тесту системи.

Будь-які операції і взаємодії покупця з касою приводять до запитів до бази даних. Тому для веб-додатку було використано технології Web API та HTTPS протоколи. Протокол передачі гіпертексту (HTTP) – це прикладний протокол для розподілених, спільних гіпермедійних інформаційних систем, який дозволяє користувачам передавати дані у всесвітню мережу та сьогоднішній день він часто використовується у сфері клієнтно-серверних взаємодій.

Протокол містить в собі двох учасників, а саме клієнта та сервера. Перший відноситься до програмнонго забезпечення, яке ініціює з’єднання, надсилає через нього запити клієнта, потім приймає і оброблює відповідь. А сервер очікує поки встановиться з'єдняння, після цього отримує запит і обробляє його. Після обробки він надсилає результат клієнтові.

Таким чином веб-додаток обмінюється даними про користувачів та покупки з сервером та забезпечує стабільну роботу каси самообслуговування.

Для будь-якого веб-додатку необхідно мати графічну частину, а саме, візуальний інтерфейс та всі необхідні функції. Для вирішення цієї задачі було використано графічний фреймворк Angular. Angular – це відкритий фреймворк і платформа для створення односторінкових програм, написаних на TypeScript та підтримуваних і розроблених Google. Angular має очевидні переваги як фреймворк, а також надає стандартну структуру для роботи розробникам. Це дозволяє користувачам створювати великі програми в обслуговуваний спосіб.

Отже, великою особливістю фреймворку Angular є те, що він використовує мову програмування TypeScript. За допомогою цього фреймворку було розроблено графічний інтерфейс веб-додатку, а також реєстрацію та авторизацію клієнтів, а також протоколи HTTP для зв’язку з базою даних.

1.4 Постановка задач розробки веб-додатку для касс самообслуговування

Після проведення аналізу питання розробки веб-додатку для кас самообслуговування, було визначено наступні завдання, які необхідно виконати для розробки веб-додатку:

- проаналізувати стан імплементації та методів розробки програмного забезпечення для кас самообслуговування;
- розробити архітектуру програмного додатку;
- розробити базу даних із товарами магазину, акаунтами клієнтів з їх особистими даними та історією замовлень;
- розробити алгоритм генерації знижок для кожного клієнта та збереження їх у базі даних;
- розробити графічний інтерфейс веб-додатку;
- розробити програмне забезпечення для кас самообслуговування на основі створеної архітектури та алгоритмів;
- провести тестування веб-додатку;
- розробити інструкцію користувача.

Технічне завдання на розробку наведено в додатку А.

1.5 Висновки

У першому розділі було розглянуто стан питання існуючих рішень для кас самообслуговування на сьогодні. Також було проведено аналіз існуючих аналогів та проведено їх порівняння між собою та розроблюваним веб-додатком. У результаті доведено доцільність розробки веб-додатку, а також проведено аналіз існуючих підходів до вирішення поставленої задачі. Було встановлено основні завдання, які необхідно виконати для розробки веб-додатку.

2 РОЗРОБКА СТРУКТУРИ ТА АЛГОРИТМІВ РОБОТИ ПРОГРАМНОГО ПРОДУКТУ

2.1 Розробка архітектури програмного продукту

Для початку потрібно визначити основні рівні та архітектуру веб-додатку [7]. Першим повинен бути рівень для бази даних, який забезпечується базою даних (Database layer); Другим на черзі є рівень доступу до даних. Він надає можливість доступитись до бази даних, створювати транзакції, викнонувати запити бази даних, закривання транзакції і т.д. (Data Access layer); Третім має бути рівень бізнес-логіки. Він в свою чергу перетворює введену користувачем інформацію в дані зрозумілі для системи(Business logic layer); Четвертим на черзі є рівень сервісів. Через нього проходять всі запити до клієнтської системи(Service layer); П’ятим є рівень візуалізації даних, який повинен реалізовувати візуалізацію веб-сторінки(Presentation layer) [8].

На рисунку 2.1 зображено приклад архітектури веб-додатку.

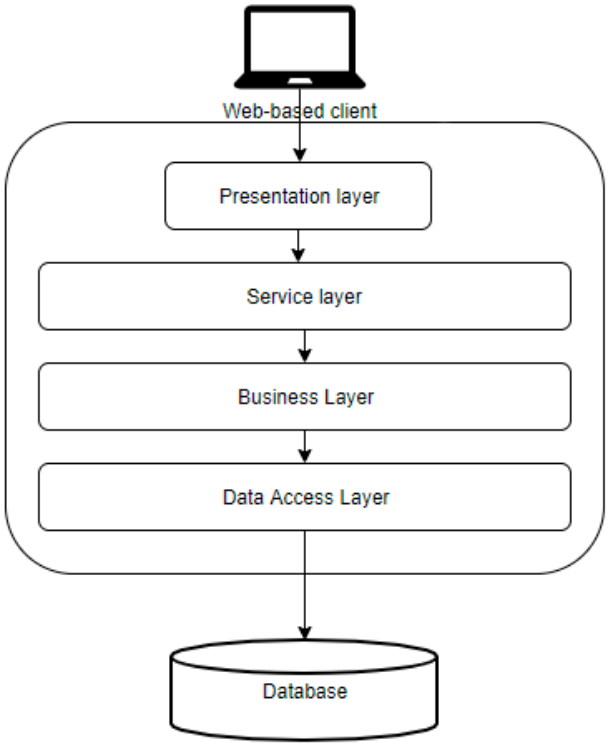


Рисунок 2.1 – Архітектура веб-додатку

Як видно з малюнка 2.1, система має клієнта – це веб-браузер користувача. При використанні веб-сайту, запити надсилаються на сервер додатків. Кожен такий запит ініціює відповідний виклик на рівні візуалізації даних.

Після цього в процес включається рівень сервісу, який визначає, які саме дані потрапляють в систему, наскільки вони точні і які подальші дії з ними потрібно виконати. Рівень сервісу – це шар між візуалізацією, яка представляє дані користувачеві, і бізнес-логікою, яка фактично виконується на стороні сервера і забезпечує легкий доступ до них [9].

Рівень бізнес-логіки обробляє перетворення та маніпулювання даними. Він сам вирішує, в якій формі і в якій таблиці відображатимуться зміни, ініційовані користувачем. Наступний крок – підключення до бази даних і запис необхідних змін. Якщо дія, яку запитав користувач, успішна, веб-сторінка відображає очікуваний результат. Якщо виникає помилка, рівень, на якому представлені дані, визначає рівень помилки, і користувач отримує відповідне сповіщення.

Детальний приклад зальної архітектури веб-додатку показано на рисунку 2.2.

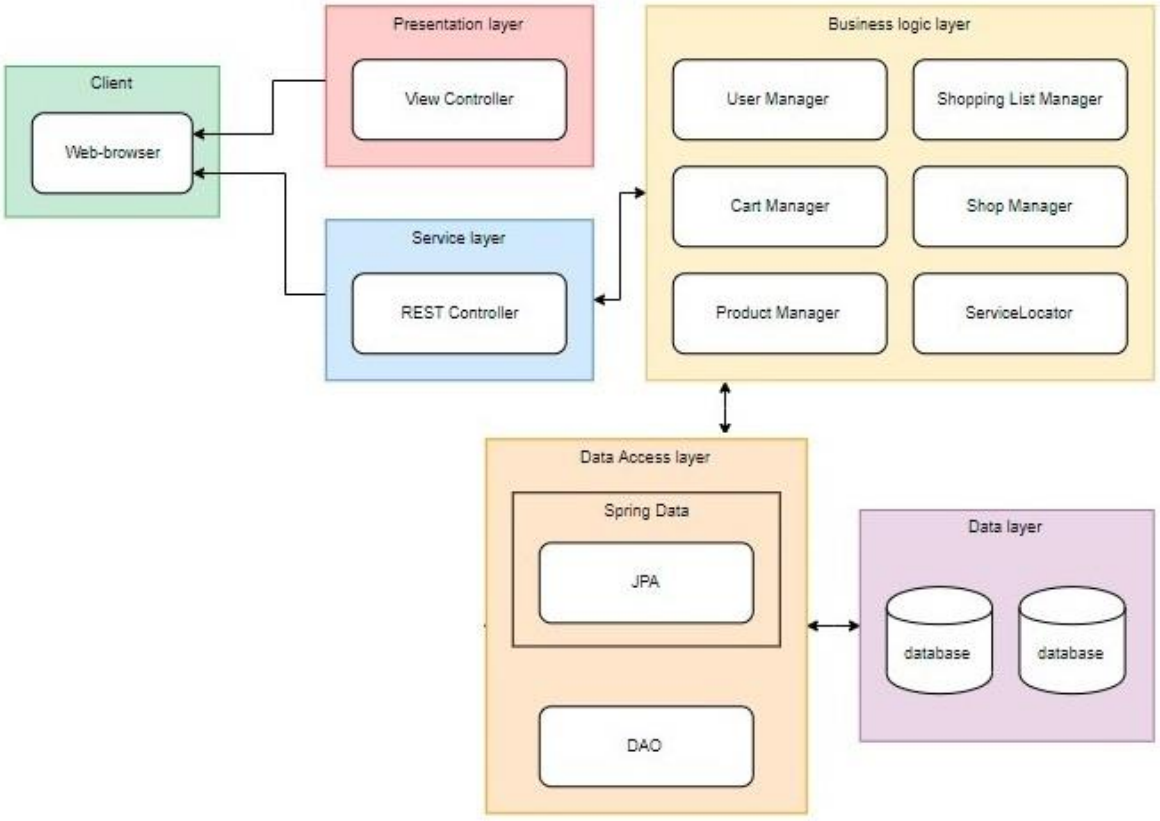


Рисунок 2.2 – Приклад загальної архітектури веб-додатку

Рівень сервісу забезпечує взаємодію між клієнтом і сервером додатків (сервер додатків).

Насправді рівень сервісу – це набір контролерів REST, які разом визначають API (інтерфейс програмного забезпечення) – інтерфейс програмного програмування.

Через цей інтерфейс система забезпечує доступ до своєї функціональності в єдиній, суворо визначеній та задокументованій формі. Рівень відтворення, у свою чергу, відповідає візуальному дизайну інформації, що відображається на веб-сторінці у веб-браузері[10].

Презентації та архітектура рівнів зображені на рисунку 2.3.

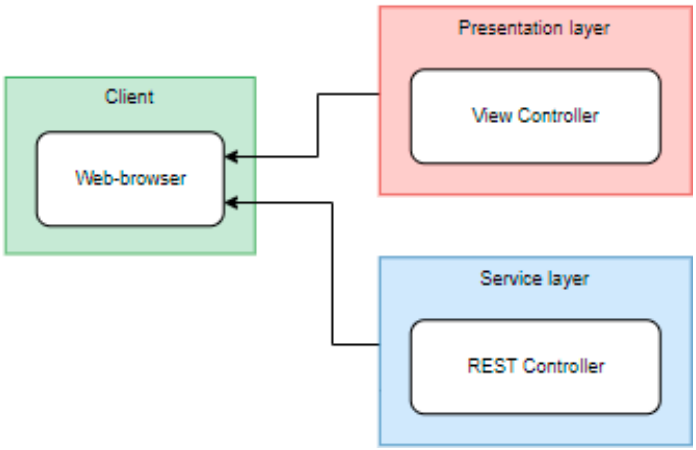


Рисунок 2.3 – Презентації та архітектура рівнів

Фактичний рівень презентації являє собою набір файлів з фреймворку Angular. Це можуть бути файли HTML, CSS, JavaScript і т.д.

Обидва рівні працюють за протоколом HTTP і приймають запити. Щоб забезпечити безпеку та конфіденційність, усі взаємодії користувачів, будь то передача логіну/паролю, створення замовлення чи будь-які транзакції з ним, мають проходити через захищену версію HTTPS.

У нашому випадку доцільно буде використовувати архутектуру Onion.

Більшість традиційних архітектур піднімають фундаментальні проблеми тісного зв’язку та розділення проблем. Onion архітектура була представлена, щоб забезпечити кращий спосіб створення додатків з точки зору кращої тестованості та

надійності. Onion архітектура вирішує проблеми, з якими стикаються трьохрівневі та багаторівнені архітектури, а також вирішує багато поширених проблем. Шари архітектури.

Onion взаємодіють один з одним за допомогою інтерфейсів. Використання інверсії залежностей у всьому проекті, залежно від абстракцій (інтерфейсів), а не від реалізацій, дозволяє нам прозоро вимкнути реалізацію під час виконання. Ми залежимо від абстракцій під час компіляції, що дає нам суворі контракти для роботи, і нам надають реалізацію під час виконання.

Ми можемо писати бізнес-логіку, не турбуючись про будь-які деталі реалізації. Якщо нам знадобиться щось із зовнішньої системи чи служби, ми можемо просто створити для неї інтерфейс і використовувати його. Нам не потрібно турбуватися про те, як це буде реалізовано. Вищі шари Onion подбають про прозору реалізацію цього інтерфейсу.

Основна ідея архітектури Onion – це потік залежностей, точніше, як шари взаємодіють один з одним. Чим глибше шар знаходиться всередині Onion, тим менше залежностей він має. Таким чином архітектура Onion найкраще підходить для розробки веб-додатку [11].

Приклад архітектури продемонстровано на рисунку 2.4.

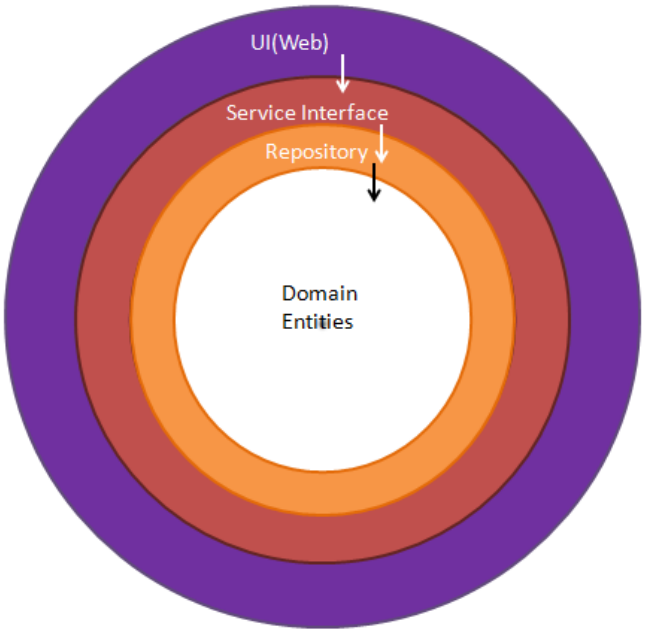


Рисунок 2.4 – Приклад архітектури Onion

Архітектура має такі рівні:

1. Рівень об’єктів домену (Domain Entities Layer).

Це центральна частина архітектури. Він містить усі об’єкти домену програми. Якщо програма розроблена за допомогою структури сутностей ORM, тоді цей рівень містить класи POCO (Code First) або Edmx (Database First) з сутностями. Ці об’єкти домену не мають жодних залежностей.

2. Рівень сховища (Repository Layer).

Рівень призначений для створення рівня абстракції між шаром сутностей домену та рівнем бізнес-логіки програми. Це шаблон доступу до даних, який спонукає до більш слабко пов’язаного підходу до доступу до даних. Ми створюємо загальний репозиторій, який запитує джерело даних для даних, зіставляє дані з джерела даних на бізнес-суб’єкт і зберігає зміни в бізнес-суб’єкті в джерелі даних.

3. Рівень сервісу (Service Layer).

Рівень містить інтерфейси, які використовуються для зв’язку між рівнем UI та рівнем сховища. Він містить бізнес-логіку для сутності, тому його також називають рівнем бізнес-логіки.

4. Рівень інтерфейсу користувача (UI Layer).

Це найбільш зовнішній шар. Це може бути веб-додаток, веб-API або проект модульного тестування. Цей рівень має реалізацію принципу інверсії залежностей, так що програма створює слабо пов’язану програму. Він зв’язується з внутрішнім рівнем через інтерфейси.

Отже, у даному підрозділі було розглянуто вимоги, які повинна мати архітектура розроблюваного веб-додатку. Продемонстровано та вибрано для розробки архітектуру Onion, оскільки вона добре підходить для розробки веб-додатку для кас самообслуговування.

2.2 Розробка алгоритмів роботи додатку

Щоб розробити веб-додаток для кас самообслуговування необхідно розробити загальний алгоритм роботи додатку (рисунок 2.5), базу даних, алгоритми обробки вхідних та вихідних даних.

Загальний алгоритм роботи додатку має такі кроки: отримання вхідних даних, обробка вхідних даних, звернення до бази даних через протоколи HTTP, формування чеку замовлення з отриманих даних від бази даних, вивід даних на екран. Блок-схема алгоритму зображена на рисунку 2.5.

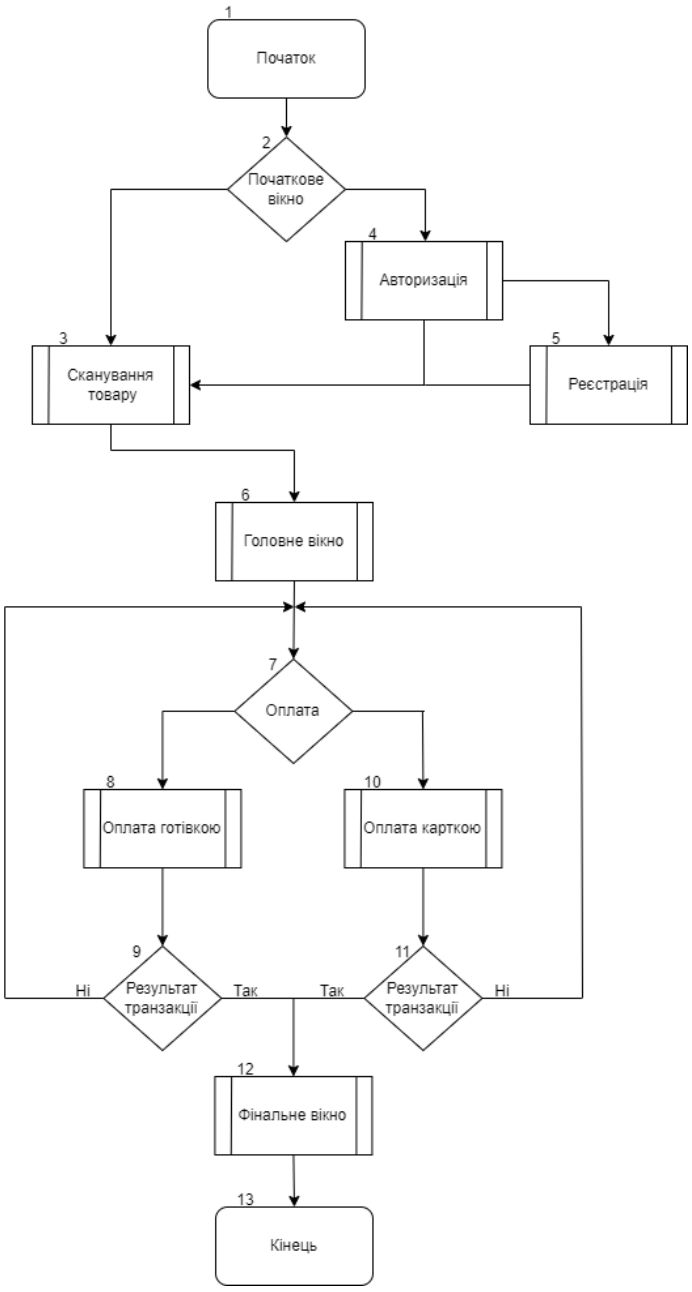


Рисунок 2.5 – Блок-схема загального алгоритму роботи веб-додатку

Для кращого розуміння роботи алгоритму, розглянемо його більш детальніше покроково:

1) Початок;

2) Початкове вікно. На ньому користувач одразу має можливість відсканувати усі необхідні для нього товари, а також перейти до вікна авторизації у сисетемі та потім до вікна реєстрації, якщо це необхідно;
3) Сканування товару. Користувач має змогу просканувати весь небхідний товар. Після цього веб-додаткок зв’язується з базою даних та переріяє чи такий товар присутній у базі даних. Після цього користувач автоматично переходить на головне вікно веб-додатку;
4) Авторизація. У цьому вікні користувач може авторизуватися у системі та одразу сканувати товар;
5) Реєстрація. У цьому вікні користвач може зареєструватися у системі та також одразу сканувати товар;
6) Головне вікно. У цьому вікні відображаються усі відскановані товари покупця та відображається загальна сума покупки. Також є вибір оплати: готівкою або карткою. Після цього користувач переходить на вікно з оплатою;
7) Оплата. На цьомі вікні користувач має можливість опатити всі покупки в залежності від вибраного способу оплати;
8) Оплата готівкою. Якщо користувач вибрав варіант оплати готівкою він оплачує та чекає результат оплати;
9) Результат оплати(готівка). Якщо результат оплати успішний, користувач переходить до фінального вікна, а якщо ні, то повертається до вікна з оплатою;
10) Оплата карткою. Якщо користувач вибрав варіант оплати карткою він оплачує та чекає результат транзакції;
11) Результат оплати (картка). Якщо результат транзакції успішний, колистувач переходить до фінального вікна, а якщо ні то повертається до вікна з оплатою;
12) Фінальне вікно. На фінальному вікні користувач бачить повідомлення про успішну оплату та подяку за покупку;
13) Кінець.

Також було розроблено алгоритм нарахування персональних знижок для клієнтів. Блок-схем алгоритму наведено на рисунку 2.6.

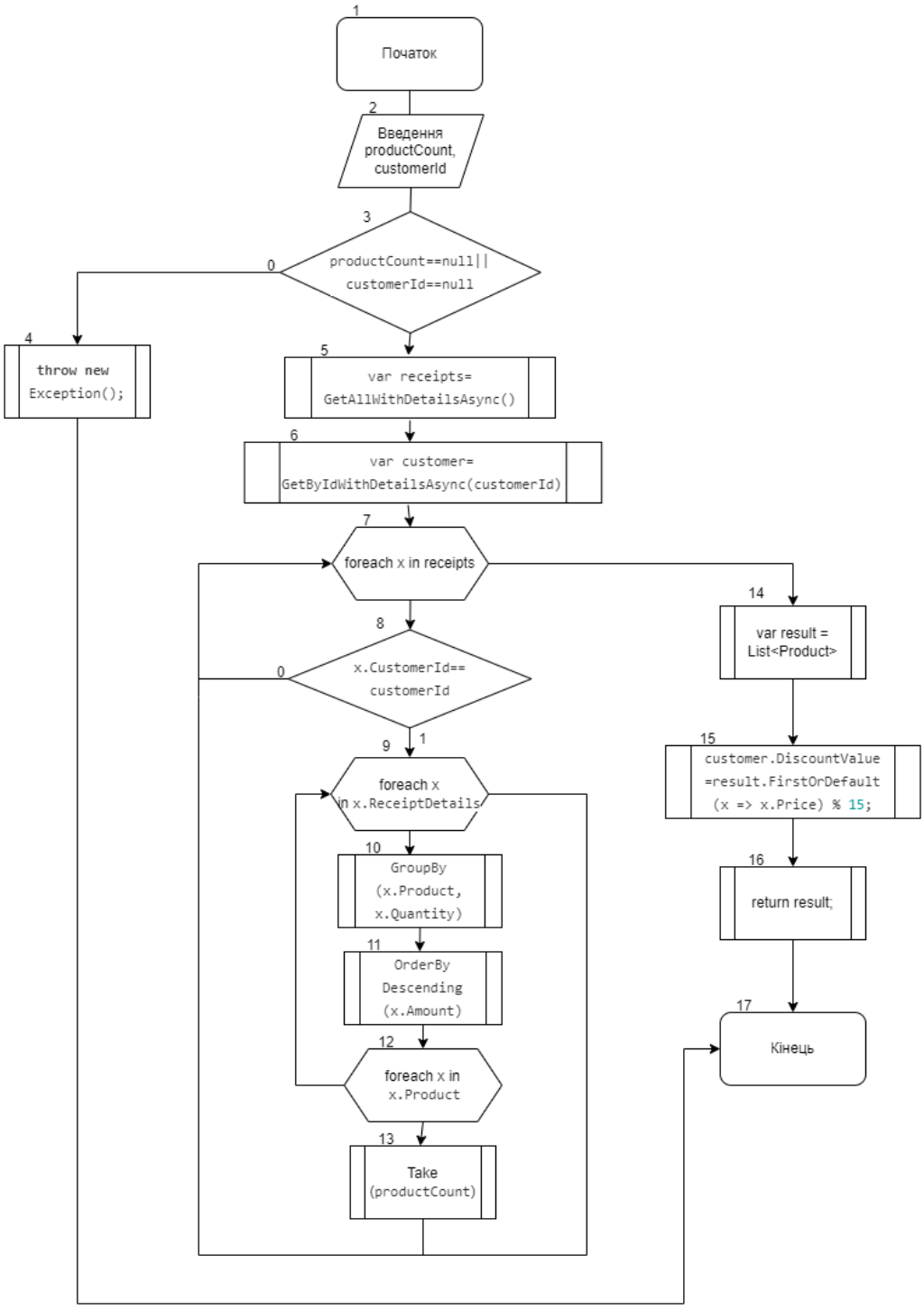


Рисунок 2.6 – Блок-схема алгоритму нарахування знижок

Для кращого розуміння роботи алгоритму, розглянемо його більш детальніше покроково:

1) Початок;
2) Введення productCount, customerId;
3) Перевірка вхідних даних на нуль;
4) Виключення якщо перевіряємі значення нуль;
5) Отримання чеків;
6) Отримання клієнтів;
7) Цикл для знаходження самого популярного товару;
8) Пошук до унікальному id;
9) Цикл для знаходження товару у детальніших записах про чеки;
10) Групування по товарах та їх кількості;
11) Сортування за зменшенням по кількості;
12) Цикл для знаходження товару;
13) Вибірка по кількості, яка була передана до алгоритму;
14) Отримання результату з циклу;
15) Установка знижки;
16) Повернення результату;
17) Кінець.

Отоже, у даному підрозділі було розроблено основний алгоритм веб-додатку та алгоритм генерування знижок, а також проаналізовано їх роботу покроково.

2.3 Розробка бази даних

В даний час існує велика кількість різноманітних баз даних, які надають можливості зіставлення даних. База даних визначає не тільки те, як дані зберігаються, а й спосіб доступу до них.

Проектування бази даних – це процес створення проекту бази даних, що призначена для підтримки функціонування економічного об'єкта та сприяє досягненню його цілей[12].

Це є трудомістким процесом, що вимагає спільних зусиль аналітиків, проектувальників та користувачів.

Під час проектування бази даних необхідно враховувати те що, що база даних має задовольняти комплексні вимоги. Ці вимоги такі:

– Цілісність бази даних - вимога повноти та несуперечності даних;
– Багаторазове використання даних;
– Швидкий пошук та отримання інформації за запитами користувачів;
– Простота оновлення даних;
– Мінімізація надмірності даних;
– Захист даних від несанкціонованого доступу, спотворення та знищення.

Етапи життєвого циклу бази даних.

Життєвий цикл – це процес проектування, реалізації та підтримки БД, що складається з наступних етапів: Попереднє планування; Перевірка здійсненності; Визначення вимог; Концептуальне проектування; Логічне проектування; фізичне проектування; Оцінка роботи та підтримка БД.

Попереднє планування – збирається інформація про програми та файли, що використовуються і розробляються, пов'язаних з ними, інформація документується у вигляді узагальненої концептуальної моделі даних.

Перевірка здійсненності – готуються звіти щодо: технологічної здійсненності; операційної здійсненності економічної ефективності.

Визначення вимог – формулюються цілі БД, інформаційні потреби підрозділів та їх керівників, вимоги до програмного та апаратного забезпечення.

Концептуальне проектування – створюється докладна модель власних уявлень даних предметної області. Ці моделі інтегруються у концептуальну модель.

Логічне проектування – перетворення концептуальної моделі на логічну модель. Попередньо потрібно вибрати тип моделі.

Фізичне проектування – створюється фізична модель БД у вигляді розширення логічної моделі з такими характеристиками: типи устрою зберігання; необхідний обсяг пам'яті; способи доступу до БД та деякі інші.

Порівняльна характеристика наведена в таблиці 2.1.

Таблиця 2.1 – Порівняння баз даних

	Oracle	MS SQL Server	MySQL	MongoDB	Cassandra
SQL синтаксис	1	1	1	0.5	0
Об’єм даних	1	1	0	0	1
Швидкість читання	1	1	1	0.5	0
Швидкість запису	0.5	0.5	0.5	1	1
Реляційна модель	1	1	1	0	0
Транзакційність	1	1	1	0.5	1
Атомарність	1	1	1	1	1
Консистентність	1	1	1	0.5	1
Ізоляція	1	1	1	0.5	0.5
Довговічність	1	1	1	1	1
Реплікація	0	0.5	0	1	1
Без- коштовна ліцензія	0	1	1	1	1
Підсумок	9.5	11	9.5	7.5	8.5

Виходячи з порівняльної характеристики, найбільш очевидним і задовільним рішенням буде база даних Microsoft SQL Server. Безсумнівною перевагою цієї бази

даних є підтримка реляційної моделі даних, підтримка принципів ACID, наявність індексів та підтримки синтаксису SQL, а також безкоштовна ліцензія, що є безумовною перевагою.

Реалізацію бази даних можна побачити на рисунку 2.7.

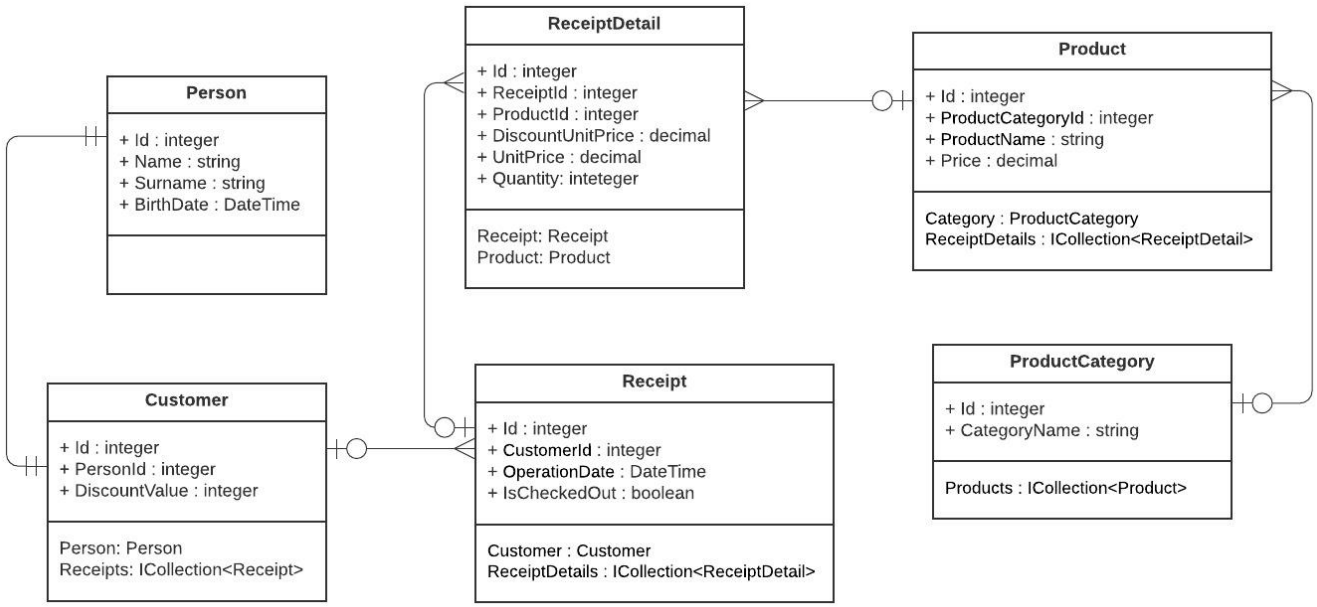


Рисунок 2.7 – Схема бази даних для кас самообслуговування

База даних розроблена за допомогую Entity Framework длякий дозволяє рохроблювати усі сущності, зв'язки та все що потрібно для бази даних, пишучи все мовою програмування C# та без жодної рядка SQL коду. Це швидкий та гнучкий потенціал у написаннях різноманітних баз даних.

Розроблена база даних містить 6 таблиць зі своїми сутностями та зв’язками. Між таблицями присутні такі зв’язки:

– Person-Customer: зв’язок «один до одного».

– Customer-Receipt: зв’язок «один до багатьох».

– Receipt-ReceiptDetail: зв’язок «один до багатьох».

– Product-ReceiptDetail: зв’язок «один до багатьох».

– Product-ProductCategory: зв’язок «один до багатьох».

У зв’язку «один до одного» рядок в таблиці Person може мати не більше одного рядка, що збігається в таблиці Customer, і навпаки. Зв’язок "один до одного"

створюється, якщо обидва зв’язані стовпці є первинними ключами або мають унікальні обмеження.

На рисунку 2.8 зв’язок «один до одного».

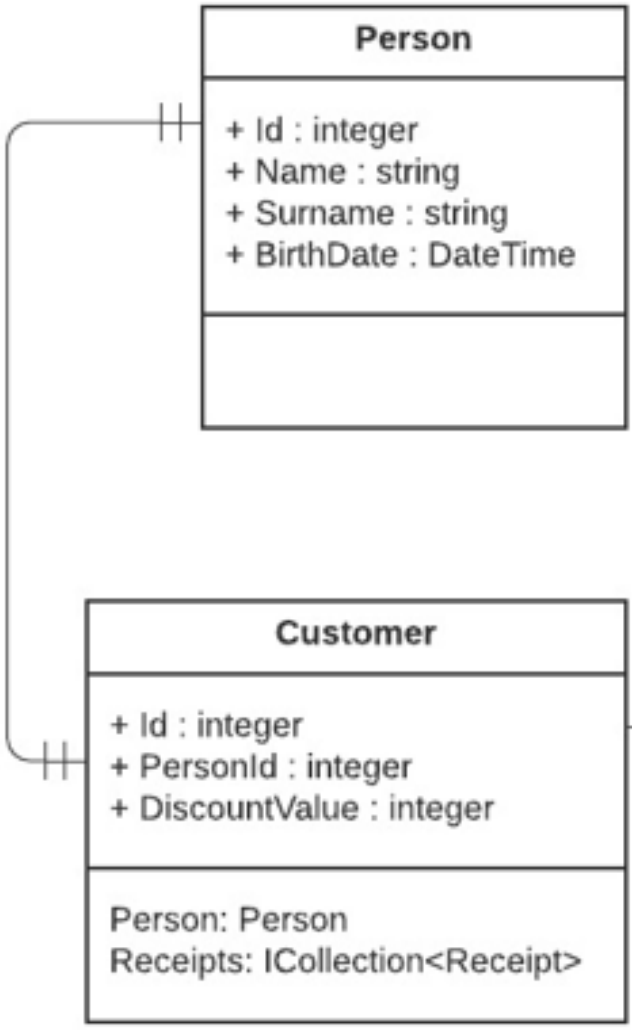


Рисунок 2.8 – Зв’язок «один до одного»

У зв’язку «один до багатьох» рядок у таблиці Customer може мати багато рядків у таблиці Receipt. Але рядок у таблиці Receipt може мати лише один рядок у таблиці Customer. Тобто кожен покупець може мати багато чеків з покупками. Але кожна покупка належить лише одному покупцю. На рисунку 2.9 продемонстровано зв’язок «один до багатьох».

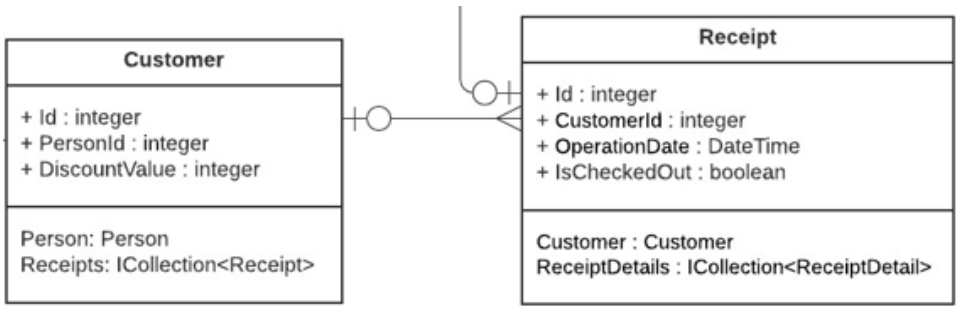


Рисунок 2.9 – Зв’язок «один до багатьох»

У розробленій базі даних є такі таблиці: Person, Customer, Receipt, ReceiptDetail, Product, ProductCategory.

Таблиця Person містить у собі інформацію про користувача який зареєструвався та має такі поля:

– Id – унікальне id користувача;
– Name – ім’я користувача;
– Surname – прізвище користувача;
– BirthDate – день народження користувача.

Таблиця Customer містить інформацію про самого клієнта та має такі поля:

– Id – унікальне id покупця;
– PersonId - унікальне id користувача;
– DiscountValue – персональна знижка.

Таблиця Receipt містить інформацію про чек клієнта та має такі поля:

– Id – унікальне id замовлення;
– CustomerId – унікальне id покупця;
– OperationDate – дата покупки;
– IsCheckedOut – результат покупки.

Таблиця ReceiptDetail містись детальну інформацію про зміст чека та має такі поля:

– Id – унікальне id детального замовлення;
– ReceiptId – унікальне id замовлення;
– ProductId – унікальне id товару;

– DiscountUnitPrice – ціна за одиницю твару зі знижкою;
– UnitPrice – ціна за одиницю;
– Quantity – кількість одиниць певного товару.

Таблиця Product містить інформацію про продукт та має такі поля:

– Id – унікальне id товару;
– ProductCategoryId – унікальне id категорії товару;
– ProductName – назва товару;
– Price – ціна товару.

Таблиця ProductCategory містить інформацію про категорію продукту та має такі поля:

– Id – унікальне id категорії товару;
– CategoryName – назва категорії товару.

Отже, в даному підрозділі було визначено основні вимоги до бази даних, порівняння самих популярних баз даних. Також було створено власну базу даних, створено та проаналізовано сутності, зв’язки та атрибути створеної бази даних.

2.4 Розробка структури графічного інтерфейсу користувача

Графічний інтерфейс (GUI) – це тип інтерфейсу користувача, який дозволяє користувачам переміщатися по комп'ютеру або пристрою і виконувати дії за допомогою візуальних індикаторів і графічних значків. Всі основні операційні системи, такі як Windows, Mac, iOS та Android, мають графічний інтерфейс, в якому ви можете клацнути значок, щоб виконати таку дію, як відкриття програми, перегляд меню або переміщення вашого пристрою.

Спочатку GUI були розроблені для використання з мишею та клавіатурою, але тепер широко використовуються в багатьох портативних мобільних пристроях, таких як смартфони та планшети, та які використовують комбінацію технологій для забезпечення платформи для взаємодії. На відміну від операційної системи з командним рядком або CUI, операційні системи GUI набагато простіше вивчати та використовувати для новачків, тому що команди не потрібно запам'ятовувати, а

користувачам не потрібно знати мови програмування. Структуру початкового вікна показано на рисунку 2.10.

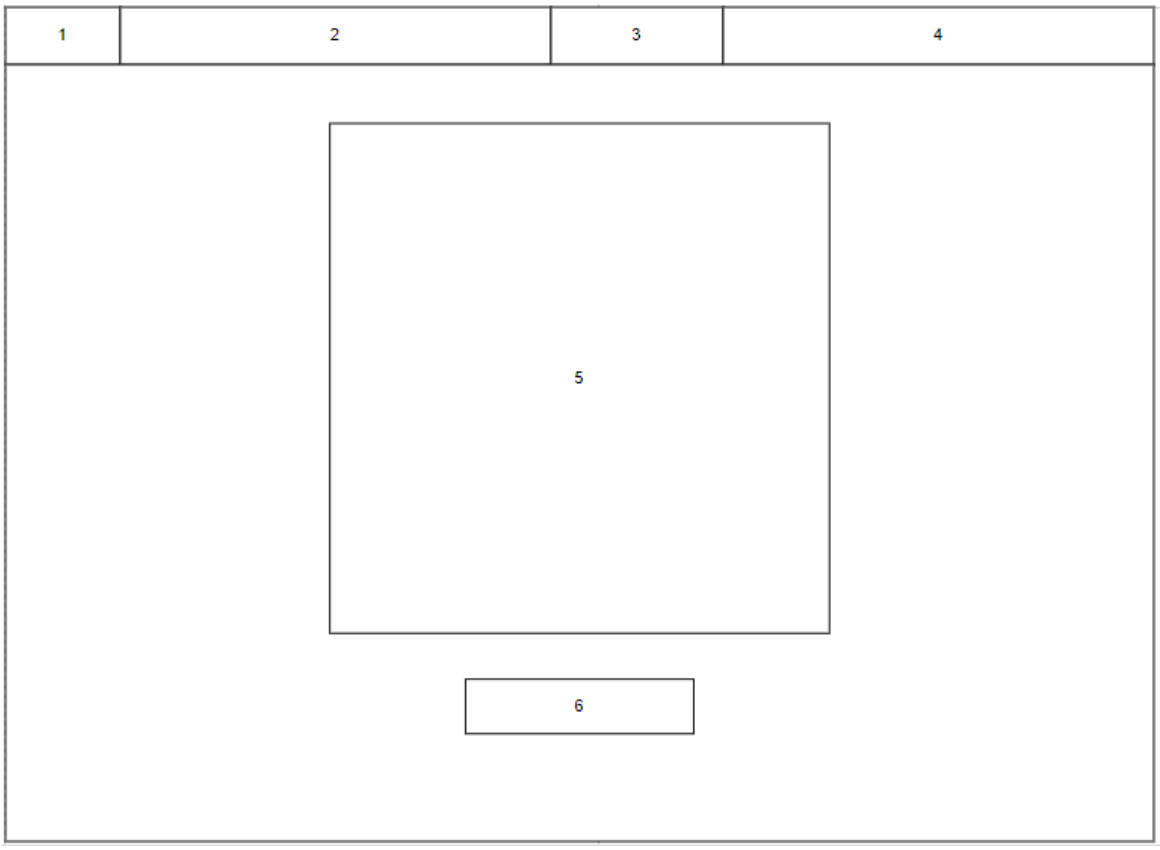


Рисунок 2.10 – Структурна схема початкового вікна

Основні елементи інтерфейсу головного вікна:

1) Кнопка зміни мови;

2) Сторінка сканування товару;

3) Сторінка з чеком та оплатою товару;

4) Фінальне вікно;

5) Привітання та логотип;

У початковому вікні клієнт бачить привітання та можливість одразу відсканувати товар або авторизуватися. Розглянемо вікно з авторизацією на рисунку 2.11.

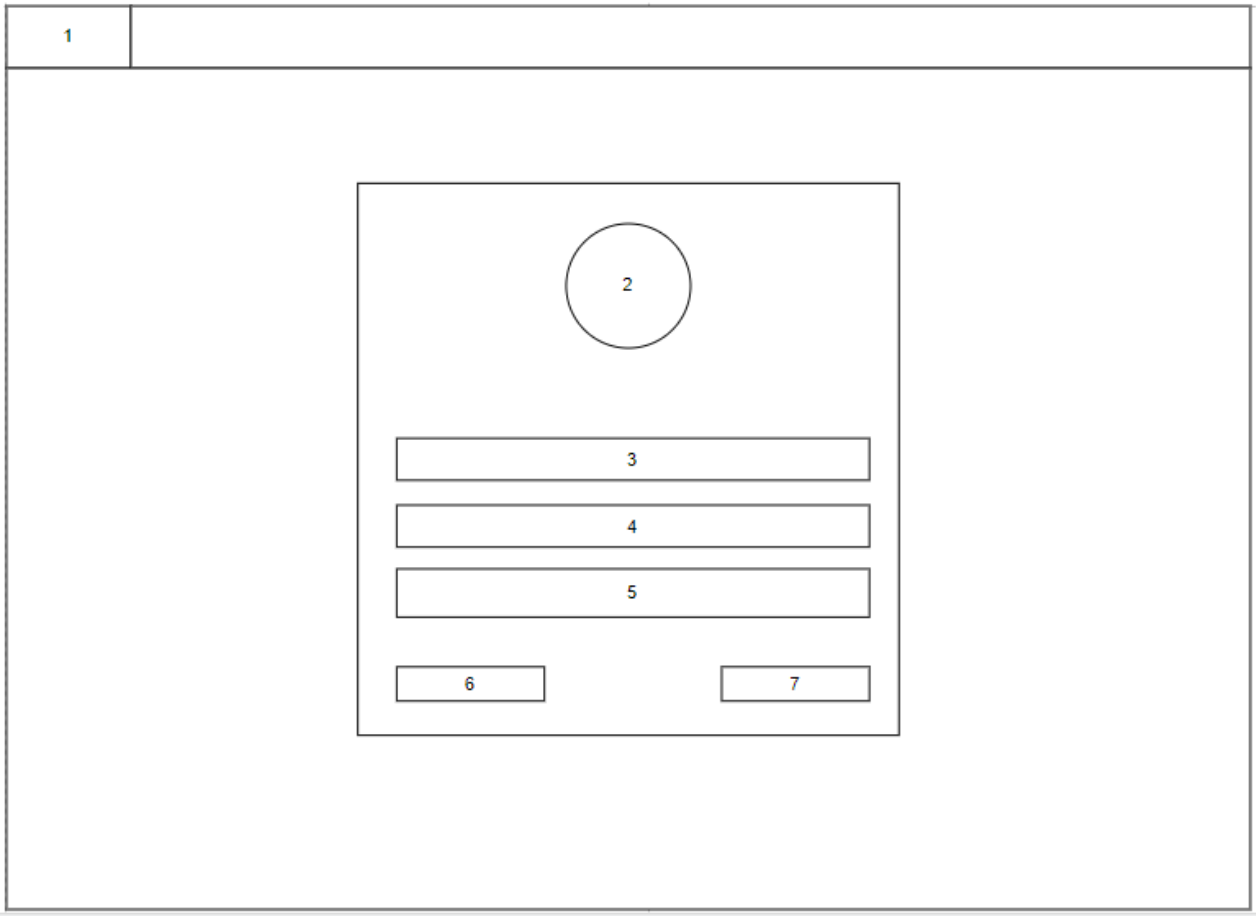


Рисунок 2.11 – Структурна схема вікна з авторизацією

Основні елементи інтерфейсу головного вікна:

1) Кнопка зміни мови;
2) Картинка авторизації;
3) Поле для введеня номеру телефону;
4) Поле для введеня паролю;
5) Кнопка для входу;
6) Кнопка назадж
7) Кнопка для реєстрації

У цьому вікні користувач може авторизуватися, або зареєструватися, а ткож вийти, нажавши на кнопку «назад». Тепер розглянемо головне вікно з товарами та способом оплати на рисунку 2.12.

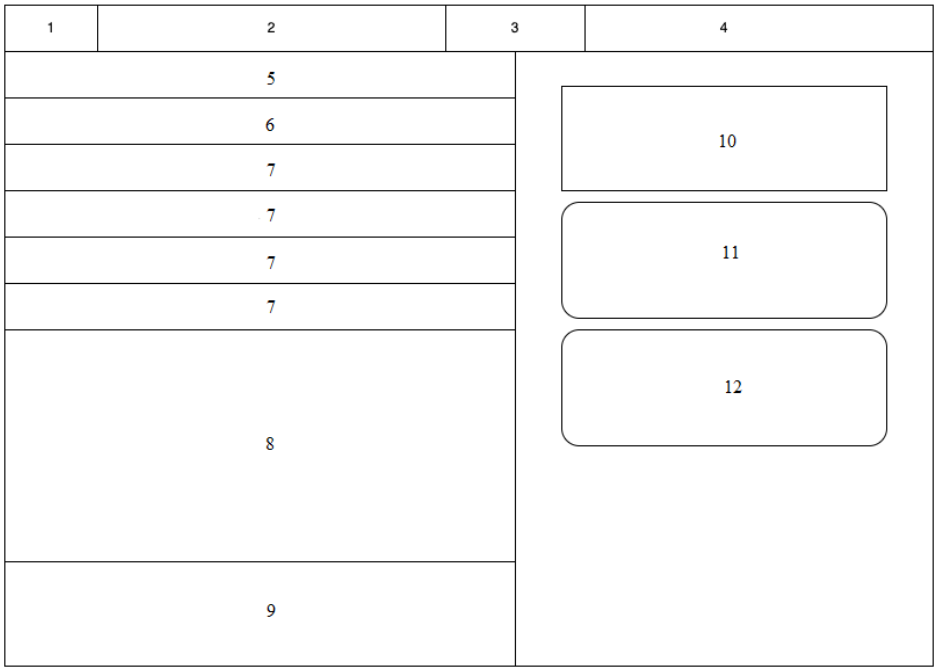


Рисунок 2.12 – Структурна схема головного вікна додатку

Основні елементи інтерфейсу головного вікна:

1) Кнопка зміни мови;
2) Сторінка сканування товару;
3) Сторінка з чеком та оплатою товару;
4) Фінальне вікно;
5) Панель кошику з товарами;
6) Панель з товаром;
7) Товар із інформацією;
8) Вікно для товарів;
9) Панель з інформацією про суму покупки;
10) Панель для вибору способу оплати;
11) Кнопка оплати банківською карткою;
12) Кнопка оплати готівкою.

У цьому вікні користувач бачить всі відскановані товари раніше та суму яку повинен оплатити за них, а також може вибрати спосіб оплати та змінити мову інтерфейсу. Розглянемо більш детально схему інтерфейсу вікна з оплатою, що зображена на рисунку 2.13.

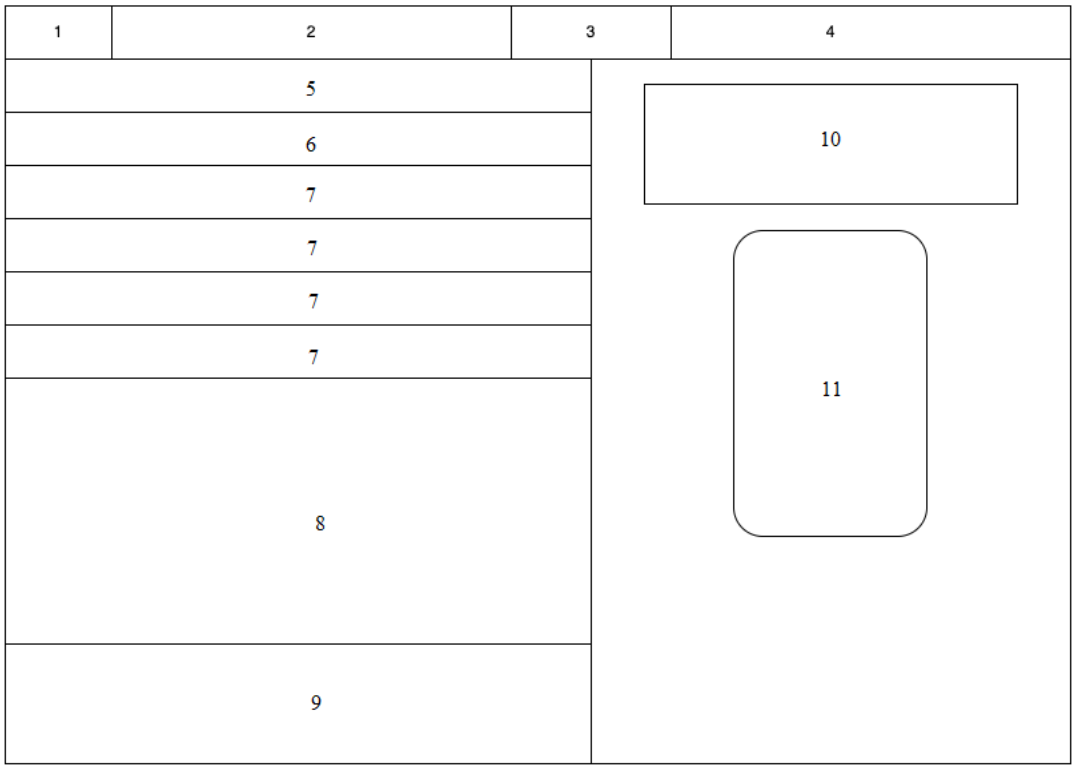


Рисунок 2.13 – Структурна схема вікна з оплатою

Основні елементи інтерфейсу головного вікна:

1) Кнопка зміни мови;
2) Сторінка сканування товару;
3) Сторінка з чеком та оплатою товару;
4) Фінальне вікно;
5) Панель кошику з товарами;
6) Панель з товаром;
7) Товар із інформацією;
8) Вікно для товарів;
9) Панель з інформацією про суму покупки;
10) Панель застосування картки;
11) Логотип карткового терміналу.

На цьому вікні користувач безпосередньо розраховується за придбаний товар безготівковим способом оплати. І на фінальному фікні буде відображено результат про успішну транзакцію та вдячність за покупку, рисунок 2.14

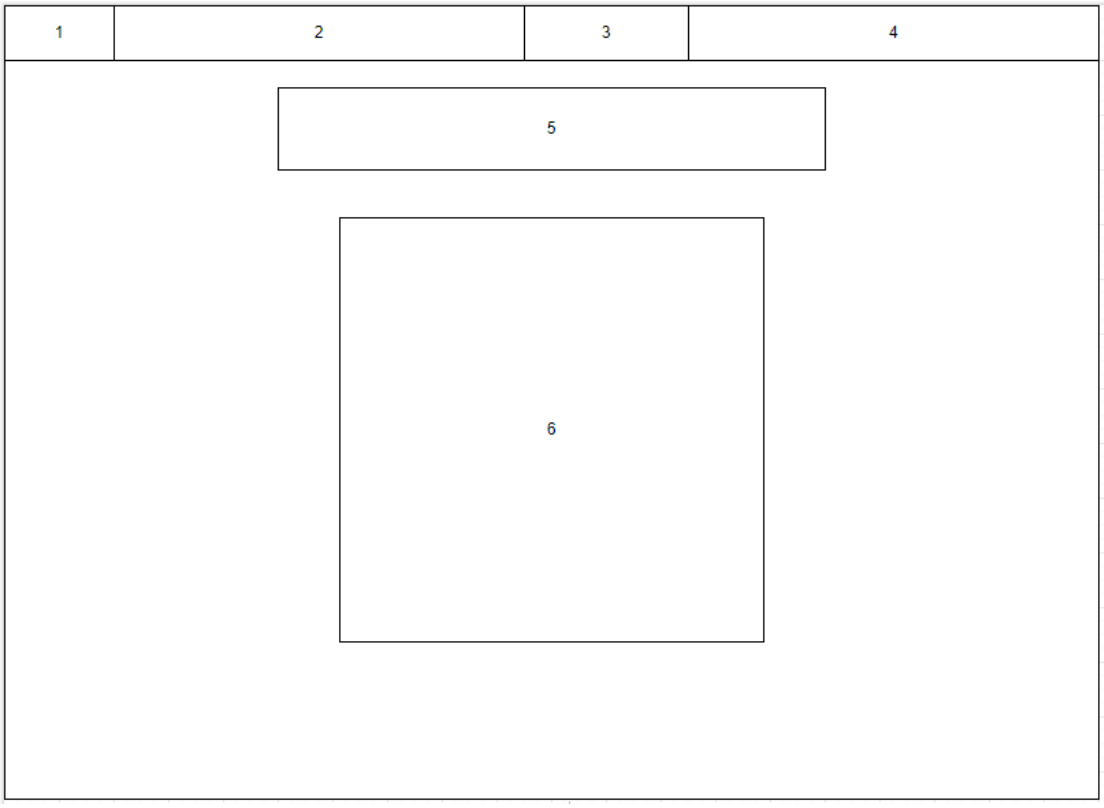


Рисунок 2.14 – Структурна схема фінального вікна

Отже, в даному підрозділі було розглянуто структуру графічного інтерфейсу користувача для створюваного додатку. Надано базові відомості про GUI, наведено особливості та вимоги при розробці інтерфейсу для веб-додатку для кас саообслуговування Розглянуто й детально описано структурну схему головного вікна додатку, наведено структурні схеми для окремих вікон. Описано призначення окремих елементів інтерфейсу при використанні різних функцій веб-додатку.

2.5 Висновки

У другому розділі було розглянуто архітектуру створюваного програмного забезпечення, наведено схему компонентів та їх взаємодії, розроблено блок-схеми загального алгоритму роботи веб-додатку, а також алгоритму генерації нарахування знижок. Було створено базу даних та розглянуто аналоги. Було розроблено структурні схеми інтерфейсу для головного вікна додатку та окремих вікон, описано призначення елементів інтерфейсу при використанні різних функцій веб-додатку.

3 РОЗРОБКА ПРОГРАМНИХ КОМПОНЕНТ ДЛЯ КАСИ САМООБСЛУГОВУВАННЯ

3.1 Варіантний аналіз і обґрунтування вибору мови програмування

На початку розробки будь-якого програмного забезпечення необхідно приділити велику увагу у виборі мови програмування паралельно з засобами для розробки. Для цього потрібно визначитись з чітко поставленими задачами та їхніми вимогами до написання проекту. Під час визначення мови програмування потрібно ознайомитися з їхніми особливостями. Популярність мови програмування залежить від функцій та утиліт, які вона надає програмістам. Особливості, які повинна мати мова програмування з метою виділитися, полягають у наступному:

- Мова програмування повинна бути простою, легкою для вивчення та використання, добре читабельною та впізнаваною людиною.
- Абстракція є обов’язковими характеристиками для мови програмування, в якій з’являється здатність визначати складну структуру, а потім і ступінь її зручності.
- Перевага завжди надається портативному мові програмування.
- Ефективність мови програмування повинна бути високою, щоб її можна було легко конвертувати в машинний код і виконувати мало місця в пам’яті.
- Мова програмування повинна бути добре структурована та задокументована, щоб вона придатна для розробки додатків.
- Необхідні інструменти для розробки, налагодження, тестування, обслуговування програми повинні бути забезпечені мовою програмування.
- Мова програмування повинна забезпечувати єдине середовище, відоме як інтегроване середовище розробки (IDE).
- Мова програмування повинна бути послідовною з точки зору синтаксису та семантики.

Отож, розглянемо найбільш популярні мови програмування, які використовуються для створення баз даних, а саме: Phyton, Java, C#.

Python – це дуже популярна мова програмування загального призначення, що широко використовується. Вона має великий набір спеціально розроблених модулів і широко використовується розробниками. Багато онлайн-сервісів надають API для Python. Python дуже простий у вивченні. Низький поріг входження робить її ідеальною першою мовою для тих, хто займається програмуванням. Такі програмні пакети як pandas, scikit-learn та Tensorflow роблять Python надійним варіантом для сучасних програм у галузі машинного навчання [13].

Серед недолікив можна зазначити її типобезпеку. Python – це динамічно типована мова, а це означає, що потрібно бути обережними під час роботи з нею. Помилки невідповідності типів (наприклад, передача рядка (string) як аргумент методу, який очікує ціле число (integer)), можуть іноді траплятися.

Java – це надзвичайно популярна мова загального призначення, вона використовує вуртуальну машину Java Virtual Machine (JVM). Це абстрактна обчислювальна система, що забезпечує плавну переносимість між платформами. В даний час підтримується корпорацією Oracle [14].

Перевагою даної мови є універсальність. Багато сучасних систем та програм розроблені за допомогою мови Java. Величезною перевагою такого ЯП є здатність інтегрувати методи науки даних безпосередньо в існуючу кодову базу.

Сувора типізація. Забезпечення типобезпеки не є порожнім звуком для Java, і в разі розробки критично важливих додатків для роботи з великими даними ця особливість як ніколи важлива.

Java – це високопродуктивна, скомпільована мова загального призначення. Це робить її придатною для написання ефективного виробничого коду ETL, а також алгоритмів машинного навчання з використанням обчислювальних засобів.

Недоліки: «багатослівність» мови Java робить її не найкращим варіантом для проведення спеціальних аналізів та розробки більш спеціалізованих статистичних додатків.

C# – це мова з C-подібним синтаксисом. В даному аспекті вона є близькою до C++ та Java. Будучи об'єктно-орієнтованою мовою, вона багато перейняла у Java та C++. Як і Java, C# спочатку призначалася для веб-розробки, і приблизно 75% її

синтаксичних можливостей подібно до мови Java. C# також іноді називають «очищеною версією Java» [15]. Об'єктно-орієнтований підхід дозволяє будувати за допомогою C# великі, але водночас гнучкі, масштабовані та розширювані додатки. Для створення та роботою з базами даних C# і .NET застосовується технологія ADO.NET. Для спрощення роботи з базами даних також застосовується Entity Framework – технологія ORM, яка дозволяє зіставляти сутності C# з таблицями у базі даних [16].

C# підтримує багато корисних функцій:

– інкапсуляція,
– успадкування,
– поліморфізм,
– перевантаження операторів,
– статична типізація.

Не дивлячись на все, C# активно розвивається та з кожними новими версіями цієї мови програмування з’являється все більше нового функціоналу. Наприклад, динамічне зв'язування, лямбди, асинхронні методи і т.д [17].

Переваги:

– Переважна більщість типів даних має фіксований розмір, що в свою чергу збільшує «мобільність» цієї мови та спрощує програмування, тому що завжди точно зрозуміло, з чим розробник має справу.
– Автоматичне «складання сміття». Це означає, що розробникам у більшості випадків не доведеться дбати про звільнення пам'яті. Вищезазначене середовище CLR власноруч викличе спеціальний збирач сміття та очистить пам'ять.
– Велика кількість синтаксичного «цукору» – спеціальних конструкцій, розроблених для розуміння та написання коду. Вони мають велике значення при компіляції.
– Низький поріг входження. Синтаксис C# має багато схожого на інші мови програмування, завдяки чому полегшується перехід для програмістів. Мова C# часто визнають найбільш зрозумілою та придатною для новачків.

– За допомогою Xamarin на C# можна писати програми для таких операційних систем, як iOS, Android, MacOS та Linux [18].

Недоліки:

– Більш оптимізована для Windows алвтформи;
– Мова переважно безкоштовна для починаючиш програмістів, студентів та колистувачів з не дуже серйозними проектами, але для великих компаній купівля ліцензії на мову обходиться в велику суму.

На таблиці 3.1 показано порівняння мов програмування.

Таблиця 3.1 – Порівняння мов програмування

Критерій	Python	Java	C#
Простота синтаксису	0	0	1
Об’єктно-орієнтованість	1	1	1
Легкість роботи із базами даних	0	1	1
Інструменти для легкого аналізу даних	1	0	1
Обширність та доступність документації	0	1	1
Різноманітність наявних фреймворків для створення бази даних	1	1	1
Підсумковий результат	3	4	6

За підсумками таблиці 3.1 можна дійти до висновку, що мова програмування C# найкраще підходить задля створення бази даних для кас самообслуговування, оскільки має зручні та необхідні фреймворки та інструменти, а також має зручний синтаксис.

3.2 Вибір середовища розробки

Важливим аспектом у розробці бази даних для кас самообслуговування є вибір середовища розробки. Інтегрованим середовищем розробки (IDE) називають велике програмне рішення для різноманітних розробок будь-якого програмного забезпечення. Найбільш популярними інтегрованими середовищами розробки, які

використовуються для реалізації програмних продуктів на мові програмування C# є Visual Studio, Project Rider, Eclipse.

Visual Studio визнана найкращою IDE для C#. Справа в тому, що обидва продукти належать корпорації Microsoft. Тому вони ідеально підходять для роботи один з одним [19].

До переваг Visual Studio входить таке:

– Найкраща IDE для C#-розробника
– Середовище містить багато інструментів, які дуже добре працюють на C#.
– Наявність безкоштовної версії – Community Edition.
– Community містить все, що потрібно для незалежного розробника.
– Найефективніше програмне забезпечення для розробки на будь-якій платформі, включаючи .Net і C#.
– Можливість зберігання даних у хмарі.

Продукт має деякі недоліки:

– Вибагливість до ресурсів.
– Після переходу на платну версію, можуть злетіти налаштування та корпоративний сервер.
– Складність при самостійному освоєнні.

Приклад інтерфейсу наведено на рисунку 3.1.

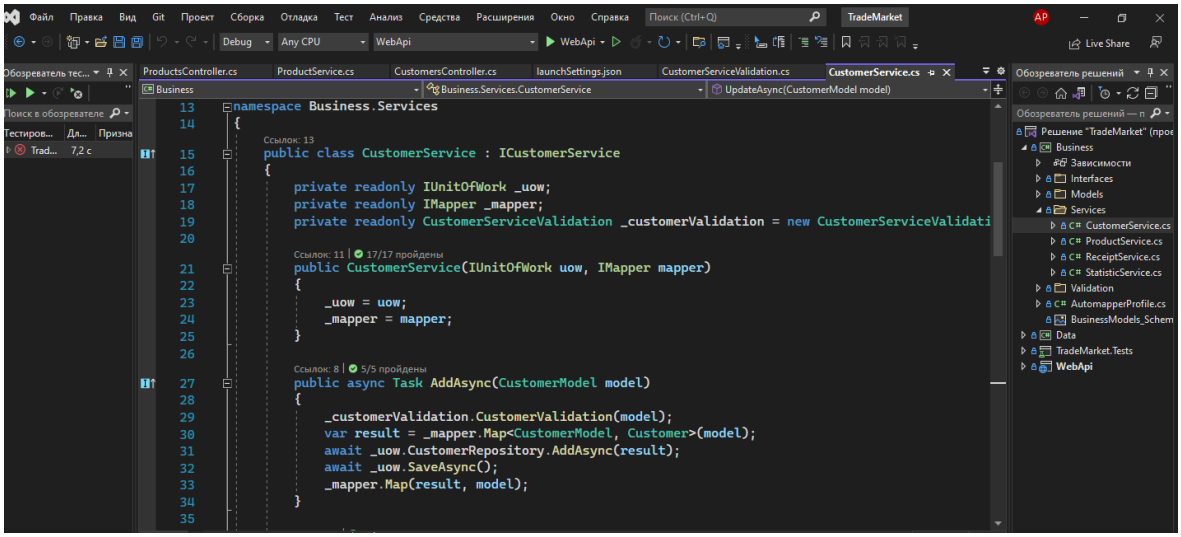


Рисунок 3.1 – Приклад інтерфейсу «Visual Studio»

Project Rider є кросплатформеною IDE для .Net. Він підходить для Windows, Linux, Mac OS X. Також базується на IntelliJ IDEA та Resharper[20].

До її переваг входить таке:

– Чудово реалізована підтримка інтелектуальних сполучень клавіш.
– Підходить для створення різного програмного забезпечення: ASP.Net, Xamarian і т.д.
– Підтримка навігації та рефакторингу.
– Зв’язок з Visual Studio та Unity.
– Підтримка HTML, XAML, JavaScript, C#, TS та інших мов.

До недоліків відносять такі властивості:

– Невелика частина основного функціоналу перебуває у процесі розробки, через це продукт містить деякі помилки та баги.
– Висока ціна. Ціна за використання платформи є великою але є trial-версія та знижки для студентів.

Приклад інтерфейсу наведено на рисунку 3.2.

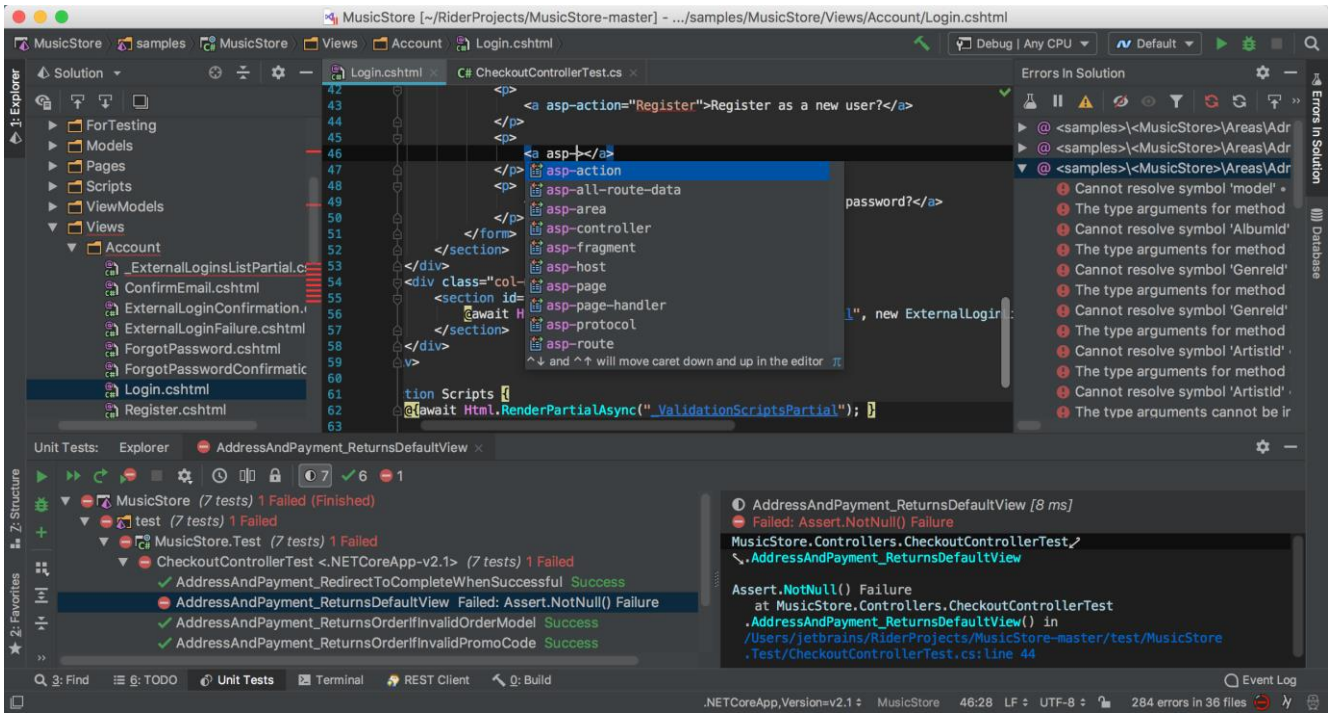


Рисунок 3.2 – Приклад інтерфейсу «Project Rider»

Eclipse aCute – плагін для Eclipse IDE. Він полегшує розробку C#. aCute дає можливість використовувати редактор C#, до складу якого входить Eclipse IDE, що підтримує мови за допомогою сервера Omni-sharp [21].

До переваг відносять наступне:

– Виділяє синтаксис кольором.
– Можливість оголошувати змінні, методи та класи.
– Є можливість без лишніх затрат розробляти пону версію .net, не виходячи із IDE.
– ПроРозроблені програми з використанням MS test та xUnit, можна виконувати у середовищі IDE.

До недоліків платформи відносять такі особливості:

– Складність з освоєнням інтерфейсу початківцям.
– Не зовсім якісно розроблений плагін, так як розробниками є спільнота.

Приклад інтерфейсу наведено на рисунку 3.3.

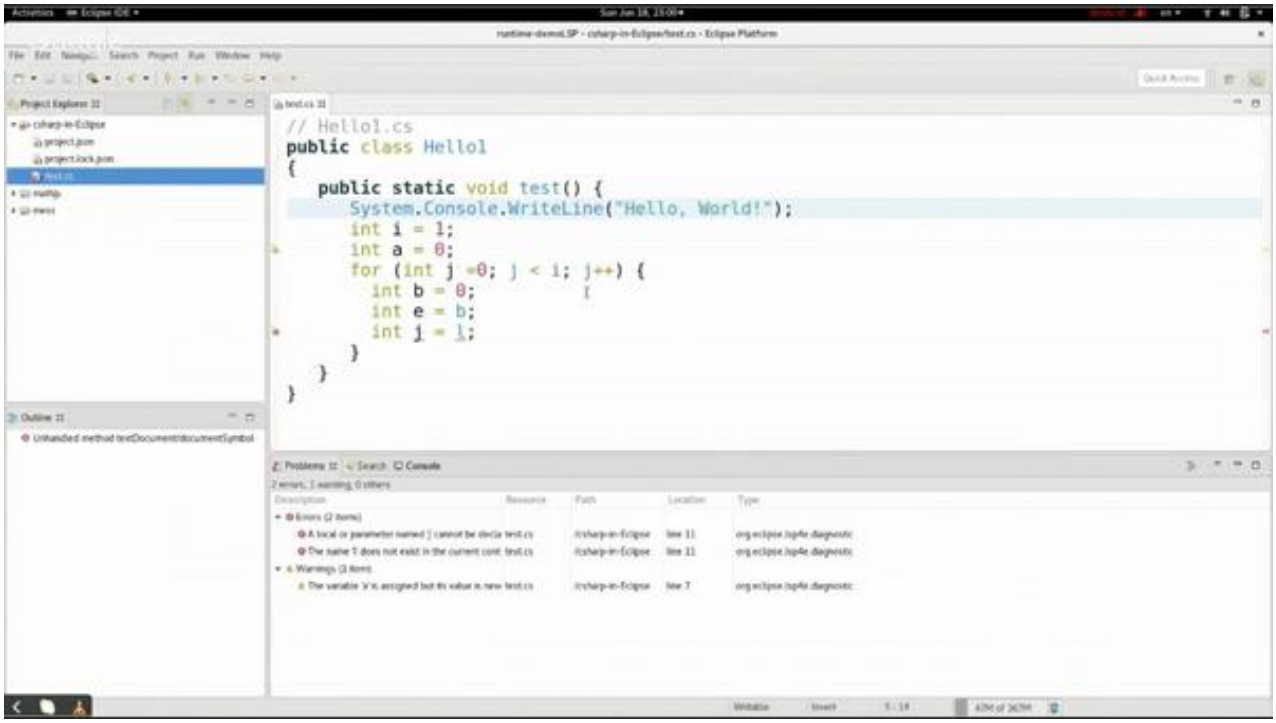


Рисунок 3.3 – Приклад інтерфейсу «Eclipse aCute»

Результати порівняння розглянутих інтегрованих середовищ розробки за обраними критеріями наведено в таблиці 3.2.

Таблиця 3.2 – Порівняння інтегрованих середовищ розробки

Критерій	Eclipse aCute	Project Rider	Visual Studio
Безкоштовність	0,5	0	0,5
Універсальність	0,5	1	1
Кросплатформність	1	1	1
Швидкість роботи	0,5	0,5	1
Невимогливість до продуктивності системи	0	0,5	0,5
Магазин розширень	0	0,5	1
Автодоповнення коду	0,5	0,5	1
Підсумковий результат	3	4	6

Отже, за результатами порівняння популярних інтегрованих середовиш розробки для мови програмування C#, найкраще підходить Microsoft Visual Studio з метою розробки бази даних для кас самообслуговування.

3.3 Програмна реалізація веб-додатку

Архітектура, яка була вибрана для розробки веб-додатку є Onion. Розробку слід розпочати з доменного рівня а саме з розробки сущностей для бази даних.

Сутність – це об’єкт, який існує. Він не повинен робити нічого, а просто існувати. У адмініструванні бази даних сутність може бути окремою річчю, особою, місцем або об’єктом. Дані про такі об’єкти можуть зберігатися. Інструмент проектування, який дозволяє адміністраторам баз даних переглядати зв’язки між кількома сутностями, називається діаграмою взаємовідносин сутностей (ERD)[22].

У адмініструванні бази даних сутністю вважаються лише ті речі, про які будуть записані або збережені дані. Якщо ніхто не збираєтеся отримувати дані про щось, немає сенсу створювати сутність у базі даних.

При розробці програм управління базами даних типи сутностей та їх властивості можуть бути представлені різними способами, наприклад:

– потрібно вибрати відому технологію (наприклад, Microsoft SQL Server, Oracle, Microsoft Access, джерело даних Microsoft ODBC тощо), як джерело даних, яке було досліджено, перевірено, стандартизовано та має велику кількість засобів керування базами даних;
– розробити власний формат бази даних і впровадити методи її обробки та взаємодійте з відомими джерелами даних у вигляді спеціальних команд, наприклад імпорт/експорт. У цьому випадку доведеться запрограмувати всю повсякденну роботу для підтримки та забезпечення надійної роботи бази даних;
– потрібно використовувати комбінацію двох вищенаведених методів. Сучасні засоби розробки програмного забезпечення мають потужний набір бібліотек для роботи зі складними колекціями та візуалізації даних у них (колекції, масиви, компоненти візуалізації тощо).

Якщо база даних реалізована в популярній СУБД то тип сутності надається таблицею. Атрибути повині відповідати полям в ER-моделі. Запис у таблиці бази даних представляє екземпляр сутності.

Усі сутності є аналогічними тим які були описані розділі 2.3, де розроблялась база даних. Реалізація сутностей показана на рисунку 3.4.

```
public class Product : BaseEntity
{
    public string ProductName { get; set; }
    public decimal Price { get; set; }

    public int ProductCategoryId { get; set; }

    public ProductCategory Category { get; set; }
    public ICollection<ReceiptDetail> ReceiptDetails { get; set; }
}

 public class Customer : BaseEntity
 {
     public int DiscountValue { get; set; }
     public int PersonId { get; set; }

     public Person Person { get; set; }
     public ICollection<Receipt> Receipts { get; set; }
 }
```

Рисунок 3.4 – Реалізація сутностей на прикладі «Продутку» та «Покупця»

Далі було розроблено інтерфейси для всіх репозиторіїв та самі репозиторії.

Репозиторій – це колекція. Колекція, яка містить сутності, і може фільтрувати та повертати результат назад залежно від вимог певної програми. Де і як він зберігає ці об'єкти є «деталями реалізації»[23]. Приклад розроблених інтерфейсів та репозиторіїв представлений на рисунку 3.5.

```
public interface IRepository<TEntity> where TEntity : BaseEntity
{
    Task<IEnumerable<TEntity>> GetAllAsync();

    Task<TEntity> GetByIdAsync(int id);

    Task AddAsync(TEntity entity);

    void Delete(TEntity entity);

    Task DeleteByIdAsync(int id);

    void Update(TEntity entity);
}

public class Repository<TEntity> : IRepository<TEntity> where TEntity : BaseEntity
{
    protected readonly TradeMarketDbContext _context;
    protected readonly DbSet<TEntity> _dbSet;
    public Repository(TradeMarketDbContext context)
    {
        _context = context;
        _dbSet = context.Set<TEntity>();
    }
    public async Task AddAsync(TEntity entity)
    {
        await _dbSet.AddAsync(entity);
    }
    public void Delete(TEntity entity)
    {
        _dbSet.Remove(entity);
    }
    public async Task DeleteByIdAsync(int id)
    {
        var entity = await _dbSet.FindAsync(id);
        _dbSet.Remove(entity);
    }
    public async Task<IEnumerable<TEntity>> GetAllAsync()
    {
        return await _dbSet.AsNoTracking().ToListAsync();
    }
    public async Task<TEntity> GetByIdAsync(int id)
    {
        return await _dbSet.FindAsync(id);
    }
    public void Update(TEntity entity)
    {
        _context.Entry(entity).State = EntityState.Modified;
    }
}
```

Рисунок 3.5 – Реалізація репозиторія та його інтерфейсу

Далі необхідно створити всі зв’язки які буду присутні між таблицями у базі даних. Які саме зв’язки розроблені у базі даних також описані у розділі 2.3. За допомогою функції OnModelCreating можна встановити всі необхідні зв’язки у базі даних. Приклад функції продемонстровано на рисунку 3.6.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .HasOne<Customer>(p => p.Customer)
        .WithOne(c => c.Person)
        .HasForeignKey<Customer>(p => p.PersonId);

    modelBuilder.Entity<Receipt>()
        .HasOne<Customer>(r => r.Customer)
        .WithMany(c => c.Receipts)
        .HasForeignKey(r => r.CustomerId);

    modelBuilder.Entity<ReceiptDetail>()
        .HasOne<Receipt>(rd => rd.Receipt)
        .WithMany(r => r.ReceiptDetails)
        .HasForeignKey(rd => rd.ReceiptId);

    modelBuilder.Entity<ReceiptDetail>()
        .HasOne<Product>(rd => rd.Product)
        .WithMany(p => p.ReceiptDetails)
        .HasForeignKey(p => p.ProductId);

    modelBuilder.Entity<Product>()
        .HasOne<ProductCategory>(p => p.Category)
        .WithMany(pc => pc.Products)
        .HasForeignKey(p => p.ProductCategoryId);
}
```

Рисунок 3.6 – Встановлення зв’язків між таблицями

Для завершення доменного рівня потрібно реалізувати міграцію для того щоб створити базу даних у MS SQL Server. Для цього нам знадобиться функціонал Entity Framework. За допомогою команди – add-migration ми створюємо міграцію і та всі необхідні для нас таблиці у базі даних. У проекті створюється папка Migrations з скриптами які відповідають за всі таблиці сущності та зв’язки. Приклад міграції продемонстровано на рисунку 3.7.

```
protected override void BuildModel(ModelBuilder modelBuilder)
{
#warning disable 612, 618
    modelBuilder
        .HasAnnotation("ProductVersion", "3.1.6")
        .HasAnnotation("Relational:MaxIdentifierLength", 128)
        .HasAnnotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn);

    modelBuilder.Entity("Data.Entities.Customer", b =>
        {
            b.Property<int>("Id")
                .ValueGeneratedOnAdd()
                .HasColumnType("int")
                .HasAnnotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn);

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

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

            b.HasKey("Id");

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

            b.ToTable("Customers");
        });
```

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Persons",
        columns: table => new
        {
            Id = table.Column<int>(nullable: false)
                .Annotation("SqlServer:Identity", "1, 1"),
            Name = table.Column<string>(nullable: true),
            Surname = table.Column<string>(nullable: true),
            BirthDate = table.Column<DateTime>(nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Persons", x => x.Id);
        });

    migrationBuilder.CreateTable(
        name: "ProductCategories",
        columns: table => new
        {
            Id = table.Column<int>(nullable: false)
                .Annotation("SqlServer:Identity", "1, 1"),
            CategoryName = table.Column<string>(nullable: true)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_ProductCategories", x => x.Id);
        });
```

Рисунок 3.7 – Створення бази даних з використанням міграцій

Далі на черзі йде сервісний рівень, який зазвичай відповідає за всю логіку нашого веб-додатку. Перше що потрібно створити це моделі всіх сущностей які були розроблені у доменному рівні, для того щоб потім за допомогою мапера можна було зіставити потрібні нам поля та мати всю необхідну інформацію про сутності. На рисунку 3.8 продемонствровано створення моделей.

```
namespace Business.Models
{
    public class ReceiptModel
    {
        public int Id { get; set; }
        public int CustomerId { get; set; }
        public DateTime OperationDate { get; set; }
        public bool IsCheckedOut { get; set; }
        public ICollection<int> ReceiptDetailsIds { get; set; }
    }
}

namespace Business.Models
{
    public class ProductModel
    {
        public int Id { get; set; }
        public int ProductCategoryId { get; set; }
        public string ProductName { get; set; }
        public string CategoryName { get; set; }
        public decimal Price { get; set; }
        public ICollection<int> ReceiptDetailIds { get; set; }
    }
}
```

Рисунок 3.8 – Реалізація моделей

Далі потрібно створити той самий мапер для зіставлення полів.

Мапінг даних – це процес зіставлення полів даних (певних елементів джерела або всього джерела) та пов'язаних з ними полів даних в іншому місці призначення. Іншими словами, це встановлення співвідношення між моделями даних, що є у різних джерелах чи системах [24]. Програмне забезпечення та інструменти мапування даних автоматично зіставляють поля даних з одного джерела даних до іншого. Використання мапінгу даних дозволяє:

– систематизувати;

– вилучати;

– аналізувати;

– розуміти величезні обсяги даних, які зберігаються у різних місцях.

Після цього можна робити виведення та виділяти цінну інформацію.

На рисунку 3.9 зображено функцію AutomapperProfile, яка і реалізує зіставлення полів для моделей.

```
public AutomapperProfile()
{
    CreateMap<Receipt, ReceiptModel>()
        .ForMember(rm => rm.ReceiptDetailsIds, r => r.MapFrom(x => x.ReceiptDetails.Select(rd => rd.Id)))
        .ReverseMap();

    CreateMap<Product, ProductModel>()
        .ForMember(pm => pm.ReceiptDetailIds, p => p.MapFrom(x => x.ReceiptDetails.Select(rd => rd.Id)))
        .ForMember(pm => pm.CategoryName, p => p.MapFrom(x => x.Category.CategoryName))
        .ReverseMap();

    CreateMap<ReceiptDetail, ReceiptDetailModel>()
        .ForMember(rd => rd.ReceiptId, r => r.MapFrom(x => x.ReceiptId))
        .ReverseMap();

    CreateMap<Customer, CustomerModel>()
        .ForMember(cm => cm.Name, c => c.MapFrom(x => x.Person.Name))
        .ForMember(cm => cm.Surname, c => c.MapFrom(x => x.Person.Surname))
        .ForMember(cm => cm.BirthDate, c => c.MapFrom(x => x.Person.BirthDate))
        .ForMember(cm => cm.ReceiptsIds, c => c.MapFrom(x => x.Receipts.Select(r => r.Id)))
        .ReverseMap();

    CreateMap<ProductCategory, ProductCategoryModel>()
        .ForMember(pm => pm.ProductIds, pc => pc.MapFrom(x => x.Products.Select(p => p.Id)))
        .ReverseMap();
}
```

Рисунок 3.9 – Реалізація мапера

Тепер необхідно створити сервіси, які містять у собі логіку та функції. За допомогою них можна виконувати різноманітні дії, такі як додавання різної інформації, виведення даних та маніпуляція з ними.

На рисунку 3.10 зображено приклад функції AddAsync яка відповідає додавання нового продукту до бази даних.

```
private readonly IUnitOfWork _uow;
private readonly IMapper _mapper;
private readonly ProductServiceValidation _productValidation = new ProductServiceValidation();

public ProductService(IUnitOfWork uow, IMapper mapper)
{
    _uow = uow;
    _mapper = mapper;
}
public async Task AddAsync(ProductModel model)
{
    _productValidation.ProductValidation(model);
    var result = _mapper.Map<ProductModel, Product>(model);
    await _uow.ProductRepository.AddAsync(result);
    await _uow.SaveAsync();
    _mapper.Map(result, model);
}
```

Рисунок 3.10 – Реалізація додавання даних до бази даних

Функція приймає товар який необхідно додати до бази даних. Далі за допомогою мапера зіставляються поля з сущностями і викликається функція первного репозиторія, який в свою чергу додає продукт до бази даних. Перед тим як додавати що-небудь у функції викликається метод ProductValidation. Він винесений в окремий клас і являє собою метод для валідації вхідних даних, які приймають наші сервіси. Приклад валідації даних на рисунку 3.11.

```
public void ProductValidation(ProductModel model)
{
    if (model == null)
    {
        throw new MarketException("Model is null!");
    }
    if (model.ProductName == null || model.ProductName == "")
    {
        throw new MarketException("Product name is null!");
    }
    if (model.Price <= 1)
    {
        throw new MarketException("Incorrect price!");
    }
}
```

Рисунок 3.11 – Приклад валідації вхідних даних

Такі валідації присутну у всіх сервісах та забезпечуються безпеку для бази даних, так як принеправильних всхідних даних валідація видасть виключення.

Ще один приклад довання відсканованого товару до чеку покупця(рисунок 3.12).

```
public async Task AddProductAsync(int productId, int receiptId, int quantity)
{
    var receipt = await _uow.ReceiptRepository.GetByIdWithDetailsAsync(receiptId);
    _receiptValidation.ReceiptValidation(receipt);
    if(receipt.ReceiptDetails == null || !receipt.ReceiptDetails.Any(x => x.ProductId == productId))
    {
        var product = await _uow.ProductRepository.GetByIdAsync(productId);
        _receiptValidation.ReceiptValidation(product);
        var receiptDetail = new ReceiptDetail()
        {
            ReceiptId = receiptId,
            ProductId = product.Id,
            Quantity = quantity,
            UnitPrice = product.Price,
            DiscountUnitPrice = product.Price - ((product.Price * receipt.Customer.DiscountValue) / 100)
        };
        await _uow.ReceiptDetailRepository.AddAsync(receiptDetail);
    }
    else
    {
        var receiptDetail = receipt.ReceiptDetails.FirstOrDefault(x => x.ProductId == productId);
        if(receiptDetail != null)
        {
            receiptDetail.Quantity += quantity;
            receiptDetail.UnitPrice = receiptDetail.UnitPrice * quantity;
            receiptDetail.DiscountUnitPrice = receiptDetail.DiscountUnitPrice * quantity;
        }
    }
    await _uow.SaveAsync();
}
```

Риснунок 3.12 – Реалізація додавання відсканово товару до чеку

Функція приймає інформацію про товар та його кількість яку потрібно додати до чеку. Алгоритм отримує всі продукти з бази даних та шукає той який потрібен саме нам. Потім перевіряє чи вже є такий товар у нашому чеку і ящо є то додає до нього необзідну кількість цього товару, а якщо немає, створю новий товар у чеку та ставить кількість в одиницях яку вказав покупець.

Також розроблена функція генерування знижок для покупців. Приклад на рисунку 3.13.

```
public async Task GenerateDiscount(int productCount, int customerId)
{
    if(productCount == null || customerId == null)
    {
        throw new Exception();
    }
    var receipts = await _uow.ReceiptRepository.GetAllWithDetailsAsync();
    var customer = await _uow.CustomerRepository.GetByIdWithDetailsAsync(customerId);
    var result = receipts.Where(x => x.CustomerId == customerId)
                         .SelectMany(x => x.ReceiptDetails)
                         .GroupBy(x => x.Product, x => x.Quantity,
                         (prod, quant) => new { Product = prod, Amount = quant.Sum() })
                         .OrderByDescending(x => x.Amount).Select(x => x.Product).Take(productCount).ToList();
    customer.DiscountValue = result.FirstOrDefault(x => x.Price) % 15;
    return customer;
}
```

Рисунок 3.13 – Реалізація генерування знижок

Функція приймає кількість товару та унікальний id покупця. Далі алгоритм отримує нашого клієнта та всі його замовлення. З них алгоритм шукає самі популярні товари та ставить їм знижку.

Тепер можна перейти до рівня інфраструктури який представляє з себе набір контролерів через які взіємодіють з веб-додатком через протоколи HTTP. Контролери приймають запит як об'єкт HttpRequestMessage, обробляють його за допомогою одного з сервісних методів і посилають у відповідь результат обробки у вигляді об'єкта HttpResponseMessage, повідосленням.

Приклад контролерів для додавання товару та категорії до бази даних продемонстровано на рисунку 3.14 – 3.15.

```
[HttpPost]
public async Task<ActionResult> AddProduct([FromBody] ProductModel value)
{
    try
    {
        await _productService.AddAsync(value);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
    return CreatedAtAction(nameof(AddProduct), value);
}
```

Рисунок 3.14 – Приклад контроллера для додавання новго товару

```
[HttpPost("categories")]
public async Task<ActionResult> AddCategory([FromBody] ProductCategoryModel value)
{
    try
    {
        await _productService.AddCategoryAsync(value);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
    return CreatedAtAction(nameof(AddCategory), value);
}
```

Рисунок 3.15 – Приклад контроллера для додавання нової категорії

Повний лістинг програмного коду наведено у додатку В.

Отже, у даному підрозділі було розглянуто програмну реалізацію основних алгоритмів роботи додатку та основних компонентів для бази даних. Було розроблено рівні архітектури Onion, а саме: доменний рівень, сервісний рівень та рівень інфраструктури. Було розроблено сутності, інтерфейси, мапер, сервіси та контролери.

3.4 Висновки

У третьому розділі було проведено аналіз та обґрунтування вибору мови програмування. Було проаналізовано переваги й недоліки C#, Python та Java. Тому для розробки веб-додатку було обрано мову C#. Також було проаналізовано варіанти вибору різноманітних IDE для розробки веб-додатку на мові C#, проаналізовано середовища розробки Project Rider, MS Visual Studio та Eclipse aCute. Виходячи з результатів порівння, було обрано середовище MS Visual Studio. Було описано основні кроки для розробки програмних компонентів веб-додатку. Продемонстровано реалізацію сущностей бази даних, процес мапінгу даних, збереження даних у репозиротріях та такі маніпуляції з базою даних як додавання даних, зміна даних та отримання даних.

4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз методів тестування веб-додатку

Тестування програмного забезпечення – це метод перевірки того, що фактичний програмний продукт відповідає очікуваним вимогам. Також необхідно переконатися, що виріб не містить дефектів. Метод передбачає виконання попередньо визначеного алгоритму з використанням ручних або автоматизованих інструментів для оцінки одного або кількох атрибутів, які цікавлять користувача. Метою тестування є виявлення помилок, прогалин або відсутніх вимог, встановлених на етапі проектування продукту [25].

Деякі вважають за краще називати тестування програмного забезпечення наступним чином:

– «Біла скринька»: коли є доступ до коду, який внаслідок тестується, прочитується (статичне тестування), запускається в дебазі, і до якого в результаті пишуться автотести;
– «Чорна скринька»: коли розробнику не відомо, як система влаштована всередині, немає доступу до коду, або відсутня можливість його прочитати, і тому є орієнтація лише на зовнішню поведінку чи ТЗ;
– «Сірий ящик»: коли розробник дивиться на код і розуміє, як він влаштований, а потім відкриває саму програму і перевіряє, як цей код відображається вже в ньому, але орієнтується більше на ТЗ (це компіляція двох вищенаведених визначень).

Простіше кажучи, тестування має на увазі перевірку програми на відповідність вимогам та стандартам якості. У даній роботі буде проведено аналіз засобів тестування та дослідження його значущості.

За потреби процес тестування має бути організований відповідно до вимог. Отже, ми можемо виділити 4 типи тестування програмного забезпечення:

– функціональне тестування;
– нефункціональне тестування;
– структурне тестування;

– тестування змін;

Функціональне тестування.

Сьогодні важко недооцінити важливість функціонального тестування, оскільки він створений для перевірки всієї функціональності системи, щоб підтвердити, що кожна функція програми працює, як задокументовано[26].

Елементи функціонального тестування:

– підготувати тестові дані згідно з описаним документом;
– вимоги в рамках функціонального тестування;
– отримати результати відповідно до специфікацій;
– пройти сам тест;
– Проаналізувати реальні та очікувані результати.

Нефункціональне тестування.

Нефункціональне тестування допоможе нам мінімізувати ризик виробництва та пов’язані з цим витрати на програмне забезпечення.

Нефункціональне тестування – це комбінація тестування продуктивності, навантаження, стресу, зручності використання та сумісності.

Нефункціональне тестування поділяється на різні частини тестування, які ми збираємося обговорити далі:

– Тестування продуктивності
– Тестування юзабіліті
– Тестування на сумісність
– Тестування юзабіліті

Інший вид нефункціонального тестування – це юзабіліті-тестування. Під час тестування зручності ми проаналізуємо зручність програми та виявимо помилки в інтерфейсі кінцевого користувача програмного забезпечення.

Тут термін зручності для користувача визначає такі аспекти програми:

– Програма повинна бути легкою для розуміння, а це означає, що всі функції повинні бути видимими для кінцевих користувачів.
– Зовнішній вигляд програми має бути хорошим, а це означає, що програма має бути приємним на вигляд і створювати відчуття для кінцевого

користувача, щоб нею користуватися.

Тестування на сумісність. Під час тестування на сумісність перевіряється функціональність програми в певних апаратних і програмних середовищах. Тільки після того, як програма стане функціонально стабільною, переходять до тестування на сумісність.

Тут програмне забезпечення означає, що можна тестувати програму в різних операційних системах та інших браузерах, а апаратне забезпечення означає, що можна тестувати програму в різних розмірах.

Метою виконання інтеграційного тестування є перевірка точності оператора між кожним модулем.

4.2 Тестування розробленого веб-додатку

Для тестування веб-додатку для кас самообслуговування було створенно низку модульних тестів, які перевіряють коректність роботи всіх алгоритмів та скриптів та інших функцій веб-додатку. На рисунку 4.1 – 4.3 зображено приклади тестів, які перевіряють функціонал різних алгоритмів.

```
[Test]
public async Task CustomerService_AddAsync_AddsModel()
{
    //arrange
    var mockUnitOfWork = new Mock<IUnitOfWork>();
    mockUnitOfWork.Setup(m => m.CustomerRepository.AddAsync(It.IsAny<Customer>()));

    var customerService = new CustomerService(mockUnitOfWork.Object, UnitTestHelper.CreateMapperProfile());
    var customer = GetTestCustomerModels.First();

    //act
    await customerService.AddAsync(customer);

    //assert
    mockUnitOfWork.Verify(x => x.CustomerRepository.AddAsync(It.Is<Customer>(x =>
                    x.Id == customer.Id && x.DiscountValue == customer.DiscountValue &&
                    x.Person.Surname == customer.Surname && x.Person.Name == customer.Name &&
                    x.Person.BirthDate == customer.BirthDate)), Times.Once);
    mockUnitOfWork.Verify(x => x.SaveAsync(), Times.Once);
}
```

Рисунок 4.1 – Тестування додавання нового користувача

```
[Test]
public async Task ReceiptService_AddAsync_AddsReceipt()
{
    //arrange
    var mockUnitOfWork = new Mock<IUnitOfWork>();
    mockUnitOfWork.Setup(m => m.ReceiptRepository.AddAsync(It.IsAny<Receipt>()));

    var receiptService = new ReceiptService(mockUnitOfWork.Object, UnitTestHelper.CreateMapperProfile());
    var receipt = GetTestReceiptsModels.First();

    //act
    await receiptService.AddAsync(receipt);

    //assert
    mockUnitOfWork.Verify(x => x.ReceiptRepository.AddAsync(It.Is<Receipt>(c => c.Id == receipt.Id)), Times.Once);
    mockUnitOfWork.Verify(x => x.SaveAsync(), Times.Once);
}
```

Рисунок 4.2 – Тестування додавання нового замовлення

```
[TestCase(1)]
[TestCase(2)]
public async Task ProductService_DeleteAsync_DeletesProduct(int id)
{
    //arrange
    var mockUnitOfWork = new Mock<IUnitOfWork>();
    mockUnitOfWork.Setup(m => m.ProductRepository.DeleteByIdAsync(It.IsAny<int>()));
    var productService = new ProductService(mockUnitOfWork.Object, UnitTestHelper.CreateMapperProfile());

    //act
    await productService.DeleteAsync(id);

    //assert
    mockUnitOfWork.Verify(x => x.ProductRepository.DeleteByIdAsync(id), Times.Once);
    mockUnitOfWork.Verify(x => x.SaveAsync(), Times.Once);
}
```

Рисунок 4.3 – Тестування видалення існуючого продукту

Для тестування коректності алгоритмів було використано фреймворк для тестування XUnit та Moq Framework. XUnit.net – це безкоштовний, орієнтований на спільноту інструмент модульного тестування з відкритим кодом для .NET Framework. xUnit – це безкоштовний інструмент тестування з відкритим вихідним кодом для .NET, який розробники використовують для написання тестів для своїх програм. По суті це середовище тестування, яке надає набір атрибутів і методів, які ми можемо використовувати для написання тестового коду для наших додатків.

Використання фреймворку Moq було необхідним для тестування алгоритмів та імітації репозиторія.

Moq це фіктивний фреймворк для C#/.NET. Він використовується в модульному тестуванні, щоб ізолювати клас, що тестується, від його залежностей і гарантувати, що викликаються правильні методи для залежних об'єктів. Говорячи простою мовою, Moq – це бібліотека, яка, коли ви включаєте її в свій проект, дає можливість легко виконувати модульне тестування. Одна функція може викликати іншу, потім іншу і так далі. Але насправді те, що нам потрібно, просто значення, що повертається з першого виклику, щоб перейти до наступного рядка. Moq допомагає ігнорувати фактичний виклик цього методу, і натомість повертається те, що повертала ця функція.

Всього було створено 24 тести зі своїми перевірками на коректну роботу розроблених алгоритмів. Результати автоматизованого тестування було продеменстровано на рисунку 4.4.

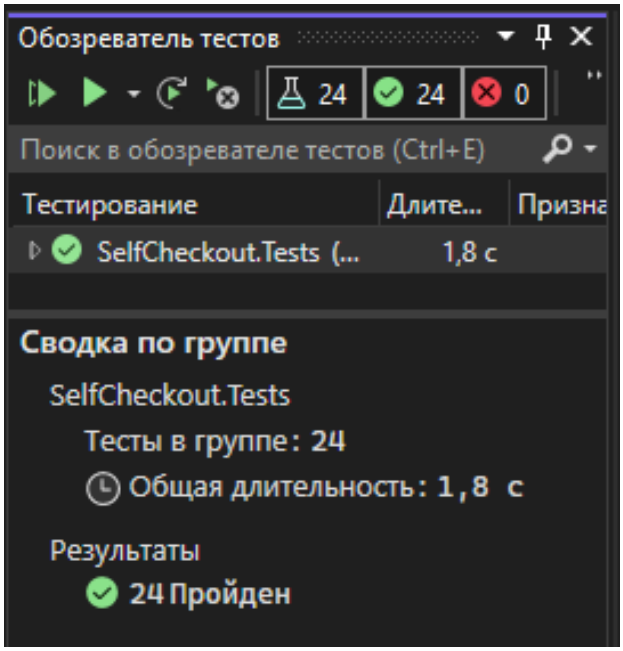


Рисунок 4.4 – Результати тестування

Отже, ми провели тестування веб-додатку для кас самообслуговування. Для перевірки коректної роботи алгоритмів було створено 24 тести з використанням фреймворків XUnit та Moq.

4.3 Розробка інструкції користувача

Для роботи з веб-додатком «Express Pay» користувачеві потрібно вибрати товари які йому потрібно купити та підійти до каси самообслуговування. На головній сторінці веб-додатку для користувача поставлений вибір між авторизацією або ж відразу відсканувати свої продукти. Приклад вікна головної сторінки зображений на рисунку 4.5.

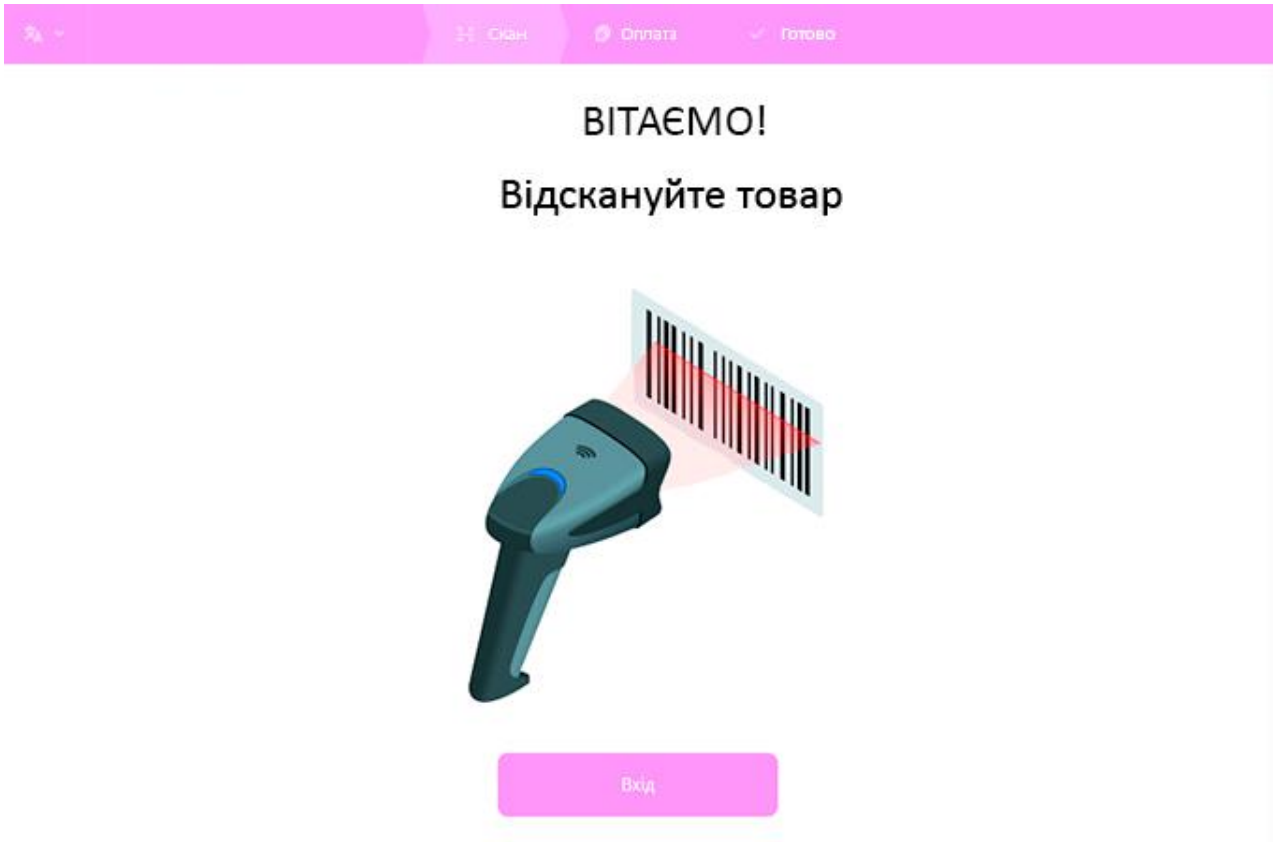


Рисунок 4.5 – Приклад вікна говного вікна

Якщо користувач натисне на кнопку вхід, перед ним відкриється вікно з авторизацією, де потрібно буде вказати свій номер телефону та пароль до свого облікового запису в системі. На цьому моменті веб-додаток посилає запит до бази даних, яка зрівнює введені користувачем дані з даними у системі. Якщо дані є коректними користувач може сканувати свої товари, але вже під своїм обліковим записом, тепер алгоритми зможуть отримати інформацію про куплені товари та згенерувати знижку. Також користувач може зареєструватися якщо ще не має облікового запису. Приклад авторизації продемонстрований на рисунку 4.6.

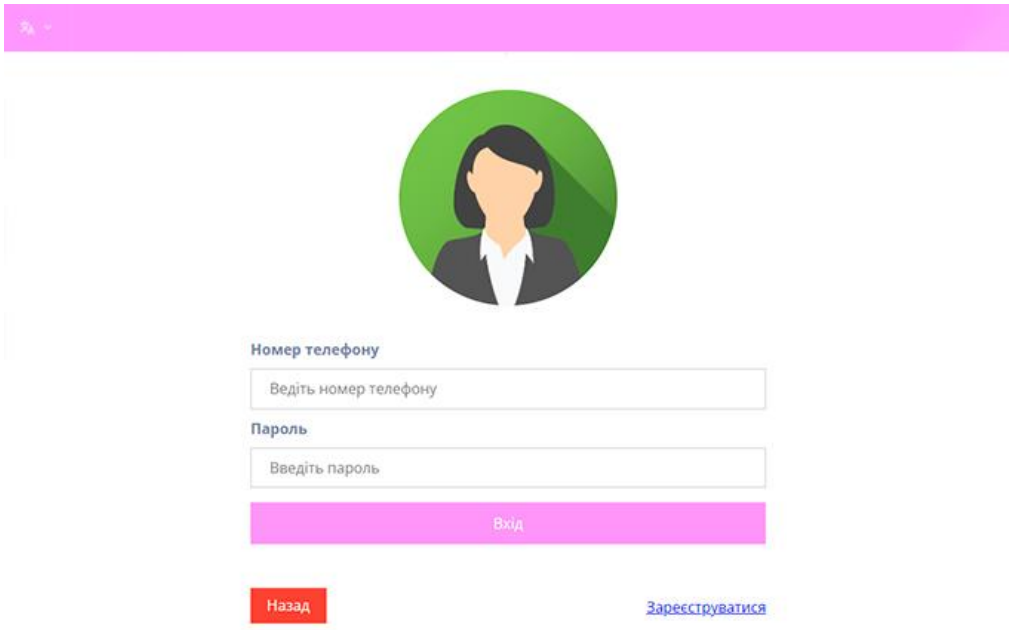


Рисунок 4.6 – Приклад вікна з авторизацією

Після авторизації та сканування товарів для користувача відкривається вікно з товарами та сумою покупок. Користувач бачить товари які він відсканував та суму за кожен окремо, а також є вікно з вибіром способу оплати. Приклад вікна з товарами зображений на рисунку 4.7.

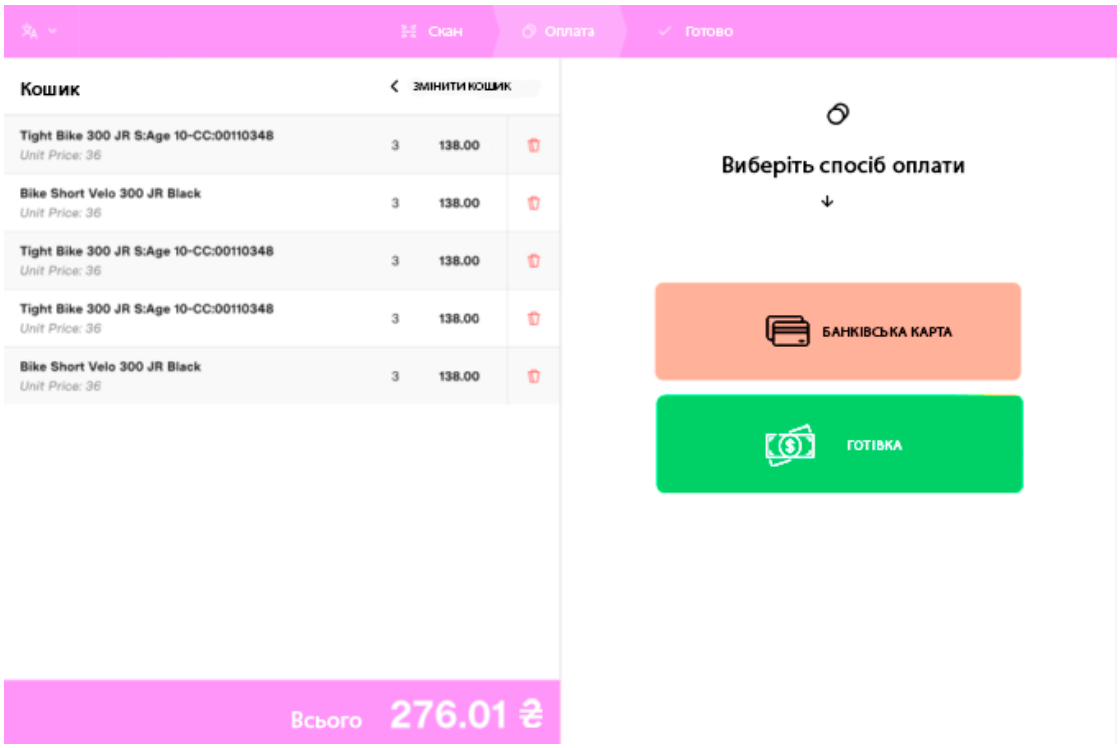


Рисунок 4.7 – Приклад вікна з товарами

Далі після вибору способу оплати користувачеві буде доступне вікно з оплатою де він зможе розрахуватись за товари. За оплатою вікдриється вікно з результатами транзакції. Приклади продемонстровані на рисунках 4.8 – 4.9.

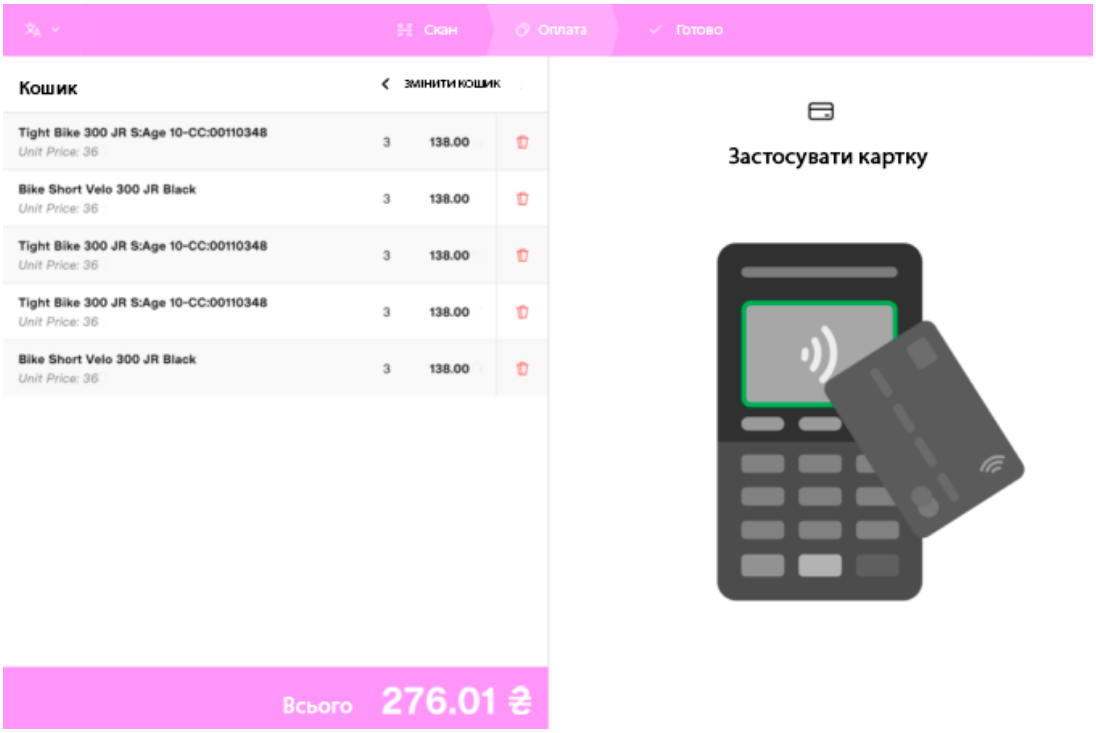


Рисунок 4.8 – Приклад вікна з оплатою

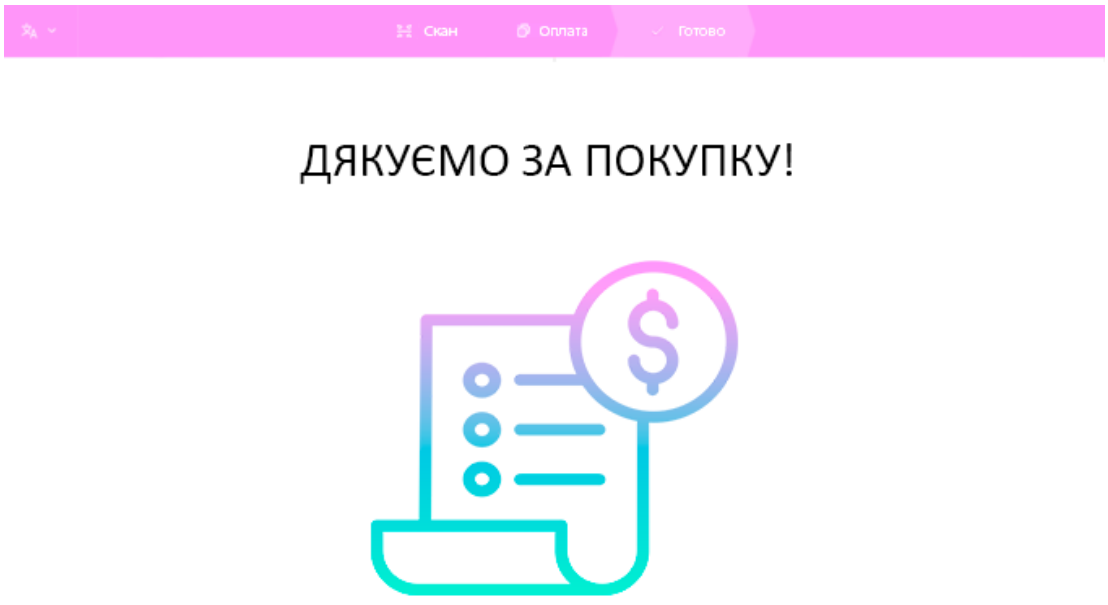


Рисунок 4.9 – Результат транзакції

Отже, було розроблено інструкцію для користувача. Докладно розглануто всі кроки для користуванням веб-додатком «Express Pay». Було розглянуто сканування товарів, реєстрація та авторизація користувачів, додавання товарів у кошик, вибір способу оплати та безпосередньо сама оплата з успішною транзакцією.

4.4 Системні вимоги

Веб-додаток «Express Pay» не є вимогливим до різноманітних систем так як працює майже з любого браузера. Він доступних для використання з різноманітних операційних систем таких як Windows, Linux та macOS. Вимоги до системи продемонстровані в табці 4.1.

Таблиця 4.1 – Вимоги до системи

Система	Конфігурація
CPU	Процесор intel core-i3 з частотою 2 ГГц
GPU	Інтегроване відеоядро intel HD Graphics
RAM	2 гб оперативної пам’яті
ОС	Windows 10, macOS X, Linux Mint

Отже, у підрозділі було наведено приклади конфігуряцій системи та вимоги до неї для коректної роботи веб-додатку «Express Pay».

4.5 Висновки

У четвертому розділі було розглянуто та проаналізовано види тестування програмних додатків. Було створено низку тестів для перевірки коректної роботи алгоритмів веб-додатку, було виконано валідацію та перевірку вхідних даних від користувача. Також було розроблено інструкцію користування веб-додатком для користувача покроково. На кінець було розглянуто системні вимоги для комфортного користування веб-додатком для кас самообслуговування.

ВИСНОВКИ

У бакалаврській дипломній роботі було розроблено веб-додаток для кас самообслуговування. Для розробки було використано середовище програмування Visual Studio 2019 та Visual Studio Code. Розроблене ПЗ призначено для підвищення швидкості та зручності процесу купівлі товарів у різних магазинах шляхом розробки програмного забезпечення для каси самообслуговування, а також підвищення попиту на більш широкий асортимент товарів роздрібної торгівлі за рахунок удосконалення алгоритму нарахування знижок.

Було проаналізовано стан даної проблеми на сучасному етапі. Розглянуто основні аналоги, визначено їхні особливості та недоліки та розроблено порівняння з власним веб-додатком.

У результаті аналізу обрано мови програмування C# та Angular для графічного інтерфейсу, бібліотеку Entity Framework для розробки бази даних.

Розроблено архітектуру веб-додатку, блок-схеми загального алгоритму роботи веб-додатку та алгоритму нарахування персональних знижок; удосконалено графічний інтерфейс веб-додатку для підвищення швикості та зручності роботи з касою самообслуговування. Розроблено базу даних для роботи з касою самообслуговування.

Було створено низку тестів для перевірки коректної роботи алгоритмів веб-додатку, проведено валідацію та перевірку вхідних даних від користувача. Також було розроблено інструкцію користування веб-додатком для користувача покроково. Визначено системні вимоги для комфортного користування веб-додатком для кас самообслуговування.

Тестування програми довело повну працездатність даного програмного продукту та відповідність поставленому завданню.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Рибак А.Ю., Романюк О.В. Особливості застосування фреймворку Angular при розробці Web-додатку. LI Науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії, Вінниця: Вінницький Національний Технічний Університет. – 2022. – URL: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki2022/paper/ view/15110/12739.

2. Каса самообслуговування URL: https://uk.wikipedia.org/wiki/self-service.

3. Переваги каси самообслуговування: URL: //nashkraj.ua/uk/blog/perevagy-kasy-samoobslugovuvannya-kso/.

4. What are self-checkout systems? URL: https://research.aimultiple.com/self-checkout/#:~:text=Self%2Dcheckouts%20(SCOs)%20are,they%20scan%20them%20through%20barcodes.

5. Програмне забезпечення кас самообслуговування URL: https://sprintingretail.com/blog/retail-self-checkout-systems/.

6. Каси самообслуговування Self-Checkout URL: https://selfservice4u.com/ua/kassa-samoosbluzhivaniya-self-checkout.

7. Роберт Мартін: Чиста архітектура: мистецтво розробки програмного забезпечення: Фабула, 2019, 416с.

8. Paul Clements, Rick Kazman: Software Architecture in Practice: Addison-Wesley Professional, 2012, 322p.

9. Mark Richards, Neal Ford: Fundamentals of Software Architecture: O'Reilly Media, Inc., 2020, 345p.

10. Архітектура та проектування програмного забезпечення URL: https://learn.ztu.edu.ua/mod/book/view.php?id=278.

11.What is Onion Architecture https://codewithmukesh.com/blog/onion-architecture-in-aspnet-core.

12. Craig Mullins: Database Administration: Addison-Wesley, 2002, 703p

13. Python Introduction URL: https://www.w3schools.com/python/python_intro.asp.

14. Java Introduction URL: https://www.w3schools.com/java/java_intro.asp.

15. Mark J. Price: C# 9 and .NET 5 - Modern Cross-Platform Development: Packt Publishing, 2020, 432p.

16. What is Entity Framework? URL: https://www.entityframeworktutorial.net /what-is-entityframework.aspx.

17. Jon Skeet: C# in Depth: Eric Lippert, 2019, 237p.

18. C# URL: https://ru.wikipedia.org/wiki/C Sharp.

19. Microsoft Visual Studio URL: https://docs.microsoft.com/ru-ru/visualstudio/ get-started/visual-studio-ide?view=vs-2022.

20. Project Rider URL: https://www.jetbrains.com/ru-ru/rider/.

21. Eclipse URL: https://www.eclipse.org/community/eclipse_newsletter/2017/ august/article3.php.

22. Основні поняття Сутностей URL: https://uadoc.zavantag.com/text/33068/ index-1.html.

23. Репозиторії та інформаційні системи URL: http://ep3.nuwm.edu.ua/9129/.

24. What is Data Mapping? URL: https://ua.talend.com/resources/data-mapping/.

25. Paul C. Jorgensen: Software Testing: A Craftsman’s Approach, Fourth Edition: Hardcover – Import, 2012, 305p.

26. Тестування програмного забезбечення URL:https://www.quality-assurance-group.com/shho-take-testuvannyaprogramnogo-zabezpechennya-ta-yake-jogoznachen-nya.

ДОДАТКИ

Додаток А. Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ

Завідувач кафедрою ПЗ

д.т.н., проф.

____________ О. Н. Романюк

31 березня 2022 р.

Технічне завдання

на бакалаврську дипломну роботу «Розробка програмного забезпечення для кас самообслуговування з використанням фреймворків .Net і Angular» за спеціальністю

121 – Інженерія програмного забезпечення

Керівник бакалаврської дипломної роботи:

______

"__" __ 2022 р.

к.т.н., доцент О.В. Бомонак

студент гр. 2ПІ-18б А.Ю. "__" __ 2022 р.

В

і

н

н

1. Найменування та галузь застосування

Бакалаврська дипломна робота: «Розробка програмного забезпечення для кас самообслуговування з використанням фреймворків .Net і Angular».

Галузь застосування – розробка програмного забезпечення для сфери роздрібної торгівлі.

2. Підстава для розробки.

Підставою для виконання бакалаврської дипломної роботи (БДР) є індивідуальне завдання на БДР та наказ № 66 від 24 березня 2022 р. ректора по ВНТУ про закріплення тем БДР.

3. Мета та призначення розробки.

Метою дослідження є підвищення швидкості та зручності процесу купівлі товарів у різних магазинах шляхом розробки програмного забезпечення для каси самообслуговування, а також підвищення попиту на більш широкий асортимент товарів роздрібної торгівлі за рахунок удосконалення алгоритму нарахування знижок.

Призначення роботи – розробка та програмна реалізація веб-додатку для кас самообслуговування.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БДР.

1. Роберт Мартін: Чиста архітектура: мистецтво розробки програмного забезпечення: Фабула, 2019, 416с.

2. Paul Clements, Rick Kazman: Software Architecture in Practice: Addison-Wesley Professional, 2012, 322p.

3. Mark Richards, Neal Ford: Fundamentals of Software Architecture: O'Reilly Media, Inc., 2020, 345p.

4. Jon Skeet: C# in Depth: Eric Lippert, 2019, 237p.

5. C# URL: https://ru.wikipedia.org/wiki/C Sharp.

6. Paul C. Jorgensen: Software Testing: A Craftsman’s Approach, Fourth Edition: Hardcover – Import, 2012, 305p.

5. Технічні вимоги

Мова програмування – C#;

Середовище розробки – MS Visual Studio, MS Visual Code;

Базада них – MS SQL Server;

Використані фреймворки – .Net, Angular.

6. Конструктивні вимоги

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред’являється по закінченню робіт:

1. Пояснювальна записка до БДР;

2. Технічне завдання;

3. Лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз завдання і вибір методу вирішення поставленої задачі дослідження	26.03.2022 - 03.04.2022	Вик.
2	Розробка загальної архітектури веб-додатку та бази даних	04.04.2022 - 14.04.2022	Вик.
3	Розробка загального алгоритму роботи веб-додаткку та алгоритму нарахування персональних знижок для користувачів	15.04.2022 - 28.04.2022	Вик.
4	Вибір середовища та мови розробки	29.04.2022 - 03.05.2022	Вик.
5	Програмна реалізація додатку для кас самооблуговування	04.05.2022 - 24.05.2022	Вик.
6	Тестування програми	25.05.2022 - 30.05.2022	Вик.
7	Оформлення матеріалів до захисту БДР	31.05.2022 - 10.06.2022	Вик.

10. Порядок контролю та прийняття

Виконання етапів бакалаврської дипломної роботи контролюється керівником згідно з графіком виконання роботи.

Прийняття бакалаврської дипломної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

Додаток Б. Протокол перевірки кваліфікаційної роботи
на наявність текстових запозичень

ПРОТОКОЛ
ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: «Розробка програмного забезпечення для кас самообслуговування з використанням фреймворків .Net і Angular»

Тип роботи: БДР

Підрозділ : кафедра програмного забезпечення, ФІТКІ

Науковий керівник: Романюк О.В.

Оригінальність	**93,5%**
Схожість	6,5%

Аналіз звіту подібності

■ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Черноволик Г. О.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck

Автор роботи _____ Рибак А.Ю.

Керівник роботи _____ Романюк О.В.

Додаток В. Лістинг програми

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Data.Entities
{
    public class Product : BaseEntity
    {
        public string ProductName { get; set; }
        public decimal Price { get; set; }

        public int ProductCategoryId { get; set; }

        public ProductCategory Category { get; set; }
        public ICollection<ReceiptDetail> ReceiptDetails { get; set; }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Data.Entities
{
```

```
    public class Person : BaseEntity
    {
        public string Name { get; set; }
        public string Surname { get; set; }
        public DateTime BirthDate { get; set; }

        public Customer Customer { get; set; }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Data.Entities
{
    public class Customer : BaseEntity
    {
        public int DiscountValue { get; set; }
        public int PersonId { get; set; }

        public Person Person { get; set; }
        public ICollection<Receipt> Receipts { get; set; }
    }
}

using System;
using System.Collections.Generic;
```

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

namespace Data.Entities
{
    public class ProductCategory : BaseEntity
    {
        public string CategoryName { get; set; }

        public ICollection<Product> Products { get; set; }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Data.Entities
{
    public class Receipt : BaseEntity
    {
        public DateTime OperationDate { get; set; }
        public bool IsCheckedOut { get; set; }

        public int CustomerId { get; set; }

        public Customer Customer { get; set; }
```

```
        public ICollection<ReceiptDetail> ReceiptDetails { get; set; }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Data.Entities
{
    public class ReceiptDetail : BaseEntity
    {
        public decimal DiscountUnitPrice { get; set; }
        public decimal UnitPrice { get; set; }
        public int Quantity { get; set; }

        public int ReceiptId { get; set; }
        public int ProductId { get; set; }

        public Receipt Receipt { get; set; }
        public Product Product { get; set; }
    }
}

using Data.Interfaces;
using Data.Entities;
using Data.Data;
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace Data.Repositories
{
    public class Repository<TEntity> : IRepository<TEntity> where TEntity : BaseEntity
    {
        protected readonly TradeMarketDbContext _context;
        protected readonly DbSet<TEntity> _dbSet;
        public Repository(TradeMarketDbContext context)
        {
            _context = context;
            _dbSet = context.Set<TEntity>();
        }
        public async Task AddAsync(TEntity entity)
        {
            await _dbSet.AddAsync(entity);
        }
        public void Delete(TEntity entity)
        {
            _dbSet.Remove(entity);
        }
        public async Task DeleteByIdAsync(int id)
        {
            var entity = await _dbSet.FindAsync(id);
            _dbSet.Remove(entity);
        }
```

```
        public async Task<IEnumerable<TEntity>> GetAllAsync()
        {
            return await _dbSet.AsNoTracking().ToListAsync();
        }
        public async Task<TEntity> GetByIdAsync(int id)
        {
            return await _dbSet.FindAsync(id);
        }
        public void Update(TEntity entity)
        {
            _context.Entry(entity).State = EntityState.Modified;
        }
    }
}

using Data.Data;
using Data.Entities;
using Data.Interfaces;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Data.Repositories
{
    public class CustomerRepository : Repository<Customer>, ICustomerRepository
    {
        public CustomerRepository(TradeMarketDbContext context) : base(context)
```

```
    {
    }

    public async Task<IEnumerable<Customer>> GetAllWithDetailsAsync()
    {
        return await _context.Customers.Include(x => x.Receipts).ThenInclude(i => i.ReceiptDetails)
                                .Include(x => x.Person)
                                .ToListAsync();
    }

    public async Task<Customer> GetByIdWithDetailsAsync(int id)
    {
        return await _context.Customers.Include(x => x.Receipts).ThenInclude(x => x.ReceiptDetails)
                                .Include(x => x.Person)
                                .FirstOrDefaultAsync(x => x.Id == id);
    }
  }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Data.Data;
using Data.Entities;
using Data.Interfaces;
using Microsoft.EntityFrameworkCore;
```

```
namespace Data.Repositories
{
   public class ProductRepository : Repository<Product>, IProductRepository
   {
      public ProductRepository(TradeMarketDbContext context) : base(context)
      {
      }

      public async Task<IEnumerable<Product>> GetAllWithDetailsAsync()
      {
         return await _context.Products.Include(x => x.Category)
                              .Include(x => x.ReceiptDetails)
                              .ToListAsync();
      }
      public async Task<Product> GetByIdWithDetailsAsync(int id)
      {
         return await _context.Products.Include(x => x.Category)
                              .Include(x => x.ReceiptDetails)
                              .FirstOrDefaultAsync(x => x.Id == id);
      }
   }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Data.Data;
```

```
using Data.Entities;
using Data.Interfaces;
using Microsoft.EntityFrameworkCore;

namespace Data.Repositories
{
   public class ReceiptRepository : Repository<Receipt>, IReceiptRepository
   {
      public ReceiptRepository(TradeMarketDbContext context) : base(context)
      {
      }

      public async Task<IEnumerable<Receipt>> GetAllWithDetailsAsync()
      {
         return await _context.Receipts.Include(x => x.ReceiptDetails).ThenInclude(i =>
i.Product)
                              .Include(x => x.ReceiptDetails).ThenInclude(i =>
i.Product.Category)
                              .Include(x => x.Customer)
                              .ToListAsync();
      }
      public async Task<Receipt> GetByIdWithDetailsAsync(int id)
      {
         return await _context.Receipts.Include(x => x.ReceiptDetails).ThenInclude(i =>
i.Product)
                              .Include(x => x.ReceiptDetails).ThenInclude(i =>
i.Product.Category)
                              .Include(x => x.Customer)
                              .FirstOrDefaultAsync(x => x.Id == id);
      }
```

```
    }
}

// <auto-generated />
using System;
using Data.Data;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;

namespace Data.Migrations
{
    [DbContext(typeof(TradeMarketDbContext))]
    partial class TradeMarketDbContextModelSnapshot : ModelSnapshot
    {
        protected override void BuildModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("ProductVersion", "3.1.6")
                .HasAnnotation("Relational:MaxIdentifierLength", 128)
                .HasAnnotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn);

            modelBuilder.Entity("Data.Entities.Customer", b =>
                {
                    b.Property<int>("Id")
                        .ValueGeneratedOnAdd()
                        .HasColumnType("int")
```

```
                .HasAnnotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn);

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

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

            b.HasKey("Id");

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

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

        modelBuilder.Entity("Data.Entities.Person", b =>
        {
            b.Property<int>("Id")
                .ValueGeneratedOnAdd()
                .HasColumnType("int")
                .HasAnnotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn);

            b.Property<DateTime>("BirthDate")
                .HasColumnType("datetime2");

            b.Property<string>("Name")
                .HasColumnType("nvarchar(max)");
```

```
            b.Property<string>("Surname")
                .HasColumnType("nvarchar(max)");

            b.HasKey("Id");

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

        modelBuilder.Entity("Data.Entities.Product", b =>
        {
            b.Property<int>("Id")
                .ValueGeneratedOnAdd()
                .HasColumnType("int")
                .HasAnnotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn);

            b.Property<decimal>("Price")
                .HasColumnType("decimal(18,2)");

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

            b.Property<string>("ProductName")
                .HasColumnType("nvarchar(max)");

            b.HasKey("Id");

            b.HasIndex("ProductCategoryId");
```

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

        modelBuilder.Entity("Data.Entities.ProductCategory", b =>
        {
            b.Property<int>("Id")
                .ValueGeneratedOnAdd()
                .HasColumnType("int")
                .HasAnnotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn);

            b.Property<string>("CategoryName")
                .HasColumnType("nvarchar(max)");

            b.HasKey("Id");

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

        modelBuilder.Entity("Data.Entities.Receipt", b =>
        {
            b.Property<int>("Id")
                .ValueGeneratedOnAdd()
                .HasColumnType("int")
                .HasAnnotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn);

            b.Property<int>("CustomerId")
                .HasColumnType("int");
```

```
                b.Property<bool>("IsCheckedOut")
                    .HasColumnType("bit");

                b.Property<DateTime>("OperationDate")
                    .HasColumnType("datetime2");

                b.HasKey("Id");

                b.HasIndex("CustomerId");

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

        modelBuilder.Entity("Data.Entities.ReceiptDetail", b =>
            {
                b.Property<int>("Id")
                    .ValueGeneratedOnAdd()
                    .HasColumnType("int")
                    .HasAnnotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn);

                b.Property<double>("DiscountUnitPrice")
                    .HasColumnType("float");

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

                b.Property<int>("Quantity")
                    .HasColumnType("int");
```

```
            b.Property<int>("ReceiptId")
                .HasColumnType("int");

            b.Property<double>("UnitPrice")
                .HasColumnType("float");

            b.HasKey("Id");

            b.HasIndex("ProductId");

            b.HasIndex("ReceiptId");

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

    modelBuilder.Entity("Data.Entities.Customer", b =>
        {
            b.HasOne("Data.Entities.Person", "Person")
                .WithOne("Customer")
                .HasForeignKey("Data.Entities.Customer", "PersonId")
                .OnDelete(DeleteBehavior.Cascade)
                .IsRequired();
        });

    modelBuilder.Entity("Data.Entities.Product", b =>
        {
            b.HasOne("Data.Entities.ProductCategory", "Category")
                .WithMany("Products")
                .HasForeignKey("ProductCategoryId")
                .OnDelete(DeleteBehavior.Cascade)
```

```
                .IsRequired();
            });

        modelBuilder.Entity("Data.Entities.Receipt", b =>
            {
                b.HasOne("Data.Entities.Customer", "Customer")
                    .WithMany("Receipts")
                    .HasForeignKey("CustomerId")
                    .OnDelete(DeleteBehavior.Cascade)
                    .IsRequired();
            });

        modelBuilder.Entity("Data.Entities.ReceiptDetail", b =>
            {
                b.HasOne("Data.Entities.Product", "Product")
                    .WithMany("ReceiptDetails")
                    .HasForeignKey("ProductId")
                    .OnDelete(DeleteBehavior.Cascade)
                    .IsRequired();

                b.HasOne("Data.Entities.Receipt", "Receipt")
                    .WithMany("ReceiptDetails")
                    .HasForeignKey("ReceiptId")
                    .OnDelete(DeleteBehavior.Cascade)
                    .IsRequired();
            });
#pragma warning restore 612, 618
        }
    }
}
```

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

namespace Data.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Persons",
                columns: table => new
                {
                    Id = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Name = table.Column<string>(nullable: true),
                    Surname = table.Column<string>(nullable: true),
                    BirthDate = table.Column<DateTime>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Persons", x => x.Id);
                });

            migrationBuilder.CreateTable(
                name: "ProductCategories",
                columns: table => new
                {
                    Id = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
```

```
        CategoryName = table.Column<string>(nullable: true)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_ProductCategories", x => x.Id);
    });

migrationBuilder.CreateTable(
    name: "Customers",
    columns: table => new
    {
        Id = table.Column<int>(nullable: false)
            .Annotation("SqlServer:Identity", "1, 1"),
        DiscountValue = table.Column<int>(nullable: false),
        PersonId = table.Column<int>(nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Customers", x => x.Id);
        table.ForeignKey(
            name: "FK_Customers_Persons_PersonId",
            column: x => x.PersonId,
            principalTable: "Persons",
            principalColumn: "Id",
            onDelete: ReferentialAction.Cascade);
    });

migrationBuilder.CreateTable(
    name: "Products",
    columns: table => new
```

```
    {
        Id = table.Column<int>(nullable: false)
            .Annotation("SqlServer:Identity", "1, 1"),
        ProductName = table.Column<string>(nullable: true),
        Price = table.Column<decimal>(nullable: false),
        ProductCategoryId = table.Column<int>(nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Products", x => x.Id);
        table.ForeignKey(
            name: "FK_Products_ProductCategories_ProductCategoryId",
            column: x => x.ProductCategoryId,
            principalTable: "ProductCategories",
            principalColumn: "Id",
            onDelete: ReferentialAction.Cascade);
    });

migrationBuilder.CreateTable(
    name: "Receipts",
    columns: table => new
    {
        Id = table.Column<int>(nullable: false)
            .Annotation("SqlServer:Identity", "1, 1"),
        OperationDate = table.Column<DateTime>(nullable: false),
        IsCheckedOut = table.Column<bool>(nullable: false),
        CustomerId = table.Column<int>(nullable: false)
    },
    constraints: table =>
    {
```

```
            table.PrimaryKey("PK_Receipts", x => x.Id);
            table.ForeignKey(
                name: "FK_Receipts_Customers_CustomerId",
                column: x => x.CustomerId,
                principalTable: "Customers",
                principalColumn: "Id",
                onDelete: ReferentialAction.Cascade);
        });

    migrationBuilder.CreateTable(
        name: "ReceiptsDetails",
        columns: table => new
        {
            Id = table.Column<int>(nullable: false)
                .Annotation("SqlServer:Identity", "1, 1"),
            DiscountUnitPrice = table.Column<double>(nullable: false),
            UnitPrice = table.Column<double>(nullable: false),
            Quantity = table.Column<int>(nullable: false),
            ReceiptId = table.Column<int>(nullable: false),
            ProductId = table.Column<int>(nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_ReceiptsDetails", x => x.Id);
            table.ForeignKey(
                name: "FK_ReceiptsDetails_Products_ProductId",
                column: x => x.ProductId,
                principalTable: "Products",
                principalColumn: "Id",
                onDelete: ReferentialAction.Cascade);
```

```
            table.ForeignKey(
                name: "FK_ReceiptsDetails_Receipts_ReceiptId",
                column: x => x.ReceiptId,
                principalTable: "Receipts",
                principalColumn: "Id",
                onDelete: ReferentialAction.Cascade);
        });

    migrationBuilder.CreateIndex(
        name: "IX_Customers_PersonId",
        table: "Customers",
        column: "PersonId",
        unique: true);

    migrationBuilder.CreateIndex(
        name: "IX_Products_ProductCategoryId",
        table: "Products",
        column: "ProductCategoryId");

    migrationBuilder.CreateIndex(
        name: "IX_Receipts_CustomerId",
        table: "Receipts",
        column: "CustomerId");

    migrationBuilder.CreateIndex(
        name: "IX_ReceiptsDetails_ProductId",
        table: "ReceiptsDetails",
        column: "ProductId");

    migrationBuilder.CreateIndex(
```

```
            name: "IX_ReceiptsDetails_ReceiptId",
            table: "ReceiptsDetails",
            column: "ReceiptId");
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "ReceiptsDetails");

            migrationBuilder.DropTable(
                name: "Products");

            migrationBuilder.DropTable(
                name: "Receipts");

            migrationBuilder.DropTable(
                name: "ProductCategories");

            migrationBuilder.DropTable(
                name: "Customers");

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

Додаток Г. Графічна частина

ГРАФІЧНА ЧАСТИНА

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ КАС САМООБСЛУГОВУВАННЯ З ВИКОРИСТАННЯМ ФРЕЙМВОРКІВ .NET І ANGULAR

Вінницький національний технічний університет

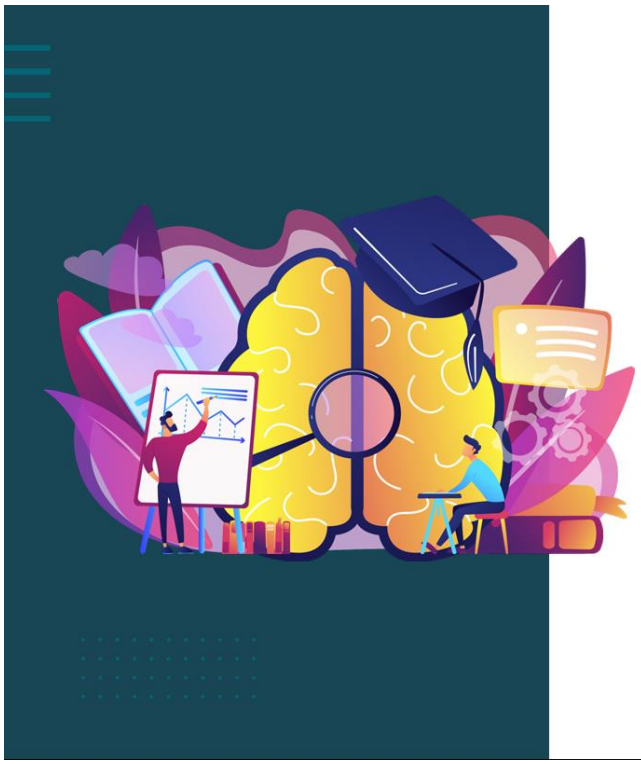
Бакалаврська дипломна робота

На тему:Розробка програмного забезпечення для кас самообслуговування з використанням фреймворків .Net та Angular

Виконав:
Студент групи 2ПІ-18б
Рибак А.Ю.

Науковий керівник:
к.т.н доцент кафедри ПЗ
Романюк О.В.

Рисунок Г.1 – Титульний лист



Актуальність теми

- Для вирішення проблем, які виникають з чергами у різноманітних точках роздрібної торгівлі, не так давно в них було встановлено каси самообслуговування. Вони безперечно полегшують покупки покупцям та зменшують черги до продавців, які працюють безпосередньо на касах, але також мають свої недоліки.
- Функціональні та візуальні недоліки можуть відбити бажання у покупця ними користуватися. Наприклад недостатньо швидка робота самої каси або неприємний чи незрозумілий інтерфейс, або нічим не примітне програмне забезпечення яке не може заманити скористуватися саме касою самообслуговування.
- Тому актуальною є задача підвищення попиту на каси самообслуговування шляхом розробки власного веб-додатку, який має вдосконалені алгоритми та візуальну частину.

Рисунок Г.2 – Актуальність теми

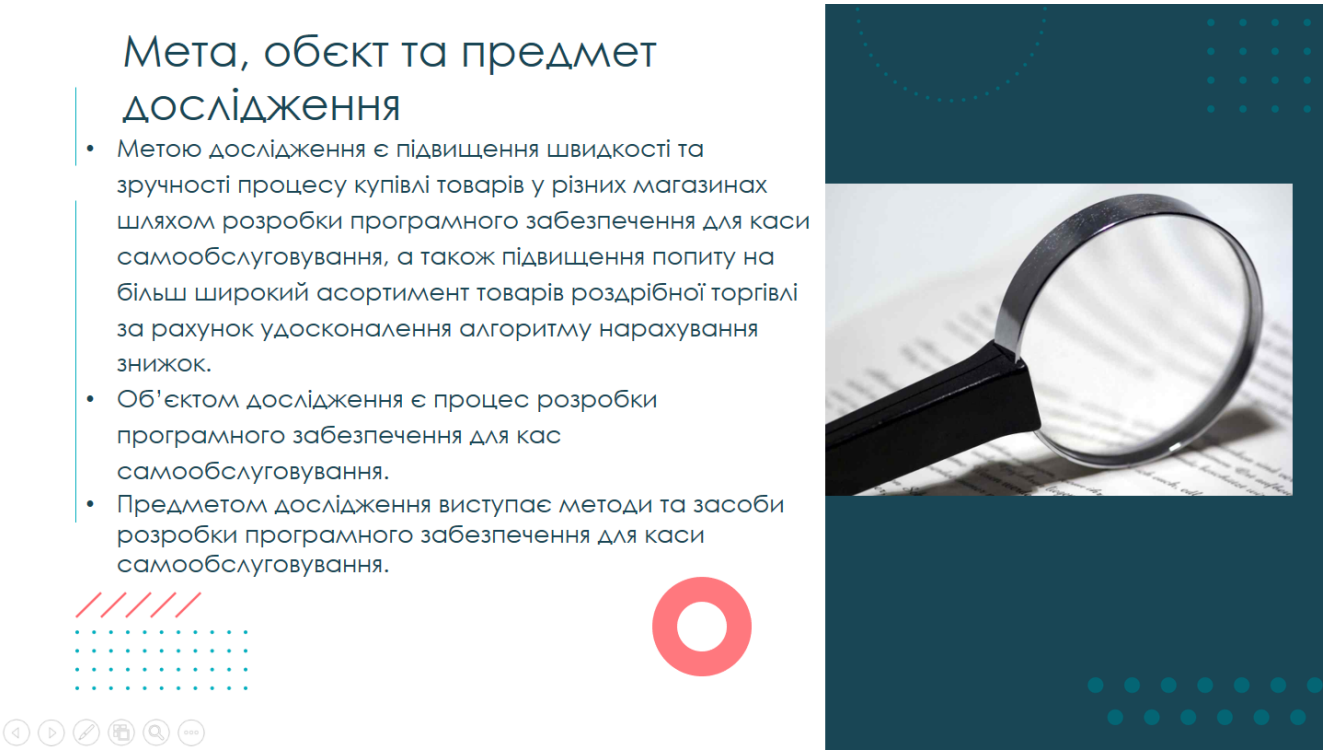


Рисунок Г.3 – Мета, об’єкт та предмет дослідження

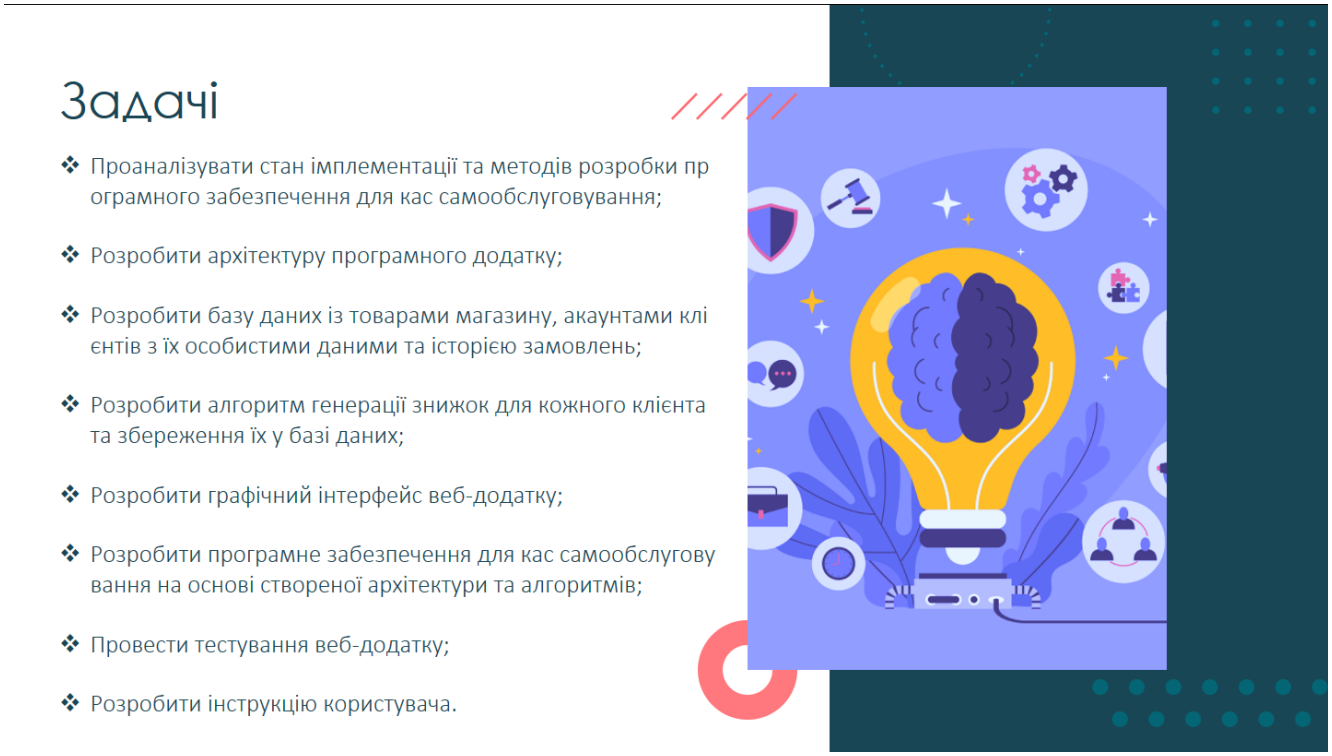


Рисунок Г.4 – Задачі дослідження

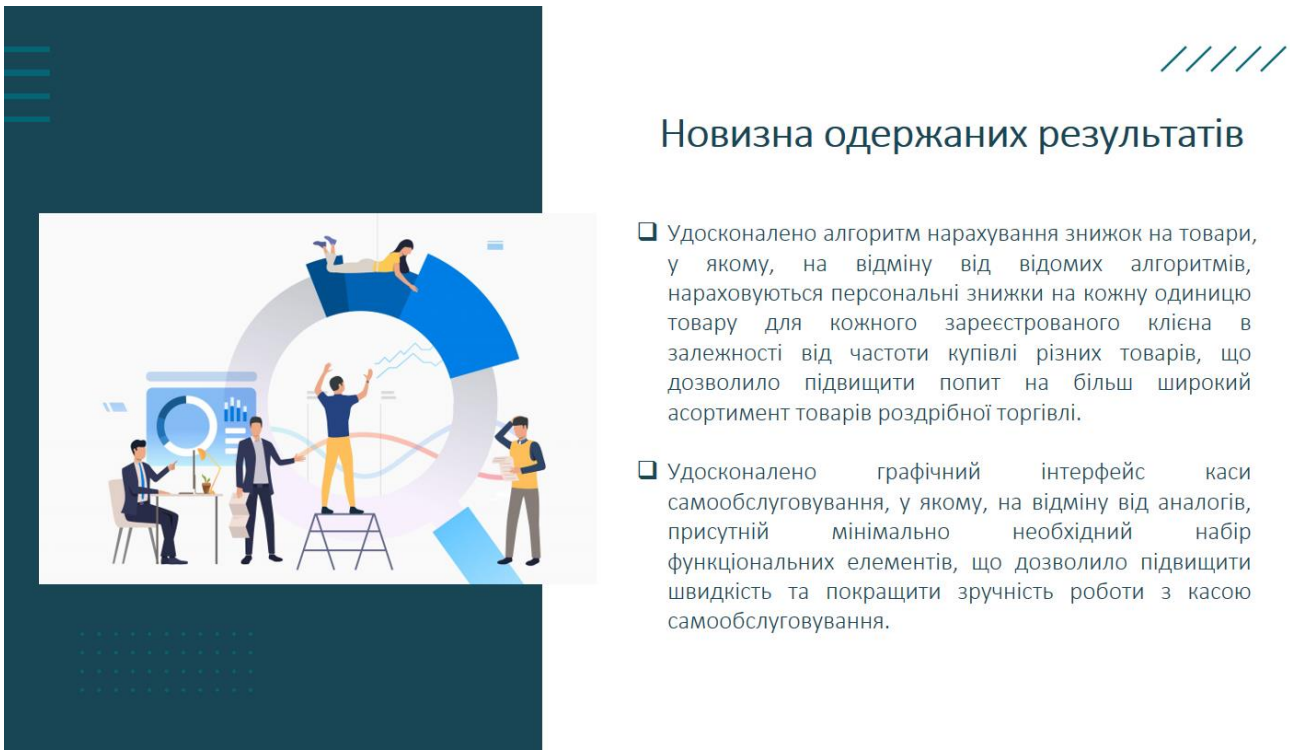


Рисунок Г.5 – Новизна одержаних результатів

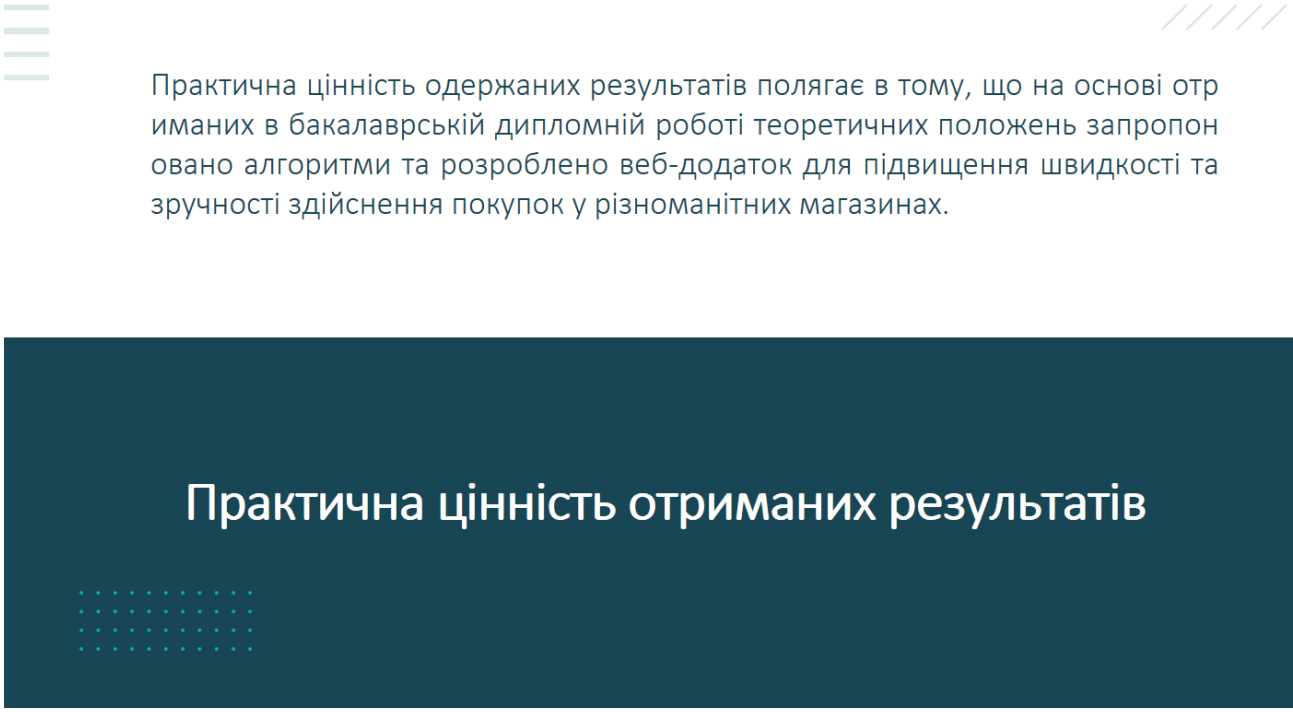


Рисунок Г.6 – Практична цінність одержаних результатів

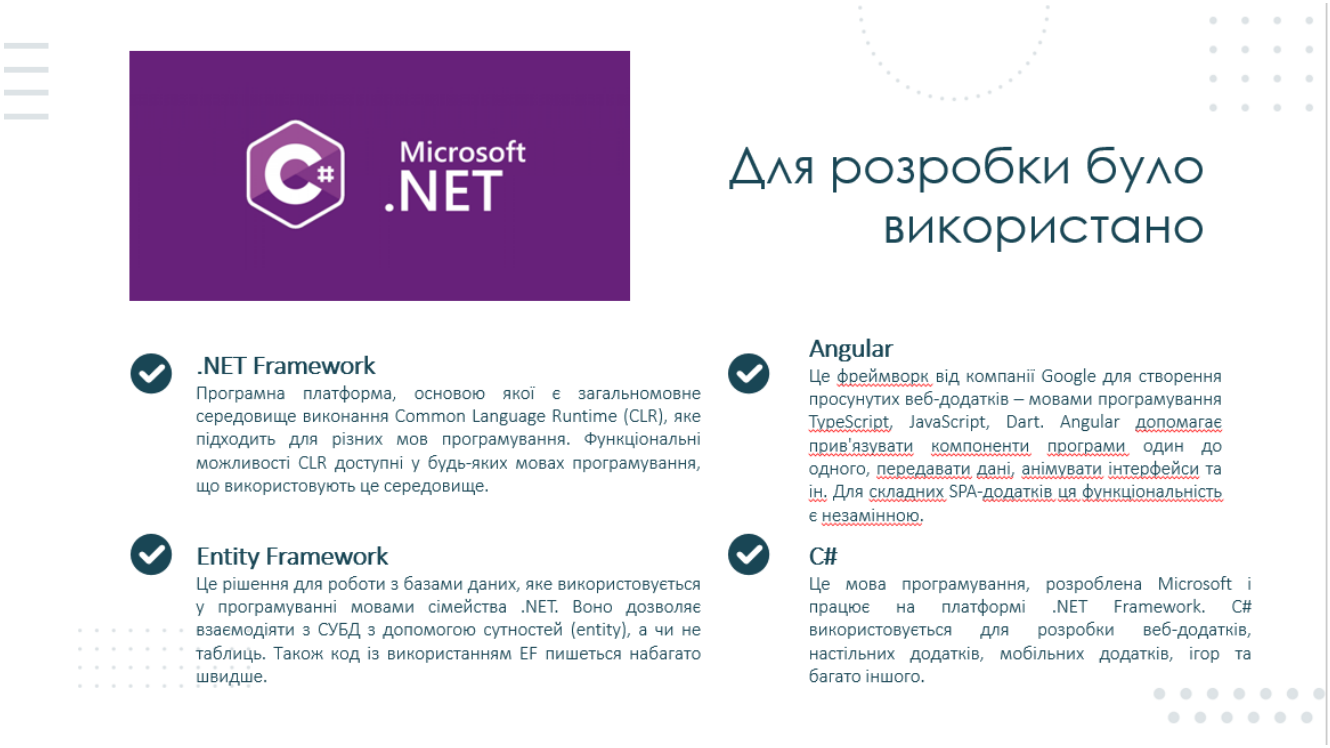


Рисунок Г.7 – Використані фреймворки

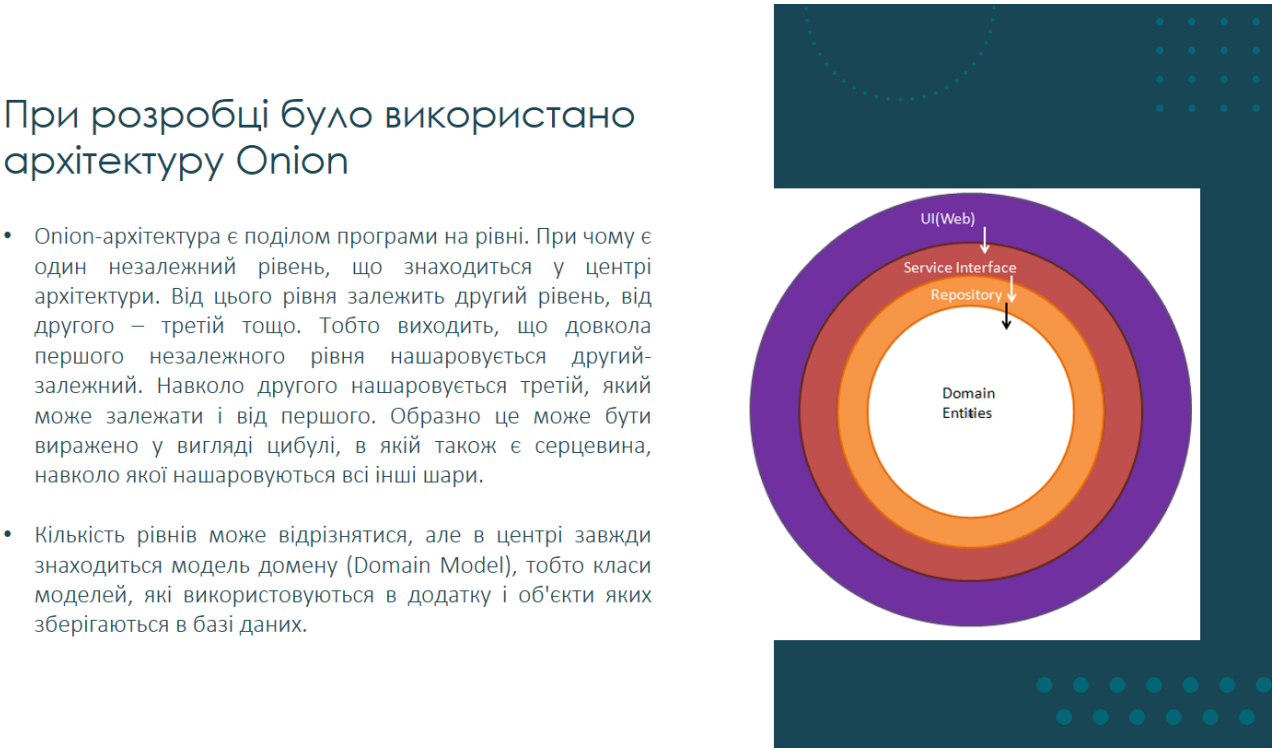


Рисунок Г.8 – Використання архітектури Onion

Для збереження інформації було використано Microsoft SQL Server

- Microsoft SQL Server — система управління базами даних, яка розробляється корпорацією Microsoft. Як сервер даних виконує головну функцію по збереженню та наданню даних у відповідь на запити інших застосунків, які можуть виконуватися як на тому ж самому сервері, так і у мережі.

- Мова, що використовується для запитів — Transact-SQL. Використовується як для невеликих і середніх за розміром баз даних, так і для великих баз даних масштабу підприємства. Багато років вдало конкурує з іншими системами керування базами даних. Тому у бакалаврській роботі чудово підходить для збереження всієї необхідної інформації.

Рисунок Г.9 – Використання Microsoft SQL Server

Схема бази даних

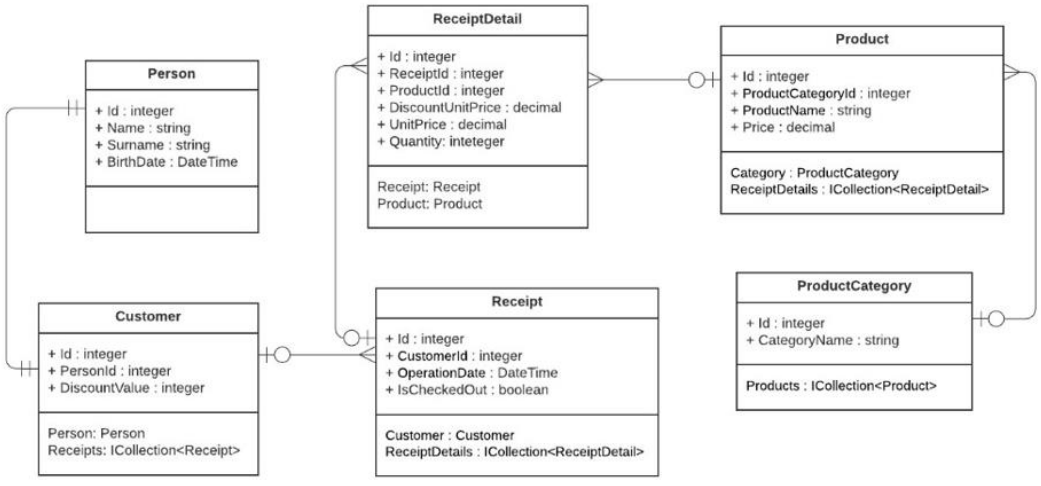
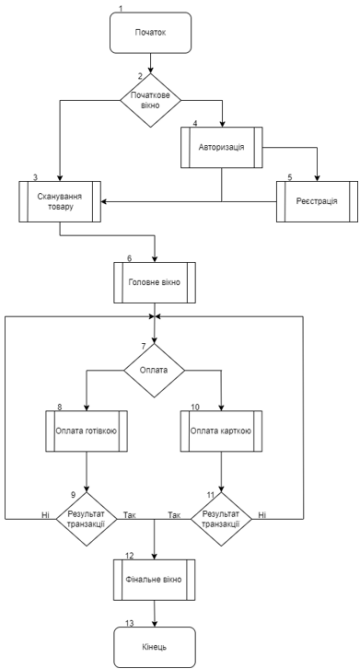


Рисунок Г.10 – Схема бази даних



Блок-схема загального алгоритму веб-додатку

Рисунок Г.11 – Блок-схема загального алгоритму

Блок-схема алгоритму генерування знижок

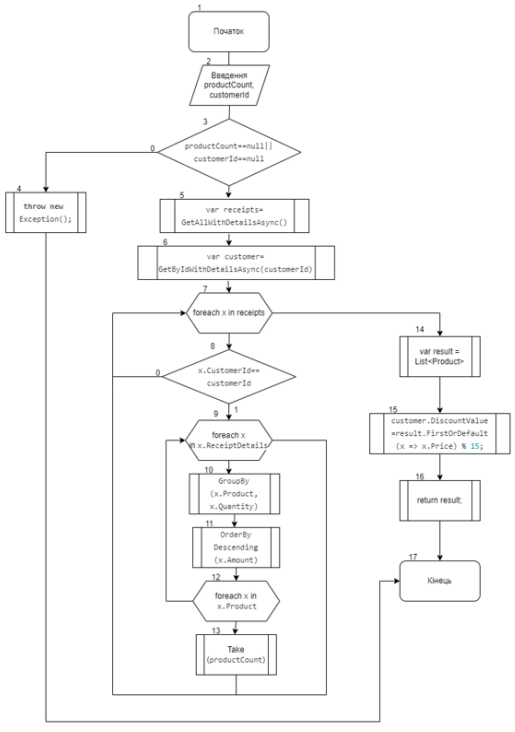


Рисунок Г.12 – Блок-схема алгоритму генерування знижок

Тестування веб-додатку

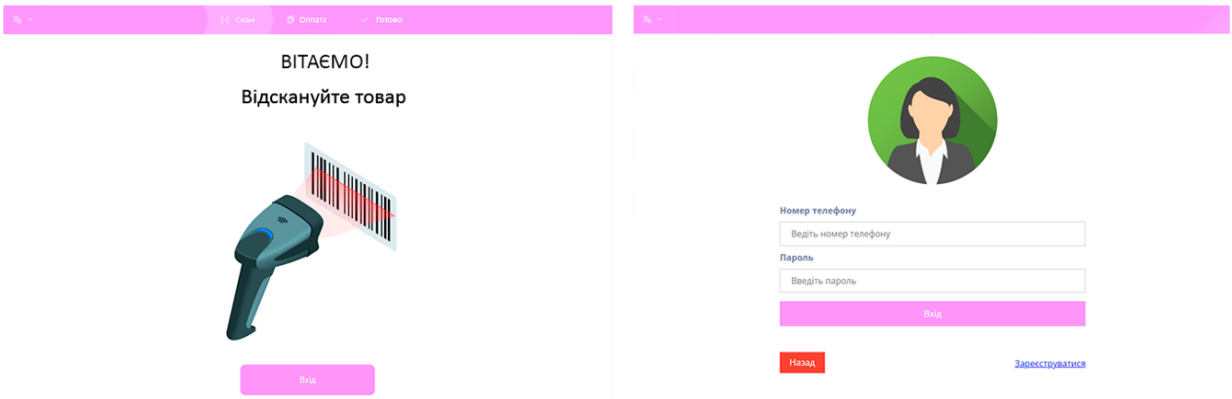


Рисунок Г.13 – Тестування веб-додатку

Тестування веб-додатку

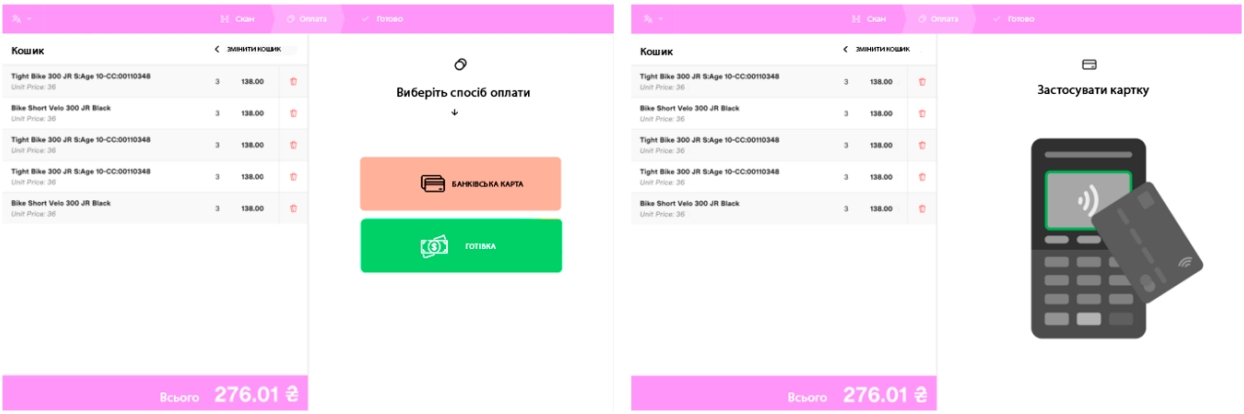


Рисунок Г.14 – Тестування веб-додатку(продовження)

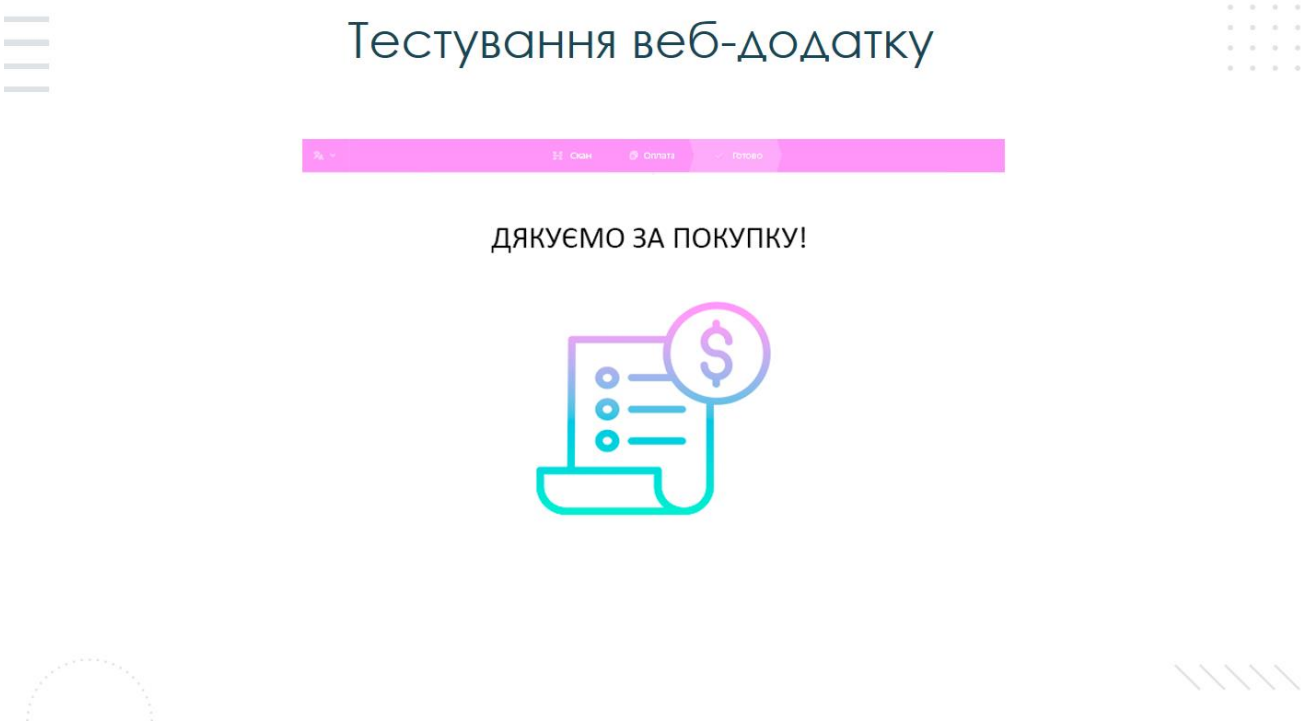


Рисунок Г.15 – Тестування веб-додатку(продовження)

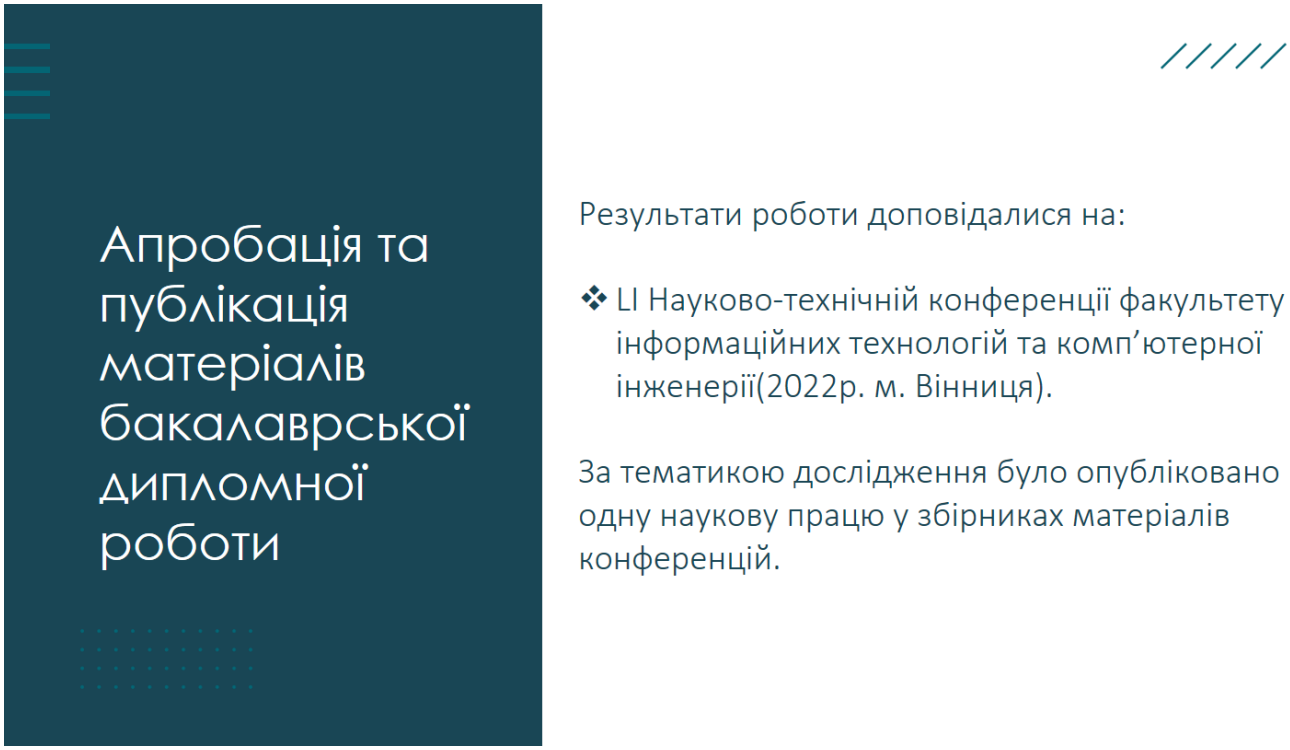


Рисунок Г.16 – Апробація та публікація матеріалів бакалаврської дипломної роботи

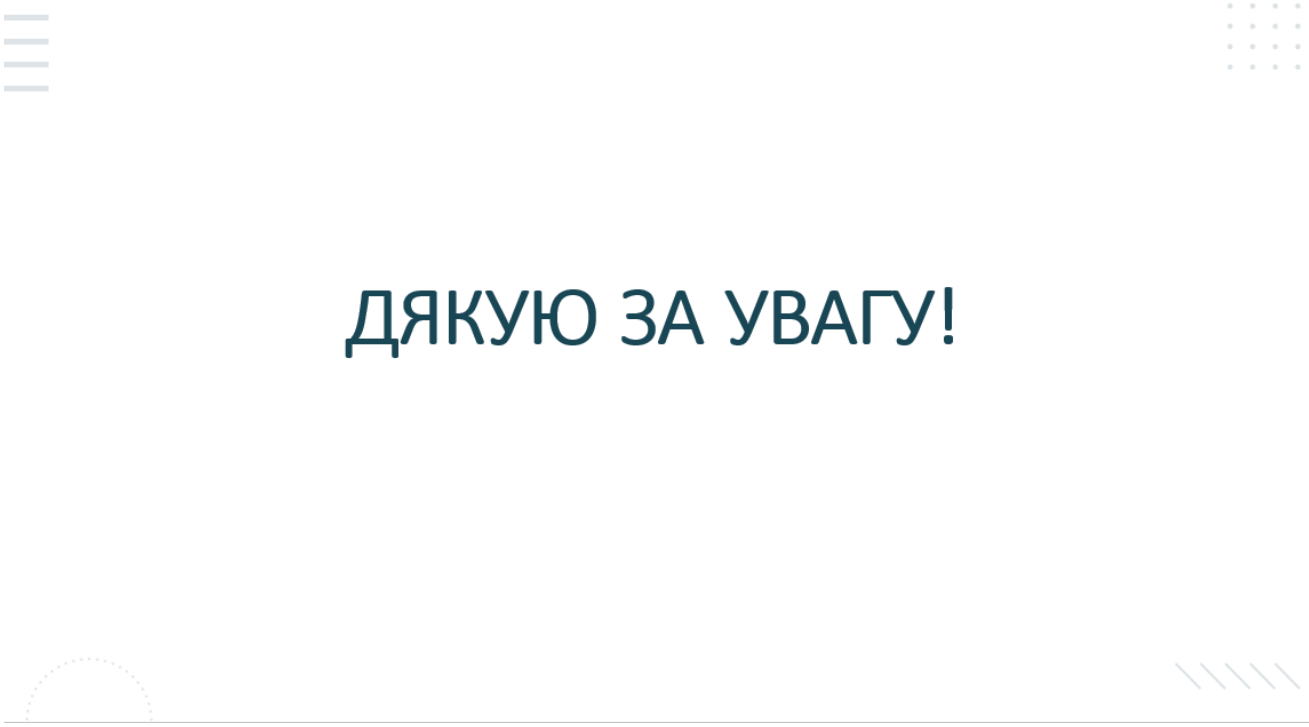


Рисунок Г.17 – Фінальний слайд