НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

КУРСОВА РОБОТА

з дисципліни «Об’єктовно Орієнтоване Програмування»

на тему: «Інтернет Магазин»

Студента групи ТР-25
зі спеціальності 122 «Комп’ютерні науки»
Кудрявцева Єгора Олеговича
(прізвище та ініціали)
Керівник доц. Шалденко О.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Кількість балів: _____ Оцінка: _____

Члени комісії _____ доц. Шалденко О.В.
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)
_____ ас. Здор К.А.
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

Київ- 2023 рік

Національний технічний університет України
“ Київський політехнічний інститут ім. Ігоря Сікорського”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ
Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ
Рівень вищої освіти Бакалавр
За освітньою програмою "“Цифрові технології в енергетиці”
Спеціальності 122 Комп’ютерні науки

З А В Д А Н Н Я
НА КУРСОВУ РОБОТУ СТУДЕНТУ

Кудрявцеву Єгору Олеговичу
1.Тема роботи: «Реалізувати систему для "Інтернет магазину"»
керівник курсової роботи - Здор Костянтин Андрійович
2. Строк подання студентом роботи: 05.01.2024
3. Вихідні дані до проєкту (роботи): середовище розробки – Visual Studio
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) –
_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1.	Отримання завдання до КР	1-3-й тижні	
2.	Вибір та дослідження методів, вибір відповідних структур даних, розробка алгоритмів	4-6-й тижні	
3.	Програмна реалізація	6-10-й тижні	
4.	Демонстрація першого варіанту	11-и тиждень	
5.	Тестування програми	11-й тиждень	
6.	Оформлення звіту	12-й тиждень	
7.	Захист курсової	13-й тиждень	

Студент _____ **Кудрявцев Є.О.**
(підпис) (ім’я ПРИЗВИЩЕ)

Керівник курсової роботи _____ **Здор К.А.**
(підпис) ім’я ПРИЗВИЩЕ)

Анотація

Цей програмний код реалізує задачу створення інтернет-магазину, реалізованого мовою програмування C# із використанням об'єктно-орієнтованого програмування.

Програма взаємодіє з користувачем через консольний інтерфейс, пропонуючи різноманітні опції, такі як реєстрація, вхід, перегляд балансу, поповнення рахунку, перегляд товарів, покупка товарів та перегляд історії покупок. Кожна опція реалізована в окремому класі, що дозволяє легко розширювати функціональність магазину.

Даний застосунок містить набір сутностей, моделей, сервісів, які мають різні методи для взаємодіями з базою даних та репозиторії як викликаються сервісами й взаємодіють безпосередньо з базою даних, записуючи або змінюючи дані. Також директорія UI містить набір команд, які викликаються в головній функції й реалізують роботу всього проєкту.

Основна логіка програми визначена в головній функції Main(), де ініціалізуються сервіси та об'єкти для обробки даних. Програма пропонує зручний інтерфейс для взаємодії з магазином, надаючи користувачеві можливість зручно керувати своїм обліковим записом та здійснювати покупки.

Annotation

This code implements the task of creating an online store implemented in the C# programming language using object-oriented programming.

The program interacts with the user through a console interface, offering various options such as registration, login, balance view, top-up account, view products, purchase products and view purchase history. Each option is implemented in a separate class, which allows you to easily expand the functionality of the store.

This application contains a set of entities, models, services that have different methods for interactions with the database and repositories as called by services and interact directly with the database, writing or changing data. Also, the UI directory contains a set of commands that are called in the main function and implement the work of the entire project.

The main logic of the program is defined in the main function Main(), where services and objects for data processing are initialized. The program offers a convenient interface for interacting with the store, giving the user the opportunity to conveniently manage their account and make purchases.

Зміст

ВСТУП 6
РОЗДІЛ 1 МОЖЛИВОСТІ ТА ВИКОРИСТАННЯ ООП 7
1.1 Модульність та перевикористання коду 7
1.2 Наслідування та розширення 8
1.3 Інкапсуляція, захист даних, поліморфізм та гнучкість 9
1.4 Реальне моделювання об'єктів з реального світу в ООП 10
1.5 Ефективне керування проєктом 11
РОЗДІЛ 2 ПОСТАНОВКА ЗАДАЧІ 13
2.1 Вибір середовища розробки 13
2.2 Вибір завдання для реалізації коду 14
РОЗДІЛ 3 СТВОРЕННЯ КОДУ ПРОГРАМИ ТА РЕАЛІЗАЦІЯ 16
3.1 Блок-схеми та Структура Коду 16
3.2 Головна Функція Main() та Завантаження Товарів 19
3.3 Реалізація сервісів і їх функціонал 21
3.4 Реалізація репозиторіїв і їх функціонал 27
3.5 Сутності програми (Entities) та база даних DbContext 29
3.6 Моделі програми 31
3.7 Реалізація Роботи Програми 34
3.8 Результат Роботи Програми 42
ВИСНОВКИ 46
ВИКОРИСТАНІ ДЖЕРЕЛА 47
ДОДАТОК 48

ВСТУП

Об'єктно-орієнтоване програмування – це парадигма програмування, яка базується на концепції об'єктів[1]. Об'єкти в програмі взаємодіють один з одним, передаючи повідомлення та викликаючи методи. Основні концепції об'єктно-орієнтованого програмування включають у себе класи та об'єкти. Клас можна розглядати як шаблон або опис, який визначає структуру та поведінку об'єктів. Об'єкт, у свою чергу, представляє собою екземпляр класу, обладнаний конкретними значеннями та можливістю зберігати стан об'єкта.

Інкапсуляція у об'єктно-орієнтованому програмуванні полягає в тому, щоб об'єднати окремі дані, змінні та методи, що працюють з цими даними в одному класі, і приховати деталі реалізації від користувача.

Наслідування дозволяє створювати новий клас на основі вже існуючого, успадковуючи його властивості та методи. Це забезпечує можливість повторного використання коду та формування ієрархії класів.

Поліморфізм в об'єктно-орієнтованому програмуванні означає, що об'єкти одного класу можуть використовуватися так, як об'єкти іншого класу. Це може виражатися у викликах методів з однаковим ім'ям, але з різною реалізацією.

Об'єктно-орієнтоване програмування своїм витоком має створення мови програмування Симула у 1960-х роках, яке відбувалося паралельно із наростанням дискусій щодо кризи програмного забезпечення. Внаслідок ускладнення як апаратної, так і програмної частини, утримати високу якість програм стало надзвичайно складно. ООП частково вирішує цю проблему, зосереджуючись на модульності програми.

У порівнянні із традиційним підходом, коли програму розглядали як сукупність підпрограм або послідовність інструкцій для комп'ютера, ООП розглядає програми як сукупність об'єктів. Згідно із парадигмою об'єктно-орієнтованого програмування, кожен об'єкт може приймати повідомлення, обробляти дані та передавати повідомлення іншим об'єктам.

РОЗДІЛ 1 МОЖЛИВОСТІ ТА ВИКОРИСТАННЯ ООП

1.1 Модульність та перевикористання коду

Однією з важливих переваг об'єктно-орієнтованого програмування є його здатність до модульності та перевикористання коду. Ці концепції визначають спосіб організації програмного забезпечення, роблячи його більш гнучким, підтримуючи легшу розширюваність та утримання[2].

Модульність означає розбиття програми на невеликі, самостійні, логічно зв'язані частини, відомі як модулі або класи. Кожен модуль відповідає за конкретну функціональність, що робить код більш читабельним та легким для управління. Модулі можуть взаємодіяти між собою, передавати дані та повідомлення, але вони залишаються відокремленими та незалежними.

Коли програма розділена на модулі, розробники можуть працювати над окремими частинами без необхідності розуміти всю систему. Це полегшує розподіл обов'язків у команді та дозволяє ефективно керувати великими проєктами.

Однією з ключових переваг ООП є можливість перевикористання коду. За допомогою концепцій наслідування та поліморфізму, розробники можуть створювати класи та об'єкти, які можна використовувати у різних частинах програми або навіть в інших проєктах.

Коли клас розроблений та перевірений, його можна легко використовувати в інших частинах програми без необхідності повторювати код. Це робить розробку ефективнішою, оскільки той самий код може бути використаний у різних контекстах, забезпечуючи консистентність та зменшуючи кількість помилок.

Загалом, модульність та перевикористання коду в ООП сприяють зростанню продуктивності розробників, полегшуючи підтримку та розширення програмного

забезпечення. Ці концепції є важливими для створення надійних та масштабованих систем.

1.2 Наслідування та розширення

Однією з ключових концепцій в ООП є наслідування та розширення, що сприяють гнучкості та легкості розробки, дозволяючи створювати ієрархії класів та ефективно розширювати функціональність програм[3].

Розширення визначає здатність створювати новий клас, використовуючи властивості та методи вже існуючого. В класі-нащадку (похідному) можна успадковувати або розширювати функціональність базового класу (батьківського). Це дозволяє створювати загальні класи, які визначають основні властивості, а потім розширювати їх у конкретних класах[4].

Використання наслідування полегшує утримання та розширення коду. Він дозволяє створювати логічні ієрархії, де класи можуть успадковувати загальні характеристики, сприяючи повторному використанню та зменшенню дублювання коду[4].

У даному проєкті ця концепція ООП надає багато можливостей, завдяки яким увесь код стає більш зрозумілим для будь-якого користувача. Також важливою перевагою є простота доповнення коду даного проєкту та можливість додавати різні нові методи, сервіси та репозиторії.

Розширення в контексті ООП означає додавання нового функціоналу до існуючого класу без зміни його вихідного коду. Це можливо завдяки поліморфізму та наслідуванню. Розробники можуть створювати нові класи, які успадковують функціональність вже існуючих, і при цьому можуть додавати або перевизначати методи для реалізації нового функціоналу[5].

Розширення робить код більш гнучким та адаптивним до змін. Відсутність необхідності змінювати вихідний код існуючих класів спрощує процес додавання нового функціоналу, забезпечуючи збереження стабільності системи.

Комбінація наслідування та розширення дозволяє створювати сильні ієрархії класів, що полегшує розробку, утримання та розширення програмного забезпечення[5]. Вони є невід'ємною частиною принципів ООП, які роблять код більш зрозумілим, гнучким та ефективним.

1.3 Інкапсуляція, захист даних, поліморфізм та гнучкість

Інкапсуляція та захист даних є ключовими концепціями в об'єктно-орієнтованому програмуванні (ООП), спрямованими на створення безпечних, модульних та невід'ємних від інших частин системи об'єктів.

Інкапсуляція визначає спосіб об'єднання даних та методів, що їх обробляють, в єдиному об'єкті або класі. Це дозволяє приховати деталі реалізації та захистити їх від несанкціонованого доступу. Зміни внутрішньої реалізації можуть бути внесені, не впливаючи на інші частини системи, що використовують цей об'єкт [6].

Інкапсуляція забезпечує контроль доступу до даних та функцій, що їх обробляють, зменшуючи ризик помилкового використання та підвищуючи надійність програм. Крім того, ця концепція робить код більш зрозумілим та обслуговуваним, забезпечуючи чіткі інтерфейси для взаємодії з об'єктами.

Захист даних – це одна з головних мет цієї концепції. Використання модифікаторів доступу, таких як private, дозволяє приховати деталі реалізації та забезпечити доступ тільки через публічні методи, які контролюють доступ до даних. Це запобігає прямому доступу та маніпуляції змінних, забезпечуючи контрольований та безпечний спосіб взаємодії з об'єктом.

Захист даних також дозволяє контролювати інтерфейси, які використовуються для доступу до інформації. Це сприяє уникненню неправильного використання даних та підтримує стабільність системи в умовах змін [6].

Загалом, інкапсуляція та захист даних роблять ООП більш безпечним та модульним, сприяючи надійності та чіткості в коду. Ці концепції грають ключову роль у створенні ефективних та стабільних програмних систем.

Поліморфізм та гнучкість представляють собою основні принципи об'єктно-орієнтованого програмування (ООП), які визначаються здатністю коду адаптуватися до змін та сприяти створенню більш гнучких та розширюваних систем[7].

Існують два типи поліморфізму: compile-time (статичний) і run-time (динамічний). Статичний поліморфізм пов'язаний із перевантаженням методів та операторів під час компіляції, тоді як динамічний поліморфізм пов'язаний із використанням віртуальних методів та перевизначенням під час виконання програми [8].

Динамічний поліморфізм включає в себе використання механізмів, таких як віртуальні методи та інтерфейси, дозволяючи об'єктам одного класу заміщувати функціонал об'єктів іншого класу. Це робить код більш адаптивним до змін та дозволяє створювати більш загальні та перевикористовувані рішення.

Гнучкість в контексті ООП означає здатність системи легко адаптуватися до нових вимог та змін. Поліморфізм є ключовим інструментом для досягнення гнучкості в ООП[8].

Поліморфізм та гнучкість спільно допомагають створювати системи, які легко розширюються та адаптуються до нових умов, що робить їх ефективними та витратними на утримання в середньо- та довгостроковій перспективі[8]. Ці концепції роблять ООП потужним підходом до розробки програмного забезпечення в умовах змінливого середовища.

1.4 Реальне моделювання об'єктів з реального світу в ООП

Однією з центральних концепцій об'єктно-орієнтованого програмування (ООП) є реальне моделювання об'єктів з реального світу. Ця ідея полягає в тому,

щоб створювати програми, які відображають структуру та взаємодію об'єктів та процесів, які спостерігаються в реальному світі.

Моделювання об'єктів з реального світу дозволяє розробникам працювати з концепціями та абстракціями, зрозумілими для них у реальному житті. Код, побудований на основі реальних об'єктів, є більш зрозумілим та інтуїтивним для розробників та тих, хто працює з програмою. Відповідність моделі об'єктів реальному світі допомагає забезпечити, що програма адекватно відображає та взаємодіє з реальними процесами та об'єктами.

Реальне моделювання об'єктів з реального світу в ООП є потужним інструментом для розробників, оскільки дозволяє створювати системи, які відображають складні взаємозв'язки та структури реального життя. Це сприяє покращенню якості та зрозумілості програмного забезпечення[8].

1.5 Ефективне керування проєктом

Ефективне керування проєктом є визначальним аспектом для досягнення успіху в сучасному розробленні програмного забезпечення. Засноване на передових підходах та найкращих практиках, таке керування стає ключовим фактором у забезпеченні якісної продукції, вчасної доставки та задоволення потреб замовників[8].

Визначення чіткої мети та конкретних задач є першим кроком до успішного керування проєктом. Важливо визначити, що саме потрібно досягти, і як це відображається у визначених завданнях.

Стратегічне планування включає в себе розробку чіткого плану, який визначає ресурси, часові рамки, бюджет та інші аспекти проєкту. План повинен бути гнучким та враховувати зміни у ході роботи[9].

Оптимальне використання ресурсів, таких як людські, фінансові та технічні, є ключем до ефективності проєкту. Команда повинна бути розподілена з урахуванням навичок та досвіду учасників.

Якісна комунікація є основою ефективного керування проєктом. Забезпечення відкритого та постійного обміну інформацією між командою та клієнтом допомагає уникнути непорозумінь та вчасно вирішувати питання.

Активний аналіз та управління ризиками дозволяє передбачати можливі труднощі та приймати стратегічні рішення для їх вирішення. Зарання виявлені ризики мають менший вплив на успішність проєкту.

Використання ітеративних та гнучких методик дозволяє адаптувати процес розроблення до змін у вимогах та забезпечує можливість швидко реагувати на зміни[10].

Систематична оцінка виконаної роботи та здійснення коригувань допомагають удосконалювати ефективність проєкту. Збирання вивчених уроків та їх застосування у майбутніх проєктах є важливою частиною цього процесу. Ключові вигоди ефективного керування проєктом:

Ефективне керування дозволяє уникнути затримок та вчасно поставити готовий продукт або його етап.

Чітке розуміння та врахування вимог замовника призводить до вищого рівня задоволеності клієнтів.

Раціональне розподіл ресурсів сприяє оптимальному використанню та зменшенню витрат.

Систематичний моніторинг та контроль ризиків сприяє стабільності та надійності продукту.

Грамотне керування підвищує продуктивність команди та сприяє створенню позитивного робочого середовища.

Усі ці аспекти створюють комплексний підхід до ефективного керування проєктом, що забезпечує найкращі умови для успішного завершення завдань у розробленні програмного забезпечення.

РОЗДІЛ 2 ПОСТАНОВКА ЗАДАЧІ

2.1 Вибір середовища розробки

Визначення середовища розробки є важливою передумовою успішної реалізації будь-якого програмного проєкту. У рамках даної роботи обране середовище розробки – Visual Studio, вважається одним із найбільш потужних та високоефективних інструментів для розробки програм на мові C# та інших мов програмування.

Visual Studio надає повноцінне інтегроване середовище розробки, що об'єднує в собі текстовий редактор, візуальний дизайнер і інші інструменти для зручного створення та редагування коду.

Visual Studio спеціально розроблено для роботи з мовою програмування C#, що дозволяє розробникам використовувати всі можливості цієї мови та отримувати всі переваги інструменту.

Інтеграція з платформою .NET дозволяє використовувати широкий спектр вбудованих бібліотек, класів та інших засобів, що значно полегшує розробку.

Visual Studio надає розширену підтримку об'єктно-орієнтованого програмування (ООП) та інших парадигм, що робить його відмінним вибором для роботи з різноманітними структурами коду.

Інструментарій Visual Studio включає в себе потужні засоби для відлагодження коду, визначення помилок та тестування програм, що значно спрощує процес розробки та виправлення помилок.

Вибір Visual Studio на основі порад викладача визначено комплексними перевагами середовища, такими як зручний інтерфейс, широкі можливості розширення та активна підтримка розробників. Дані рекомендації спрямовані на надання студентам доступу до інструментів, які сприяють не лише ефективному вивченню мови програмування, а й формуванню навичок роботи з потужними розробницькими інструментами.

За допомогою Visual Studio, з’являється можливість писати та відлагоджувати код у зручному середовищі, що стимулює активний розвиток їхніх програмістських навичок. Протягом семестру вже було вдало розроблено декілька кодів у даному середовищі, що дозволяє продовжити роботу у знайомому та продуктивному середовищі.

2.2 Вибір завдання для реалізації коду

Переглянувши всі можливі варіанти завдань, було вирішено зупинитися на реалізації Інтернет-магазину. Дане завдання передбачає створення функціональної та інтуїтивно зрозумілої системи, яка включає в себе ряд ключових елементів. Основні функціональні вимоги до системи "Інтернет-магазин":

Система має реалізовувати функціонал, який надає доступ користувачам реєструватися, вводячи необхідні особисті дані та отримуючи унікальний ідентифікатор.

Зареєстрованим користувачам має бути доступ до системи через введення ідентифікатора та пароля. Система повинна надати можливість користувачам поповнювати свій баланс, використовуючи різні методи оплати.

Наявність інтуїтивного та організованого відображення товарів, їхніх характеристик та цін є ключовим аспектом для забезпечення комфортного вибору покупцями.

Реалізація функціоналу для придбання товарів, а також оновлення бази даних після купівлі для відображення змін у доступності товарів.

Забезпечення зручного та структурованого перегляду історії покупок для кожного користувача, включаючи дані про придбані товари та витрачені кошти.

Розробка Інтернет-магазину надає практичний досвід у роботі з базами даних, функціями авторизації та іншими ключовими аспектами програмного забезпечення.

Реалізація функціоналу для реєстрації користувачів, обробки транзакцій та історії покупок дозволяє студенту розгортати різноманітні аспекти розробки програм.

Створення Інтернет-магазину сприяє реалізації об'єктно-орієнтованого програмування (ООП) та роботі з базами даних, що є ключовими концепціями в програмуванні.

Під час реалізації системи для Інтернет-магазину, необхідно враховувати аспекти безпеки, оптимізації та ефективного використання ресурсів. Також, важливо забезпечити гнучкість та масштабованість системи для подальшого розвитку та покращення.

РОЗДІЛ 3 СТВОРЕННЯ КОДУ ПРОГРАМИ ТА РЕАЛІЗАЦІЯ

3.1 Блок-схеми та Структура Коду

Блок-схеми демонструють комплексний погляд на структуру та логіку програми для інтернет-магазину. Кожна блок-схема служить важливим інструментом для візуалізації різних аспектів роботи програми та її взаємодії з користувачем, базою даних та внутрішніми компонентами.

Перший розділ блок-схем (Рисунок 3.1) присвячений основній функціональності програми, відображаючи взаємодію користувача з інтерфейсом та реакцію програми на введені дані. Це дозволяє легко розуміти порядок операцій, які виконуються при виклику різних функцій.

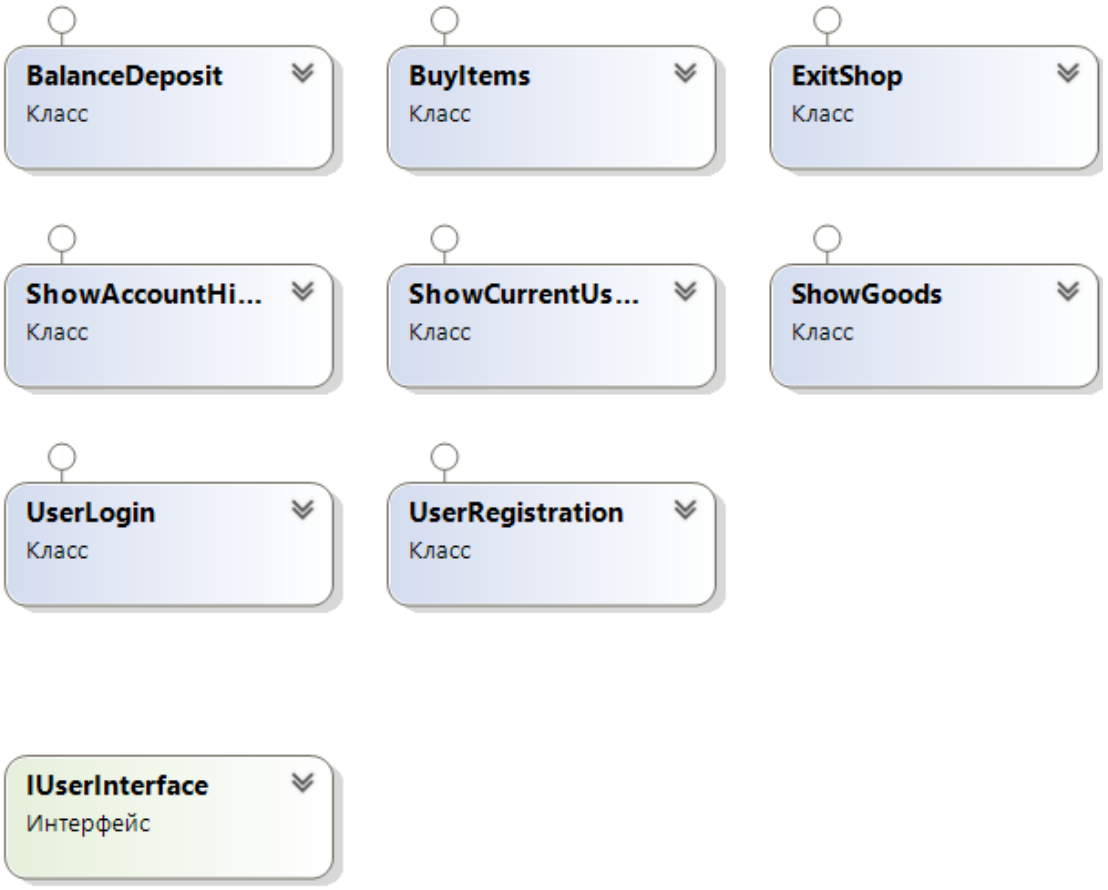

Рисунок 3.1 – Візуалізація класів команд

Другий розділ стосується внутрішніх механізмів, таких як сервіси (Рисунок 3.2) та репозиторії (Рисунок 3.3). Блок-схеми відображають, як ці компоненти взаємодіють один з одним та з базою даних, визначаючи процеси збереження, оновлення та отримання інформації.

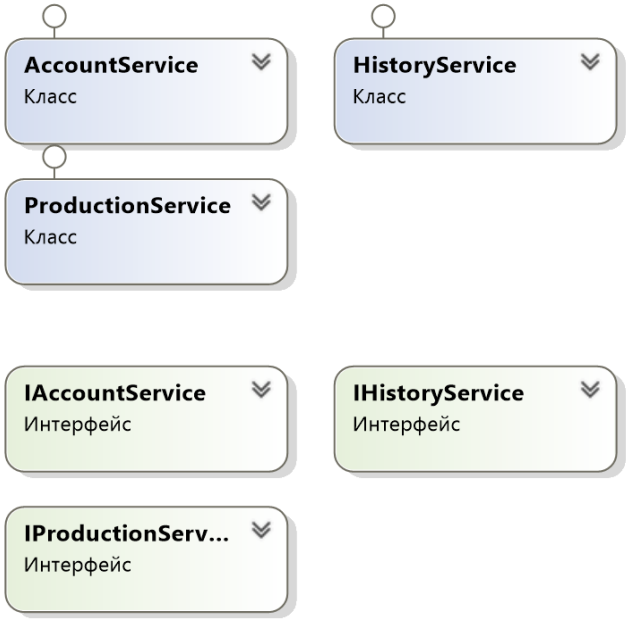

Рисунок 3.2 – Візуалізація класів сервісів

Блок-схема репозиторіїв (Рисунок 3.3) демонструє таку саму кількість класів, як і в сервісах, які викликають відповідні репозиторії.

Рисунок 3.3 – Візуалізація класів репозиторіїв

Загальна блок-схема усього застосунку (Рисунок 3.4) демонструє весь набір класів та інструментів для взаємодії коду з базою даних та реалізації коректної логіки роботи програми.

На загальній блок-схемі (Рисунок 3.4) зображено всі класи коду. Усі блоки на даній блок-схемі читаються згори-униз.

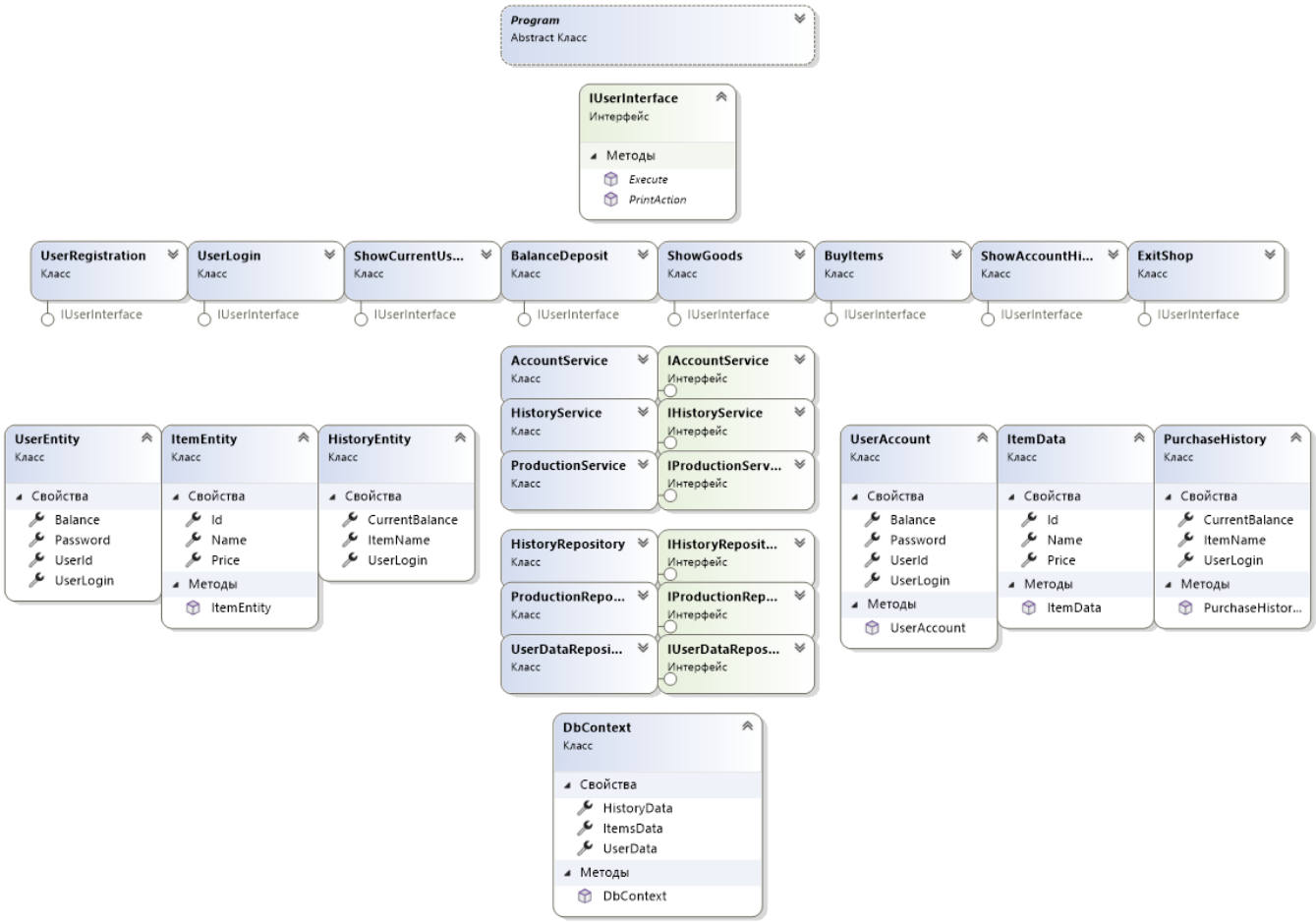

Рисунок 3.4 – Повна візуалізація класів і їх методів усього застосунку

Цей комплексний погляд на програму через блок-схеми надає глибше розуміння її функціональності та внутрішньої логіки, сприяючи як розробці, так і аналізу системи. Усі блок-схеми також були додані до відповідних директорій і завантажені разом із ними на Github.

3.2 Головна Функція Main() та Завантаження Товарів

Спочатку у класі Program створюється функція Main(), яка є головною у всій програмі. Дана функція визначає вхідну точку виконання програми та викликається автоматично при запуску застосунку. У тілі цієї функції відбувається ініціалізація різних об'єктів, налаштування середовища виконання, і завдання початкових значень змінним.

```
// Головна функця Main
Ссылок: 0
public static void Main()
{
    var context = new DbContext();
    var accountService = new AccountService(context);
    var productionService = new ProductionService(context);
    var historyService = new HistoryService(context);
```

Рисунок 3.5 – Опис головна функція Main()

У даному фрагменті коду (Рисунок 3.5) створюється ініціалізація основних об'єктів та сервісів, необхідних для функціонування Інтернет-магазину.

1. Ініціалізація об'єкту контексту бази даних (DbContext)
2. Ініціалізація сервісів обліку користувачів та товарів.

Далі в коді головної функції Main() було створено масив опцій (Рисунок 3.6).

```
// Реалізація вибору опцій через масив із командами
IUserInterface[] UIs =
[
    new ExitShop(),
    new UserRegistration(accountService),
    new UserLogin(accountService),
    new ShowCurrentUserBalance(accountService),
    new BalanceDeposit(accountService),
    new ShowGoods(productionService),
    new BuyItems(accountService, productionService, historyService),
    new ShowAccountHistory(accountService, historyService)
];
```

Рисунок 3.6 – Масив із командами

Даний масив потім викликається в циклі для вибору опцій (Рисунок 3.6).

```
// Безпосередньо можливість обрати необхідну опцію
int answer = 1, i;
while (answer != 0)
{
    for (i = 0; i < UIs.Length; i++)
    {
        Console.WriteLine($"{i}) {UIs[i].PrintAction()}");
    }


    answer = int.Parse(Console.ReadLine());
    UIs[answer].Execute();
}
```

Рисунок 3.7 – Цикл вибору команд програми

Цей фрагмент коду (Рисунок 3.7) відповідає за реалізацію вибору опцій в інтерфейсі користувача Інтернет-магазину через консоль. Давайте розглянемо кожен елемент цього фрагменту докладніше:

1. Ініціалізація масиву об'єктів «IUserInterface». В об’єктах, які реалізовують інтерфейс IUserInterface, створюється масив об’єктів. Кожен об'єкт представляє собою конкретну опцію взаємодії з магазином (наприклад, реєстрація, вхід, перегляд балансу тощо).
2. Цикл вибору опцій. У даному коді використовується цикл, який продовжується до тих пір, поки користувач не вибере опцію закриття магазину (введе 0). На кожній ітерації виводиться список доступних опцій, користувач вводить свій вибір, і відповідна опція виконується через метод Execute() об'єкта, який був створений раніше в масиві UIs.

Загалом даний механізм забезпечує інтерактивність інтерфейсу користувача, дозволяючи йому взаємодіяти з різними опціями магазину прямо з консолі. Кожна опція відповідає певному функціоналу і може бути розширена або змінена за необхідності.

Підсумовуючи загальний опис коду Main() включає в себе:

- Код реалізує консольний інтерфейс для взаємодії з Інтернет-магазином.
- Ініціалізується об'єкт контексту бази даних та сервіси для обліку облікових записів та товарів.
- Створюється масив із об'єктами інтерфейсу IUserInterface, що представляють різні опції взаємодії з магазином.
- Використовується цикл для вибору та виклику опцій взаємодії за допомогою введення користувача.

3.3 Реалізація сервісів і їх функціонал

Сервіси відповідають за реалізацію бізнес-логіки та виконання операцій, які необхідні для виконання конкретного функціоналу програми. Основні завдання сервісів включають обробку та координацію операцій над даними, взаємодію з репозиторіями для доступу до даних, а також надання інтерфейсу для взаємодії з іншими частинами системи.

Загальна структура сервісів і їх інтерфейсів програми розподілена по директоріях (Рисунок 3.8).

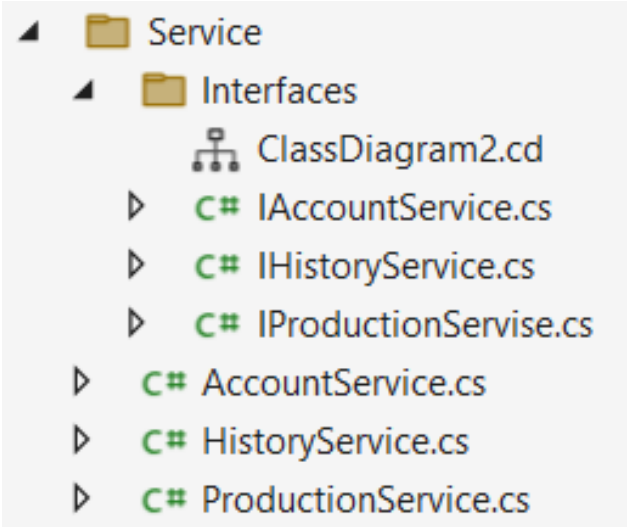

Рисунок 3.8 – Структура сервісів програми

Для реалізації сервісу AccountService, використовуються такі методи, як: додавання користувача, читання інформації про користувачів, отримання ідентифікатора залогіненого користувача, перевірка логіну й оновлення даних про користувача (Рисунок 3.9).

```
Ссылок: 1
internal interface IAccountService
{
    Ссылок: 2
    void AddUserData(UserAccount user);
    Ссылок: 9
    List<UserAccount> ReadAll();
    Ссылок: 3
    bool LoginCheck { get; set; }
    Ссылок: 3
    int CurrentUserId { get; set; }
    Ссылок: 5
    int GetCurrentUserId();
    Ссылок: 3
    void Update(UserAccount entity);

}
```

Рисунок 3.9 – Інтерфейс IAccountService

На початку даного сервісу створюється конструктор відповідного репозиторію (Рисунок 3.10).

```
// Поле, відповідає за доступ до бази даних
UserDataRepository accountRepository;

// Конструктор класу
Ссылок: 1
public AccountService(DbContext context)
{
    accountRepository = new UserDataRepository(context);
}
```

Рисунок 3.10 – Створення поля та конструктору класу AccountService

Спочатку створюється поле, яке відповідає за доступ до даних облікових записів користувачів. Такий підхід вказує на використання паттерну Dependency Injection або Service Locator для взаємодії з репозиторієм.

Далі створюється конструктор класу, який приймає екземпляр DbContext як параметр та ініціалізує поле accountRepository, створюючи новий об'єкт UserDataRepository і передаючи йому DbContext. Це вказує на використання

Dependency Injection для надання залежностей класу. Такий підхід використовується в усіх наступних сервісах.

У класі є метод AddUserData, що займається безпосередньо додаванням інформації про нового користувача (Рисунок 3.11).

```
// Метод додавання нового акаунту
Ссылок: 2
public void AddUserData(UserAccount user)
{
    accountRepository.AddUser(Map(user));
}
```

Рисунок 3.11 – Метор AddUserData()

Наданий код відображає метод AddUserData, який додає новий обліковий запис користувача, а також внутрішній метод Map, що використовується для відображення об'єкту UserAccount на відповідний об'єкт UserEntity.

Даний код використовує принципи чистого коду та розділення обов'язків. Він не лише реалізує функціонал додавання нового користувача, але й використовує окремий маппер для визначення відображення полів між об'єктами.

Наступний метод ReadAll() відповідає за отримання списку зареєстрованих користувачів у базі даних (Рисунок 3.12).

```
// Список об'єктів UserAccount, які представляють всіх користувачів
Ссылок: 9
public List<UserAccount> ReadAll()
{
    // Отримання списку користувачів з репозиторію та їх мапіння до типу UserAccount
    var list = accountRepository.ReadAll().Select(x => x != null ? Map(x) : null).ToList();
    return list;
}
```

Рисунок 3.12 – Отримання списку користувачів із бази даних

Даний код використовує принципи розділення обов'язків та має чітку структуру. Використання методу ReadAll та маппера Map дозволяє ефективно отримувати та відображати дані користувачів, роблячи код зрозумілим та готовим

до подальшого розвитку, тобто можно використовувати неодноразово за необхідної ситуації.

Наступний код включає в себе дві змінні (LoginCheck та CurrentUserId) та метод GetCurrentUserId, який використовує ці змінні для отримання ідентифікатора поточного користувача (Рисунок 3.13):

```
// Змінні для отримання даних про користувача (перевірка на залогіненість)
Ссылок: 3
public bool LoginCheck { get; set; }
Ссылок: 3
public int CurrentUserId { get; set; }

//Отримання ідентифікатора користувача
Ссылок: 5
public int GetCurrentUserId()
{
    if (LoginCheck)
    {
        return CurrentUserId;
    }
    else
    {
        Console.WriteLine("No user is currently logged in");
        return -1;
    }
}
```

Рисунок 3.13 – Змінні та метод для отримання ідентифікатора поточного користувача

Код даного методу (Рисунок 3.13) відображає підходи до управління інформацією про статус залогіненості (тобто перевіряє чи зайшов користувач у аккаунт) та ідентифікатор користувача. Введення змінних через властивості дозволяє контролювати доступ та зміну значень. Метод GetCurrentUserId забезпечує безпечний спосіб отримання ідентифікатора користувача, враховуючи його статус залогіненості. Останній метод даного сервісу Update() (Рисунок 3.14).

```
Ссылок: 3
public void Update(UserAccount entity)
{
    accountRepository.Update(Map(entity));

}
```

Рисунок 3.14 – Метод Update()

Він відповідає за оновлення інформації про користувача в системі й також використовує попередньо створений мапер, який є важливим для забезпечення правильного оновлення інформації в базі даних.

Функціонал цього сервісу складається з двох методів (Рисунок 3.15), використання яких важливе для запису історії покупок користувача та отримання масиву з цієї ж бази даних для друку в консоль.

```
Ссылок: 1
internal interface IHistoryService
{
    Ссылок: 2
    void AddPurchaseData(PurchaseHistory record);
    Ссылок: 2
    List<PurchaseHistory> ReadAll();

}
```

Рисунок 3.15 – Інтерфейс IHistoryService

Даний клас включає в себе два методи. Перший метод AddPurchaseData (Рисунок 3.16) додає запис про покупку в базу даних, та його мапер Map, що використовується для конвертації об'єкта типу PurchaseHistory в об'єкт типу HistoryEntity.

```
// Додавання запису про покупку в базу даних
Ссылок: 2
public void AddPurchaseData(PurchaseHistory record)
{
    historyRepository.AddHistoryRecord(Map(record));
}

// Його мапер
Ссылок: 1
private HistoryEntity Map(PurchaseHistory record)
{
    if (record == null)
    {
        return null;
    }
    return new HistoryEntity
    {
        UserLogin = record.UserLogin,
        ItemName = record.ItemName,
        CurrentBalance = record.CurrentBalance
    };
}
```

Рисунок 3.16 – Метод AddPurchaseData() та його Map()

Останній метод цього сервісу (Рисунок 3.17) демонструє використання мапінгу для конвертації об'єктів історії покупок з бази даних у представлення для подальшого використання.

```
// Отримання даних про історію покупок (всіх користувачів, для виводу окремого створено умовний оператор)
Ссылок: 2
public List<PurchaseHistory> ReadAll()
{
    // Отримання списку користувачів з репозиторію та їх мапіння до типу UserAccount
    var list = historyRepository.ReadAll().Select(x => x != null ? Map(x) : null).ToList();
    return list;
}

Ссылок: 1
private PurchaseHistory Map(HistoryEntity entity)
{
    return new PurchaseHistory()
    {
        UserLogin = entity.UserLogin,
        ItemName = entity.ItemName,
        CurrentBalance = entity.CurrentBalance
    };
}
```

Рисунок 3.17 – Метод ReadAll() та його Map()

Даний клас реалізує метод ReadAll() (Рисунок 3.18), який є важливим для реалізації покупок та виводу списку товарів:

```
Ссылок: 1
internal interface IProductionServise
{
    Ссылок: 3
    List<ItemData> ReadAll();
}
```

Рисунок 3.18 – Інтерфейс IProductionService

Він містить код конструктора репозиторію, для подальшої взаємодії з базою даних і метод для отримання всіх об’єктів магазину.(Рисунок 3.19).

```
// Отримання списку користувачів
Ссылок: 3
public List<ItemData> ReadAll()
{
    // Отримання списку користувачів з репозиторію та їх мапіння до типу UserAccount
    var list = productionRepository.ReadAll().Select(x => x != null ? Map(x) : null).ToList();
    return list;
}
```

Рисунок 3.19 – Метод ReadAll()

Даний код містить метод ReadAll(), який отримує та мапить всі об'єкти ItemEntity (елементи продукції) із репозиторію до об'єктів типу ItemData. Також включено визначення маппера Map, який використовується для конвертації окремого об'єкта ItemEntity до представлення елементу продукції у вигляді ItemData.

3.4 Реалізація репозиторіїв і їх функціонал

Усі репозиторії пов’язані з базою даних, повертають необхідний результат і взаємодіять із нею, викликаючися розглянутими попередньо сервісами. Загалом Репозиторії відповідають за доступ до бази даних. Їх завданням є абстрагування коду взаємодії з базою даних, надання зручного інтерфейсу для роботи з даними та виконання операцій зчитування, запису, оновлення та видалення записів.

Структора репозиторіїв виглядає наступним чином (Рисунок 3.20).

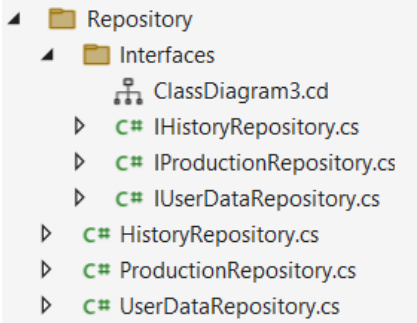

Рисунок 3.20 – Структура репозиторіїв програми

У інтерфейсі IUserDataRepository маємо 3 методи: додавання користувача, отримання даних про користувачів, оновлення користувача (Рисунок 3.21).

```
Ссылок: 1
internal interface IUserDataRepository
{
    Ссылок: 2
    void AddUser(UserEntity user);
    Ссылок: 2
    List<UserEntity> ReadAll();
    Ссылок: 2
    void Update(UserEntity entity);
}
```

Рисунок 3.21 – Інтерфейс IUserDataRepository

Для доступу безпосередньо до даних облікових записів користувачів, було створено поле в коді програми. Після цього створено конструктор для об'єкта класу UserDataRepository. Він приймає параметр dbContext, який є об'єктом типу DbContext і передається як аргумент конструктора. При ініціалізації класу UserDataRepository, поле _dbContext приймає значення переданого об'єкта DbContext, що дозволяє цьому репозиторію працювати з визначеною базою даних. Так само, як із сервісами даний підхід використовується в усіх наступних репозиторіях.

У Інтерфейсі IHistoryRepository (Рисунок 3.22) маємо два методи: AddHistoryRecord і ReadAll(). Вони дають змогу додавати запис про історію покупок до бази даних і повертати їх для подальшого друку в програмі.

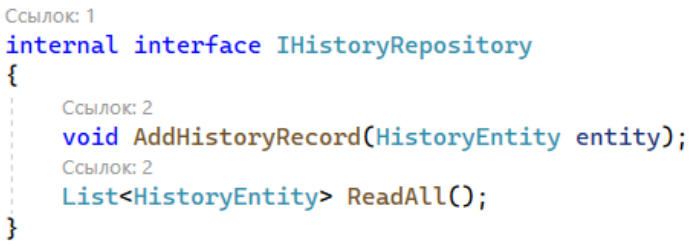

Рисунок 3.22 – Інтерфейс IHistoryRepository

Останній клас ProductionRepository (Рисунок 3.23) має лише одну функцію, яка має важливе значення для здійснення покупок, оскільки дає змогу вивести список усіх доступних товарів.

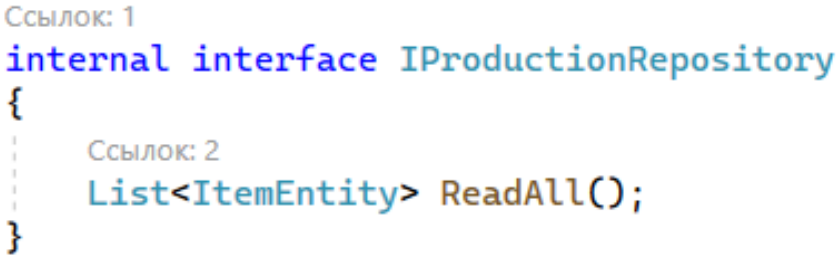

Рисунок 3.23 – Інтерфейс IProductionRepository

3.5 Сутності програми (Entities) та база даних DbContext

Сутності в програмі представляють основні об'єкти або елементи, які моделюють реальні або важливі концепції в контексті програми. У програмі існує три сутності: HistoryEntity, ItemEntity, UserEntity (Рисунок 3.24).

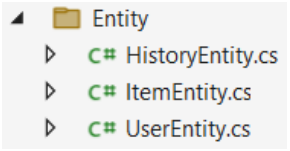

Рисунок 3.24 – Сутності програми

Клас UserEntity має в собі такі властивості, як: логін користувача, пароль, поточний рахунок, унікальний ідентифікатор користувача (Рисунок 3.25).

```
Ссылок: 11
public class UserEntity
{
    Ссылок: 3
    public string UserLogin { get; set; }
    Ссылок: 3
    public string Password { get; set; }
    Ссылок: 2
    public int Balance { get; set; }
    Ссылок: 5
    public int UserId { get; set; }
}
```

Рисунок 3.25 – Сутність користувача UserEntity

Дивлячись на клас товарів магазину, можна побачити такі властивості, як: унікальний ідентифікатор, назва та ціна (Рисунок 3.26).

```
Ссылок: 12
public class ItemEntity
{
    Ссылок: 3
    public int Id { get; set; }
    Ссылок: 3
    public string Name { get; set; }
    Ссылок: 3
    public int Price { get; set; }

    Ссылок: 6
    public ItemEntity(int id, string name, int price)
    {
        Id = id;
        Name = name;
        Price = price;
    }
}
```

Рисунок 3.26 – Сутність товару магазину ItemEntity

Клас HistoryEntity має в собі такі властивості, як: логін користувача, що здійснив покупку, назва купленого товару, поточний рахунок користувача після проведення покупки (Рисунок 3.27).

```
Ссылок: 9
public class HistoryEntity
{
    Ссылок: 2
    public string UserLogin { get; set; }
    Ссылок: 2
    public string ItemName { get; set; }
    Ссылок: 2
    public int CurrentBalance { get; set; }
}
```

Рисунок 3.27 – Сутність запису в масив історії покупок HistoryEntity

В сутності предметів маємо конструктор, завдяки якому можемо додавати предмети до бази даних. Усі назви властивостей були створені та названі таким чином, щоб можна було інтуїтивно розуміти їх ціль і задачу в коді. Яскравим прикладом є UserLogin, що означає дослівно «ЛогінКористувача», або Balance, значенням якого є поточний баланс користувача.

Клас DbContext є центральним елементом, який утримує та забезпечує доступ до даних у програмі. У цьому класі представлені списки об'єктів для користувачів UserData, товарів ItemsData та історії HistoryData (Рисунок 3.28).

```
Ссылок: 11
public class DbContext
{
    Ссылок: 5
    public List<UserEntity> UserData { get; set; } = new ();
    Ссылок: 2
    public List<ItemEntity> ItemsData { get; set; } = new ();
    Ссылок: 3
    public List<HistoryEntity> HistoryData { get; set; } = new ();
```

Рисунок 3.28 – Клас DbContext та його списки

Також у ньому міститься конструктор (Рисунок 3.29), який ініціалізує списки даних та наповнює ItemsData прикладовими об'єктами.

```
Ссылок: 1
public DbContext()
{
    UserData = new List<UserEntity>();
    HistoryData = new List<HistoryEntity>();
    ItemsData = new List<ItemEntity>
    {
        new ItemEntity (0, "Broad Axe", 80),
        new ItemEntity (1, "Hatchet", 60),
        new ItemEntity (2, "Canadian Axe", 50),
        new ItemEntity (3, "Double Bit Axe", 120),
        new ItemEntity (4, "Tomahawk", 50),
        new ItemEntity (5, "Viking Axe", 60)
    };
}
```

Рисунок 3.29 – Конструктор класу DbContext та наповнення ItemsData

Ціль даного класу – зберігання та управління даними у всій програмі. Він може бути переданий до інших частин програми, таких як сервіси або репозиторії, для забезпечення доступу до даних. Яскравим прикладом наповнення є попередньо згаданий список ItemsData. Усі назви цього класу створено для інтуїтивного розуміння по назвах, як UserData означає ІнформаціяКористувача, що сильно зпрощює розуміння коду та надає змогу спокійно працювати з ним далі.

3.6 Моделі програми

Модель – це абстракція, що представляє собою концептуальне відображення реальних об'єктів або концепцій у програмі. У коді програми вони визначають структуру та властивості даних, які обробляються програмою.

Основні їх задачі в програмі: визначення структури даних, управління та обробка цих даних і спрощення взаємодії з даними та їх ізоляція від інших частин програми.

Дана програма включає в себе три моделі (Рисунок 3.30) ItemData, PurchaseHistory та UserAccount.

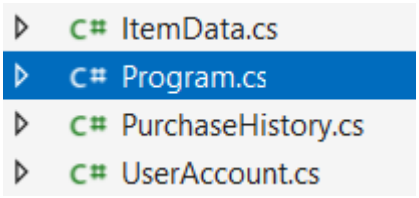

Рисунок 3.30 – Моделі програми

Дана модель (Рисунок 3.31) містить 3 властивосі та конструктор класу, призначений для створення екземпляра класу ItemData та ініціалізації його властивостей Id, Name та Price переданими значеннями.

```
Ссылок: 8
internal class ItemData
{
    Ссылок: 3
    public int Id { get; set; }
    Ссылок: 5
    public string Name { get; set; }
    Ссылок: 4
    public int Price { get; set; }

    Ссылок: 1
    public ItemData(int id, string name, int price)
    {
        Id = id;
        Name = name;
        Price = price;
    }
}
```

Рисунок 3.31 – Модель ItemData

Цей клас (Рисунок 3.32) так само містить властивості та конструктор, що ініціалізує новий екземпляр класу PurchaseHistory з вказаними значеннями для властивостей UserLogin, ItemName та CurrentBalance. Цікаво помітити, що далі

йде порожній конструктор public PurchaseHistory() { }. Він може бути використаний для створення екземплярів класу без передачі початкових значень.

```
Ссылок: 12
internal class PurchaseHistory
{
    Ссылок: 4
    public string UserLogin { get; set; }
    Ссылок: 4
    public string ItemName { get; set; }
    Ссылок: 4
    public int CurrentBalance { get; set; }

    Ссылок: 1
    public PurchaseHistory(string userLogin, string itemName, int currentBalance)
    {
        UserLogin = userLogin;
        ItemName = itemName;
        CurrentBalance = currentBalance;
    }

    Ссылок: 1
    public PurchaseHistory() { }
}
```

Рисунок 3.32 – Модель PurchaseHistory

Дана модель (Рисунок 3.33) містить 4 властивосі та конструктор класу, який ініціалізує новий екземпляр класу UserAccount з вказаними значеннями для властивостей UserLogin, Password та UserId. Цікавий момен, рахунок користувача (Balance) при створенні дорівнює 0, оскільки він ще не поповний свій рахунок.

```
Ссылок: 24
internal class UserAccount
{
    Ссылок: 11
    public string UserLogin { get; set; }
    Ссылок: 5
    public string Password { get; set; }
    Ссылок: 11
    public int Balance { get; set; }

    Ссылок: 3
    public int UserId { get; set; }

    Ссылок: 2
    public UserAccount(string userLogin, string password, int Id)
    {
        UserLogin = userLogin;
        Password = password;
        UserId = Id;
        Balance = 0;
    }
}
```

Рисунок 3.33 – Модель UserAccount

3.7 Реалізація Роботи Програми

Директорія команд має вигляд фолдеру, який включає в себе підфолдер Base в якому реалізований інтерфейс і реалізації цього інтерфейсу (Рисунок 3.34)

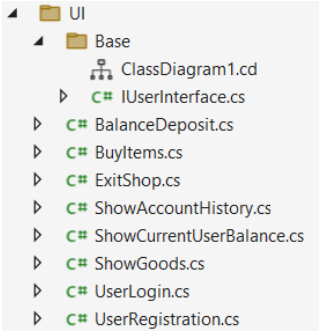

Рисунок 3.34 – Директорія команд

Інтерфейс IUserInterface має в собі такі методи: Execute() і PrintAction() (Рисунок 3.35).

```
Ссылок: 9
internal interface IUserInterface
{
    Ссылок: 10
    public void Execute();
    Ссылок: 9
    public string PrintAction();
}
```

Рисунок 3.35 – Інтерфейс IUserInterface

Перший із них – Execute() використовуємо для виконання необхідної дії в залежності від класу з всього переліку. Другий метод завжди повертає лише одну строку, що потрібно для створення зручного та приємного меню опцій під час роботи з програмою. Саме основна його ціль – дати короткий опис (2-5 слів) про роботу цього класу.

Далі будемо розглядати всі ці команди. Дивитися будемо на головну частину коду, саму логіку (проігноруємо поле та створення екземпляру сервісу в конструкторі), оскільки вони створюються в усіх них схожим чином, викликаючи однакові сервіси.

Після створення конструктору та екземпляру класу, створюється основна логіка реєстрації користувача (Рисунок 3.36). Користувач вводить логін, який перевіряється на унікальність в системі. Якщо логін вже існує, виводиться повідомлення про необхідність вибору іншого логіну. Після введення унікального логіну користувач вводить пароль. Створюється новий об'єкт UserAccount та додається до системи через accountService. Новостворений акаунт виводиться на екран для інформації користувача.

```
Ссылок: 2
public void Execute()
{
    // Запит на отримання даних
    List<UserAccount> usersList = accountService.ReadAll();
    string username = "";

    // Перевірка чи є користувач у системі
    do
    {
        Console.WriteLine("Input your login (username):");
        username = Console.ReadLine();

        // На випадок пустого масиву користувачів (тобто перший раз зареєструватися)
        if (usersList.Count == 0)
        {
            break;
        }

        if (usersList.Any(registeredUser => registeredUser.UserLogin == username))
        {
            Console.WriteLine("This login already exists, try another one");
        }
        else
        {
            break;
        }

    } while (true);


    Console.WriteLine("Input password:");
    string password = Console.ReadLine();

    // Виклик методу для створення нового акаунту
    var Id = accountService.ReadAll().Count();

    var newUser = new UserAccount(username, password, Id);
    accountService.AddUserData(newUser);

    // Дістаємо попередньо записані дані
    List<UserAccount> newUsersList = accountService.ReadAll();

    // Друк новоствореного акаунту (виводимо лише один останній акаунт)
    UserAccount user = newUsersList.Last();
    Console.WriteLine($"\nYou have just created new account, please remember your password\n" +
                      $"User: {user.UserLogin}\n" +
                      $"Password: {user.Password}\n");
}
```

Рисунок 3.36 – Клас UserRegistration

Даний клас виконує ключову роль у введенні та реєстрації нових користувачів у вашій програмі. Також є метод для його опису в меню (Рисунок 3.37).

```
// Метод для повернення інформації про дію даного класу в початкове меню
Ссылок: 2
public string PrintAction()
{
    return "Registration";
}
```

Рисунок 3.37 – Другий метод UserRegistration, називає його у меню «Реєстрацією»

Клас UserLogin відповіднальний за вхід користувача в акаунт. При успішному вході відбувається запис даних у змінні для перевірки залогіненості LoginCheck (Рисонок 3.38).

```
Ссылок: 2
public void Execute()
{
    // Отримання даних про користувачів
    List<UserAccount> usersList = accountService.ReadAll();

    string CurrentUserLogin = "", CurrentUserPassword = "";

    int CurrentUserBalance = 0, indicator = 0, index = 0;
    while (indicator == 0)
    {
        Console.WriteLine("Input your login:");
        string UserLogin = Console.ReadLine();

        if (string.IsNullOrEmpty(UserLogin))
        {
            Console.WriteLine("Input anything");
            continue;
        }
        foreach (UserAccount user in usersList)
        {
            if (user.UserLogin == UserLogin)
            {
                CurrentUserLogin = user.UserLogin;
                CurrentUserPassword = user.Password;
                CurrentUserBalance = user.Balance;
                indicator++;
                accountService.LoginCheck = true;
                accountService.CurrentUserId = index;
                break;
            }
            index++;
        }
        index = 0;
    }
```

Рисунок 3.38 – Перевірка наявності логіну в базі даних у класі UserLogin

Наступним чином програма перевіряє коректність введеного пароля для попередньо введеного акаунту (Рисунок 3.39).

```
while (true)
{
    Console.WriteLine("Input your password:");
    string UserPassword = Console.ReadLine();

    if (string.IsNullOrEmpty(UserPassword) || UserPassword != CurrentUserPassword)
    {
        Console.WriteLine("Wrong password!");
        continue;
    }
    break;
}
Console.WriteLine("\nYou have successfully logged in!\n" +
                  $"Welcome {CurrentUserLogin}!\n" +
                  $"Now you are able to purchase our goods, do not forget to check your balance before placing your order!\n" +
                  $"Your current balance is: {CurrentUserBalance}\n");
```

Рисунок 3.39 – Перевірка паролю користувача в класі UserLogin

Загальна логіка цього класу складається з трьох основних етапів:

1. Введення та перевірка логіну.
2. Введення та перевірка пароля.
3. Успішний вхід.

Клас ShowCurrentUserBalance (Рисунок 3.40) має просту задачу: показати баланс користувача.

```
Ссылок: 2
public void Execute()
{
    // Отримуємо масив зареєстрованих користувачів
    List<UserAccount> usersList = accountService.ReadAll();

    // Отримуємо індекс користувача, який залогінився
    int CurrentUserId = accountService.GetCurrentUserId();

    // Перевіряємо чи він залогінений (-1 не залогінений)
    if (CurrentUserId == -1)
    {
        return;
    }

    // Створюємо змінну нашого (залогіненого користувача)
    UserAccount CurrentUser = usersList[CurrentUserId];

    // Виводимо його баланс
    Console.WriteLine($"\n{CurrentUser.UserLogin} your balance is: {CurrentUser.Balance}\n");
}
```

Рисунок 3.40 – Клас ShowCurrentUserBalance

Загальна логіка класу ShowCurrentUserBalance має наступний вигляд:

1. Отримання та перевірка залогіненості користувача.
2. Вивід балансу користувача.

Клас BalanceDeposit (Рисунок 3.41) надає можливість користувачу поповнювати свіх рахунок в акаунті для подальших покупок товарів. Він також перевіряє чи залогінився коритувач у програмі.

```
Ссылок: 2
public void Execute()
{
    // Отримуємо масив зареєстрованих користувачів
    List<UserAccount> usersList = accountService.ReadAll();

    // Отримуємо індекс користувача, який залогінився
    int CurrentUserId = accountService.GetCurrentUserId();

    // Перевіряємо чи він залогінений (-1 не залогінений)
    if (CurrentUserId == -1)
    {
        return;
    }

    // Створюємо змінну нашого (залогіненого користувача)
    UserAccount CurrentUser = usersList[CurrentUserId];

    // Введення бажаної суми поповнення балансу
    Console.WriteLine("Type the mount of money you want to deposit on your balance:\n");
    int NewMoney = int.Parse(Console.ReadLine());



    // Додавання суми до балансу користувача
    CurrentUser.Balance += NewMoney;
    accountService.Update(CurrentUser);


    // Гарний друк про успішне додавання грошей до балансу
    Console.WriteLine("Processing...\n");
    Thread.Sleep(1500);
    Console.WriteLine($"Success! Your balance is {CurrentUser.Balance} now\n");
}
```

Рисунок 3.41 – Клас BalanceDeposit

Загальна логіка класу має наступний вигляд:

1. Отримання та перевірка залогіненості користувача.
2. Введення суми поповнення та оновлення інформації про користувача.
3. У результаті оповнення балансу вивід інформації, про успішну операцію.

Клас ShowGoods реалізовує вивід усіх товарів, попередньо записаних в бази даних (Рисунок 3.42).

```
Ссылок: 3
public void Execute()
{
    List<ItemData> itemData = productionService.ReadAll();

    Console.WriteLine("Check out our items and prices:");
    foreach (ItemData item in itemData)
    {
        Console.WriteLine($"{item.Id + 1}. {item.Name} - {item.Price}");
    }

    Console.WriteLine("");
}

Ссылок: 2
public string PrintAction()
{
    return "Show all goods available now";
}
```

Рисунок 3.42 – Методи класу ShowGoods

Клас BuyItems (Рисунок 3.43) відповідальний за реалізацію покупок предметів із бази даних, при цьому він взаємодіє з акаунтом, предметами та записує інформацію про покупку до масиву запису даних про покупки.

```
public void Execute()
{
    // Отримуємо індекс користувача, який залогінився
    int CurrentUserId = accountService.GetCurrentUserId();

    // Перевіряємо чи він залогінений (-1 не залогінений)
    if (CurrentUserId == -1)
    {
        return;
    }

    string answer;
    do
    {
        MakePurchase(accountService, productionService, CurrentUserId);

        Console.WriteLine("Do you want to make another purchase? (Y/N)");
        answer = Console.ReadLine();

    } while (answer.ToUpper() == "Y");
}
```

Рисунок 3.43 – Класс BuyItems

У даному класі присутній приватний метод MakePurchase, який безпосередньо проводить усі записи щодо проведеної покупки, а також перевіряє, чи вистачає коштів на балансі користувача, щоб купити даний товар (Рисунок 3.44).

```
Ссылок: 1
private void MakePurchase(AccountService accountService, ProductionService productionService, int CurrentUserId)
{
    List<ItemData> itemData = productionService.ReadAll();
    List<UserAccount> usersList = accountService.ReadAll();


    Console.WriteLine("To buy something chose the items number (shows by option 5)\n" +
        "To check the product info press '0'\n");
    int answer;
    answer = int.Parse(Console.ReadLine());


    if (answer == 0)
    {
        new ShowGoods(productionService).Execute();
        return;
    }


    UserAccount CurrentUser = usersList[CurrentUserId];
    int userBalance = CurrentUser.Balance;
    int itemPrice = itemData[answer - 1].Price;


    string errorMessage = $"Sorry, but You cannot afford it\n" +
        $"Your balance is {userBalance} but the price {itemPrice} is higher for {itemPrice - userBalance}\n";
    string successMessage = $"Congratulations, your {itemData[answer - 1].Name} will be sent soon\n";


    if (userBalance < itemPrice)
    {
        Console.WriteLine(errorMessage);
        return;
    }

    CurrentUser.Balance -= itemPrice;
    accountService.Update(CurrentUser);

    string itemName = itemData[answer - 1].Name;

    var newRecord = new PurchaseHistory(CurrentUser.UserLogin, itemName, CurrentUser.Balance);

    historyService.AddPurchaseData(newRecord);

    Console.WriteLine(successMessage, $"Your current balance is {CurrentUser.Balance}\n");

    return;
}
```

Рисунок 3.44 – Приватний метод MakePurchase()

Загальна логіка виглядає наступним чином:

1. Отримання індексу та перевірка залогіненості користувача.
2. Здійснення покупок у циклі методом MakePurchase().

Метод ShowAccountHistory (Рисунок 3.45) виводить усю історію покупок залогіненого користувача.

```
Ссылок: 2
public void Execute()
{
    // Отримуємо індекс користувача, який залогінився
    int CurrentUserId = accountService.GetCurrentUserId();

    // Перевіряємо чи він залогінений (-1 не залогінений)
    if (CurrentUserId == -1)
    {
        return;
    }

    List<UserAccount> usersList = accountService.ReadAll();
    UserAccount CurrentUser = usersList[CurrentUserId];


    Console.WriteLine($"Here is the full history of {CurrentUser.UserLogin}");

    List<PurchaseHistory> purchaseList = historyService.ReadAll();


    int purchaseIndex = 1;
    foreach (PurchaseHistory purchase in purchaseList)
    {
        if (purchase.UserLogin == CurrentUser.UserLogin)
        {
            Console.WriteLine($"{purchaseIndex}) Item: {purchase.ItemName}, Balance: {purchase.CurrentBalance}");
            purchaseIndex++;
        }
    }
}
```

Рисунок 3.45 – Клас ShowAccountHistory

Загальна логіка даного класу має наступний вигляд:

1. Перевірка залогіненості користувача.
2. Отримання історії покупок усіх користувачів.
3. Друк історії покупок залогіненого користувача.

Цей клас відповідає за друк строки з подякою за використання магазину (Рисунок 3.46). Оскільки він іде у масиві в головній функції Main() під індексом 0, при його виклику, відповідно до логіки головної функції, відбувається завершення роботи програми. Також має відповідну назву «Закрити магазин».

```
Ссылок: 2
public void Execute()
{
    Console.WriteLine("Thank you for choosing our shop!");
}
```

Рисунок 3.46 – Клас ExitShop

3.8 Результат Роботи Програми

Далі виконується рогляд взаємодії з програмою під час її використання. Заради більш детального розгляду будемо буде вводитися некоректа інформація, таким чином розглядаються такі випадки, як залогінитися в незареєстрований логін, введення неправильного паролю тощо.

Під час запуску програми відкривається вікно з виводом опцій програми (Рисунок 3.47).

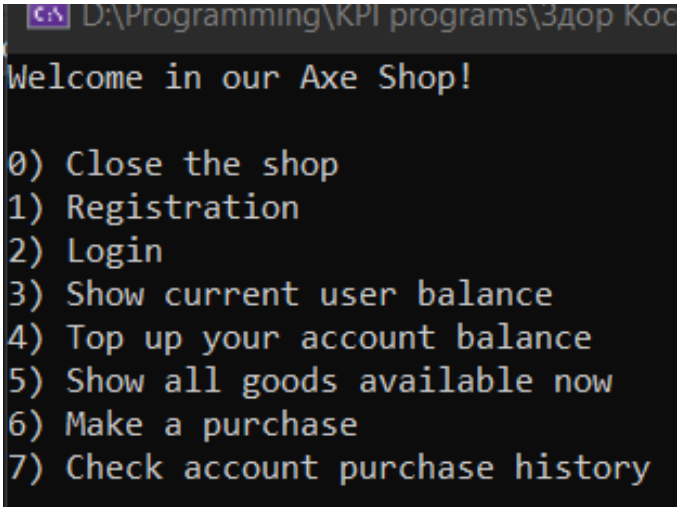

Рисунок 3.47 – Меню опцій

Функція реєстрації користувача дає можливість зареєструвати акаунт у базу даних (Рисунок 3.48)

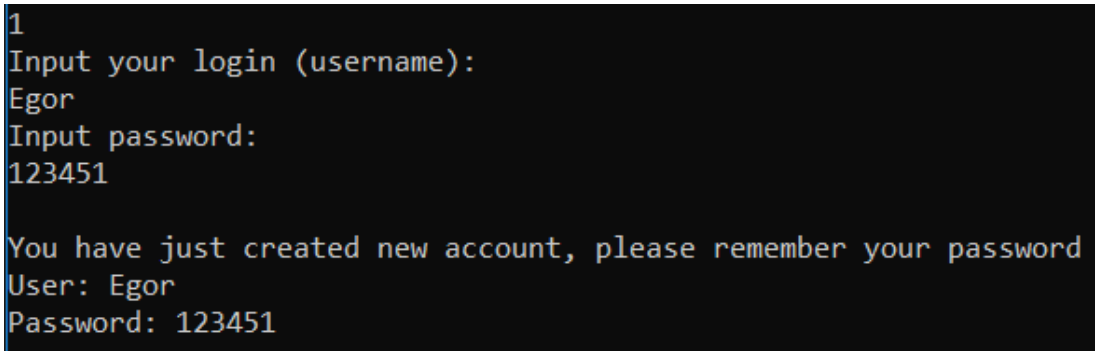

Рисунок 3.48 –Функція Реєстрації користувача

Функція логіну користувача дає змогу залогінитися. Спочатку продемонстровано введення неправильного логіну, через це програма запросила спробувати ще раз. Потім із паролем так само. У результаті, після введення всіх правильних даних користувач увійшов у систему (Рисунок 3.49).

Рисунок 3.49 – Функція Входу до акаунту

Функція перегляду балансу користувача, повертає поточний баланс користувача (Рисунок 3.50).

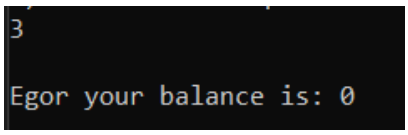

Рисунок 3.50 – ФугкціяПерегляду баланса користувача

Функція поповнення балансу користувача поповнює рахунок користувача і виводить повідомлення про результат (Рисунок 3.51).

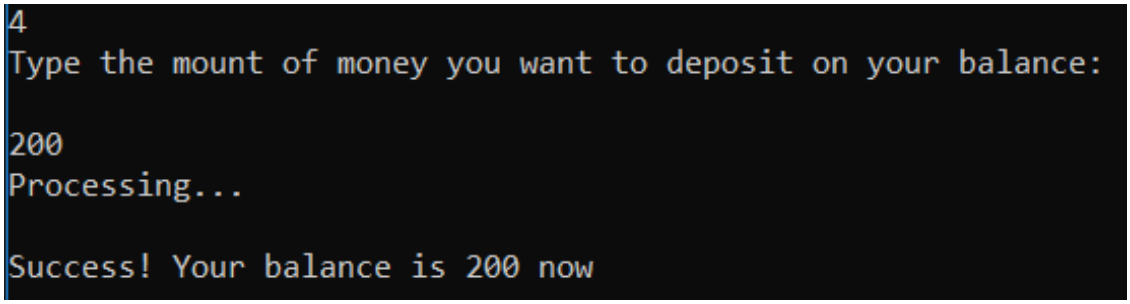

Рисунок 3.51 – Функція Поповнення балансу користувача

Функція друку списку товарів виводить список усіх доступних товарів для покупки (Рисунок 3.52).

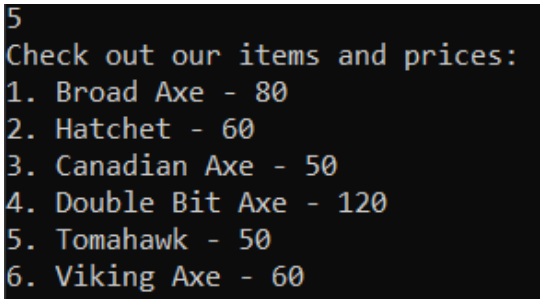

Рисунок 3.52 – Функція Друку списку товарів

Функція здійснення покупок надає можливість здійснювати безпосередньо покупку товарів, а також вивести список доступних товарів (Рисунок 3.53).

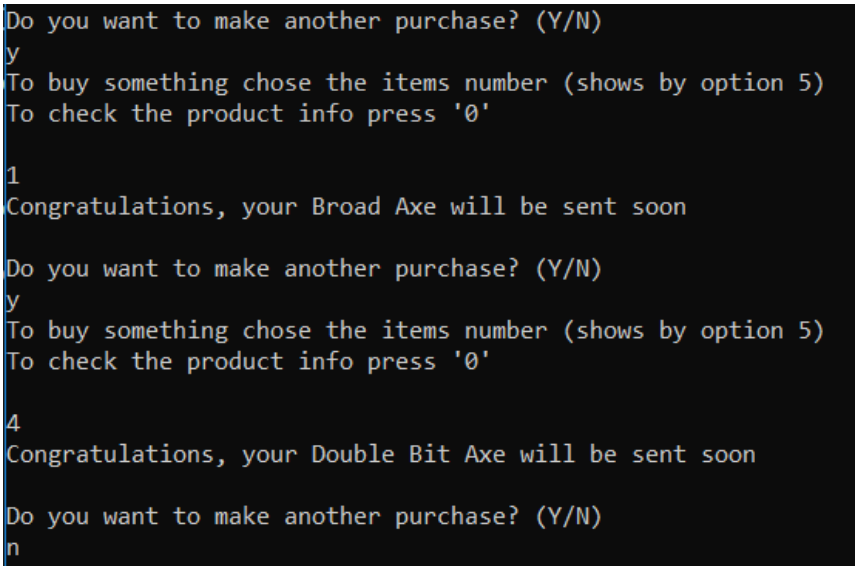

Рисунок 3.53 – Продовження покупок

Функція перегляду історії покупок повертає повну історію покупок даного користувача, де показує назву купленого товару і який став баланс користувача в результаті покупки (Рисунок 3.54).

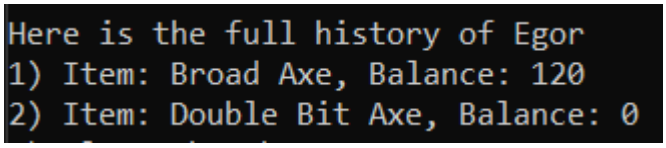

Рисунок 3.54 – Функція Перегляду історії покупок

Функція виходу з програми (Рисунок 3.55) виконує вихід із програми та дякує за використання даного магазину.

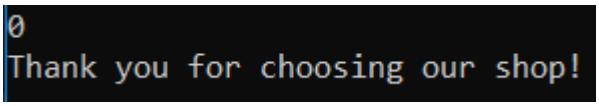

Рисунок 3.55 – Функція Виходу з програми

Отже, під час взаємодії з програмою були розглянуті різні сценарії, включаючи спроби введення некоректних даних, таких як залогінення в незареєстрований у базу даних логін чи введення неправильного паролю. Запуск програми відкриває вікно з меню опцій, що дозволяє користувачам вибирати різні функції.

Процес реєстрації користувача представлений функцією реєстрації, яка дозволяє створювати акаунт у базі даних. Функція логіну користувача дозволяє залогінитися, демонструючи обробку неправильного логіну чи паролю та успішний вхід.

Інші функції, такі як перегляд балансу, поповнення балансу та друк списку товарів, працюють ефективно, забезпечуючи користувачам доступ до важливих опцій магазину. Функція здійснення покупок і перегляд історії покупок демонструють коректну обробку та виведення інформації про покупки.

Загальним висновком є те, що програма демонструє працездатність та готовність до використання, забезпечуючи користувачам зручний інтерфейс для взаємодії з магазином і його функціональністю.

ВИСНОВКИ

У результаті виконання даної курсової роботи було реалізовано і детально розглянуто код інтернет-магазину, написаного на мові програмування C# із використанням ООП. Розглянуті фрагменти коду стосувалися основних аспектів системи, таких як реєстрація користувачів, авторизація, робота з товарами та історією покупок.

Застосунок дозволяє користувачам здійснювати реєстрацію, авторизацію, перегляд балансу, поповнення рахунку, вибір та покупку товарів, а також перегляд історії своїх покупок.

Основна логіка програми розділена на сервіси, репозиторії, моделі та функції інтерфейсу користувача. Кожен з цих компонентів взаємодіє між собою, створюючи зручну та функціональну структуру.

Блок-схеми, які були створені, графічно ілюструють роботу окремих частин програми. Вони слугують важливим інструментом для розуміння логіки та потоків даних в системі.

Основний функціонал, такий як реєстрація користувачів, керування балансом, перегляд та придбання товарів, а також ведення історії покупок, реалізовані ефективно й мають можливість викликатися за потреби користувача.

Загальна структура програми дозволяє легко розширювати та модифікувати функціональність магазину. Важливою перевагою є також використання патерну репозиторію та сервісної архітектури, що сприяє високій модульності та тестовій придатності коду.

Загальний дизайн програми є інтуїтивно зрозумілим, дозволяючи користувачеві легко взаємодіяти з магазином через консольний інтерфейс.

Отже, в даному проєкті вдало використовується низка принципів програмування та архітектурних підходів, що сприяє покращенню продуктивності та розширюваності системи.

ВИКОРИСТАНІ ДЖЕРЕЛА

1. Dan Clark, Beginning C# Object-Oriented Programming (Expert's Voice in .NET) 2nd Edition. Apress, 2013, 408 с.
2. Jon Skeet , C# in Depth: Fourth Edition 4th Edition. Manning, 2019, 528 с.
3. Joseph Albahari, C# 10 in a Nutshell: The Definitive Reference. O'Reilly Media, 2022, 1058 с.
4. Philip Japikse, Pro C# 7: With .NET and .NET Core. Apress, 2017, 1437 с.
5. Simon Kendal, Object Oriented Programming using C#. Apress, 2019, 254 с.
6. Mike McGrath, Programming in Easy Step. In Easy Steps Limited, 2011, 192 с.
7. Robert Martin, Micah Martin, Agile Principles, Patterns, and Practices in C# 1st Edition. Pearson, 2006, 768 с.
8. Grant Palmer, Ken Barker, Beginning C# 2008 Objects: From Concept to Code (Expert's Voice in .NET) 1st ed. Edition. Apress, 2008, 680 с.
9. Mary Delamater, Joel Murach, Murach's Asp.net Core Mvc 2nd Edition. Mike Murach and Associates, 2022, 810 с.
10. Andrew Stellman, Jennifer Greene, Head First C#: A Learner's Guide to Real-World Programming with C# and .NET Core 4th Edition. O'Reilly Media, 2021, 785 с.

ДОДАТОК

Весь проєкт із блок-схемами включно повністю завантажено на сервіс Github. Повне посилання на проєкт:

https://github.com/Egorkud/Internet-Shop-Coursework.git