Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: Розробка прикладного програмного інтерфейсу для формування заявок на постачання товарів з урахуванням дистрибуційних факторів»

Виконав: студент 4 курсу
групи 3ПІ-21б спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Думенко А.М.
(прізвище та ініціали)
Керівник: д.т.н., проф. каф. ПЗ Романюк О. Н.
(прізвище та ініціали)
Рецензент: к.т.н., доц. каф. ОТ Снігур А. В.
(прізвище та ініціали)

Допущено до захисту
Зав. кафедри ПЗ
д.т.н., проф. О. Н. Романюк
(ініціали та прізвище)
«09» 06 2025 р

ВНТУ – 2025

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
Романюк О. Н.
«21» березня 2025 р.

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

Думенко Анастасії Миколаївні

1. Тема роботи – «Розробка прикладного програмного інтерфейсу для формування заявок на постачання товарів з урахуванням дистрибуційних факторів»

Керівник роботи: Романюк Олександр Никифорович, д.т.н., професор кафедри ПЗ, затверджені наказом вищого навчального закладу від 20 березня 2025 р. № 97.

2. Строк подання студентом роботи 2 червня 2025 р.

3. Вихідні дані до роботи: середовище розробки IntelliJ IDEA Community Edition 2025; мова розробки Java; операційна система – Windows 10/11.

4. Зміст розрахунково-пояснювальної записки: вступ; аналіз розвитку технологій автоматизованого формування заявок на постачання товарів та постановка задач дослідження; розробка моделей та алгоритмів програмного продукту; розробка структури та програмних модулів системи формування заявок; висновки; список використаних джерел; додатки.

5. Перелік графічного матеріалу: титульний слайд – автор, науковий керівник бакалаврської кваліфікаційної роботи, тема; актуальність розробки; мета, об'єкт та предмет дослідження; постановка задачі, наукова новизна, практична цінність одержаних результатів; порівняльний аналіз аналогів, структура застосунку; метод динамічного коригування замовлень; метод інтеграції коефіцієнта рангової кореляції Кендалла для прогнозування обсягів продажів діаграми діяльності програмного додатку; модель застосунку у вигляді діаграми діяльності; модель застосунку у вигляді діаграми послідовностей; алгоритм розрахунку коефіцієнту сезонності; алгоритм обчислення коефіцієнту Кендалла; модуль ранжування товарів; модуль розрахунку товарів для

замовлення; структурна схема бази даних; функціонал розробленої системи; апробація та публікація результатів роботи; висновки; фінальний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Романюк О.Н., д.т.н., професор кафедри ПЗ	21.03.25	02.06.25

7. Дата видачі завдання 21 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз розвитку технологій автоматизованого формування заявок на постачання товарів та постановка задач дослідження	25.03.25-08.04.25	вик.
2	Розробка моделі та алгоритмів програмного продукту	09.04.25 - 20.04.25	вик.
3	Варіантний аналіз та обґрунтування вибору засобів програмної реалізації	21.04.25 - 23.04.25	вик.
4	Розробка програмних модулів та методів реалізації системи формування заявок	24.04.25 - 20.05.25	вик.
5	Тестування застосунку	21.05.25 - 25.05.25	вик.
6	Оформлення матеріалів до захисту БКР	26.05.25 - 30.05.25	вик.

Студент _____ **Думенко А. М.**
(підпис) (прізвище та ініціали)

Керівник бакалаврської кваліфікаційної роботи _____ **Романюк О. Н.**
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота обсягом 60 сторінок формату А4 містить 20 рисунків, 4 таблиці та список використаних джерел із 30 найменувань.

Метою роботи є автоматизація процесу формування заявок на постачання товарів шляхом розробки прикладного програмного інтерфейсу, який інтегрує дистрибуційні фактори, зокрема сезонні коливання попиту, географічний розподіл та логістичні обмеження. В основі обробки історичних даних використано метод рангової кореляції Кендалла, що дозволяє підвищити точність прогнозування та ефективність планування.

У рамках дослідження спроєктовано та реалізовано програмні засоби прикладного програмного інтерфейсу, які забезпечують автоматизоване формування заявок із врахуванням залишків на складах, історії продажів та логістичних характеристик. Система дозволяє оптимізувати процеси управління поставками, знижуючи час обробки заявок та підвищуючи точність розрахунків.

Запропоноване рішення може бути інтегроване у бізнес-процеси логістичних операторів і торговельних підприємств для автоматизації управління ланцюгами постачання, сприяючи підвищенню продуктивності та адаптивності до змін ринку.

Ключові слова: прикладний програмний інтерфейс, автоматизація логістики, управління запасами, прогнозування попиту, дистрибуційні фактори, обробка заявок.

ABSTRACT

The bachelor's qualification thesis comprises 60 A4 pages and includes 20 figures, 4 tables, and a list of 30 references.

The aim of the study is to automate the process of generating supply requests by developing an application programming interface that integrates distribution factors such as seasonal demand fluctuations, geographical distribution, and logistical constraints. Historical sales data is processed using Kendall's rank correlation method, which improves forecasting accuracy and planning efficiency.

As part of the research, software tools for the application programming interface were designed and implemented to enable automated request generation based on warehouse stock levels, sales history, and logistical characteristics. The system allows for the optimization of supply chain management processes by reducing request processing time and increasing calculation accuracy.

The proposed solution can be integrated into the business processes of logistics operators and commercial enterprises to automate supply chain management, thereby enhancing productivity and adaptability to market changes.

Keywords: application programming interface, logistics automation, inventory management, demand forecasting, distribution factors, request processing.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. У сучасних умовах високої турбулентності глобального ринку постачань одним із ключових факторів забезпечення стабільності та конкурентоспроможності підприємств виступає ефективне управління товарними запасами. У ланцюгу постачання надзвичайно важливою ланкою є процес формування заявок на постачання товарів, який потребує врахування низки змінних: сезонних коливань попиту, територіального розміщення складів, логістичних обмежень, фінансових витрат, особливостей контрактів із постачальниками та аналізу історичних даних про реалізацію продукції [1].

Однак традиційні інструменти, зокрема ручне планування або використання типових модулів ERP-систем часто не мають належного рівня точності. Вони або не враховують специфіку дистрибуційних факторів, або потребують ручного налаштування, що призводить до неефективного використання ресурсів, зниження точності прогнозів і зростання операційних витрат. Застосування суб’єктивних методів ухвалення рішень у такому складному середовищі часто є джерелом помилок, які мають значні наслідки в масштабах всієї логістичної системи.

Крім того, наявні рішення не завжди здатні моделювати складні й нелінійні залежності між зовнішніми умовами та попитом. Наприклад, вплив сезонності, економічних коливань або специфіки споживчих вподобань у різних регіонах часто не враховується або реалізується надто спрощено. Це обмежує адаптивність систем до швидких змін у попиті та знижує точність розрахунків.

Окрему проблему становить недостатня інтегрованість сучасних автоматизованих систем із зовнішніми службами через обмежену функціональність або відсутність відкритих прикладних програмних інтерфейсів. Це ускладнює реалізацію наскрізної цифрової трансформації логістичних процесів на підприємствах.

Важливою передумовою для подолання зазначених проблем є впровадження статистичних методів, зокрема коефіцієнта рангової кореляції Кендалла, що дозволяє виявляти кореляційні зв'язки між дистрибуційними факторами та обсягами продажів. На відміну від класичних моделей регресії, метод Кендалла є непараметричним, що робить його більш стійким до нестандартного розподілу даних, шумів та аномалій, які часто трапляються в реальних бізнес-сценаріях.

Разом з тим, інтеграція таких методів у прикладні програмні рішення автоматизації процесів постачання залишається недостатньо розробленою, особливо в аспектах масштабованості, адаптивності до змін у базах даних, багатофакторного аналізу та актуалізації даних у режимі реального часу.

Таким чином, розробка програмного забезпечення на основі сучасних підходів до обробки даних, прогнозування попиту та автоматизації планування заявок із використанням гнучкого прикладного програмного інтерфейсу є актуальною науковою та практичною задачею. Вона дозволяє покращити управління запасами, скоротити витрати та сформувати передумови для переходу підприємств до інтелектуальних логістичних систем нового покоління.

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

Мета та завдання дослідження. Метою роботи є автоматизація процесу формування заявок на постачання товарів шляхом розробки прикладного програмного інтерфейсу, який інтегрує дистрибуційні фактори (сезонність, географічний розподіл, логістичні обмеження), аналізує історичні дані про продажі за допомогою алгоритму рангової кореляції Кендалла. Автоматизація дозволить підвищити продуктивність обробки заявок.

Основними задачами дослідження є:

– провести аналіз існуючих методів автоматизації логістичних процесів для виявлення обмежень в урахуванні дистрибуційних факторів та соціальних потреб споживачів;

– запропонувати метод інтеграції коефіцієнта рангової кореляції Кендалла

для прогнозування обсягів продажів з урахуванням сезонних коливань та соціально-економічних тенденцій;

– розробити програмного модуля для автоматизації взаємодії між системами для автоматизації формування заявок із забезпеченням гнучкості та сумісності з системами планування ресурсів підприємства та системами управління взаєминами з клієнтами;

– розробити реляційну модель бази даних для структурованого зберігання інформації про товари, залишки на складах, історію продажів, параметри постачальників та логістичні обмеження;

– розробити механізм ранжування товарів із використанням алгоритму Кендалла для розрахунку коефіцієнта залежності обсягів продажів товарів певної категорії від місяця року;

– розробити механізм динамічного планування замовлень, який враховує прогнозований попит та наявні запаси товарів на складі;

– розробити модуль тестування для перевірки стабільності програмного інтерфейсу.

Об'єкт дослідження – процес автоматизації формування заявок на постачання товарів у системах управління ланцюгами постачань.

Предмет дослідження – методи та засоби оптимізації логістичних процесів на основі прикладного програмного інтерфейсу.

Методи дослідження. У процесі досліджень використовувались: теорія інформаційних системи та баз даних, математична статистика, об’єктно-орієнтоване програмування для розробки методів і програмних засобів автоматизації формування заявок, комп’ютерне моделювання для аналізу та перевірки отриманих теоретичних положень.

Наукова новизна отриманих результатів.

1. Уперше запропоновано метод, особливість якого полягає у використанні метода Кендалла для системи формування заявок, що дозволило врахувати сезонні коливання попиту та зменшити похибку прогнозу порівняно з традиційними методами.

2. Вперше розроблено метод динамічного коригування замовлень, який комбінує аналіз залишків на складі, історію продажів та логістичні обмеження, що підвищило точність планування.

3. Подальшого розвитку набув метод ранжування товарів, у якому на відміну від існуючого, використано ітераційні формули для оптимізації обчислювальних ресурсів, що забезпечило зниження часу генерації заявок.

Практичне значення отриманих результатів полягає в тому, що на основі теоретичних положень в бакалаврській кваліфікаційній роботі запропоновано алгоритми та розроблено програмне забезпечення автоматизованого формування заявок на постачання товарів. Впровадження системи забезпечує зменшення витрат часу на планування, підвищення точності прогнозів попиту, зниження ризиків дефіциту та надлишку товарів.

Особистий внесок здобувача. Усі наукові результати, викладені у БКР, отримані автором особисто. У друкованих працях, опублікованих у співавторстві, автору належать такі результати: метод інтеграції соціальних потреб у логістичні процеси [2], архітектура модуля оптимізації логістичних маршрутів [3], впровадження системи пріоритетного розподілу замовлень з урахуванням дистрибуційних факторів [4].

Апробація матеріалів бакалаврської дипломної роботи. Результати роботи доповідались на «Всеукраїнській науково-технічній конференції факультету інформаційних технологій та комп'ютерної інженерії», «XVIII Всеукраїнській науково-практичній конференції аспірантів, студентів та молодих вчених» та «XXXIII Міжнародній науково-практичній конференцій MicroCAD» у 2025 році.

Публікації. За тематикою досліджень опубліковано 3 наукові праці в матеріалах конференцій: «Всеукраїнській науково-технічній конференції факультету інформаційних технологій та комп'ютерної інженерії», «XVIII Всеукраїнській науково-практичній конференції аспірантів, студентів та молодих вчених» та «XXXIII Міжнародній науково-практичній конференцій MicroCAD» у 2025 році.

1 АНАЛІЗ РОЗВИТКУ ТЕХНОЛОГІЙ АВТОМАТИЗОВАНОГО ФОРМУВАННЯ ЗАЯВОК НА ПОСТАЧАННЯ ТОВАРІВ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Аналіз сучасних інформаційних технологій для автоматизації управління постачаннями

Сучасний розвиток інформаційних технологій суттєво впливає на процеси управління ланцюгами постачання, роблячи їх більш гнучкими, прогнозованими та ефективними. Зростання вимог до швидкості обробки замовлень, точності прогнозування попиту та зниження логістичних витрат стимулює компанії до впровадження автоматизованих рішень. Традиційні методи, що базуються на ручному введенні даних, фіксованих графіках закупівель та емпіричних оцінках, частіше поступаються місцем інтелектуальним технологіям, які забезпечують адаптивність до ринкових змін, сезонних коливань попиту, вартості ресурсів та логістичних затримок.

Одним із важливих напрямів цифрової трансформації є впровадження хмарних рішень у сфері логістики. Завдяки хмарним платформам дані про поставки, залишки товарів та прогнозування попиту можуть бути синхронізовані в режимі реального часу між усіма учасниками логістичного процесу [5]. Це дозволяє скоротити витрати на локальну інфраструктуру та підвищити доступність систем із будь-якої точки світу.

Автоматизація процесів управління постачаннями забезпечує більш точне прогнозування потреб, оптимізацію складських запасів та скорочення часу обробки заявок. Важливу роль у цьому процесі відіграє впровадження прикладного програмного інтерфейсу, що дозволяє інтегрувати різні інформаційні системи підприємств і забезпечувати обмін актуальними даними між усіма учасниками ланцюга постачання від виробників до кінцевих споживачів [6]. Такі інтерфейси сприяють виявленню дефіциту товарів, аналізу продажів і оптимізації логістики в реальному часі, що є основою ефективного управлінського рішення.

Особливу увагу приділяють також питанням інформаційної безпеки. З розвитком інтегрованих IT-систем для управління постачаннями зростає і ризик витоку або викривлення даних. Тому сучасні платформи включають засоби аутентифікації, шифрування, резервного копіювання та багаторівневого контролю доступу. Це дозволяє гарантувати безпеку даних на всіх етапах логістичного процесу.

Одним із перспективних напрямів у сфері автоматизації управління постачаннями є використання методів штучного інтелекту та аналізу великих даних [7]. Алгоритми машинного навчання дозволяють здійснювати глибокий аналіз історичних даних, виявляти закономірності змін попиту, прогнозувати ризики дефіциту або надлишку товарів, а також підбирати оптимальні стратегії закупівель [8]. Такі підходи значно підвищують точність планування, знижують витрати та сприяють адаптації логістичних процесів до змін у зовнішньому середовищі.

Крім внутрішніх параметрів, сучасні автоматизовані системи можуть враховувати й зовнішні фактори: тенденції ринку, зміни валютних курсів, вартість транспортування, погодні умови, геополітичні ризики. Це забезпечує можливість формування адаптивних стратегій управління, що відповідають поточній економічній ситуації [9]. Наприклад, логістичні компанії, які використовують інтеграцію з метеосервісами або GPS-моніторинг транспорту, здатні коригувати маршрути в реальному часі, що забезпечує зниження витрат і підвищення точності доставки.

Також слід відзначити важливість інтелектуальних систем підтримки прийняття рішень, які поєднують алгоритмічну обробку даних із можливістю вручну коригувати сценарії постачання. Це дає змогу логістичним фахівцям поєднувати переваги автоматизації та людської експертизи.

Отже, сучасні інформаційні технології стають невід’ємною частиною систем управління постачаннями. Вони не лише автоматизують процеси, а й забезпечують прийняття рішень на основі аналітики, підвищують гнучкість логістичних ланцюгів і дозволяють компаніям швидше адаптуватися до змін.

Таким чином, впровадження інтелектуальних систем прогнозування, інтеграційних рішень на базі прикладного програмного інтерфейсу та модулів машинного навчання є запорукою для досягнення високого рівня продуктивності процесів, надійності та конкурентоспроможності у сфері логістики.

1.2 Порівняльний аналіз аналогів

Аналіз сучасних програмних систем управління ланцюгами поставок дозволяє визначити переваги та недоліки існуючих інструментів, що формує основу для створення конкурентоздатного продукту, орієнтованого на конкретні потреби бізнесу. Для дослідження були обрані провідні платформи**:** SAP SCM**,** Oracle SCM Cloud**,** Blue Yonder (JDA Software) та Manhattan Associates**.**

1. SAP SCM – це одна з найпоширеніших систем управління ланцюгами постачання, що надає потужний інструментарій для планування, управління логістикою, закупівлями та прогнозування попиту (рис. 1.1). Вона дозволяє автоматизувати процеси управління запасами, що значно скорочує витрати підприємств та підвищує точність прогнозування. Однак, головним недоліком SAP SCM є висока вартість впровадження та складність налаштування, яка є проблемою для середніх та малих підприємств. [10].

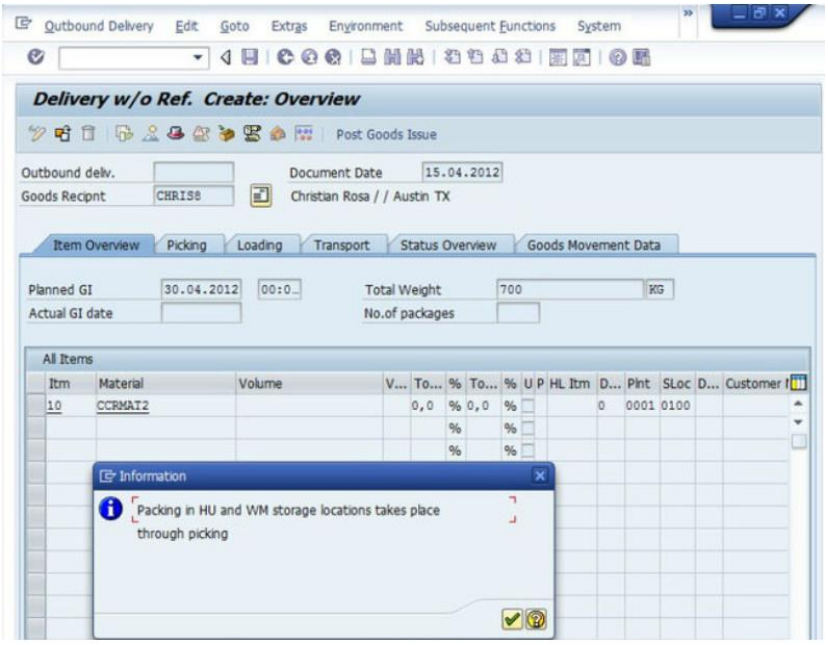


Рисунок 1.1 – Приклад роботи застосунку SAP SCM

2. Oracle SCM Cloud є хмарним рішенням для управління ланцюгами постачання, яке пропонує широкий функціонал для оптимізації логістичних процесів (див. рисунок 1.2). Його головною перевагою є гнучкість та можливість масштабування відповідно до потреб компанії. Водночас, система може мати високі вимоги до IT-інфраструктури підприємства, що ускладнює її впровадження без відповідної підготовки.

Хмарна архітектура Oracle SCM Cloud забезпечує високу доступність і захищеність даних, що особливо актуально для глобальних компаній з розгалуженою мережею постачань. Проте, через велику кількість конфігурацій і необхідність налаштування під конкретні бізнес-процеси, впровадження системи може потребувати залучення зовнішніх фахівців або сертифікованих консультантів. Це підвищує вартість володіння рішеням, особливо для компаній середнього масштабу [11].

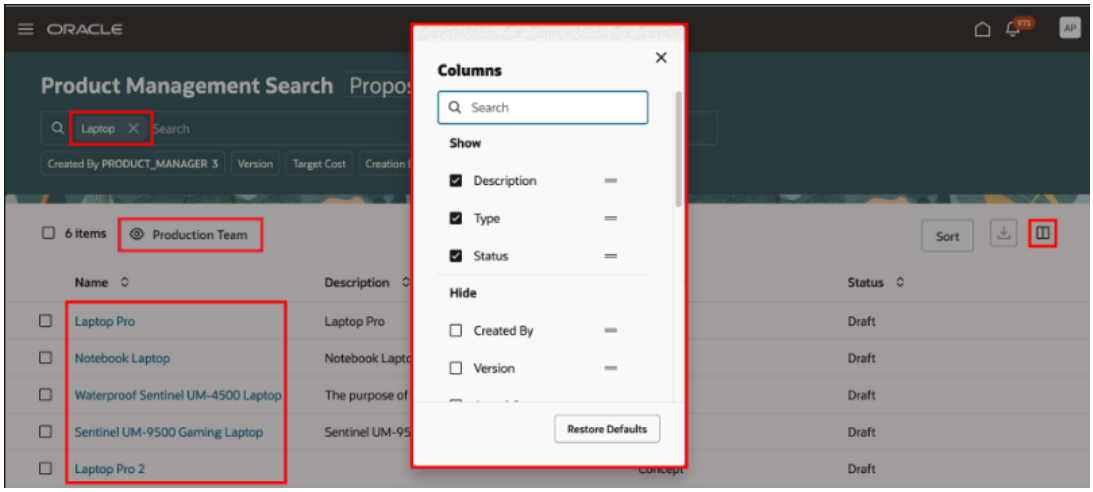


Рисунок 1.2 – Панель пошуку та налаштувань відображення пропозицій у системі Oracle SCM Cloud

3. Blue Yonder (JDA Software) – це потужна система управління ланцюгами постачання, яка активно використовує технології штучного інтелекту, машинного навчання та прогнозної аналітики для оптимізації процесів планування попиту, управління запасами та логістики (див.рис. 1.3). Основною перевагою цієї системи є високий рівень автоматизації, який дозволяє мінімізувати вплив людського фактора та зменшити кількість помилок при

прийнятті рішень [12]. Основною функцією Blue Yonder є здатність прогнозувати попит на основі багатофакторного аналізу: враховуються сезонні коливання, промоакції, регіональні особливості споживання та інші зовнішні чинники.

До недоліків системи можна віднести її складність у впровадженні, що потребує значних ресурсів як у технічному, так і в навчальному аспектах. Крім того, високі ліцензійні витрати можуть обмежити її застосування для малих та великих підприємств.

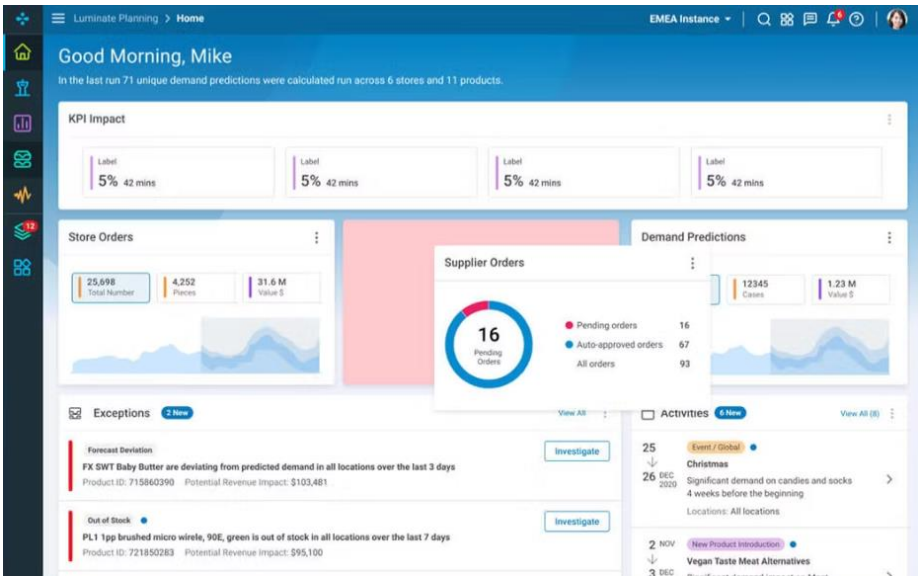


Рисунок 1.3 – Інтерфейс головної сторінки системи Blue Yonder

4. Manhattan Associates пропонує комплексні рішення для управління складською логістикою та постачаннями (див. рисунок 1.4). Основною особливістю цієї системи є її орієнтація на оптимізацію складських операцій, що дозволяє автоматизувати процеси зберігання, комплектування та відправлення товарів. Завдяки використанню аналітичних алгоритмів система забезпечує ефективне управління запасами та швидку адаптацію до змін попиту.

Manhattan Associates активно використовується великими корпораціями, що працюють у сфері роздрібної торгівлі та дистрибуції. Основний недолік – висока вартість програмного забезпечення та необхідність навчання персоналу для його ефективного використання [13].

Рисунок 1.4 – Інтерфейс панелі управління системи Manhattan Associates для моніторингу показників продуктивності складу

З метою оцінки ефективності сучасних рішень у сфері управління ланцюгами постачання було здійснено порівняльний аналіз найбільш поширених систем, що активно застосовуються на ринку. Проведене дослідження спрямоване на виявлення сильних і слабких сторін кожного рішення, а також на визначення їх відповідності актуальним потребам підприємств у сфері логістики та постачання.

У процесі аналізу було враховано основні критерії, які мають важливе значення для забезпечення ефективності управління: точність прогнозування попиту, рівень автоматизації процесів, можливість інтеграції з існуючими ERP-системами, гнучкість налаштувань під специфіку бізнесу, а також загальна вартість впровадження обраної системи.

Отримані результати дозволяють сформувати об’єктивне уявлення про переваги та обмеження кожного з досліджених рішень. На основі зібраної інформації було складено порівняльну таблицю 1.1, яка відображає основні характеристики систем та їх відповідність основним критеріям оцінки.

Таблиця 1.1 – Порівняльна характеристика аналогів

Критерії оцінювання	SAP SCM	Oracle SCM Cloud	Blue Yonder	Manhattan Associates	Власний програмний інтерфейс
Точність прогнозування	1	1	1	1	1
Автоматизація управління	1	1	1	1	1
Інтеграція з ERP-системами	1	1	1	1	1
Гнучкість налаштувань	0	0	1	0	1
Масштабованість рішення	1	1	1	1	1
Простота впровадження та налаштування	0	0	0	0	1
Гнучкість інтеграції з іншими сервісами	0	1	1	0	1
Вартість впровадження	0	0	0	0	1
Загальна оцінка	4	5	6	4	8

Відповідно до проведеного порівняльного аналізу, розробка власного програмного інтерфейсу для автоматизації формування заявок на постачання товарів є доцільною та перспективною. Дослідження показало, що жодна з існуючих систем не забезпечує повноцінного поєднання функцій для врахування дистрибуційних факторів.

Розширений функціонал, зокрема гнучкість налаштувань, масштабованість рішення та проста інтеграція з іншими сервісами, сприяє ефективному управлінню ланцюгами постачання. Використання автоматизованих засобів управління сприяє зменшенню впливу людського фактору на процеси постачання, що, у свою чергу, знижує ризики помилок і підвищує точність прогнозування. Завдяки централізованому моніторингу та координації

логістичних операцій досягається підвищення ефективності управління запасами, скорочення витрат та оптимізація термінів доставки товарів.

Окрім поточних можливостей, запропонована система має значний потенціал для подальшого вдосконалення та адаптації до індивідуальних потреб конкретного підприємства. Здатність до інтеграції з сучасними ERP-системами, можливість підключення аналітичних інструментів та розширення функціональних можливостей забезпечують створення гнучкого та масштабованого програмного продукту, здатного відповідати вимогам динамічного ринку.

Отже, результати проведеного аналізу свідчать про доцільність впровадження спеціалізованого програмного забезпечення, орієнтованого на покращення процесів постачання. Запропонований підхід дозволить підвищити загальну ефективність логістичних операцій, зменшити витрати ресурсів та забезпечити більш скоординовану взаємодію між усіма учасниками ланцюга постачання.

1.3 Аналіз методів розв’язання задачі

Розробка ефективних програмних засобів для автоматизації формування заявок на постачання товарів потребує ретельного аналізу та обґрунтованого вибору методологічних підходів, що забезпечать високу продуктивність системи, зручність експлуатації та відповідність предметній галузі. На підставі комплексного дослідження функціональних вимог, пов'язаних із прогнозуванням попиту, управлінням товарними запасами та оптимізацією логістичних операцій, було обґрунтовано та впроваджено технологічні рішення.

1. Алгоритм ранжування товарів на основі коефіцієнта рангової кореляції Кендалла. Даний метод дозволяє здійснювати об'єктивну пріоритезацію товарних позицій при формуванні заявок на постачання. Алгоритм враховує комплекс критеріїв, включаючи рівень поточного попиту, стратегічну значимість товарних груп для бізнесу, а також наявні логістичні обмеження.

Застосування цього підходу забезпечує динамічну адаптацію системи до змін ринкової кон'юнктури та дозволяє оптимізувати черговість поставок.

2. Алгоритм розрахунку оптимальних обсягів замовлення. Для категорій товарів з вираженою сезонністю попиту розроблено спеціалізований алгоритм, що враховує коефіцієнти рангової кореляції Кендалла при визначенні обсягів замовлення. Для товарів зі стабільним попитом застосовується методика розрахунку на основі середньої величини продажів за ретроспективний період. Такий підхід дозволяє мінімізувати ризики як дефіциту, так і надлишкового накопичення запасів.

3. Система управління даними на базі СУБД MySQL. Для зберігання та обробки структурованої інформації (характеристики товарів, історичні дані про продажі, параметри постачальників) обрано реляційну систему керування базами даних MySQL. Вибір обумовлений високою надійністю даної СУБД, її здатністю ефективно обробляти складні запити, а також підтримкою складних міжтабличних зв'язків.

4. REST-архітектура програмного інтерфейсу. Розробка програмного інтерфейсу здійснена з використанням архітектурного стилю REST, що забезпечує гнучкість, масштабованість та модульність системи. Даний підхід дозволяє ефективно інтегрувати розроблене рішення з існуючими корпоративними системами та зовнішніми сервісами.

5. Аналітичний модуль прогнозування попиту. Система включає спеціалізований модуль, що реалізує методи статистичного аналізу (ковзне середнє, регресійний аналіз) для прогнозування динаміки попиту на основі історичних даних. Цей модуль дозволяє оптимізувати обсяги замовлень та підвищити ефективність управління товарними запасами.

Отже, застосування вказаного комплексу методів та технологій забезпечує створення цілісної, адаптивної системи автоматизованого формування заявок на постачання. Обрані рішення відповідають сучасним вимогам до точності, гнучкості та масштабованості, дозволяють ефективно інтегруватися в бізнес-

процеси підприємств та забезпечують оперативне реагування на зміни ринкових умов.

1.4 Постановка задач розробки

Метою бакалаврської кваліфікаційної роботи є оптимізація процесу формування заявок на постачання товарів шляхом розробки програмного інтерфейсу, який інтегрує дистрибуційні фактори.

На основі попереднього аналізу вимог, дослідження сучасних технологій розробки RESTful API та аналізу аналогічних рішень (розділи 1.1–1.2) було визначено перелік задач, які необхідно реалізувати в рамках дипломної роботи.

До основних задач розробки належать:

1. Розробка функціоналу для перегляду інформації про товари, залишки та історію продажів. Програмний інтерфейс повинен забезпечувати доступ до структурованих даних про асортимент товарів, наявність на складах, параметри постачання та статистику попередніх замовлень.

2. Розробка функціоналу створення та редагування заявок завдяки якому користувачі повинні мати можливість створювати нові заявки на постачання товарів, редагувати існуючі, а також отримувати інформацію щодо замовлення на основі попиту та залишків. Заявки мають зберігатися в базі даних та бути доступними для перегляду та подальшої аналітики.

3. Інтеграція коефіцієнта рангової кореляції Кендалла для прогнозування обсягів продажів з реалізацію алгоритму, що оцінює взаємозв’язок між категорією товарів і періодичністю попиту на основі історичних даних та дозволяє враховувати сезонність і соціально-економічні зміни при формуванні замовлень.

4. Розробка реляційної моделі бази даних. Необхідно створити базу даних для зберігання інформації про товари, запаси, постачальників, логістичні обмеження та результати аналітичної обробки. База повинна підтримувати швидкий доступ і забезпечувати цілісність даних.

5. Розробка алгоритму динамічного планування замовлень. Система повинна враховувати прогнозований попит, поточні залишки, обмеження в логістиці, а також соціальні чинники для формування раціональних замовлень, орієнтованих на реальні потреби споживачів.

6. Інтеграція з ERP/CRM системами та зовнішніми джерелами. Програмний інтерфейс має підтримувати обмін даними з іншими системами, що використовуються у підприємстві, для автоматичного оновлення інформації та узгодженого планування.

7. Розробка модуля тестування та перевірки стабільності програмного інтерфейсу. Передбачено створення автоматизованих тестів для перевірки коректності відповіді на запити, стабільності функціоналу при навантаженнях, а також обробки помилок.

Отже, сформульовані задачі визначають комплексний підхід до створення прикладного програмного інтерфейсу, що дозволяє підвищити ефективність логістичного управління, враховувати динамічні фактори попиту та забезпечити сумісність із сучасними інформаційними системами підприємств.

1.5 Висновки

Метою першого розділу було обґрунтування доцільності розробки програмного інтерфейсу для автоматизованого формування заявок на постачання товарів з урахуванням дистрибуційних факторів, а також формулювання основних функціональних та технічних вимог до системи.

У процесі аналізу сучасних програмних рішень, таких як SAP SCM, Oracle SCM Cloud, Blue Yonder та Manhattan Associates, було встановлено, що платформи пропонують широкий спектр функцій, але також мають суттєві обмеження щодо інтеграції з ERP- та CRM-системами, адаптивності до особливостей підприємства, а також у сфері точного прогнозування попиту з урахуванням змінних логістичних умов і соціальних чинників.

Було проведено аналіз існуючих підходів до автоматизації логістичних процесів та обґрунтовано необхідність інтеграції алгоритму на основі

коефіцієнта рангової кореляції Кендалла для прогнозування обсягів продажів, що дозволяє враховувати сезонність та соціально-економічні тенденції.

На основі отриманих результатів було визначено основні завдання для реалізації: побудова архітектури прикладного інтерфейсу на основі моделі MVC, розробка алгоритму динамічного планування замовлень, створення реляційної моделі бази даних, реалізація механізмів авторизації, аналітики та ранжування товарів за місячною сезонністю.

Отже, результати першого розділу доводять доцільність розробки власного рішення у вигляді програмного інтерфейсу, що дозволяє інтеграцію з корпоративними системами, підвищення точність прогнозування попиту та забезпечення стабільності і оптимізації процесів постачання в сучасних умовах ринку.

2 РОЗРОБКА АЛГОРИТМІВ ПРОГРАМНОГО ПРОДУКТУ

2.1 Аналіз завдання та формалізація вхідних даних

У межах дослідження завдання було визначено, що система повинна забезпечувати ефективне управління товарними запасами, автоматизоване формування заявок на постачання, а також оптимізацію логістичних процесів на основі прогнозування попиту.

Для реалізації поставлених завдань розроблено серверну частину програмного забезпечення із використанням фреймворку Spring Boot [14], яка відповідає за обробку логіки взаємодії з базою даних, а також клієнтську частину, що забезпечує обмін інформацією через формат обміну даними MVS [15].

До основних функціональних вимог системи віднесено управління інформацією про товари, включаючи додавання нових позицій, редагування наявних записів, їхнє видалення, а також перегляд основних атрибутів товару. Зокрема, йдеться про найменування, категорію, постачальника, закупівельну та роздрібну ціну, фізичні характеристики (вагу, об’єм), залишки на складах і поточний статус активності. Уся відповідна інформація зберігається у структурованому вигляді у реляційній базі даних у межах сутності Product.

Важливою складовою є реалізація модуля прогнозування попиту. Він використовує статистичні дані про продажі для визначення прогнозованих обсягів реалізації у майбутні періоди з урахуванням сезонних змін та економічних чинників. Для товарів із сезонним характером попиту передбачено застосування коефіцієнта рангової кореляції Кендалла, що дозволяє виявити закономірності між місяцями року та інтенсивністю реалізації.

Автоматизоване формування заявок базується на розрахунках потреби у поповненні товарного запасу з урахуванням залишків на складах, очікуваного попиту та визначених пріоритетів. Сформовані заявки представлені у вигляді об’єктів Order з деталізацією позицій замовлення (OrderedProduct), зазначенням постачальників та запланованих обсягів поставки.

Інформація, що надходить у систему, включає дані про товари, постачальників та продажі. Джерелами таких даних виступають як користувацький інтерфейс, так і зовнішні системи, зокрема програмний інтерфейс або корпоративні бази даних. Усі вхідні дані проходять перевірку на достовірність, після чого зберігаються у відповідних структурах бази даних (Product, Supplier, SoldProduct). Обробка даних здійснюється на серверному рівні, що забезпечує логічну цілісність і цільове використання інформації в межах логістичних процесів.

Вихідні дані системи формуються як результат обробки вхідної інформації та включають структури замовлень, що передаються клієнтському інтерфейсу. Вони містять відомості про склад заявки, її статус, строки очікуваного постачання та супровідні логістичні параметри. Надання актуальних вихідних даних забезпечує узгоджену роботу між усіма підсистемами та підтримувати прозорість логістичних операцій.

Отже, під час проведення аналізу вихідних даних системи було виявлено, що розроблена система дозволяє поєднати функції обліку, прогнозування та інтеграції, що забезпечує її ефективне використання у практиці управління постачанням на підприємствах торгівлі або виробництва.

2.2 Розробка графічного інтерфейсу

Інтерфейс користувача є ключовим компонентом розробленої системи автоматизованого формування заявок на постачання товарів, оскільки він забезпечує зручну та ефективну взаємодію користувачів із функціоналом програми. З огляду на це, для реалізації інтерфейсу було обрано серверний шаблонізатор Thymeleaf, який інтегрується з фреймворком Spring Boot та підтримує архітектуру MVC. Шаблонізатор дозволяє чітко розділяти логіку представлення, бізнес-логіку та роботу з даними.

На першому етапі розробки було визначено головний екран інтерфейсу, який включає форму створення нової заявки на постачання товарів та таблицю для перегляду вже наявних замовлень. На рисунку 2.1 наведено зовнішній вигляд

головної сторінки, де відображається список замовлень у вигляді таблиці та доступна форма для введення параметрів замовлення дентифікатора постачальника та дати заявки.

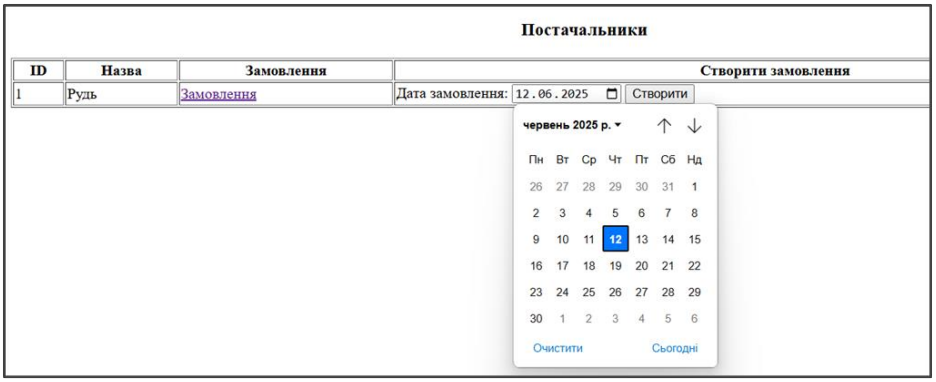


Рисунок 2.1 – Інтерфейс головної сторінки

Взаємодія між компонентами інтерфейсу реалізована через модель MVC:

1. View створюється за допомогою Thymeleaf, який динамічно формує HTML-сторінки з урахуванням даних, отриманих від контролера,

2. Controller приймає HTTP-запити, обробляє введені користувачем дані та викликає відповідні сервіси бізнес-логіки.

3. Model відповідає за роботу з базою даних та зберігання інформації про товари, заявки та постачальників.

При створенні нової заявки користувач вводить необхідні дані у форму та натискає кнопку «Створити», що ініціює HTTP POST-запит до контролера. Серверна частина перевіряє коректність отриманих параметрів, формує SQL-запит та додає запис у базу даних. Після успішного додавання даних інтерфейс автоматично оновлюється, і нове замовлення відображається у таблиці замовлень.

Загалом інтерфейс користувача розроблявся з урахуванням принципів зручності, прозорості та швидкості взаємодії. Використання Thymeleaf як шаблонізатора забезпечило гнучкість у відображенні даних і простоту інтеграції з серверною частиною, що відповідає сучасним стандартам розробки веб-додатків.

2.3 Розробка архітектури програмного продукту

Архітектура програмного продукту є ключовою складовою, яка визначає організацію системи, взаємодію її компонентів, а також забезпечує можливості масштабування та подальшого супроводу. У межах цього дослідження була розроблена архітектура системи управління замовленнями та товарами на основі моделі MVC, що реалізує клієнт-серверну взаємодію та розподіляє функціональність між серверною (Backend) та клієнтською (Frontend) частинами.

Основні принципи архітектури включають:

1. Модель MVC – серверна частина поділена на три рівні: model відповідає за зберігання та обробку даних, view формує представлення даних, controller приймає та обробляє запити користувача, керує взаємодією між model та view.

Клієнтська частина відповідає за відображення інформації та ініціацію запитів до серверу.

2. Модульність і сервісно-орієнтований підхід – обробка запитів реалізована через окремі контролери, сервіси та репозиторії, що забезпечує розподіл обов’язків та полегшує підтримку коду.

3. Використання стандартних протоколів – взаємодія між клієнтом і сервером відбувається через HTTP-протокол, що гарантує надійний обмін запитами та відповідями.

4. Зберігання даних – застосовується реляційна база даних для централізованого зберігання структурованої інформації про товари, постачальників, продажі та замовлення.

Серверна частина реалізована на основі фреймворку Spring Boot з використанням мови програмування Java 17. Сервер обробляє запити через контролери (orderController), сервісну логіку (service layer) та доступ до даних через репозиторії (Repository layer). Всі запити обробляються з архітектурною моделлю MVC, що забезпечує взаємодію з клієнтською частиною.

База даних використовує реляційну базу даних для зберігання основних сутностей системи, таких як: товари (products), категорії товарів (categories),

постачальники (suppliers), продажі (sold_products), замовлення (orders); замовлені товари (ordered_products).

Таблиці бази даних пов'язані між собою через зовнішні ключі.

У моделі MVS клієнтська частина (View) забезпечує взаємодію користувача з системою, надсилаючи HTTP-запити до RESTful API, реалізованого на стороні сервера. Отримані дані відображаються у зручному для користувача вигляді. View відповідає за ініціацію дій, таких як реєстрація товарів, створення замовлень та перегляд інформації про продажі.

Обробка запиту виконується поетапно:

1. View (клієнт) надсилає HTTP-запит до відповідного REST-ендпоінту.

2. Контролер приймає запит, проводить первинну валідацію вхідних даних та передає їх на обробку до Service-рівня.

3. Сервісна логіка (Service) реалізує бізнес-правила, взаємодіє з Model-рівнем репозиторіями та базою даних, формує відповідь.

4. Сформована відповідь повертається клієнту, де View-частина відображає її у відповідному інтерфейсі.

Архітектура системи має низку переваг, серед яких висока модульність, що забезпечується застосуванням фреймворку Spring Boot. Це дає змогу чітко розділяти логіку на окремі шари: контролери, сервіси та репозиторії, що значно спрощує підтримку коду та подальше розширення функціональності системи. Взаємодія між клієнтською та серверною частинами організована через стандартизовані HTTP-запити в рамках MVC-моделі, що спрощує інтеграцію з іншими системами та дозволяє без значних змін масштабувати систему.

Організація коду за принципами MVC також полегшує тестування окремих компонентів, що скорочує час на перевірку функціональності і сприяє швидшому виявленню помилок. Масштабованість системи забезпечується використанням загальноприйнятих технологій та підходів, що дає змогу адаптувати її до зростаючих обсягів даних і навантажень.

Водночас, існують потенційні обмеження, зокрема необхідність додаткового налаштування безпеки програмного інтерфейсу, наприклад, впровадження

механізмів авторизації та аутентифікації, таких як JWT або OAuth2, для захисту даних користувачів. Також у випадку значного збільшення обсягів даних або кількості користувачів може виникнути потреба в оптимізації запитів до бази даних або переході на більш потужні системи управління базами даних.

Отже, розроблена архітектура є ефективним рішенням, що забезпечує масштабованість, гнучкість і простоту супроводу, завдяки застосуванню моделі MVC, розподілу логіки між компонентами та використанню реляційної бази даних.

2.4 Розробка моделі системи

У процесі розробки програмного забезпечення для системи прикладного програмного інтерфейсу оптимізації управління постачання товарів було використано UML-діаграми для моделювання структури, взаємодії та поведінки компонентів системи. Основні діаграми описують взаємодію між різними ролями користувачів та системними модулями, а також алгоритмів, які забезпечують обробку запитів, взаємодію з базами даних і виконання основних операцій програмного інтерфейсу.

Діаграма діяльності використовується для моделювання бізнес-процесів та відображення послідовності дій, що виконуються в системі для досягнення певної мети [17]. Вона є корисним інструментом для візуалізації потоків контролю та даних між різними етапами виконання процесів.

Розробку діаграм діяльності, що включають отримання всіх замовлень для конкретного постачальника, створення замовлення та отримання конкретного замовлення, було виконано за допомогою програмного забезпечення Draw.io [18]. Крім того, застосування UML-діаграм сприяло покращенню комунікації між розробниками та іншими зацікавленими сторонами проекту, що забезпечило єдине розуміння структури та функціональності системи. Це дозволило вчасно виявляти потенційні проблеми на етапі проєктування та своєчасно вносити необхідні корективи, що підвищило якість кінцевого продукту та ефективність процесу розробки загалом.

1) Діаграма послідовностей отримання всіх замовлень для конкретного постачальника (див. рисунок 2.2). Процес розпочинається з того, що Клієнт ініціює запит до API за допомогою методу getOrdersBySupplier, вказуючи ідентифікатор постачальника. Цей запит надсилається серверу, який викликає відповідну процедуру для обробки запиту.

Далі API запускає процедуру GetOrderedProductsBySupplierId, яка здійснює запит до бази даних для отримання списку замовлених товарів для конкретного постачальника. Процедура в базі даних повертає список об'єктів типу OrderedProduct.

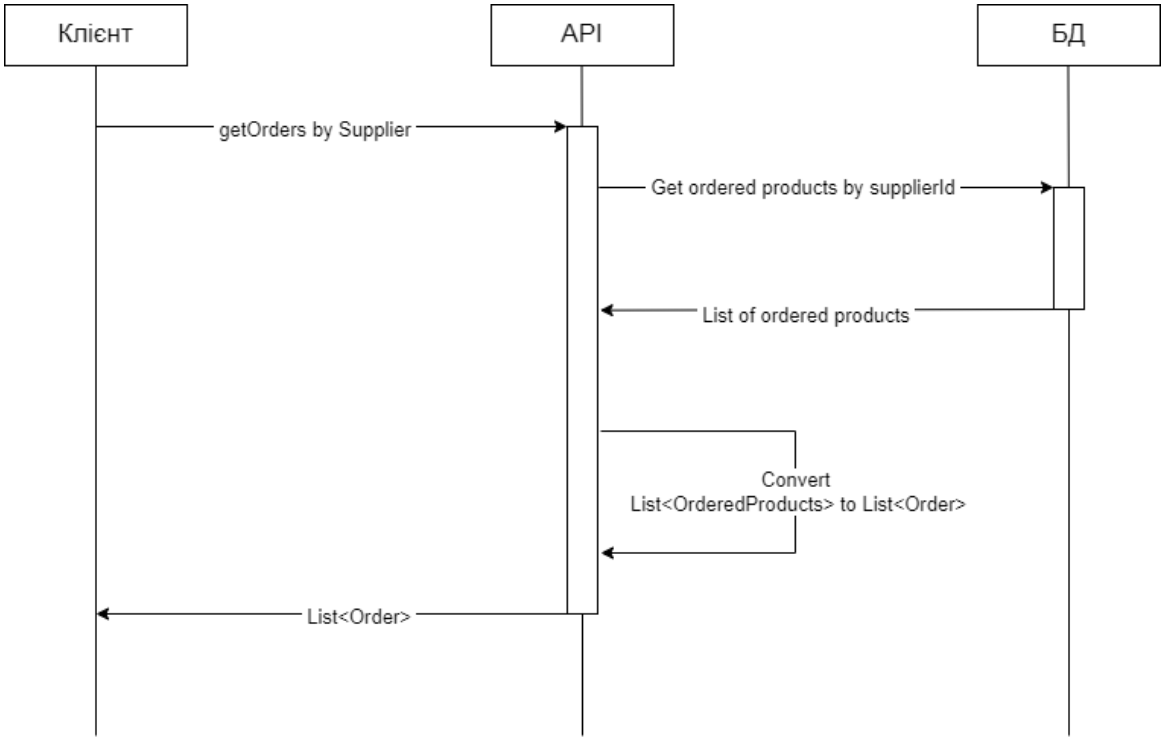


Рисунок 2.2 – Діаграма послідовностей отримання всіх замовлень для конкретного постачальника

Отримавши дані, програмний інтерфейсу виконує перетворення цього списку (List<OrderedProducts>) у формат List<Order>, щоб підготувати його до повернення клієнту. Після цього сформований список замовлень передається назад до Клієнта, який може використати отриману інформацію для відображення в інтерфейсі або подальших операцій.

2) Діаграма послідовностей для створення замовлення (див. рисунок 2.3) ілюструє процес взаємодії між клієнтом, програмним інтерфейсом, базою даних та обчисленням параметрів замовлення. Поетапно цей процес виглядає так:

Спочатку Клієнт ініціює запит на створення замовлення через програмний інтерфейс, викликаючи процедуру createOrder. Цей крок позначає початок процесу створення замовлення, який включає кілька етапів, що виконуються автоматично через серверну логіку.

Процес createOrder викликає API, яке здійснює запит до бази даних для отримання інформації про продукти, що продаються постачальником. Це реалізується за допомогою процедури «Get products sales by supplierId», яка отримує список доступних товарів разом з їхніми продажами.

Після отримання необхідних даних від бази даних, система створює замовлення, на основі чого розраховується коефіцієнт сезонності (SC) для кожного продукту. Цей розрахунок здійснюється через процедуру «Calculate seasonality coefficient» (SC), яка враховує фактори, що впливають на попит продукту в різні сезони.

Залежно від значення коефіцієнта сезонності, здійснюються додаткові розрахунки: якщо $SC > 0.5$ або $SC < -0.5$, то розраховується замовлена кількість для кожного продукту з урахуванням сезону, після чого продукти додаються до замовлення; якщо ж $-0.5 < SC < 0.5$, тоді система обчислює середній обсяг продажу для кожного продукту і відповідно коригує кількість замовлених одиниць. Після завершення розрахунків продукти додаються до замовлення згідно з отриманими значеннями, що забезпечує оптимальний розподіл товарів відповідно до попиту клієнта та доступності товарів у постачальника.

На наступному етапі здійснюється перевірка та коригування кількості замовлених товарів з урахуванням доступної ємності складу або інших ресурсних обмежень, щоб уникнути перевищення можливостей зберігання чи логістики. Вкінці сформоване замовлення зберігається в базі даних за допомогою процедури «Save created order», після чого інформація про створене замовлення

повертається клієнту для підтвердження успішного виконання операції та подальшого відображення у користувацькому інтерфейсі.

Отже, процес створення замовлення, реалізований через послідовність взаємодій між клієнтським інтерфейсом, серверною логікою та базою даних із застосуванням обчислення коефіцієнта сезонності, забезпечує коректне формування замовлень з урахуванням актуальних даних про продажі та ресурсних обмежень. Така реалізація сприяє підвищенню точності прогнозування та оптимізації обсягів замовлених товарів.

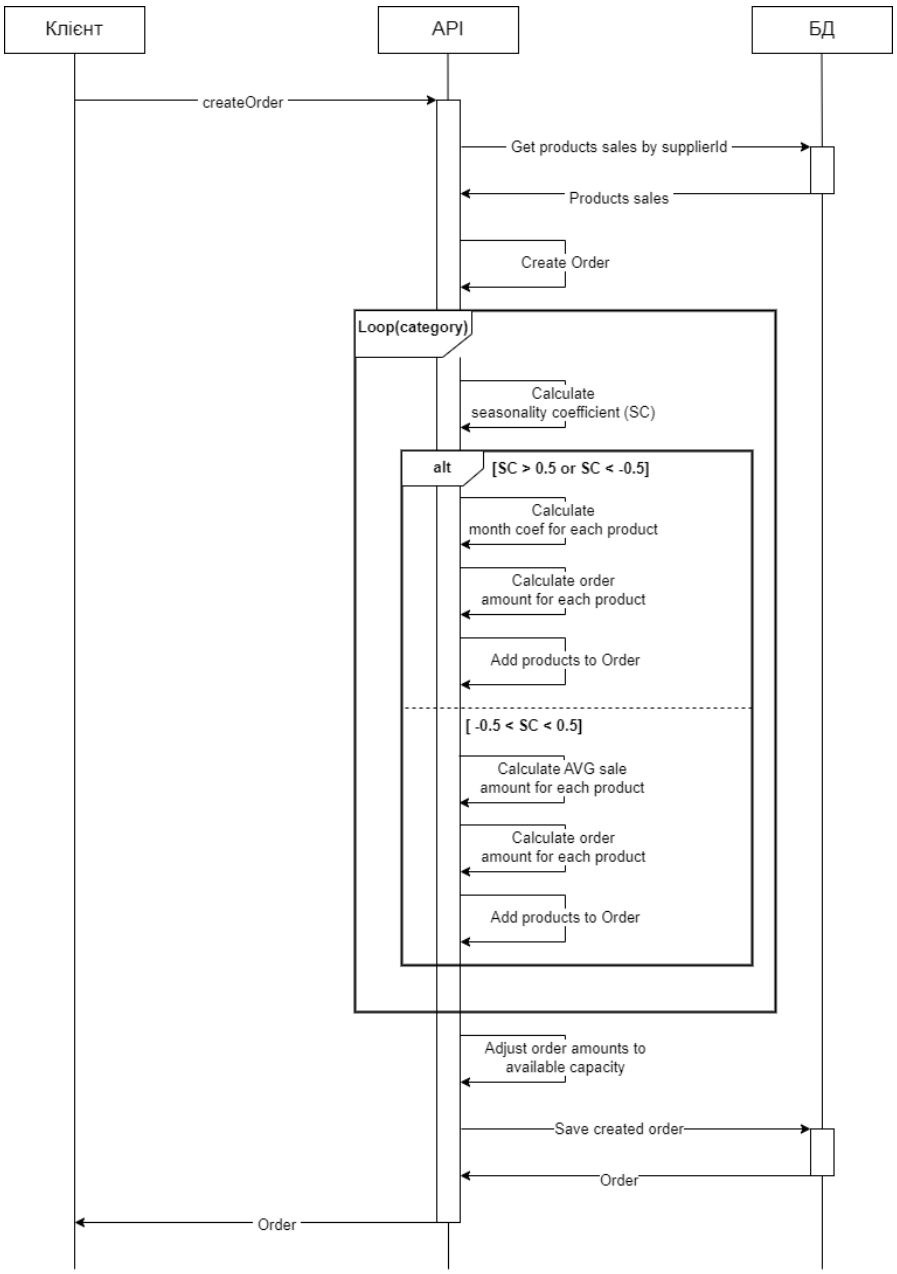


Рисунок 2.3 – Діаграма послідовностей створення замовлення

3) Діаграма послідовностей для отримання конкретного замовлення (див. рис. 2.4) демонструє процес отримання інформації про замовлення через API. Спочатку клієнт ініціює запит на отримання конкретного замовлення, викликаючи процедуру «getOrdered» через програмний інтерфейс, передаючи ідентифікатор замовлення, що дозволяє ідентифікувати необхідне замовлення.

Система приймає запит та запускає відповідну процедуру для отримання даних з бази даних, викликаючи функцію «Get ordered products by orderId». Ця процедура відповідає за отримання всіх товарів, які входять до складу конкретного замовлення, за вказаним ідентифікатором замовлення.

База даних обробляє запит і повертає список замовлених продуктів, який містить деталі кожного товару, зокрема їх назви, кількість, ціну та інші необхідні атрибути замовлення. Після отримання цього списку система конвертує його у формат об'єкта Order, щоб об'єднати всю інформацію в одну структуру, зручну для подальшої обробки.

Остаточно, клієнт отримує об'єкт Order, що містить всі деталі конкретного замовлення, де він може переглядати замовлення та здійснювати подальші операції на основі отриманої інформації.

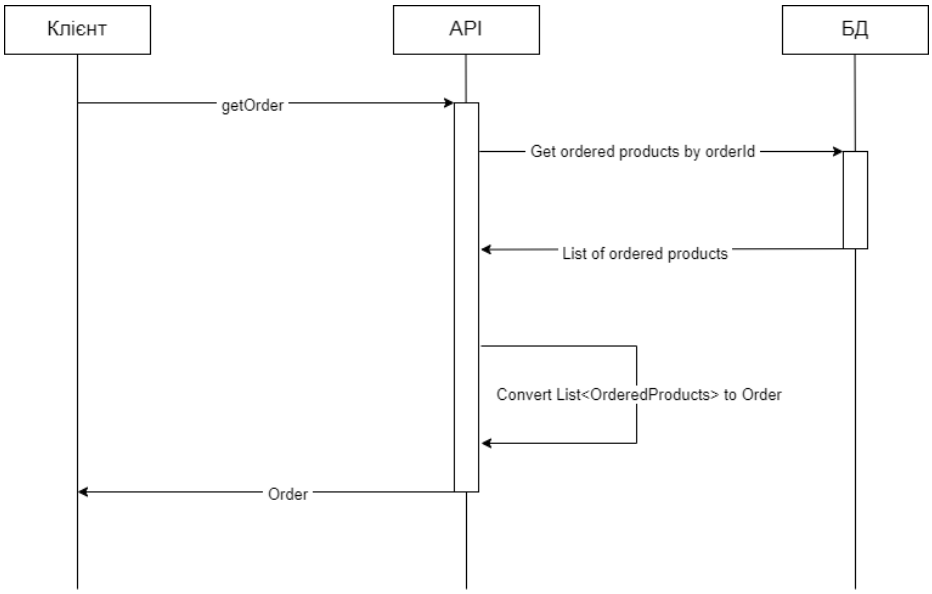


Рисунок 2.4 –Діаграма послідовностей для отримання конкретного замовлення

Блок-схему алгоритму обчислення коефіцієнта Кендалла було розроблено для наочного відображення процесу визначення рангової кореляції між місяцем продажу та обсягом реалізації товарів (див. рисунок 2.5). Вона дозволяє простежити усі етапи побудови таблиці кореляції, сортування даних, присвоєння рангів та розрахунку кількості прямих і обернених пар, що у підсумку дає змогу обчислити значення коефіцієнта Кендалла.

Алгоритм Кендалла для обчислення коефіцієнта кореляції починається з функції calculateCoefficient(productSale), яка отримує дані про продажі продуктів. Спочатку створюється таблиця кореляцій, в яку додаються елементи, що містять місяць і суму продажу. Для кожного запису перевіряється, чи місяць та рік не співпадають з поточними, після чого елементи додаються до таблиці.

Наступним кроком є сортування елементів таблиці за значеннями Y (сума продажу), після чого кожному елементу присвоюється ранг. Потім алгоритм встановлює ранги для осі X (місяць продажу), сортує таблицю за цими рангами та переходить до підрахунку кількості більших та менших елементів, порівнюючи їхні ранги по осі Y.

За допомогою підрахованих значень алгоритм обчислює коефіцієнт Кендалла, використовуючи формулу для коефіцієнта кореляції, що залежить від кількості пар елементів з більшими або меншими рангами. Результат обчислення коефіцієнта кореляції виводиться наприкінці процесу.

На рисунку 2.6 продемонстровано блок-схему розрахунку коефіцієнта сезонності була створена для систематизації етапів аналізу сезонних коливань обсягів продажу за різними місяцями. Вона допомагає візуалізувати послідовність дій: від сортування даних до обчислення середнього коефіцієнта для кожного місяця на основі обсягів продажу поточного, попереднього та наступного періодів. Опис блок-схеми виглядає наступним чином:

Процес починається з виклику функції calculateSeasonCoefficient(sales), яка приймає список продажів. Створюється об’єкт seasonCoefs для збереження коефіцієнтів сезонності. На початковому етапі дані про продажі сортуються за роком і місяцем.

Далі запускається цикл для кожного продажу. Для кожного елемента перевіряється, чи місяць та рік відрізняються від поточного. Якщо умова виконується, визначається місяць продажу та обсяг продажу. Також ініціалізуються змінні prevAmount, nextAmount і divider.

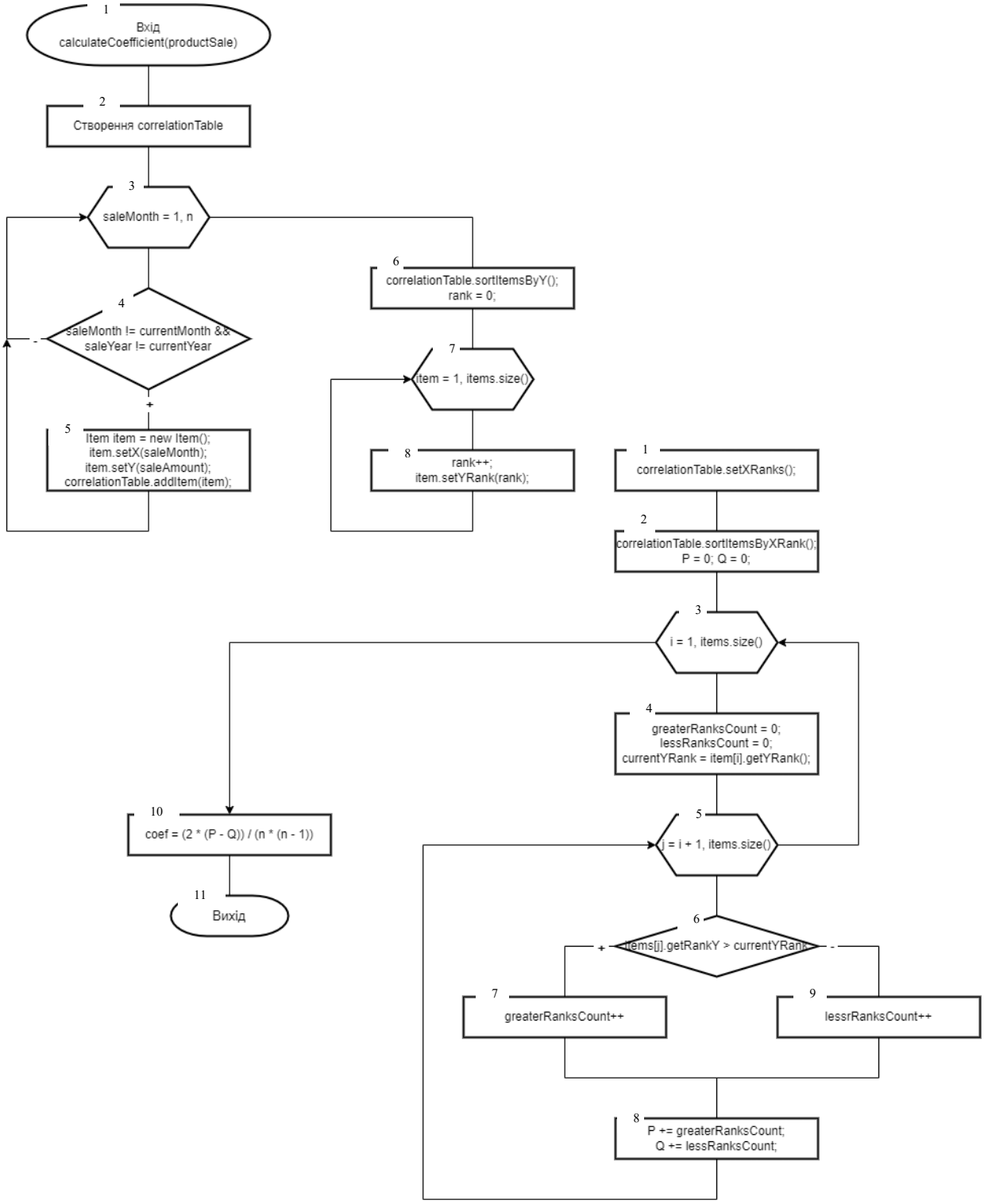


Рисунок 2.4 – Блок-схема алгоритму Кендалла

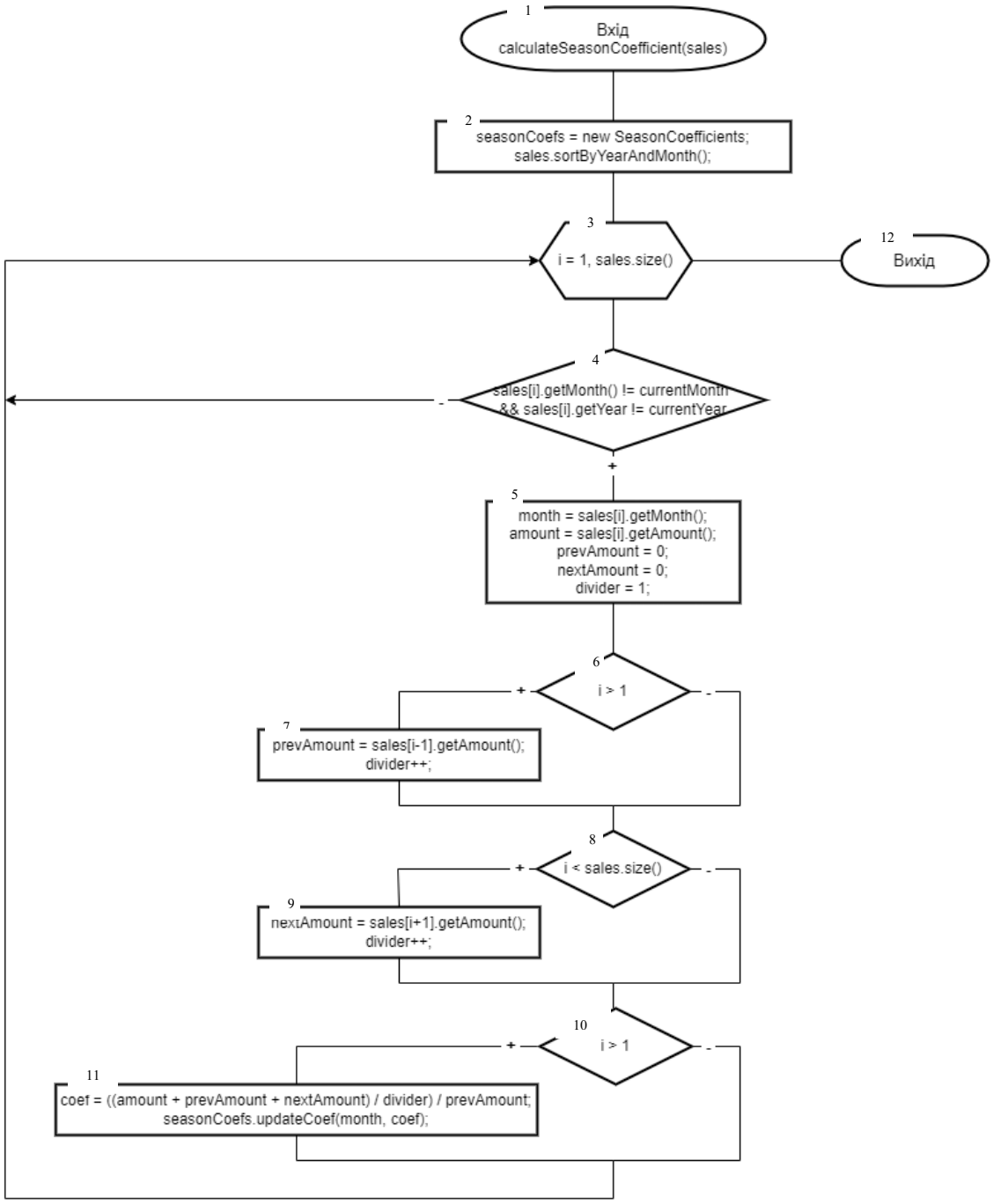


Рисунок 2.5 – Алгоритм розрахунку сезонного коефіцієнту

Якщо є попередній запис (i > 1), з нього береться обсяг продажу і збільшується дільник (divider++). За аналогічною логікою обробляється і наступний запис, якщо він існує. Після цього розраховується коефіцієнт сезонності за формулою: середнє значення обсягів продажу поділяється на попередній обсяг продажу. Отримане значення записується в об'єкт seasonCoefs для відповідного місяця. Цикл повторюється для кожного запису у списку

продажів. Після завершення обробки всіх записів функція завершує роботу, а сформовані коефіцієнти сезонності доступні для подальшого використання.

Отже, було створено моделі діаграми послідовностей і блок-схем для візуалізації основних процесів системи. Діаграми послідовностей дозволили детально змоделювати логіку взаємодії між клієнтською частиною, програмним інтерфейсом та базою даних під час виконання основних операцій: отримання всіх замовлень для постачальника, створення нового замовлення та отримання конкретного замовлення, блок-схеми дозволили чітко визначити послідовність дій, умови переходу та обчислення необхідних показників.

2.5 Висновки

У другому розділі було здійснено проєктування основних компонентів програмного інтерфейсу для формування заявок на постачання товарів відповідно до архітектури моделі MVC. Зокрема, побудовано UML-діаграми послідовностей, які детально відображають логіку взаємодії між клієнтською частиною (View), серверною логікою (Controller) та шаром доступу до даних (Model) під час виконання ключових операцій, таких як отримання замовлень постачальника, створення нового замовлення та перегляд інформації про конкретне замовлення. Для формалізації алгоритмічної структури процесів використано блок-схеми, що визначають послідовність дій, логічні переходи та умови виконання окремих етапів.

Результати аналізу підтверджують, що створена система інтегрує функціональні можливості прогнозування та управління, що формує надійну основу для ефективного використання в логістичних процесах підприємств оптової та виробничої діяльності.

3. РОЗРОБКА ПРОГРАМНОГО ЗАСТОСУНКУ

3.1 Варіантний аналіз і обґрунтування вибору засобів для реалізації програмного забезпечення

Розробка програмного інтерфейсу для формування заявок на постачання товарів з урахуванням дистрибуційних факторів вимагає правильного вибору технологічних засобів, оскільки важливими критеріями є продуктивність, масштабованість, простота інтеграції та зручність для подальшої підтримки. Для реалізації проєкту було важливо обрати саме ті інструменти, які зможуть задовольнити всі функціональні потреби.

Для реалізації серверної частини проєкту було обрано мову програмування Java 17 в поєднанні з фреймворком Spring Boot [19]. Java – це типізована об'єктно-орієнтована мова програмування, яка використовується у створенні програмного забезпечення широкого спектру: від комп'ютерних ігор і мобільних додатків до банківських систем та проєктів, спрямованих на вирішення завдань бізнесу [20].

Java є високопродуктивною мовою програмування, що гарантує стабільність і надійність у процесі розробки корпоративних систем. Оскільки розробка програмного інтерфейсу передбачає обробку значних обсягів запитів та даних, важливим фактором є продуктивність системи. Java включає численні покращення, що сприяють оптимізації виконання програм і забезпечують високу швидкість обробки даних. Це дозволяє створювати масштабовані та ефективні рішення для складних бізнес-систем.

Однією з основних характеристик Java є її здатність до масштабування. У процесі розробки складних програмних рішень, які повинні забезпечувати високу доступність та обробляти великі обсяги запитів, Java дозволяє реалізувати високонавантажені системи, які можуть ефективно працювати в умовах зростаючої кількості користувачів та інтеграції з різними сервісами [21].

Також варто зазначити, що Java є однією з найбільш безпечних мов програмування завдяки наявності вбудованих механізмів для захисту даних і

управління доступом. Це є важливою вимогою для систем, які працюють з конфіденційною інформацією та потребують забезпечення високого рівня безпеки. Можливості Java з шифрування даних та аутентифікації користувачів дозволяють створити захищене середовище для програмного інтерфейсу, що обробляє персональні дані користувачів.

Крім того, Java має розвинену екосистему, що включає велику кількість бібліотек та фреймворків, що значно спрощує процес розробки. Вибір Java забезпечує інтеграцію з новітніми стандартами, що дозволяє розробникам використовувати найсучасніші технології для створення RESTful API. Це дає змогу досягти високої ефективності при обробці запитів через HTTP та інтеграції з іншими компонентами інформаційної системи.

Таким чином, Java є оптимальним вибором для розробки програмного інтерфейсу завдяки своїй продуктивності, масштабованості, безпеці та підтримці сучасних технологій.

Для розробки серверної частини програмного забезпечення було обрано Spring Boot – фреймворк для спрощення і прискорення розробки веб- і мікросервісних додатків на базі Spring Framework [22]. Spring Boot був створений з метою спростити розробку, налаштування та запуск веб-застосунків, зокрема RESTful API, що ідеально підходить для завдань, пов’язаних із логістичними процесами, такими як формування та обробка заявок на постачання.

Порівняльна характеристика у таблиці 3.1 демонструє конкурентні переваги Spring Boot серед інших засобів реалізації.

Таблиця 3.1 – Порівняння фреймворків для веброзробки на Java

Фреймворк	**Рівень складності**	**Швидкодія**	**Спільнота**	**Розширюваність**	**Документація**
Spring Boot	Середній	Висока	Велика	Висока	Відмінна
Jakarta EE	Високий	Висока	Велика	Висока	Добра
Micronaut	Середній	Висока	Середня	Висока	Добра
Quarkus	Середній	Висока	Середня	Висока	Добра
Javalin	Низький	Висока	Мала	Середня	Середня

Продовження таблиці 3.1

Фреймворк	**Рівень складності**	**Швидкодія**	**Спільнота**	**Розширюваність**	**Документація**
Vert.x	Високий	Дуже висока	Середня	Висока	Добра
Spark Java	Низький	Висока	Мала	Середня	Середня

Spring Boot забезпечує автоматичне налаштування та має вбудований вебсервер. Завдяки цьому розробник може зосередитися безпосередньо на бізнес-логіці додатку, а не витрачати час на налаштування середовища розробки.

Однією з головних функціональних переваг Spring Boot є повна підтримка розробки а архітектурною моделлю MVC. У розробці програмного інтерфейсу реалізується обмін даними через HTTP-запити (GET, POST, PUT, DELETE), що забезпечується за допомогою анотацій таких як @RestController, @RequestMapping, @GetMapping та @PostMapping [23].

Spring Boot має вбудовану підтримку роботи з реляційними базами даних через Spring Data JPA. Це дає можливість легко реалізувати доступ до даних, без потреби написання SQL-запитів вручну. Також забезпечується підтримка транзакцій та зручне з’єднання з системою управління базами даних MySQL.

Отже, Spring Boot є ідеальним вибором для створення сучасного, масштабованого та продуктивного вебсервісу. Його застосування у розробці API для системи формування заявок дозволяє швидко розпочати розробку, зручно управляти архітектурою проєкту, ефективно працювати з базою даних та забезпечити безпеку обміну даними.

Для зберігання даних, пов’язаних із формуванням та обробкою заявок на постачання, а також супровідної інформації, такої як дані про клієнтів, транспортні засоби, маршрути, замовлення, статуси та історію змін, було обрано реляційну базу даних MySQL.

MySQL – це система управління реляційними базами даних, яка використовується для зберігання, організації та управління даними [24]. Особливо є важливою у системах, де одночасно обробляється велика кількість запитів на створення, оновлення, видалення та перегляд заявок.

MySQL підтримує схему даних, що дозволяє організувати зберігання об’єктів у вигляді таблиць із визначеними зв’язками, первинними та зовнішніми ключами. База даних забезпечує цілісність даних, що є принципово важливим для логістичних і облікових процесів, де помилки в даних можуть мати значні наслідки.

Важливим аргументом на користь вибору MySQL є її інтеграція з фреймворком Spring Boot. Завдяки використанню модуля Spring Data JPA, розробник отримує можливість працювати з базою даних за допомогою об'єктно-орієнтованих підходів, значно спрощуючи реалізацію CRUD-операцій (Create, Read, Update, Delete). Іншими словами, за допомогою Spring Data JPA більшість типових запитів можна реалізувати без написання складного SQL-коду, що скорочує час розробки і підвищує читабельність коду [25].

Крім того MySQL є безкоштовною, кросплатформною системою з відкритим вихідним кодом, що робить її зручною для використання в академічних і комерційних проектах без потреби в ліцензуванні.

Таким чином, вибір MySQL як основної системи керування базами даних у програмному продукті є обґрунтованим рішенням з огляду на її продуктивність, надійність, сумісність із інструментами фреймворку Spring та підтримку структурованих зв’язків між сутностями, які активно використовуються у процесі формування та обробки заявок.

Для формування та обробки заявок у системі застосовано архітектурну модель MVC, що розподіляє функціональність між трьома основними компонентами: Модель, Вид і Контролер [26]. Такий підхід забезпечує чітке розділення відповідальностей між логікою бізнес-процесів, інтерфейсом користувача та механізмами обробки запитів.

Модель відповідає за зберігання та обробку даних, включно з логікою взаємодії з базою даних MySQL через Spring Data JPA. Контролер приймає HTTP-запити від клієнтської частини (Вид), обробляє їх відповідно до бізнес-логіки, викликаючи необхідні сервіси Моделі, та формує відповіді для відображення.

Вид відповідає за інтерфейс користувача і взаємодіє з Контролером, забезпечуючи відображення актуальної інформації та прийом вводу користувача. Така структура дозволяє підвищити гнучкість розробки, спрощує тестування і налагодження, а також забезпечує масштабованість системи.

Взаємодія компонентів у межах MVC здійснюється через стандартизовані інтерфейси, що підтримуються фреймворком Spring Boot. Це дозволяє автоматизувати передачу даних між шарами без необхідності ручного оброблення форматів, зменшуючи обсяг коду і підвищуючи надійність системи.

Запропонована модель підтримує складні структури даних, що необхідні для формування багатокомпонентних заявок (наприклад, замовлення з декількома товарами або маршрути з кількома зупинками), і забезпечує їх ефективну обробку на серверній стороні.

Таким чином, застосування архітектури MVC гарантує високу продуктивність і зручність в обслуговуванні, сумісність із REST-підходом, гнучкість у реалізації бізнес-логіки та інтеграцію з іншими системами.

Для реалізації серверної частини було обрано Java 17 у поєднанні з фреймворком Spring Boot, що забезпечує стабільність, швидкість розробки та масштабованість. Як систему зберігання даних використано реляційну базу MySQL, яка ефективно інтегрується зі Spring Data JPA та дозволяє працювати з великими обсягами структурованої інформації.

3.2 Розробка алгоритмів роботи системи

У межах бакалаврської кваліфікаційної роботи було розроблено комплекс алгоритмічних рішень, спрямованих на забезпечення ефективної обробки вхідних даних, підвищення точності розрахунків та автоматизоване формування замовлень на постачання товарів. Ці алгоритми ґрунтуються на принципах об’єктивного аналізу, що враховує як внутрішні параметри логістичної системи, так і зовнішні дистрибуційні фактори.

Основну увагу в розробці було зосереджено на автоматизації процесу класифікації товарів за критеріями значущості, попиту та сезонності. Для цього

впроваджено механізм ранжування з використанням алгоритму рангової кореляції Кендалла, що дозволяє об’єктивно визначити пріоритетність товарів для закупівлі.

На рисунку 3.1 представлено блок-схему, що ілюструє послідовність обробки вхідних даних – від введення параметрів постачальника та початкових відомостей про замовлення до розрахунку коефіцієнтів сезонності, визначення кількості товарів для закупівлі, коригування обсягів з урахуванням обмежень складських потужностей та збереження сформованого замовлення до бази даних.

Основний алгоритм функціонування системи включає наступні послідовні етапи:

1. Введення даних постачальника та дати замовлення. Користувач надає ідентифікатор постачальника (supplierId) та дату створення нового замовлення, які слугують відправною точкою процесу формування замовлення.

2. Групування товарів за категоріями. Система класифікує товари за категоріями для подальшої аналітичної обробки в межах кожної окремої групи.

3. Розрахунок коефіцієнта сезонності. Для кожної категорії товарів виконується розрахунок сезонного коефіцієнта продажів із використанням відповідної аналітичної функції calculateCoefficient(List<ProductSaleData>).

4. Обробка категорій із високою сезонністю. Якщо розрахований коефіцієнт перевищує порогове значення (0,5), здійснюється поквартальний або помісячний аналіз продажів. Обсяг замовлення визначається за формулою (amount = previousMonthSale * seasonCoefficients.get(currentMonth)), після чого товари додаються до відповідного замовлення.

5. Обробка категорій із низькою сезонністю. Якщо коефіцієнт сезонності дорівнює або не перевищує 0,5, кількість продукції визначається як середнє арифметичне значення обсягу продажів: amount = totalSaleAmount / months.

6. Коригування замовлення згідно з ресурсами складу. Після формування попереднього замовлення система виконує перевірку відповідності загального обсягу продукції доступному вільному простору на складі та, за необхідності, коригує кількість продукції.

7. Збереження сформованого замовлення. Сформоване та відкориговане замовлення зберігається в реляційній базі даних MySQL з використанням Spring Data JPA.

8. Виведення результатів. На завершальному етапі користувачу надається підсумкове замовлення з усіма деталями, включно з кількісними показниками та категоріями товарів.

У процесі реалізації системи використано реляційну базу даних MySQL, яка складається з дев’яти основних таблиць. Ці таблиці призначені для структурованого зберігання інформації про категорії товарів, замовлення, продукти, постачальників, історію продажів та доставок. Логічна структура бази даних наведена на рисунку 3.2.

Таблиці спроєктовані таким чином, щоб забезпечити цілісність даних, оптимізовану побудову запитів та підтримку ефективних зв’язків між сутностями. Зокрема, система підтримує автоматичну синхронізацію з базою даних підприємства, що дозволяє регулярно оновлювати інформацію про фактично доставлену продукцію.

Окрема таблиця config використовується для зберігання конфігураційних параметрів системи, наприклад, порогових значень сезонності, граничних обсягів замовлень тощо. Вона не містить зовнішніх ключів, що забезпечує її автономність та спрощує внесення змін до параметрів без необхідності взаємодії з іншими таблицями.

Загалом структура бази даних була побудована з урахуванням вимог до масштабованості, узгодженості даних і забезпечення безперервної роботи алгоритмів обробки замовлень у межах усієї логістичної моделі системи.

Додатково, у процесі проектування бази даних було передбачено використання індексів для прискорення виконання типових запитів, а також реалізовано механізми контролю транзакцій для підтримки цілісності даних під час одночасного доступу кількох користувачів.

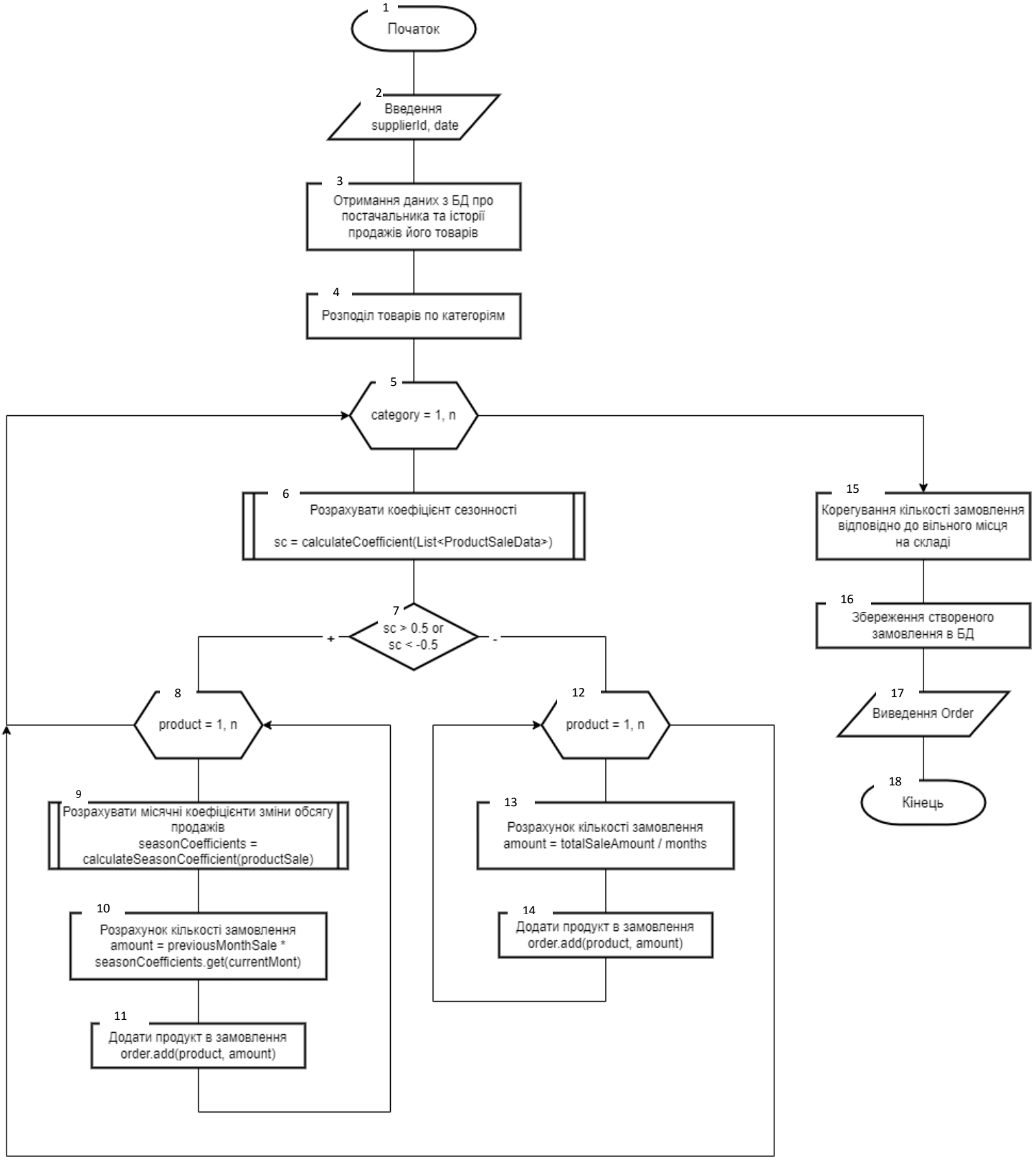


Рисунок 3.1 – Алгоритм роботи програми процесу створення замовлення

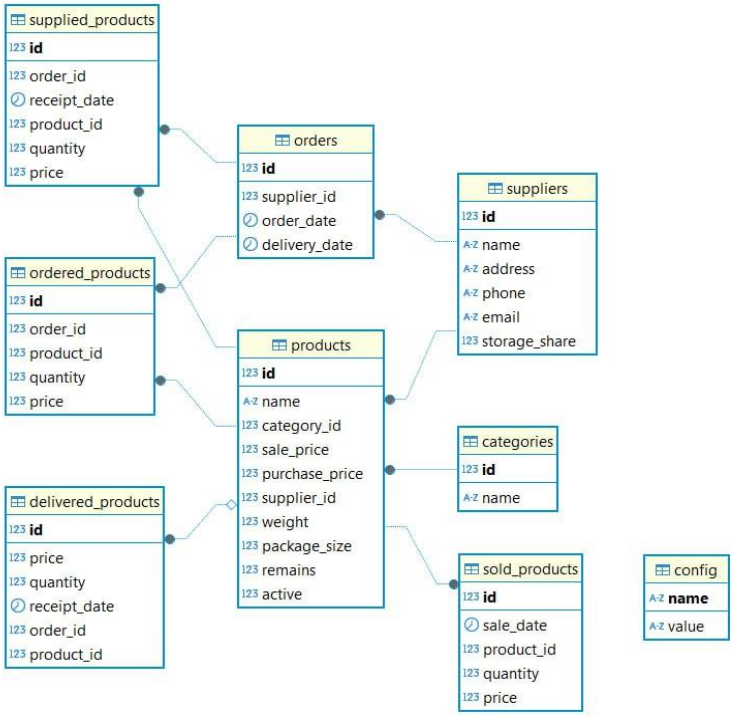


Рисунок 3.2 – Структурна схема бази даних

Таблиця 3.2 – Таблиці бази даних

№	Таблиці	Опис
1	categories	Зберігає дані про категорії товарів (id, name). Використовується для класифікації продуктів.
2	config	Зберігає параметри конфігурації системи (key, value). Використовується для налаштувань системи.
3	orders	Зберігає дані про замовлення (id, supplier_id, orderDate, deliveryDate). Зв'язано з таблицею suppliers через зовнішній ключ supplier_id.
4	ordered_products	Зберігає дані про замовлені продукти (id, order_id, product_id, quantity, price). Зв'язано з таблицями orders та products через зовнішні ключі.
5	products	Зберігає інформацію про продукти (id, name, category_id, salePrice, purchasePrice, supplier_id, weight, packageSize, remains, active). Зв'язано з таблицею categories та suppliers.
6	sold_products	Зберігає дані про продані продукти (id, saleDate, product_id, quantity, price). Зв'язано з таблицею products через зовнішній ключ product_id.
7	suppliers	Зберігає дані про постачальників (id, name, address, phone, email, storageShare).

Отже, спроєктовані структури даних та алгоритми забезпечують масштабованість та стійкість системи. Використання реляційної бази даних MySQL покращує процес зберігання та обробки даних, забезпечуючи взаємодію між основними сутностями, такими як категорії товарів, замовлення, продукти та постачальники. Чітка організація таблиць та визначення зв'язків між ними надають оптимальну продуктивність на кожному етапі обробки даних, від класифікації товарів до створення замовлень і інтеграції з інформацією про доставлені продукти з іншої бази даних підприємства.

3.3 Розробка модуля ранжування товарів

Модуль ранжування товарів розроблений для аналізу сезонних коливань попиту на товари шляхом розрахунку коефіцієнта рангової кореляції між очікуваними (плановими) та фактичними даними продажів. Його використання дозволяє виявити ступінь відповідності реальних обсягів постачання або продажу встановленим сезонним трендам.

За допомогою результатів, які обчислює модуль, система формування заявок на постачання товарів може:

– оптимізувати обсяги закупівель у різні місяці року;

– своєчасно коригувати плани постачання відповідно до фактичної поведінки попиту;

– підвищити точність прогнозування потреб клієнтів;

– зменшити ризики надлишкових запасів або нестачі товару на складах.

Для визначення наявності сезонності продажів створюється об'єкт класу CorrelationTable, у який додаються дані продажів товарів по місяцях. Поточний місяць виключається з аналізу, щоб уникнути спотворення результатів.

Після заповнення таблиці даними виконується обчислення коефіцієнта рангової кореляції між місяцем і кількістю продажів.

Отриманий коефіцієнт показує силу і напрямок залежності:

1. Значення |коефіцієнта| > 0.5 вказує на наявність вираженої сезонності.

2. Значення |коефіцієнта| ≤ 0.5 свідчить про відсутність суттєвої сезонності.

На основі розрахованого коефіцієнта обирається відповідний підхід до прогнозування замовлень:

Якщо виявлена сезонність (Math.abs(seasonalityCoefficient) > 0.5)), активується метод прогнозування, що враховує коефіцієнти сезонності.

Якщо сезонність не виявлена, застосовується прогнозування за середнім значенням продажів.

У разі наявності сезонності для кожного товару обчислюються коефіцієнти сезонності за допомогою методу calculateProductMonthCoefficients: для кожного місяця визначаються об'єми продажів поточного, попереднього та наступного місяців; обчислюється середній коефіцієнт зміни обсягів для кожного товару; результат зберігається у структурі SeasonCoefficients, яка асоційована з товаром.

Отже, модуль ранжування товарів був створений для автоматизації процесу аналізу та прогнозування попиту на товари на основі даних про їхні продажі. Основна мета цього модулю полягає в тому, щоб забезпечити точніше і ефективніше планування закупівель, враховуючи сезонні коливання попиту.

Завдяки аналізу даних продажів та використанню коефіцієнтів сезонності, модуль дозволяє компаніям автоматично визначати, які товари потребують більшої уваги і на яких з них слід зосередити зусилля в різні сезони року.

Цей модуль враховує сезонні коливання в продажах товарів і використовує дані з минулих періодів для прогнозування майбутніх потреб у товарах. Модуль також дозволяє вибирати відповідні стратегії прогнозування в залежності від того, чи є сезонність у продажах товару, чи ні. Якщо сезонність є сильно вираженою, то використовується спеціальний коефіцієнт сезонності, що дозволяє точніше передбачити потреби у товарах для наступного періоду.

3.4 Розробка модуля розрахунку кількості товарів для замовлення

Модуль розрахунку кількості товарів для замовлення призначений для визначення оптимального обсягу продукції, яку необхідно замовити з метою підтримання раціонального рівня запасів, з урахуванням динаміки продажів, сезонності та значущості кожного товару. Його функціонування базується на

принципах адаптивного прогнозування, що дозволяє враховувати зміни попиту в різні періоди року та мінімізувати ризики виникнення як дефіциту, так і надлишку продукції на складі.

Основною метою модуля є надання користувачу інструменту для точного й обґрунтованого розрахунку кількості товарів, які слід замовити в конкретний період. В основі роботи модуля лежить аналіз даних про обсяги продажів за попередні місяці року. Такий підхід дозволяє враховувати історичні тенденції споживчої активності, які часто мають циклічний характер (наприклад, підвищений попит у святкові періоди або спад у міжсезоння).

Інтеграція алгоритму рангової кореляції Кендалла у функціонал модуля дозволяє більш точно визначити місяці, які мають найбільший вплив на формування попиту на певні товари. Ранжування місяців за ступенем їхньої значущості дає змогу автоматично пріоритезувати закупівлі та підвищити точність прогнозу.

Крім того, така методика дозволяє швидко реагувати на аномальні зміни у попиті, що може бути викликане зовнішніми факторами, економічною ситуацією, кліматичними умовами, діями конкурентів тощо.

Використання даного модуля в автоматизованій системі управління постачаннями дозволяє досягти суттєвих переваг. Зокрема, покращується обґрунтованість управлінських рішень щодо закупівель, зменшуються витрати на зберігання товарів, мінімізуються втрати, пов’язані з нереалізованою продукцією або її відсутністю в критичні моменти.

Алгоритм роботи модуля включає кілька етапів:

1. Ініціалізація та додавання даних – для кожного товару додаються дані, що включають місяць року (від 1 до 12) і обсяг продажу в цьому місяці.

2. Ранжування даних – для кожного товару визначаються ранги по обсягу продажів та місяцях року.

3. За допомогою коефіцієнта рангової кореляції Кендалла оцінюється залежність між місяцями року та обсягами продажу товарів.

4. Розрахунок кількості товарів, використовуючи отриманий коефіцієнт кореляції, модуль допомагає прогнозувати кількість товарів для замовлення на наступний період.

Спочатку створюється CorrelationTable для кожного товару, в якій зберігаються дані про місяці року та відповідні обсяги продажу товарів.

Далі передбачається створення класу OrderCalculationModule, що містить методи для додавання даних та обчислення рангової кореляції Кендалла. На рисунку 3.продемонстровано алгоритм обчислення коефіцієнту кореляції в коді.

```
public float calculateCoefficient() {  4 usages   mcdoom
    // Set Y ranks
    int currentRank = 0;
    items.sort(( Item a,  Item b) -> Float.compare(a.getY(), b.getY()));
    for (Item item : items) {
        currentRank++;
        item.setYRank(currentRank);
    }

    // Set X ranks
    for (Item item : items) {
        item.setXRank(monthRanks.get(item.getX()));
    }
    // Sort by X rank
    items.sort(Comparator.comparingInt(Item::getXRank));

    // Calculate coefficient
    int n = items.size();
    int P = 0;
    int Q = 0;
    for (int i = 0; i < n - 1; i++) {
        int greaterRanksCount = 0;
        int lessRanksCount = 0;
        int currentYRank = items.get(i).getYRank();
        for (int j = i + 1; j < n; j++) {
            if (items.get(j).getYRank() > currentYRank) {
```

Рисунок 3.3 – Обчислення коефіцієнта рангової кореляції Кендалла

Отже, модуль розрахунку кількості товарів для замовлення був розроблений для оптимізації процесу планування закупівель на основі кореляції між обсягами продажів товарів та місяцями року. За допомогою алгоритму рангової кореляції Кендалла здійснюється аналіз залежностей між обсягами продажів та відповідними місяцями, що дозволяє визначити найбільш точну кількість товарів для замовлення. Використання цього методу дає змогу зменшити вплив сезонних

коливань на рівень запасів, підвищити точність прогнозування попиту та ефективність управління товарними запасами.

3.5 Висновки

Розробка прикладного програмного інтерфейсу для формування заявок на постачання товарів продемонструвала ефективність обраного технічного підходу завдяки поєднанню сучасних технологій та архітектурної строгості. Застосування мови програмування Java 17 у зв'язці з фреймворком Spring Boot дозволило створити стабільне, масштабоване та зручне в обслуговуванні серверне рішення. Інтерфейс користувача реалізовано з використанням шаблонізатора Thymeleaf, що забезпечило динамічне формування HTML-сторінок на основі даних, отриманих із сервера, та сприяло гнучкій інтеграції з існуючою веб-інфраструктурою підприємства.

Значну роль у структуризації проєкту відіграло використання архітектурної моделі MVC. Архітектура забезпечила розділення відповідальностей між компонентами системи. Це дозволило досягти зручності тестування та спрощеного супроводу системи.

У процесі реалізації було використано UML-діаграми послідовностей, які дозволили чітко змоделювати ключові процеси, зокрема отримання всіх замовлень, створення нового замовлення та перегляд конкретного замовлення. Додатково було розроблено блок-схеми реалізації алгоритмів розрахунку коефіцієнта Кендалла та коефіцієнта сезонності, що істотно полегшило реалізацію складної логіки та підвищило точність програмної реалізації.

У результаті було створено прикладний інтерфейс, що відповідає усім вимогам проєкту, забезпечує ефективну взаємодію з базою даних, підтримує облік сезонності продажів та коректне планування замовлень. Розроблений застосунок є надійною основою для подальшого розвитку системи, зокрема для розширення функціональності, підключення нових модулів або оптимізації процесів обробки даних.

4 ТЕСТУВАННЯ ПРОГРАМИ

4.1 Вибір методу тестування

Тестування програмного забезпечення є критично важливим етапом життєвого циклу розробки, який дозволяє виявити помилки, перевірити відповідність функціональності вимогам і забезпечити стабільність роботи системи в різних умовах експлуатації.

Існує кілька основних підходів до тестування програмних продуктів, кожен з яких має свої переваги та сфери застосування:

1. Метод «чорної скриньки» – передбачає тестування функціональності без знання внутрішньої реалізації системи. Тестувальник фокусується на вхідних і вихідних даних, перевіряючи, чи відповідає поведінка програми очікуваним результатам. Цей метод ідеально підходить для перевірки відповідності специфікації, тестування API-ендпоінтів, перевірки валідації введення та коректності повідомлень про помилки.

2. Метод «білої скриньки» – передбачає аналіз внутрішньої логіки програми, включаючи перевірку умов, циклів, логічних операторів та роботи з базами даних. Він забезпечує повне покриття коду, допомагає виявити логічні помилки, помилки у реалізації алгоритмів та вразливості.

3. Метод «сірої скриньки» – поєднує елементи попередніх підходів, коли тестувальник має часткове розуміння внутрішньої структури додатку. Цей метод часто застосовується у випадках, коли необхідно перевірити взаємодію модулів чи систему безпеки.

4. Інтеграційне тестування – зосереджене на перевірці взаємодії між модулями програми, правильності передачі даних між ними та стабільності API при інтеграції з іншими системами.

5. Модульне тестування – спрямоване на перевірку окремих функцій або компонентів програми в ізоляції. Дозволяє забезпечити точність логіки обробки даних на рівні найменших одиниць коду.

6. Навантажувальне тестування – визначає, як система поводиться під великим обсягом запитів чи даних. Для API це важливо для перевірки витривалості серверної частини.

Отже, обрано комбінований підхід, що поєднує методи «чорної скриньки» та «білої скриньки». Такий підхід дає змогу протестувати як зовнішню поведінку системи (інтерфейс взаємодії з API), так і внутрішню логіку обробки даних.

У процесі тестування системи, реалізованої за архітектурною моделлю MVC, використовувалися методи «чорної» та «білої» скриньки для перевірки як зовнішньої поведінки інтерфейсу, так і внутрішньої логіки окремих компонентів.

Метод «чорної скриньки» застосовувався до рівня Controller та View, де основна увага зосереджувалася на тестуванні взаємодії користувача з інтерфейсом, зокрема через шаблонізатор Thymeleaf, а також на правильності обробки запитів через REST API. Було протестовано подання нової заявки на постачання, отримання списку замовлень, обробку помилок при введенні некоректних даних (наприклад, порожніх полів або даних у неправильному форматі), а також відповідність API-ендпоінтів специфікації, описаній у документації OpenAPI. Такий підхід дозволив перевірити, чи система в цілому поводиться згідно з очікуваннями кінцевого користувача та технічними вимогами.

Метод «білої скриньки» застосовувався переважно на рівні Model, де реалізована бізнес-логіка та взаємодія з базою даних через Spring Data JPA. У межах цього тестування здійснювалась перевірка коректності SQL-запитів, що автоматично генеруються фреймворком, правильності обробки запитів на створення, зміну або видалення заявок, а також точності реалізації алгоритмів, зокрема обчислення коефіцієнта Кендалла для ранжування постачальників і розрахунку прогнозованих обсягів товарів. Це дозволило забезпечити надійність логіки, що лежить в основі прийняття рішень у системі. Крім того, проводиться ручний та автоматизований перегляд логіки обробки винятків і перевірка механізмів автентифікації та авторизації, які захищають систему від несанкціонованого доступу.

Отже, обраний комбінований підхід до тестування дозволив забезпечити як зовнішню стабільність і зручність користування прикладного програмного інтерфейсу, так і внутрішню надійність алгоритмів, безпеку даних та правильність бізнес-логіки системи.

4.2 Аналіз результатів тестування

Процес тестування програмного застосунку для формування заявок на постачання товарів здійснювався поетапно із застосуванням комбінованого підходу, який об’єднує методи «чорної» та «білої скриньки». Тестування охоплювало три основні етапи: підготовку, безпосереднє виконання тестів і подальший аналіз отриманих результатів.

На підготовчому етапі були визначені ключові функціональні сценарії, що відповідають вимогам до системи. До них належали: створення нової заявки на постачання, перевірка правильності введення вхідних даних, тестування REST API-ендпоінтів, які реалізовані в Controller-компоненті MVC-моделі, та перевірка коректної обробки запитів до бази даних на рівні Model.

У межах тестування за методом «чорної скриньки» акцент робився на перевірці функціональності системи з позиції кінцевого користувача. Було змодельовано різні варіанти створення заявок з різними комбінаціями вхідних параметрів, зокрема перевірка правильності обробки ідентифікаторів постачальників, форматів дати, а також обробка помилок, пов’язаних з порожніми або некоректними полями. Особливу увагу приділено відповідності фактичної поведінки системи очікуваній, згідно з технічною специфікацією.

Метод «білої скриньки» був застосований для тестування логіки бізнес-процесів на рівні сервера, що реалізовані в Service та Repository компонентах. Було проаналізовано й перевірено правильність виконання SQL-запитів, обробку запитів на створення, зміну та видалення заявок, а також точність роботи алгоритмів, що відповідають за обчислення кількості товару та ранжування постачальників.

Тестування проводилося як вручну за допомогою інтерфейсу, побудованого з використанням шаблонізатора Thymeleaf, так і автоматизовано з використанням відповідних інструментів для тестування API та логіки серверної частини. Такий підхід дозволив забезпечити всебічну перевірку працездатності та надійності програмного застосунку. Тестування проводилося як вручну, так і автоматизовано.

На початковому етапі тестування програмного застосунку була перевірена базова функціональність створення нової заявки на постачання товарів через головний екран інтерфейсу користувача (див. рисунок 4.1). Цей етап мав на меті перевірити коректність обробки вхідних параметрів сервером, зокрема ідентифікатора постачальника та дати заявки, а також забезпечення належної інтеграції між представленням, бізнес-логікою і базою даних відповідно до моделі MVC.

У тестовому сценарії на головному екрані інтерфейсу була активована форма створення замовлення з попередньо вказаним supplierID = 1, що відповідає конкретному постачальнику, а також вибрана дата замовлення в межах періоду 2025-06-12. Після введення цих даних натискаючи кнопку «Створити» ініціюється HTTP POST-запит до відповідного ендпоінту контролера. Метою цього етапу було перевірити, чи правильно серверна частина (Service та Repository-рівні) обробляє отримані дані, формує SQL-запит до бази даних та додає новий запис у таблицю замовлень.

Після успішного створення замовлення система автоматично оновлила візуальне подання інтерфейсу. (див. рисунок 4.2). У таблиці, де відображаються всі замовлення, з’являвся новий рядок із відповідними даними постачальника та дати. Додатково на головному екрані для кожного постачальника передбачено гіперпосилання, яке дозволяє переглянути всі замовлення, пов’язані з ним. У результаті тесту було перевірено, що щойно створене замовлення коректно зберігається в базі даних і відображається у списку замовлень постачальника після переходу за відповідним посиланням.

Таким чином, початкове тестування підтвердило, що система правильно обробляє введені користувачем дані, забезпечує збереження нової заявки в базі даних та її подальше відображення у відповідному розділі інтерфейсу. Це свідчить про правильну інтеграцію між компонентами Model, View та Controller, що є важливими для забезпечення надійної роботи всієї логістичної системи.

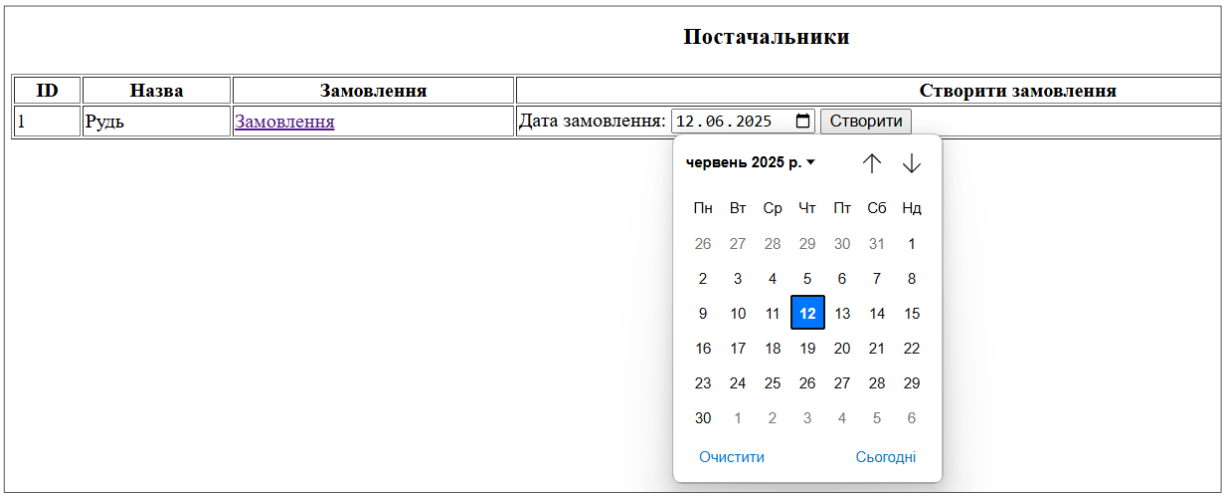


Рисунок 4.1 – Обробка запиту на створення заявки з вказаними параметрами

У результаті обробки запиту на отримання інформації про всі заявки конкретного постачальника система коректно сформувала HTML-сторінку з таблицею, яка відображає список усіх замовлень, прив’язаних до відповідного постачальника (див. рисунок 2). Кожен запис у таблиці містить унікальний ідентифікатор заявки (ID), дату створення замовлення та гіперпосилання «Переглянути», яке дозволяє здійснити перехід до детального перегляду вибраного замовлення.

Згідно з архітектурою MVC, при переході за посиланням з головної сторінки на перегляд замовлень контролер отримує запит, зчитує supplierId як параметр, звертається до відповідного сервісного класу (Service), який у свою чергу викликає метод у репозиторії для вибірки з бази даних усіх замовлень, що належать цьому постачальнику. Дані передаються назад у вигляді списку DTO після чого контролер формує відповідну модель для представлення (View) і передає її до шаблону HTML для виведення у вигляді таблиці.

Результат відображення заявок із відповідними датами свідчить про те, що запит був успішно оброблений, база даних правильно виконала вибірку, а інтерфейс належним чином візуалізував результат. Наявність активних посилань «Переглянути» поруч із кожним записом додатково підтверджує, що система підтримує навігацію до деталізованої інформації про окремі замовлення. Це забезпечує повноцінну взаємодію користувача з даними та відповідає як функціональним вимогам, так і принципам логічної моделі MVC.

Замовлення постачальника

ID	Дата	Переглянути
1	2025-04-15	Переглянути
2	2025-04-20	Переглянути
3	2025-04-25	Переглянути
4	2025-04-25	Переглянути
5	2025-05-22	Переглянути
6	2025-05-20	Переглянути
7	2025-05-26	Переглянути
8	2025-05-30	Переглянути
9	2025-05-31	Переглянути
10	2025-06-01	Переглянути
11	2025-06-03	Переглянути
12	2025-06-14	Переглянути

На головну

Рисунок 4.2 – Обробка запиту для отримання списку заявок, що належать постачальнику з конкретним ідентифікатором

Обробка запиту для отримання інформації про конкретну заявку в системі здійснюється через передачу ідентифікатора заявки (orderId) до відповідного контролера, який звертається до сервісного шару для отримання повної інформації про замовлення (див. рисунок 4.3). Сервіс витягує з бази даних перелік товарів у заявці, їхню кількість, параметри пакування (кількість одиниць у ящику), а також інформацію про наявність на складах. На основі цих даних розраховується кількість ящиків для кожного товару, виходячи з округлення в більший бік (цілочисельне ділення з округленням вгору).

Після цього відбувається інтеграція з модулем ранжування товарів, який здійснює аналіз сезонних коливань попиту шляхом розрахунку коефіцієнта рангової кореляції між обсягами продажів і місяцями, виключаючи поточний місяць. Якщо виявлено наявність сезонності (|коефіцієнт| > 0.5), система коригує

обсяги постачання згідно з обчисленими коефіцієнтами сезонності для відповідного товару.

У протилежному випадку прогноз ґрунтується на середньому значенні продажів. Скориговані дані використовуються для перевірки залишків товарів на складах, де визначається оптимальний розподіл між доступними локаціями.

Результатом обробки запиту є згенерована відповідь, що містить детальну інформацію про товари у заявці, кількість одиниць і ящиків, обрані склади для відвантаження.

Замовлення

Постачальник: Рудь

Дата: 2025-05-20

Назва продукту	Кількість, ящ.
Пустунчик абрикос в/ст	96
Пустунчик вишня в/ст	92

На головну

Рисунок 4.3 – Обробка запиту для отримання інформації про конкретну заявку в системі

Для тестування функціональності обробки запиту на отримання інформації про конкретну заявку постачальника використовувався інтерфейс з реалізацією на основі шаблонізатора, що дозволяло динамічно відображати отримані з API-ендпоінтів дані без потреби в ручному оновленні сторінки.

Після вибору заявки за її ідентифікатором (orderId) у таблиці, серверна частина генерує відповідний SQL-запит, який повертає детальну інформацію про замовлення: перелік товарів, їх кількість, характеристики тари (зокрема кількість ящиків, яка обчислюється на основі загальної кількості товару та місткості ящика).

Отримані дані далі інтегруються у HTML-шаблон, де автоматично відображаються на сторінці з деталями заявки. Завдяки використанню шаблонізатора забезпечено точне та структуроване відображення інформації, включаючи назви товарів, кількість у штуках і ящиках, а також загальну

структуру заявки. Метод «білої скриньки» дозволив перевірити правильність логіки формування відображення, відповідність обчислень у моделі, а також поведінку системи при тестуванні з різними типами заявок як повних, так і неповних.

У тестуванні коду для класу CorrelationTableTest, зокрема для методу testCalculateCoefficient_positiveSeasonDependentCategory,перевірено коректність обчислення коефіцієнта кореляції для сезонно залежної категорії даних.

Спочатку створюється об'єкт класу CorrelationTable, в який додаються пари даних, що складаються з місяців та відповідних значень. Потім викликається метод calculateCoefficient(), який обчислює коефіцієнт кореляції для цього набору даних. Тест перевіряє, чи отриманий коефіцієнт кореляції більше 0,5, що вказує на наявність сильної позитивної залежності між значеннями. Якщо умова виконується, тест вважається успішним.

TestCalculateCoefficient_negativeSeasonDependentCategory, для іншого набору даних, проводиться подібне тестування, де значення мають негативну кореляцію. Дані, додані до таблиці, показують зниження показників зі зміною місяця. Метод calculateCoefficient() має обчислити коефіцієнт кореляції, що буде менше -0,5, що вказує на сильну негативну залежність. Тест перевіряє, чи виконується це, і при успішному результаті підтверджує, що кореляція є від'ємною.

Для тесту testCalculateCoefficient_seasonIndependentCategory обрано набір даних, де значення змінюються незалежно від сезону або місяця. Додані значення не мають чіткої тенденції до збільшення чи зменшення. Коефіцієнт кореляції для такого набору має бути близьким до нуля, що вказує на відсутність кореляції між змінними. Тест перевіряє, чи значення коефіцієнта кореляції потрапляють у діапазон від -0,5 до 0,5, що підтверджує відсутність сильного взаємозв'язку.

На рисунку 4.5 продемонстровано результат виконання тестування роботи методу calculateCoefficient() в різних умовах і з різними наборами даних.

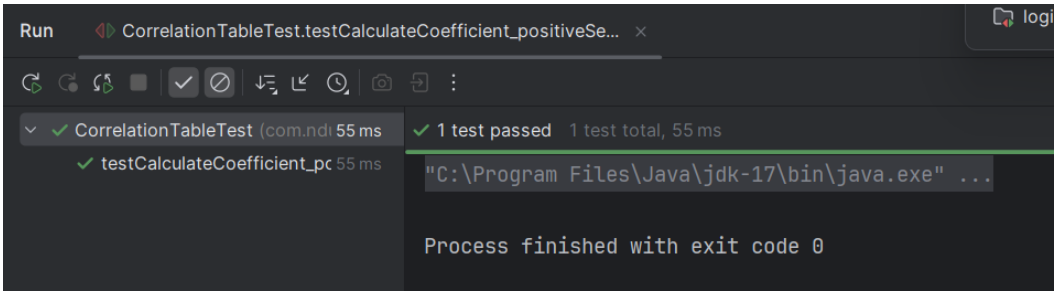


Рисунок 4.5 – Тестування методу calculateCoefficient()

Тест виконано успішно, що підтверджує правильність роботи методу calculateCoefficient() в різних умовах і з різними наборами даних. Результати тестування свідчать про те, що метод коректно обчислює коефіцієнт кореляції в позитивних, негативних та нейтральних сценаріях.

Тестування в класі OrderServiceImplTest охоплює основні методи сервісу OrderServiceImpl, що відповідає за роботу з замовленнями у додатку. Використано технологію Mockito для створення мок-об'єктів, що заміняють реальні залежності, такі як репозиторії та сервіси, щоб перевірити поведінку сервісу без взаємодії з реальною базою даних [29].

Перший тест, createOrder_validSupplierAndData, перевіряє створення замовлення для валідного постачальника з правильною датою. Для цього надаються мок-об'єкти для необхідних залежностей, таких як supplierRepository, soldProductsRepository, dataConverter та інші. Тест підтверджує, що результат виконання методу createOrder не є нульовим, а також перевіряється, що методи збереження замовлення і продуктів були викликані один раз. Це означає, що замовлення було успішно створено, і дані збережено в базі.

Наступний тест, createOrder_supplierNotFound, перевіряє ситуацію, коли постачальник не знайдений за вказаним ідентифікатором. В цьому випадку очікується, що метод викине виключення IllegalArgumentException, що підтверджує коректну обробку помилки у випадку відсутності постачальника.

Тест getOrder_orderExist перевіряє отримання замовлення за його ідентифікатором. Для цього надається мок-дані замовлення, і після виклику

методу getOrder перевіряється, що результат відповідає очікуваному значенню. Якщо замовлення існує, метод повертає правильний об'єкт замовлення.

У тесті getOrder_orderNotExist перевіряється ситуація, коли замовлення з вказаним ідентифікатором не існує. Очікується, що метод поверне null у такому випадку, що підтверджує коректну обробку відсутності замовлення.

Тести getOrders_ordersExist та getOrders_ordersNotExist перевіряють отримання списку замовлень для постачальника. Перший тест перевіряє, чи метод коректно повертає список замовлень, коли вони існують, а другий – коли замовлення для постачальника відсутні. У разі відсутності замовлень очікується, що метод поверне порожній список.

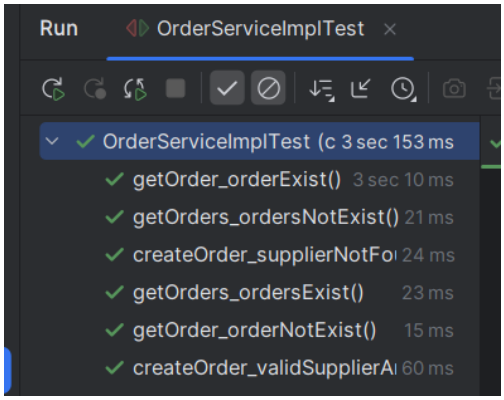


Рисунок 4.6 – Тестування сервісу замовлень за допомогою Mockito.

Усі тести були виконані успішно, і результати підтвердили коректну роботу класу OrderServiceImpl. Було забезпечено правильну взаємодію з репозиторіями, коректну обробку помилок, а також реалізацію бізнес-логіки для створення та отримання замовлень.

Отже, в результаті виконаних тестувань було здійснено комплексну перевірку ключових функціональних компонентів системи. Використовуючи різноманітні підходи, включаючи техніки «чорної» та «білої» скриньки, були протестовані основні методи сервісу замовлень, зокрема створення та отримання замовлень, а також правильність обробки даних із зовнішніх ресурсів, таких як репозиторії та сервіси.

Тестування з використанням бібліотеки Mockito підтвердило правильність роботи логіки створення замовлення за заданими параметрами. У ході тестування було виявлено, що всі методи виконуються відповідно до очікуваних результатів. Створення замовлень для валідних постачальників працює коректно, а система відмовляється обробляти запити з неіснуючими постачальниками, що відповідає бізнес-логіці. Перевірка отримання замовлень підтвердила правильну обробку випадків як наявності замовлень, так і їх відсутності.

4.3 Розробка інструкції користувача

Інструкція користувача описує технічні вимоги для запуску системи управління замовленнями товарів. Для забезпечення стабільної та ефективної роботи з програмним продуктом необхідно дотримуватися мінімальних та рекомендованих технічних вимог, які наведені в таблиці 4.1.

Таблиця 4.1 – Вимоги до апаратного забезпечення

Вимоги до конфігурування	Рекомендовано	Мінімально
Процесор	Intel Core i7 або аналогічний	Intel Core i5 або аналогічний
Оперативна пам’ять	16 ГБ RAM	8 ГБ RAM
Вільний простір на диску	20 ГБ	5 ГБ
Відеокарта	NVIDIA GeForce GTX 1650 або аналогічна	Вбудована графіка (Intel HD Graphics 4000 або краще)
Операційна система	Windows 10, macOS Monterey або Ubuntu 22.04	Windows 7, macOS Mojave або Ubuntu 18.04
Монітор	Роздільна здатність 1920x1080 пікселів, діагональ 15" або більше	Роздільна здатність 1280x720 пікселів, діагональ 13" або більше

Перед запуском вебсистеми необхідно встановити такі програмні компоненти:

1. Java 17 або новішу версію — обов’язкова для запуску системи;

2. Spring Boot — використовується для реалізації серверної частини;

3. MySQL або сумісна система керування базами даних.

Після встановлення MySQL необхідно створити нову базу даних, наприклад, orders_db. Далі потрібно налаштувати параметри підключення до бази даних у конфігураційному файлі application.properties або application.yml.

За потреби, таблиці для зберігання даних (замовлення, продукти, постачальники, користувачі тощо) можна створити вручну через SQL-запити або дозволити Spring Boot створити їх автоматично, якщо активована відповідна опція в конфігурації.

Далі, для створення бази даних, потрібно вручну створити відповідні таблиці для зберігання інформації про замовлення, постачальників, продукти та інші необхідні дані.

Щоб запустити сервер, можна використовувати командний рядок, що ініціює запуск сервера, який почне обробляти запити від користувачів.

Після успішного запуску серверу, для доступу до вебсистеми потрібно відкрити браузер та ввести адресу http://localhost:8080, що дасть змогу взаємодіяти з вебсистемою через інтерфейс користувача. Також є можливість переглядати статистичні дані щодо замовлень. Користувач може взаємодіяти з іншими даними через базу даних, такими як інформація про постачальників і продукти, щоб керувати процесами.

4.4 Висновки

У процесі тестування прикладного програмного інтерфесу для формування заявок на постачання товарів було застосовано методи «чорної скриньки» та «білої скриньки», що дозволило комплексно перевірити правильність роботи як з точки зору користувача, так і внутрішньої логіки системи.

Завдяки методу чорної скриньки вдалося підтвердити коректність функціонування основних сценаріїв взаємодії користувача із системою: оформлення та перегляд замовлень, керування товарами, постачальниками, а

також доступ до статистичних даних. Всі протестовані функції працювали згідно з очікуваними результатами, що підтверджує відповідність системи вимогам.

За допомогою методу білої скриньки було успішно перевірено логіку обробки запитів, взаємодію з базою даних та стійкість до помилкових ситуацій.

Результати тестування підтверджують, що вебсистема є стабільною, функціонально повною та готовою до використання. Додатково розроблена інструкція користувача забезпечує чітке розуміння вимог до середовища, процесу налаштування й запуску системи, що підвищує зручність її впровадження.

ВИСНОВКИ

У рамках виконання бакалаврської кваліфікаційної роботи було розроблено прикладний програмний інтерфейс для автоматизованого формування заявок на постачання товарів з урахуванням дистрибуційних факторів. Програмний продукт включає розробку серверної частини, яка здійснює аналіз залишків на складі, змін попиту та логістичних параметрів для раціонального планування постачання. Система побудована за архітектурною моделлю MVC із використанням шаблонізатора Thymeleaf для формування інтерфейсу користувача, а також REST API із форматом обміну даними JSON, що забезпечує взаємодію з внутрішніми ERP-системами та зовнішніми сервісами. Реалізація виконана мовою Java 17 з використанням фреймворку Spring Boot, а для збереження даних використовується СУБД mySQL. Бакалаврська кваліфікаційна робота відповідає вимогам згідно методичним вказівкам [30].

Під час розробки було проведено детальний аналіз основних вимог до системи, зокрема, необхідність автоматизації процесу формування заявок з урахуванням таких факторів, як залишки на складі, попит на продукцію та логістичні обмеження. Розроблено REST API для створення та перегляду заявок, а також модель бази даних mySQL, що зберігає інформацію про товари, заявки та залишки продукції. Розроблено механізм ранжування товарів за допомогою алгоритму Кендалла, що дозволяє розраховувати коефіцієнт залежності об'ємів продажів товарів від місяця року. У рамках реалізації були також розроблені алгоритми для розрахунку кількості товарів для замовлення в залежності від коефіцієнта рангової кореляції Кендалла.

Для забезпечення надійності роботи системи було проведено тестування програмного забезпечення на відповідність поставленим вимогам, що підтвердило його працездатність і стабільність. Тестування включало перевірку функціональності програмного інтерфейсу, а також перевірку роботи з базою даних і алгоритмами розрахунку. Система стабільно працює і готова до використання для процесу формування заявок на постачання товарів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. D. Longo, A. Cimino, and G. Mirabelli, “A general simulation framework for supply chain modeling: State of the art and case study,” Apr. 2022.

2. О. Н. Романюк, А. М. Думенко, «Використання штучного інтелекту для оптимізації поставок,» XVIII всеукраїнська науково-практична web конференція аспірантів, студентів та молодих вчених, 2025. [Електронний ресурс]. Доступно: https://sites.google.com/view/kicm/

3. О. Н. Романюк, А. М. Думенко, «Оптимізація логістичних процесів у сфері постачання товарів,» LIV Всеукраїнська науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії, 2025. [Електронний ресурс]. Доступно: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2025

4. О. Н. Романюк, А. М. Думенко, «Автоматизація логістики постачання товарів засобами API з урахуванням соціальних потреб,» XXXIII Міжнародна науково-практична конференція MicroCAD-2025. [Електронний ресурс].Доступно: https://ndch.kpi.kharkov.ua/wp-content/uploads/2025/05/Zbirnik-tez-2025.pdf

5. «Що таке API», [Електронний ресурс]. Доступно: https://hostiq.ua/blog/ukr/what-is-api/. Дата звернення: 28.03.2025.

6. «REST API як спосіб спілкування компонент вебдодатків», [Електронний ресурс]. Доступно: https://foxminded.ua/shcho-take-rest-api/. Дата звернення: 28.03.2025.

7. «Використання штучного інтелекту в логістиці та управлінні ланцюгами постачань», [Електронний ресурс]. Доступно: https://doi.org/10.31891/dsim-2024-8 . Дата звернення: 28.03.2025.

8. H. Murugan, *Big Data: Principles and Paradigms*. Wiley, 2023.

9. «Стратегії управління коригування маршрутів із використанням штучного інтелекту в логістиці», [Електронний ресурс]. Доступно:

https://www.zfort.com.ua/blog/chatgpt-i-shtuchnii-intelekt-u-logistici-30-prikladiv-vikoristannya . Дата звернення: 01.04.2025.

10. «SAP SCM (Supply Chain Management)», [Електронний ресурс]. Доступно: https://www.sap.com/products/supply-chain-management.html. Дата звернення: 01.04.2025.

11. «Oracle SCM Cloud», [Електронний ресурс]. Доступно: https://www.oracle.com/scm. Дата звернення: 01.04.2025.

12. «Blue Yonder (JDA Software)», [Електронний ресурс]. Доступно: https://www.blueyonder.com/. Дата звернення: 01.04.2025.

13. «Manhattan associates», [Електронний ресурс]. Доступно: https://www.manh.com/ . Дата звернення: 01.04.2025.

14. «Як Spring Boot допомагає прискорити розробку додатків», [Електронний ресурс]. Доступно: https://foxminded.ua/spring-boot/. Дата звернення: 09.04.2025.

15. «Model-View-Controller (MVC) Architecture», [Електронний ресурс]. Доступно: https://www.geeksforgeeks.org/mvc-design-pattern/ . Дата звернення: 15.04.2025.

16. «REST API: що це, як працює, переваги і приклади», [Електронний ресурс]. Доступно: https://goit.global/ua/articles/rest-api-shcho-tse-iak-pratsiuie-perevahy-i-pryklady/ . Дата звернення: 15.04.2025.

17. «Діаграма діяльності (UML) [Електронний ресурс]. Доступно: https://www.javatpoint.com/uml-activity-diagram Дата звернення: 20.04.2025.

18. «Draw.io», [Електронний ресурс]. Доступно: https://app.diagrams.net/. Дата звернення: 20.04.2025.

19. «Spring Boot makes it easy to create stand-alone», [Електронний ресурс]. Доступно: https://spring.io/projects/spring-boot . Дата звернення: 20.04.2025.

20. «New Features in Java 17», [Електронний ресурс]. Доступно: https://www.baeldung.com/java-17-new-features . Дата звернення: 20.04.2025.

21. H. Schildt, *Java: The Complete Reference*, 12th ed. New York, NY: McGraw-Hill, 2024.

22. C. Walls, *Spring Boot in Action*. Shelter Island, NY: Manning Publications, 2023.

23. «HTTP-запит», [Електронний ресурс]. Доступно: https://qalight.ua/baza-znan/http-zapyt-http-request/ . Дата звернення: 25.04.2025.

24. «MySQL: визначення, основні характеристики та компоненти», [Електронний ресурс]. Доступно: https://simplentrec.com/mysql-introduction/ . Дата звернення: 02.05.2025.

25. P. Bauer, *Spring Data: Modern Data Access for Enterprise Java*. Shelter Island, NY: Manning Publications, 2022.

26. «Introduction to Thymeleaf Template Engine», [Електронний ресурс]. Доступно: https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html . Дата звернення: 05.05.2025.

27. «Black-Box тестування», [Електронний ресурс]. Доступно: ttps://mate.academy/blog/qa/black-box-testing . Дата звернення 17.05.2025.

28. «White/black/grey box-тестування», [Електронний ресурс]. Доступно: https://qalight.ua/baza-znan/white-black-grey-box-testuvannya/ . Дата звернення 25.05.2025.

29. «Mockito – Java mocking framework», [Електронний ресурс]. Доступно: https://site.mockito.org/ . Дата звернення: 25.05.25.

30. Методичні вказівки до виконання бакалаврських кваліфікаційних робіт для студентів спеціальності 121 «Інженерія програмного забезпечення» / Уклад: О. Н. Романюк, Г. О. Черноволик. – Вінниця : ВНТУ, 2022. – 51 с.

ДОДАТКИ

Додаток А. Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедри
Романюк О. Н.
«25» березня 2025 р.

Технічне завдання
на бакалаврську кваліфікаційну роботу
«Розробка прикладного програмного інтерфейсу для формування заявок на постачання товарів з урахуванням дистрибуційних факторів»
за спеціальністю 121 – Інженерія програмного забезпечення

Керівник бакалаврської кваліфікаційної роботи:
д.т.н., проф. Романюк О. Н.
«24» березня 2025 р.

Виконала:
студент гр. ЗПІ-21б Думенко А. М.
«24» березня 2025 р.

Вінниця – 2025 року

1. Найменування та галузь застосування

Бакалаврська кваліфікаційна робота: «Розробка прикладного програмного інтерфейсу для формування заявок на постачання товарів з урахуванням дистрибуційних факторів».

Галузь застосування – інформаційні технології у сфері логістики та управління ланцюгами постачання.

2. Підстава для розробки

Підставою для виконання бакалаврської кваліфікаційної роботи (БКР) є індивідуальне завдання на БКР та наказ №97 від 20 березня 2025 року про закріплення тем БКР.

3. Мета та призначення розробки

Метою роботи є автоматизація процесу формування заявок на постачання товарів шляхом розробки прикладного програмного інтерфейсу, який інтегрує дистрибуційні фактори (сезонність, географічний розподіл, логістичні обмеження), аналізує історичні дані про продажі за допомогою алгоритму рангової кореляції Кендалла.

Призначення роботи – методи та засоби оптимізації логістичних процесів на основі прикладного програмного інтерфейсу.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БКР:

1. Методичні вказівки до виконання бакалаврських кваліфікаційних робіт для студентів спеціальності 121 «Інженерія програмного забезпечення» / Уклад: О. Н. Романюк, Г. О. Черноволик. – Вінниця : ВНТУ, 2022. – 51 с.

2. D. Longo, A. Cimino, and G. Mirabelli, “A general simulation framework for supply chain modeling: State of the art and case study,” Apr. 2022.

3. H. Schildt, *Java: The Complete Reference*, 12th ed. New York, NY: McGraw-Hill, 2024.

4. «REST API як спосіб спілкування компонент вебдодатків», [Електронний ресурс]. Доступно: https://foxminded.ua/shcho-take-rest-api/. Дата звернення: 28.03.2025.

5. Технічні вимоги

Вихідні дані до роботи: тип роботи – веборієнтований серверний застосунок; модель розробки – ітеративна модель; засоби організації даних програми – фреймворк Spring Boot для серверної частини, MySQL як система керування базами даних, можливість інтеграції з клієнтськими інтерфейсами через REST API; вхідні дані: інформація про товари (назва, категорія, ціна, залишки), дані про постачальників, історія продажів, запити користувачів, результати попередніх прогнозів; вихідні дані: сформовані заявки на постачання з детальною специфікацією; ранжовані списки товарів з урахуванням коефіцієнта Кендалла та сезонності; аналітичні звіти про прогнозований попит; статуси заявок та інформація про обробку замовлень; середовище розробки – IntelliJ IDEA; мова розробки – Java; операційна система – Windows 10/11.

6. Конструктивні вимоги

Вебзастосунок та вебклієнт повинні відповідати ергономічним та технічним вимогам. Текстова та графічна документація повинна відповідати діючим стандартам України.

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

- пояснювальна записка до бакалаврської кваліфікаційної роботи;
- технічне завдання;
- 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 кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз розвитку технологій автоматизованого формування заявок на постачання товарів та постановка задач дослідження	25.03.25 – 08.04.25
2	Розробка моделі та алгоритмів програмного продукту	09.04.25 – 20.04.25
3	Варіантний аналіз та обґрунтування вибору засобів програмної реалізації	21.04.25 – 23.04.25
4	Розробка програмних модулів та методів реалізації системи формування заявок	24.04.25 – 20.05.25
5	Тестування застосунку	21.05.25 – 25.05.25
6	Оформлення матеріалів до захисту БКР	26.05.25 – 30.05.25

10. Порядок контролю та прийняття.

Усі етапи бакалаврської кваліфікаційної роботи контролюються науковим керівником згідно плану по виконанню роботи.

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

Додаток Б. Протокол перевірки бакалаврської кваліфікаційної роботи на наявність текстових запозичень

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Розробка прикладного програмного інтерфейсу для формування заявок на постачання товарів з урахуванням дистрибуційних факторів

Тип роботи: бакалаврська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ: кафедра програмного забезпечення, ФІТКІ, група ЗПІ-21б
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 3,8 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне).

■ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту.

□ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

□ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

_____ _____
(прізвище, ініціали, посада) (підпис)

_____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Черноволик Г. О.

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Романюк О. Н.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Думенко А. М.
(підпис) (прізвище, ініціали)

Додаток В. Лістинг програми

OrderController.java

```
package com.ndumenko.logistic_helper.controller;

import com.ndumenko.logistic_helper.service.OrderService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;

import java.time.LocalDate;

@Controller
public class OrderController {

    @Autowired
    private OrderService orderService;

    @PostMapping(path = "/order")
    public String createOrder(@RequestParam("supplierId") int supplierId,
@RequestParam("date") LocalDate date, Model model) {
        model.addAttribute("order", orderService.createOrder(supplierId, date));
        return "order";
    }

    @GetMapping(path = "/order")
    public String getOrder(@RequestParam("orderId") int orderId, Model model) {
        model.addAttribute("order", orderService.getOrder(orderId));
        return "order";
    }
```

```
    @GetMapping(path = "/orders")
    public String getOrders(@RequestParam("supplierId") int supplierId, Model model) {
        model.addAttribute("orders", orderService.getOrders(supplierId));
        return "supplierOrders";
    }
}
```

SupplierController.java

```
package com.ndumenko.logistic_helper.controller;
import com.ndumenko.logistic_helper.db.repository.SupplierRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class SupplierController {

    @Autowired
    private SupplierRepository supplierRepository;

    @GetMapping(path = "/")
    public String getSuppliers(Model model) {
        model.addAttribute("suppliers", supplierRepository.findAll());
        return "suppliers";
    }
}
```

Category.java

```
package com.ndumenko.logistic_helper.db.dto;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.Data;
```

```
@Data
@Entity
@Table(name = "categories")
public class Category {

    @Id
    private int id;
    private String name;
}
```

ConfigurationParameter.java

```
package com.ndumenko.logistic_helper.db.dto;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.Data;

@Data
@Entity
@Table(name = "config")
public class ConfigurationParameter {

    @Id
    private String key;
    private String value;
}
```

Order.java

```
package com.ndumenko.logistic_helper.db.dto;

import jakarta.persistence.*;
import lombok.Data;
```

```
import java.sql.Timestamp;

@Data
@Entity
@Table(name = "orders")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @OneToOne
    @JoinColumn(name = "supplier_id")
    private Supplier supplier;
    private Timestamp orderDate;
    private Timestamp deliveryDate;
}
```

OrderedProduct.java

```
package com.ndumenko.logistic_helper.db.dto;

import jakarta.persistence.*;
import lombok.Data;

@Data
@Entity
@Table(name = "ordered_products")
public class OrderedProduct {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @OneToOne
    @JoinColumn(name = "order_id")
    private Order order;
    @OneToOne
```

```
    @JoinColumn(name = "product_id")
    private Product product;
    private float quantity;
    private float price;
}
```

Product.java

```
package com.ndumenko.logistic_helper.db.dto;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import lombok.Data;

@Data
@Entity
@Table(name = "products")
public class Product {

   @Id
   private int id;
   private String name;
   @OneToOne
   @JoinColumn(name = "category_id")
   private Category category;
   private float salePrice;
   private float purchasePrice;
   @OneToOne
   @JoinColumn(name = "supplier_id")
   private Supplier supplier;
   private float weight;
   private float packageSize;
   private float remains;
```

```
    private boolean active;
}
```

SoldProduct.java

```
package com.ndumenko.logistic_helper.db.dto;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import lombok.Data;

import java.sql.Timestamp;

@Data
@Entity
@Table(name = "sold_products")
public class SoldProduct {

    @Id
    private long id;
    private Timestamp saleDate;
    @OneToOne
    @JoinColumn(name = "product_id")
    private Product product;
    private float quantity;
    private float price;
}
```

CorrelationTable.java

```
package com.ndumenko.logistic_helper.model.correlation;

import lombok.Data;
```

```
import java.util.*;

@Data
public class CorrelationTable {

    private List<Item> items = new ArrayList<>();
    private static Map<Integer, Integer> monthRanks = new HashMap<>();

    static {
        monthRanks.put(1, 3);
        monthRanks.put(2, 1);
        monthRanks.put(3, 4);
        monthRanks.put(4, 6);
        monthRanks.put(5, 9);
        monthRanks.put(6, 10);
        monthRanks.put(7, 11);
        monthRanks.put(8, 12);
        monthRanks.put(9, 8);
        monthRanks.put(10, 7);
        monthRanks.put(11, 2);
        monthRanks.put(12, 5);
    }

    public void addItem(int x, float y) {
        items.add(new Item(x, y));
    }

    public float calculateCoefficient() {

        // Set Y ranks
        int currentRank = 0;
        items.sort((a, b) -> Float.compare(a.getY(), b.getY()));
        for (Item item : items) {
            currentRank++;
```

```
        item.setYRank(currentRank);
    }

    // Set X ranks
    for (Item item : items) {
        item.setXRank(monthRanks.get(item.getX()));
    }
    // Sort by X rank
    items.sort(Comparator.comparingInt(Item::getXRank));

    // Calculate coefficient
    int n = items.size();
    int P = 0;
    int Q = 0;
    for (int i = 0; i < n - 1; i++) {
        int greaterRanksCount = 0;
        int lessRanksCount = 0;
        int currentYRank = items.get(i).getYRank();
        for (int j = i + 1; j < n; j++) {
            if (items.get(j).getYRank() > currentYRank) {
                greaterRanksCount++;
            } else if (items.get(j).getYRank() < currentYRank) {
                lessRanksCount++;
            }
        }
        P += greaterRanksCount;
        Q += lessRanksCount;
    }
    return ((float) 2 * (P - Q)) / (n * (n - 1));
}

@Data
public static class Item {
    private int x;
    private float y;
```

```
        private int xRank;
        private int yRank;

        public Item(int x, float y) {
            this.x = x;
            this.y = y;
        }
    }
}
```

Order.java

```
package com.ndumenko.logistic_helper.model;

import com.ndumenko.logistic_helper.db.dto.Product;
import com.ndumenko.logistic_helper.db.dto.Supplier;
import lombok.Data;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

@Data
public class Order {

    private int id;
    private LocalDate date;
    private Supplier supplier;
    private List<ProductOrder> products = new ArrayList<>();

    public void addProduct(Product product, int quantity) {
        ProductOrder productOrder = new ProductOrder();
        productOrder.setId(product.getId());
        productOrder.setName(product.getName());
        productOrder.setQuantity(quantity);
        productOrder.setPrice(product.getPurchasePrice());
```

```
        products.add(productOrder);
    }
}
```

ProductStatistic.java

```
package com.ndumenko.logistic_helper.model;

import com.ndumenko.logistic_helper.db.dto.Product;
import com.ndumenko.logistic_helper.db.dto.SoldProduct;

public class ProductStatistic {

    public ProductStatistic (SoldProduct soldProduct) {
        this.product = soldProduct.getProduct();
    }

    private Product product;
    private float priceCoefficient;
    private float seasonCoefficient;
}
```

SeasonCoeffic.java

```
package com.ndumenko.logistic_helper.model;
import java.util.Map;
public class SeasonCoefficients {
    private final Map<Integer, Float> monthCoefficients = new java.util.HashMap<>();
    public void updateMonthCoefficient(int month, float coeficient) {
        monthCoefficients.merge(month, coeficient, (a, b) -> (a + b) / 2);
    }
    public Float getMonthCoefficient(int month) {
        return monthCoefficients.get(month);
    }
}
```

DataConvecterlmpl.java

```
package com.ndumenko.logistic_helper.service.impl;

import com.ndumenko.logistic_helper.db.dto.OrderedProduct;
import com.ndumenko.logistic_helper.db.dto.Product;
import com.ndumenko.logistic_helper.db.dto.SoldProduct;
import com.ndumenko.logistic_helper.model.Order;
import com.ndumenko.logistic_helper.model.ProductSaleData;
import com.ndumenko.logistic_helper.service.DataConverter;
import org.springframework.stereotype.Service;

import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.List;

@Service
public class DataConverterImpl implements DataConverter {

    @Override
    public List<ProductSaleData> convertSoldProducts(List<SoldProduct> soldProducts) {
        List<ProductSaleData> converted = soldProducts.stream().map(this::convert).toList();
        List<ProductSaleData> groupedByYearAndMonth = new ArrayList<>();
        converted.forEach(sale -> {
            ProductSaleData existing = groupedByYearAndMonth.stream()
                    .filter(saleData -> saleData.getProduct().getId() == sale.getProduct().getId()
                            && saleData.getYear() == sale.getYear()
                            && saleData.getMonth() == sale.getMonth())
                    .findFirst()
                    .orElse(null);

            if (existing != null) {
                existing.setQuantity(existing.getQuantity() + sale.getQuantity());
                existing.setPrice(existing.getPrice() + sale.getPrice());
            } else {
```

```
                groupedByYearAndMonth.add(sale);
            }
        });
        return groupedByYearAndMonth;
    }

    @Override
    public List<OrderedProduct> convertOrder(Order order) {
        com.ndumenko.logistic_helper.db.dto.Order orderDto = new
com.ndumenko.logistic_helper.db.dto.Order();
        orderDto.setSupplier(order.getSupplier());
        orderDto.setOrderDate(Timestamp.valueOf(order.getDate().atStartOfDay()));
        orderDto.setDeliveryDate(Timestamp.valueOf(order.getDate().atStartOfDay()));

        List<OrderedProduct> result = new ArrayList<>();

        order.getProducts().forEach(product -> {
            Product productDto = new Product();
            productDto.setId(product.getId());
            productDto.setName(product.getName());

            OrderedProduct orderedProductDto = new OrderedProduct();
            orderedProductDto.setOrder(orderDto);
            orderedProductDto.setProduct(productDto);
            orderedProductDto.setQuantity(product.getQuantity());
            orderedProductDto.setPrice(product.getPrice());

            result.add(orderedProductDto);
        });
        return result;
    }

    private ProductSaleData convert(SoldProduct soldProduct) {
        ProductSaleData productSaleData = new ProductSaleData();
        productSaleData.setProduct(soldProduct.getProduct());
```

```
    Calendar calendar = Calendar.getInstance();
    calendar.setTime(soldProduct.getSaleDate());
    productSaleData.setMonth(calendar.get(Calendar.MONTH) + 1);
    productSaleData.setYear(calendar.get(Calendar.YEAR));
    productSaleData.setPrice(soldProduct.getPrice());
    productSaleData.setQuantity(soldProduct.getQuantity());
    return productSaleData;
}

@Override
public Order convertToOrder(List<OrderedProduct> orderedProducts) {
    if (orderedProducts == null || orderedProducts.isEmpty()) {
        return null;
    }
    com.ndumenko.logistic_helper.db.dto.Order orderDto = orderedProducts.get(0).getOrder();
    Order order = new Order();
    order.setSupplier(orderDto.getSupplier());
    order.setDate(orderDto.getOrderDate().toLocalDateTime().toLocalDate());

    orderedProducts.forEach(orderedProduct -> {
        Product product = new Product();
        product.setId(orderedProduct.getProduct().getId());
        product.setName(orderedProduct.getProduct().getName());
        order.addProduct(product, (int) orderedProduct.getQuantity());
    });

    return order;
}

@Override
public List<Order> convertToOrders(List<OrderedProduct> orderedProducts) {
    if (orderedProducts == null || orderedProducts.isEmpty()) {
        return List.of();
    }
    // Map to group ordered products by order ID
```

```
        var orderMap = new java.util.HashMap<Integer, Order>();
        orderedProducts.forEach(orderedProduct -> {
            int orderId = orderedProduct.getOrder().getId();
            Order order = orderMap.computeIfAbsent(orderId, id -> {
                Order newOrder = new Order();
                newOrder.setId(orderId);
                newOrder.setSupplier(orderedProduct.getOrder().getSupplier());

newOrder.setDate(orderedProduct.getOrder().getOrderDate().toLocalDateTime().toLocalDate());
                return newOrder;
            });

            Product product = new Product();
            product.setId(orderedProduct.getProduct().getId());
            product.setName(orderedProduct.getProduct().getName());
            order.addProduct(product, (int) orderedProduct.getQuantity());
        });
        // Add all orders to the list
        return new ArrayList<>(orderMap.values());
    }
}
```

OrderServiceImpl.java

```
package com.ndumenko.logistic_helper.service.impl;

import com.ndumenko.logistic_helper.db.dto.*;
import com.ndumenko.logistic_helper.db.repository.OrderRepository;
import com.ndumenko.logistic_helper.db.repository.OrderedProductRepository;
import com.ndumenko.logistic_helper.db.repository.SoldProductsRepository;
import com.ndumenko.logistic_helper.db.repository.SupplierRepository;
import com.ndumenko.logistic_helper.model.Order;
import com.ndumenko.logistic_helper.model.ProductOrder;
import com.ndumenko.logistic_helper.model.ProductSaleData;
import com.ndumenko.logistic_helper.model.SeasonCoefficients;
import com.ndumenko.logistic_helper.model.correlation.CorrelationTable;
```

```
import com.ndumenko.logistic_helper.service.ConfigurationService;
import com.ndumenko.logistic_helper.service.DataConverter;
import com.ndumenko.logistic_helper.service.OrderService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.time.LocalDate;
import java.util.*;
import java.util.stream.Collectors;

@Service
public class OrderServiceImpl implements OrderService {

    public static final Comparator<ProductSaleData>
PRODUCT_SALE_DATA_COMPARATOR = (o1, o2) -> {
        if (o1.getYear() == o2.getYear()) {
            return Integer.compare(o1.getMonth(), o2.getMonth());
        } else {
            return Integer.compare(o1.getYear(), o2.getYear());
        }
    };

    @Autowired
    private ConfigurationService configurationService;
    @Autowired
    private SupplierRepository supplierRepository;
    @Autowired
    private SoldProductsRepository soldProductsRepository;
    @Autowired
    private DataConverter dataConverter;
    @Autowired
    private OrderedProductRepository orderedProductRepository;
    @Autowired
    private OrderRepository orderRepository;
```

```
    @Override
    public Order createOrder(int supplierId, LocalDate date) {

        Supplier supplier = supplierRepository.findById(supplierId).orElseThrow(() -> new IllegalArgumentException("Supplier not found"));

        Order order = new Order();
        order.setSupplier(supplier);
        order.setDate(date);

        List<SoldProduct> soldProductsDB = soldProductsRepository.findByProductSupplierId(supplierId);
        List<ProductSaleData> soldProducts = dataConverter.convertSoldProducts(soldProductsDB);

        Map<Category, List<ProductSaleData>> soldProductsByCategory = soldProducts.stream()
                .collect(Collectors.groupingBy(sale -> sale.getProduct().getCategory()));

        soldProductsByCategory.forEach((category, productSaleData) -> processCategory(date, productSaleData, order));

        correctOrderQuantity(supplier, soldProducts, order);
        saveOrder(order);
        return order;
    }

    private void saveOrder(Order order) {
        List<OrderedProduct> orderedProducts = dataConverter.convertOrder(order);
        com.ndumenko.logistic_helper.db.dto.Order savedOrder = orderRepository.save(orderedProducts.get(0).getOrder());
        orderedProducts.forEach(product -> product.setOrder(savedOrder));
        orderedProductRepository.saveAll(orderedProducts);
    }
```

```
    private void processCategory(LocalDate date, List<ProductSaleData> productSaleData, Order order) {
        // Calculate rank correlation coefficients for each category
        // Calculate rank correlation coefficient for seasonality
        CorrelationTable seasonCorrelationTable = new CorrelationTable();
        LocalDate now = LocalDate.now();
        int currentMonth = now.getMonth().getValue();
        int currentYear = now.getYear();
        productSaleData.forEach(soldProduct -> {
            if (soldProduct.getMonth() != currentMonth || soldProduct.getYear() != currentYear) {
                seasonCorrelationTable.addItem(soldProduct.getMonth(), soldProduct.getQuantity());
            }
        });
        float seasonalityCoefficient = seasonCorrelationTable.calculateCoefficient();

        if (seasonalityCoefficient > 0.5 || seasonalityCoefficient < -0.5) {
            // If the seasonality coefficient is greater than 0.5 or less than -0.5, then the seasonality coefficient is used
            processSeasonDependentCategory(date, productSaleData, order);
        } else {
            // Calculate average sales for each product
            processSeasonIndependentCategory(date, productSaleData, order);
        }
    }

    private void processSeasonDependentCategory(LocalDate date, List<ProductSaleData> productSaleData, Order order) {
        // Calculate month coefficients for each product
        Map<Product, SeasonCoefficients> productSeasonCoefficients = new HashMap<>();
        Map<Product, List<ProductSaleData>> salesByProduct = productSaleData.stream().collect(Collectors.groupingBy(ProductSaleData::getProduct));
        salesByProduct.forEach((product, sales) -> calculateProductMonthCoefficients(product, sales, productSeasonCoefficients));

        // Get product sales for the previous month
```

```
        int previousMonth = date.minusMonths(1).getMonth().getValue();
        int year = date.minusMonths(1).getYear();
        Map<Product, ProductSaleData> previousMonthSales = productSaleData.stream()
                .filter(sale -> sale.getMonth() == previousMonth && sale.getYear() == year)
                .collect(Collectors.toMap(ProductSaleData::getProduct, sale -> sale));

        // Get sales for the current month
        int currentMonth = date.getMonth().getValue();
        int currentYear = date.getYear();
        Map<Product, ProductSaleData> currentMonthSales = productSaleData.stream()
                .filter(sale -> sale.getMonth() == currentMonth && sale.getYear() == currentYear)
                .collect(Collectors.toMap(ProductSaleData::getProduct, sale -> sale));

        // Calculate predicted sales for each product for the current month
        productSeasonCoefficients.forEach((product, seasonCoefficients) -> {
            ProductSaleData previousSale = previousMonthSales.get(product);
            if (previousSale != null) {
                float predictedQuantity = previousSale.getQuantity() *
seasonCoefficients.getMonthCoefficient(currentMonth);
                float currentQuantity = currentMonthSales.get(product) != null ?
currentMonthSales.get(product).getQuantity() : 0;
                order.addProduct(product, Math.max(Math.round(predictedQuantity - currentQuantity
- product.getRemains()), 0));
            } else {
                order.addProduct(product, 0);
            }
        });

    }

    private void calculateProductMonthCoefficients(Product product, List<ProductSaleData>
productSaleData, Map<Product, SeasonCoefficients> productSeasonCoefficients) {
        productSaleData.sort(PRODUCT_SALE_DATA_COMPARATOR);
        SeasonCoefficients seasonCoefficients = new SeasonCoefficients();
        LocalDate now = LocalDate.now();
```

```
            int currentMonth = now.getMonth().getValue();
            int currentYear = now.getYear();

            for (int i = 0; i < productSaleData.size(); i++) {
                ProductSaleData currentSale = productSaleData.get(i);
                int month = currentSale.getMonth();
                float quantity = currentSale.getQuantity();
                if (month == currentMonth && currentSale.getYear() == currentYear) {
                    continue; // Skip current month sales
                }
                float previousQuantity = 0;
                float nextQuantity = 0;
                int divider = 1;
                if (i > 0) {
                    previousQuantity = productSaleData.get(i - 1).getQuantity();
                    divider++;
                }
                if (i < productSaleData.size() - 1) {
                    nextQuantity = productSaleData.get(i + 1).getQuantity();
                    divider++;
                }
                if (i > 0) {
                    float coefficient = ((quantity + previousQuantity + nextQuantity) / divider) /
previousQuantity;
                    seasonCoefficients.updateMonthCoefficient(month, coefficient);
                }
            }

            productSeasonCoefficients.put(product, seasonCoefficients);
        }

        private void processSeasonIndependentCategory(LocalDate date, List<ProductSaleData>
productSaleData, Order order) {

            // Calculate average sales for each product
```

```
        Map<Product, Float> productAverageSales = new HashMap<>();
        productSaleData.sort(PRODUCT_SALE_DATA_COMPARATOR);
        productSaleData.forEach(sale -> {
            Product product = sale.getProduct();
            productAverageSales.merge(product, sale.getQuantity(), (average, current) -> (average + current) / 2);
        });

        // Get sales for the current month
        int currentMonth = date.getMonth().getValue();
        int currentYear = date.getYear();
        Map<Product, ProductSaleData> currentMonthSales = productSaleData.stream()
                .filter(sale -> sale.getMonth() == currentMonth && sale.getYear() == currentYear)
                .collect(Collectors.toMap(ProductSaleData::getProduct, sale -> sale));

        productAverageSales.forEach((product, avgQuantity) -> {
            ProductSaleData currentSale = currentMonthSales.get(product);
            float currentSaleQuantity = currentSale != null ? currentSale.getQuantity() : 0;
            order.addProduct(product, Math.max(Math.round(avgQuantity - currentSaleQuantity - product.getRemains()), 0));
        });
    }

    private void correctOrderQuantity(Supplier supplier, List<ProductSaleData> soldProducts, Order order) {
        Float maxStorageCapacity = configurationService.getMaxStorageCapacity();
        float maxStoreQuantity;
        if (maxStorageCapacity == null) {
            throw new IllegalStateException("Max store capacity is not set");
        } else {
            maxStoreQuantity = maxStorageCapacity * supplier.getStorageShare() / 100;
        }

        float currentRemains = soldProducts.stream()
                .map(ProductSaleData::getProduct)
```

```
                .distinct()
                .map(Product::getRemains)
                .reduce(0f, Float::sum);
        float availableOrderQuantity = maxStoreQuantity - currentRemains;
        int totalOrderQuantity = order.getProducts().stream()
                .map(ProductOrder::getQuantity)
                .reduce(0, Integer::sum);
        float correctionCoefficient = availableOrderQuantity / totalOrderQuantity;

        if (correctionCoefficient < 1) {
            order.getProducts().forEach(productOrder -> productOrder.setQuantity(Math.round(productOrder.getQuantity() * correctionCoefficient)));
        }
    }

    @Override
    public Order getOrder(int orderId) {
        List<OrderedProduct> allByOrderId = orderedProductRepository.findAllByOrderId(orderId);
        return dataConverter.convertToOrder(allByOrderId);
    }

    @Override
    public List<Order> getOrders(int supplierId) {
        List<OrderedProduct> allBySupplierId = orderedProductRepository.findAllByOrder_Supplier_Id(supplierId);
        return dataConverter.convertToOrders(allBySupplierId);
    }
}
```

CorrelationTableTest.java

```
package com.ndumenko.logistic_helper.model.correlation;

import org.junit.jupiter.api.Test;
```

```
import static org.junit.jupiter.api.Assertions.assertTrue;

public class CorrelationTableTest {

    @Test
    public void testCalculateCoefficient_positiveSeasonDependentCategory() {
        // Given
        CorrelationTable correlationTable = new CorrelationTable();
        correlationTable.addItem(1, 30.0f);
        correlationTable.addItem(2, 10.0f);
        correlationTable.addItem(3, 40.0f);
        correlationTable.addItem(4, 60.0f);
        correlationTable.addItem(5, 90.0f);
        correlationTable.addItem(6, 100.0f);
        correlationTable.addItem(7, 110.0f);
        correlationTable.addItem(8, 120.0f);
        correlationTable.addItem(9, 80.0f);
        correlationTable.addItem(10, 70.0f);
        correlationTable.addItem(11, 20.0f);
        correlationTable.addItem(12, 50.0f);

        // When
        float coefficient = correlationTable.calculateCoefficient();

        // Then
        assertTrue(coefficient > 0.5, "Coefficient should be greater than 0.5");
    }

    @Test
    public void testCalculateCoefficient_negativeSeasonDependentCategory() {
        // Given
        CorrelationTable correlationTable = new CorrelationTable();
        correlationTable.addItem(1, 100.0f);
        correlationTable.addItem(2, 120.0f);
        correlationTable.addItem(3, 90.0f);
```

```
    correlationTable.addItem(4, 70.0f);
    correlationTable.addItem(5, 40.0f);
    correlationTable.addItem(6, 30.0f);
    correlationTable.addItem(7, 20.0f);
    correlationTable.addItem(8, 10.0f);
    correlationTable.addItem(9, 50.0f);
    correlationTable.addItem(10, 60.0f);
    correlationTable.addItem(11, 110.0f);
    correlationTable.addItem(12, 80.0f);

    // When
    float coefficient = correlationTable.calculateCoefficient();

    // Then
    assertTrue(coefficient < -0.5, "Coefficient should be less than -0.5");
}

@Test
public void testCalculateCoefficient_seasonIndependentCategory() {
    // Given
    CorrelationTable correlationTable = new CorrelationTable();
    correlationTable.addItem(1, 25.0f);
    correlationTable.addItem(2, 30.0f);
    correlationTable.addItem(3, 30.0f);
    correlationTable.addItem(4, 35.0f);
    correlationTable.addItem(5, 30.0f);
    correlationTable.addItem(6, 25.0f);
    correlationTable.addItem(7, 30.0f);
    correlationTable.addItem(8, 35.0f);
    correlationTable.addItem(9, 25.0f);
    correlationTable.addItem(10, 35.0f);
    correlationTable.addItem(11, 25.0f);
    correlationTable.addItem(12, 30.0f);

    // When
```

```
        float coefficient = correlationTable.calculateCoefficient();

        // Then
        assertTrue(coefficient < 0.5 && coefficient > -0.5, "Coefficient should be less than 0.5 and
greater then -0.5");
    }
}
```

OrderedServiceImplTest.java

```
package com.ndumenko.logistic_helper.service.impl;
import com.ndumenko.logistic_helper.db.dto.*;
import com.ndumenko.logistic_helper.db.repository.OrderRepository;
import com.ndumenko.logistic_helper.db.repository.OrderedProductRepository;
import com.ndumenko.logistic_helper.db.repository.SoldProductsRepository;
import com.ndumenko.logistic_helper.db.repository.SupplierRepository;
import com.ndumenko.logistic_helper.model.Order;
import com.ndumenko.logistic_helper.model.ProductSaleData;
import com.ndumenko.logistic_helper.service.ConfigurationService;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import java.time.LocalDate;
import java.util.List;
import java.util.Optional;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.ArgumentMatchers.anyList;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
public class OrderServiceImplTest {
    @Mock
```

```
    private ConfigurationService configurationService;
    @Mock
    private SupplierRepository supplierRepository;
    @Mock
    private SoldProductsRepository soldProductsRepository;
    @Mock
    private OrderedProductRepository orderedProductRepository;
    @Mock
    private OrderRepository orderRepository;
    @Mock
    private DataConverterImpl dataConverter;
    @InjectMocks
    private OrderServiceImpl orderService;

    private Supplier supplier;
    private LocalDate date;

    @BeforeEach
    void setUp() {
        supplier = new Supplier();
        supplier.setId(1);
        supplier.setStorageShare(50f);
        date = LocalDate.of(2023, 10, 1);
    }

    @Test
    void createOrder_validSupplierAndData() {
        // GIVEN

when(supplierRepository.findById(supplier.getId())).thenReturn(Optional.of(supplier));

when(soldProductsRepository.findByProductSupplierId(supplier.getId())).thenReturn(List.o
f(new SoldProduct()));
        Category category = new Category();
        Product product = new Product();
```

```
        product.setCategory(category);
        ProductSaleData saleData = new ProductSaleData();
        saleData.setProduct(product);
        saleData.setMonth(1);
        saleData.setYear(2025);
        saleData.setQuantity(200f);
        when(dataConverter.convertSoldProducts(anyList())).thenReturn(List.of(saleData));

        when(configurationService.getMaxStorageCapacity()).thenReturn(1000f);

        OrderedProduct orderedProduct = new OrderedProduct();
        orderedProduct.setOrder(new com.ndumenko.logistic_helper.db.dto.Order());
        when(dataConverter.convertOrder(any())).thenReturn(List.of(orderedProduct));

        when(orderRepository.save(any())).thenReturn(new
com.ndumenko.logistic_helper.db.dto.Order());
        when(orderedProductRepository.saveAll(anyList())).thenReturn(null);

        // WHEN
        Order result = orderService.createOrder(supplier.getId(), date);
        // THEN
        assertNotNull(result);
        verify(orderRepository, times(1)).save(any());
        verify(orderedProductRepository, times(1)).saveAll(anyList());
    }

    @Test
    void createOrder_supplierNotFound() {
        // GIVEN
        when(supplierRepository.findById(supplier.getId())).thenReturn(Optional.empty());
        // WHEN & THEN
        assertThrows(IllegalArgumentException.class,                ()                ->
orderService.createOrder(supplier.getId(), date));
    }
```

```
    @Test
    public void getOrder_orderExist() {
        // GIVEN
        int orderId = 1;
        List<OrderedProduct> orderedProducts = List.of(new OrderedProduct());

when(orderedProductRepository.findAllByOrderId(orderId)).thenReturn(orderedProducts);

        Order expectedResult = new Order();
        when(dataConverter.convertToOrder(orderedProducts)).thenReturn(expectedResult);

        // WHEN
        Order result = orderService.getOrder(orderId);
        // THEN
        assertEquals(expectedResult, result);
    }

    @Test
    public void getOrder_orderNotExist() {
        // GIVEN
        int orderId = 1;
        when(orderedProductRepository.findAllByOrderId(orderId)).thenReturn(List.of());
        // WHEN
        Order result = orderService.getOrder(orderId);
        // THEN
        assertNull(result);
    }

    @Test
    public void getOrders_ordersExist() {
        // GIVEN
        int supplierId = 1;
        List<OrderedProduct> orderedProducts = List.of(new OrderedProduct());
```

```
when(orderedProductRepository.findAllByOrder_Supplier_Id(supplierId)).thenReturn(order
edProducts);

        List<Order> expectedResult = List.of(new Order());
        when(dataConverter.convertToOrders(orderedProducts)).thenReturn(expectedResult);

        // WHEN
        List<Order> result = orderService.getOrders(supplierId);
        // THEN
        assertEquals(expectedResult, result);
    }

    @Test
    public void getOrders_ordersNotExist() {
        // GIVEN
        int supplierId = 1;

when(orderedProductRepository.findAllByOrder_Supplier_Id(supplierId)).thenReturn(List.
of());

        // WHEN
        List<Order> result = orderService.getOrders(supplierId);
        // THEN
        assertEquals(List.of(), result);
    }
}
```

Додаток Г. Ілюстративна частина

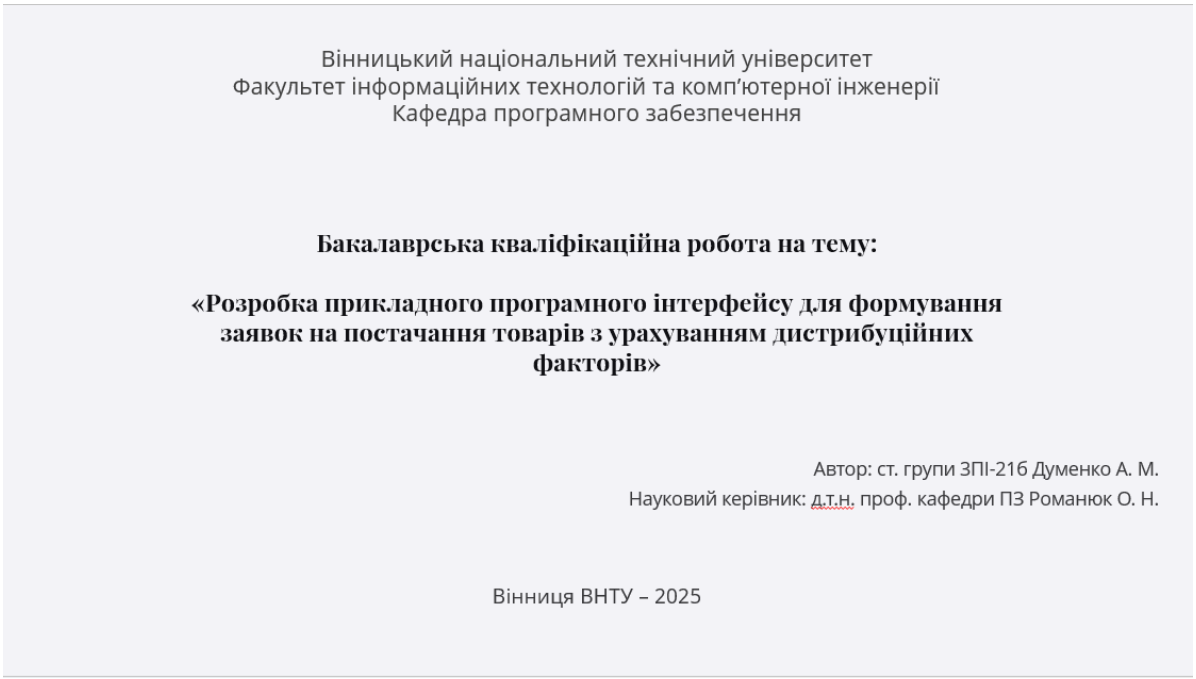


Рисунок Г.1 – Титульний слайд

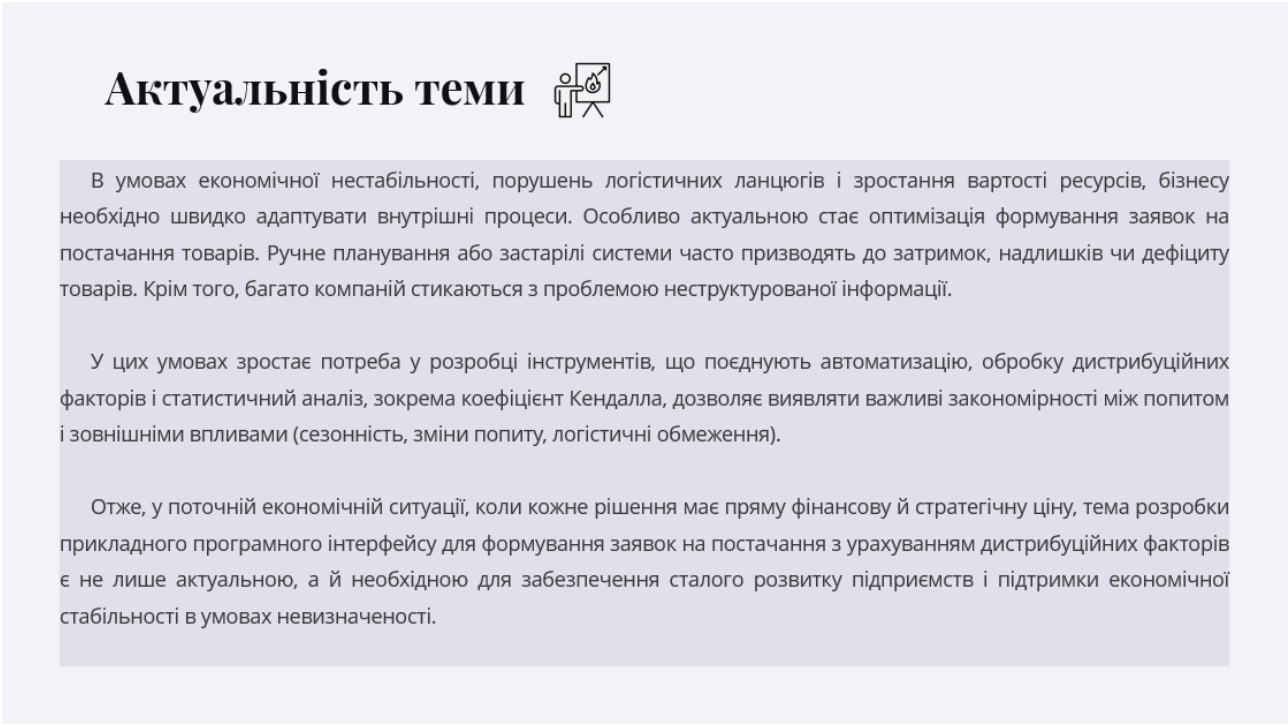


Рисунок Г.2 – Актуальність теми

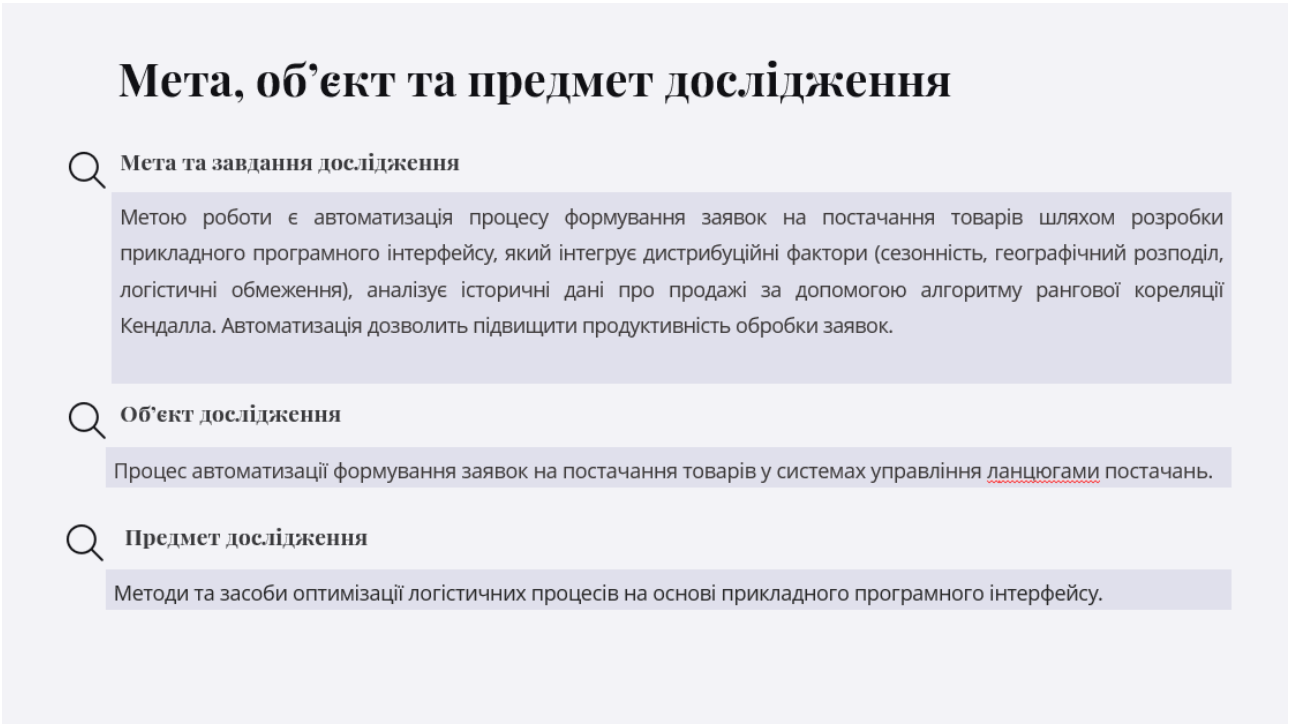


Рисунок Г.3 – Мета, об’єкт та предмет дослідження

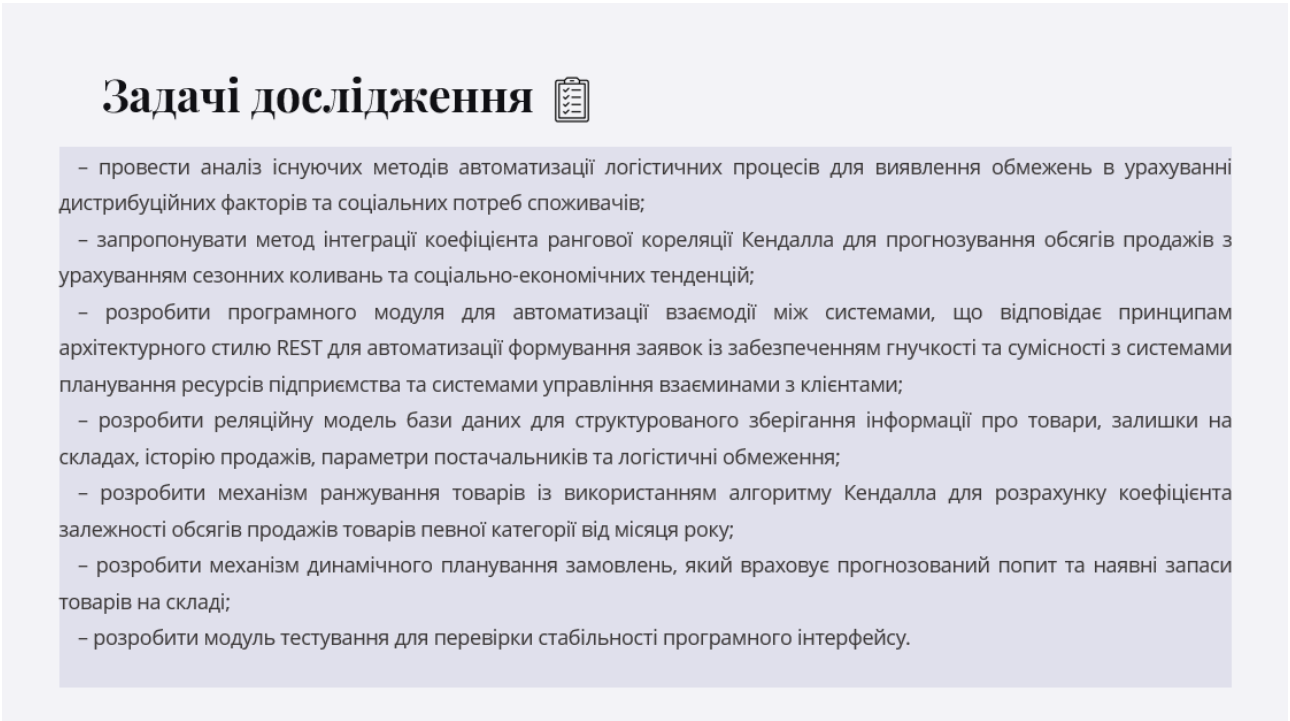


Рисунок Г.4 – Завдання дослідження

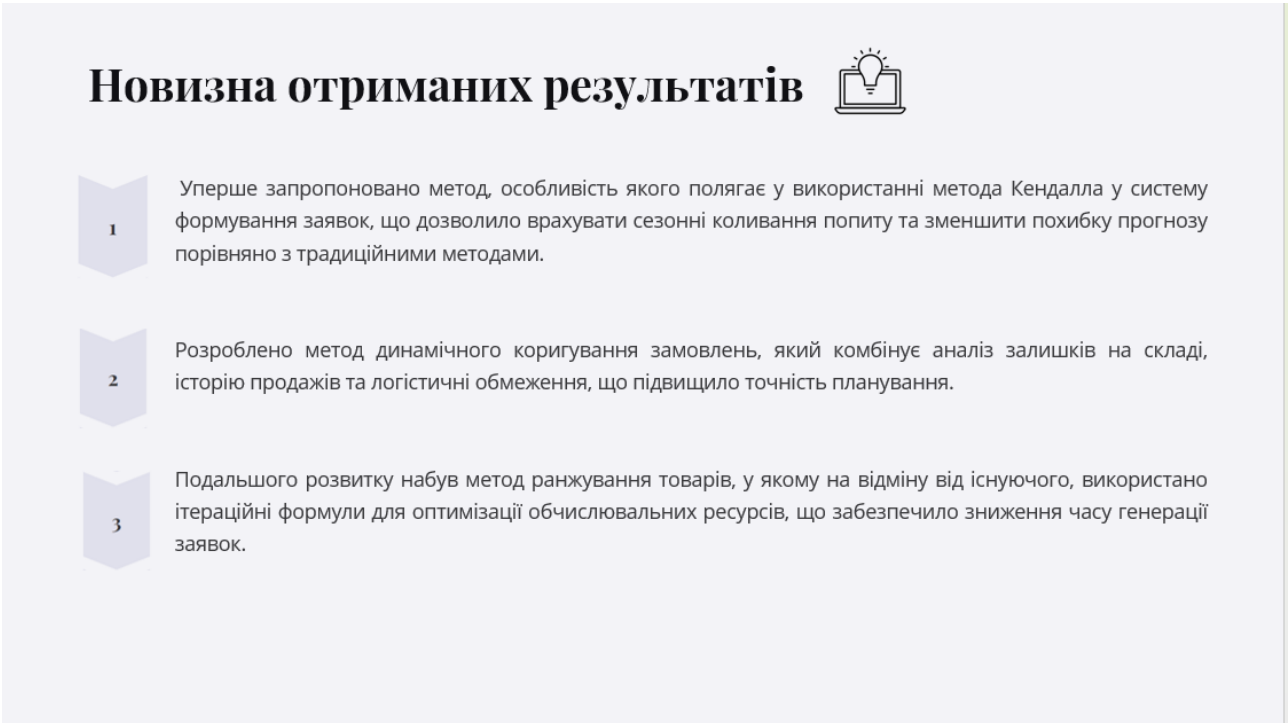


Рисунок Г.5 – Новизна отриманих результатів

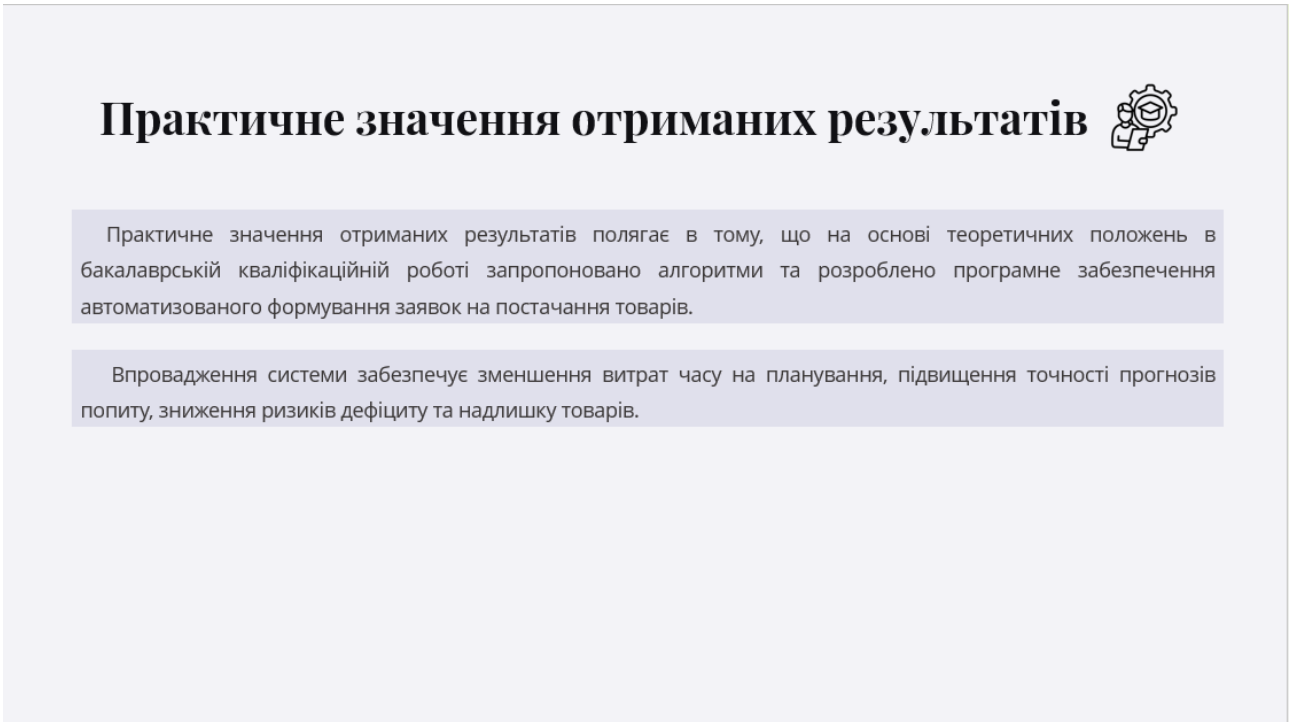


Рисунок Г.6 – Практичне значення отриманих результатів

Аналіз сучасних систем автоматизації управління постачаннями

Проведений аналіз сучасних систем підтвердив те, що існуючі рішення мають певні обмеження:

1. SAP SCM: висока вартість впровадження, складність налаштування, недоступна для малого/середнього бізнесу
2. Oracle SCM Cloud: високі вимоги до IT-інфраструктури, складність конфігурації, потреба в зовнішніх фахівцях
3. Blue Yonder: складне впровадження, дорогі ліцензії, потреба в тривалому навчанні персоналу
4. Manhattan Associates: висока вартість, потреба в спеціалізованому навчанні, орієнтованість на великі корпорації

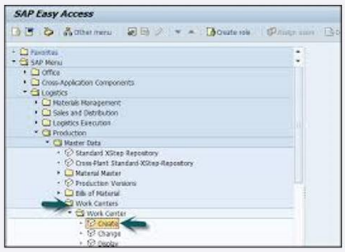
Отже, у програмному застосунку необхідно поєднати: хмарну архітектуру для масштабування, інтеграцію через API для обміну даними в реальному часі, модулі машинного навчання для прогнозування та оптимізації та інтуїтивно зрозумілий інтерфейс. Використання автоматизованих засобів управління сприяє зменшенню впливу людського фактору на процеси постачання, що, у свою чергу, знижує ризики помилок і підвищує точність прогнозування.

Рисунок Г.7 – Аналіз сучасних систем

Аналоги системи управління ланцюгами постачання

VS

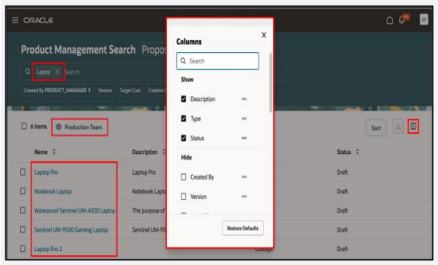
1



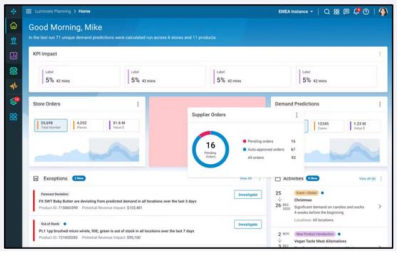
До найбільш популярних аналогів відносять:

1. SAP SCM
2. Oracle SCM Cloud
3. Blue Yonder (JDA Software)
4. Manhattan Associates.

2



3



4

Рисунок Г.8 – Аналоги системи управління постачаннями

Основний алгоритм функціонування системи

1. Користувач надає ідентифікатор постачальника (supplierId) та дату створення нового замовлення, які слугують відправною точкою процесу формування замовлення.
2. Система класифікує товари за категоріями для подальшої аналітичної обробки в межах кожної окремої групи
3. Розрахунок коефіцієнта сезонності. Для кожної категорії товарів виконується розрахунок сезонного коефіцієнта продажів із використанням відповідної аналітичної функції calculateCoefficient(List<ProductSaleData>).
4. Якщо розрахований коефіцієнт перевищує порогове значення (0,5), здійснюється поквартальний або помісячний аналіз продажів. Обсяг замовлення визначається за формулою (amount=previousMonthSale* seasonCoefficients.get(currentMonth)), після чого товари додаються до відповідного замовлення.
5. Якщо коефіцієнт сезонності дорівнює або не перевищує 0,5, кількість продукції визначається як середнє арифметичне значення обсягу продажів: amount = totalSaleAmount / months.
6. Після формування попереднього замовлення система виконує перевірку відповідності загального обсягу продукції доступному вільному простору на складі та, за необхідності, коригує кількість продукції.
7. Сформоване та відкориговане замовлення зберігається в базі даних.
8. На завершальному етапі користувачу надається підсумкове замовлення з усіма деталями, включно з кількісними показниками та категоріями товарів.

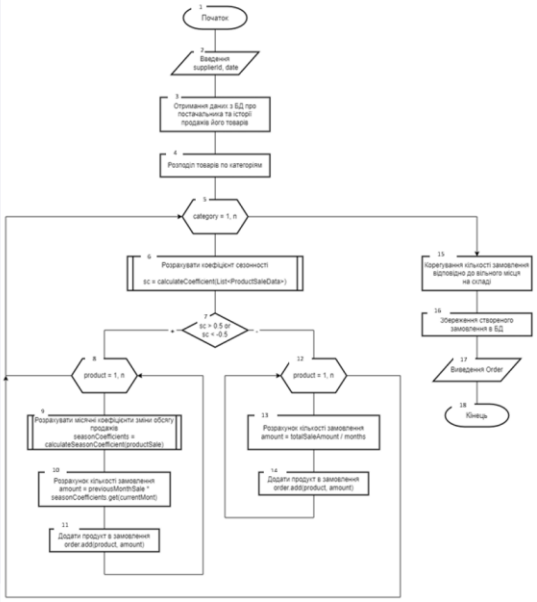


Рисунок Г.9 – Основний алгоритм функціонування системи

Алгоритм Кендалла для обчислення коефіцієнту кореляції

Алгоритм Кендалла для обчислення коефіцієнта кореляції починається з функції calculateCoefficient(productSale), яка отримує дані про продажі продуктів. Спочатку створюється таблиця кореляцій, в яку додаються елементи, що містять місяць і суму продажу. Для кожного запису перевіряється, чи місяць та рік не співпадають з поточними, після чого елементи додаються до таблиці.

Наступним кроком є сортування елементів таблиці за значеннями Y (сума продажу), після чого кожному елементу присвоюється ранг. Потім алгоритм встановлює ранги для осі X (місяць продажу), сортує таблицю за цими рангами та переходить до підрахунку кількості більших та менших елементів, порівнюючи їхні ранги по осі Y.

За допомогою підрахованих значень алгоритм обчислює коефіцієнт Кендалла, використовуючи формулу для коефіцієнта кореляції, що залежить від кількості пар елементів з більшими або меншими рангами. Результат обчислення коефіцієнта кореляції виводиться наприкінці процесу.

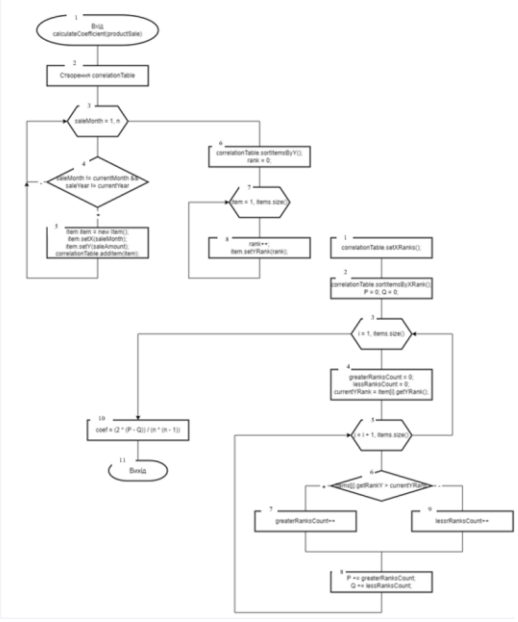


Рисунок Г.10 – Алгоритм Кендалла для обчислення коефіцієнту кореляції

Алгоритм розрахунку сезонного коефіцієнту

Процес починається з виклику функції calculateSeasonCoefficient(sales), яка приймає список продажів. Створюється об'єкт seasonCoefs для збереження коефіцієнтів сезонності. На початковому етапі дані про продажі сортуються за роком і місяцем.
Далі запускається цикл для кожного продажу. Для кожного елемента перевіряється, чи місяць та рік відрізняються від поточного. Якщо умова виконується, визначається місяць продажу та обсяг продажу. Також ініціалізуються змінні prevAmount, nextAmount і divider.

Якщо є попередній запис (i >1), з нього береться обсяг продажу і збільшується дільник (divider++). За аналогічною логікою обробляється і наступний запис, якщо він існує. Після цього розраховується коефіцієнт сезонності за формулою: середнє значення обсягів продажу поділяється на попередній обсяг продажу. Отримане значення записується в об'єкт seasonCoefs для відповідного місяця. Цикл повторюється для кожного запису у списку продажів. Після завершення обробки всіх записів функція завершує роботу, а сформовані коефіцієнти сезонності доступні для подальшого використання.

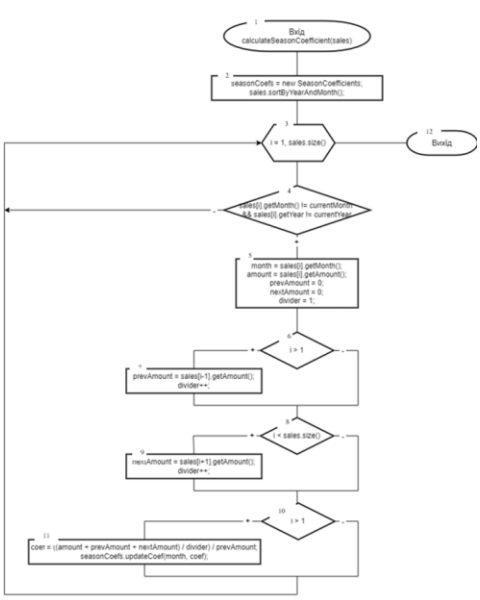
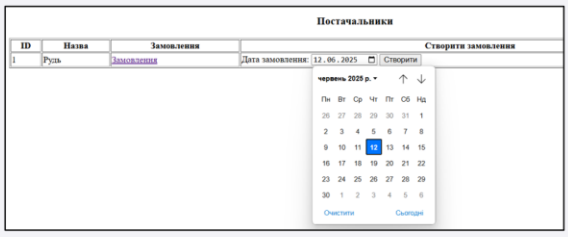
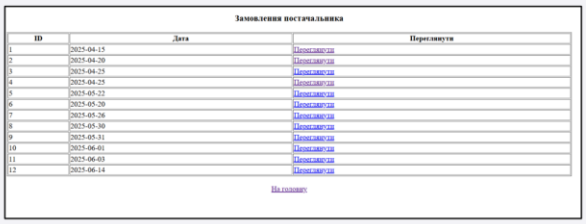


Рисунок Г.11 – Алгоритм розрахунку сезонного коефіцієнту

Розробка графічного інтерфейсу



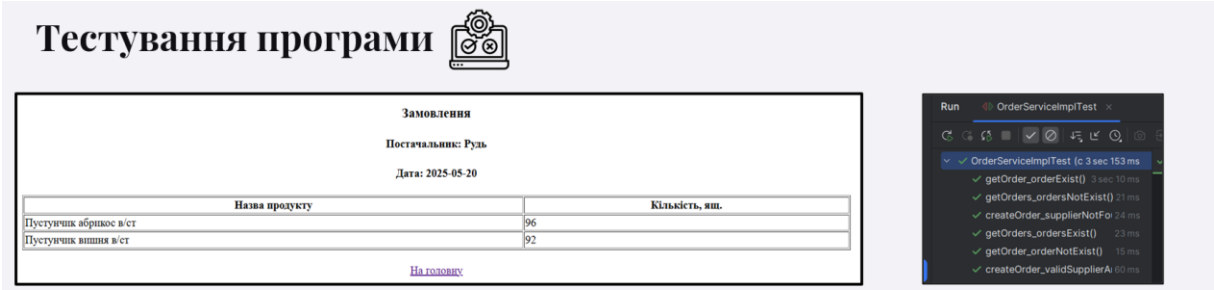


Інтерфейс розроблено з урахуванням потреб підприємств, що займаються постачанням товарів, а також з дотриманням ключових принципів юзабіліті. Основна мета – забезпечити швидке, зрозуміле та ефективне управління замовленнями, що сприяє підвищенню продуктивності роботи користувачів.

Головна сторінка містить таблицю постачальників із відображенням їх ID, назв та замовлень. Користувач може перейти до перегляду замовлень конкретного постачальника, де представлені дати створення замовлень. При виборі конкретного замовлення відображаються деталі: обсяг вільного складського простору, перелік товарів та їх кількість по місяцях.

Інтерфейс створено за допомогою шаблонізатора Thymeleaf, який дозволяє безперервно оновлювати контент сторінок без перезавантаження, що підтримує зручність використання; забезпечує просту передачу даних між серверною логікою та інтерфейсом, спрощуючи розробку; дозволяє легко будувати складні візуалізації і структури даних;

Рисунок Г.12 – Розробка графічного інтерфейсу



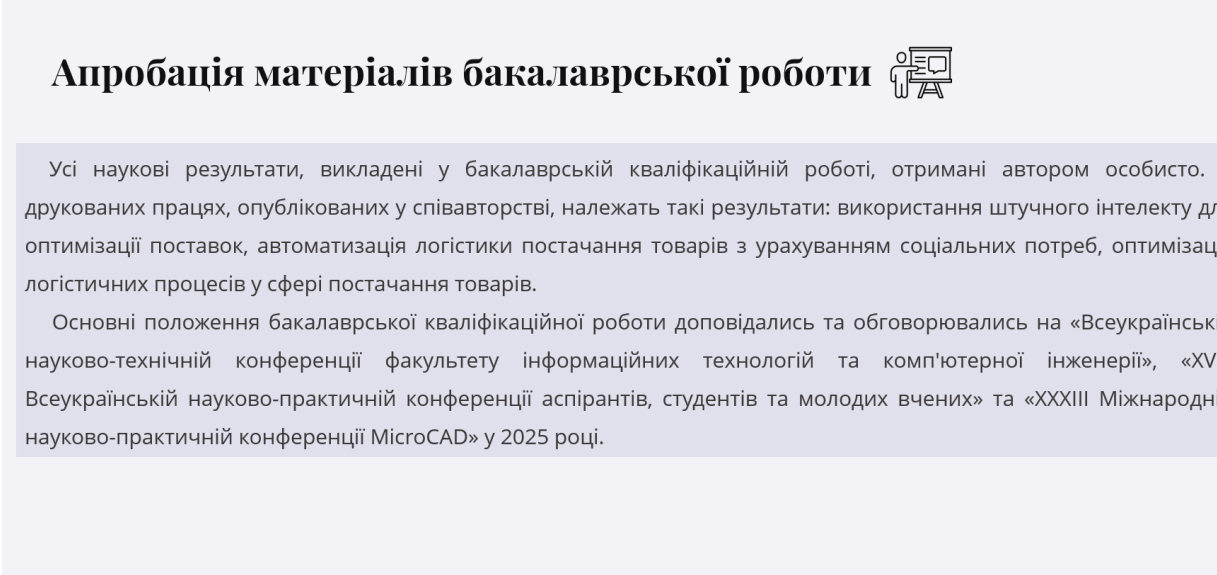
З метою перевірки працездатності, надійності та відповідності функціональності розробленого програмного забезпечення було проведено комплексне тестування із застосуванням методів «чорної скриньки» та «білої скриньки».

Метод «чорної скриньки» використовувався для перевірки роботи системи з точки зору кінцевого користувача. Було розроблено набір тестових випадків, які моделювали типові сценарії використання, зокрема створення заявки з різними параметрами. Метою було переконатися, що система коректно реагує на валідні та невалідні введення, надає очікувані повідомлення про помилки та виконує необхідні дії відповідно до введених даних.

Метод «білої скриньки» дозволив провести тестування внутрішньої логіки системи. Особливу увагу приділено перевірці правильності обробки даних на стороні сервера та коректності виконання SQL-запитів. Результати показали, що коефіцієнти коректно відображають позитивні, негативні або відсутні залежності, що підтверджує правильність реалізації логіки. Також протестовано отримання замовлень за ідентифікатором і список усіх замовлень для постачальника.

Усі тести були успішно виконані, що свідчить про стабільну роботу системи, правильну реалізацію бізнес-логіки та відповідність програмного забезпечення поставленим вимогам.

Рисунок Г.13 – Тестування програми



Усі наукові результати, викладені у бакалаврській кваліфікаційній роботі, отримані автором особисто. У друкованих працях, опублікованих у співавторстві, належать такі результати: використання штучного інтелекту для оптимізації поставок, автоматизація логістики постачання товарів з урахуванням соціальних потреб, оптимізація логістичних процесів у сфері постачання товарів.

Основні положення бакалаврської кваліфікаційної роботи доповідались та обговорювались на «Всеукраїнській науково-технічній конференції факультету інформаційних технологій та комп'ютерної інженерії», «XVIII Всеукраїнській науково-практичній конференції аспірантів, студентів та молодих вчених» та «XXXIII Міжнародній науково-практичній конференції MicroCAD» у 2025 році.

Рисунок Г.14 – Апробація матеріалів бакалаврської роботи

Висновки

У бакалаврській кваліфікаційній роботі виконано розробку розробку програмного засобу у вигляді програмного інтерфейсу, що покращує системи формування заявок на постачання товарів шляхом інтеграції статистичних методів аналізу дистрибуційних факторів та сприяє зменшенню витрат, підвищенню точності прогнозування попиту та адаптації логістичних систем до динамічних ринкових умов.

Розробка виконана в середовищі IntelliJ IDEA з використанням мови програмування Java 17, фреймворку Spring Boot, MySQL для зберігання даних та модель MVC для обробки запитів.

В процесі виконання бакалаврської кваліфікаційної роботи було проведено аналіз існуючих систем управління постачаннями, зокрема таких, що використовують традиційні методи прогнозування та планування, як-от ручне введення даних, статичні шаблони замовлень або прості моделі. Було розглянуто основні механізми таких систем, зокрема використання фіксованих замовлень на основі історичних даних без урахування сезонних змін, відсутність автоматизованих розрахунків коефіцієнтів кореляції та обмеженість у гнучкій візуалізації даних.

Проведено порівняльний аналіз, який виявив переваги створеного застосунку. Серед них – простий та зрозумілий інтерфейс, автоматизоване формування заявок з урахуванням сезонних залежностей, динамічне оновлення даних за допомогою шаблонізатора Thymeleaf, а також швидкий доступ до інформації, що разом забезпечує вищу ефективність і зручність у щоденній роботі користувача.

Рисунок Г.15 – Висновки