ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Автоматизована система для фінансово-облікових операцій на базі веб-додатку»

на здобуття освітнього ступеня магістра
зі спеціальності 123 Комп'ютерної інженерії
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні системи та мережі
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ *Едуард Веремійчик*
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач(ка) вищої освіти гр. КСДМ-62
Едуард Веремійчик
Ім'я, ПРІЗВИЩЕ

Керівник: Віталій Волошин
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Рецензент: _____
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Комп'ютерної інженерії

Ступінь вищої освіти Магістр

Спеціальність 123 «Комп'ютерна інженерія»

Освітньо-професійна програма Комп'ютерні системи та мережі

ЗАТВЕРДЖУЮ
Завідувач кафедрою Наталія ЛАЩЕВСЬКА
_____Ім'я, ПРІЗВИЩЕ
«_____» _____ 2024р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Веремійчик Едуард Русланович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Автоматизована система для фінансово-облікових операцій на базі веб-додатку »
керівник кваліфікаційної роботи Віталій Волошин,
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024р. №320,
2. Строк подання кваліфікаційної роботи «29» грудня 2024р.
3. Вихідні дані до кваліфікаційної роботи: Визначити основні вимоги та завдання, які повинні бути вирішені під час виконання дипломної роботи. Розробка цієї системи має на меті автоматизувати процеси обліку фінансових операцій, таких як облік доходів та витрат, створення бюджетів, а також забезпечення аналітики фінансової ситуації для користувачів. Задля цього необхідно створити інтерфейс, який дозволяє зручним чином вводити, переглядати та редагувати фінансові операції в режимі реального часу. Крім того, веб-додаток повинен підтримувати генерацію фінансових звітів, графіків і діаграм для більш детального аналізу даних. Важливим аспектом є забезпечення безпеки особистої та фінансової інформації користувачів, що потребує використання сучасних методів шифрування і захисту даних. Розроблений додаток має бути доступний через веб-браузери і працювати на всіх основних платформах. Вибір технологій для реалізації системи передбачає використання таких інструментів, як HTML, CSS, JavaScript, а також серверних мов програмування і баз даних для зберігання фінансової інформації..
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Розробка теоретичної частини
2. Вивчення функціоналу вимоги до системи
3. Аналіз архітектури
4. Ознайомлення з інтерфейсом веб-додатку, вивчення елементів управління
5. Перелік ілюстративного матеріалу: *презентація*
6. Дата видачі завдання «01» вересня 2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Збір даних	10.10-12.10.2024	Виконано
2	Обробка матеріалу	12.10-15.10.2024	Виконано
3	Розробка теоретичної частини	15.10-18.10.2024	Виконано
4	Вивчення функціональних вимог до системи: управління обліковими даними, генерація звітності, безпека даних.	18.10-22.10.2024	Виконано
5	Аналіз архітектури: структура бази даних, клієнт-серверна взаємодія, основні компоненти веб-додатку	22.10-26.10.2024	Виконано
6	Ознайомлення з інтерфейсом веб-додатку, вивчення елементів управління, налаштування облікових записів, доступ до звітності.	26.10-30.10.2024	Виконано
7	Висновки за результатами аналізу	30.10-31.10.2024	Виконано
8	Розробка презентації	31.10-01.11.2024	Виконано
9	Передзахист	09.12-11.12.2024	
10	Здача в деканат	20.12-29.12.2024	

Здобувач вищої освіти _____ Едуард Веремійчик
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Віталій Волошин
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістр: 65 стор., 25 мал., 1 табл., 20 джерел.

Мета роботи – розробити веб-сайт, що автоматизуватиме процес обчислення та управління фінансами.

Об'єкт дослідження – розробка та впровадження автоматизованої системи для управління фінансово-обліковими процесами з використанням веб-додатку.

Предмет дослідження – процеси розробки, впровадження та функціонування автоматизованої системи для фінансово-облікових операцій на основі веб-застосунку.

Короткий зміст роботи: У роботі розглянуто розробку та впровадження сучасної автоматизованої системи для управління фінансово-обліковими операціями.

Проаналізовано сучасні тенденції у сфері автоматизації фінансового обліку, включаючи зростаючу потребу в надійності, продуктивності та безпеці даних. Машинне навчання застосовується для автоматичного аналізу фінансових транзакцій, виявлення аномалій та створення прогнозів. Хмарні обчислення забезпечують масштабовану обробку великих обсягів фінансових даних, підвищуючи швидкість та доступність системи для користувачів з різних регіонів.

КЛЮЧОВІ СЛОВА: ФІНАНСОВИЙ ОБЛІК, ВЕБ-ДОДАТОК, ІНТЕГРАЦІЯ ДАНИХ, ХМАРНІ ОБЧИСЛЕННЯ, ЕФЕКТИВНІСТЬ ОБЛІКОВИХ ПРОЦЕСІВ, БЕЗПЕКА ДАНИХ, ОПТИМІЗАЦІЯ ВИТРАТ, МАСШТАБОВАНІСТЬ, АНАЛІЗ ТРАНЗАКЦІЙ, ВИЯВЛЕННЯ АНОМАЛІЙ, ІНТЕРФЕЙС КОРИСТУВАЧА, БУХГАЛТЕРСЬКИЙ ОБЛІК, УПРАВЛІННЯ ФІНАНСАМИ, СИСТЕМНА АРХІТЕКТУРА, ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ, АВТОМАТИЗАЦІЯ БІЗНЕС-ПРОЦЕСІВ, РЕАЛЬНИЙ ЧАС.

ABSTRACT

The text part of the qualification work for obtaining the master's degree: 65 pages, 25 figures, 1 table, 20 sources.

The purpose of the work is to develop a website that automates the process of calculating and managing finances.

The object of the research is the development and implementation of an automated system for managing financial and accounting processes using a web application.

The subject of the research is the processes of development, implementation, and operation of an automated system for financial and accounting operations based on a web application.

Summary of the work: The study examines the development and implementation of a modern automated system for managing financial and accounting operations. Current trends in the field of financial accounting automation are analyzed, including the growing need for reliability, efficiency, and data security. Machine learning is applied for automatic analysis of financial transactions, anomaly detection, and forecasting. Cloud computing provides scalable processing of large volumes of financial data, enhancing the system's speed and accessibility for users from various regions.

KEYWORDS: FINANCIAL ACCOUNTING, WEB APPLICATION, DATA INTEGRATION, CLOUD COMPUTING, ACCOUNTING PROCESS EFFICIENCY, DATA SECURITY, COST OPTIMIZATION, SCALABILITY, TRANSACTION ANALYSIS, ANOMALY DETECTION, USER INTERFACE, ACCOUNTING, FINANCIAL MANAGEMENT, SYSTEM ARCHITECTURE, INFORMATION TECHNOLOGY, BUSINESS PROCESS AUTOMATION, REAL-TIME.

ЗМІСТ

ВСТУП

У сучасному світі веб-технології стали незамінною частиною нашого щоденного життя та важливим інструментом у фінансовому управлінні. Розробка та вдосконалення веб-сайтів, які допомагають користувачам ефективно контролювати свої фінанси, планувати бюджет, вести облік доходів і витрат, стали значущим елементом для досягнення фінансової стабільності та успіху. Завдяки сучасним онлайн-платформам користувачі отримують доступ до зручних інструментів для ведення фінансових обліків, що сприяє покращенню фінансової грамотності та відповідального ставлення до споживання.

Створення веб-сайту для управління фінансами має на меті не лише відслідковувати доходи та витрати, а й надавати рекомендації щодо оптимізації витрат, заощадження та досягнення фінансових цілей. Важливими аспектами є зручний інтерфейс, доступність функцій та інтеграція з іншими фінансовими системами, що дає можливість користувачам аналізувати свої фінансові потоки в будь-який час і з будь-якого місця.

Веб-сайти для фінансового управління надають користувачам можливість не лише вести детальний облік витрат, а й створювати бюджети, отримувати фінансові звіти, встановлювати цілі та відслідковувати їх досягнення. Крім того, вони часто мають функції для налаштування нагадувань про регулярні платежі або важливі фінансові події, що допомагає підтримувати фінансову дисципліну і уникати непередбачених ситуацій.

Робота з такими веб-сайтами дозволяє користувачам краще розуміти свої фінанси та приймати зважені рішення щодо заощаджень і інвестицій. У світі, де економічні умови постійно змінюються, ці інструменти стають важливими для збереження та примноження особистих фінансів. Інтерактивні функції, такі як графіки та аналіз фінансових звітів, надають чітке уявлення про фінансову ситуацію, даючи змогу коригувати стратегії витрат вчасно.

Отже, використання веб-сайтів для фінансового управління є важливою складовою фінансової грамотності та успішного управління грошовими ресурсами. Вони дозволяють не тільки контролювати витрати, але й планувати своє фінансове майбутнє, роблячи цей процес простим, зрозумілим і доступним для кожного. Сучасні технології відкривають нові можливості для ефективного фінансового управління, а вміння користуватися такими інструментами є необхідною навичкою для тих, хто прагне до фінансової незалежності.

РОЗДІЛ 1 АЛГОРИТМИ І СТРУКТУРИ ДАНИХ

Алгоритми і структура даних є фундаментальними поняттями в програмуванні та комп’ютерних науках, що визначають, як ми можемо обробляти, зберігати та маніпулювати даними. Алгоритм — це чіткий та скінченний набір інструкцій або правил, що описує, як розв’язати певну задачу чи досягти конкретного результату, починаючи з вихідних даних. Це послідовність дій, яка веде від проблеми до її рішення. Алгоритми знаходять застосування в програмуванні, математиці, інженерії та багатьох інших галузях для обробки, аналізу даних та автоматизації процесів.

Щоб набір інструкцій можна було вважати алгоритмом, він повинен мати кілька важливих властивостей: скінченність, тобто алгоритм повинен завершуватись за скінченну кількість кроків; однозначність, коли кожна команда повинна бути точною та зрозумілою; результативність, що означає отримання конкретного результату після завершення алгоритму; визначеність, коли алгоритм має давати однаковий результат при однакових вхідних даних; і ефективність, що полягає в оптимальному використанні ресурсів, таких як час і пам'ять.

Алгоритми можна класифікувати за призначенням, способом реалізації та структурою даних. За призначенням виділяють алгоритми сортування, пошуку та графічні алгоритми. За способом реалізації розрізняють рекурсивні та ітеративні алгоритми. Що стосується структури даних, алгоритми можуть бути орієнтовані на списки, черги, стеки, дерева та хеш-таблиці.

Алгоритми є центральною частиною програмування, оскільки вони визначають спосіб обробки та маніпуляції даними. Правильно обраний і оптимізований алгоритм може значно підвищити ефективність роботи програми. Програмісти та інженери оцінюють алгоритми за швидкістю виконання (часова складність) та за обсягом пам'яті (просторова складність), необхідного для їх виконання.

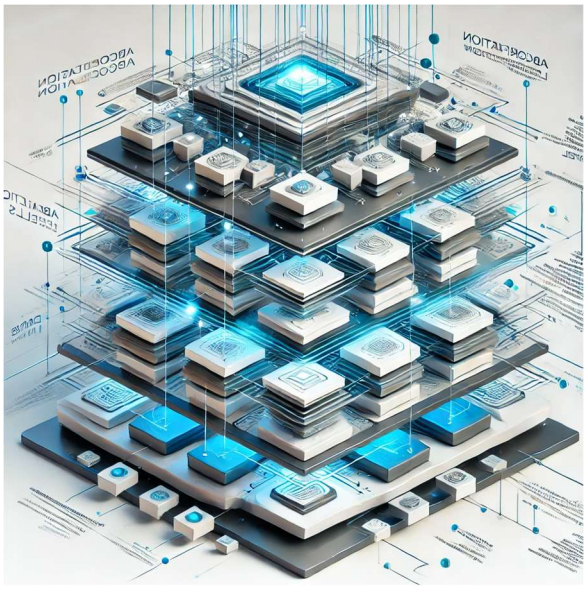
Ефективність алгоритму зазвичай оцінюється за допомогою двох основних критеріїв: часова складність, що вимірює, скільки часу потрібно алгоритму для виконання завдання, та просторова складність, яка визначає, скільки пам'яті використовує алгоритм під час виконання.

Алгоритми є основою сучасних комп’ютерних технологій. Вони використовуються в штучному інтелекті, криптографії, обробці великих даних та Інтернеті речей. Алгоритми дозволяють вирішувати задачі швидко і з мінімальними ресурсами, що робить їх невід'ємною частиною всього, що відбувається у світі інформаційних технологій.

1.1 Структурування і абстракція програм

У процесі проектування складних програмних систем суттєву роль відіграють принципи структуризації та абстрагування. Імплементація цих принципів дозволяє розробникам поділити основну задачу на менші підзадачі, визначити ключові компоненти разом із їх абстракціями, а також акцентувати увагу на більш істотних аспектах програми, абстрагуючи від незначних деталей. Це сприяє більш ефективному управлінню складними структурами, спрощуючи процес розробки й полегшуючи підтримку та оновлення програмного продукту.

У процесі розробки програм застосовуються два ключові підходи до їх структурування: низхідне проектування (зверху вниз) і висхідне проектування (знизу вгору). Низхідне проектування розпочинається з визначення загальних функціональних можливостей програми, після чого вони розподіляються на дрібніші завдання, що забезпечує ясну організацію взаємодій між модулями на різних рівнях, які відповідають за різні аспекти функціонування програми. Висхідне проектування акцентується на організації даних: процес створення програми стартує з елементарних типів даних, які об'єднуються в єдину структуру, в межах якої всі операції зосереджені на трансформації вхідних даних у вихідні.



Малюнок 1.1 – Структурування і абстракція програм

Одним з головних складових структурування є інкапсуляção, яка дозволяє формувати "чорні ящики", модулі або класи, які приховують свої внутрішні реалізації.

Це знижує ризик випадкових помилок, забезпечуючи доступ лише через визначені інтерфейси, що полегшує підтримку та розширення коду. Комбіноване використання низхідного і висхідного проектування дозволяє досягти оптимальної структуризації програм. Такий підхід дозволяє гнучко адаптувати програму до вимог проекту, що є основою сучасних методів розробки програмного забезпечення, таких як об'єктно-орієнтоване програмування. Це підвищує надійність і гнучкість під час розробки, полегшуючи модифікацію, тестування та масштабування програмного забезпечення.

Концепція структур даних. Структури даних і алгоритми є основними компонентами програм, а також самих комп'ютерних систем. Внутрішні структури даних представлені регістрами та словами пам’яті, в яких зберігаються бінарні величини. Алгоритми, що закладені в апаратну частину, реалізують жорсткі правила, за якими дані інтерпретуються як команди для виконання. Вся діяльність комп’ютера зосереджується на обробці бітів, а вся інформація обробляється згідно з незмінним набором алгоритмів, які визначаються архітектурою центрального процесора.

Код легше підтримувати та розширювати, надаючи доступ лише через визначений інтерфейс. Оптимального структурування програм можна досягти при комбінованому використанні висхідного та низхідного проектування. Такий підхід дозволяє адаптувати програму до вимог проекту, що є основою сучасних методів розробки програмного забезпечення. Завдяки цьому підвищенню надійності та гнучкості легше модифікувати, тестувати та масштабувати програмне забезпечення.

Поняття структур даних. Основними компонентами програм є структури даних. Структури даних представлені регістром і словами пам'яті. Дані інтерпретуються як команди за правилами, вбудованими в апаратне забезпечення. Уся діяльність комп’ютера зосереджена навколо обробки бітів, і вся інформація обробляється відповідно до незмінного набору алгоритмів.

Проблеми, які розв'язує комп'ютер, зазвичай представлені у вигляді чисел, символів, текстів і більш складних структур, таких як списки або дерева. Структура даних — це концепція, яка показує організацію інформації. Рецептом обробки даних служить алгоритм, який виступає процедурним елементом.

Структури даних можуть бути дуже різноманітними і складними. Ключ до успішного впровадження програми полягає у виборі оптимального представлення даних і може вплинути на продуктивність більше, ніж деталі реалізації. Залежно від ситуації, до цього питання необхідно підходити творчо, оскільки універсальної теорії вибору структур даних не існує.

Дані в пам'яті комп'ютера представлені у вигляді послідовності бітів. Різні структури даних є «будівельними блоками» для різних типів інформації.

Фізична структура даних відображає те, як вони представлені в пам'яті комп'ютера, тоді як абстрактна структура зосереджена на логіці та характеристиках цих даних Для адекватного розуміння структур даних необхідно розглядати їх на різних рівнях: функціональному, логічному та фізичному. Функціональна специфікація визначає допустимі операції та їх властивості, логічний опис показує, як об'єкти поділяються на елементарні, а фізичне представлення показує, як ці дані розміщуються в пам'яті.

Різні логічні описи можуть відповідати одній функціональній специфікації та можуть бути реалізовані через різні фізичні структури. Важливо, що кожен рівень декомпозиції відображає попередній, що сприяє кращому розумінню та ефективній реалізації структур даних у програмуванні.

Існує класифікація структур даних. У структурі даних є елементи даних і зв’язки. Можна класифікувати прості та інтегровані структури даних.

Прості структури даних не можна розділити на більші частини. Розмір простого типу та його організацію в пам'яті можна визначити з фізичної точки зору. Прості дані можна розглядати як неподільні одиниці.

Прості та інтегровані структури даних можна знайти в інтегрованих структурах даних. Програміст використовує інструменти мов програмування для об’єднання даних.

Наявність або відсутність чітко визначених зв'язків між елементами є важливим критерієм класифікації структур даних. Розрізняють зв’язані та незв’язані конструкції.

Існує можливість зміни кількості елементів і зв'язків у структурі даних. Структури можуть бути статичними з фіксованою кількістю елементів, напівстатичними з обмеженими можливостями зміни або динамічними, коли кількість елементів може вільно змінюватися.

Структури даних можна класифікувати на основі природи їх упорядкування. За цим критерієм розрізняють два типи структур: лінійні структури, де елементи розташовані в послідовному порядку, і нелінійні структури, які можуть мати складніші зв’язки між елементами.

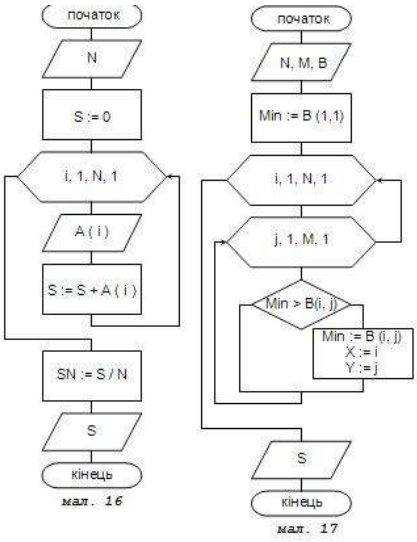
Є операції над структурами даних. Є чотири основні операції, які можна виконати над усіма структурами даних.

1.2 Стистичні структури даних

Статичні структури даних відносяться до класу структур, які складаються з певної кількості базових примітивних елементів, упорядкованих певним чином.

Важливою характеристикою цих структур є те, що їхній розмір залишається незмінним протягом усього часу існування, що дозволяє заздалегідь визначити необхідний обсяг пам'яті. Таке виділення пам'яті може здійснюватися автоматично, наприклад, на етапі компіляції або під час виконання програми, коли активується відповідний блок коду. Хоча деякі мови програмування дозволяють програмістам динамічно виділяти пам'ять для статичних структур під час виконання програми, навіть у цьому випадку обсяг виділеної пам'яті залишається незмінним до моменту знищення структури.

Однією з основних переваг виділення пам'яті на етапі компіляції є зручність і передбачуваність, що дозволяє програмістам використовувати статичні структури навіть для об'єктів, розмір яких може варіюватися. Наприклад, якщо розмір масиву наперед невідомий, для нього можна зарезервувати максимально можливий обсяг пам'яті, що гарантує безпеку при доступі до елементів цього масиву.



Малюнок 1.2 – Приклад масивів

У багатьох мовах програмування статичні структури даних реалізуються через структуровані типи, які дозволяють створювати дані будь-якої складності. До таких типів належать масиви, записи та їхні похідні структури. Масиви являють собою логічну структуру, в якій елементи одного типу об'єднані в впорядковану послідовність. Масиви можуть бути класифіковані за кількістю вимірів: одновимірні масиви (вектори), двовимірні (матриці) та багатовимірні масиви (три, чотири і більше вимірів).

Основні характеристики масиву, як логічної структури даних, включають:

- Фіксований набір елементів одного типу.
- Унікальні значення індексів для кожного елемента.
- Кількість індексів визначає вимірність масиву.
- Доступ до елементів здійснюється через ім'я масиву та індекси.

Фізична структура масиву визначає, як елементи масиву розміщуються в пам'яті комп'ютера. Кожен елемент займає певний обсяг пам'яті, що відповідає його базовому типу. Розмір масиву в пам'яті визначається кількістю елементів та розміром кожного елемента. Основною операцією на фізичному рівні є доступ до елементів масиву. Після того як доступ отримано, можна виконати будь-яку операцію, яка підтримується типом даних елемента. Лінеаризація — це процес перетворення багатовимірного масиву в одновимірну структуру для зручності зберігання та доступу.

Адреса масиву — це адреса першого елемента, що розміщується в пам'яті. Наприклад, у мовах C/C++ індексація масивів завжди починається з нуля. До операцій на логічному рівні масивів відносяться сортування, пошук елементів та інші маніпуляції з даними масиву.

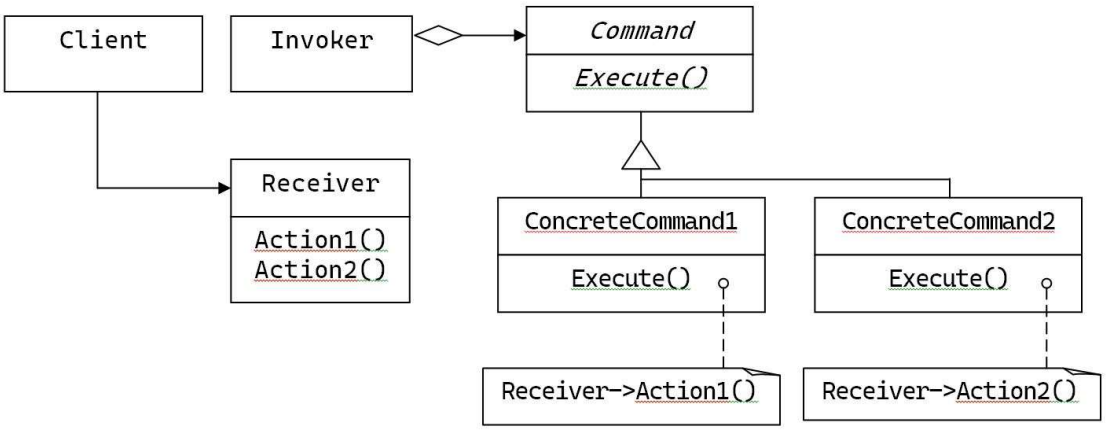
У реальних сценаріях роботи з великими масивами можуть виникати ситуації, коли багато елементів масиву займають пам'ять частково. Розріджені масиви, що мають значну кількість елементів з однаковими значеннями (наприклад, фонового значення), дозволяють значно економити пам'ять. У таких масивах зберігається лише мінімальна кількість елементів, що відрізняються від фонового значення. Це дозволяє ефективно використовувати пам'ять, зберігаючи тільки значущі елементи.

Існують два основні типи розріджених масивів:

1. Масиви з математично передбачуваним розташуванням ненульових елементів — елементи масиву розташовані за певною закономірністю, і для їх зберігання використовується одновимірний масив з індексами, що розраховуються за спеціальними формулами.

2. Масиви з випадковим розташуванням ненульових елементів— елементи не мають передбачуваного порядку, і для їх зберігання використовуються методи, які дозволяють зменшити обсяг пам'яті, забезпечуючи ефективне використання ресурсів.

У разі використання таких методів, зокрема для великих матриць, пам'ять може бути значно зекономлена, а операції з масивами можуть бути виконані значно швидше. Проте, зміни, такі як додавання або видалення елементів, можуть вимагати перестановки значної кількості елементів, що робить такі методи менш ефективними для часто змінюваних структур.



Малюнок 1.3 – Множини

Що стосується типів даних, то множини — це колекції елементів, де кожен елемент унікальний і порядок його розташування не має значення. Множини не є стандартним типом даних в C/C++, але вони широко використовуються у програмуванні через спеціальні реалізації. Основні операції, що виконуються над множинами, включають об'єднання, перетин, різницю та симетричну різницю між множинами. Множина може бути порожньою, і тоді її називають нульовою.

Структури даних відрізняються від масивів і множин тим, що можуть містити елементи різних типів. Елементи структури називаються полями і можуть бути різних типів, але всі вони об'єднані в одну структуру. Операції на структурах включають доступ до їх полів і виконання операцій для кожного з полів. Об'єднання є різновидом структури, де всі поля займають одну і ту ж адресу в пам'яті. Це дозволяє заощаджувати пам'ять, коли необхідно зберігати тільки одне значення з набору полів.

Іншим важливим елементом є бітові типи даних, які широко використовуються в системному програмуванні. Вони дозволяють працювати з окремими бітами даних, упакованими в байти чи слова, і використовуються для оптимізації пам'яті та роботи з флагами або прапорцями. Таблиці, у свою чергу, є структурами, що містять елементи, до яких можна звертатися за ключем, замість традиційної індексації. Ключ дозволяє швидко знаходити необхідну інформацію серед багатьох елементів, що робить таблиці ефективним інструментом для зберігання та обробки даних.

1.3 Напівстатистині структури даних

Напівстатичні структури даних характеризуються низкою важливих особливостей, серед яких можна виділити такі основні:

1. Змінна довжина: Одна з ключових характеристик напівстатичних структур полягає в їхній здатності змінювати довжину залежно від потреби. Це означає, що елементи можуть бути додані або вилучені в будь-який час, що дозволяє зручно працювати з даними.

2. Обмеження на довжину: Хоча структура є змінною, її довжина обмежена певними межами, встановленими на етапі проектування або виконання програми. Вона не може перевищувати задане максимальне значення, що забезпечує контроль за використанням пам'яті.

Логічно напівстатична структура може бути представлена як послідовність даних, елементи якої пов'язані між собою певними лінійними відносинами. Оскільки доступ до елементів здійснюється через їхній порядковий номер, ці структури дозволяють ефективно маніпулювати даними.

Фізичне представлення таких структур у пам'яті найчастіше має вигляд послідовності комірок, де кожен наступний елемент розміщується безпосередньо після попереднього. Іншим варіантом є використання однонаправленого зв'язного списку, де кожен елемент містить покажчик на наступний. У такому разі обмеження на довжину структури стають менш строгими, оскільки можна динамічно змінювати кількість елементів без необхідності переміщення даних.

Стек є одним з основних типів напівстатичних структур даних, який працює за принципом LIFO (Last In, First Out — „останнім прийшов — першим вийшов”). Це структура, де елементи додаються і вилучаються лише з одного кінця, що називається вершиною стеку. Решта елементів складають тіло стеку.

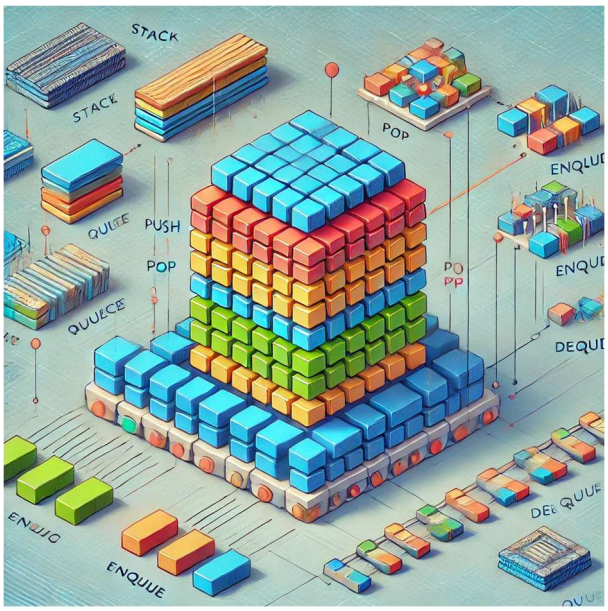
Основні операції над стеком включають:

- Push (додавання елемента на вершину).
- Pop (видалення елемента з вершини).

Допоміжні операції можуть включати:

- Перевірку, чи стек порожній.
- Перегляд останнього доданого елемента без вилучення.
- Підрахунок кількості елементів у стеку.

Стек може бути представленим в пам'яті як масив або як зв'язаний список. У випадку масиву, покажчик на вершину змінюється, коли елементи додаються або видаляються. У зв'язному представленні кожен елемент містить дані і покажчик на попередній елемент, що дозволяє зберігати структуру навіть при зміні розміру.



Малюнок 1.4 – Напівстатистині структури даних

Черга є ще одним важливим типом змінної структури даних, що працює за принципом FIFO (First In, First Out — „першим прийшов — першим вийшов”). У черзі елементи додаються в один кінець (хвіст) і вилучаються з іншого (голова).

Основні операції над чергою включають:

- Enqueue (додавання елемента).
- Dequeue (видалення елемента).
- Перевірку на порожнечу.
- Очищення черги.

Черга може бути організована двома способами: статично (з використанням двох покажчиків на голову і хвіст) або динамічно, використовуючи кільцеву структуру для уникнення витрат пам'яті.

Дек (двостороння черга, або **double-ended queue**) дозволяє виконувати операції додавання та видалення елементів з обох кінців. Це дозволяє реалізувати структури, де елементи можуть бути як додані з початку, так і з кінця черги. Операції над деком включають:

- Додавання елемента зліва або справа.
- Видалення елемента зліва або справа.

Фізично дек можна представити як кільцеву чергу або як зв'язаний список, де покажчики на лівий і правий кінець дозволяють ефективно маніпулювати елементами.

Лінійні списки — це узагальнення попередніх структур, що дозволяють ефективно працювати з даними, не порушуючи їх послідовності. Лінійні списки

можуть бути представлені як масиви або як зв'язні структури, де кожен елемент містить покажчик на наступний.

Основні операції з лінійними списками:

- Вставка елемента в середину списку.
- Локалізація елемента за його значенням.
- Видалення елемента.

Особливістю лінійних списків є їх здатність до динамічного розширення без потреби переміщення інших елементів списку, що робить їх надзвичайно зручними для різноманітних задач у програмуванні.

Дерево є ієрархічною структурою даних, що складається з вузлів, кожен з яких може мати кілька дочірніх елементів. Дерева широко використовуються для організації даних у вигляді каталогів, баз даних та систем пошуку.

Граф — це абстрактна структура, що складається з набору вершин (вузлів) та ребер, що з'єднують ці вершини. Графи є основою для моделей соціальних мереж, маршрутизації в комп'ютерних мережах та багатьох інших застосувань.

У реальному використанні дерева та графи служать основними структурами для представлення даних з ієрархічними або складними зв'язками між елементами.

1.4 Динамічні структури даних

Динамічні структури даних відрізняються від статичних тим, що елементи таких структур не зберігаються обов'язково в безпосередньо суміжних областях пам'яті. Це дозволяє структурам бути більш гнучкими, оскільки їх розмір може змінюватися під час виконання програми, що дає змогу додавати або видаляти елементи в будь-який момент. Замість того, щоб елементи були фізично розміщені у визначених межах пам'яті, їх розташування є довільним. Як наслідок, адреса кожного елемента не може бути визначена за допомогою простих обчислень, що засновані на розташуванні інших елементів або на відносному порядку їхнього розміщення.

Основною особливістю таких структур є використання покажчиків, які створюють явні зв'язки між елементами. Це дозволяє забезпечити доступ до елементів, навіть якщо вони розташовані в випадкових областях пам'яті. Структури, побудовані таким чином, називаються зв'язними, оскільки кожен елемент містить посилання на інші елементи, що забезпечує зв'язок між ними.

Кожен елемент такої структури має два основних компоненти. Перший — це інформаційне поле, яке містить власне дані, що зберігаються в структурі, зокрема значення, для яких структура була створена. Другий компонент — це поле зв'язку,

яке включає один або кілька покажчиків, що з'єднують поточний елемент з іншими елементами структури. Поле зв'язку не взаємодіє безпосередньо з користувачем, адже користувач зазвичай працює лише з даними, що зберігаються в інформаційному полі.

Малюнок 1.5 – Динамічні структури даних

Зв'язне представлення даних має кілька значних переваг, головною з яких є його гнучкість. Оскільки розмір структури може змінюватися під час виконання програми, це дозволяє легко додавати або видаляти елементи без необхідності переміщувати інші дані в пам'яті. Крім того, структура обмежена лише доступним обсягом пам'яті, а не фіксованими розмірами, що робить її більш адаптивною до змінних вимог програми.

Також, зв'язне представлення дозволяє легко змінювати логічну послідовність елементів, оскільки для цього достатньо оновити покажчики, без необхідності переміщати фізичні дані в пам'яті. Однак ця перевага також має свої недоліки. Зокрема, робота з покажчиками потребує більшої уважності від програмістів, оскільки неправильне управління покажчиками може призвести до помилок, таких як порушення зв'язку між елементами або втрати даних.

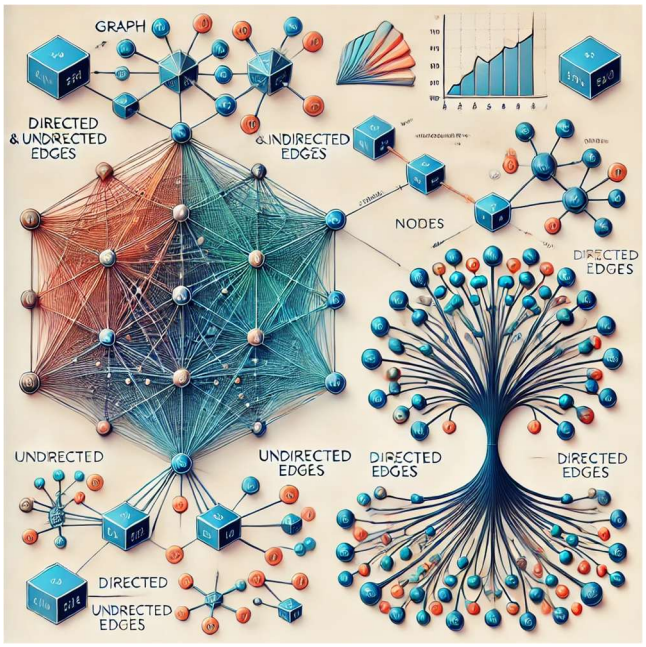
Інший недолік зв'язного представлення — це витрати пам'яті, оскільки для кожного елемента потрібно додаткове місце для зберігання покажчиків. Крім того, доступ до елементів може бути менш ефективним, оскільки для того, щоб отримати доступ до конкретного елемента, необхідно пройти через ланцюг покажчиків, що може зайняти більше часу порівняно з іншими методами, де доступ до елементів здійснюється за допомогою індексів.

Зв'язне представлення є дуже корисним у випадках, коли логічна структура даних потребує специфічних операцій доступу або маніпуляцій, таких як у списках,

деревах або таблицях. Водночас, якщо структура даних передбачає послідовний доступ за індексами, як у масивах або векторах, зв'язне представлення використовувати не так доцільно, оскільки індексований доступ є більш ефективним для таких задач.

Граф — це складна і нелінійна структура даних, що використовується для моделювання зв'язків між елементами. Його головною характеристикою є те, що елементи (вузли) можуть бути пов'язані між собою через різні типи зв'язків (ребра), які можуть мати різну природу. Відмінною рисою графа є можливість наявності будь-якої кількості зв'язків між елементами. Такі зв'язки можуть бути спрямованими або неспрямованими, а також можуть містити додаткову інформацію, таку як вага, яка вказує на величину або важливість певного зв'язку.

У графах кожен елемент, або вузол, містить інформацію про сам елемент, а зв'язки між ними, що є ребрами, визначають структуру графа. Графи можуть бути різними за своїми властивостями. Вони можуть бути орієнтованими, коли кожен зв'язок має певний напрямок, наприклад, від одного елемента до іншого. Існують також неорієнтовані графи, де зв'язки не мають напрямку, що означає, що зв'язок між двома елементами є двостороннім. Окрім того, можуть існувати змішані графи, які поєднують як орієнтовані, так і неорієнтовані зв'язки.



Малюнок 1.6 – Графи

Графи використовуються для зберігання та обробки складної інформації, де елементи можуть бути взаємозалежними. Наприклад, вони широко застосовуються в базах даних, системах штучного інтелекту, в алгоритмах пошуку та

маршрутизації, а також у програмному моделюванні складних систем, де потрібно описувати різні види зв'язків і відносин.

Що стосується представлення графів у пам'яті комп'ютера, то для цього використовуються два основних підходи: матричне представлення та представлення через зв'язні списки. Перший метод зазвичай застосовується, коли структура графа є стабільною і зміни в графі відбуваються рідко. Це дозволяє створювати матрицю суміжності, де для кожного пари вузлів вказано наявність або відсутність зв'язку між ними. У разі зважених графів, ця матриця містить не лише одиниці та нулі, а й ваги зв'язків, що дозволяє враховувати важливість або відстань між вузлами. Зв'язкове представлення є більш гнучким і підходить для графів, що часто змінюються, оскільки дозволяє динамічно додавати або видаляти елементи без необхідності змінювати всю структуру.

Дерево є специфічним типом графа з особливими властивостями. Воно складається з вузлів, де один з них є коренем, а всі інші мають лише одного батьківського елемента, з якого вони походять. Особливістю дерева є те, що від кореня можна досягти будь-якого іншого елемента, і кожен вузол, крім кореня, має лише одного батька. Зв'язки між вузлами дерева називаються гілками, а вузли без нащадків називаються листям. Вузли, які мають нащадків, є проміжними.

Дерева часто представляються в пам'яті за допомогою зв'язних списків, оскільки це дозволяє ефективно відображати ієрархічну структуру. Хоча можливе й використання масивів для представлення дерев, зв'язні списки краще відображають їхню природну структуру. Основні операції, що виконуються над деревами, включають пошук вузлів, додавання нових елементів, видалення вузлів та різні типи обробки дерева через обхід.

Обхід дерева — це процес проходження всіх його вузлів у певному порядку. Існують різні стратегії обходу дерева, серед яких найпоширенішими є низхідний, змішаний та висхідний обхід. Низхідний обхід полягає в тому, що спочатку обробляється корінь дерева, а потім по черзі обробляються ліве та праве піддерева. У разі змішаного обходу спочатку здійснюється перехід до лівого піддерева, після чого обробляється вузол, і переходять до правого сина. Висхідний обхід передбачає спочатку обробку лівого піддерева, потім правого, а лише на завершення — обробку кореня.

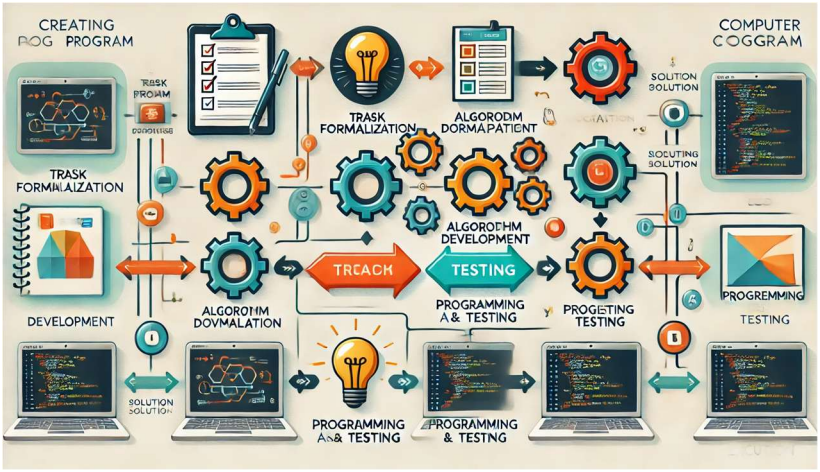
Ці операції та різні методи обходу дерева дозволяють ефективно працювати з даними, організованими в ієрархічну структуру, забезпечуючи швидкий доступ до необхідної інформації та можливість гнучко змінювати структуру дерева під час виконання програм.

1.5 Побудова і аналіз алгоритмів

Процес створення комп'ютерної програми для вирішення практичних завдань можна поділити на кілька важливих етапів. Спочатку необхідно розробити формалізацію задачі, що включає чітке визначення технічного завдання. Це дає змогу зрозуміти, які вимоги ставляться до майбутнього розв'язку і яким чином алгоритм повинен взаємодіяти з даними. Після цього розробляється сам алгоритм вирішення задачі — набір чітко визначених кроків, який веде до досягнення бажаного результату. Наступним етапом є написання програми, її тестування, налагодження та документування, що забезпечує коректну реалізацію алгоритму на практиці. Завершальним етапом є отримання рішення початкової задачі, яке досягається шляхом виконання програми.

Процес розв'язання задачі не завжди буває таким простим. На самому початку багато практичних задач не мають чіткого або повного формулювання, і це ускладнює їх комп'ютерне вирішення. Іноді деякі завдання взагалі не можуть бути виражені в термінах, які дозволяють застосувати комп'ютерне рішення. Навіть у випадку, якщо завдання можна вирішити, для його формалізації часто необхідно врахувати багато параметрів, які потребують додаткових експериментів для визначення своїх допустимих меж.

Коли певні аспекти задачі можна виразити через формальну модель, це має сенс зробити, оскільки модель допоможе визначити, чи існують методи і алгоритми для її вирішення. Якщо таких методів наразі немає, формалізація допоможе в подальшому процесі їх розробки. Багато галузей математики і наук можуть бути використані для моделювання конкретних класів задач. Наприклад, для числових завдань часто використовуються системи лінійних рівнянь або диференціальних рівнянь, тоді як для задач із текстовими або символьними даними можуть застосовуватися формальні граматики або символьні послідовності. Розв'язання таких задач зазвичай передбачає етапи компіляції та інформаційного пошуку.



Малюнок 1.7 – Алгоритми

Коли необхідна модель задачі створена, наступним кроком є пошук розв'язку в межах цієї моделі. Це передбачає розробку алгоритму, що складається з чітко визначених інструкцій, які повинні бути виконані за обмежений час з мінімальними витратами на обчислення. Алгоритм повинен завершуватися після виконання обмеженої кількості кроків, що гарантує його скінченність навіть при різних вхідних даних. Важливо також, щоб кожна інструкція в алгоритмі мала чітке і зрозуміле значення, а її виконання вимагало кінцевих обчислювальних затрат.

Алгоритм є центральною частиною програмування, і його розробка потребує детального проектування. Першим кроком є створення математичної моделі задачі, що дозволяє абстрагуватися від конкретних деталей і зосередитися на вирішенні проблеми на більш високому рівні. Потім формується неформальний алгоритм, який містить загальні інструкції для вирішення задачі. Однак ці інструкції часто потребують подальшої деталізації для перетворення їх у конкретний код комп'ютерної програми.

Поступово цей алгоритм переходить до етапу, коли він записується на псевдомові, що являє собою суміш загальних конструкцій і синтаксису конкретної мови програмування. Псевдомова повинна бути достатньо чіткою, щоб дозволити правильно визначити типи даних і операції над ними. Після цього створюються абстрактні типи даних, для кожного з яких призначаються функції для виконання необхідних операцій.

На наступному етапі відбувається програмування, що полягає в реалізації абстрактних типів даних через конкретні функції і заміни неформальних операторів псевдомови на реальний код. Це дозволяє отримати виконувану програму. Важливо, що алгоритм повинен забезпечувати правильність і завершеність розв'язку, а також оптимальність у використанні ресурсів. Головними вимогами до алгоритму є:

1. Завершеність у часі — алгоритм повинен закінчувати свою роботу за обмежений час.

2. Правильність — алгоритм повинен давати правильне рішення.

3. Детермінованість — однакові вхідні дані повинні завжди приводити до однакового результату.

4. Скінченність — кількість кроків алгоритму має бути обмеженою.

5. Однозначність — кожен крок алгоритму повинен мати чітке тлумачення.

Незважаючи на це, існують алгоритми, що не гарантують точного розв'язку, — це так звані евристичні алгоритми. Вони протилежні класичним, оскільки не завжди можуть привести до правильного результату. Існують також проміжні типи алгоритмів, такі як наближені та ймовірнісні, що можуть давати оптимальне рішення в деяких випадках.

Вибір правильного алгоритму часто буває складним, оскільки існує необхідність задоволення декількох вимог одночасно. З одного боку, алгоритм повинен бути простим для розуміння і переведення в програмний код, з іншого — він має бути ефективним у використанні ресурсів комп'ютера і швидким у виконанні. Якщо програму потрібно виконати лише кілька разів, важливіший перший аспект. Однак якщо задача вимагає значних обчислювальних затрат, то економічно вигідніше використовувати більш складний алгоритм.

Для оцінки складності алгоритмів існують різні підходи. Одним із найбільш популярних є вимірювання часу виконання програми на конкретному комп'ютері з визначеним набором вхідних даних. Однак це не завжди дає точну картину, оскільки час виконання залежить від різних факторів, таких як архітектура комп'ютера або ефективність компілятора. Тому важливу роль відіграє асимптотична складність, яка оцінює ефективність алгоритму залежно від розміру вхідних даних.

Іноді для оцінки алгоритмів використовуються поняття просторово-часової складності, де оцінка проводиться за двома критеріями: часом виконання та пам'яттю. Часова ефективність зазвичай оцінюється в найгіршому випадку, коли алгоритм виконує найгірше можливе обчислення для певного розміру вхідних даних.

Таким чином, вибір і розробка алгоритму для вирішення конкретної задачі є складним процесом, що вимагає врахування безлічі факторів, зокрема швидкості виконання, використання пам'яті та вартості виконання програми.

1.6 Алгоритми і сортування

Задача сортування полягає в упорядкуванні множини елементів, які спочатку можуть бути неструктурованими, у впорядковану послідовність за певним критерієм — чи то за зростанням, чи за спаданням. Це одна з основних задач в області програмування, і для її вирішення існує величезна кількість різних алгоритмів. Однак, вибір оптимального алгоритму сортування залежить від ряду чинників, кожен з яких має важливе значення на етапі розробки програмного забезпечення.

Перш за все, необхідно враховувати наявні ресурси пам'яті. Якщо вхідна та вихідна множини мають бути розташовані в різних ділянках пам'яті, це вимагає додаткових ресурсів для виділення пам'яті. Водночас, якщо одна множина формується на місці іншої, то може виникнути потреба в динамічному перерозподілі пам'яті. Це особливо важливо для великих обсягів даних, де недостатність пам'яті може призвести до зниження продуктивності програми.

Іншим важливим аспектом є початкова впорядкованість вхідних даних. Якщо множина частково впорядкована, деякі алгоритми можуть працювати значно швидше, ніж інші. Наприклад, для вже відсортованих або майже відсортованих даних деякі алгоритми демонструють дуже хорошу ефективність. З іншого боку, деякі алгоритми не враховують цього факту і мають однаковий час виконання для будь-яких вхідних множин.

Часові характеристики операцій також є важливими. Вони, зазвичай, пропорційні кількості порівнянь, що здійснюються між елементами. Наприклад, порівняння числових значень зазвичай виконується значно швидше, ніж порівняння рядкових значень, що може впливати на ефективність алгоритму. Крім того, час, необхідний для пересилання елементів, залежить від обсягу даних, що переміщуються, що може істотно вплинути на загальний час виконання.

Не менш важливим є аспект складності алгоритму, що пов'язаний з його реалізацією. Простота реалізації може значно знизити ймовірність помилок у програмному коді, що є важливим, особливо при розробці програм у стислі терміни або за умов високих вимог до надійності програмного забезпечення.

Алгоритми сортування оцінюються за двома основними критеріями: кількістю порівнянь елементів і кількістю пересилань елементів. Алгоритм сортування також може бути названий "усталеним", якщо він зберігає відносний порядок однакових елементів у вихідному масиві.

Існує кілька стратегій, які застосовуються в алгоритмах сортування. Одна з них — стратегія вибірки, яка передбачає вибір наступного елемента з вхідної множини і додавання його до вихідної послідовності. Інша стратегія — це стратегія

включення, яка полягає в виборі наступного елемента та його вставці на відповідну позицію в уже відсортованій частині множини. Існує також стратегія розподілу, що передбачає розбиття вхідної множини на підмножини і сортування кожної з них окремо. І, нарешті, стратегія злиття передбачає створення вихідної множини шляхом злиття вже впорядкованих підмножин.

Розглянемо деякі конкретні алгоритми сортування. Алгоритм сортування вибіркою використовує стратегію вибірки. Проблеми можуть виникати під час його реалізації, особливо при обробці "порожніх" значень. У таких випадках для полегшення роботи можуть бути використані різні підходи, наприклад, введення спеціальних значень для позначення відсутніх елементів. Загальна ідея алгоритму — пошук мінімального елемента в масиві і його обмін з першим елементом, після чого процес повторюється для залишкової частини масиву.

Малюнок 1.8 – Алгоритми сортування: їхня складність і вибір підходящого

Сортування бульбашкою — ще один поширений метод сортування, при якому елементи порівнюються попарно і міняються місцями, якщо їх порядок не відповідає критерію впорядкованості. Цей процес повторюється до того часу, поки весь масив не буде впорядкований. Сортування бульбашкою має складність $O(N^2)$, що робить його не надто ефективним для великих обсягів даних, хоча різні модифікації, такі як шейкерсортування, можуть зменшити кількість проходів, здійснюючи їх у обох напрямках.

Алгоритм сортування включенням полягає в тому, що кожен елемент вхідного масиву вставляється на правильне місце в уже відсортовану частину масиву. Він має аналогічну складність $O(N^2)$, але потребує більших затрат на пересилання елементів, що робить його менш ефективним, ніж сортування

вибіркою. Використання дихотомічного пошуку може дещо покращити його ефективність, зменшуючи кількість порівнянь у разі великих масивів.

Турнірне сортування використовує принцип змагань, де елементи порівнюються і "піднімаються" через дерево, що побудоване для сортування. Цей метод дозволяє досягти складності $O(N*\log_2(N))$, що робить його ефективнішим для великих наборів даних, особливо якщо початкова впорядкованість є частковою.

Таким чином, вибір конкретного алгоритму сортування залежить від різноманітних факторів, зокрема, від розміру та структури вхідних даних, доступної пам'яті, часу виконання та специфічних вимог до надійності та простоти реалізації.

1.7 Алгоритми пошуку

Пошук є однією з основних і найпоширеніших операцій у програмуванні, оскільки в багатьох випадках потрібно знайти певний елемент серед великої кількості даних. Завдання пошуку полягає в тому, щоб знайти елемент, чий ключ збігається з заданим «аргументом пошуку». Після цього отриманий індекс дозволяє отримати доступ до всіх полів цього елемента.

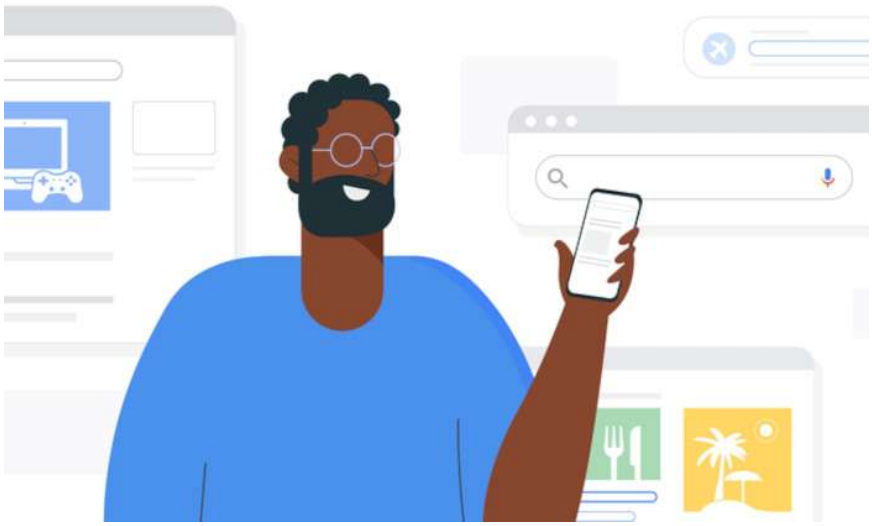
Одним із найпростіших підходів до пошуку є послідовний, або лінійний, пошук, який використовується в основному для роботи з невпорядкованими наборами даних. Метод полягає в тому, щоб пройти через кожен елемент набору один за одним, поки не буде знайдений шуканий елемент. Якщо весь набір було переглянуто і елемент не знайдений, це свідчить про те, що елемент відсутній. Таке рішення також відоме як метод повного перебору. За статистикою, середня кількість порівнянь для цього методу становить приблизно N/2, що робить час виконання алгоритму лінійним, тобто O(N). При цьому важливо, що якщо елемент знайдений, то він буде виявлений з найменшим можливим індексом, тобто буде повернуто перше знайдене значення. Якщо ж кількість ітерацій дорівнює N, це означає, що елемент не існує в наборі. Модифікацією лінійного пошуку є використання так званого «бар'єра»: наприкінці масиву додається один додатковий елемент, який дорівнює шуканому ключу. Це дозволяє уникнути додаткової перевірки умови циклу, що може спростити реалізацію алгоритму.

Існують і більш ефективні методи пошуку, наприклад, бінарний (дихотомічний) пошук, який значно пришвидшується, якщо дані впорядковані. В основі цього методу лежить ідея того, що елемент шукається не в середині масиву, а шляхом поділу його на дві частини. Спочатку перевіряється середній елемент

масиву, і залежно від того, чи більше або менше його значення за шукане, пошук продовжується у відповідній частині масиву — ліворуч чи праворуч. Оскільки на кожному кроці кількість елементів, що перевіряються, зменшується вдвічі, то для пошуку потрібного елемента в найгіршому випадку буде потрібно лише $\log_2(N)$ порівнянь, що значно зменшує час виконання алгоритму в порівнянні з лінійним пошуком.

Іншим методом є метод інтерполяції, який використовує припущення про те, що значення елементів зростають від початку до кінця масиву більш-менш рівномірно. У цьому випадку не потрібно перевіряти середній елемент, а обчислюється позиція елемента на основі пропорції між мінімальним і максимальним значеннями. Якщо таке припущення є справедливим, метод інтерполяції може працювати навіть швидше за бінарний пошук, оскільки він дозволяє здійснювати пошук на основі більш точної інформації про розподіл даних.

У випадках, коли йдеться про пошук послідовностей у тексті, наприклад, для пошуку слів у рядках, також існують спеціалізовані алгоритми. Один із найпростіших методів — це послідовне порівняння кожного символу в рядку з символами масиву. Якщо символи співпадають, порівнюються наступні символи, і так до повного збігу слова з відповідною частиною тексту. Пошук продовжується до наступного символу в тексті, якщо збіг не відбувся.



Малюнок 1.9 – Алгоритми пошуку

Проте, існують більш ефективні методи для пошуку в текстах. Наприклад, алгоритм Кнута-Морріса-Пратта (КМП), який значно підвищує швидкість пошуку завдяки тому, що після часткового збігу початкової частини слова з відповідними символами тексту, можна здійснити зсув слова на кілька позицій, а не лише на

одну. КМП використовує таблицю зсувів, що дозволяє визначити, на скільки символів потрібно пересунути слово після кожного невдалого збігу. Ці зсуви залежать лише від самого слова, а не від тексту, що дозволяє швидко зреагувати на невдачі і уникати зайвих порівнянь.

Ще одним вдосконаленням є алгоритм Боуєра-Мура, який порівнює символи починаючи з кінця слова, а не з початку, що дозволяє зменшити кількість порівнянь. Як і в КМП, перед пошуком будується таблиця зсувів, але в цьому випадку таблиця залежить від позицій найбільш правих входжень символів у слові. При збігу символів слово може бути зсунуте значно більше, ніж на один символ, що дозволяє швидше рухатися по тексту. Алгоритм Боуєра-Мура також працює ефективно навіть у випадку, коли символ, що не збігається, взагалі відсутній у слові.

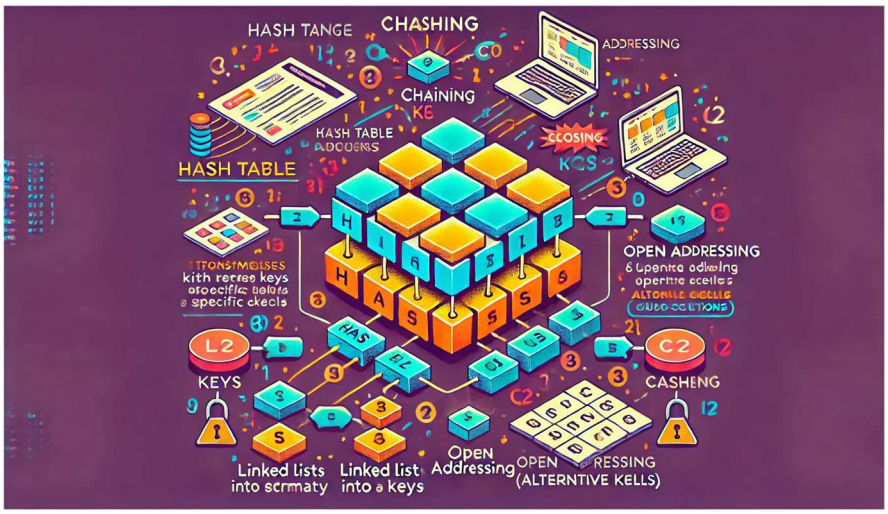
Методи пошуку послідовностей можуть бути модифіковані таким чином, щоб пошук не йшов до кінця кожного рядка, а зупинявся на певному обмеженні, якщо слово не може бути розташоване в кінці одного рядка і на початку наступного. Це може бути корисним для більш швидкої обробки великих текстових масивів, де багато рядків.

1.8 Методи швидкого доступу до даних

Хешування є важливою технікою для швидкого доступу до даних, яку активно використовують для організації інформації в таблицях, побудованих на основі значень ключів. Принцип роботи хешування полягає в тому, що за допомогою хеш-функції, яка обчислює адресу для кожного ключа, можна значно зменшити час, необхідний для пошуку потрібних даних. Ідеальним варіантом є така хеш-функція, яка для кожного унікального ключа генерує унікальну адресу, забезпечуючи ефективність доступу. У випадку, коли набір ключів відомий і обмежений, це можливо досягти за допомогою так званого «досконалого хешування».

Уявімо ситуацію, коли ми маємо набір записів із унікальними ключами від 1 до 100. Створюється масив з 100 осередків, в який записуються дані таким чином, що кожен запис потрапляє у відповідну осередок за своїм ключем. Наприклад, запис із ключем 37 буде поміщений у 37-у позицію масиву. Завдяки такій організації даних можна швидко виконувати операції додавання, пошуку та вилучення елементів. Це дозволяє значно пришвидшити обробку даних у порівнянні з іншими методами пошуку.

Однак створення досконалого хешування для невизначеного набору значень є складним завданням. У таких випадках часто використовуються хеш-функції, які не гарантують унікальність адрес. Внаслідок цього можуть виникати колізії — ситуації, коли кілька ключів мають однакові адреси в хеш-таблиці. Це є нормальним явищем, і для вирішення проблеми колізій необхідні спеціальні алгоритми.



Малюнок 1.10 – Швидкий доступ

Алгоритми для вирішення колізій повинні бути частиною схеми хешування і мати здатність знаходити альтернативні осередки таблиці, якщо обчислена адреса вже зайнята. Такі алгоритми працюють шляхом пошуку вільної комірки в таблиці, використовуючи певну послідовність перевірок, яка визначається так званою тестовою послідовністю.

Для реалізації процесу хешування необхідні три основні елементи: хеш-таблиця, яка слугує для зберігання даних; хеш-функція, яка здійснює відповідність між значеннями ключів і адресами в таблиці; і алгоритм для вирішення колізій, який визначає дії в разі виникнення конфліктів між ключами.

Існують два основних підходи для вирішення колізій: метод ланцюжків і метод відкритої адресації.

У методі ланцюжків для кожної комірки хеш-таблиці створюється список, в якому зберігаються всі елементи, що потрапили до цієї комірки через колізію. Коли відбувається колізія, новий елемент додається до цього списку. Пошук і вилучення елементів здійснюються шляхом перебору цього списку. Перевагою цього методу є те, що хеш-таблиці не переповнюються, а також спрощується вставка і видалення елементів. Проте, якщо списки стають дуже довгими, час доступу до елементів може збільшитися.

Метод відкритої адресації полягає в тому, що в разі колізії проглядаються інші осередки таблиці до тих пір, поки не буде знайдена вільна комірка. Існує кілька варіантів цього методу, зокрема лінійне випробування, при якому перевірка здійснюється по черзі за фіксованим кроком, квадратичне випробування, де крок залежить від номеру спроби, і подвійне хешування, яке використовує додаткову хеш-функцію для обчислення кроку. Перевагою методу є його простота, проте він може призвести до циклічних переходів, коли комірки таблиці закриваються і утворюють нескінченні цикли.

Іноді для оптимізації роботи хеш-таблиць може бути необхідне рехешування. Це процес, який включає в себе збільшення розміру таблиці, зміну хеш-функції або перенесення даних у нову структуру. Рехешування допомагає усунути негативні наслідки від великої кількості колізій або погано підібраної хеш-функції. Важливо контролювати густоту заповнення таблиці, оскільки надмірна щільність може призвести до збільшення кількості колізій і уповільнення роботи алгоритму.

Для вторинних ключів, які не є унікальними ідентифікаторами записів, застосовуються інвертовані індекси. Ці структури зберігають відсортовані списки адрес основних записів для кожного вторинного ключа. Інвертовані індекси дозволяють здійснювати швидкий пошук без необхідності звертатися до основної таблиці, але при цьому мають значні витрати часу на складання і оновлення, що може бути неефективно при великих обсягах даних. Для малих таблиць також можуть використовуватися бітові карти, які дозволяють ефективно обробляти складні запити за допомогою логічних операцій. Однак, при великих обсягах даних бітові карти можуть стати менш ефективними.

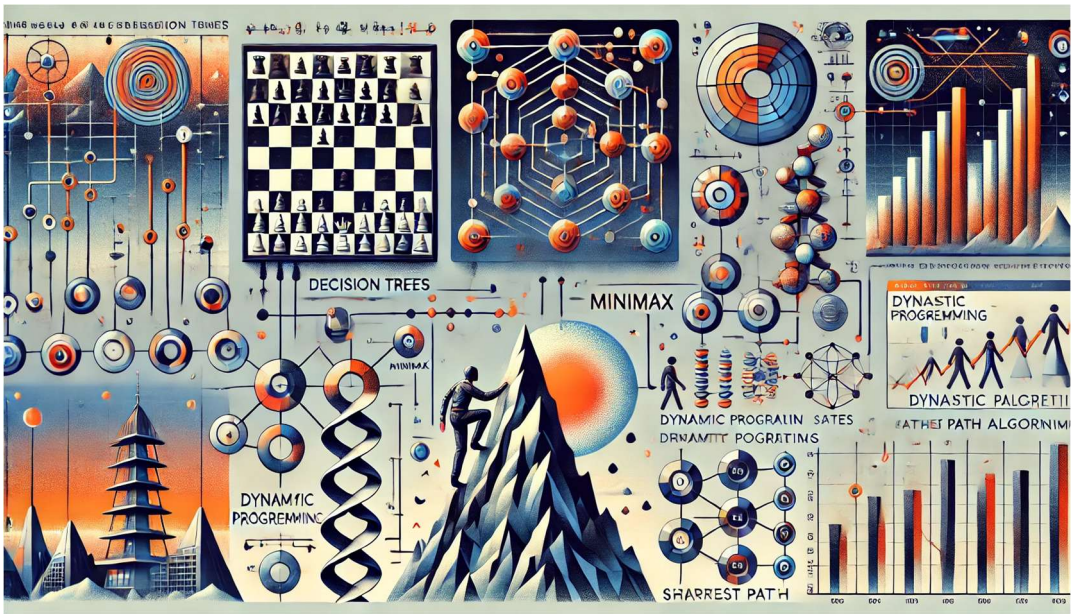
1.9 Методи розробки алгоритмів

Основні методи комп'ютерного розв'язання складних задач становлять фундамент для розвитку інтелектуальних систем і алгоритмів, здатних вирішувати проблеми, які раніше вважалися доступними лише для людини. Такими є задачі, в яких потрібно знайти найкраще рішення серед великої кількості можливих варіантів, часто навіть нескінченної множини. Людина, підходячи до таких проблем, не переглядає всі варіанти, а покладається на свій досвід, відкидаючи явно неприйнятні варіанти, що дозволяє зменшити обсяг пошуку. Чим більше досвіду у людини, тим швидше вона знаходить правильний підхід і вирішення проблеми. У світі обчислювальної техніки такі процеси зазвичай автоматизуються через алгоритми штучного інтелекту, які моделюють цей досвід.

Алгоритми, які працюють на основі перебору варіантів, називаються алгоритмами перебору. Вони намагаються знайти найкраще рішення серед всіх можливих варіантів. Завдання програміста полягає в тому, щоб максимально ефективно побудувати алгоритм, вводячи в нього різноманітні «умови досвіду», які дозволяють скоротити кількість варіантів для перевірки. Якщо пошук здійснюється цілеспрямовано, то програма стає більш «розумною», що дозволяє швидше досягти найкращого результату. На практиці більшість методів є спеціалізованим перебором варіантів, хоча в широкому сенсі їх можна віднести до алгоритмів штучного інтелекту, особливо коли вони виконують більш складні завдання.

Одним з таких підходів є метод часткових цілей, який полягає в поділі великої задачі на кілька менших підзадач. Це дозволяє зменшити складність вирішення кожної окремої частини, а отже, зробити загальне вирішення задачі більш доступним. Кожну з підзадач можна вирішити окремо, застосовуючи вже знайомі або простіші алгоритми. Наприклад, у задачі сортування масиву метод часткових цілей полягає у пошуку мінімуму в несортованій частині масиву та його додаванні до вже відсортованої частини. Таким чином, задача розв'язується поступово, за допомогою кількох простих кроків.

Яскравим прикладом методу часткових цілей є класична задача "Ханойських веж". Вона передбачає перенесення дисків з одного стержня на інший, дотримуючись певних правил, таких як заборона ставити більший диск на менший. Рішення цієї задачі можна розглянути як рекурсивний процес, де вежу з n дисків можна розділити на менші вежі з n-1 диска, що значно полегшує вирішення задачі.



Малюнок 1.11 – Розробка алгоритмів

Ще одним важливим методом розв'язання складних задач є динамічне програмування. Цей метод застосовується, коли задачу неможливо поділити на кілька простих підзадач, рішення яких можна об'єднати для отримання загального результату. У таких випадках задача розбивається на ще менші частини, і процес розв'язання триває доти, поки не буде знайдений оптимальний варіант. Це дозволяє значно знизити складність задачі, хоча часто така стратегія може призвести до експоненційного часу виконання. Однак, зазвичай, вдається досягти поліноміального часу виконання, зберігаючи результати проміжних рішень і використовуючи їх при подальших розрахунках. Динамічне програмування — це процес поетапного вибору оптимальних рішень на кожному етапі, застосовуючи принцип оптимальності Белмана, що дозволяє знаходити найкраще рішення з множини можливих.

Метод сходження орієнтується на поступове покращення рішення шляхом пошуку все кращих варіантів. Це нагадує процес сходження на вершину, коли на кожному кроці обирається найбільш оптимальний варіант. Наприклад, задача мандрівного крамаря, яка полягає у пошуку найкоротшого циклу для відвідування n міст, може бути вирішена таким способом. Спочатку можна вибрати простий варіант маршруту, рухаючись від одного міста до найближчого, а потім по ходу вдосконалювати його, знаходячи більш короткі шляхи. Подібний підхід також застосовується в "скупих" алгоритмах, таких як алгоритм Дейкстри для побудови найкоротшого шляху або алгоритм Крускала для побудови мінімального остовного дерева.

Однак не всі "скупі" алгоритми дають глобально оптимальний результат; іноді вони дають лише локальний оптимум, і для досягнення найкращого глобального результату потрібно комбінувати їх з іншими методами. Ідея методу сходження полягає в тому, щоб постійно покращувати рішення, закріплюючи зміни, що ведуть до оптимізації.

Дерева розв’язків є важливим інструментом для моделювання складних задач, зокрема в настільних іграх. Кожен вузол дерева представляє один крок у вирішенні задачі, а гілки — різні варіанти рішень, що можуть привести до більш повного розв'язку. Листя дерева — це остаточні рішення задачі. У випадку таких ігор, як шахи або хрестики-нулики, кожен хід можна трактувати як новий вузол дерева. Стратегії, як мінімакс, можуть бути застосовані для вибору оптимальних ходів, намагаючись мінімізувати найгіршу ситуацію для себе та максимально покращити свою позицію.

Загалом, описані методи розв'язання складних задач дозволяють комп'ютерам імітувати людські стратегії розв'язання проблем і значно просунутися в розвитку інтелектуальних систем.

РОЗДІЛ 2 АРХІТЕКТУРА СИСТЕМИ

2.1 Клієнт-серверна архітектура

Клієнт-серверна архітектура є основною моделлю для розподілених систем, в якій функції та обчислювальні ресурси розподіляються між двома основними елементами: клієнтом та сервером. Клієнт виступає як ініціатор з'єднання, надсилаючи запити на сервер для виконання певних дій або отримання необхідних даних. Сервер, у свою чергу, відповідає за обробку цих запитів, управління даними, виконання бізнес-логіки та обробку інформації, яка надійшла від клієнта. Цей підхід став надзвичайно популярним у веб-розробці завдяки своїй ефективності у використанні ресурсів, а також здатності масштабувати додатки в міру зростання навантаження. У такій архітектурі клієнт зазвичай є інтерфейсом, через який користувач взаємодіє із системою. Клієнт може бути представленим як веб-додаток, що працює у браузері, або мобільним чи десктопним додатком. Його головною метою є забезпечити користувачеві зручний інтерфейс для введення даних, перегляду інформації, налаштування параметрів системи та виконання інших операцій. Клієнт надсилає запити на сервер, щоб виконати дії, такі як збереження даних, обробка транзакцій, або отримання аналітичної інформації. Після цього він отримує відповідь від сервера і відображає її у зрозумілому для користувача вигляді.

Малюнок 2.1 – Клієнт-серверна архітектура

У сучасних веб-додатках клієнт може виконувати частину обробки даних на своєму боці, наприклад, здійснювати валідацію форм перед їх надсиланням або кешувати дані для більш швидкого доступу до них. Сервер же є основним елементом архітектури, який відповідає за обробку запитів, виконання бізнес-логіки та взаємодію з базою даних. Він може виконувати складні обчислення, керувати транзакціями, зберігати дані та виконувати інші операції, що необхідні для виконання запитів клієнтів. Сервер отримує запити від клієнтів, обробляє їх і відправляє відповідь, яка може містити необхідні дані або результати виконаних операцій. Крім того, сервер займається автентифікацією та авторизацією користувачів, контролює доступ до ресурсів, забезпечує безпеку та захист даних. У великих системах серверна частина може бути розділена на кілька компонентів, таких як сервери обробки додатків, сервери для баз даних, сервери для аналітики або кешування, що дозволяє розподіляти навантаження і забезпечувати масштабованість системи.

База даних є центральним елементом зберігання даних у клієнт-серверній архітектурі. Вона виконує роль сховища для всієї інформації, яка використовується додатком, і відповідає за забезпечення швидкого доступу до цих даних, а також за виконання операцій з пошуку, фільтрації, оновлення та видалення інформації. Для забезпечення надійності зберігання і відмовостійкості база даних часто підтримує резервне копіювання та шифрування даних. Сервер звертається до бази даних через запити, щоб виконати необхідні операції, такі як отримання інформації для відображення користувачу або збереження змін у фінансових записах.

Процес роботи в клієнт-серверній архітектурі починається з того, що клієнт ініціює з'єднання із сервером, надсилаючи запит на виконання операцій, наприклад, для отримання фінансового звіту чи обробки транзакції. Сервер отримує цей запит, перевіряє права доступу користувача, виконує необхідні операції, звертається до бази даних для зберігання або отримання даних і після цього формує відповідь, яку відправляє назад клієнту. Клієнт отримує цю відповідь і відображає її у вигляді, зручному для користувача.

Клієнт-серверна архітектура може реалізовуватися в різних варіантах. У простій однорівневій архітектурі всі компоненти розміщуються на одному сервері, що є досить простим рішенням, але обмежує масштабованість і продуктивність системи. У дворівневій архітектурі сервер обробляє бізнес-логіку та керує доступом до бази даних, в той час як клієнт відповідає за інтерфейс користувача, що дозволяє підвищити продуктивність і зменшити навантаження на сервер. Багаторівнева (N-tier) архітектура додає додаткові рівні обробки, такі як сервери API, аналітики або кешування, що дозволяє більш ефективно розподіляти обчислювальні ресурси та масштабувати систему.

Основними перевагами клієнт-серверної архітектури є масштабованість, безпека, модульність і гнучкість. Масштабованість досягається завдяки можливості додавати нових користувачів і розширювати серверну інфраструктуру. Централізоване управління на сервері полегшує контроль доступу, а також дозволяє зберігати і застосовувати політики безпеки та шифрування даних. Модульність дозволяє розділяти компоненти системи, що спрощує їх обслуговування та оновлення. Гнучкість архітектури дозволяє використовувати клієнтів на різних платформах, таких як веб, мобільні або десктопні додатки.

Однак, попри численні переваги, клієнт-серверна архітектура має свої недоліки. Система залежить від серверного обладнання, і у разі його відмови клієнти втрачають доступ до даних. Велике навантаження на сервер може призвести до затримок у відповідях та уповільнення роботи додатка, що вимагає постійної оптимізації інфраструктури. Крім того, впровадження та підтримка такої архітектури потребують витрат на налаштування серверів, забезпечення їх резервного копіювання, моніторингу та підтримки стабільної роботи.

Для забезпечення надійності, відмовостійкості та покращення продуктивності клієнт-серверні системи часто використовують додаткові елементи, такі як технології кешування, що допомагають зменшити час відгуку. Кешування дозволяє зберігати часто використовувані дані в оперативній пам'яті сервера, що значно знижує навантаження на базу даних і пришвидшує обробку запитів. Для великих додатків також використовується балансування навантаження, коли запити розподіляються між кількома серверами, що дозволяє уникнути перевантажень.

Іншими сучасними варіантами реалізації клієнт-серверної архітектури є мікросервіси. Вони дають ще більше гнучкості в масштабуванні системи, оскільки бізнес-логіка розподіляється між незалежними модулями — мікросервісами. Кожен мікросервіс виконує одну конкретну задачу, що полегшує оновлення та обслуговування без зупинки всієї системи, а також дозволяє масштабувати окремі компоненти в залежності від навантаження, покращуючи відмовостійкість.

Безпека також є важливою складовою клієнт-серверної архітектури. Системи аутентифікації та авторизації забезпечують захист даних, контролюючи доступ до ресурсів. Протоколи шифрування, такі як SSL/TLS, гарантують безпеку даних під час їх передачі. Сервери також здійснюють моніторинг для виявлення і запобігання загрозам, таких як SQL-ін'єкції чи DDoS-атаки.

Всі ці фактори роблять клієнт-серверну архітектуру ідеальним вибором для створення надійних, масштабованих систем, що можуть підтримувати велику кількість користувачів і обробляти значний обсяг операцій, що робить її невід'ємною частиною сучасної інфраструктури для різноманітних корпоративних і фінансових додатків, а також веб- та мобільних сервісів.

2.2 Модуль обробки даних

Модуль обробки даних в дипломній роботі, присвяченій автоматизованій системі для фінансово-облікових операцій на базі веб-додатку, займає центральне місце в структурі системи. Цей компонент відповідає за весь процес збирання, збереження, обробки, аналізу та маніпулювання фінансовими даними, забезпечуючи надійність та ефективність роботи системи. Мета цього модуля полягає в автоматизації операцій, пов'язаних із веденням фінансового обліку, що включає обробку транзакцій, перевірку та збереження фінансової інформації, створення різноманітних звітів, а також здійснення необхідних фінансових розрахунків для правильного функціонування системи.

Основним завданням модуля є автоматизоване виконання різних фінансових операцій, таких як облік коштів, управління транзакціями, аналіз витрат та доходів, а також генерування аналітичних звітів для подальшого прийняття рішень. Кожна операція в системі, від здійснення транзакції до оновлення даних в базі, проходить через цей модуль. Коли користувач здійснює фінансову операцію, модуль обробляє вхідні дані, включаючи інформацію про тип операції, суму, час її виконання та користувача, після чого автоматично оновлює записи в базі даних. Це дозволяє точно відображати фінансовий стан рахунків і гарантує, що кожна операція враховується без помилок.

Малюнок 2.2 – Модуль обробки даних

Для підвищення ефективності та зниження навантаження на сервер, модуль оптимізований для роботи з великими обсягами даних та високочастотними операціями. Важливим елементом є кешування даних, яке дозволяє зберігати найбільш часто використовувану інформацію в оперативній пам'яті. Завдяки цьому скорочується час доступу до даних, що є особливо корисним під час обробки великих фінансових запитів, наприклад, генерації щомісячних звітів або обчислення статистики за великий період. Окрім кешування, система використовує паралельну обробку, що дозволяє значно підвищити швидкість виконання складних операцій.

Алгоритми перевірки та валідації даних є важливим аспектом модуля. Вони відповідають за забезпечення точності та достовірності фінансової інформації, що надходить у систему. Під час надходження нових фінансових даних система здійснює перевірку правильності введених значень, їх відповідність встановленим правилам, а також наявність можливих дублювань. Це дає змогу уникнути помилок, таких як неправильне класифікування операцій чи введення некоректних даних, що критично важливо для фінансових систем, де навіть незначні похибки можуть призвести до серйозних наслідків.

Важливою частиною модуля є система звітності та аналітики, яка дозволяє користувачам отримувати детальну інформацію про фінансовий стан організації. Звітність формуються на основі запитів користувачів і дають змогу отримувати фінансові підсумки за обраний період, здійснювати аналіз доходів і витрат, оцінювати динаміку операцій та виконувати інші аналітичні операції. Візуалізація даних через графіки, таблиці або діаграми полегшує сприйняття і аналіз результатів, допомагаючи у прийнятті важливих управлінських рішень.

Модуль обробки даних активно взаємодіє з іншими компонентами системи, зокрема, з модулем автентифікації, базою даних, а також із зовнішніми сервісами. Це дозволяє інтегрувати автоматичне завантаження банківських виписок, синхронізувати з іншими обліковими системами, а також здійснювати обмін даними з податковими службами чи іншими органами. Такі інтеграції значно знижують необхідність ручного введення даних, скорочуючи час на їх оновлення та обробку.

Безпека даних є ключовим аспектом в роботі з фінансовою інформацією, тому модуль обробки даних включає кілька рівнів захисту. Використовуються методи шифрування даних як під час збереження, так і при передачі, що забезпечує захист від несанкціонованого доступу. Крім того, система застосовує політику обмеження доступу на основі ролей, що дає змогу контролювати, хто і які дані може переглядати чи змінювати. Логування всіх операцій допомагає відстежувати

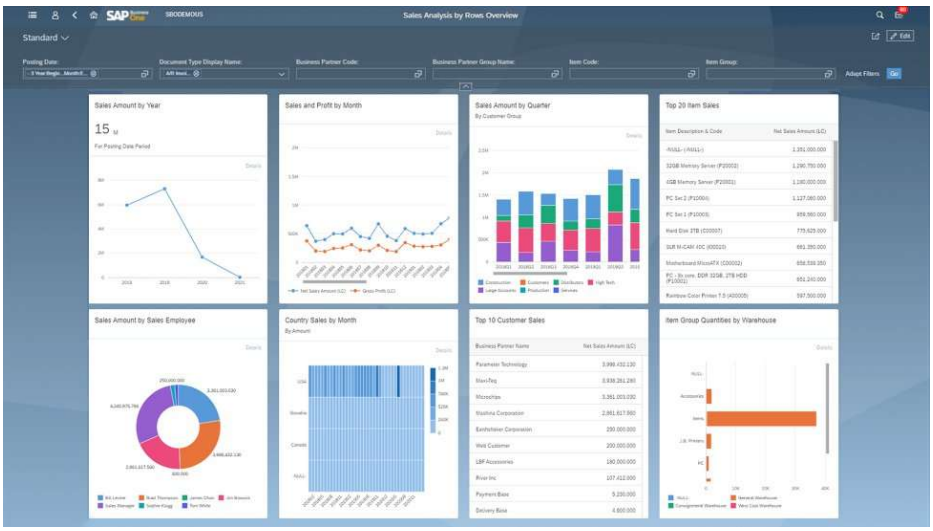
будь-які зміни, що забезпечує додатковий рівень безпеки та дозволяє вчасно виявляти можливі загрози.

Модуль обробки даних є важливою складовою частиною автоматизованої системи фінансово-облікових операцій, яка не лише забезпечує надійне і точне управління фінансовими потоками, але й дозволяє здійснювати необхідну аналітику та генерувати звіти. Він дозволяє оперативно приймати рішення в рамках фінансової діяльності організації, забезпечуючи повну обробку і збереження даних, а також надаючи користувачам всі необхідні інструменти для ефективного управління фінансами.

2.3 Модуль звітності та аналітики

Модуль звітності та аналітики є однією з найважливіших складових автоматизованої системи для фінансово-облікових операцій, яка відіграє критичну роль у наданні компанії глибокого розуміння своїх фінансових показників і підтримці прийняття обґрунтованих рішень. Його основне призначення — автоматизоване створення звітів, обробка і аналіз даних, а також надання інструментів для прогнозування, що дозволяє керівництву та аналітикам ретельно відслідковувати фінансовий стан підприємства, оперативно реагувати на зміни в ситуації і здійснювати точне планування на основі наявних даних.

Основними завданнями цього модуля є регулярне генерування фінансових звітів різної складності та формату, аналіз фінансових показників, а також надання потужних інструментів для візуалізації та прогнозування фінансових результатів. Це дозволяє не лише створювати базові звіти, такі як баланси, звіти про прибутки і збитки, звіти про рух грошових коштів, але й забезпечувати можливість глибокого аналізу, що охоплює такі аспекти, як рентабельність, ліквідність і оборотність активів. Візуалізація даних через графіки, діаграми та таблиці дозволяє користувачам легше інтерпретувати інформацію і виявляти тренди та аномалії, які можуть бути неочевидними при роботі з сирими числовими даними.



Малюнок 2.3 – Звітності та аналітика

Модуль забезпечує можливість отримувати звіти у різних форматах і за різні часові періоди — щодня, щомісяця, щокварталу чи щорічно. Це дозволяє компанії здійснювати моніторинг фінансового стану в режимі реального часу або ж отримувати періодичні звіти для виконання податкових зобов'язань чи підготовки до аудиту. Кожен звіт можна адаптувати під індивідуальні вимоги користувачів, наприклад, за допомогою фільтрів або зміни періодів звітності. Це дає змогу отримати максимально релевантну інформацію, що відповідає специфічним запитам різних відділів компанії або окремих керівників.

Окрім стандартних звітів, модуль аналітики має можливості для проведення поглибленого аналізу фінансових даних. За допомогою обчислення ключових фінансових показників і їх порівняння з запланованими значеннями можна відслідковувати динаміку витрат і доходів, а також оцінювати фінансові результати в контексті попередніх періодів. Це дозволяє вчасно виявляти проблеми або можливості для зростання, а також аналізувати тренди в бізнес-процесах компанії.

Однією з важливих можливостей модуля є функціональність прогнозування фінансових показників. Використовуючи алгоритми машинного навчання і статистичні методи, система здатна передбачати майбутні фінансові результати, створюючи моделі розвитку за різними сценаріями. Це особливо важливо для планування бюджету, оцінки ефективності нових проектів і прогнозування загального фінансового стану на різних етапах. Завдяки таким можливостям компанія може будувати більш точні фінансові плани, виявляти потенційні ризики та розробляти стратегії для їх мінімізації.

Для забезпечення більшої гнучкості в роботі з звітами та аналітичними даними, модуль надає користувачам можливість кастомізувати звіти відповідно до їхніх індивідуальних потреб. Це включає в себе налаштування фільтрів, вибір

конкретних фінансових операцій, а також можливість експортувати дані у популярні формати, такі як PDF, Excel або CSV. Це дозволяє зберігати, аналізувати і передавати звіти зацікавленим сторонам, що значно полегшує процес обміну інформацією всередині компанії та з іншими організаціями.

Безпека даних є важливим аспектом роботи модуля звітності та аналітики. Для цього в системі реалізовані багаторівневі механізми контролю доступу, засновані на ролях. Це дозволяє чітко визначити, хто з користувачів має доступ до певних фінансових звітів і аналітичних даних, що в свою чергу знижує ризик несанкціонованого доступу до чутливої інформації. Система веде журнал дій користувачів, що дозволяє забезпечити прозорість операцій і здійснювати аудит змін у фінансових даних, що є важливим для дотримання вимог конфіденційності та безпеки.

Модуль звітності та аналітики має інтеграційні можливості з іншими частинами системи, зокрема з модулем обробки даних та базою даних, що дозволяє забезпечити своєчасне оновлення і синхронізацію фінансових даних. Таке інтегроване рішення забезпечує оперативний доступ до актуальних фінансових показників і дозволяє швидко реагувати на будь-які зміни в ситуації. Крім того, модуль може бути інтегрований з іншими зовнішніми системами, такими як бухгалтерські програми або банківські сервіси, що дозволяє розширити можливості для більш глибокої фінансової аналітики.

Загалом, модуль звітності та аналітики є потужним інструментом для автоматизованої обробки фінансових даних, аналізу показників, прогнозування та створення звітності. Його використання дає компанії можливість ефективно управляти фінансовими потоками, оптимізувати процеси прийняття рішень, покращити прозорість фінансових операцій і забезпечити контроль над фінансовим станом, що сприяє підвищенню конкурентоспроможності і стійкому розвитку організації.

2.4 База даних

База даних (БД) являє собою організовану структуру зберігання та управління даними, що дозволяє здійснювати ефективний доступ до інформації, її збереження, редагування та відновлення. Вона грає важливу роль у забезпеченні цілісності та безпеки даних, а також у підтримці швидкого доступу до необхідної інформації. Бази даних використовуються в різних сферах, починаючи від фінансових та медичних установ і закінчуючи соціальними мережами, комерційними платформами і науковими дослідженнями. Особливо важливою є їх

роль у великих системах, де обсяг інформації може досягати великих масштабів і потребує організованого зберігання і швидкого оброблення.

База даних складається з кількох основних елементів, таких як дані, що є головною частиною структури, представлені у вигляді таблиць, які містять записи з різними атрибутами. Дані в базах даних організовані у систематизованій формі, що дає можливість легко здійснювати їх пошук та оновлення. Іншою важливою складовою є система керування базами даних (СКБД), яка виступає як програмне забезпечення для організації доступу до інформації, забезпечення цілісності даних та управління правами доступу. Вона дозволяє користувачам створювати, редагувати, видаляти й читати дані, забезпечуючи при цьому безпеку і контроль.

Крім того, база даних має таку структуру, як схема, що визначає логічну організацію даних, включаючи таблиці, поля, типи даних, індекси та відносини між різними елементами. Індекси — це спеціальні структури, що дозволяють пришвидшити доступ до даних, покращуючи ефективність пошуку та обробки запитів. Ключі відіграють важливу роль у зв'язуванні різних таблиць між собою: первинні ключі гарантують унікальність записів, тоді як зовнішні ключі визначають зв'язки між таблицями.

Сучасні бази даних поділяються на кілька основних типів, кожен з яких має свої особливості та підходить для різних задач. Реляційні бази даних, що є найбільш популярними, організовують дані у вигляді таблиць, і для маніпуляцій з ними використовують SQL — структуровану мову запитів. Вони ідеально підходять для завдань, де важлива структура даних та збереження зв'язків між таблицями. У свою чергу, нереляційні бази даних, такі як NoSQL, пропонують більшу гнучкість, зберігаючи дані в інших формах, таких як документи чи графи. Це дозволяє ефективно працювати з великими обсягами неструктурованих даних і досягати горизонтального масштабування. Графові бази даних використовуються в ситуаціях, де важливі зв'язки між об'єктами, як, наприклад, у соціальних мережах або системах рекомендацій. Об'єктно-орієнтовані бази даних зберігають інформацію у вигляді об'єктів, подібно до об'єктно-орієнтованого програмування, і використовуються, зокрема, в складних програмних системах. Для специфічних задач, таких як аналіз даних, що змінюються з часом, оптимізовані бази даних з часовими рядами, як-от InfluxDB, дозволяють зберігати дані, які мають часову складову, наприклад, фінансові або метеорологічні показники.

Малюнок 2.4 – База даних

Основні операції, які виконуються над базами даних, згруповані в CRUD-операції: створення (Create), читання (Read), оновлення (Update) та видалення (Delete). У реляційних базах даних ці операції виконуються за допомогою SQL-запитів, таких як SELECT, INSERT, UPDATE та DELETE. Однак для більш складних задач і забезпечення узгодженості бази даних, важливим є управління транзакціями, яке гарантує виконання операцій у межах однієї транзакції. Основними принципами транзакцій є ACID — атомарність (операція або виконується повністю, або не виконується взагалі), цілісність (відновлення коректного стану бази після операцій), ізоляція (незалежність транзакцій) і довговічність (стійкість даних після завершення транзакції).

Реплікація даних дозволяє створювати їх копії в іншій базі даних для підвищення надійності та продуктивності системи. Це особливо важливо для великих розподілених систем, де безперервний доступ до даних є критичним. Процес резервного копіювання дає змогу створювати резервні копії даних, що дозволяє швидко відновити їх у разі збоїв або інших непередбачуваних ситуацій.

Безпека бази даних є важливим аспектом управління даними, оскільки захист від несанкціонованого доступу до інформації є пріоритетним завданням. Для цього використовуються різноманітні механізми, зокрема аутентифікація та авторизація користувачів, шифрування даних і журналювання всіх операцій з базою даних. Це допомагає контролювати доступ до чутливої інформації, забезпечувати її конфіденційність і запобігати витоку даних.

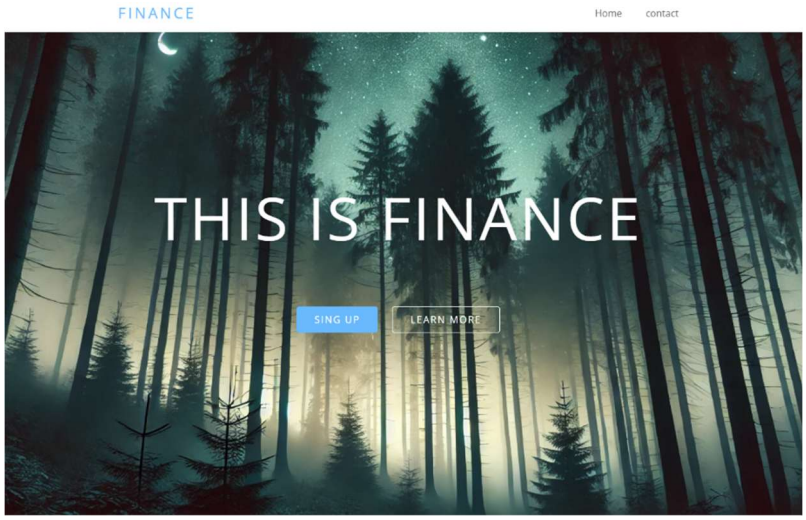
Використання баз даних у сучасних технологіях є надзвичайно широким. Вони застосовуються в фінансових системах для обробки та зберігання транзакцій, у електронній комерції для керування інформацією про товари і клієнтів, у соціальних мережах для зберігання даних про користувачів, а також в науці для обробки великих обсягів даних. Важливу роль бази даних також відіграють у розвитку Інтернету речей (IoT), де необхідно обробляти дані з безлічі сенсорів і пристроїв.

Сучасні бази даних все більше орієнтовані на гнучкість і масштабованість. Хмарні рішення для баз даних дозволяють здійснювати горизонтальне масштабування і знижувати витрати на інфраструктуру. Водночас розвиток великих даних (Big Data) та штучного інтелекту вимагає нових типів баз даних, які здатні ефективно обробляти великі обсяги інформації в режимі реального часу, забезпечуючи оперативний доступ до аналітичних даних і підтримку складних алгоритмів для прогнозування та аналізу.

РОЗДІЛ 3 СТВОРЕННЯ Й РОЗРОБКА САЙТУ FINANCE

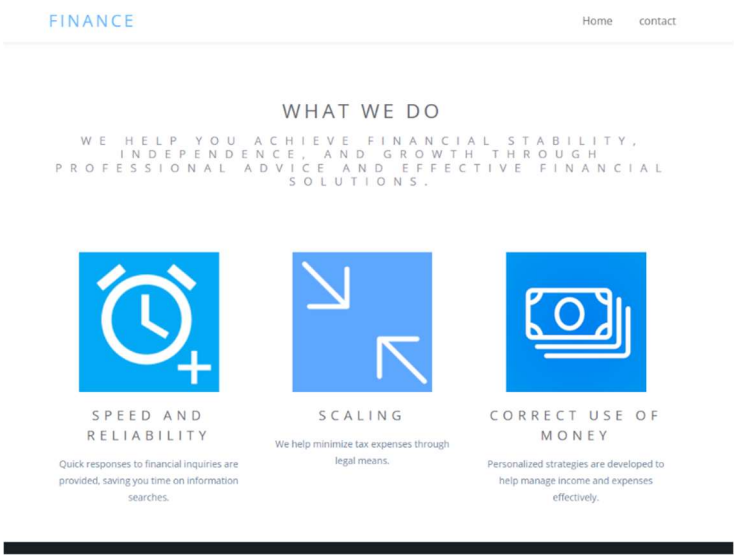
3.1 Дизайн сайту

Заголовок сайту "THIS IS FINANCE" створює чітке перше враження та відразу повідомляє користувачу основну тему ресурсу – фінанси. У поєднанні з атмосферним зображенням на фоні, він підкреслює надійність, стабільність і професійність. Лаконічний текст дозволяє акцентувати увагу на важливості фінансових рішень, а мінімалістичний дизайн із двома кнопками ("Sign Up" і "Learn More") спонукає до взаємодії з платформою. Такий підхід створює відчуття сучасності й інноваційності, заохочуючи користувачів до подальшого вивчення сайту.



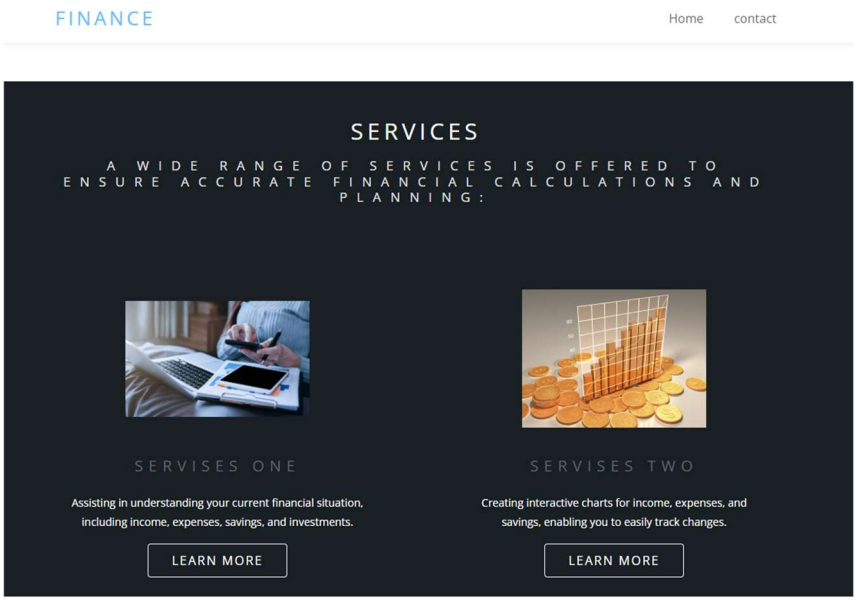
Малюнок 3.1 – Головна сторінка FINANCE

На цьому зображення є сторінкою з описом фінансових послуг. На ній представлено заголовок "WHAT WE DO" та текст про надання професійних фінансових рішень, спрямованих на досягнення фінансової стабільності, незалежності та зростання. Також є три іконки, які символізують ключові аспекти послуг, з коротким текстом під ними. Загальний стиль оформлення лаконічний, із домінуванням блакитних відтінків.



Малюнок 3.2 – What we do

Основний заголовок — "SERVICES". Згадується широкий спектр послуг, які спрямовані на забезпечення точних фінансових розрахунків і планування. Є два візуальних елементи, які представляють різні типи послуг. Кожна послуга має кнопку "LEARN MORE" для отримання додаткової інформації. Дизайн виконаний у темних тонах, що створює професійний вигляд.



Малюнок 3.3 – Сервіси які пропонує сайт

Зображення демонструє футер вебсторінки, розділений на три секції. Перша секція містить короткий текст "ABOUT VELOCITY" із заповнювачем (Lorem ipsum), що описує компанію. Друга секція — "USEFUL LINKS", яка містить кілька

гіперпосилань. Третя секція під назвою "SOCIAL" показує список іконок соціальних мереж із відповідними назвами платформ. У нижній частині футера є інформація про авторські права і згадка про платформу, на якій створено сайт. Дизайн мінімалістичний і організований.

Малюнок 3.4 – Соціальні мережі

3.2 Реєстрація

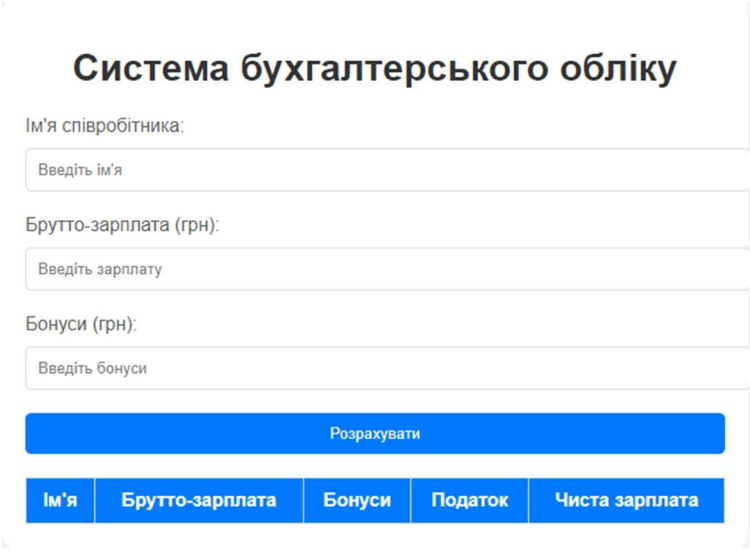
Для створення і оформлення реєстраційного вікна на сайті потрібно врахувати функціональність і дизайн, щоб забезпечити зручність для користувача. Вікно має містити поля для введення необхідної інформації, таких як ім'я, електронна пошта, пароль, а також кнопки для підтвердження реєстрації й можливого повернення на попередню сторінку.

Дизайн вікна повинен бути зрозумілим і адаптивним, з урахуванням кольорової гами сайту. Використовуйте чіткі шрифти, видимі мітки для полів вводу та інтерактивні елементи, такі як підсвічування кнопок при наведенні. Для безпеки рекомендується додати валідацію введених даних і можливість показу чи приховування пароля. Щоб уникнути перевантаження користувача, реєстраційне вікно варто зробити простим і мінімалістичним.

Малюнок 3.8 – Реєстрація

3.3 Функціонал сайту

Дана таблиця виконує функції обліку заробітної плати співробітників, автоматизуючи процес розрахунків і створення звітності. Розглянемо її функціонал детальніше:



Малюнок 3.5 – Розрахунок коштів

Таблиця виконує широкий спектр функцій, які спрямовані на автоматизацію процесу обліку заробітної плати та спрощення управління даними. Користувач має можливість працювати з інформацією швидко, ефективно й без зайвих зусиль.

Користувач може внести необхідну інформацію про кожного співробітника, включаючи його ім’я, розмір брутто-зарплати та бонуси. Усі ці дані вводяться за допомогою інтерактивної форми. Після завершення заповнення даних та їх відправлення, система обробляє інформацію і додає її до таблиці, доповнюючи результатами автоматичних розрахунків.

Таблиця автоматично здійснює обчислення основних обов’язкових податків, зокрема податку на доходи фізичних осіб (18%) та соціального внеску (1.5%). Для цього програма спершу підсумовує брутто-зарплату і бонуси, визначаючи загальний дохід співробітника. Потім обчислюється загальна сума податків, яка виводиться у відповідному стовпці.

Чиста зарплата, яку співробітник отримує "на руки", розраховується шляхом віднімання загальної суми податків і зборів із загального доходу. Це значення також автоматично додається до таблиці та відображається в окремому стовпці, що дозволяє зручно аналізувати кінцеві результати.

Після виконання всіх розрахунків система відображає повідомлення, яке інформує користувача про успішне завершення процесу. У цьому повідомленні зазначається ім’я співробітника, для якого виконувалися розрахунки, а також розмір його чистої зарплати.

Всі обчислення й отримані результати автоматично додаються до таблиці, яка виконує функцію сховища даних. Це дозволяє зберігати інформацію про всіх співробітників у одному місці для зручності перегляду й аналізу.

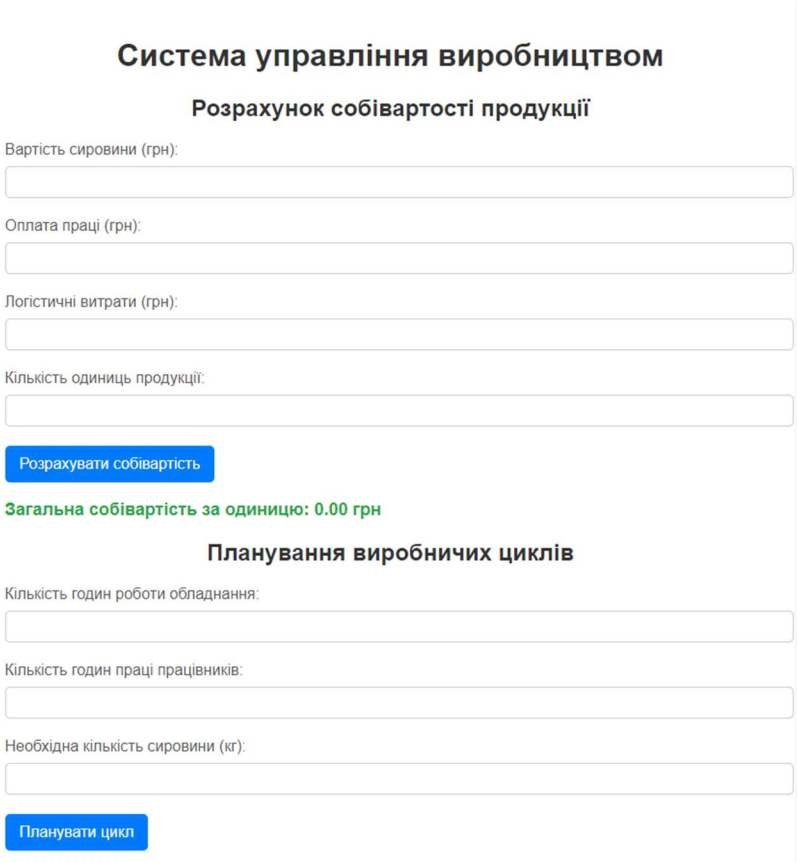
Після додавання даних до таблиці система автоматично очищує форму введення, що дозволяє швидко переходити до обробки інформації про наступного співробітника. Це забезпечує безперервність і зручність роботи.

Зберігання та доступ до інформації

Таблиця слугує місцем зберігання всієї введеної інформації разом із розрахованими результатами. Завдяки цьому користувач має зручний доступ до даних усіх співробітників, що робить систему ефективним інструментом для аналізу та управління фінансовою інформацією.

Таким чином, ця таблиця забезпечує повний цикл роботи з даними про заробітну плату співробітників, від збору вихідних даних до відображення кінцевих результатів у зрозумілому і структурованому форматі.

Ця система управління виробництвом розроблена для оптимізації ключових процесів, зокрема обчислення собівартості продукції та планування виробничих циклів. Вона спрямована на автоматизацію основних управлінських операцій, зменшення ризику помилок і підвищення ефективності обліку та управління ресурсами. Завдяки зручному інтерфейсу та інтерактивним можливостям, система забезпечує швидке та точне виконання розрахунків і планування. Розглянемо основні функціональні можливості більш детально.



Малюнок 3.6 – Розрахунок коштів

Цей компонент системи надає користувачам інструменти для швидкого і точного визначення собівартості виробництва однієї одиниці продукції. Основна мета — забезпечити прозорість і легкість у визначенні витрат, що сприяє ефективному фінансовому плануванню.

Система дозволяє користувачам ввести дані про ключові витрати, такі як вартість сировини, витрати на оплату праці та логістичні витрати. Ці дані автоматично обробляються для визначення середньої собівартості продукції на основі заданої кількості вироблених одиниць. У разі введення некоректних даних, таких як відсутність виробничого обсягу, система сповіщає користувача про помилку, що дозволяє уникнути неточностей.

Результати розрахунків представлені у зручному форматі для швидкого ознайомлення. Високий рівень точності забезпечує надійну базу для подальшого аналізу витрат і оптимізації виробничих процесів.

Другий ключовий модуль системи відповідає за ефективне управління ресурсами на виробництві. Він дозволяє користувачам розподіляти ресурси, такі як робочий час обладнання, людські ресурси та кількість необхідної сировини.

Система запитує у користувача дані про доступні ресурси та автоматично аналізує їх для складання плану виробничого циклу. У разі введення некоректних або неповних даних система видає попередження, забезпечуючи коректність і повноту інформації.

Після аналізу введених параметрів система надає користувачеві зворотний зв’язок у формі детального повідомлення про заплановане використання ресурсів. Це дозволяє користувачам відразу оцінити готовність до запуску виробничого циклу та внести корективи в разі потреби.

Система працює за принципом зчитування даних, їх обробки та виведення результатів. Завдяки автоматизованим алгоритмам, дані швидко аналізуються та використовуються для виконання необхідних розрахунків або планування.

У розрахунковому модулі враховуються всі введені витрати для отримання середньої собівартості продукції, що дає змогу користувачам планувати бюджети і приймати стратегічні рішення. У модулі планування система обробляє параметри та створює рекомендації щодо ефективного використання наявних ресурсів.

Система управління виробництвом має низку переваг, які роблять її незамінним інструментом для підприємств:

- Зручність і простота у використанні: Інтуїтивно зрозумілий інтерфейс і легкість у введенні даних дозволяють швидко освоїти систему.

- Автоматизація процесів: Ручні розрахунки більше не потрібні, що зменшує ризик людської помилки.

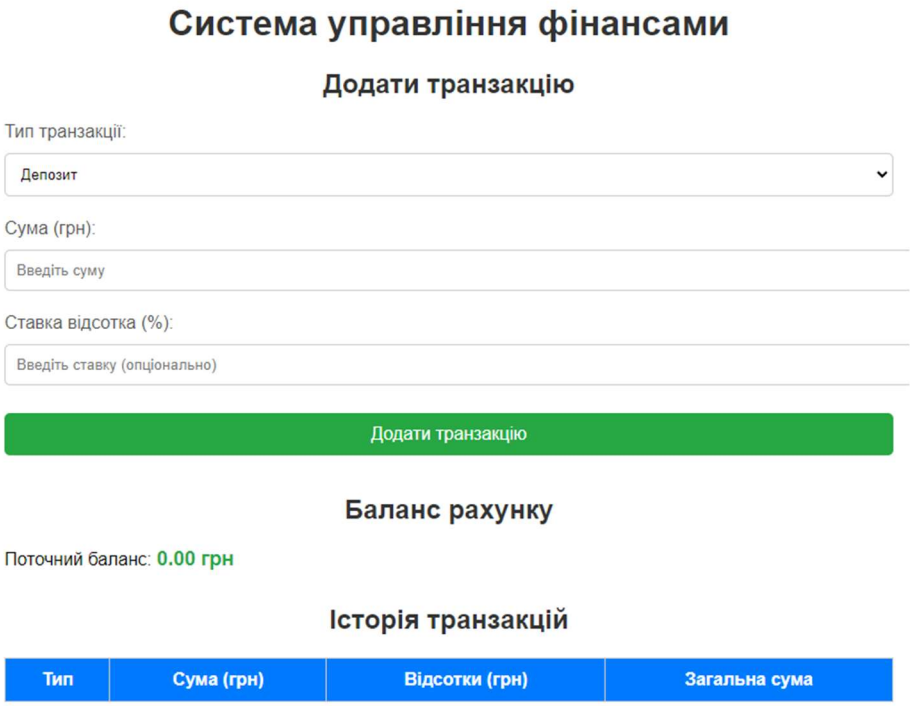
- Миттєвий доступ до результатів: Користувачі отримують результати в режимі реального часу, що значно пришвидшує процес ухвалення рішень.

- Гнучкість і адаптивність: Можливість змінювати параметри та коригувати плани залежно від поточних потреб виробництва.

- Прозорість і контроль: Усі розрахунки та плани базуються на чітких даних, що підвищує рівень довіри до результатів.

Ця таблиця є частиною інтегрованої фінансової системи, яка забезпечує облік транзакцій, контролює зміни балансу та автоматизує розрахунки, пов'язані з депозитами, зняттям коштів і кредитами. Вона дозволяє користувачам здійснювати операції, впливати на фінансовий баланс та надавати візуальне представлення фінансових змін у реальному часі. Система взаємодіє з кількома основними фінансовими функціями, такими як додавання транзакцій, оновлення балансу та розрахунок відсотків. Вся інформація про транзакції зберігається в таблиці, що дозволяє легко відслідковувати фінансові зміни.

Процес роботи з таблицею включає введення різних типів транзакцій, зокрема депозитів, зняття коштів і кредитів, а також введення відповідних сум і процентних ставок. Після подачі форми, кожна транзакція додається до історії, і це автоматично відображається в балансі. Для депозитів і кредитів система обчислює відсотки, додаючи їх до основної суми, що дозволяє точніше обчислити загальну суму на рахунку.



Малюнок 3.7 – Розрахунок коштів

Баланс оновлюється після кожної операції, при цьому враховуються як введена сума, так і можливі відсотки для депозитів та кредитів. У випадку зняття коштів, система просто віднімає зазначену суму. Крім того, таблиця забезпечує просту перевірку коректності введених даних, що допомагає запобігти помилкам при введенні некоректних або порожніх значень.

Інтерфейс системи дозволяє користувачам зручно вносити зміни, при цьому таблиця відображає всі актуальні операції в реальному часі, оновлюючи баланс і додаючи нові рядки в історію транзакцій. Таким чином, система забезпечує прозорість і контроль за фінансами, даючи користувачам змогу відслідковувати всі зміни та забезпечує ефективний фінансовий облік.

3.4 Забезпечення безпеки та захисту даних у децентралізованих системах

У процесі виконання дипломної роботи використовувались різні мови програмування, кожна з яких мала свою важливу роль у розробці вебдодатку. Основними з них були HTML, CSS та JavaScript, які забезпечували структуру, стиль та функціональність створеної системи.

HTML (HyperText Markup Language) виступала базовою мовою розмітки, яка визначала каркас вебсторінок. Вона забезпечувала основу для розташування всіх елементів, таких як текст, зображення, відео, посилання, таблиці та форми. HTML не лише створює структуру сторінки, але й дозволяє задавати метадані, що використовуються для SEO-оптимізації, полегшуючи індексацію сторінки пошуковими системами. Крім того, ця мова надає можливість інтеграції з іншими технологіями, такими як CSS для стилізації або JavaScript для додавання інтерактивності.

CSS (Cascading Style Sheets) відіграє ключову роль у візуалізації сторінок, створених на HTML. Завдяки використанню CSS, вдалося створити привабливий і функціональний дизайн. Ця мова дозволяє налаштовувати кольори, шрифти, відступи, розташування елементів, а також забезпечувати адаптивність вебдодатку, що є важливим для коректного відображення на різних пристроях. Анімації, переходи та медіа-запити значно покращують користувацький досвід, роблячи сторінки більш динамічними та зручними для взаємодії.

JavaScript додав вебдодатку інтерактивності та динамічності, перетворивши статичні сторінки на функціональні веб-додатки. За допомогою JavaScript було реалізовано такі елементи, як випадаючі меню, слайдери, модальні вікна та валідація форм на стороні клієнта. Особливо важливим було використання технологій AJAX та інтеграція з API, що забезпечило взаємодію з сервером у реальному часі. Завдяки цьому додаток міг динамічно завантажувати та оновлювати дані без необхідності перезавантаження сторінки. Крім того, JavaScript дозволяє працювати з мультимедійним контентом, анімаціями та навіть тривимірними об'єктами через WebGL, що значно розширює можливості сучасних вебдодатків.

Комбінація цих технологій дозволила створити ефективний, естетично привабливий і функціональний вебдодаток, який відповідає сучасним стандартам розробки. HTML забезпечив структуру, CSS — стиль та адаптивність, а JavaScript — інтерактивність і взаємодію з сервером. Завдяки цьому підходу вдалося досягти гармонійного балансу між функціональністю, продуктивністю та зручністю для користувачів.

Таблиця порівняння HTML, CSS та JavaScript

Фреймворк	Призначення	Функції	Основні особливості
HTML	Мова розмітки для створення структури вебсторінок.	Створення каркасу сторінки, додавання тексту, зображень, відео, форм і посилань.	Визначає елементи сторінки за допомогою тегів. Оновлена версія HTML5 підтримує мультимедіа (відео, аудіо) та семантичні елементи (<header>, <footer>).
CSS	Мова стилів для візуального оформлення вебсторінок.	Задає кольори, шрифти, розміри, відступи, вирівнювання, адаптивність, анімацію.	Підтримує каскадність і спадковість. Сучасні інструменти, як-от Flexbox і Grid, полегшують створення адаптивного дизайну.
JavaScript	Мова програмування для додавання динамічності та інтерактивності вебсторінкам.	Реалізація інтерактивних елементів, динамічного оновлення контенту, роботи з API, валідації форм, створення мультимедіа та анімацій.	Використовує DOM для роботи з HTML-структурою. Підтримує асинхронне програмування через AJAX, Promise та async/await.

РОЗДІЛ 4 ТЕСТУВАННЯ І ПІДТРИМКА ПРОГРАМИ

4.1 Види тестування

Тестування відіграє важливу роль на всіх етапах розробки програмного забезпечення, оскільки дозволяє виявити можливі проблеми ще до того, як продукт потрапить до кінцевого користувача. Його основна мета — забезпечення високої якості програмного забезпечення, що, в свою чергу, допомагає підвищити рівень задоволеності користувачів. Тестування також важливе для зниження ймовірності виникнення збоїв в роботі програми в майбутньому, особливо в критичних галузях, таких як фінансові та облікові системи, де навіть найменша помилка може призвести до серйозних наслідків.

У сфері забезпечення якості (QA) існує безліч видів тестування, які умовно поділяються на дві основні категорії: функціональне та нефункціональне тестування. Кожен з цих видів має свої особливості та спрямований на досягнення різних цілей. Функціональне тестування передбачає перевірку здатності програми виконувати визначені функції, згідно з вимогами замовника та очікуваними результатами, тоді як нефункціональне тестування зосереджується на перевірці таких аспектів, як продуктивність, безпека, масштабованість і зручність використання.

Малюнок 4.1 – Тестування

Функціональне тестування можна розглядати як комплекс перевірок, які гарантують, що кожна складова програми працює належним чином. Одним із важливих етапів цього процесу є модульне тестування. Модульне тестування є одним із найперших етапів перевірки, під час якого тестуються окремі компоненти або модулі програми. Це дозволяє виявити помилки на ранніх етапах розробки, до того як програма буде інтегрована в більш складну систему.

Основним завданням модульного тестування є забезпечення коректності функціонування кожного модуля. Тестування здійснюється шляхом ізоляції кожного компонента, що дозволяє точніше ідентифікувати та виправити помилки на ранніх етапах розробки. Важливо, що цей етап допомагає також підвищити якість коду, оскільки тестування стимулює розробників до написання більш чистого і організованого коду, що в свою чергу зменшує кількість помилок на подальших етапах інтеграції.

Процес модульного тестування складається з кількох ключових етапів. Спочатку розробляються тестові випадки, які базуються на вимогах до кожного модуля. Потім проводиться ізоляція модуля для перевірки його функціональності окремо від інших частин системи. Далі здійснюється виконання тестів, яке може бути як ручним, так і автоматизованим, за допомогою спеціальних інструментів для автоматизації тестування. Після цього результати тестування аналізуються, і в разі виявлення помилок, вони виправляються, а тестування повторюється до досягнення бажаних результатів.

У фінансово-облікових системах, де точність і стабільність мають критичне значення, модульне тестування стає особливо важливим. Наприклад, перевірка калькуляційного модуля, що здійснює обчислення податкових ставок або відсотків, повинна забезпечити правильність результатів навіть при великих обсягах даних. Модуль транзакцій, у свою чергу, перевіряється на коректність обробки валютних курсів, запису транзакцій та валідації введених даних. Модуль звітності тестується для забезпечення правильної генерації звітів за запитами користувачів, що включають перевірку доступу та здатності обробляти великі обсяги інформації.

Для автоматизації тестування використовуються різноманітні інструменти, які залежать від мови програмування та специфіки проекту. Серед найбільш популярних інструментів можна виділити JUnit для Java, NUnit для .NET, PyTest для Python та Mocha/Chai для JavaScript. Використання таких інструментів дозволяє прискорити процес тестування і забезпечити регулярність перевірок, що особливо важливо при частих змінах у коді.

Модульне тестування має кілька значних переваг. Однією з головних є рання виявленість помилок, що дозволяє зменшити витрати на їх виправлення. Також важливою перевагою є полегшення інтеграції, оскільки виявлення проблем на рівні окремих компонентів допомагає уникнути складнощів під час об'єднання модулів

у єдину систему. Крім того, постійне тестування підвищує якість коду, оскільки розробники змушені дотримуватися високих стандартів програмування. Автоматизація процесу тестування робить його швидким та ефективним, знижуючи ризики помилок і підвищуючи стабільність системи.

Однак, як і в будь-якому процесі, модульне тестування має й певні недоліки. Наприклад, тестування кожного модуля окремо може не дати повної картини того, як компоненти взаємодіють між собою в реальних умовах. Також, якщо код часто змінюється, тестові випадки потребують регулярного оновлення, що може потребувати додаткових ресурсів. Іншим недоліком є залежність від ізольованого тестового середовища, де реальна інтеграція з іншими системами чи компонентами може викликати нові проблеми.

Важливість модульного тестування особливо яскраво проявляється у фінансових системах, де точність і надійність кожного компонента є критичними. Кожен помилково оброблений модуль може призвести до великих фінансових збитків або серйозних помилок у звітності.

Далі йде інтеграційне тестування, яке є етапом перевірки того, як компоненти або модулі взаємодіють між собою в рамках всієї системи. Після того, як модульне тестування дозволило переконатися у коректності роботи окремих компонентів, інтеграційне тестування гарантує, що вони правильно взаємодіють між собою, забезпечуючи стабільність і точність всього програмного забезпечення. Це особливо важливо для фінансово-облікових систем, де взаємодія компонентів, таких як бази даних, API, модулі обробки транзакцій, повинна бути бездоганною для запобігання помилок у фінансових розрахунках і забезпечення належної роботи всіх функцій системи.

Інтеграційне тестування є важливим етапом перевірки програмного забезпечення, спрямованим на виявлення помилок, що виникають через взаємодію між різними компонентами системи. Його головною метою є забезпечення того, щоб усі частини програмного забезпечення, які раніше проходили окремі модульні тести, коректно працювали разом. Це дозволяє виявити дефекти, які можуть виникнути під час передачі даних між модулями або при обробці складних сценаріїв взаємодії. У процесі інтеграційного тестування тестується не лише функціональність окремих компонентів, а й їхня здатність працювати в єдиній системі, зберігаючи правильність обробки даних і виконання складних бізнес-процесів.

Інтеграційне тестування має на меті кілька важливих завдань. Одним із головних є перевірка правильності інтеграції модулів, що гарантує, що всі частини системи можуть взаємодіяти без помилок і збоїв. Також важливо виявляти будь-які конфлікти між модулями, наприклад, невідповідності в інтерфейсах або несумісність форматів даних, що передаються між компонентами. Крім того, цей

етап тестування передбачає перевірку складних бізнес-процесів, де декілька компонентів системи повинні працювати разом для досягнення результату, наприклад, для виконання фінансових операцій або розрахунків. Іншим важливим аспектом є аналіз поведінки системи під час передачі даних між модулями, що допомагає виявити потенційні втрати або спотворення інформації в процесі взаємодії компонентів.

Існує кілька підходів до проведення інтеграційного тестування, і вибір конкретного методу залежить від архітектури системи та вимог проекту. Наприклад, один із підходів — інкрементне тестування, що включає кілька варіантів: нисхідне, висхідне та міжрівневе тестування. У низхідному тестуванні перевірка починається з високорівневих компонентів і поступово переходить до нижчих, тоді як у висхідному — тестування починається з базових модулів, поступово інтегруючи більш складні частини системи. Міжрівневе тестування поєднує обидва ці підходи для досягнення більш комплексної перевірки. Інший варіант — це метод «Big Bang», при якому всі модулі інтегруються одночасно і перевіряються як єдине ціле. Хоча цей метод може бути ефективним у випадках, коли більшість модулів уже готові та протестовані окремо, його основним недоліком є складність у визначенні джерела проблеми, оскільки всі компоненти тестуються разом.

Інтеграційне тестування також може включати використання спеціальних програм, таких як драйвери і заглушки. Драйвери імітують роботу вищих рівнів системи, що дозволяє тестувати інші компоненти до їхнього повного завершення. Заглушки, у свою чергу, імітують роботу нижчого рівня, що дає змогу тестувати інтеграцію між різними частинами системи до того, як всі модулі будуть готові.

Важливість інтеграційного тестування особливо велика для фінансово-облікових систем, оскільки будь-які дефекти в обміні даними або взаємодії компонентів можуть призвести до серйозних помилок, таких як неправильні фінансові розрахунки або втрати даних. Наприклад, перевірка інтеграції бази даних із фінансовими модулями допомагає переконатися в тому, що всі транзакції коректно зберігаються і оновлюються в реальному часі, а також що відображення даних в облікових розрахунках відбувається без помилок. Іншим важливим аспектом є тестування взаємодії з API для обміну валют, що дозволяє перевірити правильність інтеграції з зовнішніми системами для отримання актуальних курсів валют і їхнього коректного застосування у фінансових розрахунках. Також велику увагу приділяють тестуванню функцій формування звітів, що мають збирати дані з різних модулів і коректно їх відображати.

Для автоматизації інтеграційного тестування використовуються різні інструменти. Наприклад, для тестування API активно застосовуються такі інструменти, як Postman і SoapUI. Для проведення інтеграційних тестів у Java

широко використовуються JUnit або TestNG, а для автоматизації тестування інтерфейсу користувача й взаємодії з бекендом — Selenium. У випадках, коли потрібно створити заглушки і драйвери для ізольованого тестування окремих компонентів, часто використовують Mockito.

Інтеграційне тестування має низку переваг, серед яких основними є можливість виявлення проблем у взаємодії між модулями та підвищення стабільності системи. Завдяки своєчасному виявленню помилок на етапі інтеграції зменшується ризик виникнення критичних проблем при подальших етапах тестування, таких як системне тестування. Це також допомагає забезпечити стабільність роботи системи в цілому. Однак цей етап має й певні недоліки, зокрема складність у локалізації помилок, оскільки в інтегрованих системах важко визначити точний компонент, який спричиняє проблему. Крім того, для тестування часто необхідно використовувати спеціальні інструменти, що може ускладнювати процес, а також збільшувати витрати на автоматизацію та підтримку тестування.

Таким чином, інтеграційне тестування є важливою частиною процесу забезпечення якості програмного забезпечення, що забезпечує коректну роботу всієї системи в цілому. Це тестування особливо критичне для фінансово-облікових систем, де точність і стабільність є найвищим пріоритетом.

Системне тестування є важливим етапом процесу забезпечення якості програмного забезпечення, що передбачає перевірку усіх функцій і компонентів системи в цілому. Основною метою цього виду тестування є виявлення дефектів, які можуть виникнути в результаті взаємодії різних частин системи, а також підтвердження того, що програма відповідає всім вимогам і здатна працювати стабільно в реальному середовищі.

Ціль системного тестування полягає в тому, щоб перевірити, як система в цілому виконує свої функції і як усі компоненти програми взаємодіють між собою. Важливо виявити дефекти, які не були помічені на попередніх етапах тестування, оскільки деякі з них можуть з'явитися лише при повному завантаженні системи або при одночасному використанні всіх її модулів. Окрім того, системне тестування оцінює відповідність продукту вимогам замовника, перевіряючи не тільки функціональність, але й нефункціональні аспекти, такі як продуктивність, безпека і зручність використання. Одна з ключових задач системного тестування – це перевірка стабільності й надійності системи, що є особливо критичним для таких сфер, як фінансові й облікові системи, де навіть незначні збої можуть призвести до серйозних наслідків.

Етапи системного тестування включають кілька ключових фаз. Першою з них є підготовка тестового середовища, яке імітує реальні умови роботи з урахуванням усіх зовнішніх залежностей і систем. Після цього складається план тестування, який включає визначення обсягу тестів і відповідних тест-кейсів для

перевірки усіх функцій системи. Далі виконується запуск тестів, після чого здійснюється аналіз результатів, порівняння фактичних результатів з очікуваними й виявлення відхилень. Якщо під час тестування виявлено дефекти, створюються звіти, і після їх виправлення проводяться ретестування і регресійне тестування для перевірки, що виправлення не вплинули на інші компоненти.

Завершенням системного тестування є виконання функціональних і нефункціональних тестів. Функціональне тестування включає перевірку основних функцій, сценаріїв використання програми та валідацію бізнес-логіки, що є критичним для фінансових систем, де помилки в обчисленнях можуть призвести до серйозних наслідків. Нефункціональне тестування охоплює перевірки продуктивності (наприклад, навантажувальне і стресове тестування), безпеки (захист даних, шифрування, автентифікація), сумісності (перевірка роботи на різних платформах) і доступності (забезпечення функціональності для користувачів з обмеженими можливостями).

Системне тестування в фінансово-облікових системах є особливо важливим, оскільки воно охоплює перевірку всього процесу обробки фінансових операцій – від введення транзакцій до формування звітності, що дозволяє оцінити точність розрахунків і коректність обробки помилок, а також безпеку транзакцій і захист даних.

Для автоматизації системного тестування застосовуються різноманітні інструменти, такі як Selenium для тестування інтерфейсу користувача, JMeter для перевірки продуктивності, Postman для тестування API, а також OWASP ZAP або Burp Suite для тестування безпеки.

Переваги системного тестування включають перевірку готовності програмного забезпечення до реального використання, забезпечення відповідності вимогам замовника, а також виявлення дефектів до випуску продукту, що допомагає уникнути проблем при його експлуатації. Водночас, одним із основних недоліків є висока вартість і трудомісткість цього етапу тестування, оскільки воно вимагає значних ресурсів і часу для підготовки тестового середовища та виконання всіх необхідних перевірок.

Приймальне тестування, що відбувається після системного тестування, є фінальним етапом, на якому перевіряється, чи відповідає продукт вимогам замовника і чи готовий він до впровадження. Це тестування оцінює, чи готове програмне забезпечення до використання в реальних умовах і чи задовольняє вимоги користувачів щодо функціональності, стабільності та зручності.

Приймальне тестування є критичним етапом процесу розробки програмного забезпечення, на якому перевіряється відповідність продукту вимогам та очікуванням кінцевих користувачів. Це тестування може здійснюватися як самими замовниками, так і спеціалістами тестувальної команди, або ж окремими

експертами. Зазвичай приймальне тестування включає кілька різновидів, що орієнтовані на різні аспекти готовності програмного продукту.

Одним із основних видів є альфа-тестування, яке проводиться внутрішньою командою розробників або обмеженою групою користувачів, що дозволяє виявити дефекти, не помічені під час попередніх етапів тестування. Цей тип тестування проводиться в умовах розробки і спрямований на перевірку всіх основних функцій системи. Після нього зазвичай слідує бета-тестування, яке проводиться вже у реальному середовищі. У цьому випадку продукт тестується кінцевими користувачами, що дозволяє виявити недоліки, які можуть виникнути тільки в реальних умовах використання. Завдяки зворотному зв'язку від користувачів можна зрозуміти, чи задовольняє продукт їхні очікування щодо функціональності та зручності використання.

Іншим важливим видом тестування є операційне приймальне тестування, яке перевіряє готовність продукту до експлуатації, зокрема його безпеку, здатність до аварійного відновлення та інші критичні аспекти для безперебійної роботи в реальних умовах. Крім того, регуляторне приймальне тестування забезпечує відповідність продукту нормативним вимогам, зокрема для фінансових чи медичних програм, що мають суворі вимоги до безпеки та захисту даних. Для спеціалізованих галузей, таких як фінанси або охорона здоров'я, також проводиться контрактне приймальне тестування, яке забезпечує виконання всіх умов, визначених в угодах між замовником і розробником. І, звісно, тестування кінцевого користувача є важливим етапом, на якому перевіряється, наскільки продукт відповідає вимогам тих, хто безпосередньо його використовуватиме в реальних умовах.

Процес приймального тестування включає кілька етапів. Спочатку розробляється детальний план тестування, в якому визначаються тестові сценарії, необхідні ресурси та умови. Далі, на етапі розробки тестових сценаріїв, підбираються варіанти, що охоплюють усі функціональні можливості системи. Після цього тестувальники переходять до проведення тестування, де перевіряється фактична поведінка системи на відповідність до очікуваних результатів. Якщо виявляються дефекти, створюються відповідні звіти для подальшого виправлення. Після завершення тестування оцінюються результати, і приймається рішення щодо готовності продукту до запуску або необхідності доопрацювання. Після цього готується фінальний звіт, що підтверджує відповідність програмного забезпечення вимогам і готовність до впровадження.

Особливо важливими є приклади приймального тестування в сфері фінансово-облікових систем. Це включає перевірку обробки транзакцій, що включає тестування всіх можливих сценаріїв обробки для перевірки точності та швидкості виконання. Також тестуються фінансові звіти, що мають відповідати

нормативним вимогам, а також перевіряється користувацький інтерфейс, щоб він був зручним і відповідав вимогам кінцевих користувачів. Не менш важливим є тестування безпеки транзакцій, щоб забезпечити належний рівень захисту даних і фінансових операцій.

Приймальне тестування може виконуватись як вручну, так і автоматизовано, залежно від складності продукту та потреб проекту. Для цього застосовуються різні інструменти автоматизації тестування, такі як Selenium для перевірки інтерфейсу, Jira для відстеження дефектів, а також TestRail для управління вимогами та тестовими сценаріями.

Серед переваг приймального тестування варто зазначити те, що воно забезпечує відповідність продукту всім вимогам, знижує ризики виявлення дефектів у реальному середовищі, а також забезпечує високий рівень задоволеності замовників і кінцевих користувачів. Завдяки цьому етапу підтверджується, що продукт готовий до запуску і відповідатиме вимогам, що ставляться до нього в реальному використанні.

Однак цей процес має й недоліки. Він може бути досить тривалим, особливо якщо виявляються серйозні дефекти, що потребують виправлення перед випуском продукту. Крім того, залучення кінцевих користувачів до тестування може збільшити витрати, а недостатня увага до попередніх етапів тестування може призвести до проблем на етапі приймального тестування. Суб'єктивність оцінки зручності використання також може ускладнити прийняття рішення про готовність продукту, оскільки різні користувачі можуть мати різні очікування та досвід.

Нарешті, важливим аспектом є ретельне тестування всіх можливих сценаріїв використання продукту. Якщо деякі з них залишаються непоміченими під час тестування, це може призвести до того, що критичні дефекти будуть виявлені вже після запуску продукту в реальні умови.

4.2 Методи тестування

Методи тестування є ключовими інструментами в процесі перевірки якості програмного забезпечення. Вони включають різноманітні підходи і техніки, що допомагають визначити, чи відповідає продукт вимогам, чи працює він стабільно, і чи забезпечує він належний досвід користувачів. Тестування є необхідним етапом на всіх стадіях розробки програмного продукту, адже лише через нього можна виявити дефекти та проблеми, що можуть виникнути в кінцевому продукті. Оскільки жоден продукт не є ідеальним, тестування допомагає забезпечити його надійність і відповідає всім критеріям, які ставлять кінцеві користувачі. Методи

тестування можна умовно поділити на кілька категорій, таких як ручне й автоматизоване тестування, а також функціональне і нефункціональне тестування.

Малюнок 4.2 – Методи тестування

Ручне тестування є одним із найпоширеніших методів оцінки якості програмного забезпечення. Воно полягає в тому, що тестувальники виконують всі тестові сценарії вручну, без використання автоматизованих інструментів. Цей підхід дає можливість більш глибоко оцінити користувацький досвід і зручність взаємодії з програмним продуктом. Під час ручного тестування тестувальники мають можливість адаптувати свої дії до поведінки програми, що особливо важливо на етапах, коли продукт потребує оцінки зручності інтерфейсу та візуальних елементів. Ручне тестування включає такі типи перевірок, як функціональне тестування, тестування на зручність, сумісність з різними системами та пристроями, а також регресійне тестування, яке дозволяє перевірити, чи не порушені вже перевірені функції після внесення змін до коду.

Основні переваги цього методу полягають у його гнучкості та здатності виявляти непередбачені проблеми, зокрема у взаємодії з кінцевими користувачами. Тестувальники можуть швидко адаптувати свої підходи до змін у продукті, що дозволяє виявляти нові, раніше не передбачені дефекти. Однак ручне тестування також має ряд недоліків, серед яких часоємність і схильність до людських помилок, що можуть призвести до пропуску важливих дефектів.

Автоматизоване тестування, своєю чергою, використовує спеціальні програмні інструменти для виконання тестів, що значно зменшує час, необхідний

для перевірки великих обсягів даних. Замість того, щоб вручну виконувати тестові сценарії, тестувальники створюють скрипти, які автоматично виконують завдання, порівнюють результати з очікуваними і фіксують будь-які відхилення. Автоматизація тестування є особливо корисною для виконання повторюваних тестів, таких як регресійне тестування, оскільки забезпечує швидке виявлення помилок після внесення змін у код.

Цей метод дозволяє зменшити ймовірність помилок, спричинених людським фактором, оскільки всі тести виконуються автоматично, без участі тестувальників. Крім того, автоматизовані тести є більш стабільними і можуть виконуватися в різних середовищах або на різних версіях програмного забезпечення без необхідності їх ручного налаштування. Однак автоматизація тестування вимагає наявності спеціалізованих технічних навичок для розробки тестових сценаріїв і підтримки тестових скриптів.

Комбіноване тестування (Hybrid Testing) є підходом, який поєднує в собі переваги як ручного, так і автоматизованого тестування. Цей метод дозволяє використовувати найкращі практики з обох підходів, що забезпечує більшу гнучкість і ефективність тестування. Завдяки комбінованому підходу можна зменшити витрати на тестування, оскільки автоматизація забезпечує швидке виконання повторюваних завдань, а ручне тестування дозволяє більш детально перевіряти складні або нестандартні сценарії. Таким чином, комбіноване тестування дозволяє досягти найкращих результатів, забезпечуючи високу якість продукту та зручність для користувачів.

Комбіноване тестування є потужним підходом до перевірки програмного забезпечення, який поєднує як ручні, так і автоматизовані методи тестування. Цей метод особливо ефективний, коли потрібно адаптувати стратегію тестування до специфічних вимог проекту, зберігаючи при цьому високу ефективність і якість продукту. Наприклад, в тестуванні веб-додатків часто автоматизують перевірку бази даних або функціональності веб-форм, що дозволяє значно зекономити час. Водночас ручне тестування залишається критично важливим для оцінки інтерфейсу користувача, його зручності та адаптивності до різних пристроїв. У випадку мобільних додатків автоматизація використовується для перевірки основних функцій, таких як взаємодія з сервером або робота з базою даних, а ручне тестування необхідне для оцінки поведінки програми за різних умов, таких як зміна мережі чи наявність різних версій операційних систем.

Комбіноване тестування також знаходить широке застосування в таких складних системах, як фінансові додатки. Автоматизація тестування регресії дозволяє забезпечити точність і стабільність обробки транзакцій, в той час як ручне тестування є незамінним для перевірки логіки обліку та зручності взаємодії з

користувачем. Це дозволяє отримати найкращий результат, поєднуючи переваги обох підходів.

Однією з головних переваг комбінованого тестування є його гнучкість у виборі методів тестування, що дозволяє командам ефективно використовувати ресурси та максимально швидко виявляти дефекти. Крім того, такий підхід дозволяє знизити витрати та час на тестування, зберігаючи високу якість продукту. Поєднання різних методів також сприяє всебічній перевірці програмного забезпечення, що значно покращує його якість та забезпечує задоволення кінцевих користувачів.

Проте комбіноване тестування має й певні недоліки. Зокрема, управління різними підходами може бути складним, особливо якщо команди не мають достатньої досвідченості для синхронізації ручного та автоматизованого тестування. Крім того, для успішної реалізації комбінованого тестування потрібні фахівці з високим рівнем кваліфікації, здатні ефективно працювати з обома методами.

Тестування за рівнями є важливою частиною стратегії забезпечення якості програмного забезпечення. Цей підхід передбачає проведення тестів на різних етапах розробки продукту, кожен з яких фокусується на виявленні дефектів на відповідному рівні системи. Перше, що перевіряється, — це модулі або компоненти програми на етапі модульного тестування, яке має на меті забезпечити стабільність окремих частин системи. Після цього проводиться інтеграційне тестування, яке оцінює взаємодію між модулями, щоб забезпечити коректність і сумісність компонентів програми.

Системне тестування фокусується на перевірці всієї програми в цілому, щоб оцінити її відповідність вимогам і специфікаціям. Тут тестувальники оцінюють функціональність, безпеку, продуктивність та інші аспекти продукту. Після цього проводиться приймальне тестування, яке дозволяє підтвердити готовність продукту до впровадження та оцінити, чи відповідає він вимогам кінцевих користувачів. Це є останнім етапом перед релізом продукту.

Основною перевагою тестування за рівнями є його здатність структуровано і системно виявляти дефекти на різних етапах розробки, що знижує ймовірність появи серйозних помилок на пізніх етапах. Такий підхід дозволяє зосередитися на кожному етапі окремо, що значно покращує ефективність тестування та якість кінцевого продукту. Тестування за рівнями також допомагає організувати роботу команди і забезпечити чітку структуру тестового процесу, що в результаті веде до зниження ризиків і підвищення загальної стабільності та надійності програмного забезпечення.

Тестування на основі ризиків (Risk-Based Testing, RBT) є стратегією, що зосереджується на ідентифікації, оцінці та пріоритезації ризиків у процесі

тестування програмного забезпечення. Основна мета цього підходу — оптимізувати ресурси, визначивши найбільш критичні та ризиковані аспекти системи, які мають найбільший потенційний вплив на бізнес. Враховуючи важливість виявлення та усунення високих ризиків до випуску продукту, тестувальники можуть сконцентрувати свої зусилля на тих частинах системи, які найбільше потребують перевірки, тим самим знижуючи ймовірність появи серйозних дефектів у фінальній версії програми.

Однією з основних переваг тестування на основі ризиків є ефективне використання обмежених ресурсів, що дозволяє зосередитись на найважливіших частинах програмного забезпечення. Це не тільки підвищує якість кінцевого продукту, але й значно зменшує ймовірність появи критичних помилок, які можуть призвести до серйозних проблем після випуску. Також, завдяки своєчасному виявленню та усуненню основних ризиків, ймовірність задоволення замовника значно зростає, оскільки кінцевий продукт виявляється більш стабільним і функціональним.

Проте, цей підхід не позбавлений недоліків. Наприклад, ідентифікація та оцінка ризиків можуть бути досить складними і вимагати значних зусиль з боку тестувальників. Оцінка ризиків також може бути суб'єктивною, що призводить до можливих упущень важливих аспектів або неточностей у визначенні критичних ризиків. Тому для ефективного застосування цього методу необхідна глибока експертиза та ретельний підхід до аналізу.

Контекстуальне тестування (Context-Driven Testing) є ще одним підходом, що акцентує увагу на специфіці тестувального контексту. Згідно з цією методологією, не існує єдиного "правильного" способу тестування, оскільки ефективність тестувального процесу безпосередньо залежить від унікальних особливостей проекту, команди, продукту та кінцевих користувачів. Це означає, що процес тестування потрібно адаптувати до конкретних умов і потреб, а не слідувати за стандартними методами, які можуть не підходити для кожного окремого випадку.

Основна ідея контекстуального тестування полягає в тому, що тестувальники повинні враховувати всі аспекти контексту — тип продукту, його аудиторію, вимоги замовника, обмеження часу і бюджету, а також використовувані технології. Це дозволяє створити гнучкий підхід до тестування, який буде найбільш ефективним саме для цього конкретного проекту. Гнучкість є основною перевагою цього методу, оскільки тестувальники можуть швидко адаптувати свої методи та техніки на основі нових даних або результатів, що дозволяє зосередитись на найбільш важливих аспектах системи.

Ще однією важливою рисою контекстуального тестування є те, що тестувальники повинні чітко розуміти цілі тестування та зосереджуватися на них. Це може бути перевірка відповідності вимогам замовника, оцінка користувацького

досвіду чи виявлення дефектів. Оскільки цей підхід передбачає активну співпрацю між різними учасниками процесу (розробниками, тестувальниками, бізнес-аналітиками), він сприяє більш глибокому розумінню продукту та його потреб, що підвищує ймовірність виявлення критичних дефектів, які можуть бути упущені при використанні стандартних підходів.

Основні етапи контекстуального тестування включають початковий аналіз контексту, розробку стратегії тестування, виконання тестів відповідно до цієї стратегії, а також оцінку результатів, щоб виявити дефекти і надати рекомендації для вдосконалення продукту. Цей підхід дозволяє тестувальникам не лише перевіряти конкретні функціональні аспекти, а й враховувати більш глобальні вимоги, що дає можливість зробити тестування більш ефективним і зосередженим на важливих проблемах проекту.

Попри свої численні переваги, контекстуальне тестування має і певні недоліки. Оскільки воно вимагає високої кваліфікації тестувальників і глибокого розуміння проекту, цей підхід може бути складним у реалізації. Крім того, оскільки контекстуальне тестування базується на індивідуальних підходах, це може ускладнити порівняння результатів між різними проектами або командами, оскільки відсутня чітка стандартизація процесу.

Щодо тестування безпеки, це один з важливих аспектів тестування програмного забезпечення, який полягає в застосуванні різноманітних методів і технік для виявлення вразливостей і потенційних загроз у програмному забезпеченні та системах. Тестування безпеки є важливим для забезпечення конфіденційності, цілісності та доступності даних, а також для запобігання несанкціонованому доступу і атакам на систему. Існують різні методи тестування безпеки, і кожен з них має свої особливості в залежності від цілей, таких як виявлення вразливих місць у програмному забезпеченні, перевірка на стійкість до атак або тестування на можливість несанкціонованого доступу.

Методи тестування безпеки є основними інструментами для забезпечення цілісності, конфіденційності та доступності даних у програмному забезпеченні та системах. Вони сприяють виявленню вразливостей і потенційних загроз, що можуть бути використані для несанкціонованого доступу або атак. Існує кілька методів, кожен з яких має свої особливості і підходи до тестування безпеки.

Одним із важливих методів є динамічне тестування, яке передбачає перевірку програмного забезпечення під час його виконання. Цей процес дозволяє тестувальникам спостерігати за тим, як система взаємодіє з користувачами, іншими системами і даними, виявляючи можливі вразливості, які можуть виникнути саме в процесі роботи програми. Наприклад, такі загрози, як SQL-ін'єкції, атаки на сесії або витоки даних, можуть бути виявлені саме під час тестування в реальних умовах.

Цей метод є надзвичайно корисним для виявлення проблем, які можуть проявлятися лише під час виконання програмного коду.

У протилежність йому існує статичне тестування, яке полягає в аналізі коду без його виконання. Для цього використовуються спеціалізовані інструменти для статичного аналізу, які перевіряють програмний код на наявність вразливостей, помилок стилю програмування або невідповідностей з вимогами безпеки. Статичне тестування дозволяє виявити потенційно небезпечні аспекти на ранніх етапах розробки, такі як незахищене зберігання паролів чи неправильно налаштовані механізми аутентифікації, що можуть стати серйозними проблемами в кінцевому продукті.

Тестування на проникнення є ще одним важливим методом, коли тестувальники намагаються виявити та експлуатувати вразливості системи, імітуючи реальні атаки з боку зловмисників. Такий підхід дозволяє не лише виявити уразливості, але й оцінити їх потенційну небезпеку для системи в умовах реальних загроз. Це тестування дозволяє зрозуміти, як система може реагувати на спроби несанкціонованого доступу або інші атаки.

Ще одним методом є тестування уразливостей, яке передбачає використання автоматизованих інструментів для сканування системи на наявність відомих вразливостей. Цей підхід дає змогу швидко і ефективно виявити потенційні проблеми, без необхідності глибокого аналізу коду, і надати необхідну інформацію для подальших заходів щодо забезпечення безпеки.

Тестування конфігурації зосереджено на перевірці налаштувань програмного та апаратного забезпечення відповідно до найкращих практик безпеки. Це включає перевірку налаштувань серверів, баз даних, мережевих пристроїв тощо, для виявлення можливих недоліків у конфігураціях, які можуть призвести до зниження рівня безпеки. Тестування конфігурації допомагає переконатися, що всі елементи системи налаштовані відповідно до вимог безпеки, що знижує ризик несанкціонованого доступу або зломів.

Тестування доступності стосується перевірки механізмів контролю доступу до системи. Важливо, щоб система забезпечувала правильну авторизацію та аутентифікацію користувачів, щоб уникнути проблем, таких як недостатня аутентифікація або помилки в логіці доступу. Цей метод дозволяє виявити дефекти в управлінні доступом, які можуть бути використані зловмисниками для доступу до чутливої інформації.

Аудит безпеки включає всебічний огляд і аналіз політик безпеки, процедур, інфраструктури та програмного коду для виявлення вразливостей та слабких місць. Аудит безпеки може проводитися як внутрішніми, так і зовнішніми фахівцями і дозволяє оцінити загальний стан безпеки системи. Він допомагає виявити ризики

та пропонує рекомендації щодо їх усунення, що забезпечує покращення загальної безпеки проекту.

І, нарешті, тестування на відповідність стандартам спрямоване на перевірку системи на відповідність міжнародним стандартам безпеки, таким як PCI DSS, HIPAA, GDPR тощо. Це включає в себе перевірку процедур обробки даних, механізмів захисту даних та управління доступом. Тестування на відповідність стандартам є необхідним для забезпечення того, щоб система відповідала всім юридичним і регуляторним вимогам, що стосуються безпеки даних, і для запобігання юридичним проблемам, які можуть виникнути у разі порушення цих вимог.

Таким чином, ці методи тестування безпеки сприяють створенню більш безпечних і надійних систем, забезпечуючи захист від численних загроз і ризиків, що можуть виникнути на різних етапах життєвого циклу програмного забезпечення.

4.3 Інструменти тестування

Інструменти тестування є важливими програмними засобами, які використовуються для підтримки та автоматизації процесів тестування програмного забезпечення. Вони значно полегшують роботу тестувальників, підвищуючи ефективність, точність і повторюваність тестування, одночасно зменшуючи трудозатрати та час, необхідний для виконання тестових сценаріїв. В залежності від їх призначення, ці інструменти можна поділити на кілька категорій.

Однією з основних категорій є інструменти для автоматизації тестування, які дозволяють значно прискорити процес виконання тестів. Вони допомагають автоматизувати запуск тестових сценаріїв, зменшуючи кількість ручної праці і дозволяючи виконувати тести швидше та точніше. Наприклад, Selenium є одним з найбільш відомих інструментів для автоматизації тестування веб-додатків, що підтримує різні браузери та мови програмування. Appium, у свою чергу, орієнтований на тестування мобільних додатків на таких популярних платформах, як iOS та Android. Інший потужний інструмент, Cypress, спеціалізується на автоматизації тестування фронтенд-додатків, забезпечуючи високу швидкість виконання та зручний інтерфейс для тестувальників.

Малюнок 4.3 – Інструменти

Для тестування продуктивності існує ряд інструментів, які дозволяють перевіряти, як система справляється з навантаженням. Одним із таких є JMeter, що активно використовується для тестування навантаження веб-додатків та баз даних, дозволяючи аналізувати їхню продуктивність при великих обсягах запитів. LoadRunner є ще одним потужним комерційним інструментом для моделювання навантаження та аналізу продуктивності системи. Gatling також спеціалізується на тестуванні навантаження, але використовує мову Scala для створення тестів і підтримує високу швидкість тестування.

Інструменти для тестування безпеки використовуються для виявлення вразливостей та потенційних загроз у системах. Наприклад, Burp Suite є популярним інструментом для тестування безпеки веб-додатків, який включає широкий спектр функцій для сканування та аналізу безпеки. OWASP ZAP (Zed Attack Proxy) - безкоштовний інструмент, що пропонує простий у використанні інтерфейс для тестування безпеки веб-додатків. Також існує Nessus, який є сканером вразливостей, здатним виявляти потенційні загрози в системах безпеки.

Для управління тестуванням є інструменти, які допомагають організувати весь процес тестування, планувати задачі, відслідковувати дефекти та результати тестування. Jira є популярним інструментом для управління проектами, який також використовується для відстеження дефектів і управління тестовими завданнями. TestRail дозволяє планувати і організовувати тестування, а також відслідковувати його результати, забезпечуючи зручний інтерфейс для тестувальників. Quality Center (ALM) від Micro Focus є потужним інструментом для управління життєвим циклом тестування, що охоплює всі етапи розробки.

Існують також інструменти для статичного аналізу коду, які дозволяють виявляти помилки в програмному коді без його виконання. SonarQube є однією з найпопулярніших платформ для аналізу якості коду, яка дозволяє виявляти

помилки, вразливості та проблеми з підтримкою. Для JavaScript-коду ідеально підходить ESLint, який допомагає виявляти проблеми зі стилем і потенційними помилками в коді. PMD - це інструмент для аналізу Java-коду, який дозволяє виявляти недоліки та погані практики програмування.

Для тестування інтерфейсу існують інструменти, які дозволяють перевірити зручність і доступність користувацьких інтерфейсів. Accessibility Insights спеціалізується на тестуванні доступності веб-додатків, допомагаючи виявляти проблеми для користувачів з обмеженими можливостями. TestComplete є потужним інструментом для автоматизації тестування інтерфейсу, що підтримує різні платформи та мови програмування.

Ще однією важливою категорією є інструменти для тестування API, що дозволяють перевіряти функціональність та продуктивність інтерфейсів прикладних програм. Postman є одним з найбільш відомих інструментів для тестування API, дозволяючи тестувальникам легко створювати та відправляти запити і аналізувати відповіді. SoapUI підтримує тестування як REST, так і SOAP API, а RestAssured є бібліотекою для автоматизації тестування RESTful веб-сервісів на Java.

Загалом, існує широкий спектр інструментів тестування, які покривають всі етапи процесу тестування програмного забезпечення. Вибір інструменту залежить від специфіки тестування, вимог проекту та технічних характеристик програмного забезпечення, що тестується. Вони допомагають підвищити якість і швидкість тестування, знижуючи при цьому ймовірність помилок і недоліків у фінальному продукті.

4.4 Підтримка програми

Підтримка програмного забезпечення — це ключовий процес, що включає забезпечення постійної допомоги, обслуговування та оновлення програм після їх випуску. Вона має на меті забезпечення стабільної роботи програм, коригування виявлених помилок, удосконалення функціональності та адаптацію до змінюваних вимог користувачів і технологічного середовища. Цей етап є невід'ємною частиною життєвого циклу програмного забезпечення і охоплює широкий спектр заходів, спрямованих на покращення якості та актуальності продукту протягом його існування.

Один із основних аспектів підтримки програмного забезпечення — це виправлення помилок. Коли в програмі виявляються помилки, важливо оперативно реагувати на них. Це може включати технічну підтримку, коли користувачам

надається допомога для вирішення проблем, з якими вони стикаються при використанні програми, а також випуск оновлень програмного забезпечення, таких як патчі або нові версії, що усувають виявлені помилки або вразливості. Такі коригування є необхідними для забезпечення належної функціональності та безпеки продукту.

Підтримка також передбачає оновлення та вдосконалення програмного забезпечення, що включає додавання нових можливостей та удосконалення вже існуючих функцій. Оновлення можуть бути спрямовані на покращення продуктивності програми, оптимізацію її коду або архітектури, що, у свою чергу, підвищує швидкість роботи та ефективність. Ці зміни можуть також відображати зміни в потребах користувачів, що дозволяє продукту залишатися конкурентоспроможним і актуальним на ринку.

Малюнок 4.4 – Підтримка програми

Оскільки технології постійно еволюціонують, важливим аспектом підтримки є адаптація програми до нових технологій. Це включає забезпечення сумісності з новими операційними системами, веб-браузерами та апаратним забезпеченням, що з'являються на ринку. Крім того, програма може потребувати інтеграції з новими сервісами та платформами, що також є важливим аспектом підтримки, аби забезпечити її інтеграцію з іншими продуктами або API.

Управління документацією є ще одним важливим елементом підтримки. Це не лише актуалізація технічної документації, користувацьких посібників та навчальних матеріалів, але й створення звітів, які фіксують внесені зміни в програмне забезпечення, виправлення помилок і додавання нових функцій. Така документація допомагає користувачам швидко зорієнтуватися в нових можливостях програми і полегшує процес її використання.

Ще одним критичним етапом підтримки є тестування та перевірка змін перед їх випуском. Регресійне тестування важливе для того, щоб після внесення змін у програму перевірити, чи не були порушені інші частини її функціональності. Оцінка нових функцій є необхідною для перевірки їх роботи і відповідності вимогам, що ставляться до програми. Це дозволяє уникнути помилок після впровадження оновлень.

Збір зворотного зв’язку від користувачів є важливою частиною підтримки програмного забезпечення, оскільки дає змогу зрозуміти, як користувачі сприймають продукт, які проблеми вони мають і що можна поліпшити. Це можна здійснювати через опитування, фокус-групи або через моніторинг запитів у службі підтримки. Аналіз цих відгуків дозволяє виявити основні проблеми, що виникають у користувачів, і визначити пріоритети для виправлення або вдосконалення функціоналу.

Таким чином, підтримка програмного забезпечення є надзвичайно важливим процесом, який включає виправлення помилок, оновлення функцій, адаптацію до нових технологій, вдосконалення документації, тестування змін і збирання зворотного зв'язку від користувачів. Всі ці заходи сприяють тому, щоб програма залишалася актуальною, безпечною та ефективною в процесі її використання.

ВИСНОВКИ

В процесі виконання дипломної роботи на тему "Автоматизована система для фінансово-облікових операцій на базі веб-додатку" було створено комплексне програмне рішення, яке відповідає сучасним вимогам автоматизації фінансових та облікових процесів в організаціях. Розроблене програмне забезпечення має на меті значне підвищення ефективності ведення обліку, мінімізацію кількості помилок, а також поліпшення зручності та доступності для кінцевих користувачів, що дозволяє оптимізувати роботу з фінансовою інформацією.

В ході розробки системи була здійснена детальна оцінка існуючих рішень на ринку фінансових та облікових програм, що допомогло виявити основні функціональні вимоги до системи. Серед них — ефективна обробка фінансових транзакцій, генерація звітності та інтеграція з іншими зовнішніми системами. Одним з основних аспектів проекту стало забезпечення високого рівня безпеки даних, що є надзвичайно важливим при роботі з фінансовою інформацією. Для цього в системі були реалізовані сучасні методи шифрування та аутентифікації, що забезпечують надійний захист інформації від несанкціонованого доступу.

Тестування системи підтвердило її працездатність і надійність, а також виявило декілька напрямків, у яких можна провести подальші вдосконалення для підвищення ефективності роботи. Впровадження автоматизованої системи дозволяє організаціям значно зменшити витрати на обробку фінансових операцій, поліпшити точність звітності та забезпечити оперативний доступ до необхідної фінансової інформації. Крім того, система дозволяє скоротити час, витрачений на виконання рутинних завдань, що, в свою чергу, підвищує загальну продуктивність праці.

Отже, результати виконаної роботи стали основою для створення ефективного інструменту, який не лише відповідає вимогам сучасного бізнесу, але й має великий потенціал для подальшої адаптації та розширення функціоналу. В майбутньому, з метою досягнення максимальної ефективності, слід провести додаткові дослідження щодо практичного впровадження цієї системи в реальних умовах. Також важливим є вивчення можливостей її інтеграції з іншими корпоративними системами, що дозволить значно розширити її функціональність і забезпечити більш ефективне управління фінансовими потоками в організації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Основи автоматизації бухгалтерського обліку. Джерело: (https://stud.com.ua/22861/audit_ta_buhoblik/osnovi_avtomatizatsiyi_buhgalterskogo_obliku)

2. Автоматизована форма бухгалтерського обліку. Джерело: (https://pidru4niki.com/10180406/buhgalterskiy_oblik_ta_audit/avtomatizovana_forma_buhgalterskogo_obliku)

3. Організація автоматизованого обліку фінансово-розрахункових операцій. Джерело: (https://buklib.net/books/27801/)

4. Загальна характеристика та класифікація інформаційних систем обліку. Джерело: (https://buklib.net/books/22551/)

5. Автоматизована інформаційна облікова система. Джерело: (https://pidru4niki.com/1554081264614/buhgalterskiy_oblik_ta_audit/avtomatizovana_informatsiyna_oblikova_sistema)

6. Механізовані й автоматизовані форми ведення бухгалтерського обліку. Джерело:(https://pidru4niki.com/69381/buhgalterskiy_oblik_ta_audit/mehanizovani_avtomatizovani_formi_vedennya_buhgalterskogo_obliku)

7. Етапи впровадження автоматизованої системи управлінського обліку. Джерело:(https://science.lpnu.ua/sites/default/files/journalpaper/2023/dec/32695/menedzhment223maket-152-159.pdf)

8. Автоматизація бухгалтерського та податкового обліку. Джерело: [magazine.faaf.org.ua](https://magazine.faaf.org.ua/avtomatizaciya-buhgalterskogo-ta-podatkovogo-obliku.html)

9. Автоматизація обліку в сучасних умовах. Джерело: [evnuir.vnu.edu.ua](https://evnuir.vnu.edu.ua/bitstream/123456789/8522/1/automation.pdf)

10. Розв’язування обліково-аналітичних завдань у системі «ISpro» на підприємстві.Джерело:(https://dspace.wunu.edu.ua/bitstream/316497/44366/1/%D0%9B%D1%83%D0%B1%20%D0%9D.%D0%9E..pdf)

11. Автоматизація бухгалтерського обліку. Джерело: (https://stud.com.ua/10780/audit_ta_buhoblik/avtomatizatsiya_buhgalterskogo_obliku)

12. Автоматизація бухгалтерського обліку: сучасні тенденції та перспективи. Джерело: [nauka.zinet.info](http://www.nauka.zinet.info/9/tkachuk.php)

13. Впровадження інформаційних технологій в облікову систему підприємства. Джерело: (https://economyandsociety.in.ua/journals/10_ukr/30.pdf)

14. Андрій Креневич – Алгоритми і структури даних. Джерело: Підручник, присвячений вивченню алгоритмів і структур даних у програмуванні.

15. Алгоритми та структури даних – КПІ ім. Ігоря Сікорського. Джерело: Навчальний посібник, що охоплює питання рекурсії, роботи з масивами, сортування та пошуку, а також абстрактні структури даних.

16. Алгоритми і структури даних – Тернопільський національний економічний університет. Джерело: Навчальний посібник, що надає короткий виклад основних понять про структури даних та алгоритми роботи з ними.

17. Структури даних та алгоритми – Чернівецький національний університет. Джерело: Підручник, присвячений теоретичним та практичним аспектам структур даних та алгоритмів, з типовими прикладами.

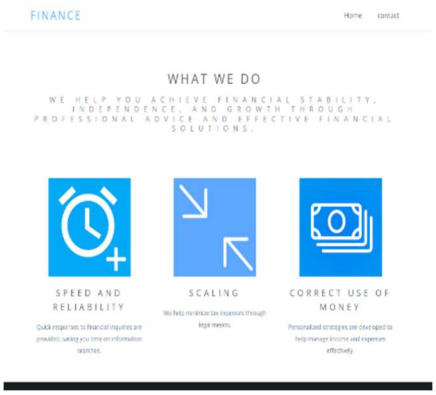
18. Алгоритми та структури даних – Харківський національний економічний університет. Джерело: Навчальний посібник, що розглядає формалізацію понять "алгоритм" та "структура даних", а також загальні принципи алгоритмізації.

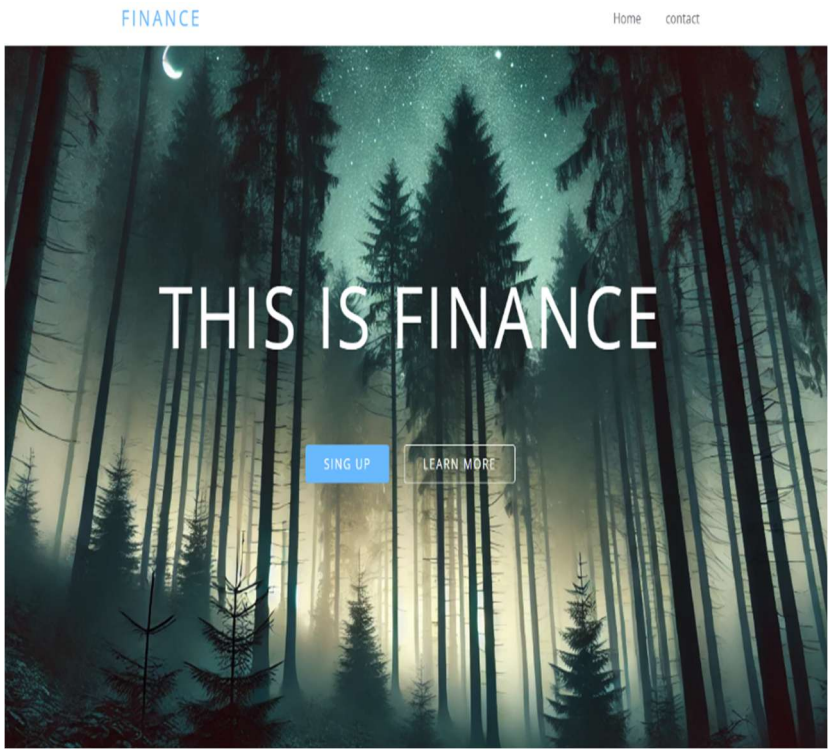
19. Алгоритми і структури даних – Львівська політехніка. Джерело: Навчальний посібник, що охоплює основні поняття та методи роботи з алгоритмами і структурами даних.

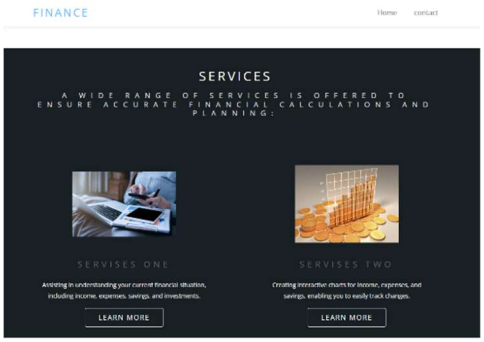
20. Алгоритми і структури даних – Тернопільський національний економічний університет. Джерело: Лекційний матеріал, що розглядає алгоритми як процедурні елементи в структурі програми та структури даних, які застосовуються в алгоритмах.

Додаток А

Дизайн і послуги які пропонує сайту FINANCE

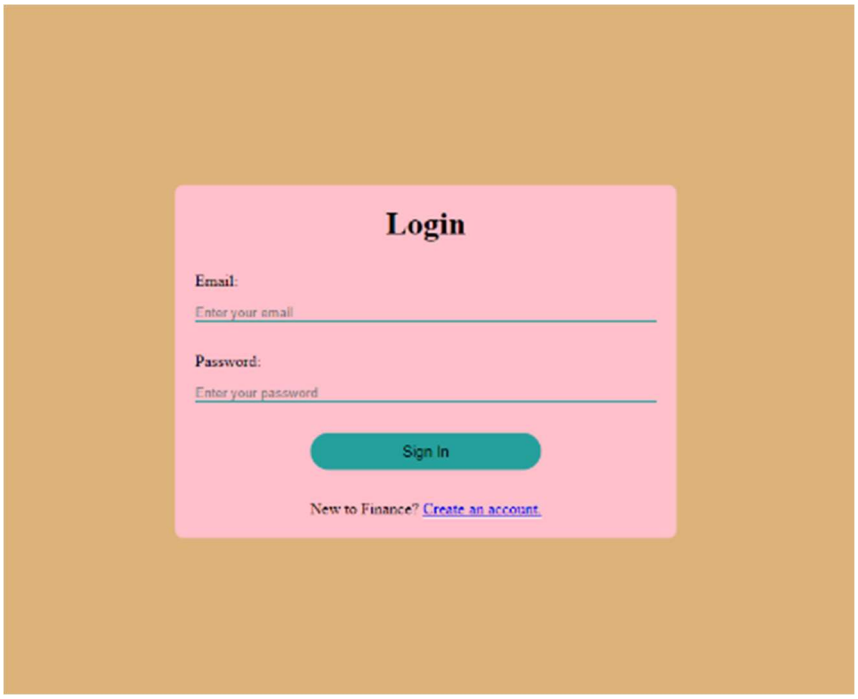






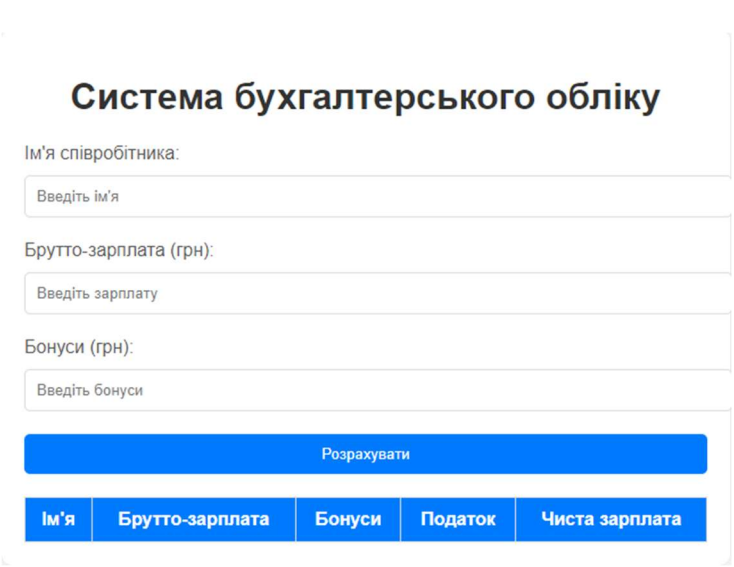
Додаток Б

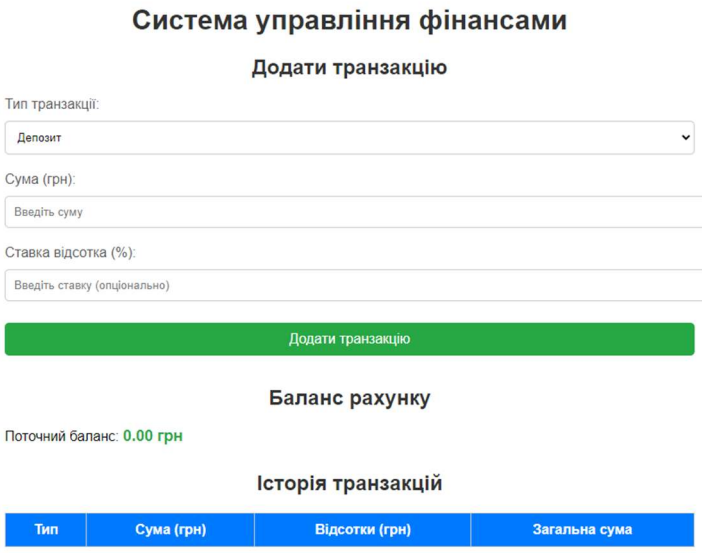
Реєстраційне поле введення

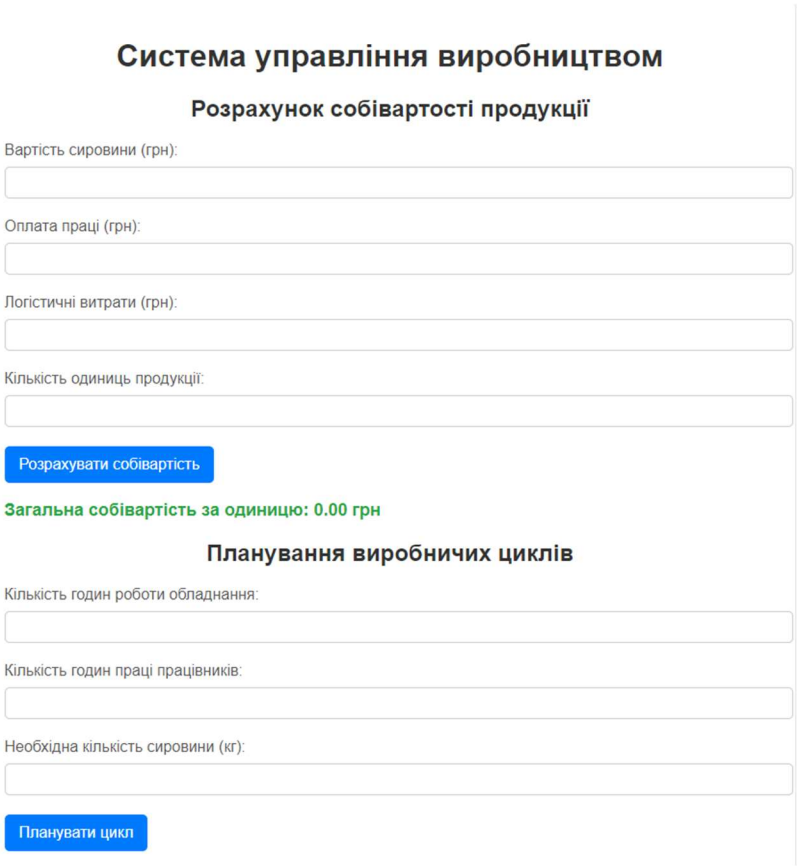


Додаток В

Функціонал який пропонує сайт







Додаток Г

Порівняльна таблиця HTML, CSS та JavaScript

Фреймворк	Призначення	Функції	Основні особливості
HTML	Мова розмітки для створення структури вебсторінок.	Створення каркасу сторінки, додавання тексту, зображень, відео, форм і посилань.	Визначає елементи сторінки за допомогою тегів. Оновлена версія HTML5 підтримує мультимедіа (відео, аудіо) та семантичні елементи (<header>, <footer>).
CSS	Мова стилів для візуального оформлення вебсторінок.	Задає кольори, шрифти, розміри, відступи, вирівнювання, адаптивність, анімацію.	Підтримує каскадність і спадковість. Сучасні інструменти, як-от Flexbox і Grid, полегшують створення адаптивного дизайну.
JavaScript	Мова програмування для додавання динамічності та інтерактивності вебсторінкам.	Реалізація інтерактивних елементів, динамічного оновлення контенту, роботи з API, валідації форм, створення мультимедіа та анімацій.	Використовує DOM для роботи з HTML-структурою. Підтримує асинхронне програмування через AJAX, Promise та async/await.

Додаток Г

Програмний код

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Finans</title>
    <!-- Підключення файлів -->
    <link rel="stylesheet" href="reset.css">
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header class="header ">
        <div class="container header__conteiner">
            <h1 class="logo">Finance</h1>

            <nav class="header_nav">
            <ul>
                <li><a href="./index.html">Home</a></li>
                <li><a href="#social">contact</a></li>
            </ul>
        </nav>

        </div>
    </header>

    <main class="main ">
        <section class="welcome">
            <div class="container">
                <h2 class="welcome__heading"> This is Finance</h2>

                <div class="welcome__links">
                    <a href="./regist/signIn.html" class="link-primary">sing up</a>
                    <a href="#!" class="link">Learn more</a>
                </div>
            </div>
        </section>



        <section class="about common-section ">
            <div class="container">
                <div class="title-wrapper">
                    <h3 class="title">WHAT WE DO</h3>
                    <p class="subtitle">We help you achieve financial stability, independence, and growth through professional advice and effective financial solutions.</p>
                </div>

                <div class="card-wapper">
```

```html
    <main class="main ">
        <section class="about common-section ">
            <div class="container">
                <p class="subtitle">We help you achieve financial stability, independence, and growth through professional advice and effective financial s
                </div>

                <div class="card-wapper">
                    <div class="card">
                    <img src="img/images1.png"  alt="icon">
                        <h4>speed and reliability</h4>
                        <p>Quick responses to financial inquiries are provided, saving you time on information searches.</p>
                    </div>


                        <div class="card">
                        <img src="img/images2.png"  alt="icon">
                            <h4>Scaling</h4>
                            <p>We help minimize tax expenses through legal means.</p>
                        </div>

                            <div class="card">
                            <img src="img/images3.png"  alt="icon">
                                <h4>Correct use of money</h4>
                                <p>Personalized strategies are developed to help manage income and expenses effectively.</p>
                            </div>

                </div>


                </div>
                <section class="services about common-section--dark " >
                    <div class="container">
                        <div class="title-wrapper">
                            <h3 class="title">Services</h3>
                            <p class="subtitle">A wide range of services is offered to ensure accurate financial calculations and planning:</p>
                        </div>

                        <div class="card-wapper">
                            <div class="card">
                            <img src="img/services1.png" alt="img">
                                <h4>SERVISES ONE</h4>
                                <p>Assisting in understanding your current financial situation, including income, expenses, savings, and investments.</p>
                                <a href="./calculation/editor.html" class="link">Learn more</a>
                            </div>
                                <div class="card">
                                <img src="img/services2.jpg"  alt="img">
                                    <h4>SERVISES TWo</h4>
```

```
                nulla, ut commodo dlam libero vitae erat</p>
            </div>
            <div class="footer__links">
                <h4>USEFUL LINKS</h4>
                <ul>
                    <li><a href="#!">Phasellus provida semper nisi</a></li>
                    <li><a href="#!">Suspendisse nisl elit</a></li>
                    <li><a href="#!">Dellentesqut habitant morbi</a></li>
                    <li><a href="#!">Etiam sollicitudin ipsum</a></li>
                </ul>
            </div>
            <div class="footer__social" id="social">
                <h4>SOCIAL</h4>
                <ul>
                    <li>
                        <img src="img/twiter.jpg" width="30" height="30" alt="twiter">
                        <a href="#!">Twiter</a>
                    </li>
                    <li>
                        <img src="img/соціальна іконка фейсбук.jpg" width="30" height="30" alt="facebook">
                        <a href="#!">Facebook</a>
                    </li>
                    <li>
                        <img src="img/Pinterest.jpg" width="30" height="30" alt="pinterest">
                        <a href="#!">Pinterest</a>
                    </li>
                    <li>
                        <img src="img/google1.png" width="30" height="30" alt="google">
                        <a href="#!">Google</a>
                    </li>
                    <li>
                        <img src="img/webflow.png" width="30" height="30" alt="webflow">
                        <a href="#!">Webflow</a>
                    </li>
                </ul>
            </div>

        </div>
    </div>

    <div class="footer__copy">
        <div class="container">
            <p>Copyright Velocity Inc. Made in Webflow.</p>
```

```
                    <h4>SERVISES TWo</h4>
                    <p>Creating interactive charts for income, expenses, and savings, enabling you to easily track changes.</p>
                    <a href="#!" class="link">Learn more</a>
                </div>


            </div>


    </section>


    </section>
</main>

<footer class="footer">
    <div class="footer__desc">
        <div class="container footer__desc-container">
            <div class="footer__about">
                <h4>ABOUT VELOCITY</h4>
                <p>Lorem ipsum dolor sit amet, consectetur
                    adipiscing elit. Suspendisse varius enim in
                    eros elementum tristique. Duis cursus, mi quis viverra ornare, eros dolor interdum
                    nulla, ut commodo dlam libero vitae erat</p>
            </div>
            <div class="footer__links">
                <h4>USEFUL LINKS</h4>
                <ul>
                    <li><a href="#!">Phasellus provida semper nisi</a></li>
                    <li><a href="#!">Suspendisse nisl elit</a></li>
                    <li><a href="#!">Dellentesqut habitant morbi</a></li>
                    <li><a href="#!">Etiam sollicitudin ipsum</a></li>
                </ul>
            </div>
            <div class="footer__social" id="social">
                <h4>SOCIAL</h4>
                <ul>
                    <li>
                        <img src="img/twiter.jpg" width="30" height="30" alt="twiter">
                        <a href="#!">Twiter</a>
                    </li>
                    <li>
```

```css
@import url('https://fonts.googleapis.com/css2?family=Open+Sans&display=swap');

/* Сумісні стилі */
*{
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
body{
    font-family: 'Open Sans', sans-serif;
    font-weight: 400;
    font-size: 14px;
}

.container{
    max-width: 970px;
    padding: 0 15px;
    margin: 0 auto;
    margin-bottom: 30px;

}


p{
    color: #6a859c;
    line-height: 1.8;
}

.title-wrapper {
    margin-bottom: 60px;
}

.hide {
    display: none  !important;
}

.title {
    margin-bottom: 17px;
    padding-top: 5%;
    font-size: 30px;
    line-height: 1.2;
    letter-spacing: 5px;
    text-transform: uppercase;
    color: #676770;
    }

    .subtitle {
        font-size: 17px;
```

```css
    }

    .subtitle {
        font-size: 17px;
        line-height: 1.2;
        letter-spacing: 1cap;
        text-transform: uppercase;
        color: #8e8e9c;
    }

    .common-section {
        text-align: center;
        padding: 80px 0;
    }

    .common-section--dark {
        background-color: #192024;
    }

    .common-section--dark * {
        color: #fff;
    }

    .common-section--dark .subtitle{
        color: #e8e8e8;
    }


/* ----------HEADER-----------*/


.header{
    padding: 20px 0;
    width: 100%;

    position: fixed;
    top: 0;
    left: 0;
    z-index: 100;


    background-color: #fff;
    box-shadow: 0px 0px 8px rgba(0, 0, 0, 0.1);
}

.header__conteiner {
```

```
    margin-bottom: 0;
    display: flex;
    justify-content: space-between;
    align-items: center;
}


.logo{
    font-size: 25px;
    line-height: 25px;
    letter-spacing: 4px;
    text-transform: uppercase;
    color: #69b9ff;
}

.header_nav ul {
    display: flex;

}

.header_nav li + li {
    margin-left: 40px;

}

.header_nav a {
    font-size: 16px;
    color: #676770;
    transition: color 0.2s;
}

.header_nav a:hover{
    color: #0082f3;
}

.header_nav a:active{
    color: #0062b6;
}
```

```
/* ----------WELCOME------------*/

.welcome {
    min-height: 600px;
    height: 100vh;
    padding: 195px 0;

    display: flex;
    justify-content: center;
    align-items: center;

    color: white;
    text-align: center;

    background-color: #282936;
    background-image: url("img/head4.webp");
    background-size: cover;
    background-position: center;
}


.welcome__heading {
    margin-bottom: 100px;
    font-size: 100px;
    line-height: 1;
    letter-spacing: 4px;
    text-transform: uppercase;
}

.welcome__links > * + * {
    margin-left: 20px;
}

.link-primary {
    display: inline-block;
    padding: 11px 30px;

    font-size: 16px;
    letter-spacing: 2px;
    text-transform: uppercase;

    background-color: #69b9ff;
    border-radius: 4px;

    transition: backgroud-color 0.2s;
}
```

```css
.link-primary:hover{
    background-color: #7cc2ff;
}

.link-primary:active{
    background-color: #0f8fff;
}

.link {
    display: inline-block;
    padding: 10px 30px;

    font-size: 16px;
    letter-spacing: 2px;
    text-transform: uppercase;

    border: 1px solid white;
    border-radius: 4px;
    transition: background-color 0.1s, border-color 0.1s;
}

.link:hover {
    background-color: rgba(0, 0, 0, 0.2);
    border-color: rgba(0, 0, 0, 0.5);
}

.link:active {
    background-color: rgba(0, 0, 0, 0.5);
    border-color: rgba(0, 0, 0, 0.5);
}
```

```css
/* ----------ABOUT(WHAT WE DO)------------*/

.about {

}

.common-section {
    text-align: center;
    padding: 80px 0;
}

.card{
border: none;
}

.about .card-wapper {
    flex-direction: row;
    display: flex;
    border: none;
    justify-content: space-between;
}

.about .card {
    padding: 35px 15px 25px 15px;
    width: 300px;

    border-radius: 5px;
}


.about .card img{
    margin-bottom: 20px;
    width: 240px;
    height: 210px;
    object-fit: contain;
}

.about .card h4 {
    margin-bottom: 15px;

    font-size: 20px;
    line-height: 1.5;
    letter-spacing: 7px;
    text-transform: uppercase;
    color: #676770;
```

```css
/* ----------Services-------------*/
.services{
margin-top: 30px;
}

.services .cards-wrapper{
    display: flex;
    justify-content: space-between;
}

.services .card{
    width: 45%;
}

.services .card img{
    margin-bottom: 20px;
}

.services .card h4{
    margin-bottom: 20px;
    font-size: 20px;
    line-height: 1.5;
    letter-spacing: 7px;
    text-transform: uppercase;
}

.services .link{
    margin-top: 15px;
}
```

```css
/* ----------FOOTER------------*/

.footer{
    padding: 35px 0;
}

.footer__desc{

}

.footer__desc h4 {
    margin-bottom: 25px;

    font-size: 18px;
    line-height: 1;
    letter-spacing: 4px;
    text-transform: uppercase;
    color: #676770;
}

.footer__desc-container{
    display: flex;
    justify-content: space-between;
    align-items: flex-start;
}

.footer__desc-container > * {
    width: 30%;
}

.footer__desc a{
    font-size: 13px;
    color: #668cad;
}

.footer__desc li {
    height: 30px;
    position: relative;
}

.footer__desc li::before{
    content: "";
    width: 100%;
```

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕСТИТЕТ ІНФОРМАЦІЙНО-ТЕЛЕКОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНОЇ ІНДЖЕНЕРЕЇ

Презентація до дипломної роботи на тему:

"АТОМАТИЗОВАНА СИСТЕМА ДЛЯ ФІНАНСОВО-ОБЛІКОВИХ ОПЕРАЦІЙ НА БАЗІ ВЕБ-ДОДАТКУ"

Виконав студент групи КСДМ-62
Веремійчик Едуард Русланович

Керівник дипломної роботи
Волохін Віталій Васильович
канд, техн.наук, доцент

ОБ'ЄКТ, ПРЕДМЕТ ДОСЛІДЖЕННЯ ТА МЕТА РОБОТИ

- **Об'єкт дослідження:** розробка та впровадження автоматизованої системи для управління фінансово-обліковими процесами з використанням веб-додатку.
- **Предмет дослідження:** процеси розробки, впровадження та функціонування автоматизованої системи для фінансово-облікових операцій на основі веб-застосунку.
- **Мета роботи:** розробити веб-сайт, що автоматизуватиме процес обчислення та управління фінансами.

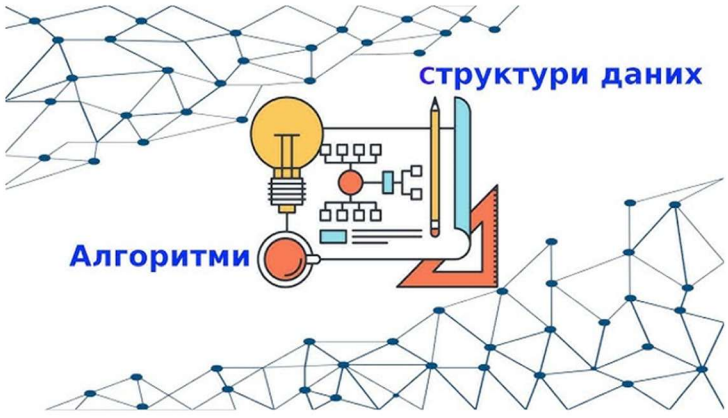
АКТУАЛЬНІСТЬ ТЕМИ

Актуальність роботи полягає в підвищенні ефективності управління фінансами в умовах сучасного бізнес-середовища. Зростаюча складність фінансових процесів та вимоги до точності обліку спонукають підприємства до автоматизації, що дозволяє зменшити ризики помилок і забезпечити швидкість виконання операцій. Використання веб-технологій забезпечує доступ до системи з будь-якого місця, що є важливим для віддаленої роботи та гнучкості бізнесу.

Алгоритми і структури даних

Структури даних — це способи організації та зберігання даних у пам'яті комп'ютера, що дозволяють алгоритмам працювати ефективніше.

Алгоритми — це чіткі послідовності кроків, що використовуються для розв'язання задач або виконання обчислень.



Архітектура системи

1. Клієнт-серверна архітектура — це модель, в якій клієнти (клієнтські програми) запитують ресурси або послуги у серверів (серверних програм), які обробляють запити та надають відповіді.

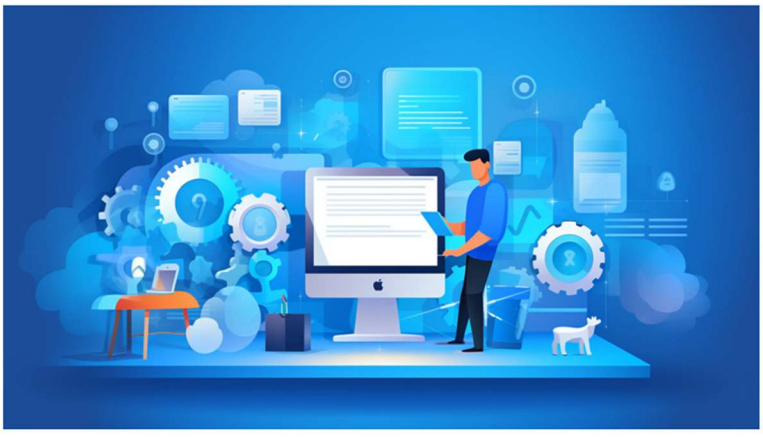
2. Модуль обробки даних — це частина програми, яка виконує маніпуляції з даними, такі як збирання, обробка, аналіз і зберігання.

3. Модуль звітності та аналітики — це частина програми, яка генерує звіти та проводить аналіз даних.

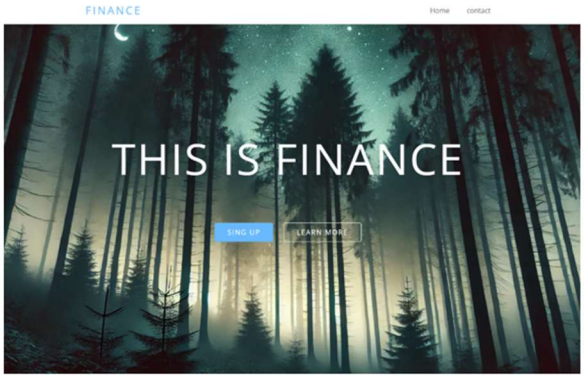
Тестування і підтримка програми

Види тестування — це різні методи перевірки програмного забезпечення для виявлення помилок і забезпечення його якості. Основні види:

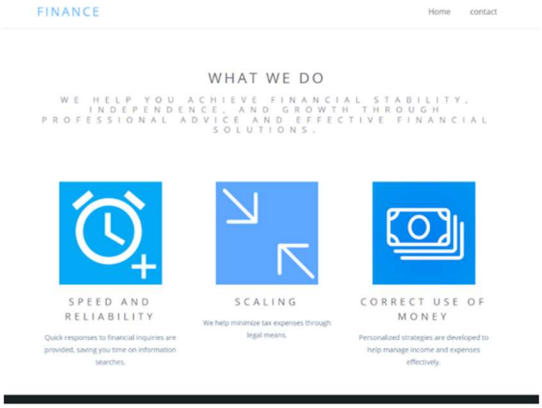
- **Функціональне тестування** — перевірка функцій програми.
- **Нефункціональне тестування** — оцінка продуктивності, безпеки, зручності тощо.
- **Модульне тестування** — тестування окремих компонентів програми.
- **Інтеграційне тестування** — перевірка взаємодії між модулями.
- **Системне тестування** — тестування всієї системи в цілому.
- **Регресійне тестування** — перевірка наявності нових помилок після змін у коді.



Дизайн сайту

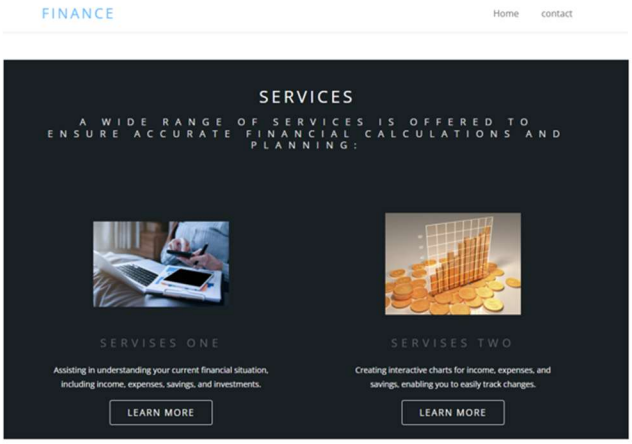


Голона сторінка FINANCE



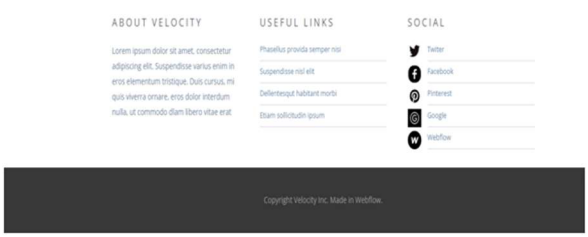
Що саме пропонує сайт

Дизайн сайту

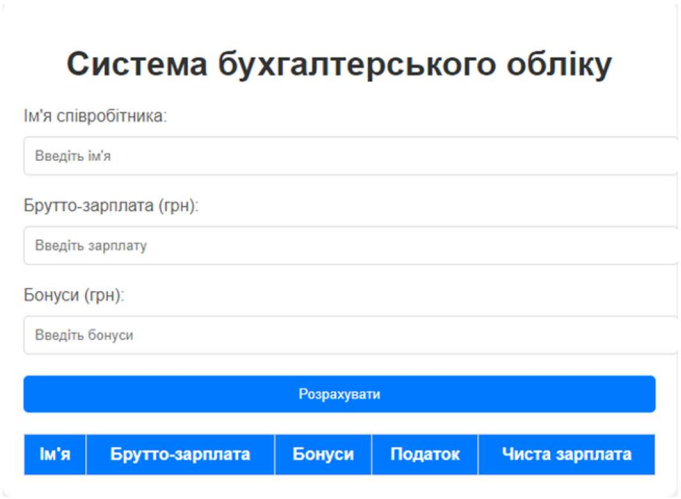


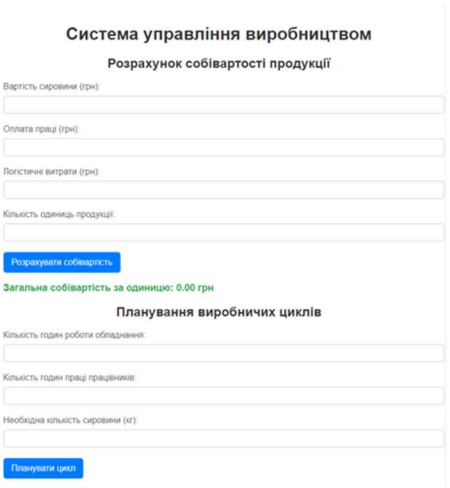
Сервіси які пропонує сайт

Соціальні мережі

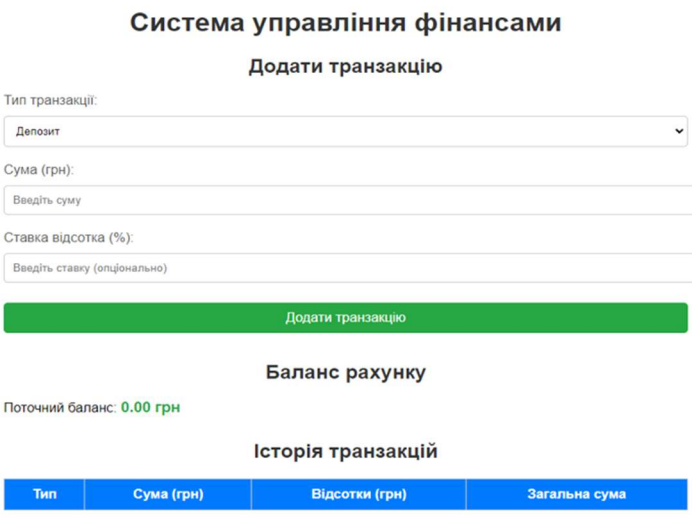


Функціонал сайту





Функціонал сайту



Висновок

В результаті виконання дипломної роботи на тему "Автоматизована система для фінансово-облікових операцій на базі веб-додатку" була розроблена комплексна система, що відповідає сучасним вимогам до автоматизації фінансових та облікових процесів. Реалізоване програмне забезпечення має на меті підвищення ефективності обліку, зменшення кількості помилок, а також покращення доступності та зручності для користувачів.

У результаті, дана дипломна робота стала основою для розробки ефективного інструменту, який відповідає вимогам сучасного бізнесу і має потенціал для подальшої адаптації та розширення функціоналу.