НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

(назва навчального закладу)

Кафедра ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Дисципліна «Технології розроблення програмного забезпечення» Курс 3 Група ІА-12 Семестр 5

ЗАВДАННЯ

на курсову роботу студента

Лещенко Денис Сергійович

(прізвище, ім’я, по батькові)

1. Тема роботи: «E-mail клієнт»
2. Строк здачі студентом закінченої роботи: 20.01.2024
3. Вихідні дані до роботи: тема «Email клієнт», опис програми та основних можливостей: E-mail клієнт повинен сприймати і коректно обробляти pop3/smtp/imap протоколи, мати функції автонастройки основних поштових провайдерів для України (gmail/ukr.net/i.ua), розділяти повідомлення на папки/категорії/важливість, зберігати чернетки незавершених повідомлень, прикріплювати і обробляти прикріплені файли.
4. Зміст розрахунково – пояснювальної записки (перелік питань, що підлягають розробці):

огляд існуючих рішень, загальний опис проекту, вимоги до застосунку проекту, сценарії використання системи, концептуальна модель системи, вибір бази даних, вибір мови програмування та середовища розробки, проектування розгортання системи, структура бази даних, архітектура системи та інструкція користувача.

Додатки:

Додаток А - діаграма класів; Діаграма Б - код проекту.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

Діаграма класів, діаграма розгортання, діаграма прецедентів, скріншоти фрагментів коду, скріншоти графічного інтерфейсу системи.

6. Дата видачі завдання: 22.09.2023

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Видача теми курсової роботи	22.09.2023	
2.	Загальний опис проекту	25.09.2023	
3.	Огляд існуючих рішень	2.10.2023	
4.	Визначення вимог до системи	9.10.2023	
5.	Визначення сценаріїв використання	16.10.2023	
6.	Концептуальна модель системи	30.10.2023	
7.	Вибір БД	7.11.2023	
8.	Вибір мови програмування та IDE	7.11.2023	
9.	Проектування розгортання системи	14.11.2023	
10.	Структура БД	17.11.2023	
11.	Визначення специфікації системи	20.11.2023	
12.	Вибір та обгрунтування патернів проектування	26.11.2023	
13.	Реалізація проекту	01.12.2023	
14.	Захист курсової		

Студент_____ _ Денис ЛЕЩЕНКО
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник_____ Валерій КОЛЕСНІК
(підпис) (Ім’я ПРІЗВИЩЕ)

«____»_____20___р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Кафедра інформаційних систем та технологій

Тема **E-mail клієнт**

Курсова робота
З дисципліни «Технології розроблення програмного забезпечення»

Керівник
Колеснік В. М.
«Допущений до захисту»
_____
(Особистий підпис керівника)
« »_____2024р.
Захищений з оцінкою
_____
(оцінка) Члени комісії:
_____ _____
(особистий підпис) (розшифровка підпису)
_____ _____
(особистий підпис) (розшифровка підпису)

Виконавець
ст. Лещенко Д. С.
залікова книжка № ІА –1215
гр. ІА-12
_____
(особистий підпис виконавця)
« »_____2024р.

Київ – 2024

ЗМІСТ

ВСТУП

В сучасному світі, де електронна пошта залишається ключовим засобом комунікації в бізнесі та особистому спілкуванні, створення ефективного та надійного поштового клієнта є важливим завданням. У рамках цієї курсової роботи розглядається розробка програмного забезпечення, спрямованого на створення функціонального поштового клієнта.

Об’єктом дослідження є програма для обробки електронних листів, яка підтримує різноманітні протоколи (pop3/smtp/imap) та наділена рядом ключових функцій. Розглянемо і аналізуємо функціональні можливості, такі як автонастройка для основних поштових провайдерів України, розділення повідомлень на папки, категорії та важливість, зберігання чернеток незавершених листів, а також можливість прикріплення та обробки файлів.

Ця робота спрямована на вивчення технологій розробки програмного забезпечення та їх практичне застосування для створення надійного та зручного поштового клієнта, здатного задовольнити потреби широкого кола користувачів. У результаті вивчення та аналізу різноманітних аспектів розробки програмного забезпечення, ми розкриємо ключові принципи та технічні аспекти, які лежать в основі ефективного та зручного поштового клієнта.

Ця робота є спробою поєднати теоретичні знання з практичними навичками розробки програмного забезпечення, щоб створити корисний та інноваційний продукт у сфері цифрової комунікації.

1 ПРОЄКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

1. Mozilla Thunderbird

Mozilla Thunderbird є безкоштовним поштовим клієнтом, який підтримує протоколи POP3, IMAP і SMTP. Він надає користувачам широкі можливості для управління своєю електронною поштою, включаючи функції фільтрації, сортування повідомлень та підтримку різних розширень. Однак, інтерфейс програми може виглядати застарілим, і деякі користувачі можуть зіткнутися з певними труднощами під час налаштування облікових записів.

2. Mailbird[1]

Mailbird - це сучасний поштовий клієнт для операційної системи Windows, який підтримує протоколи POP3, IMAP і SMTP. Його особливістю є чистий та інтуїтивно зрозумілий інтерфейс, а також інтеграція з різними сторонніми сервісами, такими як WhatsApp, Facebook, Dropbox та інші.

3. The Bat!

The Bat! - це комерційний поштовий клієнт, який славиться високим рівнем безпеки та шифруванням. Він підтримує всі основні поштові протоколи та пропонує розширені можливості для управління поштою. Однак, його інтерфейс може виявитися складним для новачків, і вартість ліцензії може бути високою для деяких користувачів.

Кожен з цих поштових клієнтів має свої унікальні переваги та обмеження. Огляд існуючих рішень у цій сфері дозволяє виявити, які аспекти можуть бути оптимізовані або які нові функції можна впровадити для покращення функціоналу майбутнього поштового клієнта, який розробляється у рамках цієї курсової роботи.

1.2. Загальний опис проєкту

Мета цього проєкту полягає в розробці поштового клієнта з використанням різних патернів програмування, таких як singleton, builder, decorator, template method, interpreter та SOA. Цей поштовий клієнт має бути функціонально схожим на поштові програми Mozilla Thunderbird та The Bat.

Основні функціональні вимоги до поштового клієнта включають:

- Підтримку протоколів pop3/smtp/imap для отримання та відправки електронних листів.
- Автоналаштування для основних поштових провайдерів України, таких як gmail, ukr.net, i.ua.
- Організацію листів у папки, категорії та встановлення їх важливості.

Проєкт буде розділений на три рівні взаємодії:

- Рівень інтерфейсу користувача для взаємодії з клієнтом.
- Рівень бізнес-логіки для обробки листів та автонастройки.
- Рівень взаємодії з поштовими протоколами.

1.3. Вимоги до застосунків системи

1.3.1. Функціональні вимоги до системи

Система має відповідати наступним функціональним вимогам:

- Робота з вкладеннями
- Автоналаштування поштових провайдерів.

1.3.2. Нефункціональні вимоги до системи

Система має відповідати наступним нефункціональним вимогам:

– система повинна правильно відображати дані та надійно працювати.

– інтерфейс користувача має бути зручним та інтуїтивно-зрозумілим;

– Система повинна мати модульну архітектуру

1.4. Сценарії використання системи

Діаграма прецедентів в Уніфікованій мові моделювання (UML) використовується для моделювання функціональних можливостей системи та взаємодій між користувачами системи (акторами) та самою системою. Ця діаграма має динамічний характер і допомагає у зборі вимог для системи, а також уявити, як користувачі будуть взаємодіяти з системою через різні сценарії (прецеденти).

Діаграми варіантів використання є важливим інструментом при зборі вимог до програмного продукту та його реалізації. Ця модель створюється на аналітичному етапі розробки програмного продукту, коли здійснюється збір і аналіз вимог, і вона допомагає бізнес-аналітикам отримати більш повний і зрозумілий огляд того, яке програмне забезпечення потрібно створити. Діаграми варіантів використання відображають різні сценарії взаємодії між користувачами і системою, що сприяє

кращому розумінню вимог і їхній подальшій реалізації.

Діаграма варіантів використання має свій набір основних елементів, включаючи варіанти використання (прецеденти), акторів (дійові особи) та відносини між ними. Для створення таких діаграм UML використовується універсальний інструмент, яким є draw.io [2]. Цей інструмент дозволяє легко і швидко створювати різноманітні діаграми, використовуючи зрозумілий і простий інтерфейс. З його допомогою можна створювати візуалізацію варіантів використання та взаємодії акторів з системою, що спрощує роботу з моделями системи.

У процесі проектування системи було побудовано діаграму прецедентів представлену на рис 1.1.

Актором є користувач системи: звичайний користувач.

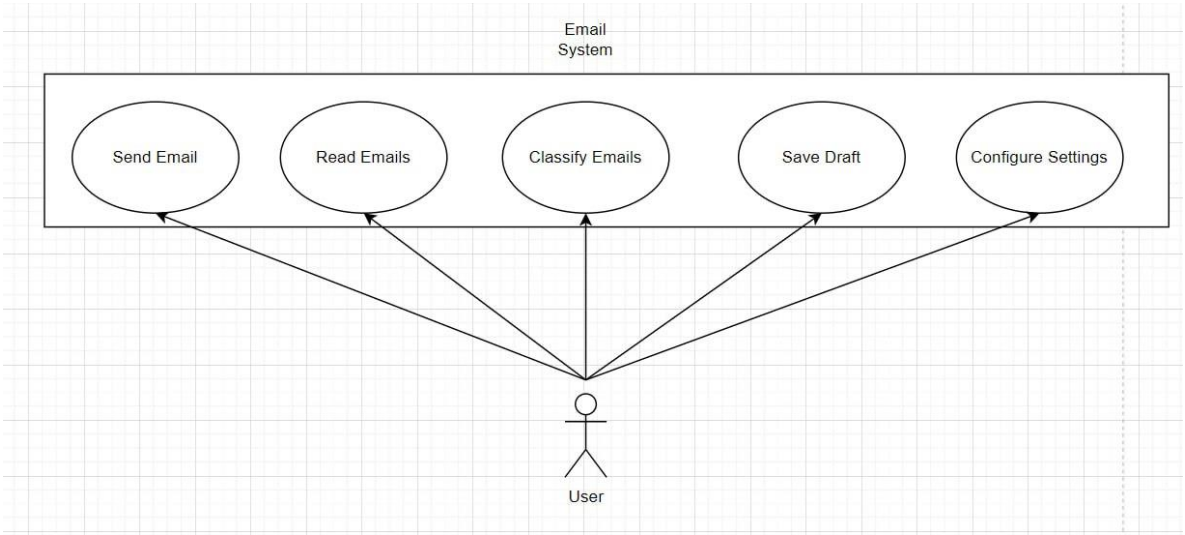


Рис 1.1 – Діаграма прецедентів

Детальні описи деяких сценаріїв використання наведено нижче.

Таблиця 1.1 – Сценарій використання «Вхід користувача»

Назва	Вхід користувача
Передумови	Сервер парцює, користувач підєднаний
Постумови	Користувач увійшов до системи
Актори	Користувач, сервер
Опис	Користувач підєднується до сервера та вводить дані для: provider, email address, password
Основний хід подій	Сервер отримує команду, авторизовує користувача
Винятки та примітки	Немає

Таблиця 1.2 – Сценарій використання «Надисилання електроного листа»

Назва	Надсилання електроного листа
Передумови	Користувач увійшов до системи, має доступ до серверу.
Післяумови	Лист надіслано
Сторони, що взаємодіють	Користувач, сервер
Опис	Користувач вводить команду send email recipient subject body
Основний потік подій	Сервер отримує команду "send email", Обробляє команду та надсилає листа
Виняткові та примітки	Користувач повинен ввести кореткі дані

Таблиця 1.3 – Сценарій використання «Переглід листів»

Назва	Переглід листів
Передумови	Користувач увійшов до системи, має доступ до серверу та ввів команду для перегляду
Післяумови	Користувач бачить листи
Сторони, що взаємодіють	Користувач, сервер

Опис	Користувач вводить команду "read emails".
Основний потік подій	Сервер отримує команду "read emails", обробляє команду та надсилає відповідь
Виняткові та примітки	Немає

1.5 Концептуальна модель системи

У моєму додатку я буду використовувати архітектуру клієнт-сервер [3]. Проект може бути розділений на кілька рівнів:

Клієнтська частина містить в собі візуальний інтерфейс для користувача та логіку обробки дій користувача. Вона також включає код для налаштування зв'язку з сервером і виконання відповідних запитів до нього.

Серверна частина включає в себе основну логіку програми, що відповідає за бізнес-процеси, а також частину, яка відповідає за зберігання і обмін даними між клієнтом і сервером, або між різними клієнтами.

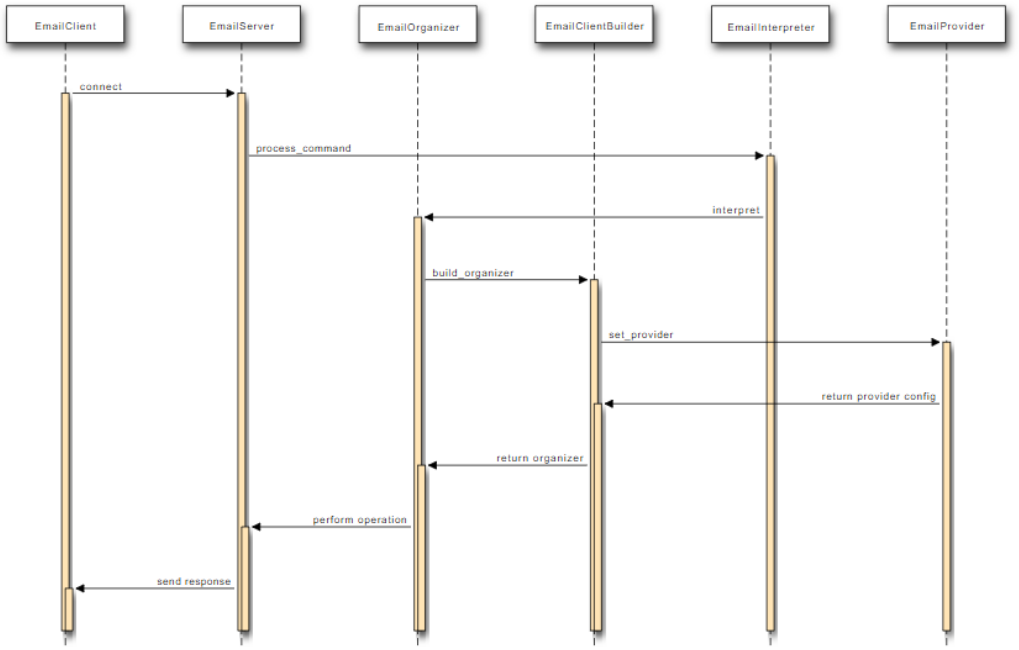


Рисунок 1.2 – Діаграма послідовностей

Структура серверної частини наведені на рисунку 1.3.

Вся бізнес-логіка, яка стосується відтворення команд в терміналі знаходиться саме тут.

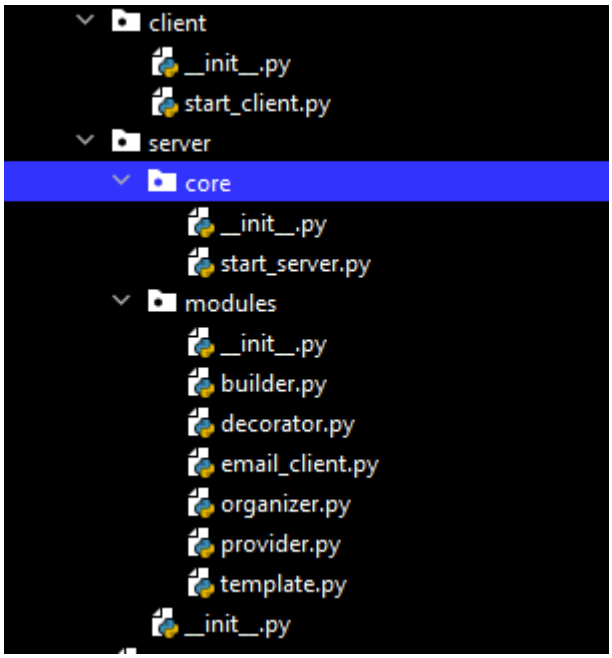


Рисунок 1.3 – Структура серверної частини

Структура клієнтської частини наведені на рисунку 1.4.

Тут знаходиться логіка з'єднання з сервером за допомогою сокету і потім входить в цикл, де користувач може вводити команди.

Введена команда надсилається серверу, і клієнт отримує та друкує відповідь сервера.

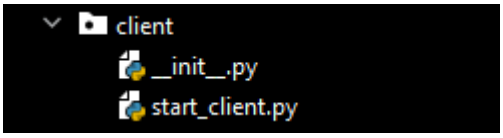


Рисунок 1.4– Структура клієнтської частини

Загальна діаграма класів системи представлено в додатку А.

1.6 Вибір мови програмування та середовища розробки

У курсовій роботі моєю обраною мовою програмування стала Python[4], а в якості середовища розробки я використовував PyCharm. Вибір Python обумовлений кількома ключовими факторами. Ця мова програмування славиться своєю простотою вивчення, читабельністю коду та багатим екосистемом бібліотек. Вона також відома своєю швидкістю розробки прототипів, що стало важливим для створення працездатного E-mail клієнта.

У свою чергу, PyCharm[5] відмінно підходить для роботи з Python. Це інтегроване середовище розробки забезпечує зручні інструменти для написання коду, відлагодження, контролю версій та управління проектом. Він допомагає спростити навігацію по проекту, аналізувати код та виявляти можливі проблеми, сприяючи поліпшенню якості та ефективності розробки програмного продукту. Об'єднання Python та PyCharm створило оптимальне середовище для створення E-mail клієнта з високою продуктивністю та якістю коду.

1.7 Проєктування розгортання системи

Діаграма розгортання в Уніфікованій мові моделювання (UML) використовується для моделювання фізичного аспекту об'єктно-орієнтованої програмної системи. Ця діаграма представляє собою структурну схему, яка відображає архітектуру системи, розгортання програмних артефактів для процесу розгортання. Артефакти - це конкретні елементи у фізичному світі, які виникають під час розробки програми. Діаграма розгортання моделює конфігурацію часу виконання у статичному вигляді і візуалізує розподіл артефактів у програмі. Зазвичай це включає моделювання апаратних конфігурацій разом із компонентами програмного забезпечення, які працюють на цих апаратах.

Нижче на рисунку 1.7 зображено діаграму розгортання системи.

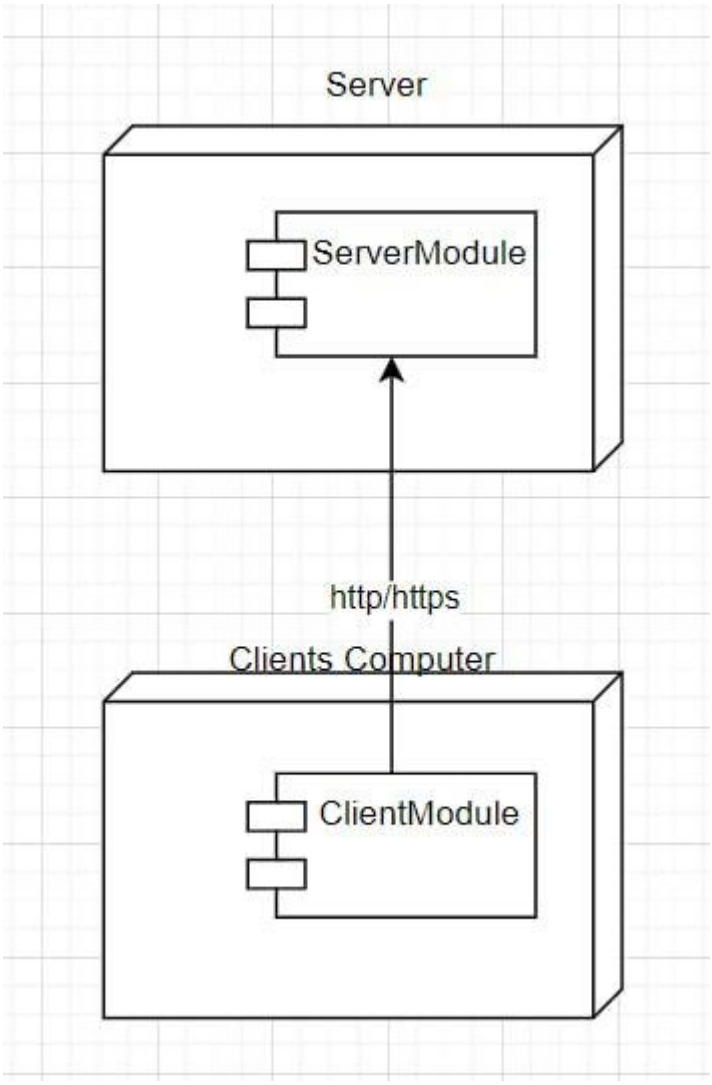


Рис 1.7 – Діаграма розгортання системи

2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1. Архітектура системи

2.1.1. Специфікація системи

Система реалізована у формі консольного застосунку. Консольні застосунки - це програми, які працюють у текстовому режимі та взаємодіють з користувачем через командний рядок або термінал. У них немає графічного інтерфейсу, а замість цього вони використовують текстові команди для введення даних та отримання виводу.

Консольні застосунки відзначаються своєю простотою, легкістю використання та потужністю. Їх часто використовують для автоматизації завдань, взаємодії з операційною системою, роботи з файлами, виконання скриптів та інших завдань у текстовому режимі. Це дозволяє користувачам використовувати команди і скрипти для виконання різноманітних операцій без необхідності великого графічного інтерфейсу. Консольні застосунки є дуже потужними інструментами для взаємодії з операційною системою і вирішення різних завдань на рівні тексту.

Ці консольні застосунки можуть бути використані в різноманітних сферах, включаючи системні утиліти, такі як командний рядок у Windows або Terminal у macOS, а також для створення скриптів для обробки даних, автоматизації рутинних завдань та створення інтерфейсів для різних операцій. Вони надають користувачам зручний і потужний інструмент для взаємодії з комп'ютерною системою та виконання різних завдань за допомогою текстового введення та команд.

Однією з основних переваг консольних застосунків є їх універсальність та можливість використання на різних платформах без суттєвих змін. Вони також можуть бути дуже ефективними для роботи з великою кількістю даних або для виконання складних завдань, де важлива швидкість обробки.

2.1.2. Вибір та обґрунтування патернів реалізації

Патерн проектування[6] представляє собою типовий метод вирішення певних проблем, які часто виникають при проектуванні архітектури програм

Відмінною особливістю патерна є те, що його неможливо просто взяти і скопіювати в програму, як це можливо з готовими функціями або бібліотеками. Патерн - це загальний принцип розв'язання певної проблеми, який майже завжди потребує налаштування та адаптації під конкретні потреби програми.

Паттерн проектування, навпаки, є високорівневим описом загального рішення для типової проблеми. Він подібний до плану або концепції, яка дає загальну ідею, але залишає простір для творчості та адаптації у конкретному контексті. Реалізація паттерну може варіюватися в різних програмах, але основний принцип залишається незмінним. Таким чином, паттерни проектування надають абстракцію та структуру для проектування програм, роблячи їх більш зрозумілими та підготовленими до майбутнього розвитку.

При розробці проекту був застосований патерн «Decorator»[7].

Декоратор — це структурний патерн проектування, що дає змогу динамічно додавати об’єктам нову функціональність, загортаючи їх у корисні «обгортки».

Decorator використовується для додавання додаткової функціональності до методів класу без їх зміни. Декоратор **track_execution_time** вимірює і виводить час виконання методу, до якого він застосований. Наприклад, коли він застосований до методу **connect_to_server** класу **EmailClient**, цей метод автоматично отримує додаткову функціональність відстеження часу виконання. Цей патерн дозволяє динамічно додавати нову поведінку до об'єктів, не змінюючи їх коду.

Наступним патерном, що було застосовано, став «Template»[8].

MailTemplate визначає основу алгоритму відправки електронної пошти у методі perform_email_operation, який викликає абстрактні методи (connect_to_server, prepare_message, send_message, disconnect_from_server). EmailClient наслідує MailTemplate і реалізує ці абстрактні методи, забезпечуючи конкретну логіку для кожного кроку. Це дозволяє змінювати поведінку окремих частин алгоритму без зміни його загальної структури.

Наступним патерном, що було застосовано, став «Singleton»[9].

Патерн Singleton використовується для забезпечення того, що клас MailServiceProvider має лише один екземпляр для кожного унікального набору параметрів smtp_server, smtp_port, imap_server, та pop3_server. Клас MailServiceProvider має приватний статичний атрибут _instances, який зберігає вже створені екземпляри класу за ключами (наборами параметрів). Метод __new__ перевіряє, чи існує вже екземпляр класу з такими самими параметрами, за допомогою ключа key (зі значеннями smtp_server, smtp_port, imap_server, та pop3_server). Якщо такий екземпляр існує, то повертається вже існуючий екземпляр. Якщо екземпляр не існує, то створюється новий екземпляр класу, зберігається в _instances і повертається цей новий екземпляр. Метод __init__ встановлює параметри smtp_server, smtp_port, imap_server, і pop3_server для екземпляра класу. Цей підхід гарантує, що для кожного унікального набору параметрів створюється лише один екземпляр класу MailServiceProvider. Цей підхід гарантує, що для кожного унікального набору параметрів створюється лише один екземпляр класу MailServiceProvider

Наступним патерном, що було застосовано, став «Builder»[10].

Паттерн "Builder" використовується коді для створення складних

об'єктів класів MailClient та MailManager, які мають багато параметрів конфігурації. Клас MailClientBuilder створюється для побудови об'єктів класу MailClient і MailManager. Він містить приватні поля для зберігання параметрів, які необхідно передати під час створення цих об'єктів (наприклад, provider, user_email, user_password). Клас MailClientBuilder надає публічні методи, такі як set_provider, set_user_email, та set_user_password, для встановлення параметрів конфігурації об'єкта. Методи set_provider, set_user_email, та set_user_password повертають self, щоб надати можливість ланцюжкового виклику, тобто ви можете послідовно викликати ці методи для встановлення параметрів. Коли всі необхідні параметри встановлені, можна викликати методи build або build_organizer, які створюють і повертають об'єкт класу MailClient або MailManager відповідно. За допомогою цього паттерна можна створювати об'єкти зі складною конфігурацією, не забиваючи конструктор класу великою кількістю аргументів. Це робить код більш зрозумілим і об'єктно-орієнтованим.

Наступним патерном, що було застосовано, став «Interpreter»[11].

Паттерн "Interpreter" використовується для інтерпретації команд, які надсилаються клієнтом до сервера. В класі MailProcessor визначено метод interpret, який приймає команду як вхідний параметр. Кожна команда перевіряється на наявність певних ключових слів, таких як "send email", "classify emails", "save draft", "read emails" тощо. Коли знайдено відповідне ключове слово, виконується відповідна операція. Наприклад, якщо команда містить "send email", то викликається метод perform_email_operation. Паттерн "Interpreter" дозволяє розділити логіку обробки різних команд на окремі методи і виконувати їх в залежності від команди, що надійшла. Цей підхід робить код більш зрозумілим і підтримує розширення функціональності. Додавання нових команд стає досить простою задачею, оскільки достатньо створити новий метод для інтерпретації команди та додати відповідний умовний оператор.

2.2. Інструкція користувача

На рисунку 2.8 зображено графічний інтерфейс системи. Так як це консольний застосунок, то ми бачимо консоль.

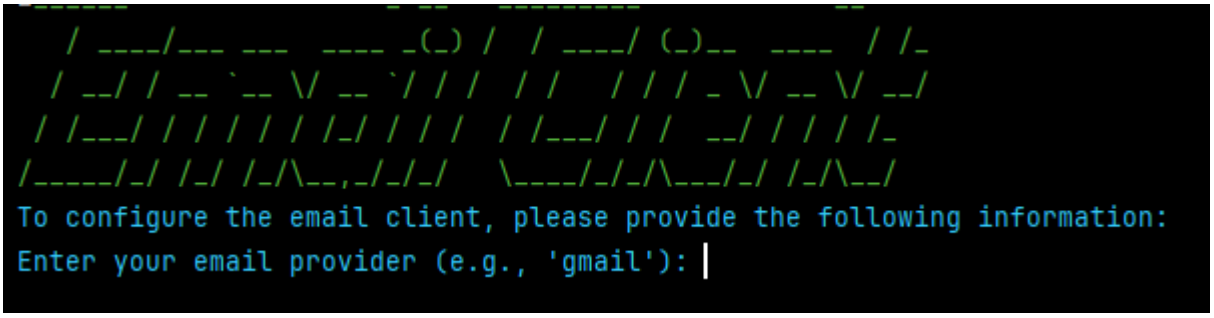


Рисунок 2.8 – графічний інтерфейс системи

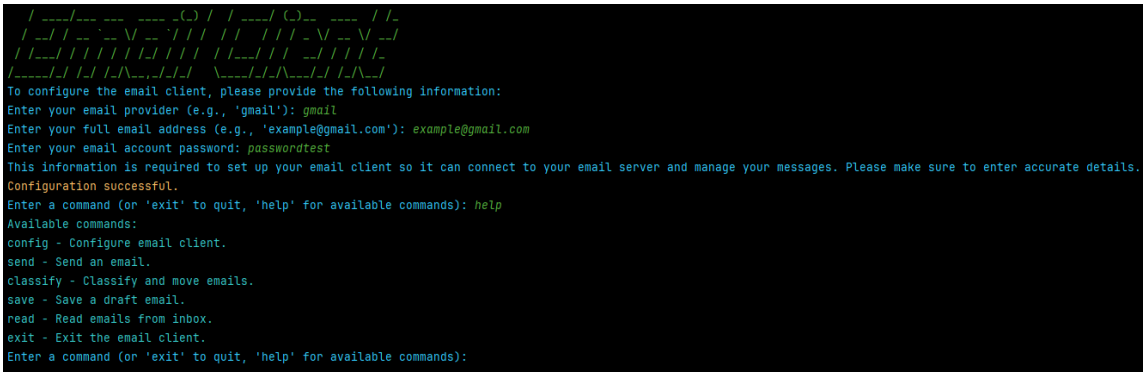


Рисунок 2.9 – відображення доступних команд за допомогою команди help

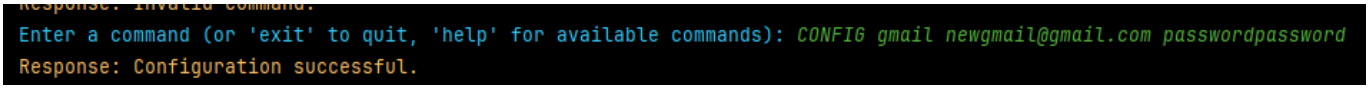


Рисунок 2.10 – Автоконфігурація за новим провайдером, логіном та паролем

Далі відправляємо лист за адресою

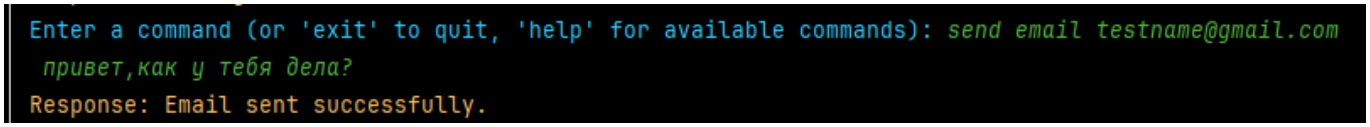


Рисунок 2.11 – відправка листа

Вводимо команду classify emails для розподілення листів по папкам

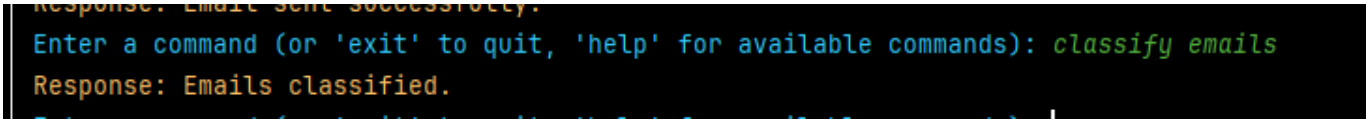


Рисунок 2.12 – Вигляд вікна з історією терміналу

ВИСНОВКИ

Під час розробки поштового клієнта було проведено дослідження та реалізовано різноманітні функціональні можливості, спрямовані на створення продукту, який відповідає потребам та вимогам сучасних поштових програм. Новий поштовий клієнт був створений, беручи за основу функціонал відомих поштових програм, таких як Mozilla Thunderbird і The Bat, та зорієнтований на оптимальну роботу з різними поштовими протоколами, включаючи pop3, smtp і imap.

Головні риси поштового клієнта включають в себе можливість автоматичної настройки облікових записів для основних поштових провайдерів в Україні, таких як Gmail, Ukr.net і I.ua, що робить процес налаштування більш простим. Клієнт також надає можливість зручного сортування повідомлень за папками, категоріями та ступенем важливості, що сприяє більш ефективній організації поштової скриньки користувача. Крім того, він здатний зберігати чернетки невідправлених повідомлень та ефективно обробляти прикріплені файли.

Результати розробки підтверджують, що поштовий клієнт успішно реалізовує всі основні функції, необхідні для зручної та продуктивної роботи з електронною поштою. Цей продукт пропонує користувачам інтуїтивний інтерфейс та відповідає високим стандартам ефективної поштової комунікації. Для подальшого розвитку поштового клієнта можна розглянути можливості розширення функціоналу та підтримки додаткових поштових сервісів, що ще більше підвищить його привабливість для користувачів. Під час розробки були використані сучасні технології програмування, що гарантує високу ефективність та надійність поштового клієнта на ринку електронної пошти.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Mailbird [Електронний ресурс]. Доступ: https://en.wikipedia.org/wiki/Mailbird Дата звернення: 05.10.2023

[2] Draw.io. [Електронний ресурс]. Доступ: https://drawio-app.com/tutorials/. Дата звернення: 07.10.2023

[3] Client/Server Architecture. [Електронний ресурс]. Доступ: https://www.linkedin.com/pulse/client-server-architecture-eloghene-otiede . Дата звернення: 8.10.2023

[4] Python. [Електронний ресурс]. Доступ: https://www.python.org/. Дата звернення: 11.10.2023

[5] PyCharm. [Електронний ресурс]. Доступ: https://www.jetbrains.com/help/pycharm/getting-started.html. Дата звернення: 15.11.2023

[6] Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra, *HeadFirst Design Patterns*, O'Reilly Media, Inc., 2004. Дата звернення: 17.11.2023

[7] Decorator Pattern. [Електронний ресурс]. Доступ: https://refactoring.guru/uk/design-patterns/decorator Дата звернення:01.12.2023

[8] Template Pattern. [Електронний ресурс]. Доступ: https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) Дата звернення:10.12.2023

[9] Singleton Pattern [Електронний ресурс]. Доступ: https://refactoring.guru/ru/design-patterns/singleton Дата звернення: 10.12.2023

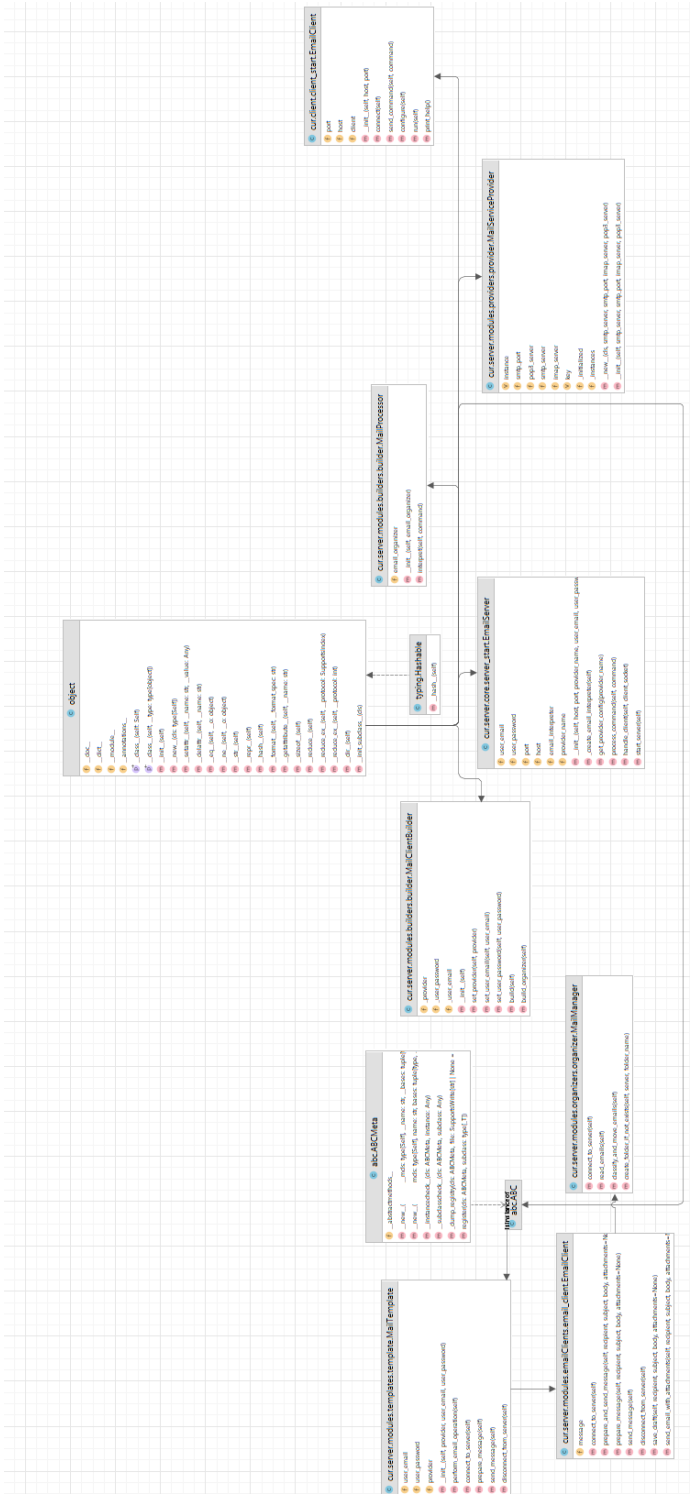
[10] Builder Pattern [Електронний ресурс] Доступ: https://refactoring.guru/design-patterns/builder Дата звернення: 10.12.2023

[11]Interpreter Pattern [Електронний ресурс]. Доступ: https://en.wikipedia.org/wiki/Interpreter_pattern Дата звернення: 11.12.2023

ДОДАТКИ

ДОДАТОК А

Діаграма класів



ДОДАТОК Б

Код проєкту

client_start.py

```
import socket
import os
from colorama import init, Fore
import pyfiglet

class EmailClient:
    """
    A simple email client for managing email messages through a command-line
interface.

    Attributes:
    - host (str): The email server's hostname or IP address.
    - port (int): The port number to connect to on the email server.
    - client (socket.socket): A socket object for communication with the email
server.

    Methods:
    - __init__(self, host, port): Initializes the EmailClient instance with the
provided host and port.
    - connect(self): Establishes a connection to the email server.
    - send_command(self, command): Sends a command to the email server and returns
the response.
    - configure(self): Configures the email client by requesting user input for email
provider, user email, and password.
    - run(self): Runs the email client's main loop to process user commands.
    - print_help(): Static method that prints the available commands and their
descriptions.
    """

    def __init__(self, host, port):
        """
        Initializes a new EmailClient instance.

        Args:
        - host (str): The hostname or IP address of the email server.
        - port (int): The port number to connect to on the email server.
        """
        self.host = host
        self.port = port
        self.client = None

    def connect(self):
        """
        Establishes a connection to the email server using a socket.
        """
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client.connect((self.host, self.port))

    def send_command(self, command):
        """
```

```
        Sends a command to the email server and receives the response.

        Args:
        - command (str): The command to send to the email server.

        Returns:
        - response (str): The response received from the email server.
        """
        self.client.send(command.encode('utf-8'))
        response = self.client.recv(4096).decode('utf-8')
        return response

    def configure(self):
        """
        Configures the email client by obtaining necessary information from the user,
such as email provider,
        user email address, and password.
        """
        init(autoreset=True)
        os.system('cls' if os.name == 'nt' else 'clear')
        banner = pyfiglet.figlet_format("Email Client", font="slant")
        print(f"{Fore.GREEN}{banner.strip()}{Fore.RESET}")

        print(f"{Fore.BLUE}To configure the email client, please provide the
following information:")
        provider_name = input(f"{Fore.BLUE}Enter your email provider (e.g., 'gmail'):
{Fore.RESET}")
        user_email = input(f"{Fore.BLUE}Enter your full email address (e.g.,
'example@gmail.com'): {Fore.RESET}")
        user_password = input(f"{Fore.BLUE}Enter your email account password:
{Fore.RESET}")

        explanation = f"{Fore.BLUE}This information is required to set up your email
client " \
                      f"so it can connect to your email server and manage your
messages. " \
                      f"Please make sure to enter accurate details.{Fore.RESET}"

        response = self.send_command(f"CONFIG {provider_name} {user_email}
{user_password}")
        print(explanation)
        print(f"{Fore.YELLOW}{response}{Fore.RESET}")

    def run(self):
        """
        Runs the main loop of the email client, processing user commands until the
user decides to exit.
        """
        while True:
            command = input(f"{Fore.BLUE}Enter a command (or 'exit' to quit, 'help'
for available commands): {Fore.RESET}")
            if command.lower() == 'exit':
                break
            elif command.lower() == 'help':
                self.print_help()
                continue
            response = self.send_command(command)
            print(f"{Fore.YELLOW}Response: {response}{Fore.RESET}")

        self.client.close()

    @staticmethod
    def print_help():
        """
```

```
        Static method that prints the available commands and their descriptions.
        """
        print(f"{Fore.CYAN}Available commands:")
        print(f"{Fore.CYAN}config - Configure email client.")
        print(f"{Fore.CYAN}send - Send an email.")
        print(f"{Fore.CYAN}classify - Classify and move emails.")
        print(f"{Fore.CYAN}save - Save a draft email.")
        print(f"{Fore.CYAN}read - Read emails from inbox.")
        print(f"{Fore.CYAN}exit - Exit the email client.{Fore.RESET}")

if __name__ == "__main__":
    HOST = 'localhost'
    PORT = 12348
    email_client = EmailClient(HOST, PORT)
    email_client.connect()
    email_client.configure()
    email_client.run()
```

server_start.py

```
import socket
import threading
from cur.server.modules.builders.builder import MailClientBuilder, MailProcessor
from cur.server.modules.providers.provider import MailServiceProvider


class EmailServer:
    """
    A simple email server that handles client connections and email operations.

    Attributes:
    - host (str): The hostname or IP address where the server will listen for
incoming connections.
    - port (int): The port number to bind the server socket to.
    - provider_name (str): The name of the email service provider.
    - user_email (str): The user's email address.
    - user_password (str): The user's email account password.
    - email_interpreter (MailProcessor): An instance of MailProcessor for handling
email operations.

    Methods:
    - __init__(self, host, port, provider_name, user_email, user_password):
Initializes the EmailServer instance.
    - _create_email_interpreter(self): Creates an email interpreter based on the
provided provider and user credentials.
    - get_provider_config(provider_name): Returns the configuration for a given email
service provider.
    - process_command(self, command): Processes incoming client commands and executes
corresponding actions.
    - handle_client(self, client_socket): Handles communication with a connected
client.
    - start_server(self): Starts the email server and listens for incoming
connections.
    """
    def __init__(self, host, port, provider_name, user_email, user_password):
        """
        Initializes a new EmailServer instance.

        Args:
        - host (str): The hostname or IP address where the server will listen for
incoming connections.
        - port (int): The port number to bind the server socket to.
```

```
        - provider_name (str): The name of the email service provider.
        - user_email (str): The user's email address.
        - user_password (str): The user's email account password.
        """
        self.host = host
        self.port = port
        self.provider_name = provider_name
        self.user_email = user_email
        self.user_password = user_password
        self.email_interpreter = self._create_email_interpreter()

    def _create_email_interpreter(self):
        """
        Creates an email interpreter based on the provided provider and user
credentials.

        Returns:
        - MailProcessor: An instance of MailProcessor for handling email operations.
        """
        config = self.get_provider_config(self.provider_name)
        if config:
            client_builder = MailClientBuilder()
            organizer = (client_builder.set_provider(config)
                                       .set_user_email(self.user_email)
                                       .set_user_password(self.user_password))
            return MailProcessor(organizer.build_organizer())
        else:
            raise ValueError(f"Провайдер '{self.provider_name}' не найден.")

    @staticmethod
    def get_provider_config(provider_name):
        """
        Returns the configuration for a given email service provider.

        Args:
        - provider_name (str): The name of the email service provider.

        Returns:
        - MailServiceProvider: The configuration for the specified provider.
        """
        providers = {
            'gmail': MailServiceProvider('smtp.gmail.com', 587, 'imap.gmail.com',
'pop.gmail.com'),
            'ukr.net': MailServiceProvider('smtp.ukr.net', 465, 'imap.ukr.net',
'pop3.ukr.net'),
            'i.ua': MailServiceProvider('smtp.i.ua', 465, 'imap.i.ua', 'pop3.i.ua')
        }
        return providers.get(provider_name.lower())

    def process_command(self, command):
        """
        Processes incoming client commands and executes corresponding actions.

        Args:
        - command (str): The command received from the client.

        Returns:
        - str: The response to be sent back to the client.
        """
        if command.startswith("CONFIG"):
            try:
                _, provider_name, user_email, user_password = command.split(" ", 3)
                config = self.get_provider_config(provider_name)
                if config:
```

```
                    client_builder = MailClientBuilder()
                    organizer = (client_builder.set_provider(config)
                                 .set_user_email(user_email)
                                 .set_user_password(user_password))
                    self.email_interpreter =
MailProcessor(organizer.build_organizer())
                    return "Configuration successful."
                else:
                    return f"Provider '{provider_name}' not found."
            except Exception as e:
                return f"Error in configuration: {e}"

        elif command.startswith("send email"):
            _, recipient, subject, body = command.split(" ", 3)

self.email_interpreter.email_organizer.prepare_and_send_message(recipient, subject,
body)
            return "Email sent successfully."

        elif command.startswith("classify emails"):
            self.email_interpreter.email_organizer.classify_and_move_emails()
            return "Emails classified."

        elif command.startswith("read emails"):
            self.email_interpreter.email_organizer.read_emails()
            return "Emails read."

        elif command.startswith("save draft"):
            params = command.split(" ", 3)[1:]
            if len(params) >= 3:
                recipient, subject, body = params
                self.email_interpreter.email_organizer.save_draft(recipient, subject,
body)
                return "Draft saved."
            else:
                return "Insufficient parameters for 'save draft'."

        else:
            return "Invalid command."

    def handle_client(self, client_socket):
        """
        Handles communication with a connected client.

        Args:
        - client_socket (socket.socket): The socket connected to the client.
        """
        while True:
            request = client_socket.recv(1024)
            if not request:
                break
            response = self.process_command(request.decode('utf-8'))
            client_socket.send(response.encode('utf-8'))
        client_socket.close()

    def start_server(self):
        """
        Starts the email server and listens for incoming connections.
        """
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind((self.host, self.port))
        server.listen(5)
        print(f"Server listening on {self.host}:{self.port}")
```

```
        while True:
            client, address = server.accept()
            client_handler = threading.Thread(target=self.handle_client,
args=(client,))
            client_handler.start()


if __name__ == "__main__":
    HOST = 'localhost'
    PORT = 12348
    PROVIDER_NAME = 'gmail'
    USER_EMAIL = 'your-email@example.com'
    USER_PASSWORD = 'your_password'

    email_server = EmailServer(HOST, PORT, PROVIDER_NAME, USER_EMAIL, USER_PASSWORD)
    email_server.start_server()
```

builder.py

```
from cur.server.modules.emailClients.email_client import EmailClient
from cur.server.modules.organizers.organizer import MailManager

class MailClientBuilder:
    """
    A builder class for creating instances of EmailClient and MailManager.

    Methods:
    - __init__(self): Initializes a new MailClientBuilder instance.
    - set_provider(self, provider): Sets the email service provider for the builder.
    - set_user_email(self, user_email): Sets the user's email address for the
builder.
    - set_user_password(self, user_password): Sets the user's email account password
for the builder.
    - build(self): Builds and returns an EmailClient instance based on the provided
parameters.
    - build_organizer(self): Builds and returns a MailManager instance based on the
provided parameters.
    """

    def __init__(self):
        """
        Initializes a new MailClientBuilder instance.
        """
        self._provider = None
        self._user_email = None
        self._user_password = None

    def set_provider(self, provider):
        """
        Sets the email service provider for the builder.

        Args:
        - provider: The email service provider.

        Returns:
        - MailClientBuilder: The builder instance with the provider set.
        """
        self._provider = provider
        return self

    def set_user_email(self, user_email):
        """
```

```
        Sets the user's email address for the builder.

        Args:
        - user_email (str): The user's email address.

        Returns:
        - MailClientBuilder: The builder instance with the user's email set.
        """
        self._user_email = user_email
        return self

    def set_user_password(self, user_password):
        """
        Sets the user's email account password for the builder.

        Args:
        - user_password (str): The user's email account password.

        Returns:
        - MailClientBuilder: The builder instance with the user's password set.
        """
        self._user_password = user_password
        return self

    def build(self):
        """
        Builds and returns an EmailClient instance based on the provided parameters.

        Returns:
        - EmailClient: An EmailClient instance.
        """
        if not all([self._provider, self._user_email, self._user_password]):
            raise ValueError("Required fields are missing.")
        return EmailClient(self._provider, self._user_email, self._user_password)

    def build_organizer(self):
        """
        Builds and returns a MailManager instance based on the provided parameters.

        Returns:
        - MailManager: A MailManager instance.
        """
        if not all([self._provider, self._user_email, self._user_password]):
            raise ValueError("Required fields are missing.")
        return MailManager(self._provider, self._user_email, self._user_password)

class MailProcessor:
    """
    A class for interpreting and executing email-related commands.

    Attributes:
    - email_organizer (MailManager): An instance of MailManager for email operations.

    Methods:
    - __init__(self, email_organizer): Initializes a new MailProcessor instance.
    - interpret(self, command): Interprets a command and performs the corresponding
email operation.
    """

    def __init__(self, email_organizer):
        """
        Initializes a new MailProcessor instance.

        Args:
```

```
        - email_organizer (MailManager): An instance of MailManager for email
operations.
        """
        self.email_organizer = email_organizer

    def interpret(self, command):
        """
        Interprets a command and performs the corresponding email operation.

        Args:
        - command (str): The command to interpret and execute.
        """
        if "send email" in command:
            self.email_organizer.perform_email_operation()
        elif "classify emails" in command:
            self.email_organizer.classify_and_move_emails()
        elif "save draft" in command:
            self.email_organizer.save_draft("recipient@example.com", "Draft Subject",
"Draft Body")
        elif "list folders" in command:
            print("List of folders...")
        elif "read emails" in command:
            self.email_organizer.read_emails()
        else:
            print("Invalid command.")
```

decorator.py

```
import time

def track_execution_time(func):
    """
    A decorator that tracks and prints the execution time of a wrapped function.

    Args:
    - func (callable): The function to be wrapped.

    Returns:
    - wrapper: The wrapped function.
    """

    def wrapper(*args, **kwargs):
        """
        Calculates and prints the execution time of the wrapped function.

        Args:
        - *args: Positional arguments to be passed to the wrapped function.
        - **kwargs: Keyword arguments to be passed to the wrapped function.

        Returns:
        - result: The result of the wrapped function.
        """
        start_time = time.time()
        print(f"Executing operation '{func.__name__}'...")
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Operation '{func.__name__}' executed in {execution_time:.4f}
seconds.")

        return result
```

```
    return wrapper
```

email_client.py

```
import os
import smtplib
from email import encoders
from email.mime.base import MIMEBase
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

from cur.server.modules.decorators.decorator import track_execution_time
from cur.server.modules.templates.template import MailTemplate


class EmailClient(MailTemplate):
    """
    A class representing an email client for sending and managing emails.

    Attributes:
    - user_email (str): The user's email address.
    - user_password (str): The user's email account password.
    - provider (MailServiceProvider): The email service provider configuration.
    - message (MIMEMultipart): The email message to be sent.

    Methods:
    - connect_to_server(self): Connects to the SMTP server for sending emails.
    - prepare_and_send_message(self, recipient, subject, body, attachments=None):
Prepares and sends an email message.
    - prepare_message(self, recipient, subject, body, attachments=None): Prepares an
email message without sending it.
    - send_message(self): Sends a prepared email message.
    - disconnect_from_server(self): Disconnects from the SMTP server.
    - save_draft(self, recipient, subject, body, attachments=None): Saves an email
draft locally.
    - send_email_with_attachments(self, recipient, subject, body, attachments=None):
Sends an email with attachments.
    """
    @track_execution_time
    def connect_to_server(self):
        """
        Connects to the SMTP server for sending emails.
        """
        try:
            print("Підключення до SMTP серверу...")

            with smtplib.SMTP(self.provider.smtp_server, self.provider.smtp_port) as
server:
                server.starttls()
                server.login(self.user_email, self.user_password)

        except Exception as e:
            print(f"Помилка підключення до SMTP серверу: {e}")



    def prepare_and_send_message(self, recipient, subject, body, attachments=None):
        """
        Prepares and sends an email message.

        Args:
        - recipient (str): The recipient's email address.
```

```
        - subject (str): The subject of the email.
        - body (str): The body text of the email.
        - attachments (list of str, optional): List of file paths for email
attachments.
        """
        print("Підготовка та відправка повідомлення...")

        self.prepare_message(recipient, subject, body, attachments)
        self.send_message()

    def prepare_message(self, recipient, subject, body, attachments=None):
        """
        Prepares an email message without sending it.

        Args:
        - recipient (str): The recipient's email address.
        - subject (str): The subject of the email.
        - body (str): The body text of the email.
        - attachments (list of str, optional): List of file paths for email
attachments.
        """
        print("Підготовка повідомлення...")

        self.message = MIMEMultipart()
        self.message['From'] = self.user_email
        self.message['To'] = recipient
        self.message['Subject'] = subject
        self.message.attach(MIMEText(body, 'plain'))

        if attachments:
            for file_path in attachments:
                part = MIMEBase('application', "octet-stream")
                with open(file_path, 'rb') as file:
                    part.set_payload(file.read())
                encoders.encode_base64(part)
                part.add_header('Content-Disposition', f'attachment;
filename={os.path.basename(file_path)}')
                self.message.attach(part)

    def send_message(self):
        """
        Sends a prepared email message.
        """
        print("Відправлення повідомлення...")

        try:
            with smtplib.SMTP(self.provider.smtp_server, self.provider.smtp_port) as
server:
                server.starttls()
                server.login(self.user_email, self.user_password)
                server.sendmail(self.user_email, [self.message['To']],
self.message.as_string())
            print("Email sent successfully!")
        except Exception as e:
            print(f"Error sending email: {e}")


    def disconnect_from_server(self):
        """
        Disconnects from the SMTP server.
        """
        print("Відключення від SMTP сервера...")

    def save_draft(self, recipient, subject, body, attachments=None):
```

```
        """
        Saves an email draft locally.

        Args:
        - recipient (str): The recipient's email address.
        - subject (str): The subject of the email draft.
        - body (str): The body text of the email draft.
        - attachments (list of str, optional): List of file paths for email draft
attachments.
        """
        print("Збереження чернетки...")

        msg = MIMEMultipart()
        msg['From'] = self.user_email
        msg['To'] = recipient
        msg['Subject'] = subject
        msg.attach(MIMEText(body, 'plain'))

        if attachments:
            for file_path in attachments:
                part = MIMEBase('application', "octet-stream")
                with open(file_path, 'rb') as file:
                    part.set_payload(file.read())
                encoders.encode_base64(part)
                part.add_header('Content-Disposition', f'attachment;
filename={os.path.basename(file_path)}')
                msg.attach(part)

        with open(f"{subject}_draft.eml", "w") as draft_file:
            draft_file.write(msg.as_string())
        print("Draft saved successfully.")


    def send_email_with_attachments(self, recipient, subject, body,
attachments=None):
        """
        Sends an email with attachments.

        Args:
        - recipient (str): The recipient's email address.
        - subject (str): The subject of the email.
        - body (str): The body text of the email.
        - attachments (list of str, optional): List of file paths for email
attachments.
        """
        print("Відправлення листа з додатками...")

        msg = MIMEMultipart()
        msg['From'] = self.user_email
        msg['To'] = recipient
        msg['Subject'] = subject
        msg.attach(MIMEText(body, 'plain'))

        if attachments:
            for file_path in attachments:
                part = MIMEBase('application', "octet-stream")
                with open(file_path, 'rb') as file:
                    part.set_payload(file.read())
                encoders.encode_base64(part)
                part.add_header('Content-Disposition', f'attachment;
filename={os.path.basename(file_path)}')
                msg.attach(part)

        try:
```

```
            with smtplib.SMTP(self.provider.smtp_server, self.provider.smtp_port) as
server:
                server.starttls()
                server.login(self.user_email, self.user_password)
                server.sendmail(self.user_email, [recipient], msg.as_string())
            print("Email with attachments sent successfully!")
        except Exception as e:
            print(f"Error sending email with attachments: {e}")
```

organizer.py

```
import email
import imaplib
from cur.server.modules.decorators.decorator import track_execution_time
from cur.server.modules.emailClients.email_client import EmailClient


class MailManager(EmailClient):
    """
    A class representing a mail manager for reading, classifying, and moving emails.

    Inherits from EmailClient.

    Methods:
    - connect_to_server(self): Connects to the IMAP server for reading emails.
    - read_emails(self): Reads and displays emails from the inbox.
    - classify_and_move_emails(self): Classifies and moves emails to specific
folders.
    - create_folder_if_not_exists(self, server, folder_name): Creates a folder on the
server if it doesn't exist.
    """
    @track_execution_time
    def connect_to_server(self):
        """
        Connects to the IMAP server for reading emails.
        """
        try:
            print("Підключення до IMAP серверу...")

            with imaplib.IMAP4_SSL(self.provider.imap_server) as server:
                server.login(self.user_email, self.user_password)
                server.select('inbox')
        except Exception as e:
            print(f"Помилка підключення до IMAP серверу: {e}")

    def read_emails(self):
        """
        Reads and displays emails from the inbox.
        """
        print("Підключення до IMAP серверу...")

        try:

            with imaplib.IMAP4_SSL(self.provider.imap_server) as server:
                server.login(self.user_email, self.user_password)
                server.select('inbox')

                typ, messages = server.search(None, 'ALL')
                if typ != 'OK':
                    print("Не удалось найти сообщения.")
                    return
```

```
            for num in messages[0].split()[:5]:
                typ, data = server.fetch(num, '(RFC822)')
                if typ != 'OK':
                    continue

                msg = email.message_from_bytes(data[0][1])
                print(f"Письмо от: {msg['from']}")
                print(f"Тема: {msg['subject']}")
                print("Содержание:")
                if msg.is_multipart():
                    for part in msg.walk():
                        if part.get_content_type() == 'text/plain':
                            print(part.get_payload(decode=True).decode())
                else:
                    print(msg.get_payload(decode=True).decode())

    except Exception as e:
        print(f"Ошибка при чтении писем: {e}")


def classify_and_move_emails(self):
    """
    Classifies and moves emails to specific folders.
    """
    print("Класифікація та переміщення листів...")
    try:
        with imaplib.IMAP4_SSL(self.provider.imap_server) as server:
            server.login(self.user_email, self.user_password)
            server.select('inbox')
            typ, messages = server.search(None, 'ALL')

            if typ != 'OK':
                print("No messages to classify.")
                return

            for num in messages[0].split():
                typ, data = server.fetch(num, '(RFC822)')
                if typ != 'OK':
                    continue

                msg = email.message_from_bytes(data[0][1])
                subject = msg['subject'].lower()


                if 'important' in subject:
                    self.create_folder_if_not_exists(server, 'Important')
                    server.copy(num.decode('utf-8'), 'Important')
                    server.store(num, '+FLAGS', '\\Deleted')
                elif 'work' in subject:
                    self.create_folder_if_not_exists(server, 'Work')
                    server.copy(num.decode('utf-8'), 'Work')
                    server.store(num, '+FLAGS', '\\Deleted')

            server.expunge()
    except Exception as e:
        print(f"Ошибка при классификации и перемещении писем: {e}")

def create_folder_if_not_exists(self, server, folder_name):
    """
    Creates a folder on the server if it doesn't exist.

    Args:
    - server: The IMAP server connection.
    - folder_name (str): The name of the folder to create.
```

```
        """
        print(f"Створення папки '{folder_name}', якщо вона не існує...")
```

provider.py

```
class MailServiceProvider:
    """
    A class representing an email service provider configuration.

    This class uses the Singleton design pattern to ensure that only one instance
    is created for each unique set of SMTP and IMAP/POP3 server configurations.

    Attributes:
    - smtp_server (str): The SMTP server address for sending emails.
    - smtp_port (int): The SMTP server port number.
    - imap_server (str): The IMAP server address for receiving emails.
    - pop3_server (str): The POP3 server address for receiving emails.

    Methods:
    - __new__(cls, smtp_server, smtp_port, imap_server, pop3_server): Creates a new
instance or returns an existing one.
    - __init__(self, smtp_server, smtp_port, imap_server, pop3_server): Initializes
the provider with server configurations.
    """

    _instances = {}
    def __new__(cls, smtp_server, smtp_port, imap_server, pop3_server):
        """
        Creates a new instance or returns an existing one based on server
configurations.

        Args:
        - smtp_server (str): The SMTP server address for sending emails.
        - smtp_port (int): The SMTP server port number.
        - imap_server (str): The IMAP server address for receiving emails.
        - pop3_server (str): The POP3 server address for receiving emails.

        Returns:
        - instance: An instance of MailServiceProvider.

        Note:
        If an instance with the same server configurations exists, it is returned.
        Otherwise, a new instance is created and stored for future use.
        """
        key = (smtp_server, smtp_port, imap_server, pop3_server)
        if key not in cls._instances:
            instance = super(MailServiceProvider, cls).__new__(cls)
            cls._instances[key] = instance
            return instance
        return cls._instances[key]

    def __init__(self, smtp_server, smtp_port, imap_server, pop3_server):
        """
        Initializes the provider with server configurations.

        Args:
        - smtp_server (str): The SMTP server address for sending emails.
        - smtp_port (int): The SMTP server port number.
        - imap_server (str): The IMAP server address for receiving emails.
        - pop3_server (str): The POP3 server address for receiving emails.

        Note:
```

```
        This method is called when a new instance is created, but it only initializes
        the instance if it's the first time for the given server configurations.
        """
        if not hasattr(self, '_initialized'):
            self.smtp_server = smtp_server
            self.smtp_port = smtp_port
            self.imap_server = imap_server
            self.pop3_server = pop3_server
            self._initialized = True
```

template.py

```
from abc import ABC, abstractmethod

class MailTemplate(ABC):
    """
    An abstract base class representing a template for email operations.

    This class defines the structure for performing email operations, such as
connecting to
    the server, preparing a message, sending a message, and disconnecting from the
server.

    Attributes:
    - provider: The email service provider configuration.
    - user_email (str): The user's email address.
    - user_password (str): The user's email account password.

    Methods:
    - perform_email_operation(self): Performs a sequence of email operations
(template method).
    - connect_to_server(self): Abstract method to connect to the email server.
    - prepare_message(self): Abstract method to prepare an email message.
    - send_message(self): Abstract method to send an email message.
    - disconnect_from_server(self): Abstract method to disconnect from the email
server.
    """
    def __init__(self, provider, user_email, user_password):
        """
        Initializes the MailTemplate with provider and user credentials.

        Args:
        - provider: The email service provider configuration.
        - user_email (str): The user's email address.
        - user_password (str): The user's email account password.
        """
        self.provider = provider
        self.user_email = user_email
        self.user_password = user_password

    def perform_email_operation(self):
        """
        Performs a sequence of email operations (template method).

        Subclasses should implement the abstract methods to define the specific
        behavior of connecting to the server, preparing a message, sending a message,
        and disconnecting from the server.
        """
        self.connect_to_server()
        self.prepare_message()
        self.send_message()
        self.disconnect_from_server()
```

```
@abstractmethod
def connect_to_server(self):
    """
    Abstract method to connect to the email server.

    Subclasses should implement this method with the specific logic for
    connecting to the email server.
    """
    pass

@abstractmethod
def prepare_message(self):
    """
    Abstract method to prepare an email message.

    Subclasses should implement this method with the specific logic for
    preparing the email message.
    """
    pass

@abstractmethod
def send_message(self):
    """
    Abstract method to send an email message.

    Subclasses should implement this method with the specific logic for
    sending the email message.
    """
    pass

@abstractmethod
def disconnect_from_server(self):
    """
    Abstract method to disconnect from the email server.

    Subclasses should implement this method with the specific logic for
    disconnecting from the email server.
    """
    pass
```