

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 1

ЗАВДАННЯ

на курсову роботу студента

Одемчука Назара Олександровича

(прізвище, ім'я, по батькові)

1. Тема Роботи: IRC Client

2. Строк здачі студентом закінченої роботи 30.12.2023

3. Вихідні дані до роботи:

Готовий IRC Client із графічним інтерфейсом, сервером для збереження переписки та реалізовані патерни («COMPOSITE», «BRIDGE», «FACTORY METHOD» та «COMMAND»).

4. Зміст розрахунково – пояснювальної записки (перелік питань, що підлягають розробці)

Огляд існуючих рішень, загальний опис проєкту, вимоги до застосунків системи, сценарії використання системи, концептуальна модель системи, вибір мови програмування та середовища розробки, проєктування розгортання системи, архітектура системи, інструкція користувача.

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

Діаграма класів, Діаграма варіантів використання, Діаграма розгортання, Діаграма компонентів, Діаграма послідовностей

6. Дата видачі завдання 02.10.23

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Отримання теми та завдання на курсову роботу	02.10.23	
2.	Підбір та вивчення літератури	02-23/10/2023	
3.	Аналіз вимог	23/09/2023-20/10/2023	
4.	Проектування	21/10/2023-20/11/2023	
5.	Реалізація	21/11/2023-10/12/2023	
6.	Тестування	10/12/2023-18/12/2023	
7.	Документація та завершення	22-28/12/2023	
8.	Надсилання фінальної версії курсової роботи на перевірку	28-30/12/2023	
9.	Захист курсової роботи	01/01/2024	
10.			
11.			
12.			
13.			
14.			
15.			
16.			
17.			
18.			

Студент Н. Одемчук
(підпис)

Назар Одемчук
(Ім'я ПРІЗВИЩЕ)

Керівник Валерій Колеснік
(підпис)

Валерій Колеснік
(Ім'я ПРІЗВИЩЕ)

«30» грудня 2023 р.

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

Тема IRC Client

Курсова робота

З дисципліни «Технології розроблення програмного забезпечення»

Керівник

ас. Колеснік В. М.

«Допущений до захисту»

(Особистий підпис керівника)

« » _____ 2023р.

Захищений з оцінкою

(оцінка)

Члени комісії:

(особистий підпис)

(особистий підпис)

Виконавець

ст. Одемчук Н. О.

залікова книжка № ІА – 1222

гр. ІА-12

(особистий підпис виконавця)

«30» грудня 2023р.

(розшифровка підпису)

(розшифровка підпису)

Київ – 2023

ЗМІСТ

ВСТУП	3
1 ПРОЄКТУВАННЯ СИСТЕМИ	4
1.1. Огляд існуючих рішень	4
1.2. Загальний опис проєкту	6
1.3. Вимоги до застосунків системи	7
1.3.1. Функціональні вимоги до системи	7
1.3.2. Нефункціональні вимоги до системи	8
1.4. Сценарії використання системи	9
1.5. Концептуальна модель системи	29
1.6. Вибір мови програмування та середовища розробки	34
1.7. Проєктування розгортання системи	37
2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ	46
2.1. Архітектура системи	46
2.1.1. Специфікація системи	46
2.1.2. Вибір та обґрунтування патернів реалізації	59
2.2. Інструкція користувача	66
ВИСНОВКИ	69
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ	71
ДОДАТКИ	72

ВСТУП

У сучасному світі програмне забезпечення стає невід'ємною частиною повсякденного життя, розвивається і змінюється відповідно до потреб користувачів. Однією з популярних форм комунікації в інтернеті є IRC (Internet Relay Chat), яка дозволяє взаємодіяти між користувачами у режимі реального часу. IRC (Internet Relay Chat) клієнт - це програмне забезпечення, призначене для взаємодії користувачів з протоколом IRC. IRC є стандартом обміну текстовими повідомленнями через мережу Інтернет. Клієнт IRC дозволяє користувачам приєднуватися до різних каналів чату, обмінюватися приватними повідомленнями, а також брати участь в групових обговореннях. Розробка IRC клієнта – це важливий аспект для тих, хто бажає створити ефективний і зручний інструмент для спілкування через IRC протокол. Мета курсової роботи – вивчення технологій розроблення програмного забезпечення з фокусом на IRC клієнтах та їх функціональних можливостях. У процесі дослідження планується розгляд таких аспектів, як архітектура програмного забезпечення, використання мережевих технологій, інтерфейсного дизайну, патернів проєктування та UML діаграми для опису систем. Додатково буде вивчено особливості протоколу IRC і його роль у взаємодії між клієнтом і сервером під час обміну повідомленнями. Аналіз популярних IRC клієнтів та їх функціональності допоможе визначити основні вимоги до розробки власного клієнта. Розгляд різних мов програмування та інструментів розробки дозволить вибрати оптимальні засоби для створення IRC клієнта. Важливо також розглянути можливості розширення та підтримки клієнта для забезпечення його актуальності та конкурентоспроможності. Результатом курсової роботи буде спроектований та розроблений IRC клієнт, який відповідає сучасним вимогам та забезпечує користувачам зручність та ефективність в комунікації через мережу IRC.

1 ПРОЄКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

Розглянемо декілька вже існуючих IRC клієнтів:

1. HexChat:

- HexChat є відкритим IRC-клієнтом, який користується популярністю на платформах Windows та Linux.
- Його активна спільнота та постійні оновлення забезпечують стабільну роботу та вдосконалення функціоналу.
- HexChat підтримує різні можливості налаштувань та має інтерфейс, що дозволяє користувачам легко взаємодіяти з IRC.

2. Quassel IRC:

- Quassel IRC виділяється завдяки своїй унікальній можливості зберігання чатів на сервері для зручного доступу з різних пристроїв.
- Його клієнт-серверна архітектура дозволяє зберігати дані та з'єднуватися з різних пристроїв для тривалого обміну повідомленнями.
- Quassel IRC підтримує безліч розширень та налаштувань, що робить його гнучким та потужним клієнтом.

3. XChat:

- XChat, доступний для Linux та Windows, надає користувачам можливість вибору між безкоштовною та комерційною версією.
- Його інтуїтивний інтерфейс та розширені функції роблять його зручним використанням як для новачків, так і для досвідчених користувачів.
- XChat підтримує широкий спектр плагінів і скриптів, що розширює можливості клієнта.

4. IRSSI:

- IRSSI вирізняється своєю текстовою орієнтацією та роботою в терміналі, надаючи зручний інтерфейс для користувачів командного рядка.
- Цей клієнт популярний серед фанатів текстових інтерфейсів за своєю легкістю та можливістю використовувати його в системах без графічного інтерфейсу.
- IRSSI також володіє розширеною системою плагінів, яка дозволяє користувачам налаштовувати його під свої потреби.

5. WeeChat:

- WeeChat є легковаговим клієнтом для терміналу, який підтримує різні протоколи, включаючи IRC.
- Завдяки своїй модульній структурі, WeeChat може бути розширений за допомогою плагінів для додаткових функцій та можливостей.
- Його активна спільнота та постійні оновлення роблять WeeChat одним із виборів для тих, хто шукає простий та функціональний клієнт.

6. mIRC:

- mIRC є комерційним IRC-клієнтом, який володіє великою популярністю серед користувачів Windows.
- З його допомогою можна легко підключатися до різних серверів та каналів, використовуючи простий та зручний інтерфейс.
- mIRC також підтримує скриптинг, що дозволяє користувачам створювати власні скрипти для автоматизації різних завдань.

7. Kiwi IRC:

- Kiwi IRC відзначається тим, що він пропонує веб-клієнт, який можна використовувати безпосередньо в браузері.
- Зручність установки та використання робить Kiwi IRC зручним вибором для тих, хто шукає швидкий доступ до IRC з будь-якого пристрою.
- Його інтуїтивний інтерфейс та можливість налаштування роблять його популярним серед користувачів, які цінують простоту.

8. KVirc:

- KVirc є безкоштовним IRC-клієнтом, який вирізняється своєю розширеністю функцій та гнучкістю налаштувань.
- Цей клієнт надає багато можливостей для адаптації під конкретні потреби користувача, що робить його важливим інструментом для досвідчених користувачів.
- KVirc підтримує різні теми оформлення та інші елементи налаштувань для персоналізації інтерфейсу.

9. Colloquy:

- Colloquy є IRC-клієнтом для macOS, який відзначається своєю інтеграцією з іншими службами, такими як Bonjour та IRC.
- З його допомогою користувачі можуть легко об'єднати різні розмови та взаємодіяти з друзями через різні платформи.
- Colloquy також підтримує зовнішні плагіни для розширення функціоналу та надання додаткових можливостей.

1.2. Загальний опис проєкту

Проект є клієнтом для IRC-чатів, спрямованим на забезпечення користувачам зручного та налаштованого інтерфейсу для спілкування в мережах IRC. Одна з ключових можливостей цього клієнта - можливість вказати адресу сервера та порт для з'єднання, що дозволяє користувачам гнучко налаштовувати параметри підключення.

Основні функції проєкту включають базові команди для роботи з IRC, такі як підключення до чату, створення нового чату, встановлення імені користувача, можливість реєстрації облікового запису, а також отримання допомоги щодо

доступних команд та функціоналу. Клієнт створений з орієнтацією на простоту використання, дозволяючи користувачам ефективно взаємодіяти з IRC-мережами навіть при обмеженому досвіді використання подібного програмного забезпечення.

Окрім того, клієнт підтримує отримання метаданих про канал, що дозволяє користувачам отримувати додаткову інформацію про обрані чати, наприклад, список учасників та інші характеристики. Це допомагає поліпшити взаємодію користувачів з чатами та забезпечує більший контроль над обраною IRC-мережею.

1.3. Вимоги до застосунків системи

1.3.1. Функціональні вимоги до системи

Функціональні вимоги до системи включають:

1. Підключення до IRC-мережі:

- Система повинна дозволяти користувачам вказувати адресу та порт сервера для підключення до IRC-мережі.

2. Створення та приєднання до чатів:

- Користувачі повинні мати можливість створювати нові чати та приєднуватися до існуючих чатів за допомогою відповідних команд чи інтерфейсу.

3. Базові команди IRC:

- Система повинна підтримувати базові команди IRC, такі як надсилання повідомлень, зміна імені користувача, підключення та реєстрація облікового запису.

4. Отримання метаданих про канали:

- Клієнт повинен отримувати та відображати метадані про канали, такі як тема чату, список учасників та інші характеристики.

5. Модульність та розширюваність:

- Система повинна бути модульною, з можливістю легкого розширення функціоналу через плагіни чи додаткові модулі.

1.3.2. Нефункціональні вимоги до системи

Нефункціональні вимоги до системи включають:

1. Ефективність та продуктивність:

- Забезпечення ефективної роботи системи з мінімальними затримками під час обміну повідомленнями.

2. Масштабованість:

- Здатність системи обслуговувати різні кількості користувачів та каналів, забезпечуючи високу продуктивність.

3. Безпека:

- Забезпечення конфіденційності та цілісності даних, використовуючи методи шифрування та інші засоби безпеки.

4. Сумісність:

- Сумісність системи з різними операційними системами (Windows, Linux, macOS) та веб-браузерами для веб-версії.

5. Інтерфейс користувача:

- Забезпечення інтуїтивно зрозумілого та зручного інтерфейсу користувача для легкої взаємодії з основними функціями системи.

6. Доступність:

- Забезпечення доступності системи для користувачів з різним рівнем досвіду, включаючи новачків та досвідчених користувачів IRC.

7. Надійність та стабільність:

- Забезпечення стабільної роботи системи та відсутності частих відмов чи помилок.

1.4. Сценарії використання системи

UseCase – це текстовий опис сукупності сценаріїв, що виконуються користувачем при роботі з системою для досягнення певної мети [1].

Сценарій – послідовність дій при взаємодії користувача із системою для виконання певної операції [1].

Сценарії використання системи зображають на діаграмі варіантів використання (англ. Use Case Diagram).

Діаграма варіантів використання UML відображає взаємозв'язки між акторами та варіантами використання в межах системи. Їх часто використовують для:

- Надання огляду всіх або частини вимог до використання для системи або організації у формі суттєвої моделі (Constantine and Lockwood 1999, Ambler 2004) або бізнес-моделі (Rational Corporation 2002);
- Комунікації щодо обсягу проекту розробки;
- Моделювання аналізу вимог до використання у формі моделі варіантів використання системи (Cockburn 2001).

Модель варіантів використання складається з однієї чи декількох діаграм варіантів використання та будь-якої допоміжної документації, такої як специфікації варіантів використання та визначення акторів. У більшості моделей варіантів використання специфікації варіантів використання, як правило, є основним артефактом, а діаграми варіантів використання UML виконують підтримуючу роль як клей, який утримує вашу модель вимог разом. Моделі варіантів використання слід розробляти з точки зору зацікавлених сторін вашого проекту, а не з (часто технічної) точки зору розробників [2].

Сценарії використання системи зображено на діаграмі варіантів використання, яка знаходиться у Додатку А.

Опис сценаріїв варіантів використання:

- 1) Сценарій використання: Надсилання повідомлення в чат-кімнату IRC (рис. 1.4.1)

Попередні умови: Користувач увійшов до IRC-клієнта, приєднався до

чат-кімнати, і чат-кімната активна і доступна.

Пост умови: Після успішного виконання, повідомлення користувача надсилається в чат-кімнату і стає видимим для інших учасників. У протилежному випадку стан системи залишається незмінним.

Учасники: Користувач, сервер IRC, учасники чат-кімнати.

Короткий опис: Цей сценарій використання описує процес користувача, який надсилає повідомлення в чат-кімнату IRC, що дозволяє йому спілкуватися з іншими учасниками чат-кімнати.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли користувач бажає надіслати повідомлення в чат-кімнату IRC.

1. Користувач вводить своє повідомлення в поле введення тексту чату.
2. Користувач відправляє повідомлення на відправку.
3. IRC-клієнт перевіряє, чи повідомлення в межах обмежень на кількість символів та містить дійсний вміст.
4. Якщо повідомлення є дійсним, IRC-клієнт надсилає повідомлення на сервер IRC.
5. Сервер IRC транслює повідомлення всім учасникам чат-кімнати, зробивши його видимим для інших.

Виключення:

Виключення № 1:

Недійсне повідомлення. Якщо система виявляє, що повідомлення користувача є недійсним, наприклад, містить заборонений вміст або перевищує обмеження на кількість символів, вона відображає повідомлення про помилку. Користувач може відредагувати повідомлення та повторно надіслати його. Якщо повідомлення систематично порушує правила, користувач може стати об'єктом санкцій або вилучення з чат-кімнати.

Виключення № 2:

Втрата з'єднання з сервером IRC. Якщо IRC-клієнт втрачає з'єднання з

сервером під час відправки повідомлення, система відображає повідомлення про помилку. Користувач може вибрати перепідключення або вийти з чат-кімнати.

Примітки:

- Від користувачів очікується дотримання правил поведінки в чат-кімнаті і умов обслуговування мережі IRC.
- Вміст повідомлення може підлягати модерації або фільтрації для дотримання правил і рекомендацій чат-кімнати.
- Успішне виконання цього сценарію використання передбачає успішний вхід користувача до IRC-клієнта та приєднання до чат-кімнати.

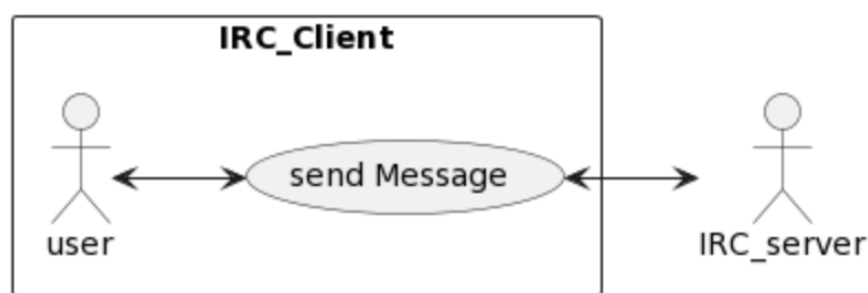


Рисунок 1.4.1 – Діаграма варіантів використання для сценарія надсилання повідомлення в чат-кімнату IRC [Додаток А]

2) Сценарій використання: Приєднання до чат-кімнати IRC (рис. 1.4.2)

Попередні умови: Користувач увійшов в IRC-клієнт.

Пост умови: У разі успішного виконання користувач стає учасником обраної чат-кімнати IRC. У протилежному випадку стан системи залишається незмінним.

Учасники: Користувач, сервер IRC, учасники чат-кімнати.

Короткий опис: Цей сценарій використання описує процес приєднання користувача до чат-кімнати IRC, що дозволяє йому приймати участь у обговореннях і взаємодіяти з іншими учасниками обраної чат-кімнати.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли користувач бажає приєднатися до чат-кімнати IRC.

1. Користувач ініціює дію приєднання до чат-кімнати, зазвичай вводячи назву чат-кімнати або вибираючи її зі списку доступних кімнат.
2. IRC-клієнт надсилає запит на сервер IRC для приєднання до обраної чат-кімнати.
3. Сервер IRC обробляє запит і перевіряє, чи користувач має дозвіл на приєднання до чат-кімнати.
4. Якщо користувачу дозволено приєднатися, сервер IRC додає його до списку учасників чат-кімнати.
5. IRC-клієнт оновлює інтерфейс користувача, відображаючи чат-кімнату та її поточні розмови.

Виключення:

Виключення № 1:

Невірна назва чат-кімнати. Якщо користувач вводить невірну або неіснуючу назву чат-кімнати, IRC-клієнт відображає повідомлення про помилку і просить користувача ввести правильну назву чат-кімнати.

Виключення № 2:

Доступ заборонено. Якщо користувач не має необхідних дозволів для приєднання до певної чат-кімнати (наприклад, вона захищена паролем або обмежена), сервер IRC відхиляє запит і повідомляє користувача повідомленням про помилку.

Виключення № 3:

Помилка з'єднання. У разі відключення від сервера IRC під час процесу приєднання, IRC-клієнт відображає повідомлення про помилку, і користувач може спробувати перепідключитися або вийти з процесу.

Примітки:

- Після приєднання користувачам може знадобитися дотримуватися правил та рекомендацій чат-кімнати, і ці правила можуть варіюватися від кімнати до кімнати.
- Деякі чат-кімнати можуть вимагати пароль для входу, який користувач повинен надати під час процесу приєднання, якщо це необхідно.
- Успішне виконання цього сценарію використання залежить від того, що користувач увійшов в IRC-клієнт і має активне з'єднання з сервером IRC.

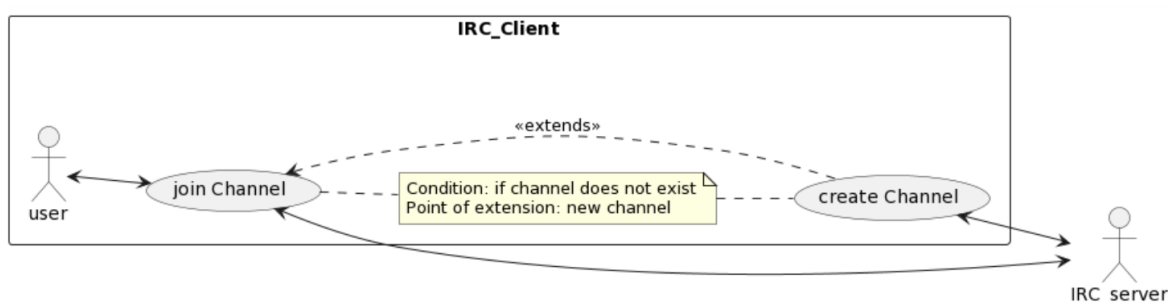


Рисунок 1.4.2 – Діаграма варіантів використання для сценарія приєднання до чат-кімнати IRC [Додаток А]

3) Сценарій використання: Вихід з чат-кімнати IRC (рис. 1.4.3)

Попередні умови: Користувач увійшов в IRC-клієнт і є учасником обраної чат-кімнати IRC.

Пост умови: У разі успішного виконання користувач покидає чат-кімнату і більше не є учасником. У протилежному випадку користувач залишається в чат-кімнаті.

Учасники: Користувач, сервер IRC, учасники чат-кімнати.

Короткий опис: Цей сценарій використання описує процес виходу користувача з чат-кімнати IRC, що дозволяє йому закінчити участь в обговореннях чат-кімнати та вийти з неї.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли користувач бажає

покинути чат-кімнату IRC.

1. Користувач ініціює дію виходу з чат-кімнати, зазвичай вибираючи відповідний пункт у користувацькому інтерфейсі IRC-клієнта.
2. IRC-клієнт надсилає запит до сервера IRC для виходу з обраної чат-кімнати.
3. Сервер IRC обробляє запит та видаляє користувача зі списку учасників чат-кімнати.
4. IRC-клієнт оновлює інтерфейс користувача, видаляючи чат-кімнату та її поточні розмови.

Виключення:

Виключення № 1:

Помилка з'єднання. У разі відключення від сервера IRC під час процесу виходу, IRC-клієнт відображає повідомлення про помилку. Користувач може спробувати перепідключитися або вийти з процесу.

Примітки:

- Вихід з чат-кімнати зазвичай є простою операцією і не вимагає спеціальних дозволів.
- Успішне виконання цього сценарію використання залежить від того, що користувач увійшов в IRC-клієнт і має активне з'єднання з сервером IRC.
- Після виходу з чат-кімнати користувач не отримує оновлень або повідомлень від цієї чат-кімнати, поки не повернеться до неї.

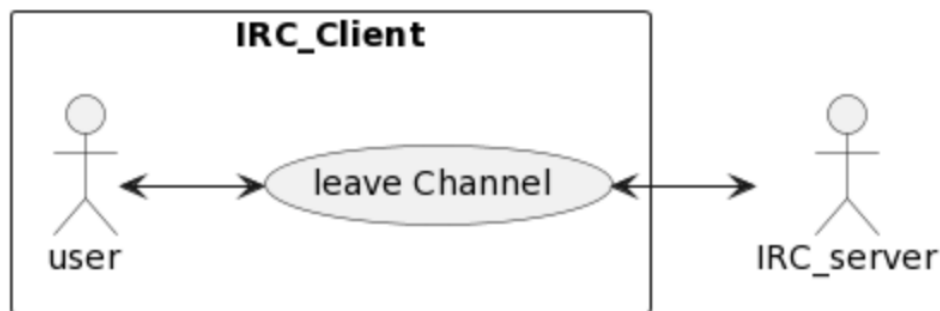


Рисунок 1.4.3 – Діаграма варіантів використання для сценарія вихід з чат-кімнати IRC [Додаток А]

4) Сценарій використання: Зміна нікнейму користувача на IRC (рис. 1.4.4)

Попередні умови: Користувач увійшов в IRC-клієнт та має активне з'єднання з сервером IRC.

Пост умови: У разі успішного виконання, система оновлює нікнейм користувача на IRC-мережі. У протилежному випадку стан системи залишається незмінним.

Учасники: Користувач, сервер IRC.

Короткий опис: Цей сценарій використання описує процес зміни нікнейму користувача на IRC-мережі, що дозволяє користувачеві представлятися під обраним ім'ям в чатах та обговореннях.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли користувач бажає змінити свій нікнейм на IRC-мережі.

1. Користувач ініціює дію зміни нікнейму, використовуючи відповідну команду чи інтерфейс в IRC-клієнті.
2. IRC-клієнт надсилає запит на сервер IRC для оновлення нікнейму користувача.
3. Сервер IRC перевіряє, чи новий нікнейм доступний та чи він вільний для використання.

4. Якщо новий нікнейм доступний, сервер IRC змінює нікнейм користувача та повідомляє IRC-клієнт про успішну операцію.

5. IRC-клієнт оновлює інтерфейс користувача, відображаючи оновлений нікнейм.

Виключення:

Виключення № 1:

Недоступність нового нікнейму. Якщо обраний користувачем новий нікнейм вже зайнятий іншим користувачем на IRC-мережі, сервер IRC повідомляє про конфлікт та просить користувача обрати інший нікнейм.

Виключення № 2:

Помилка з'єднання. У разі відключення від сервера IRC під час процесу зміни нікнейму, IRC-клієнт відображає повідомлення про помилку, і користувач може спробувати перепідключитися або вийти з процесу.

Примітки:

- Зміна нікнейму може вплинути на ідентифікацію користувача в чатах та обговореннях, і новий нікнейм повинен відповідати правилам IRC-мережі.

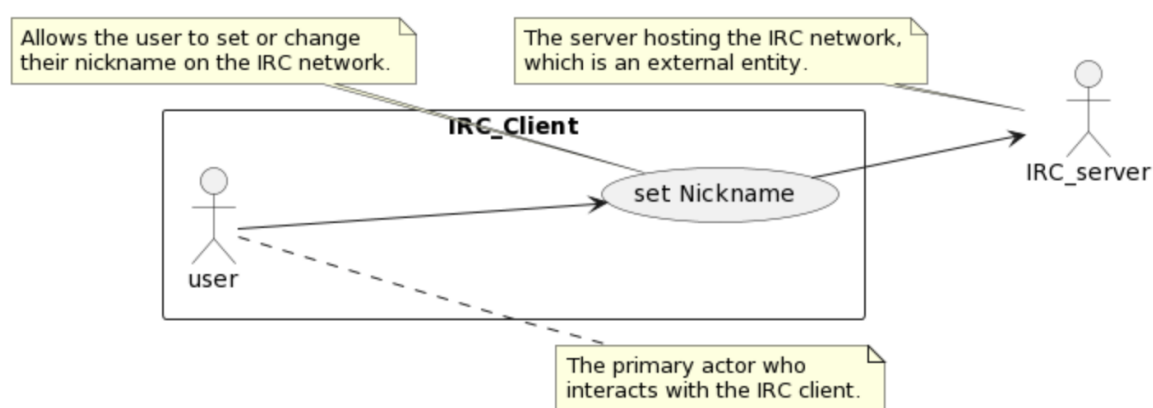


Рисунок 1.4.4 – Діаграма варіантів використання для сценарія зміна нікнейму користувача на IRC [Додаток А]

5) Сценарій використання: Під'єднання до IRC серверу (рис. 1.4.5)

Попередні умови: Користувач запустив IRC-клієнт та має активне з'єднання з Інтернетом.

Пост умови: У разі успішного виконання, користувач встановлює з'єднання з IRC сервером та може брати участь у чатах та обговореннях. У протилежному випадку стан системи залишається незмінним.

Учасники: Користувач, сервер IRC.

Короткий опис: Цей сценарій використання описує процес під'єднання користувача до IRC серверу для участі в чатах та обговореннях на IRC-мережі.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли користувач бажає під'єднатися до IRC серверу.

1. Користувач ініціює дію під'єднання до IRC серверу, вибираючи опцію "Connect" або вводячи адресу сервера та порт вручну.
2. IRC-клієнт надсилає запит на сервер IRC для встановлення з'єднання.
3. Сервер IRC приймає запит та ініціює процес з'єднання з користувачем.
4. Якщо з'єднання успішно встановлено, сервер IRC відправляє підтвердження, і користувач отримує доступ до IRC-мережі.
5. IRC-клієнт оновлює інтерфейс користувача, відображаючи список доступних чат-кімнат та можливість взаємодії з іншими користувачами.

Виключення:

Виключення № 1:

Помилка з'єднання. У разі невдалого з'єднання з сервером IRC (наприклад, через неправильну адресу сервера або відсутність з'єднання з Інтернетом), IRC-клієнт відображає повідомлення про помилку та просить користувача перевірити параметри з'єднання.

Виключення № 2:

Неуспішне підтвердження сервера. Якщо сервер IRC не може підтвердити ініційоване з'єднання, IRC-клієнт відображає повідомлення про невдачу та пропонує спробувати під'єднатися ще раз.

Примітки:

- Перед під'єднанням до IRC серверу, користувач може мати можливість налаштувати додаткові параметри, такі як ідентифікація, пароль або SSL-з'єднання в залежності від конкретних вимог мережі.

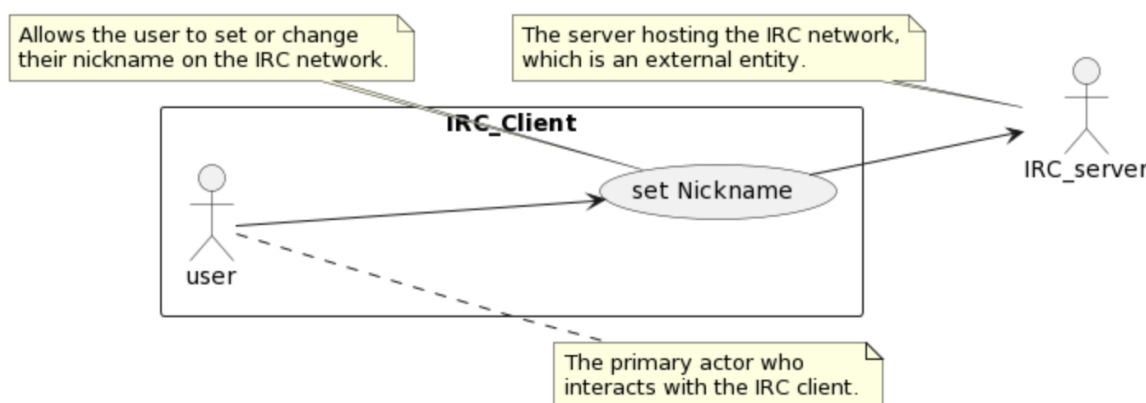


Рисунок 1.4.5 – Діаграма варіантів використання для сценарія під'єднання до IRC серверу [Додаток А]

6) Сценарій використання: Реєстрація облікового запису на IRC (рис. 1.4.6)

Попередні умови: Користувач запустив IRC-клієнт та має активне з'єднання з Інтернетом.

Пост умови: У разі успішного виконання, користувач реєструє свій обліковий запис на IRC-мережі та може використовувати додаткові функціональності. У протилежному випадку стан системи залишається незмінним.

Учасники: Користувач, сервер IRC.

Короткий опис: Цей сценарій використання описує процес реєстрації облікового запису на IRC-мережі, який дозволяє користувачеві мати персональний ідентифікатор та використовувати розширені функціональності.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли користувач бажає зареєструвати свій обліковий запис на IRC-мережі.

1. Користувач ініціює дію реєстрації, вибираючи опцію Register або вводячи відомості для реєстрації через інтерфейс IRC-клієнта.
2. IRC-клієнт надсилає запит на сервер IRC для реєстрації облікового запису.
3. Сервер IRC перевіряє, чи новий обліковий запис доступний та чи він вільний для використання.
4. Якщо новий обліковий запис доступний, сервер IRC відправляє підтвердження ініціювання реєстрації.
5. IRC-клієнт надсилає запит на встановлення нікнейму для облікового запису, і сервер IRC призначає нікнейм користувача.
6. Користувач вводить пароль для облікового запису, який буде використовуватися для подальшого входу на IRC-мережу.
7. Сервер IRC підтверджує успішну реєстрацію облікового запису, і користувач отримує можливість використовувати додаткові функціональності, які доступні зареєстрованим користувачам.

Виключення:

Виключення № 1:

Невдача реєстрації. Якщо сервер IRC відхиляє запит на реєстрацію (наприклад, через використання вже існуючого нікнейму або через відмову мережі), IRC-клієнт відображає повідомлення про помилку та просить користувача спробувати інший нікнейм або інші відомості.

Виключення № 2:

Помилка з'єднання. У разі відключення від сервера IRC під час процесу реєстрації, IRC-клієнт відображає повідомлення про помилку, і користувач може спробувати перепідключитися або вийти з процесу.

Примітки:

- Реєстрація облікового запису може включати додаткові кроки, такі як введення електронної пошти чи інших відомостей для додаткової ідентифікації користувача.

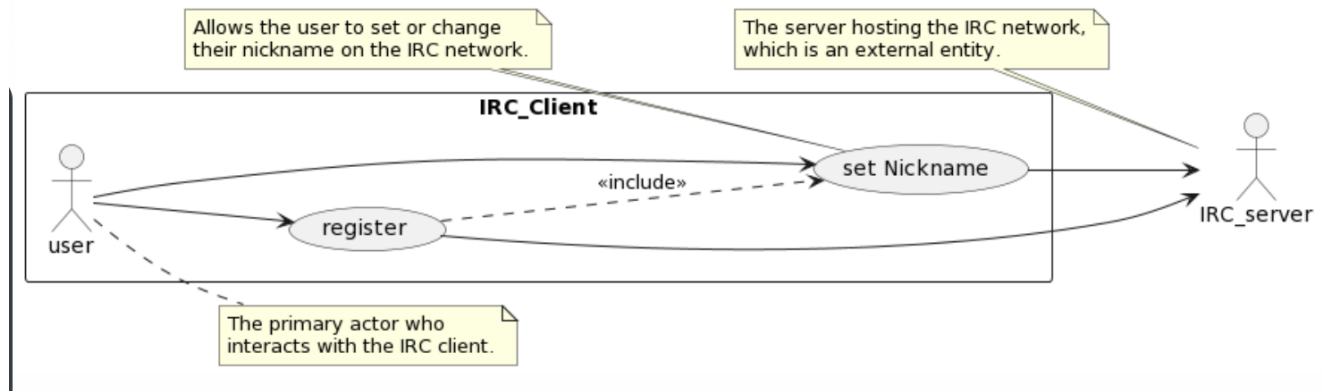


Рисунок 1.4.6 – Діаграма варіантів використання для сценарія реєстрація облікового запису на IRC [Додаток А]

7) Сценарій використання: Отримання повідомлення на IRC (рис. 1.4.7)

Попередні умови: Користувач увійшов в IRC-клієнт та має активне з'єднання з сервером IRC.

Пост умови: Користувач отримав нове повідомлення в чаті на IRC та може переглядати його в інтерфейсі IRC-клієнта. У протилежному випадку стан системи залишається незмінним.

Учасники: Користувач, сервер IRC.

Короткий опис: Цей сценарій використання описує процес отримання повідомлення на IRC, коли інший користувач або бот відправляє повідомлення у спільний чат або особистий канал.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли інший користувач або бот надсилає повідомлення користувачеві на IRC.

1. Сервер IRC отримує повідомлення від іншого користувача або бота.
2. Сервер IRC визначає, в якому чаті або на якому каналі перебуває користувач, який має отримати повідомлення.
3. Сервер IRC відправляє повідомлення до IRC-клієнта користувача.
4. IRC-клієнт відображає нове повідомлення в інтерфейсі користувача.

5. Користувач може переглядати та взаємодіяти з отриманим повідомленням, наприклад, відповідати або іншим чином реагувати на нього.

Виключення:

Виключення № 1:

Неприйняття повідомлення. Якщо з якихось причин повідомлення не може бути доставлено до користувача (наприклад, через відключення користувача від мережі або інші проблеми з'єднання), IRC-клієнт може відобразити повідомлення про неприйняття та сповістити користувача про можливу причину.

Виключення № 2:

Помилка з'єднання. У разі відключення від сервера IRC під час процесу отримання повідомлення, IRC-клієнт відображає повідомлення про помилку, і користувач може спробувати перепідключитися або вийти з процесу.

Примітки:

- Отримані повідомлення можуть бути текстовими, аудіо або мультимедійними та можуть включати різноманітні дані та вкладені елементи.

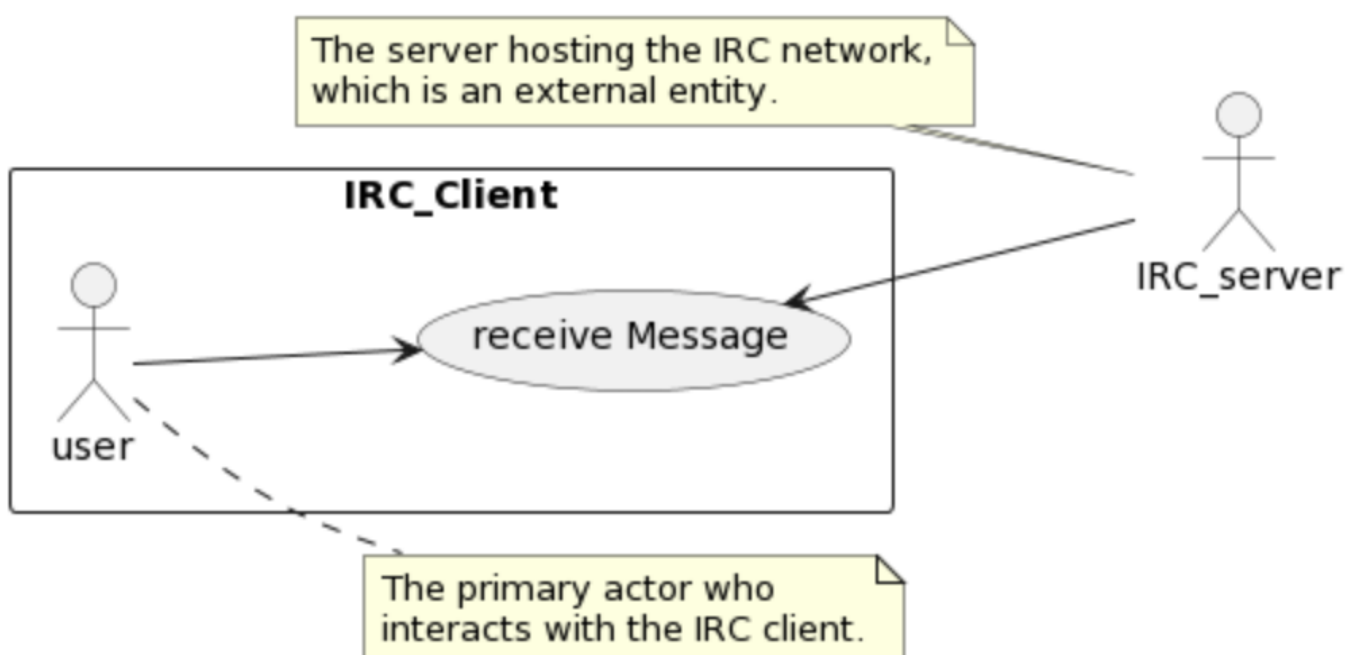


Рисунок 1.4.7 – Діаграма варіантів використання для сценарія отримання повідомлення на IRC [Додаток А]

8) Сценарій використання: Отримання інформації про користувачів на IRC (рис 1.4.8)

Попередні умови: Користувач увійшов в IRC-клієнт та має активне з'єднання з сервером IRC.

Пост умови: Користувач отримав інформацію про інших користувачів, які перебувають на тому ж сервері IRC, і може взаємодіяти з ними. У протилежному випадку стан системи залишається незмінним.

Учасники: Користувач, сервер IRC.

Короткий опис: Цей сценарій використання описує процес отримання інформації про інших користувачів на IRC-мережі, таких як їхні нікнейми, статуси та інші відомості.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли користувач бажає отримати інформацію про інших користувачів на сервері IRC.

1. Користувач ініціює дію, запитуючи сервер IRC про список користувачів або використовуючи відповідну команду (наприклад, /list або /who).
2. IRC-клієнт відправляє запит на сервер IRC для отримання списку користувачів або детальної інформації про них.
3. Сервер IRC відповідає, надсилаючи інформацію про користувачів, які перебувають на тому ж сервері.
4. IRC-клієнт відображає інформацію про користувачів у вікні чата або в інтерфейсі користувача.
5. Користувач може переглядати статус, нікнейми та інші відомості про інших користувачів та взаємодіяти з ними за необхідності.

Виключення:

Виключення № 1:

Помилка отримання інформації. У разі виникнення помилки під час отримання інформації про користувачів (наприклад, через відключення від сервера IRC або інші технічні проблеми), IRC-клієнт відображає повідомлення про помилку та просить користувача спробувати ще раз.

Виключення № 2:

Відсутність інформації. Якщо на сервері IRC відсутні інші користувачі або немає детальної інформації про них, сервер IRC повідомляє IRC-клієнт про це, і користувач отримує відповідне повідомлення.

Примітки:

- Інформація про користувачів може включати їхні статуси (наприклад, online, away, offline), нікнейми, ідентифікатори чат-кімнат, до яких вони приєдналися, та інші відомості, які доступні на сервері IRC.

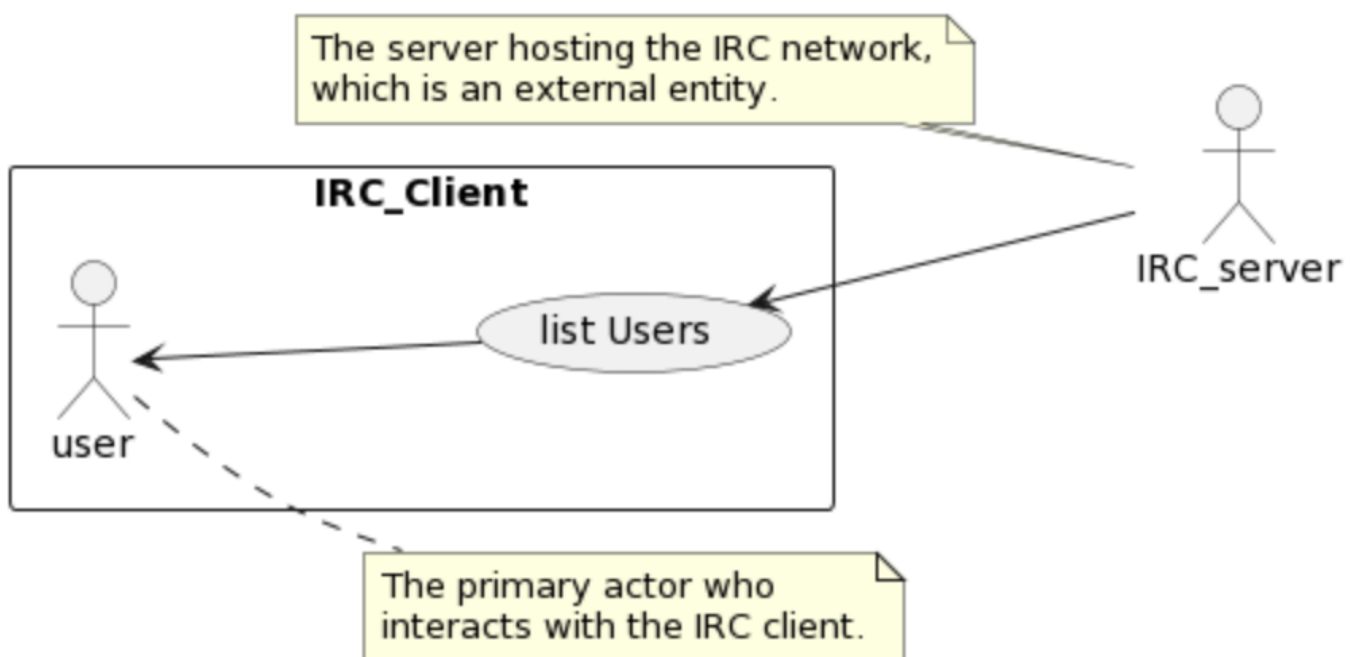


Рисунок 1.4.8 – Діаграма варіантів використання для сценарія отримання інформації про користувачів на IRC [Додаток А]

9) Сценарій використання: Відключення від IRC сервера (рис. 1.4.9)

Попередні умови: Користувач увійшов в IRC-клієнт та має активне з'єднання з сервером IRC.

Пост умови: Користувач вийшов з IRC-клієнта та закрив з'єднання з сервером IRC. У протилежному випадку стан системи залишається незмінним.

Учасники: Користувач, сервер IRC.

Короткий опис: Цей сценарій використання описує процес відключення користувача від сервера IRC, коли він закінчує сесію чи виходить з чат-кімнати.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли користувач бажає вийти з IRC-мережі або відключитися від сервера IRC.

1. Користувач ініціює дію відключення, використовуючи відповідну команду (наприклад, /quit або /disconnect) або вибираючи опцію вийти з IRC-клієнта.
2. IRC-клієнт відправляє запит на сервер IRC для відключення користувача.
3. Сервер IRC отримує запит і обробляє його, видаляючи користувача зі списку активних учасників чат-кімнати чи закриваючи з'єднання.
4. Сервер IRC повідомляє IRC-клієнта про успішне відключення.
5. IRC-клієнт відображає повідомлення про відключення та закриває з'єднання з сервером IRC.

Виключення:

Виключення № 1:

Помилка відключення. У разі виникнення помилки під час процесу відключення (наприклад, через відключення від Інтернету або інші технічні проблеми), IRC-клієнт може відобразити повідомлення про помилку та просити користувача спробувати ще раз або перевірити з'єднання.

Виключення № 2:

Неочікуване відключення. Якщо відключення сталося не через ініціативу користувача (наприклад, через втрату з'єднання або проблеми сервера), IRC-клієнт може відобразити повідомлення про неочікуване

відключення та пропонувати користувачу перепідключитися або вийти з програми.

Примітки:

- Користувач може використовувати команди чи інтерфейс IRC-клієнта для виходу з усіх чат-кімнат перед відключенням від сервера IRC.

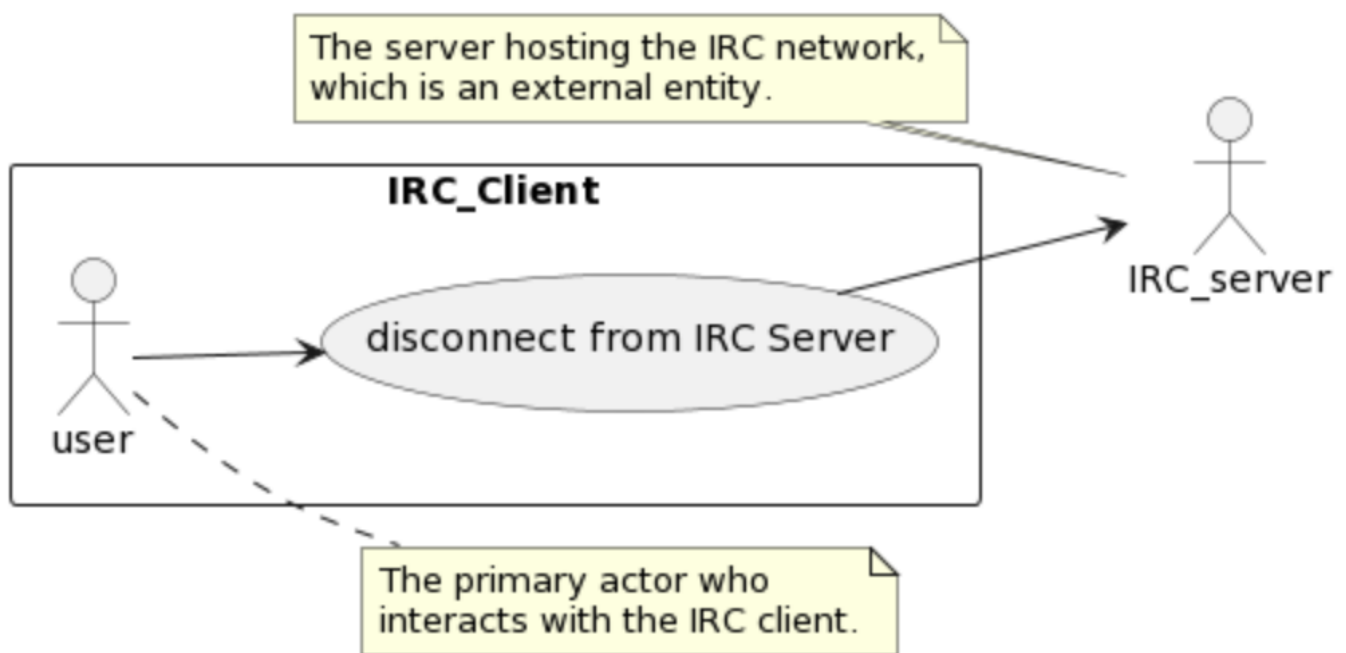


Рисунок 1.4.9 – Діаграма варіантів використання для сценарія відключення від IRC сервера [Додаток А]

10) Сценарій використання: Отримання метаданих про канал на IRC (рис. 1.4.10)

Попередні умови: Користувач увійшов в IRC-клієнт та має активне з'єднання з сервером IRC.

Пост умови: Користувач отримав метадані про обраний чат-канал на IRC та може переглядати їх в інтерфейсі IRC-клієнта. У протилежному випадку стан системи залишається незмінним.

Учасники: Користувач, сервер IRC.

Короткий опис: Цей сценарій використання описує процес отримання метаданих про канал на IRC, таких як тема чату, список учасників, правила та інші важливі відомості.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли користувач бажає отримати додаткову інформацію про обраний чат-канал на IRC.

1. Користувач ініціює дію, використовуючи команду або вибираючи опцію взяти метадані про канал.
2. IRC-клієнт надсилає запит на сервер IRC для отримання метаданих про обраний чат-канал.
3. Сервер IRC перевіряє запит та відправляє метадані про канал на IRC-клієнт.
4. IRC-клієнт відображає метадані про канал у вікні чата або в інтерфейсі користувача.
5. Користувач може переглядати та аналізувати інформацію про канал, таку як тему, список учасників, правила чату та інші відомості.

Виключення:

Виключення № 1:

Помилка отримання метаданих. У разі виникнення помилки під час процесу отримання метаданих про канал (наприклад, через відключення від сервера IRC або інші технічні проблеми), IRC-клієнт може відобразити повідомлення про помилку та просити користувача спробувати ще раз або перевірити з'єднання.

Виключення № 2:

Відсутність метаданих. Якщо на сервері IRC відсутні інші важливі метадані про канал, IRC-клієнт повідомляє користувача про це, і користувач отримує відповідне повідомлення.

Примітки:

- Метадані про канал можуть містити інформацію про тему чату, правила поведінки, перелік учасників, адміністраторів чату та інші важливі

деталі, які допомагають користувачам краще розуміти характер чат-каналу на IRC.

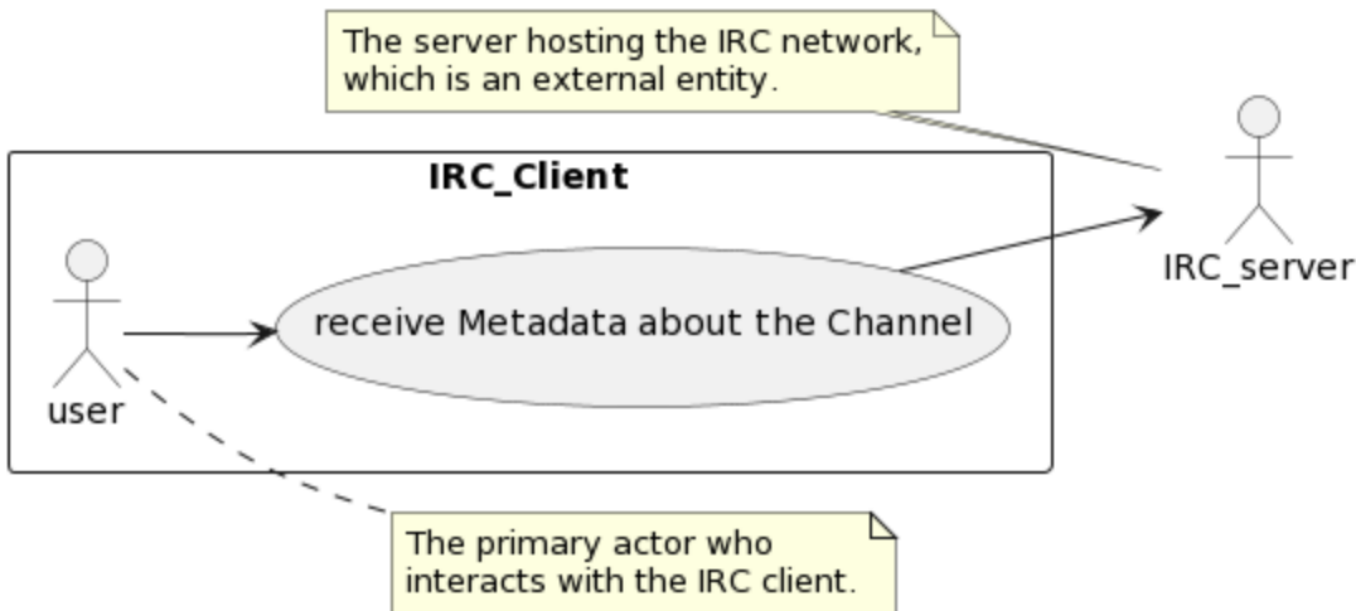


Рисунок 1.4.10 – Діаграма варіантів використання для сценарія отримання метаданих про канал на IRC [Додаток А]

11) Сценарій використання: Отримання допомоги на IRC (рис. 1.4.11)

Попередні умови: Користувач увійшов в IRC-клієнт та має активне з'єднання з сервером IRC.

Пост умови: Користувач отримав допомогу чи інструкції з використання IRC-клієнта та мережі. У протилежному випадку стан системи залишається незмінним.

Учасники: Користувач, сервер IRC.

Короткий опис: Цей сценарій використання описує процес отримання допомоги чи інструкцій з використання IRC-клієнта чи мережі.

Основна послідовність подій:

Цей сценарій використання розпочинається тоді, коли користувач потребує допомоги чи інструкцій з використання IRC-клієнта чи мережі.

1. Користувач ініціює дію, використовуючи команду або вибираючи опцію отримання допомоги (наприклад, /help або /assist).
2. IRC-клієнт надсилає запит на сервер IRC, який може бути направлений до бота допомоги чи до спеціального каналу із питаннями та відповідями.
3. Сервер IRC або бот надсилає користувачеві інформацію, допомогу чи відповідь на його запитання.
4. IRC-клієнт відображає інструкції чи відповіді у вікні чата або в інтерфейсі користувача.
5. Користувач може прочитати, розібратися та використовувати надані відомості для поліпшення свого досвіду використання IRC.

Виключення:

Виключення № 1:

Помилка отримання допомоги. У разі виникнення помилки під час процесу отримання допомоги (наприклад, через відключення від сервера IRC або інші технічні проблеми), IRC-клієнт може відобразити повідомлення про помилку та просити користувача спробувати ще раз або перевірити з'єднання.

Виключення № 2:

Відсутність допомоги. Якщо вказаний бот чи канал із питаннями та відповідями відсутній або не може надати допомогу, користувач отримує повідомлення про відсутність допомоги та може спробувати інші засоби або ресурси для отримання допомоги.

Примітки:

Допомога може включати інструкції з використання команд, пояснення функціоналу IRC-клієнта, вказівки щодо налаштувань та будь-яку іншу інформацію, яка допомагає користувачеві ефективно взаємодіяти з IRC-мережею.

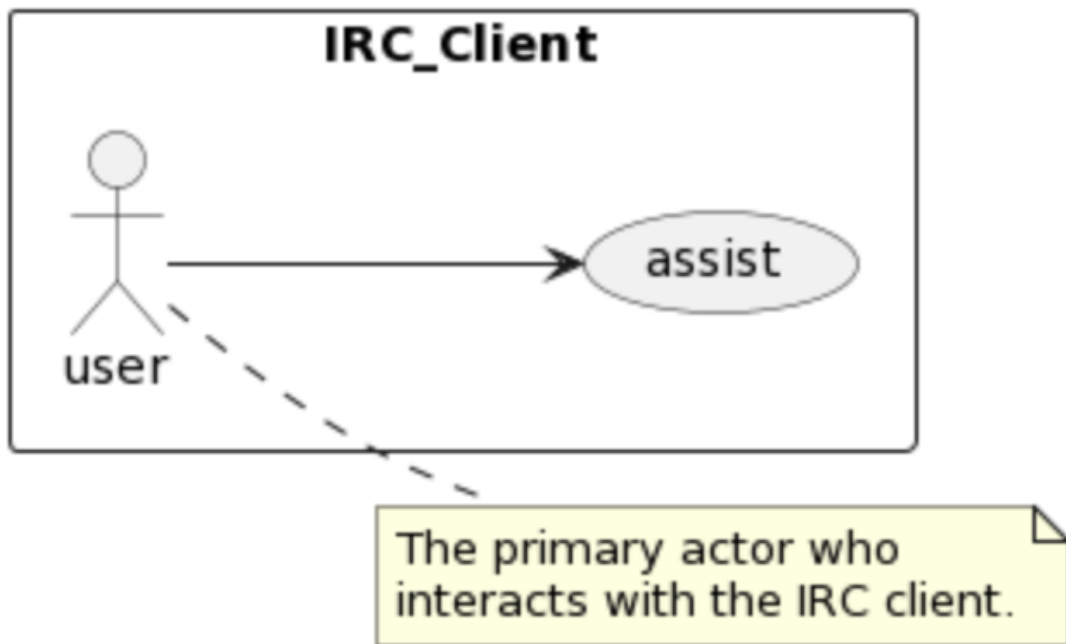


Рисунок 1.4.11 – Діаграма варіантів використання для сценарія отримання допомоги на IRC [Додаток А]

1.5. Концептуальна модель системи

Концептуальна (змістовна) модель — це абстрактна модель, що визначає структуру модельованої системи, властивості її елементів і причинно-наслідкові зв'язки, властиві системі і суттєві для досягнення мети моделювання [3].

В даній роботі концептуальна модель системи описана за допомогою діаграми класів мови UML.

Діаграма класів (англ. *Class diagram*) — статичне представлення структури моделі в UML. Відображає статичні (декларативні) елементи, такі як: класи, типи даних, їх зміст та відношення. Діаграма класів може містити позначення для пакетів та може містити позначення для вкладених пакетів. Також, діаграма класів може містити позначення деяких елементів поведінки, однак їх динаміка розкривається в інших типах діаграм [4].

Діаграма класів в даній роботі зображена в Додатку Б.

Діаграма класів використовується для візуалізації структури програмного коду та взаємодій між його складовими. Давайте розглянемо основні класи та їхні взаємовідношення на цій діаграмі:

- Базова конфігурація сервера (ServerConfig) для зберігання історії чатів (рис 1.5.1):
 - ServerConfig – містить адресу сервера (address), порт (port), та максимальний розмір буфера (max_buffer_size).
- Клас сервера (Server) для зберігання історії чатів (рис 1.5.1):
 - Server – ініціалізується з екземпляром ServerConfig.
 - Має методи start для початку роботи сервера та handle_client для взаємодії з клієнтом.

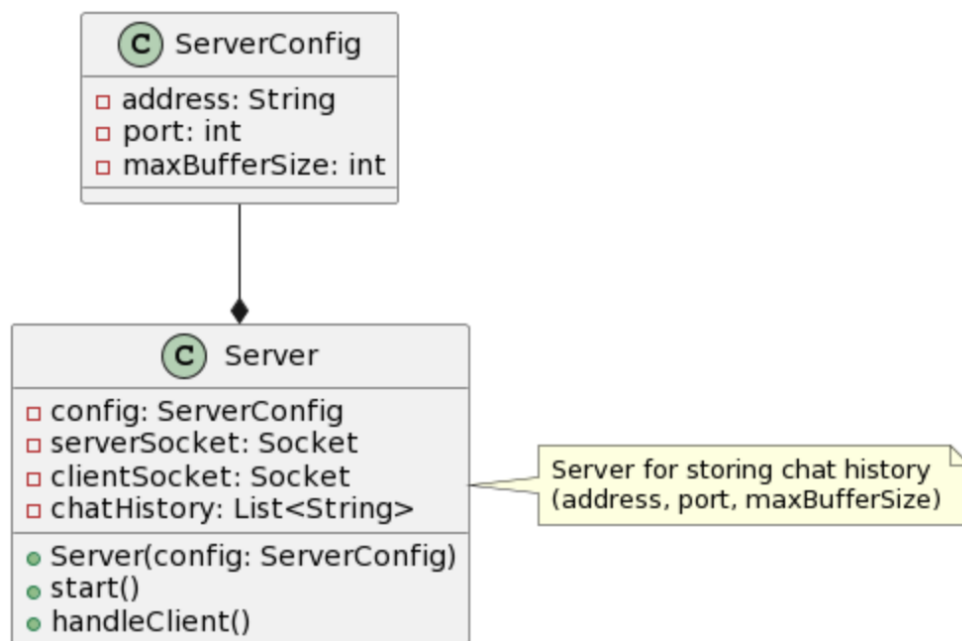


Рисунок 1.5.1 – Діаграма класів для Server та ServerConfig [Додаток Б]

- Клас клієнта IRC (IRCCClient) (рис. 1.5.2):
 - IRCCClient - представляє клієнт IRC.

- Володіє об'єктами конфігурації сервера (server_config_validator), об'єктом для взаємодії з сервером (bridge), об'єктом обробника команд (command_handler), та об'єктом для ведення історії (history_manager).
- Використовує об'єкт ConfigValidation для перевірки конфігурації сервера.
- Пов'язаний з об'єктом MessageHandler для обробки команд користувача.
- Здійснює взаємодію з сервером через IRCConnection.
- Взаємодіє з GUI для відображення.
- Композиція між IRCClient та ConfigValidation, MessageHandler, OutputValidation, HistoryServerHandler, IRCConnection, GUI (рис. 1.5.2):
 - IRCClient володіє об'єктами ConfigValidation, MessageHandler, OutputValidation, HistoryServerHandler, IRCConnection, та GUI.
- Графічний інтерфейс (GUI):
 - GUI – відповідає за графічний інтерфейс для взаємодії з користувачем.

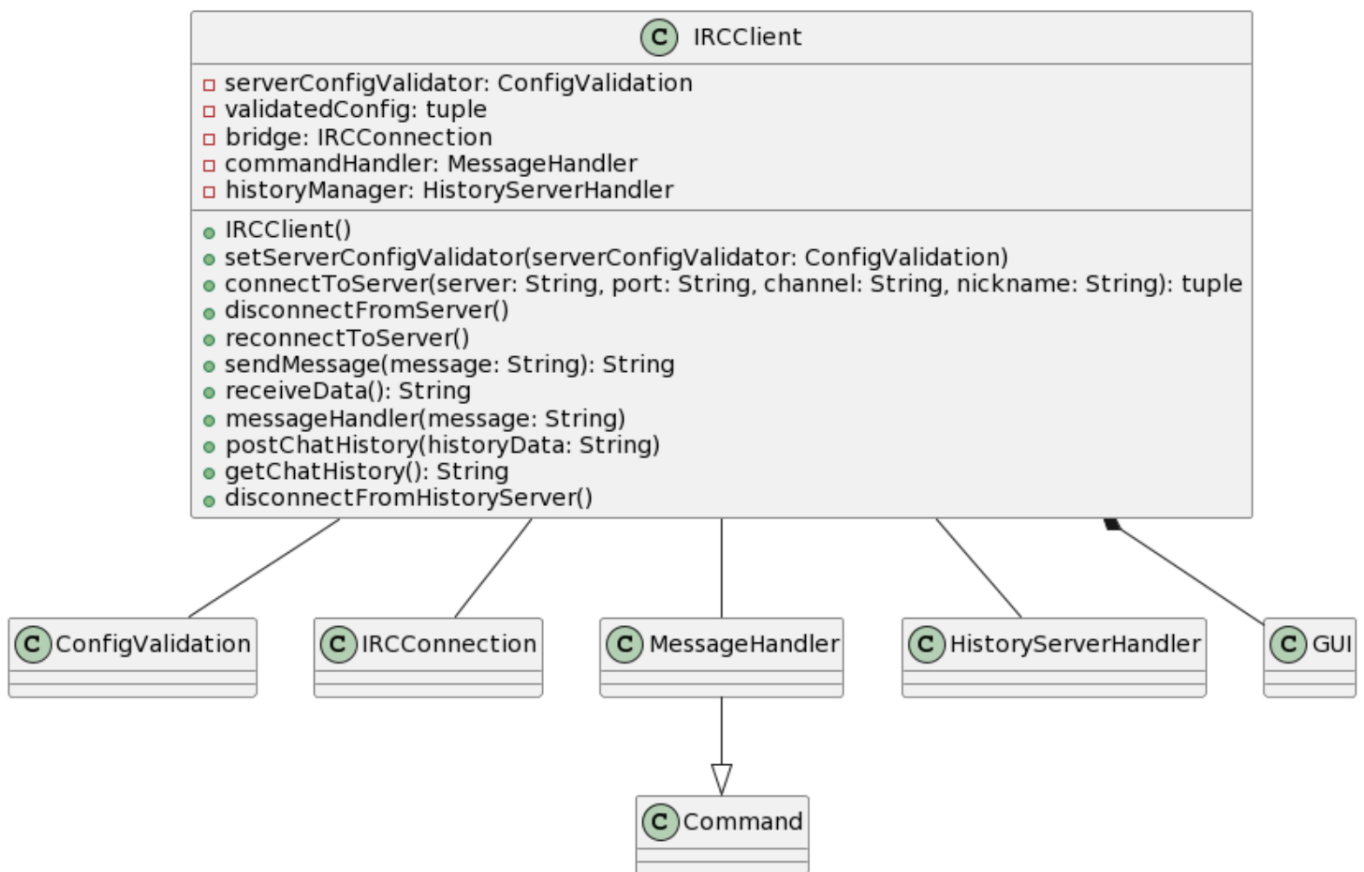


Рисунок 1.5.2 – Діаграма класів для IRCClient [Додаток Б]

- Клас обробника повідомлень (MessageHandler) (рис. 1.5.3):
 - MessageHandler - відповідає за виконання IRC-команд на основі введеного користувачем повідомлення.
 - Взаємодіє з різними командами (а саме з JoinChannelCommand, LeaveChannelCommand, DisplayHelpCommand, ChannelInfoCommand, ListUsersCommand, ChangeNicknameCommand, QuitCommand, CompositeCommand) та обробляє їх.
- Композитна команда (CompositeCommand) (рис. 1.5.3):
 - CompositeCommand – дозволяє виконувати кілька команд одночасно.
 - Різні команди (JoinChannelCommand, LeaveChannelCommand, DisplayHelpCommand, ChannelInfoCommand, ListUsersCommand, ChangeNicknameCommand, QuitCommand): – Різні класи команд, які успадковують від абстрактного класу Command та виконують конкретні дії.
- Команда (Command) (рис. 1.5.3):
 - Абстрактний клас Command – визначає інтерфейс для різних команд.
 - Реалізує загальні частини логіки для всіх команд.
- Команда об'єднання кількох команд (CompositeCommand) (рис. 1.5.3):
 - Реалізує виконання декількох команд одночасно. Команда (Command) :
 - Абстрактний клас Command – визначає інтерфейс для різних команд.
 - Реалізує загальні частини логіки для всіх команд.

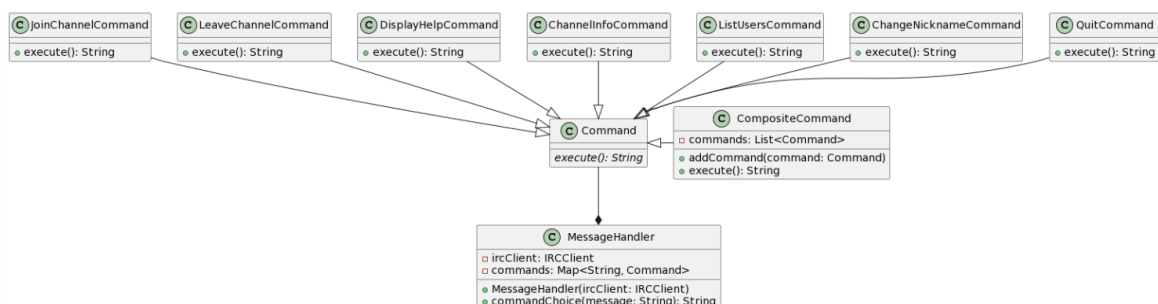


Рисунок 1.5.3 – Діаграма класів для MessageHandler [Додаток Б]

- Валідація виводу (OutputValidation) (рис. 1.5.4):
 - OutputValidation – відповідає за перевірку виводу перед відображенням його користувачеві.
- Вхідна валідація (InputValidation) (рис. 1.5.4):
 - Відповідає за перевірку введення користувача.
- Клас об'єкта історії (HistoryServerHandler) (рис. 1.5.4):
 - HistoryServerHandler – взаємодіє з історією сервера (зберігання та передача історії).
- Валідація конфігурації сервера (ConfigValidation):
 - Абстрактний клас ConfigValidation (рис. 1.5.4) – визначає інтерфейс для класів, які виконують валідацію конфігурації сервера.
 - Класи BasicConfigValidation та AdvancedConfigValidation реалізують цей інтерфейс (рис. 1.5.4).
- IRC-з'єднання (IRCConnection) (рис. 1.5.4):
 - Абстрактний клас IRCConnection – визначає інтерфейс для класів, що реалізують з'єднання з сервером (IRCSocketConnection, IRCWebSocketConnection).
- Взаємодія з сервером через IRCConnection:
 - IRCClient взаємодіє з сервером через об'єкт IRCConnection (IRCSocketConnection, IRCWebSocketConnection).

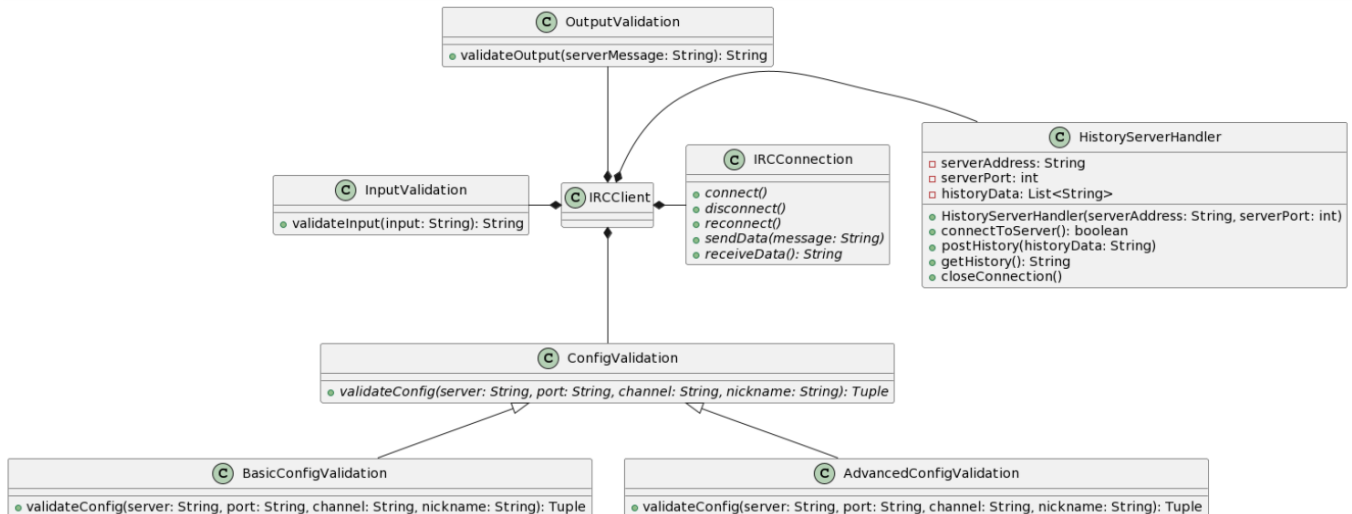


Рисунок 1.5.4 – Діаграма класів для OutputValidation, InputValidation HistoryServerHandler, ConfigValidation, IRCCConnection [Додаток Б]

Ця діаграма використовує різні типи відносин (композиція, успадкування) для відображення структури та взаємодій між класами в системі.

1.6. Вибір мови програмування та середовища розробки

Для розробки самого проєкту IRC Client я вибрав мову програмування Python. Також я використовав фреймворк tkinter для графічного інтерфейсу. Середовищем розробки було вибрано VSCode.

Мова програмування (Python):

1. Простота та читабельність коду: Python має простий та лаконічний синтаксис, що полегшує розробку та розуміння коду.
2. Багатофункціональність: Python володіє багатофункціональністю, що полегшує розробку складних програм та додатків.
3. Велика спільнота: Python має велику та активну спільноту розробників, що дозволяє отримати швидку підтримку та доступ до різноманітних бібліотек.
4. Переносимість: Код, написаний на Python, може бути легко перенесений між різними платформами, що спрощує роботу на різних операційних системах.

5. Розширюваність: Python має багато розширень та бібліотек для розвитку функціоналу програми.

Графічний інтерфейс (tkinter):

1. Вбудованість: Tkinter є вбудованим модулем Python, що полегшує встановлення та використання без додаткових завдань.
2. Простота використання: Tkinter надає простий та інтуїтивно зрозумілий інтерфейс для створення графічних елементів.
3. Портативність: Розроблені за допомогою Tkinter додатки можна легко використовувати на різних платформах.
4. Можливості кастомізації: Tkinter дозволяє легко налаштовувати та кастомізувати графічний інтерфейс за допомогою різноманітних віджетів.
5. Активний розвиток: Tkinter продовжує активно розвиватися та оновлюватися, забезпечуючи підтримку нових функцій та можливостей.

Середовище розробки (VSCode):

1. Легкість встановлення та налаштування: VSCode швидко встановлюється, має інтуїтивний інтерфейс та прості налаштування.
2. Велика спільнота та розширюваність: VSCode має велику кількість розширень та плагінів, що полегшує роботу та розширює функціонал середовища.
3. Підтримка Python: VSCode забезпечує потужну підтримку Python, включаючи автодоповнення, відладку та інші корисні функції.
4. Інтеграція з системами керування версіями: VSCode легко інтегрується з різними системами керування версіями, що полегшує спільну роботу в команді.
5. Широкі можливості відладки: VSCode надає потужні інструменти відладки, що спрощує пошук та виправлення помилок.

Загальні переваги та оптимізація розробки:

1. Швидкість розробки: Вибір Python, Tkinter та VSCode сприяє швидкому написанню коду та розвитку проєкту.

2. Ефективність ресурсів: Python є ефективною мовою програмування з низьким витратами ресурсів.
3. Легкість тестування: Програми, написані на Python, легше тестувати, що полегшує виявлення та виправлення помилок.
4. Сумісність з багатьма бібліотеками: Python і Tkinter добре взаємодіють із багатьма бібліотеками, що розширює функціонал додатку.
5. Крос-платформенність: Вибір Python, Tkinter та VSCode забезпечує можливість створення крос-платформених додатків для різних операційних систем

Спільна робота та розвиток:

1. Кодекс для збереження стилю: Python має PEP 8, що стандартизує стиль коду та полегшує спільну роботу в команді.
2. Можливість паралельної роботи: Використання Python дає можливість паралельної розробки різних частин додатку.
3. Зручна система контролю версій: VSCode забезпечує зручний інтерфейс для роботи з Git та іншими системами контролю версій.
4. Інтеграція з іншими інструментами: VSCode може легко інтегруватися з іншими інструментами розробки, що полегшує роботу з різними розширеннями.
5. Можливості командної роботи: VSCode забезпечує можливості командної роботи та обміну інформацією серед учасників проєкту.

Споживання ресурсів та оптимізація продуктивності:

1. Мале споживання пам'яті: Python має ефективне керування пам'яттю, що зменшує споживання ресурсів.
2. Багаторівнева архітектура Tkinter: Tkinter дозволяє побудовувати багаторівневий та структурований графічний інтерфейс.
3. Швидкість виконання: Python є достатньо швидкою мовою для багатьох типів додатків, зокрема, для IRC-клієнта.

4. Оптимізація шляхом використання бібліотек: Використання спеціалізованих бібліотек для мережевого взаємодії допомагає оптимізувати швидкість роботи IRC-клієнта.
5. Інструменти профілінгу в VSCode: Використання вбудованих інструментів профілінгу у VSCode дозволяє виявляти та усувати можливі швидкісні проблеми.

1.7. Проектування розгортання системи

Розгортання – це стадія розвитку, яка описує конфігурацію працюючої системи в реальному середовищі. Для розгортання необхідно приймати рішення щодо параметрів конфігурації, продуктивності, розподілу ресурсів, розподілу та паралелізму. Результати цієї фази фіксуються в конфігураційних файлах, а також у виді розгортання [5].

Діаграма розгортання – це схема, яка відображає конфігурацію вузлів обробки часу виконання та екземплярів компонентів та об'єктів, які працюють на них. Компоненти представляють час виконання виявлення кодових блоків. Компоненти, які не існують як час виконання (оскільки вони були скомпільовані), не з'являються на цих діаграмах; їх слід відображати на діаграмах компонентів. Діаграма розгортання показує екземпляри, тоді як діаграма компонентів показує самі визначення типів компонентів [5].

Діаграма розгортання для даного IRC клієнта (рис. 1.7.1)[Додаток В] представляє взаємодію між сервером IRC (Internet Relay Chat) і комп'ютером користувача, який використовує IRC-клієнт. IRC є протоколом для обміну повідомленнями в режимі реального часу через Інтернет.

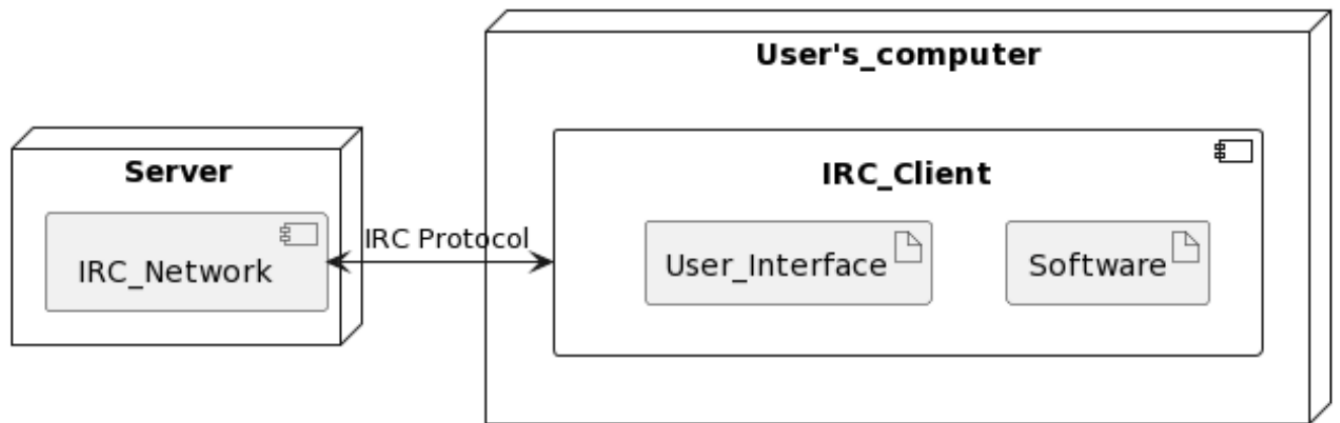


Рисунок 1.7.1 – Діаграма розгортання для IRC клієнта [Додаток В]

Основні елементи діаграми:

1. Вузол "Server" (Сервер):

- Містить компонент IRC_Network (IRC-мережа).
- IRC-мережа взаємодіє з IRC-клієнтами і забезпечує обмін повідомленнями за допомогою IRC-протоколу.

2. Вузол User's_computer (Комп'ютер користувача):

- Містить компонент IRC_Client (IRC-клієнт).
- IRC-клієнт має два артефакти: Software (програмне забезпечення) і User_Interface (інтерфейс користувача).

3. Зв'язок між IRC_Client та IRC_Network:

- Стрілка, яка позначає взаємодію між IRC-клієнтом та IRC-мережею за допомогою IRC-протоколу.
- Зліва від стрілки зазначено, що IRC_Client використовує IRC-протокол для комунікації з IRC_Network.
- Зправа від стрілки зазначено, що ця взаємодія є двосторонньою (взаємною).

Загалом, ця діаграма вказує на те, що IRC-клієнт на комп'ютері користувача спілкується з IRC-мережею за допомогою IRC-протоколу, що дозволяє обмінюватися повідомленнями в режимі реального часу.

Діаграма компонентів – це схема, яка відображає організації та залежності між типами компонентів.

Діаграма компонентів показує залежності між програмними компонентами, включаючи компоненти вихідного коду, бінарного коду та виконуваних компонентів. Модуль програмного забезпечення може бути представлений у вигляді компонента. Деякі компоненти існують на етапі компіляції, деякі – на етапі лінкування, інші - під час виконання; деякі можуть існувати одночасно на кількох етапах. Компонент, який має сенс лише на етапі компіляції, називається компонентом тільки для компіляції. У випадку використання тільки компіляційного компонента, виконавчий компонент буде, наприклад, виконуваним програмою.

Компонент представляє фізичну частину реалізації системи, включаючи програмний код (вихідний, бінарний або виконуваний) чи еквіваленти, такі як скрипти або файли команд. Деякі компоненти мають ідентичність і можуть мати власні фізичні об'єкти, до яких відносяться об'єкти часу виконання, документи, бази даних і так далі. Компоненти існують в області виконання – це фізичні одиниці на комп'ютерах, які можуть бути підключені до інших компонентів, замінені еквівалентними компонентами, переміщені, архівовані і так далі. Моделі можуть відображати залежності між компонентами, такі як залежності між компіляторами та часами виконання чи інформаційні залежності в організації людей. Екземпляр компонента може використовуватися для відображення реалізаційних одиниць, які існують під час виконання, включаючи їх розташування на екземплярах вузлів.

Діаграма компонентів відображає компонентні класифікатори, класи, визначені в них, та взаємозв'язки між ними. Компонентні класифікатори також можуть бути вкладені в інші компонентні класифікатори, щоб показати відносини визначення.

Клас, визначений у межах компонента, може бути відображений всередині нього, хоча для систем будь-якого розміру може бути зручніше надати список класів, визначених у межах компонента, замість відображення символів.

Діаграма, яка містить компонентні класифікатори та класифікатори вузлів, може використовуватися для відображення залежностей компілятора, які показуються пунктирними стрілками (залежності) від клієнтського компонента до постачальського компонента, від якого він залежить якимось чином. Типи залежностей є залежними від мови програмування і можуть бути показані як стереотипи залежностей.

Діаграму також можна використовувати для відображення інтерфейсів та залежностей виклику між компонентами, використовуючи пунктирні стрілки від компонентів до інтерфейсів на інших компонентах [6].

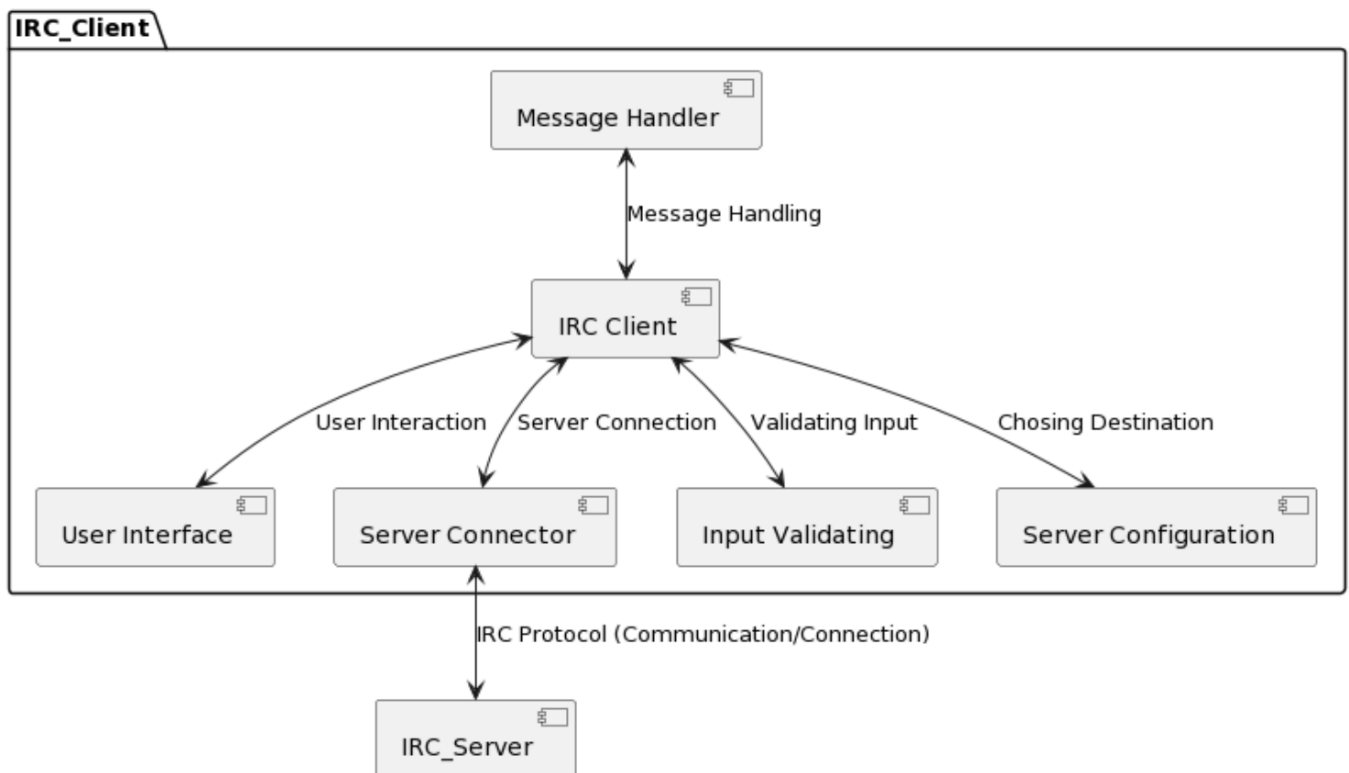


Рисунок 1.7.2 – Діаграма компонентів для IRC клієнта [Додаток Г]

Ця діаграма (рис. 1.7.2)[Додаток Г] відображає структуру та взаємодії між компонентами системи IRC клієнта. Нижче подано опис кожного компонента та їхніх взаємодій:

1. Пакет IRC_Client:

- IRC Client (Клієнт IRC): Представлений як компонент Client, він взаємодіє з іншими компонентами для реалізації клієнтської частини IRC.
- User Interface (Користувацький інтерфейс): Взаємодіє з клієнтом для обробки взаємодії користувача (User Interaction).
- Message Handler (Обробник повідомлень): Взаємодіє з клієнтом для обробки повідомлень (Message Handling).
- Server Connector (Підключення до сервера): Забезпечує з'єднання з сервером IRC (Server Connection).
- Input Validating (Перевірка введених даних): Взаємодіє з клієнтом для перевірки коректності введених даних (Validating Input).
- Server Configuration (Конфігурація сервера): Взаємодіє з клієнтом для вибору налаштувань сервера (Choosing Destination).

2. IRC_Server:

- IRC Server (Сервер IRC): Представлений як компонент Server, це відокремлений компонент, який здійснює серверну частину IRC.

3. Взаємодії:

- Відносини між клієнтом та іншими компонентами показані стрілками, які вказують на взаємодії, такі як User Interaction, Validating Input, Choosing Destination, Server Connection, і Message Handling.
- Server Connection показує взаємодію між клієнтом та компонентом "Connector", який в свою чергу взаємодіє з сервером (IRC Protocol (Communication/Connection)).

Ця діаграма надає візуальний зразок архітектури та взаємодій між різними компонентами системи IRC.

Діаграма послідовності (sequence diagram) відображає взаємодію у вигляді двовимірної діаграми. Вертикальний напрямок – це вісь часу; час просувається вниз по сторінці. Горизонтальний напрямок показує класифікаторні ролі, які представляють окремі об'єкти в колаборації. Кожна роль класифікатора представлена

вертикальним стовпчиком – лінією життя. Протягом часу існування об'єкта роль показується пунктирною лінією. Протягом часу активації процедури на об'єкті, лінія життя малюється подвійною лінією.

Повідомлення відображається як стрілка від лінії життя одного об'єкта до іншого. Стрілки впорядковані за часовим порядком вниз по діаграмі [7].

Діаграми послідовностей для опису процесів в нашому IRC клієнті зображені в [додатку Д].

Ця діаграма взаємодії (рис. 1.7.3) ілюструє процес взаємодії між користувачем, IRC-клієнтом і IRC-сервером під час використання команд `/help` та `/join #channel` у клієнтському інтерфейсі IRC.

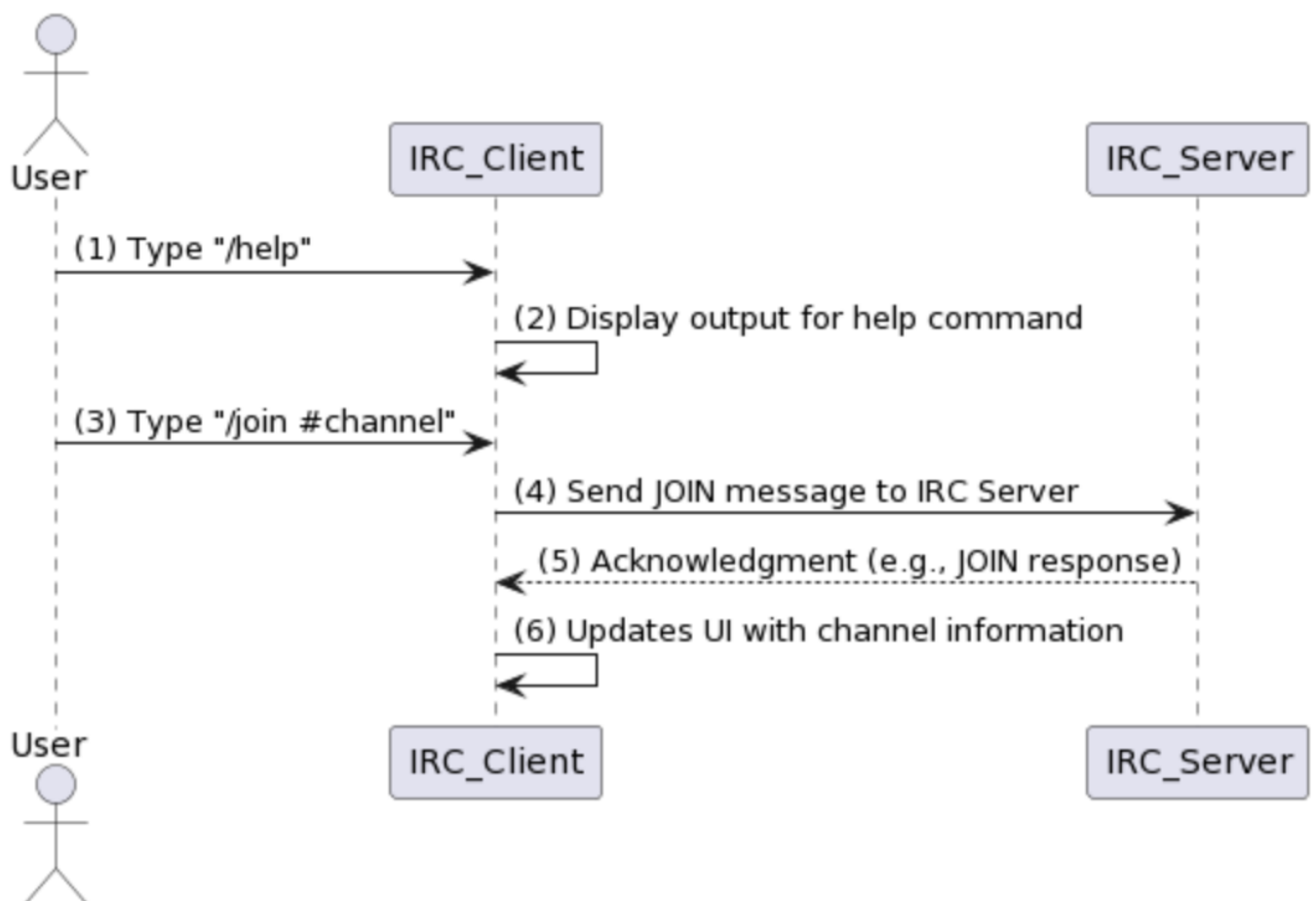


Рисунок 1.7.3 – Діаграма послідовностей для IRC клієнта процес взаємодії між користувачем під час використання команд `/help` та `/join #channel` [Додаток Д]

Основні етапи взаємодії:

1. Користувач (User):

- Взаємодіє з IRC-клієнтом.

2. IRC-клієнт (IRC_Client):

- Отримує від користувача команду /help (крок 1).
- Виконує внутрішню обробку команди /help та відображає відповідні відомості в інтерфейсі (крок 2).
- Отримує від користувача команду /join #channel (крок 3).
- Взаємодіє з IRC-сервером, надсилаючи повідомлення JOIN (крок 4).

3. IRC-сервер (IRC_Server):

- Отримує від IRC-клієнта повідомлення JOIN (крок 4).
- Виконує обробку та повертає підтвердження (наприклад, повідомлення JOIN) до IRC-клієнта (крок 5).

4. IRC-клієнт (IRC_Client):

- Отримує від IRC-сервера підтвердження (наприклад, JOIN response) (крок 5).
- Оновлює інтерфейс з інформацією про канал (крок 6).

Описи кожного кроку надані в коментарях під кожним стрілковим записом. Ця діаграма чітко показує послідовність подій та взаємодію між користувачем, клієнтом і сервером під час використання команд IRC.

Ось діаграма (рис. 1.7.4) взаємодії для підключення до IRC-серверу та зміни нікнейму:

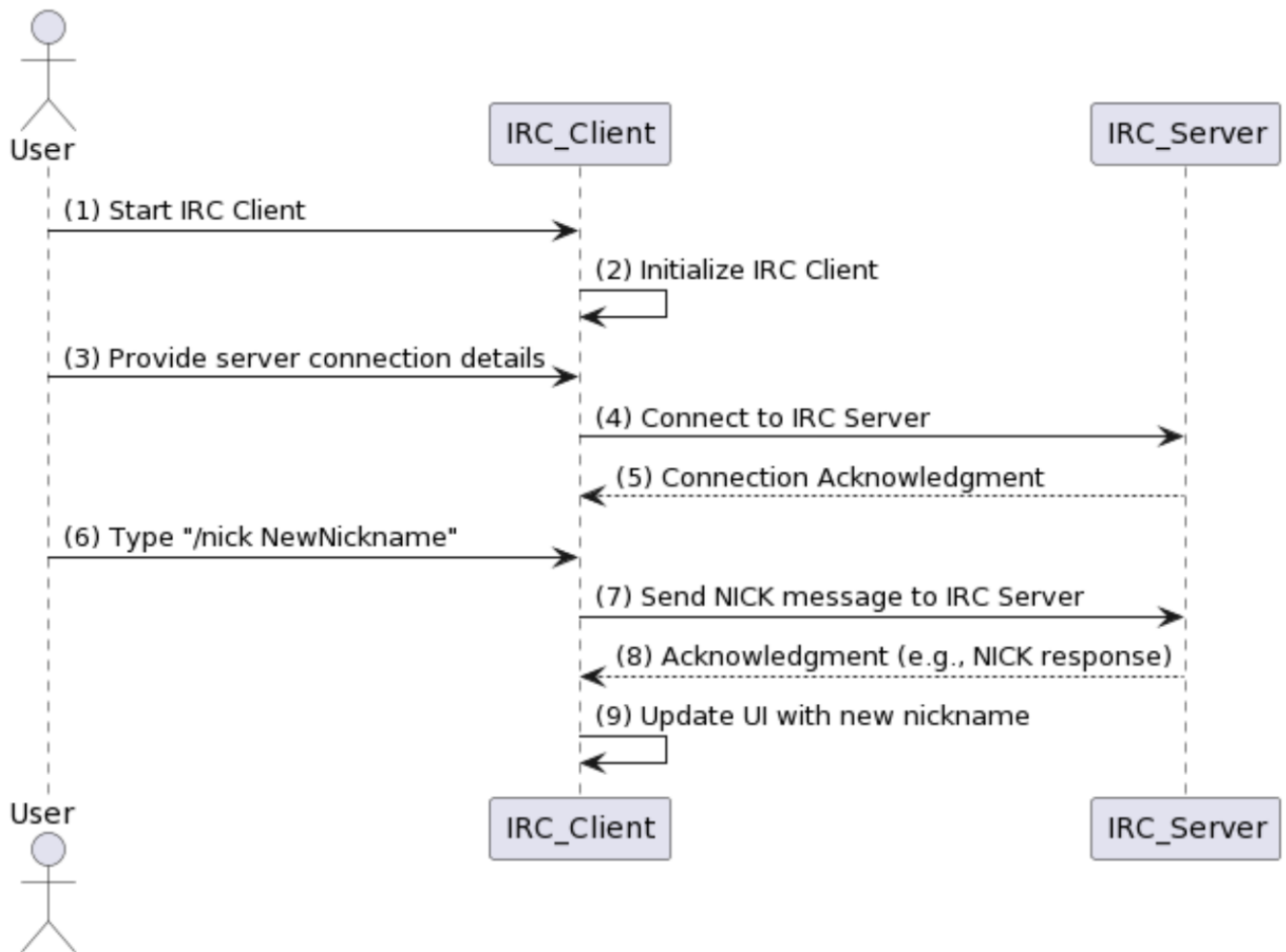


Рисунок 1.7.4 – Діаграма послідовностей для IRC клієнта процес підключення до IRC-серверу та зміни нікнейму [Додаток Д]

Опис кожного кроку:

1. Користувач (User):

- Запускає IRC-клієнт (крок 1).

2. IRC-клієнт (IRC_Client):

- Ініціалізується (крок 2).
- Отримує від користувача дані для підключення до сервера (крок 3).
- Підключається до IRC-сервера (крок 4).

3. IRC-сервер (IRC_Server):

- Підтверджує підключення IRC-клієнта (крок 5).

4. IRC-клієнт (IRC_Client):

- Користувач вводить команду для зміни нікнейму /nick NewNickname (крок 6).
- Відправляє повідомлення NICK на IRC-сервер (крок 7).

5. IRC-сервер (IRC_Server):

- Підтверджує зміну нікнейму (крок 8).

6. IRC-клієнт (IRC_Client):

- Оновлює інтерфейс з новим нікнеймом (крок 9).

Ця діаграма ілюструє послідовність дій взаємодії між користувачем, IRC-клієнтом та IRC-сервером під час підключення до сервера та зміни нікнейму.

2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1. Архітектура системи

2.1.1. Специфікація системи

Компоненти:

1. IRC-клієнт (GUI):

- Відповідає за взаємодію з користувачем.
- Використовує клас `IRCCClient`.
- Підключається до IRC-сервера, надсилає та отримує повідомлення.
- Обробляє введення користувача через графічний інтерфейс та делегує команди обробнику повідомлень (`MessageHandler`).

2. IRC-сервер:

- Відповідає за обмін даними з клієнтами.
- Використовує клас `Server`.
- Приймає підключення клієнтів і створює потоки для обробки кожного клієнта окремо.
- Зберігає історію чату, яка є спільною для всіх підключених клієнтів.

3. Обробник повідомлень (`MessageHandler`):

- Відповідає за виконання команд IRC на основі введення користувача.
- Використовує паттерн Команда із різними класами команд (наприклад, `JoinChannelCommand`, `ListUsersCommand`).
- Дозволяє виконання кількох команд, розділених ";".

4. Класи команд:

- Різні класи команд, які інкапсулюють поведінку команд IRC.
- Наприклад, `JoinChannelCommand`, `ListUsersCommand`, тощо.

5. Валідація конфігурації сервера:

- Класи `BasicConfigValidation` та `AdvancedConfigValidation` валідують конфігурації сервера.

6. З'єднання IRC (`IRCSocketConnection`):

- Управляє підключенням до IRC-сервера за допомогою сокетів.

- Реалізує методи для підключення, відключення, повторного підключення, надсилання та отримання даних.

7. Валідація виводу (OutputValidation):

- Валідує повідомлення, отримані від сервера, перед їх відображенням користувачеві.

8. Обробник історії сервера:

- Відповідає за обмін даними з окремим сервером історії для збереження та отримання історії чату.

Потік взаємодії:

Для клієнта IRC (Internet Relay Chat) реалізована архітектура CLIENT-SERVER (рис. 2.1.1 Діаграма класів), (рис. 2.1.1 Діаграма розгортання). Взаємодія клієнта та сервера встановлюється через такі компоненти:

1. Клас IRCClient:

- Представляє додаток клієнта.
- Обробляє загальні функції клієнта, включаючи підключення до сервера, відправку та отримання повідомлень і відключення.
- Використовує екземпляр класу IRCConnection для взаємодії з сервером IRC.

2. Клас IRCConnection (абстрактний):

- Служить абстрактним базовим класом, який визначає інтерфейс для мостів зв'язку.
- Два конкретні виконання (IRCSocketConnection та IRCWebSocketConnection) розширюють цей клас, надаючи конкретні методи взаємодії.

3. Клас IRCSocketConnection:

- Конкретна реалізація класу IRCConnection з використанням TCP-сокету для зв'язку.
- Обробляє методи, такі як connect, disconnect, reconnect, send_data та

receive_data, що специфічні для зв'язку через сокет.

4. Клас IRCWebSocketConnection:

- Заготовка для потенційної альтернативної реалізації за допомогою WebSocket для зв'язку.
- Зараз цей клас порожній і не реалізований.

5. Валідація конфігурації сервера:

- Два класи (BasicConfigValidation та AdvancedConfigValidation) реалізують логіку перевірки конфігурації сервера.
- Клас IRCClient використовує екземпляр одного з цих класів для перевірки деталей сервера перед встановленням з'єднання.

6. Клас MessageHandler:

- Обробляє виконання команд IRC на основі введення користувача.
- Відображає команди користувача на конкретні класи команд (наприклад, JoinChannelCommand, LeaveChannelCommand) та виконує їх.

7. Класи команд:

- Класи, що представляють конкретні команди IRC (наприклад, join, part, names), які реалізують інтерфейс Command.
- У кожному класі команди є метод execute, який виконує конкретну дію, пов'язану з командою.

8. Клас InputValidation:

- Надає методи для перевірки введення користувача перед відправленням його на сервер.
- Модифікує введення користувача на основі конкретних команд (наприклад, /join, /part, /nick).

9. Клас OutputValidation:

- Перевіряє повідомлення, отримані від сервера, перед їхнім відображенням користувачу.
- Додає мітку часу до отриманих даних для відображення.

10. Клас GUI:

- Представляє графічний інтерфейс користувача за допомогою Tkinter.
- Надає рамки для реєстрації (введення деталей сервера) та чату (відображення повідомлень, відправлення повідомлень).
- Взаємодіє з екземпляром IRCClient для обробки дій користувача

11. Основне виконання:

- В блоку `__main__` створюється екземпляр IRCClient, і встановлюється перевірка конфігурації сервера.
- Створюється екземпляр класу GUI з екземпляром IRCClient.
- Клас GUI запускає головний цикл Tkinter.

Клієнт взаємодіє з сервером, підключаючись до нього за допомогою методу `connect_to_server`, надсилаючи повідомлення за допомогою `send_message`, отримуючи повідомлення за допомогою `receive_data` та відключаючись за допомогою `disconnect_from_server`. Клієнт взаємодіє з сервером через з'єднання за допомогою сокету, а користувач взаємодіє з клієнтом через графічний інтерфейс користувача (GUI).

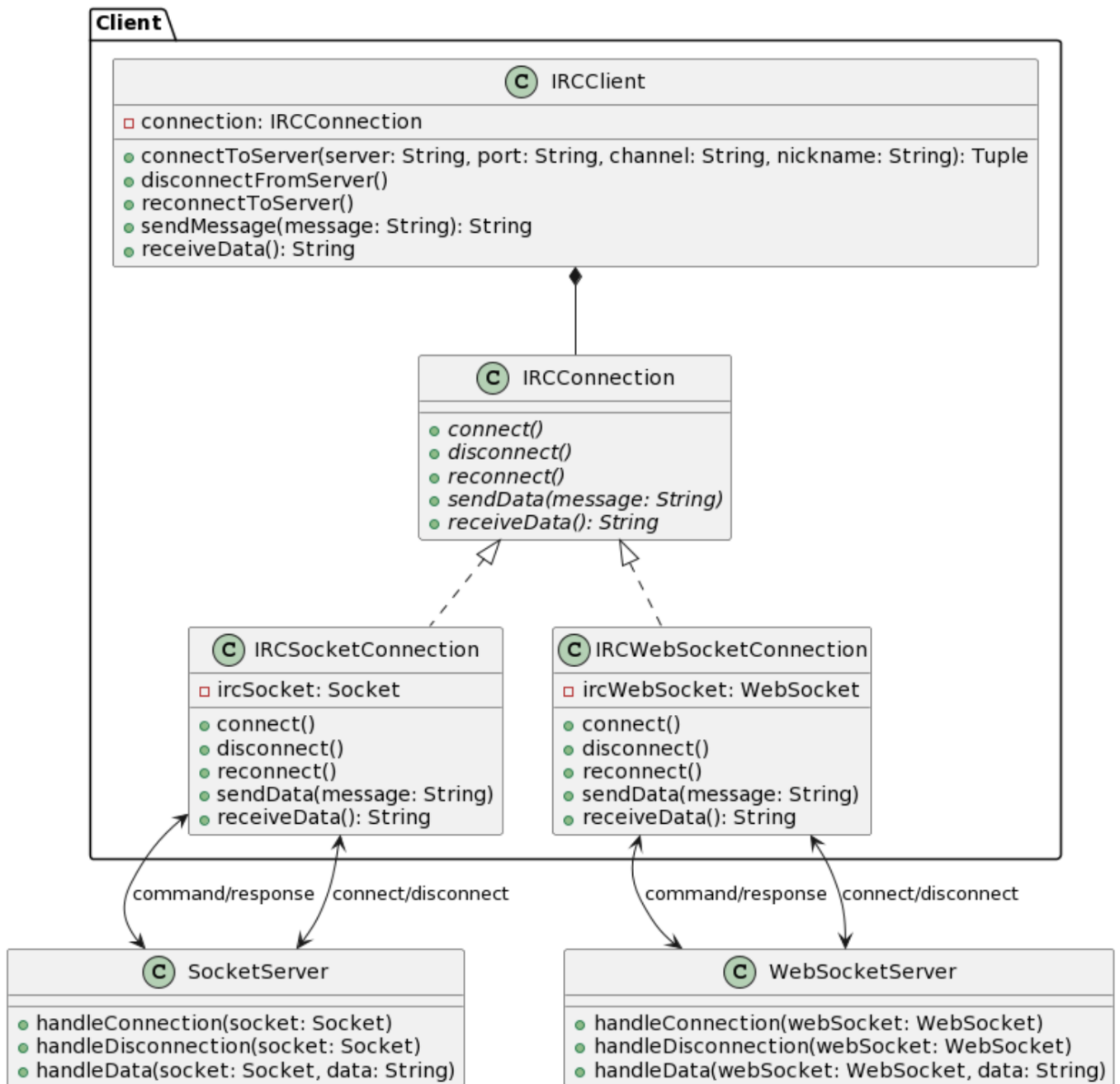


Рисунок 2.1.1 – Діаграма класів для архітектури CLIENT-SERVER [Додаток Е]

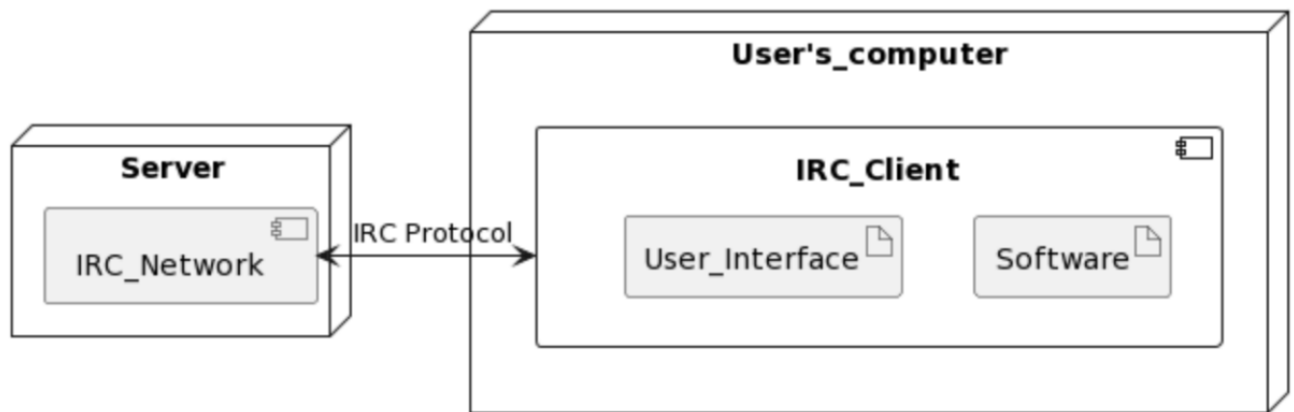


Рисунок 2.1.2 – Діаграма розгортання CLIENT-SERVER [Додаток В]

Застосування CLIENT-SERVER архітектури для зберігання історії чату можна розглянути на діаграмі класів (рис. 2.1.3).

Проект реалізує просту архітектуру клієнт-сервер для зберігання та отримання історії чату. Ось огляд того, як реалізована архітектура:

1. Серверна частина:

- Клас `Server` відповідає за обробку вхідних підключень клієнтів та управління комунікацією з клієнтами.
- Після прийняття підключення від клієнта створюється новий потік (`client_handler_thread`), який обробляє комунікацію з цим клієнтом незалежно.
- Сервер безперервно очікує вхідних підключень у головному потоці.

2. Сервер історії:

- Клас `HistoryServerHandler` відповідає за комунікацію з окремим сервером історії.
- Створюється сокет сервера, і сервер безперервно очікує вхідних підключень.
- Метод `handle_client` класу `Server` керує комунікацією з підключеним

клієнтом. Він отримує повідомлення, оновлює історію чату (`chat_history`) та надсилає оновлену історію клієнту.

3. Клієнтська сторона:

- Клас `IRCCClient` відповідає за управління клієнтською функціональністю.
- Використовується екземпляр `HistoryServerHandler` для підключення до сервера історії, і клієнт безперервно надсилає та отримує історію чату від/до сервера.
- Клієнт надсилає повідомлення на сервер за допомогою методу `send_message`. Перед відправленням повідомлення воно перевіряється за допомогою `InputValidation`.
- Клієнт отримує та обробляє дані від сервера за допомогою методу `receive_data`. Отримані дані перевіряються за допомогою `OutputValidation`.
- Клієнт також надсилає історію чату на сервер історії за допомогою методу `post_chat_history`.
- Клієнт може запросити історію чату від сервера історії за допомогою методу `get_chat_history`.

4. Потокове виконання:

- Використовуються потоки як на стороні сервера, так і на стороні клієнта для обробки одночасних підключень та безперервного отримання та обробки даних.

Загалом ця архітектура дозволяє зберігати та отримувати історію чату на сервері, а окремий сервер історії відповідає за обробку цих даних. Клієнт та сервер спілкуються за допомогою сокетів та обмінюються повідомленнями, що забезпечує просту, але ефективну модель клієнт-сервер для управління історією чату.

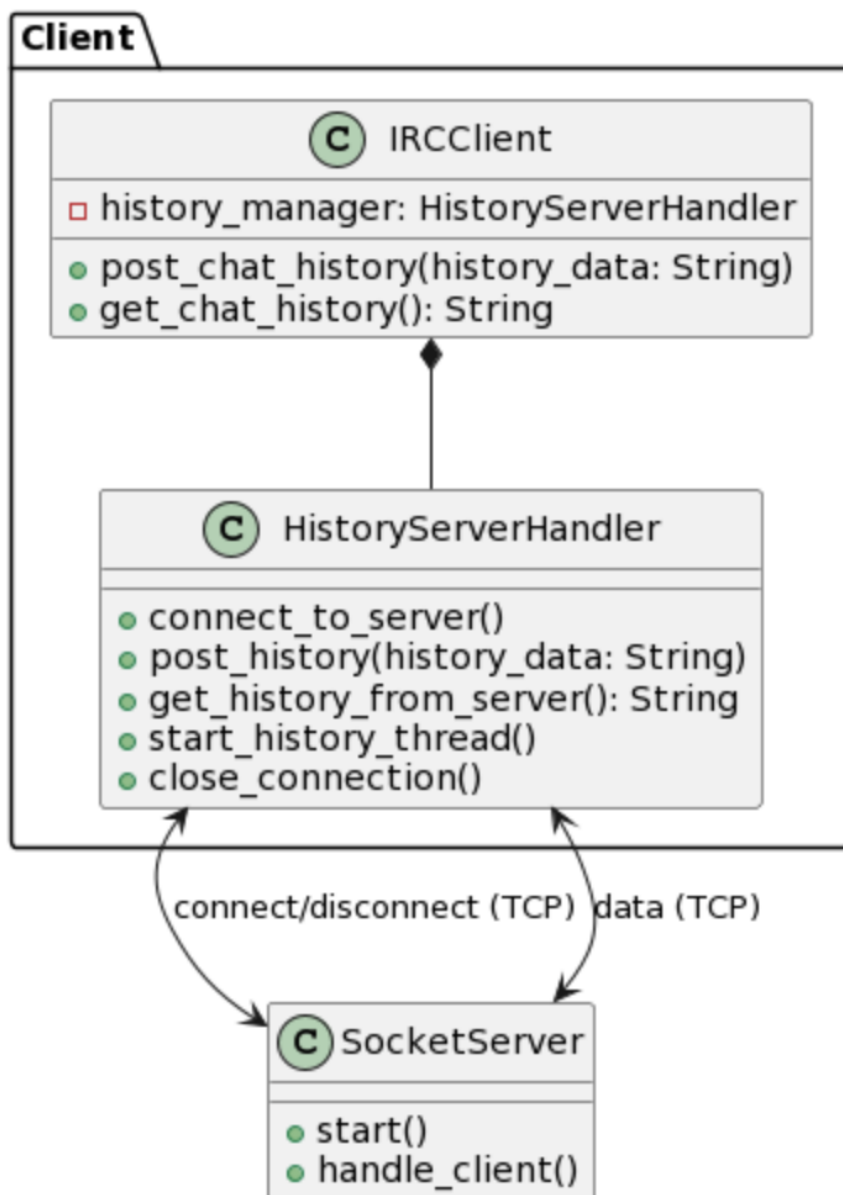
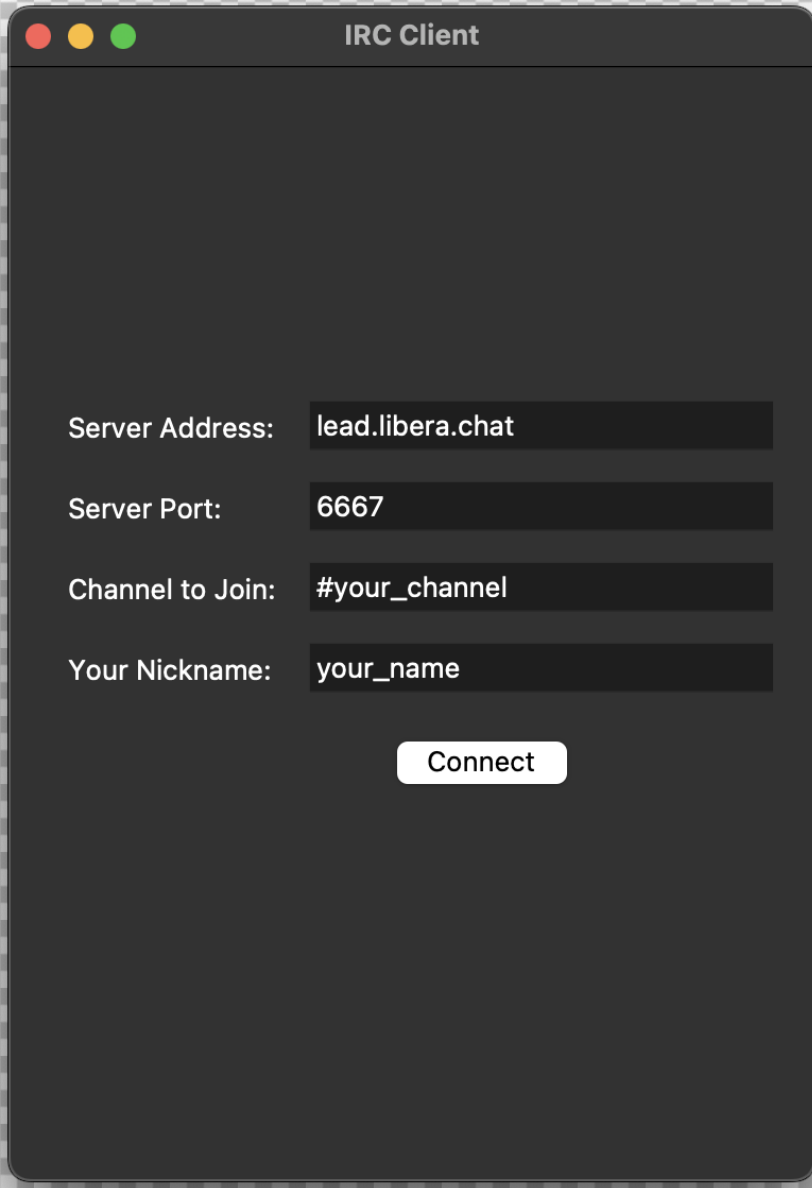


Рисунок 2.1.3 – Діаграма класів для архітектури CLIENT-SERVER для відображення історії чату [Додаток Е]

Графічний інтерфейс (GUI) (рис. 2.1.4), (рис. 2.1.5), (рис. 2.1.6) є ключовою частиною системи, яка надає зручний спосіб взаємодії з користувачем. У цій системі GUI відповідає за відображення вікна реєстрації та чату, обробку введення користувача та відправку команд на IRC-сервер.



The image shows a dark-themed window titled "IRC Client". It contains four input fields for registration, each with a label to its left. The fields are: "Server Address" with the value "lead.libera.chat", "Server Port" with the value "6667", "Channel to Join" with the value "#your_channel", and "Your Nickname" with the value "your_name". Below these fields is a white button with the text "Connect".

Server Address:	lead.libera.chat
Server Port:	6667
Channel to Join:	#your_channel
Your Nickname:	your_name

Connect

Рисунок 2.1.4 – Зображення Екрану Реєстрації у GUI [Додаток Є]

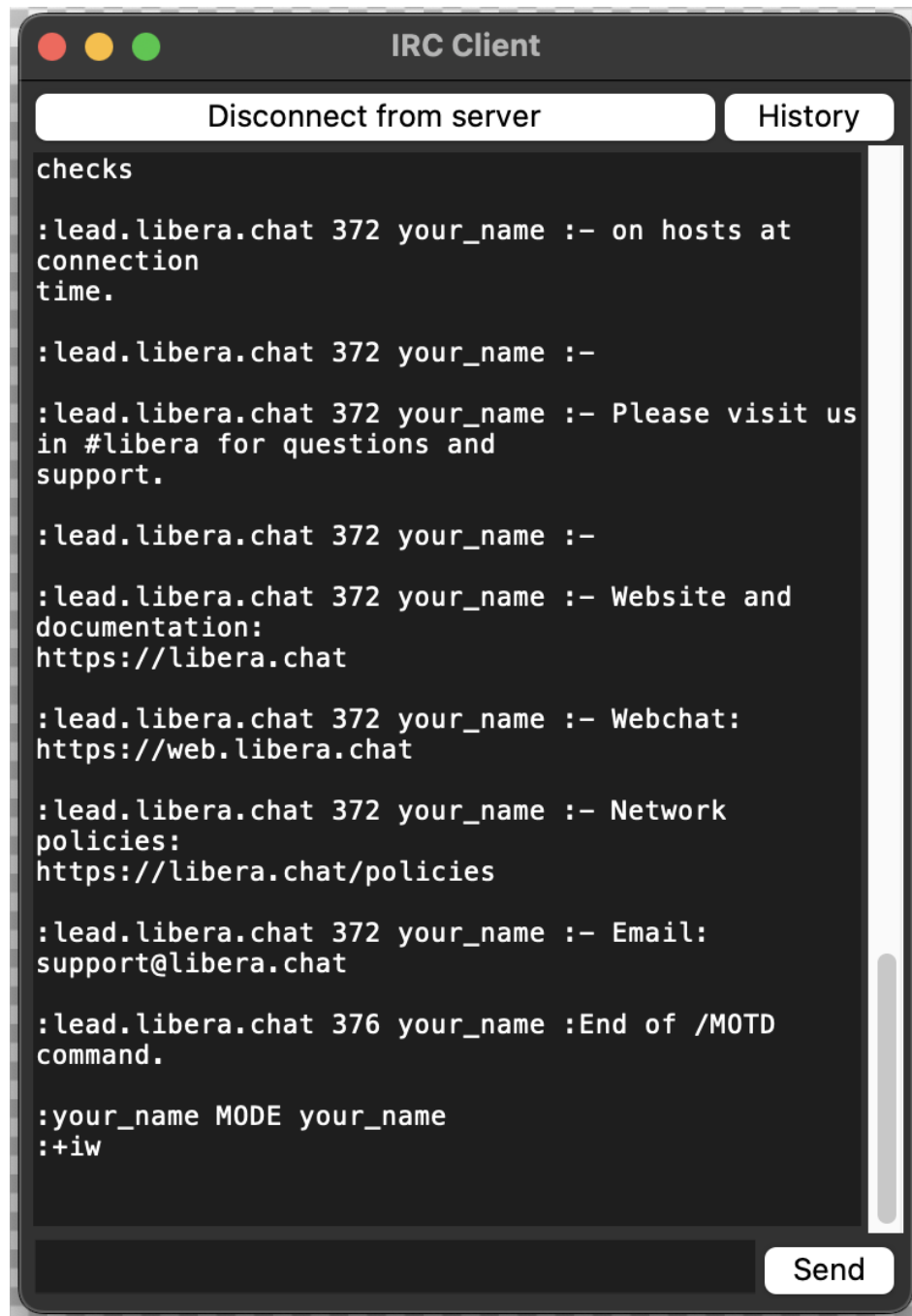


Рисунок 2.1.5 – Зображення Екрану Чату у GUI [Додаток Є]

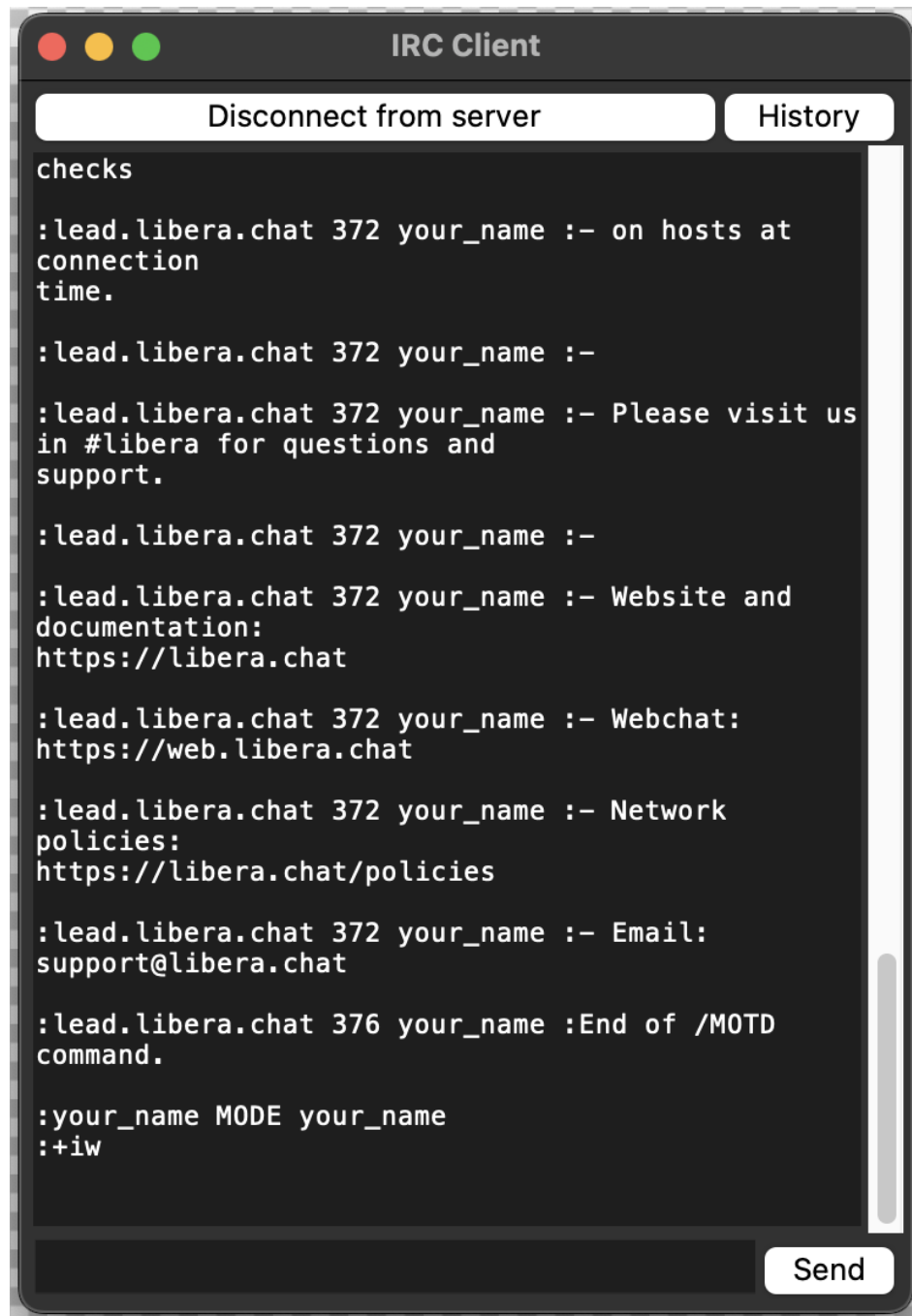


Рисунок 2.1.6 – Зображення Екрану Історії у GUI [Додаток Є]

Давайте розглянемо основні компоненти та функціональність GUI:

Основні елементи GUI:

1. Вікно Реєстрації:

- Відповідає за введення основних налаштувань, таких як адреса сервера, порт, канал та нікнейм користувача.

- Включає поля для введення та кнопку для початку процесу підключення.

2. Вікно Чату:

- Відображає повідомлення та історію чату.
- Має поле для введення нових повідомлень.

3. Кнопки та Елементи Керування:

- Можливість виклику команд, таких як /join, /part, /names, /info, /nick, /help, /quit, тощо.
- Кнопки або інші елементи для здійснення зв'язку з сервером, наприклад, Connect, Disconnect, Reconnect.

4. Інші Функції:

- Можливість відобразити історію чату.
- Обробка введення користувача та надсилання повідомлень на сервер.
- Інші опції конфігурації та відображення стану підключення.

Взаємодія GUI з іншими компонентами:

1. Взаємодія з IRC-клієнтом (IRCClient):

- Графічний інтерфейс викликає методи IRCClient для підключення, відключення, перепідключення, відправки та отримання повідомлень.
- Отримує повідомлення від IRC-клієнта та відображає їх у вікні чату.

2. Взаємодія з Обробником Повідомлень (MessageHandler):

- Обробка команд користувача, які вводяться через графічний інтерфейс.
- Делегування введених команд на IRC-сервер для виконання.

3. Взаємодія з Сервером Історії Чату:

- Вивантаження та відображення історії чату в окремому вікні, яке може бути відкрите за відповідною командою.

Графічний інтерфейс в цьому проекті служить як інтуїтивний та ефективний засіб взаємодії між користувачем та IRC-сервером, забезпечуючи зручний інтерфейс для введення команд і спостереження за чатом.

Код проекту побудований на основі об'єктно-орієнтованого програмування (ООП). Давайте розглянемо деякі ключові концепції ООП, які використовуються в цьому проекті:

1. Класи та Об'єкти:

- `IRCCClient`: Клас, який представляє IRC-клієнта. Містить методи для з'єднання з сервером, відправлення та отримання повідомлень.
- `IRCSocketConnection`: Підклас `IRCConnection`, що визначає конкретну реалізацію IRC-з'єднання через сокети.
- `IRCWebSocketConnection`: Інший підклас `IRCConnection`, який представляє альтернативну реалізацію IRC-з'єднання через `WebSocket` (порожній у вашому коді).

2. Абстракція:

- `IRCConnection` (Абстрактний клас): Визначає загальний інтерфейс для об'єктів, які представляють IRC-з'єднання. Має абстрактні методи, які повинні бути реалізовані підкласами.

3. Інкапсуляція:

- Пакети та Методи: Код розділений на пакети, що дозволяє групувати пов'язані класи та функціональність. Методи класів використовуються для реалізації конкретної функціональності.

4. Наслідування:

- `IRCSocketConnection` та `IRCWebSocketConnection`: Наслідування від `IRCConnection` для визначення різних видів IRC-з'єднань.

5. Поліморфізм:

- Метод `validate_config` в `ConfigValidation` та його підкласах: Різні підкласи можуть надавати різні реалізації методу для валідації конфігурації сервера.

6. Композиція та Декомпозиція:

- IRCClient використовує IRCSocketConnection, MessageHandler, та інші класи: Компоненти системи об'єднуються в єдиний об'єкт IRCClient, що дозволяє декомпонувати функціональність на окремі класи.

7. Многозадачність (Threading):

- Використання threading для обробки одночасних завдань: Зокрема, використання окремого потоку для прийому даних з сервера у класі IRCSocketConnection.

8. Винятки (Exception Handling):

- Використання блоків try, except для обробки винятків: Наприклад, в коді обробляються винятки, пов'язані з сокетом.

Цей проект створений з дотриманням ключових принципів ООП, таких як абстракція, інкапсуляція, наслідування та поліморфізм, що дозволяє розширювати та підтримувати код у більш систематизований та модульний спосіб.

2.1.2. Вибір та обґрунтування патернів реалізації

Шаблон Команда – це паттерн проектування, який перетворює запит на самостійний об'єкт, який містить всю інформацію про запит. Це перетворення дозволяє параметризувати методи різними запитами, відкладати або ставити в чергу виконання запиту та підтримувати відмінні операції [8].

В розробці програмного забезпечення, паттерн Команда є паттерном проектування поведінки, який перетворює запит в самостійний об'єкт. Цей об'єкт містить всю інформацію про запит, дозволяючи параметризацію клієнтів різними запитами, утримання запитів в черзі та реєстрацію операцій. Ключові компоненти патерну: команда (інтерфейс і абстрактний клас), конкретна команда, інвокер, отримувач, клієнт. Тепер давайте розглянемо ці компоненти у моєму коді (рис. 2.1.7):

- Команда (Абстрактний Клас): Клас Command є абстрактним базовим класом для класів команд IRC. Він декларує метод execute.

- Конкретна Команда:
 - Класи `JoinChannelCommand`, `LeaveChannelCommand`, `ListUsersCommand`, `ChannelInfoCommand`, `ChangeNicknameCommand`, `DisplayHelpCommand`, та `QuitCommand` є конкретними класами команд. Кожен реалізує метод `execute`, визначаючи конкретні дії, пов'язані з командами IRC.
- Інвокер:
 - Клас `MessageHandler` виступає в ролі інвокера. Він вибирає та виконує команди IRC на основі введення користувача.
- Отримувач:
 - Клас `ServerHandler` виступає в ролі отримувача. Він виконує дії, пов'язані з командами IRC, такі як підключення, відключення та відправлення даних на сервер.
- Клієнт:
 - Клас `IRCCClient` виступає в ролі клієнта. Він створює та конфігурує об'єкти команд (`Command`), враховуючи введення користувача.

Паттерн Команда в моєму коді дозволяє інкапсулювати різні команди IRC у вигляді об'єктів, що полегшує розширення та додавання нових команд без змінення існуючого коду. Він також сприяє розділенню відправника (Інвокера) та отримувача, сприяючи гнучкості та зручності у підтримці кодової бази.

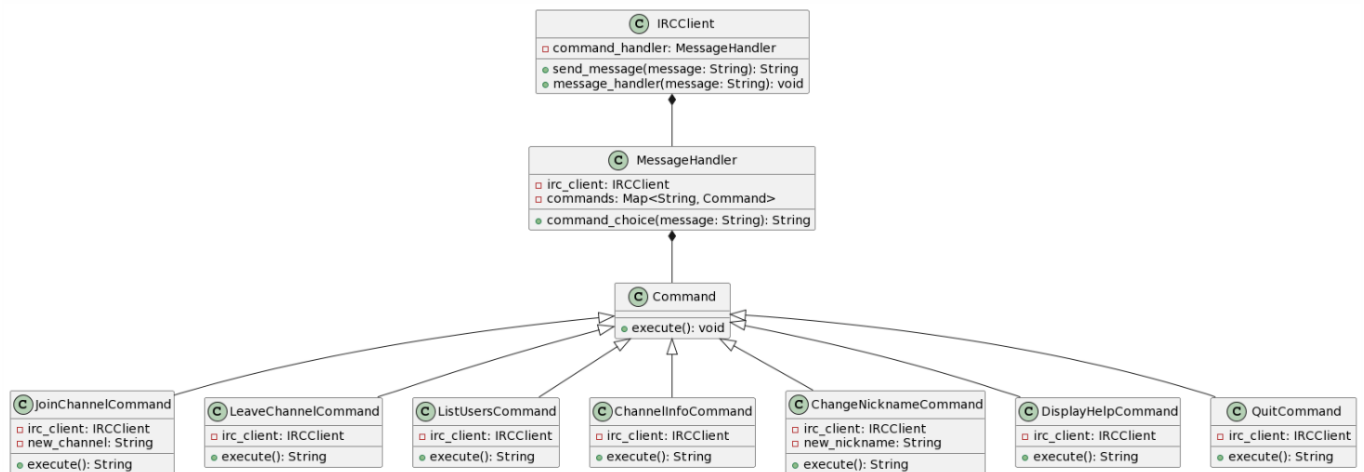


Рисунок 2.1.7 – Діаграма класів для шаблону «COMMAND» [Додаток Ж]

Factory Method (Фабричний метод) – це паттерн проектування, що належить до категорії створення об'єктів. Він надає інтерфейс для створення об'єктів у суперкласі, але дозволяє підкласам змінювати тип об'єктів, які будуть створені [9].

Шаблон "фабричний метод" визначає інтерфейс для створення об'єктів певного базового типу. Це зручно, коли хочеться додати можливість створення об'єктів не базового типу, а деякого дочірнього. Фабричний метод у такому разі є зачіпкою для впровадження власного конструктора об'єктів. Основна ідея полягає саме в заміні об'єктів їх підтипами, що при цьому зберігає ту ж функціональність; інша частина поведінки об'єктів не є інтерфейсною (AnOperation) і дозволяє взаємодіяти із створеними об'єктами як з об'єктами базового типу. Тому шаблон "фабричний метод" носить ще назву "Віртуальний конструктор".

У моєму коді (рис. 2.1.8):

1. Клас ConfigValidation служить абстрактним базовим класом (ABC) для перевірки конфігурації сервера. Він оголошує абстрактний метод validate_config, який повинен бути реалізований конкретними підкласами.
2. BasicConfigValidation та AdvancedConfigValidation - це конкретні реалізації абстрактного класу ConfigValidation. Вони надають різну логіку перевірки для конфігурацій сервера.

3. Клас `IRCCClient` використовує шаблон фабричний метод для створення відповідного об'єкта `ConfigValidation`. Метод `set_server_config_validator` - це саме фабричний метод тут. Він дозволяє клієнту встановлювати тип об'єкта `ConfigValidation`, який він хоче використовувати для перевірки конфігурацій сервера.
4. В блоку `__main__` створюється екземпляр `IRCCClient`, і його метод `set_server_config_validator` викликається з екземпляром `BasicConfigValidation`. Це встановлює базову логіку перевірки конфігурації для IRC-клієнта.

Використовуючи шаблон Фабричний Метод, IRC-клієнт може легко перемикатися між різними типами логіки перевірки конфігурації, змінюючи тип об'єкта `ConfigValidation`. Це сприяє гнучкості і дозволяє легко розширювати код з новими стратегіями перевірки у майбутньому.

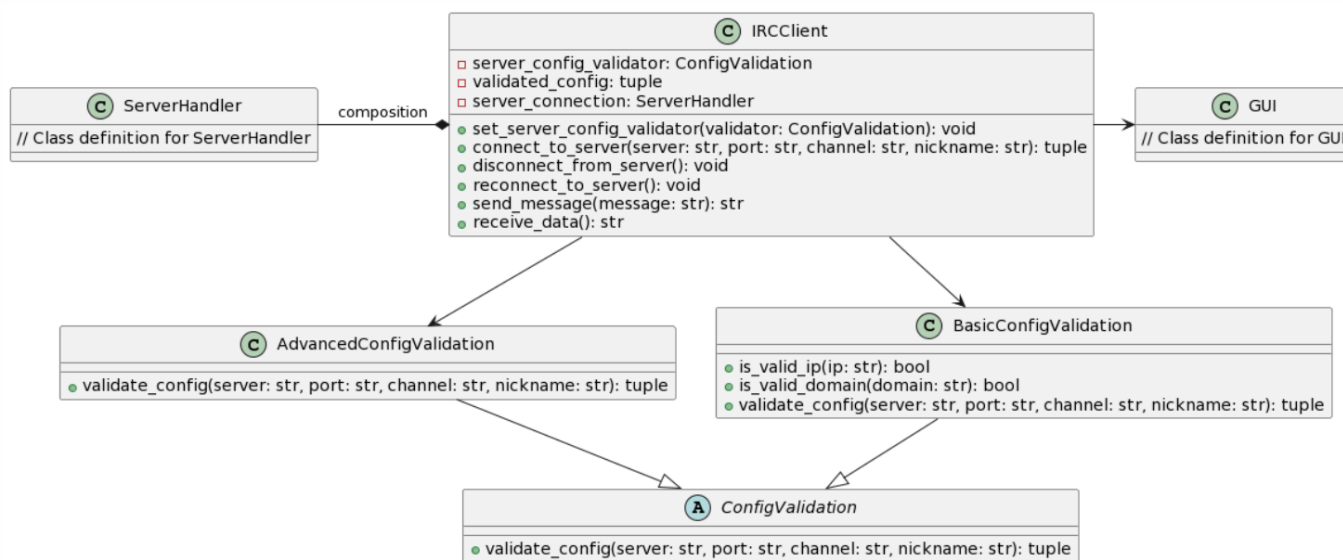


Рисунок 2.1.8 – Діаграма класів для шаблону «Factory Method» [Додаток Ж]

Bridge (Міст) - це паттерн проектування структури, який дозволяє розділити великий клас або набір тісно пов'язаних класів на дві окремі ієрархії – абстракцію та

реалізацію, які можуть розвиватися незалежно одна від одної [10]. Він особливо корисний, коли ви хочете уникнути постійного зв'язку між абстракцією та її реалізацією, дозволяючи їм еволюціонувати незалежно.

У даному проекті шаблон Міст реалізований з абстрактним класом `IRConnection` та його конкретними реалізаціями (`IRCSocketConnection` та `IRCWebSocketConnection`). Розглянемо складові цього коду:

1. Абстракція (`IRConnection`):

- Клас `IRConnection` виступає як абстракція в шаблоні Міст. Він оголошує інтерфейс, яким повинні користуватися конкретні реалізації. Визначає методи, такі як `connect`, `disconnect`, `reconnect`, `send_data` та `receive_data`.

2. Вдосконалені абстракції (`IRCSocketConnection` та `IRCWebSocketConnection`):

- Конкретні класи `IRCSocketConnection` та `IRCWebSocketConnection` розширюють клас `IRConnection`, надаючи конкретні реалізації методів, оголошених в абстракції.
- У цьому випадку `IRCSocketConnection` представляє з'єднання IRC на основі сокетів, тоді як `IRCWebSocketConnection` призначений для представлення з'єднання IRC на основі `IRCSocketConnection`.

3. Клієнт (`IRCCClient`):

- Клас `IRCCClient` представляє клієнта, який використовує міст IRC для підключення, відключення та спілкування з сервером IRC.
- Він має посилання на інтерфейс `IRConnection` (або `IRCSocketConnection`, або `IRCWebSocketConnection`) і використовує його для комунікації.

4. Композиція (IRCCClient використовуючи IRCConnection):

- Клас IRCCClient використовує композицію для утримання екземпляру інтерфейсу IRCConnection. Це дозволяє клієнту переходити між різними реалізаціями моста без зміни власного коду.
- Клієнт взаємодіє з абстракцією (IRCConnection), а не безпосередньо з її реалізаціями (IRCSocketConnection або IRCWebSocketConnection).

Використовуючи шаблон Міст, код досягає гнучкості, роз'єднуючи високорівневого клієнта IRC від конкретних деталей реалізації зв'язку IRC. Якщо у майбутньому вам потрібно додати новий тип з'єднання IRC (наприклад, інший протокол), ви можете створити нову конкретну реалізацію без зміни існуючого коду клієнта.

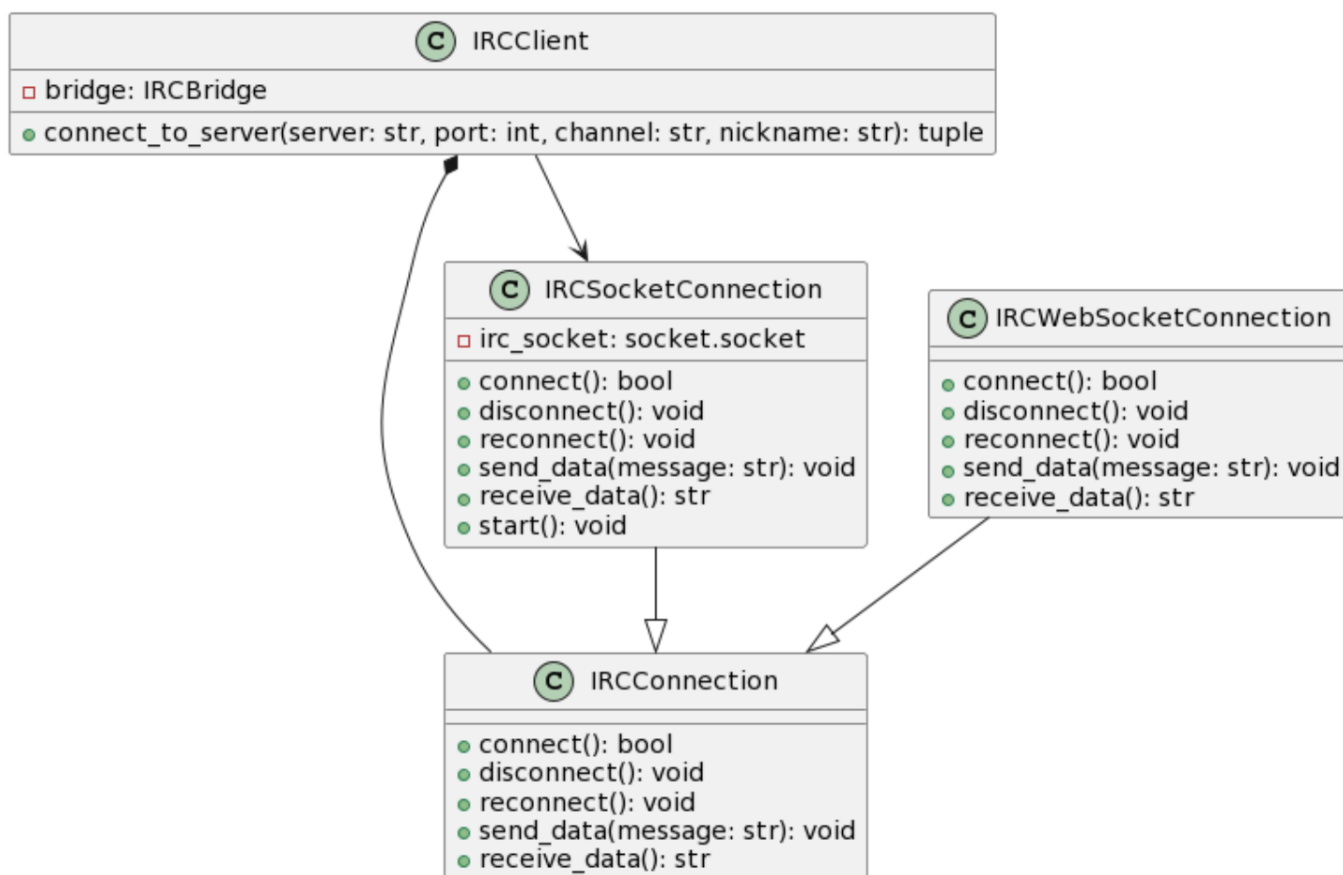


Рисунок 2.1.9 – Діаграма класів для шаблону «BRIDGE» [Додаток Ж]

Composite (Компоновщик) – це паттерн проектування структури, який дозволяє складати об'єкти в деревоподібні структури, а потім працювати з цими структурами так, ніби вони були окремими об'єктами [11].

Паттерн Композит використовується у даному проєкті для створення структури, де окремі команди та комбіновані команди (що складаються з декількох команд) обробляються однаково. Цей паттерн дозволяє будувати складні структури, використовуючи прості окремі об'єкти та об'єднувати їх у великі структури.

Ось як застосовується паттерн Композит у наданому коді:

1. Абстрактний базовий клас (Command):

- Command – це абстрактний базовий клас (ABC), який визначає загальний інтерфейс для всіх конкретних класів команд.
- Він оголошує абстрактний метод execute(), який повинен бути реалізований конкретними підкласами.

2. Листові вузли (JoinChannelCommand, LeaveChannelCommand, і т.д.):

- Кожна конкретна команда, така як JoinChannelCommand, LeaveChannelCommand і т.д., розширює базовий клас Command та реалізує метод execute.
- Ці класи представляють листові вузли композитної структури.

3. Композитний вузол (CompositeCommand):

- CompositeCommand – це конкретний клас, який розширює базовий клас Command.
 - У нього є внутрішній список (self.commands), щоб зберігати колекцію об'єктів Command, які можуть бути листовими вузлами або іншими композитними вузлами.
- Він реалізує метод execute, який ітерується по своєму списку команд та виконує

кожну, повертаючи конкатенований результат.

4. Використання у MessageHandler:

- MessageHandler використовує паттерн Композит для обробки як окремих команд, так і комбінованих команд.
- Клас CompositeCommand використовується для агрегації та виконання декількох команд, коли користувач надає список команд, розділених крапкою з комою.
- Метод `command_choice` перевіряє, чи містить введене повідомлення декілька команд (визначається наявністю ';'). У разі позитивної відповіді він створює об'єкт `CompositeCommand` та додає до нього окремі команди.

Цей паттерн дозволяє вам обробляти окремі команди та комбіновані команди однаково. Незалежно від того, чи є команда окремою операцією чи складається з кількох операцій, клієнтський код (у цьому випадку, `MessageHandler`) може виконувати їх у єдинообразний спосіб. Це сприяє гнучкості та масштабованості при обробці складних структур команд.

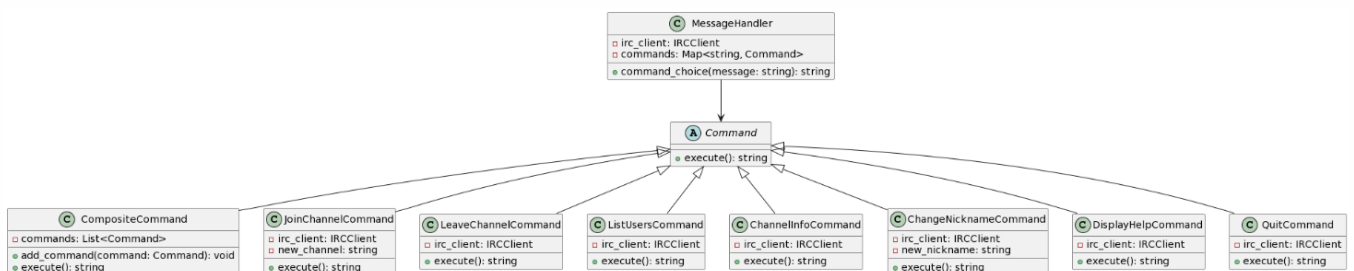


Рисунок 2.1.10 – Діаграма класів для шаблону «COMPOSITE» [Додаток Ж]

2.2. Інструкція користувача

Інструкція користувача для IRC-клієнта виглядає наступним чином:

1. Запуск клієнта:

- Запустіть скрипт `main_script.py`.
- GUI клієнта повинно відкритися.
- Сервер для збереження історії чату почне слухати підключення.

2. Взаємодія сервера для історії з клієнтами:

- Після запуску сервера, він автоматично очікуватиме підключень від клієнтів.
- Кожен клієнт може вводити повідомлення, які будуть відображені в історії чату.
- При відключенні клієнта, сервер виведе відповідне повідомлення в консоль.

3. Підключення до сервера:

- Введіть необхідні дані (адресу сервера, порт, канал та ім'я користувача) в текстові поля.
- Клікніть кнопку `Connect` для підключення до IRC-сервера.

4. Взаємодія з IRC-сервером:

- Після підключення, ви можете вводити IRC-команди в текстове поле та використовувати кнопку `Send` для їх відправлення.
- Для взаємодії з сервером введіть текст повідомлення або команди у форматі, наприклад: `/join #channel`, `/nick new_nickname`, тощо.

5. Чат та Історія повідомлень:

- Нижній блок містить чат, де відображаються отримані повідомлення.
- Для виведення історії повідомлень, використайте кнопку `History`.

6. Відключення від сервера:

- Клікніть кнопку `Disconnect`, щоб відключитися від IRC-сервера.

7. Вихід:

- Закрийте GUI клієнта для завершення роботи.
- Закрийте консоль, щоб завершити роботу сервера.

ВИСНОВКИ

У ході курсової роботи було реалізовано клієнт-серверну архітектуру для IRC (Internet Relay Chat), що дозволяє здійснювати взаємодію між користувачами через централізований сервер. Клієнтська частина була спроектована як GUI-застосунок, який взаємодіє з сервером за допомогою IRC-протоколу. Серверна частина реалізована як простий IRC-сервер, який забезпечує обробку повідомлень, обмін даними та зберігання історії чату. Архітектура використовує класичну модель клієнт-сервер, де клієнти надсилають запити на сервер та отримують відповіді. Патерн Команда використовується для реалізації об'єктів-команд, які представляють різні IRC-команди, що можуть виконуватися клієнтом. Патерн Фабричний Метод використовується для створення об'єктів валідації конфігурації сервера, дозволяючи гнучко масштабувати процес валідації. Патерн Міст використовується для розділення клієнта та серверного коду, що дозволяє зберігати їх незалежними та легко розширюваними. Патерн Компоновщик застосований для створення складних команд, які можуть складатися з кількох простіших команд та виконуватися разом. Система використовує підходи ООП для організації коду, забезпечуючи високий рівень розширюваності та підтримки майбутніх змін. Комунікація між клієнтом та сервером здійснюється за допомогою сокетів, що забезпечує надійний обмін даними між компонентами. Інтерфейс користувача (GUI) клієнта реалізований як окремий компонент, що полегшує його розширення та зміну. Патерн Фабричний Метод застосований для створення екземплярів об'єктів валідації конфігурації сервера з різними правилами валідації. Принцип інкапсуляції дозволяє приховати внутрішню реалізацію класів та взаємодіяти з ними лише через визначені інтерфейси. Патерн Міст сприяє розділенню обов'язків між клієнтом та сервером, покращуючи читабельність та обслуговуваність коду. Код системи має високий рівень абстракції, що дозволяє легко змінювати та доповнювати функціонал. Принцип поліморфізму дозволяє об'єктам різних класів використовувати один і той же інтерфейс. Патерн Компоновщик полегшує додавання нових команд та комбінування їх в складні

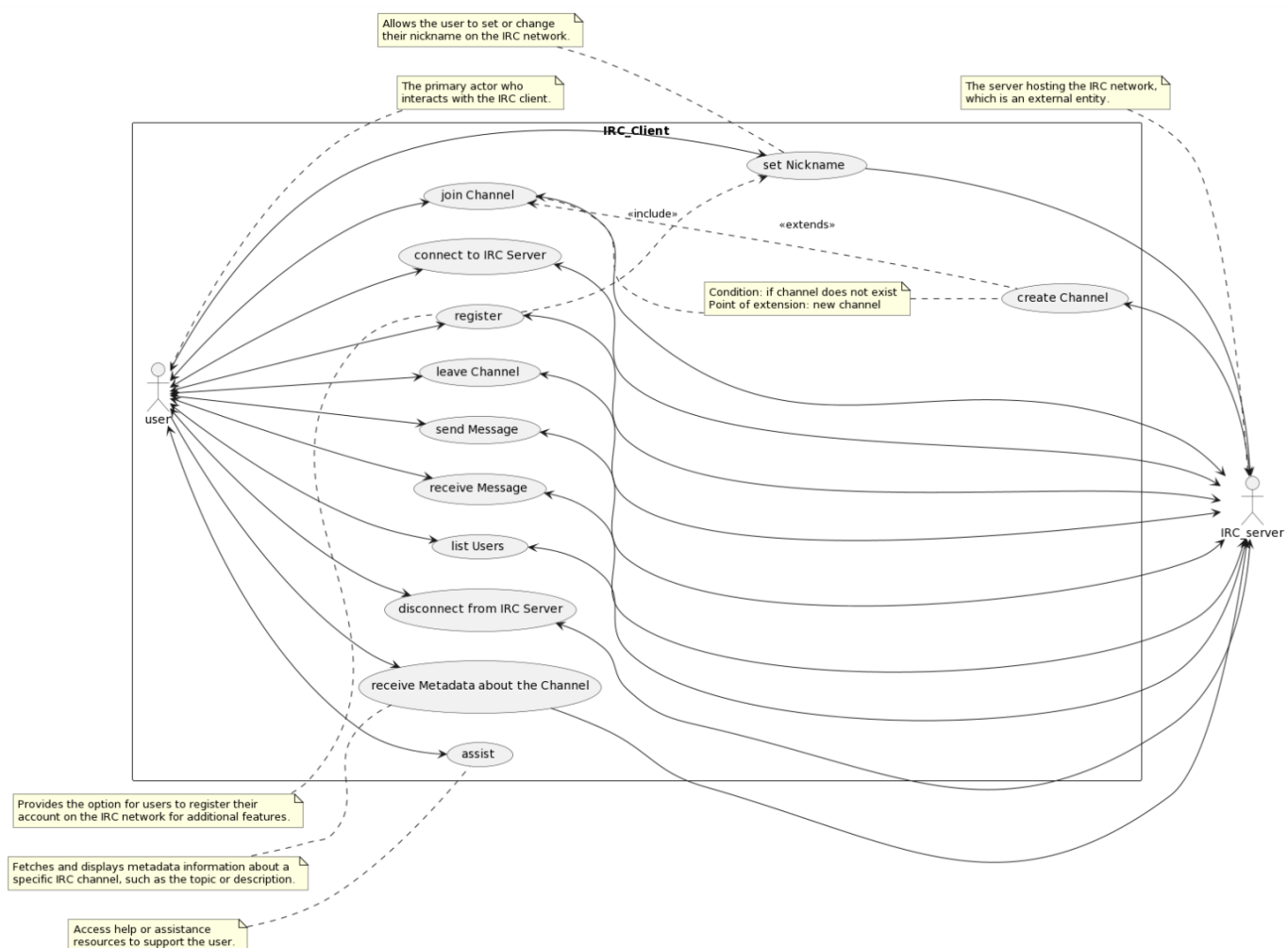
команди. Використання патернів дозволяє виконувати принципи SOLID, полегшуючи розширення та підтримку коду в майбутньому. Код системи є модульним та легко розширюється, завдяки чому він може бути використаний як основа для подальших розробок. Реалізація патернів дозволяє досягти гнучкості та підтримує простоту управління функціоналом системи.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

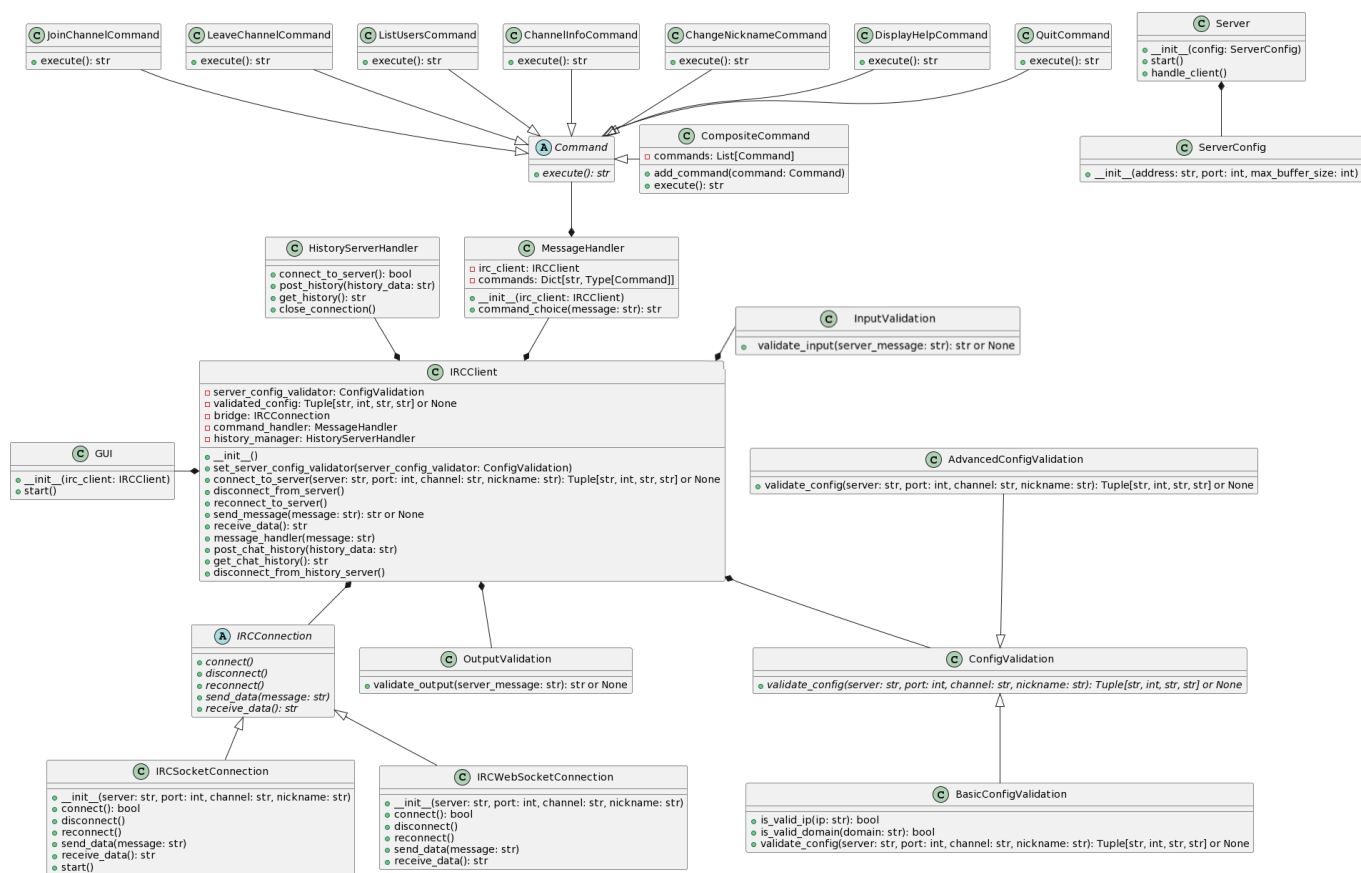
- [1] А. В. Яковенко, та О. О. Коновал, “методичні вказівки до виконання комп’ютерних практикумів з дисципліни «Управління ІТ-проектами». Управління ІТ-проектами.”, К.: НТУУ «КПІ ім. І. Сікорського», с. 21, 2017.
- [2] Cambridge University Press, “The Elements of UML 2.0 Style”, с. 33 - 46, 2005.
- [3] Д. Батра, “Концептуальне моделювання даних”, Журнал управління базами даних 16 (2005) с. 84-106.
- [4] Джеймс Рамбо, Івар Якобсон, Грейді Буч, “Довідковий посібник зі згуртованої мови моделювання.”, Addison Wesley Longman Inc., с. 190, 1999.
- [5] Джеймс Рамбо, Івар Якобсон, Грейді Буч, “Довідковий посібник зі згуртованої мови моделювання.”, Addison Wesley Longman Inc., с. 252, 1999.
- [6, с. 39] Джеймс Рамбо, Івар Якобсон, Грейді Буч, “Довідковий посібник зі згуртованої мови моделювання.”, Addison Wesley Longman Inc., с. 222, 1999.
- [7] Джеймс Рамбо, Івар Якобсон, Грейді Буч, “Довідковий посібник зі згуртованої мови моделювання.”, Addison Wesley Longman Inc., с. 87, 1999.
- [8] Олександр Швець, "Занурення в ШАБЛони проектування", RefactoringGuru, с. 268, 2019.
- [9] Олександр Швець, "Занурення в ШАБЛони проектування", RefactoringGuru, с. 71, 2019.
- [10] Олександр Швець, "Занурення в ШАБЛони проектування", RefactoringGuru, с. 162, 2019.
- [11] Олександр Швець, "Занурення в ШАБЛони проектування", RefactoringGuru, с. 177, 2019.

ДОДАТКИ

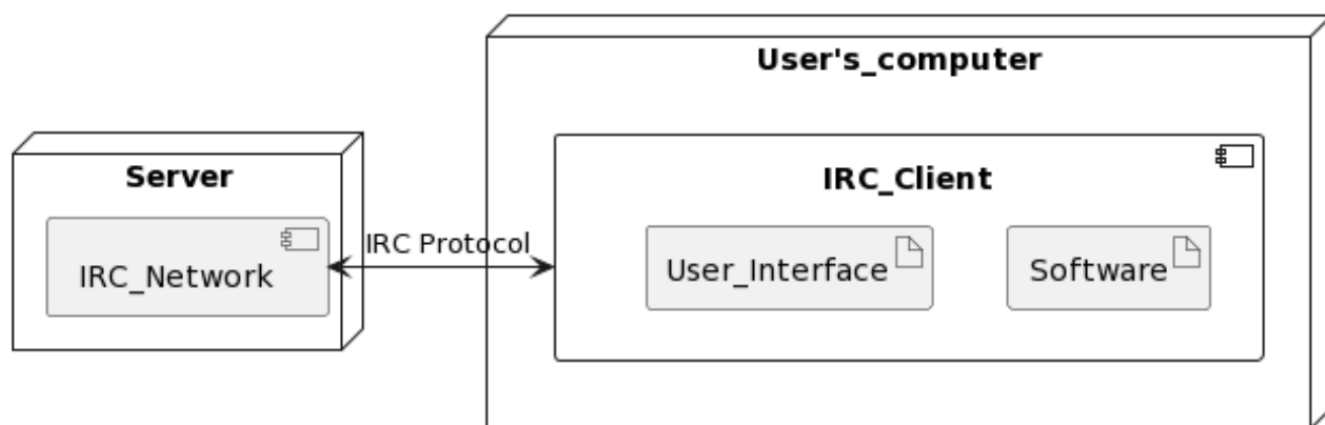
Додаток А



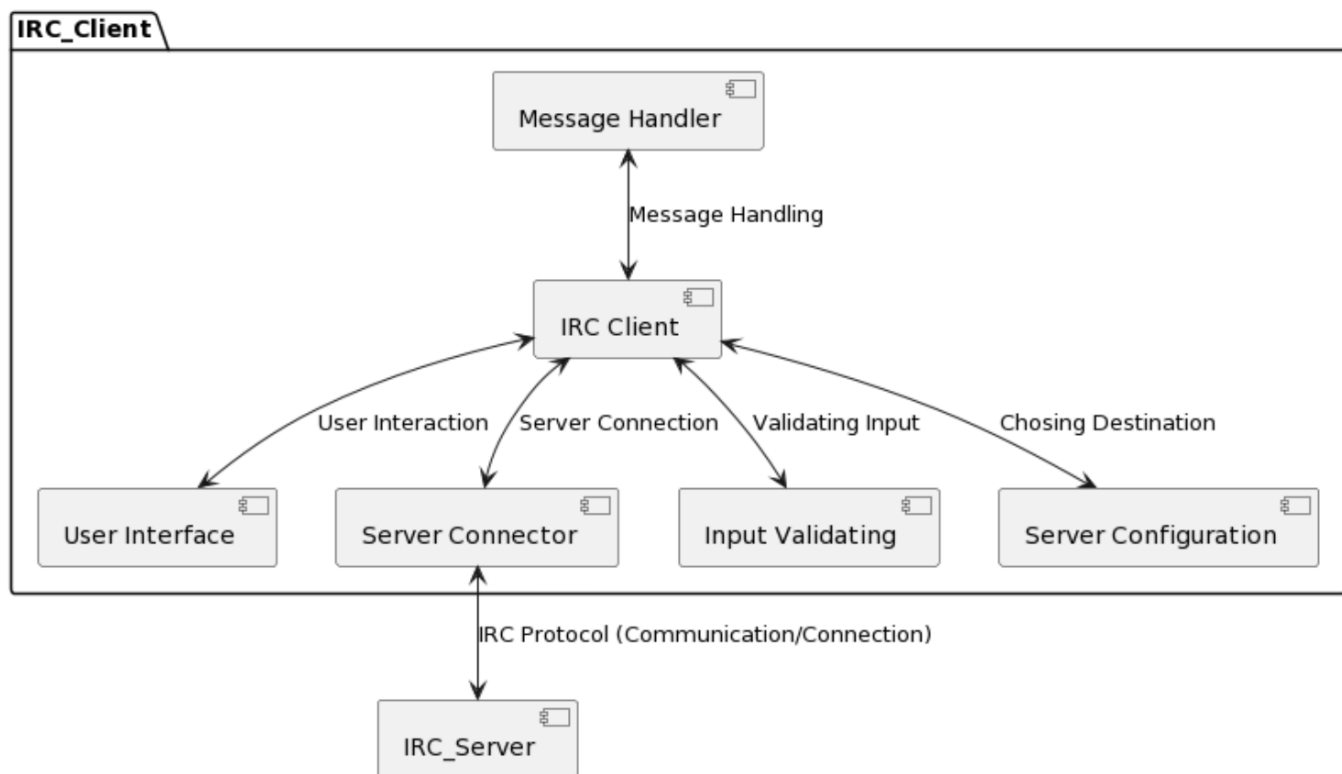
Додаток Б



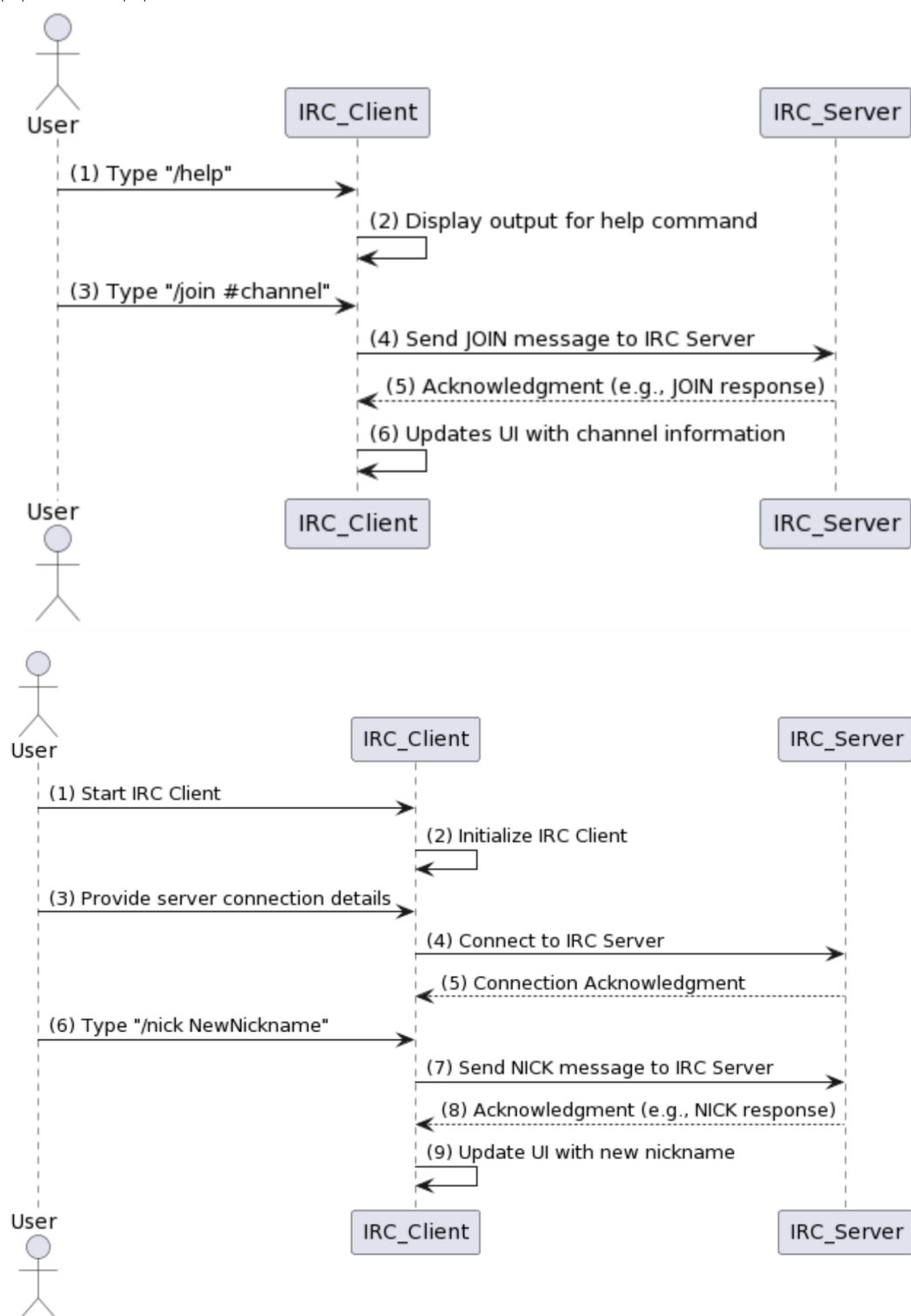
Додаток В



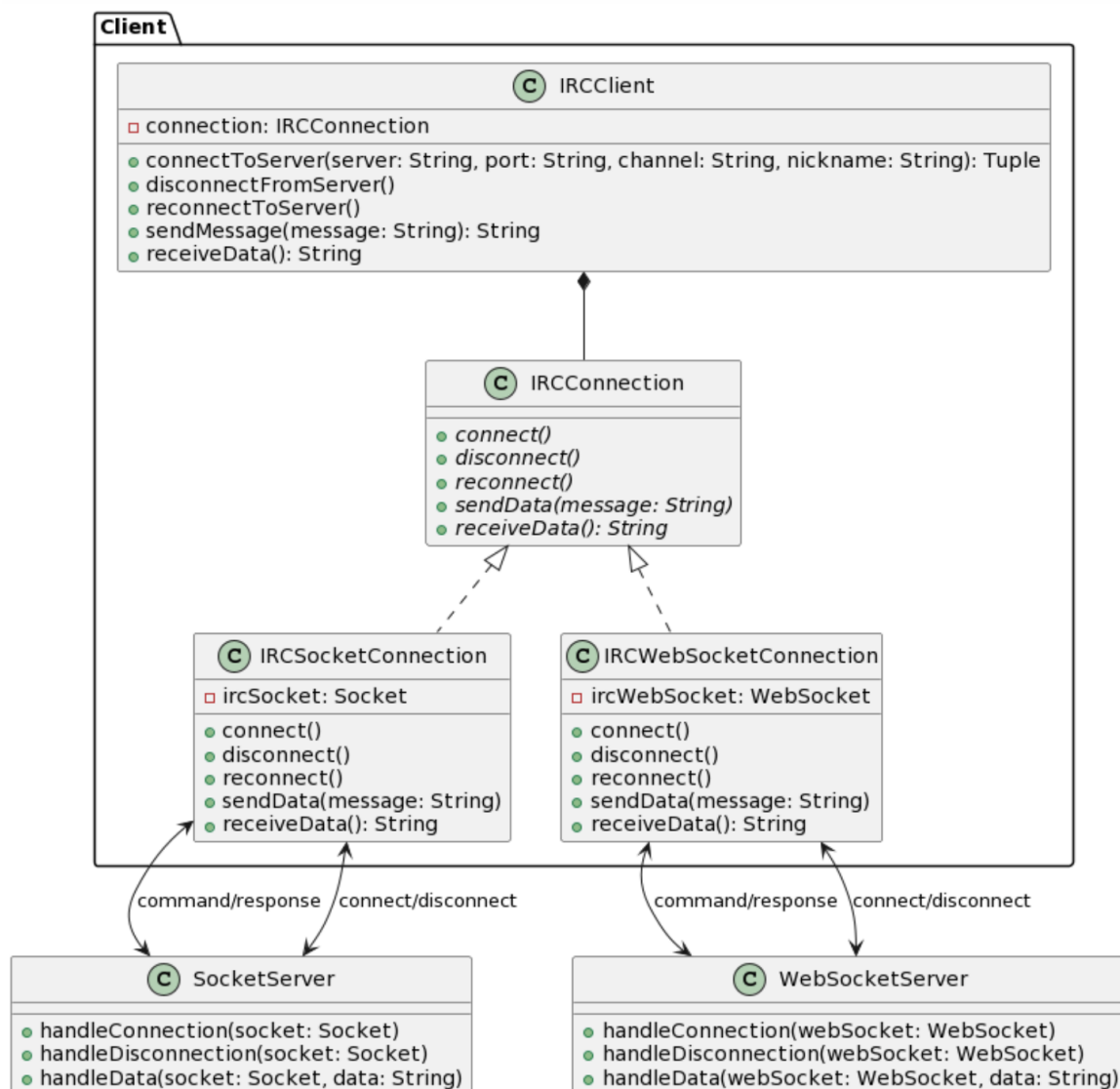
Додаток Г

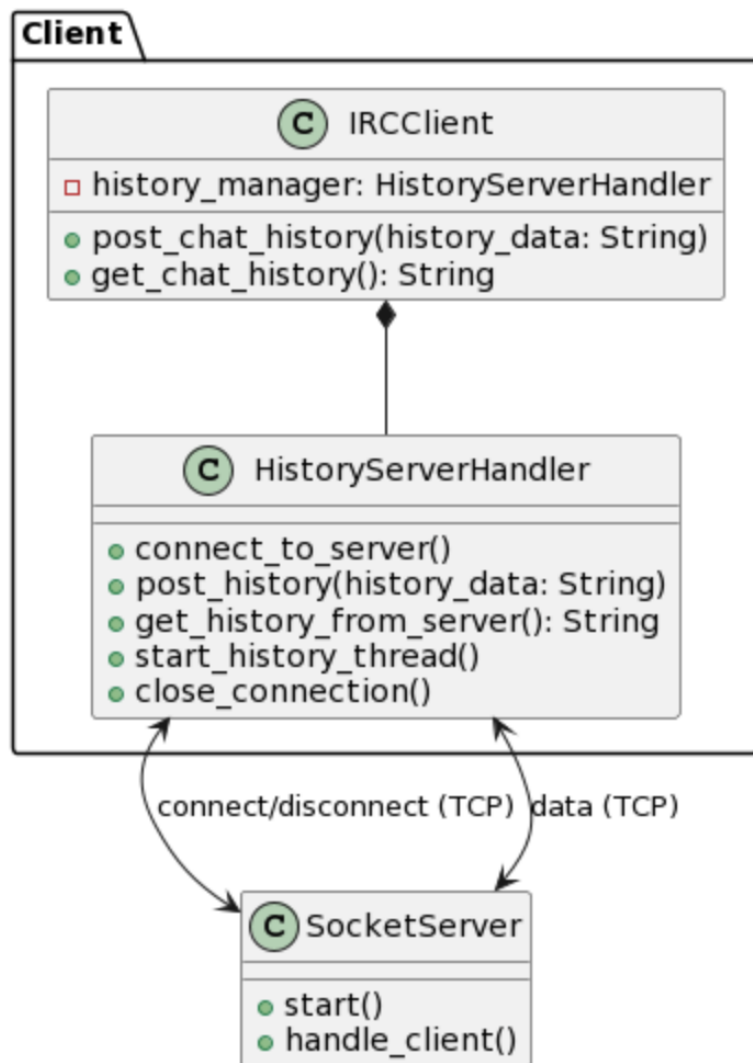


Додаток Д

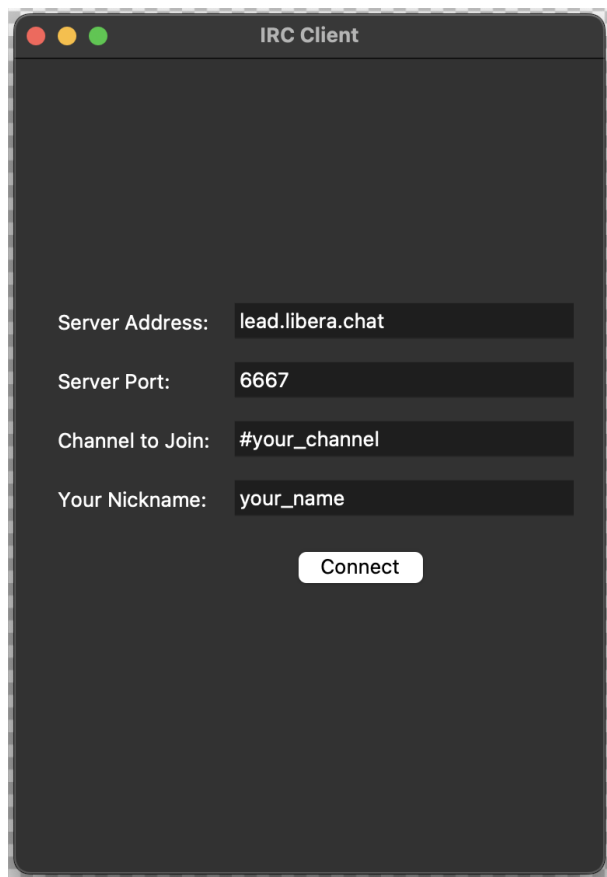


Додаток Е





Додаток Є



The image shows a screenshot of a macOS-style window titled "IRC Client". The window has a dark gray background and rounded corners. At the top, there are three colored window control buttons (red, yellow, green) on the left. Below the title bar, there are four text input fields arranged vertically. Each field has a label to its left and a text input area to its right. The labels are "Server Address:", "Server Port:", "Channel to Join:", and "Your Nickname:". The input areas contain the text "lead.libera.chat", "6667", "#your_channel", and "your_name" respectively. Below these fields, centered horizontally, is a white button with the text "Connect".

IRC Client

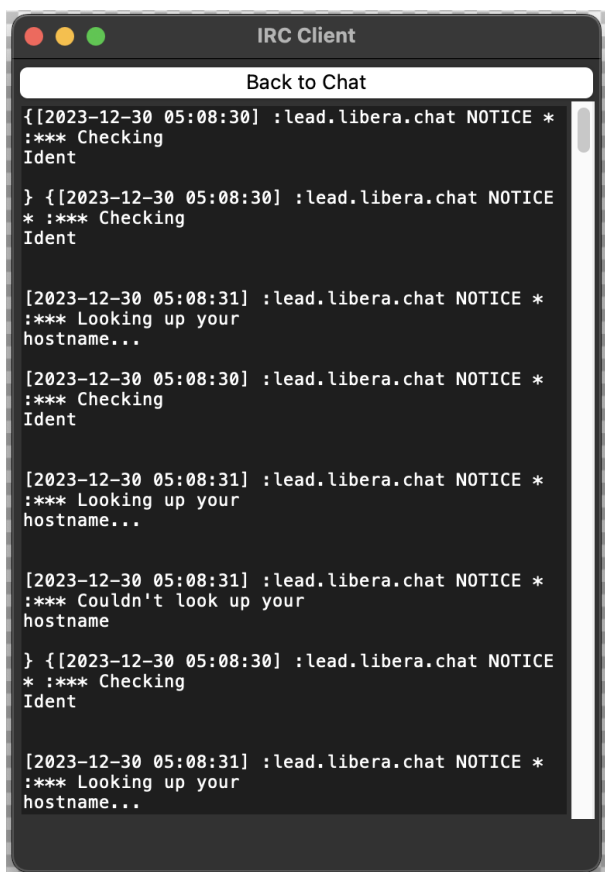
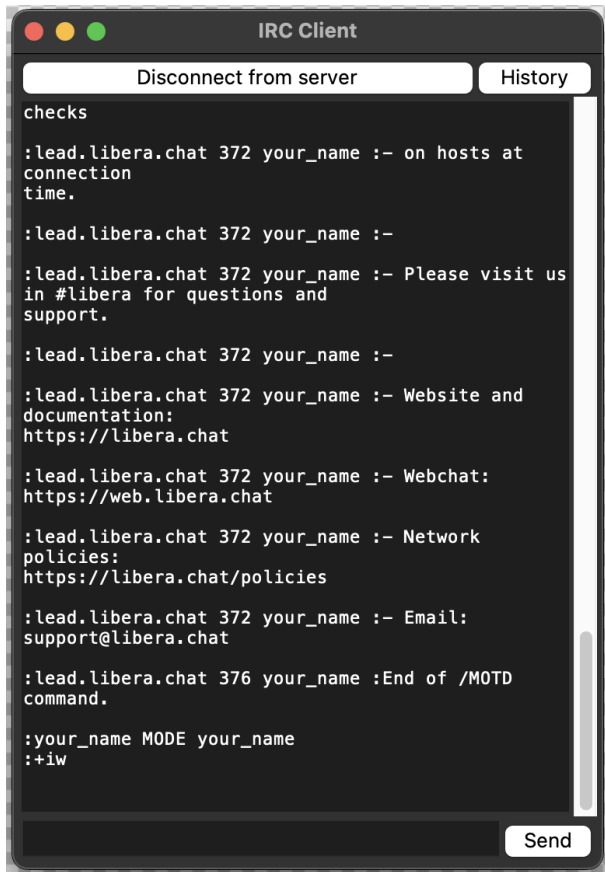
Server Address: lead.libera.chat

Server Port: 6667

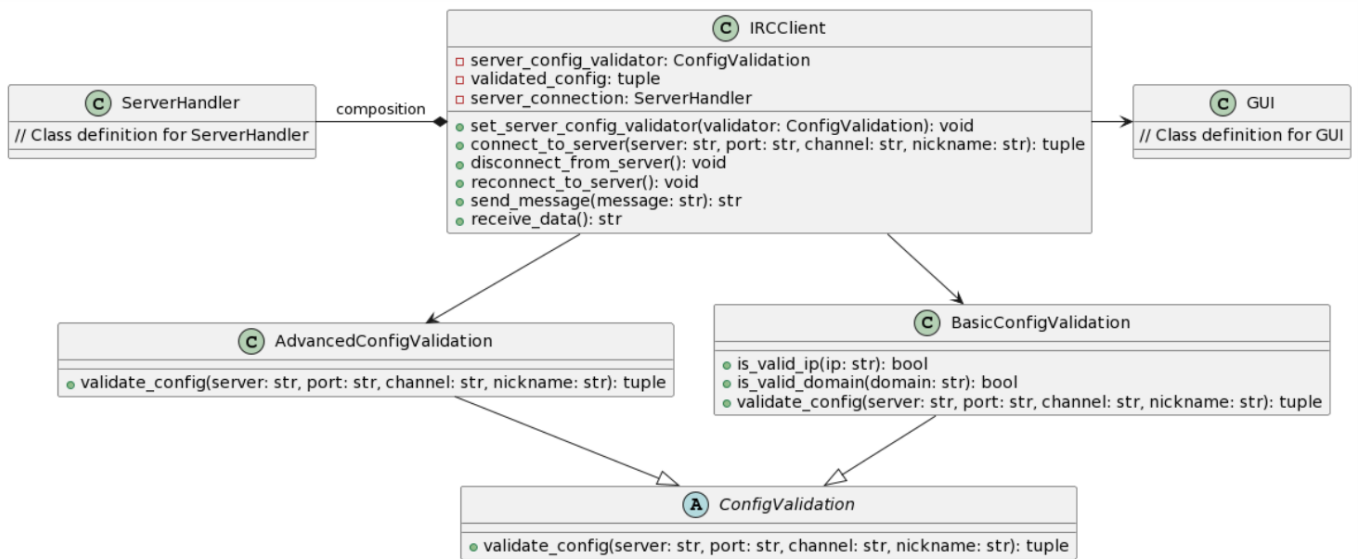
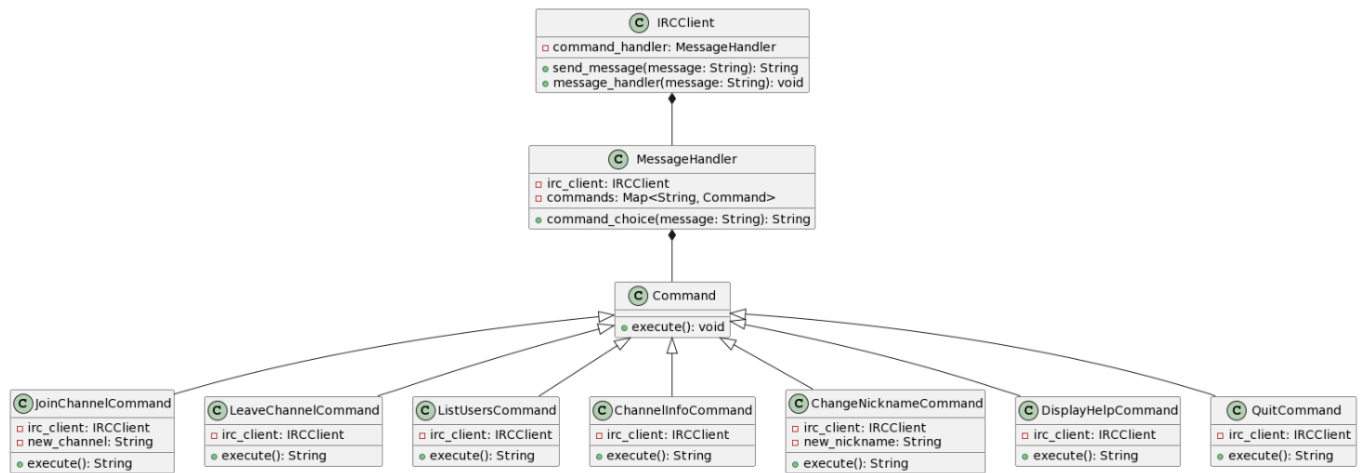
Channel to Join: #your_channel

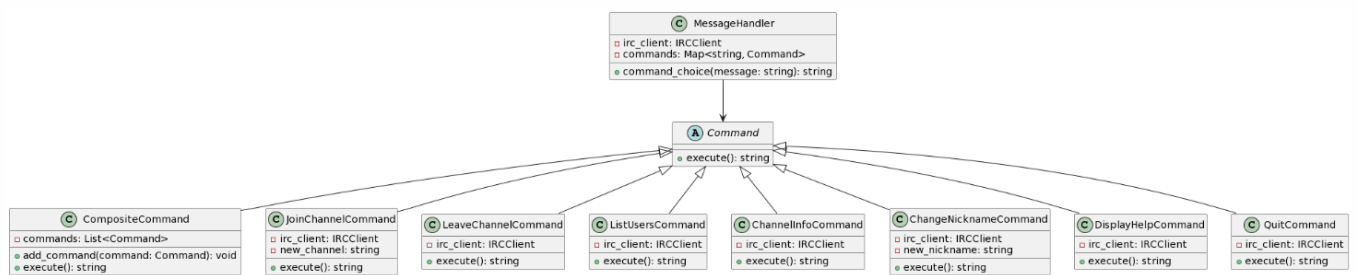
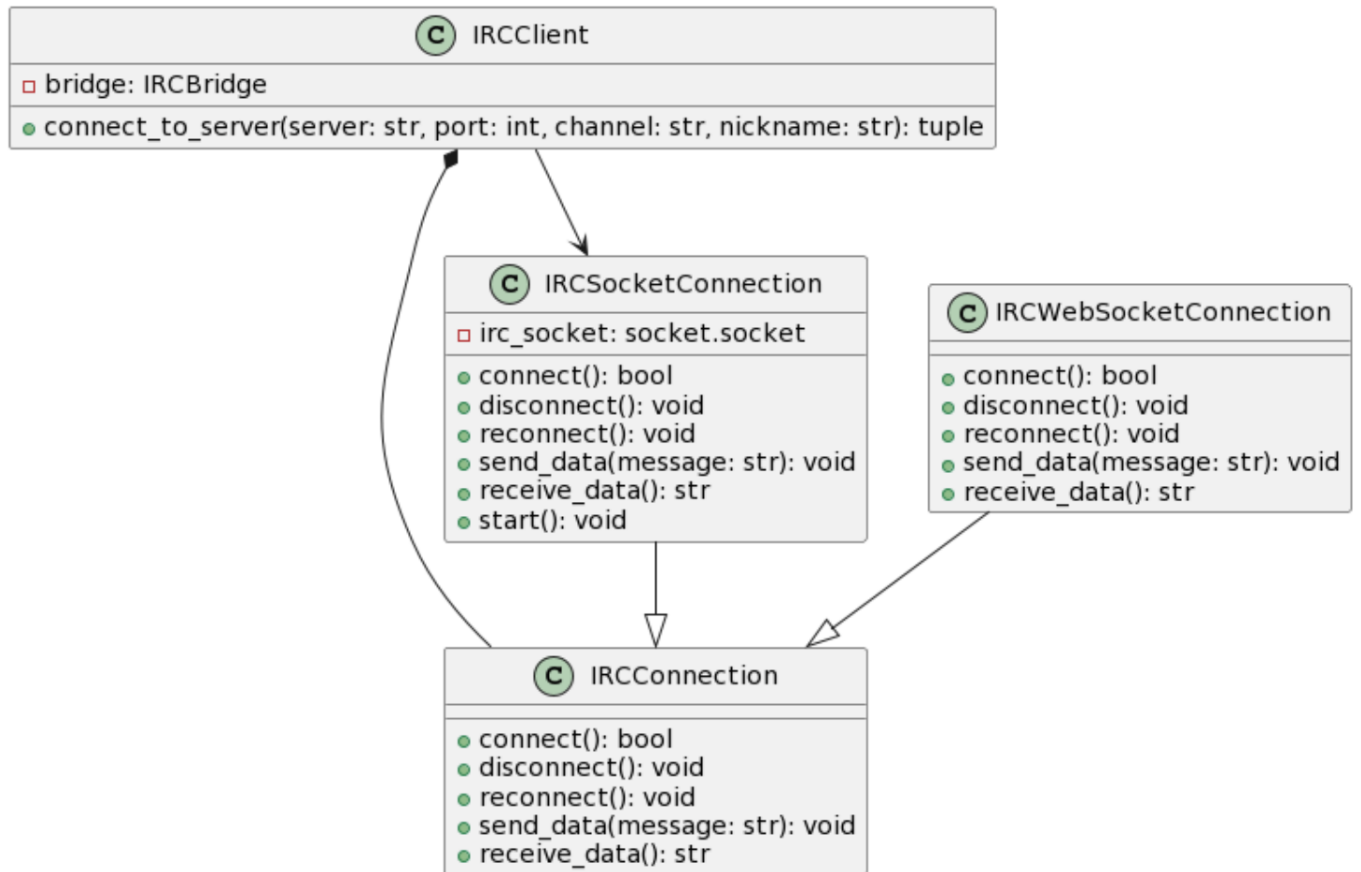
Your Nickname: your_name

Connect

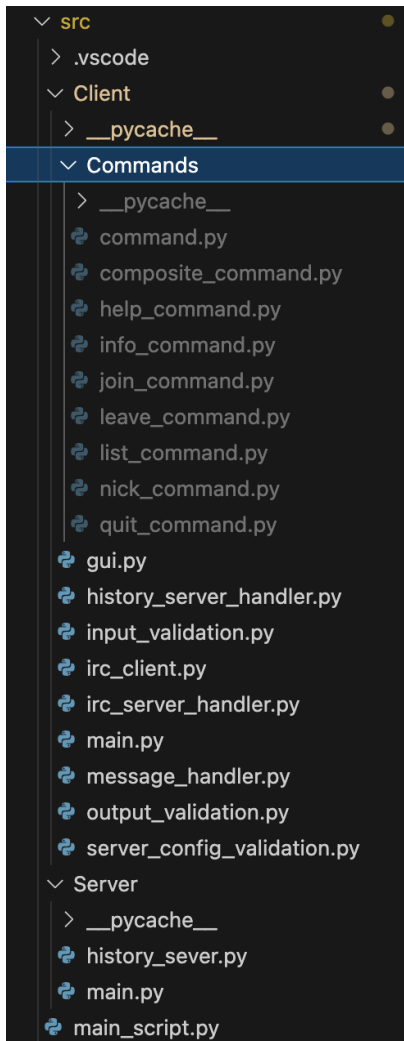


Додаток Ж





Додаток 3



command.py

from abc import ABC, abstractmethod

class Command(ABC):

"""

Abstract base class for IRC command classes.

"""

```
@abstractmethod
```

```
def execute(self):
```

```
    """
```

```
    Execute the IRC command.
```

```
    """
```

```
    pass
```

```
composite_command.py
```

```
from Client.Commands.command import Command
```

```
class CompositeCommand(Command):
```

```
    """
```

```
    A command that aggregates and executes multiple commands.
```

```
    """
```

```
def __init__(self):
```

```
    """
```

```
    Initialize an instance of CompositeCommand.
```

```
    """
```

```
    self.commands = []
```

```
def add_command(self, command):
```

```
    """
```

Add a command to the list of commands to be executed.

Parameters:

- command: An instance of a Command subclass.

```
    """
```

```
    self.commands.append(command)
```

```
def execute(self):
```

```
    """
```

Execute each command in the list and concatenate the results.

Returns:

- str: The concatenated result of executing each command.

```
    """
```

```
    result = ""
```

```
    for command in self.commands:
```

```
        result += command.execute()
```

```
return result
```

```
help_command.py
```

```
from Client.Commands.command import Command
```

```
class DisplayHelpCommand(Command):
```

```
    """
```

```
    Command class for handling the "/help" command.
```

```
    """
```

```
    def __init__(self, irc_client):
```

```
        """
```

```
        Initialize the DisplayHelpCommand.
```

```
        Parameters:
```

```
        - irc_client: The IRC client instance.
```

```
        """
```

```
        self.irc_client = irc_client
```

```
    def execute(self):
```



```
"""
```

Display available commands and their descriptions.

Returns:

- Help message string.

```
"""
```

```
return "Available commands:\n/join - Join a channel/create a new channel\n/part - Leave
the current channel\n/names - List users in the current channel\n/info - Display
information about the current channel\n/nick - Change your nickname. Usage: /nick
<new_nickname>\n/quit - Gracefully disconnect from the server\n/help - Display this help
message\n"
```

join_command.py

```
from Client.Commands.command import Command
```

```
class JoinChannelCommand(Command):
```

```
"""
```

Command class for handling the "/join" command.

```
"""
```

```
def __init__(self, irc_client, new_channel):
```

```
    """
```

Initialize the JoinChannelCommand.

Parameters:

- irc_client: The IRC client instance.
- new_channel: The new channel to join.

```
    """
```

```
    self.irc_client = irc_client
```

```
    self.new_channel = new_channel
```

```
def execute(self):
```

```
    """
```

Join a new IRC channel.

Returns:

- The IRC JOIN message string.

```
    """
```

```
    self.irc_client.channel = self.new_channel
```

```
return f"JOIN {self.irc_client.channel}\r\n"
```

leave_command.py

```
from Client.Commands.command import Command
```

```
class LeaveChannelCommand(Command):
```

```
    """
```

Command class for handling the "/part" command.

```
    """
```

```
    def __init__(self, irc_client):
```

```
        """
```

Initialize the LeaveChannelCommand.

Parameters:

- irc_client: The IRC client instance.

```
        """
```

```
        self.irc_client = irc_client
```

```
def execute(self):
```

```
    """
```

Leave the current IRC channel.

Returns:

- The IRC PART message string.

```
    """
```

```
    return f"PART {self.irc_client.channel}\r\n"
```

list_command.py

```
from Client.Commands.command import Command
```

```
class ListUsersCommand(Command):
```

```
"""
```

Command class for handling the "/names" command.

```
"""
```

```
def __init__(self, irc_client):
```

```
"""
```

Initialize the ListUsersCommand.

Parameters:

- irc_client: The IRC client instance.

```
"""
```

```
self.irc_client = irc_client
```

```
def execute(self):
```

```
"""
```

List users in the specified IRC channel.

Returns:

- The IRC NAMES message string.

```
"""
```

```
return f'NAMES {self.irc_client.channel}\r\n'
```

```
nick_command.py
```

```
from Client.Commands.command import Command
```

```
class ChangeNicknameCommand(Command):
```

```
    """
```

```
    Command class for handling the "/nick" command.
```

```
    """
```

```
    def __init__(self, irc_client, new_nickname):
```

```
        """
```

```
        Initialize the ChangeNicknameCommand.
```

```
        Parameters:
```

- irc_client: The IRC client instance.
- new_nickname: The new nickname to set.

```
        """
```

```
        self.irc_client = irc_client
```

```
        self.new_nickname = new_nickname
```

```
def execute(self):
```

```
"""
```

Change the user's nickname.

Returns:

- The IRC NICK message string.

```
"""
```

```
return f"NICK {self.new_nickname}\r\n"
```

quit_command.py

```
from Client.Commands.command import Command
```

```
class QuitCommand(Command):
```

```
"""
```

Command class for handling the "/quit" command.

```
"""
```

```
def __init__(self, irc_client):
```

```
    """
```

Initialize the QuitCommand.

Parameters:

- irc_client: The IRC client instance.

```
    """
```

```
self.irc_client = irc_client
```

```
def execute(self):
```

```
    """
```

Gracefully disconnect from the IRC server.

Returns:

- The IRC QUIT message string.

```
    """
```

```
return f'QUIT :{self.irc_client.nickname} is disconnecting\r\n'
```



```
gui.py
```

```
import threading
```

```
import tkinter as tk
```

```
from tkinter import messagebox
```

```
from tkinter import scrolledtext
```

```
class GUI:
```

```
def __init__(self, irc_client):
```

```
    """
```

```
    Initialize the GUI for the IRC client.
```

```
    Parameters:
```

```
    - irc_client (IRCCClient): An instance of the IRCCClient class.
```

```
    """
```

```
        self.irc_client = irc_client
```

```
        self.root = tk.Tk()
```

```
        self.root.title("IRC Client")
```

```
self.root.geometry("380x525")
```

```
self.current_frame = None
```

```
self.registration_frame()
```

```
def registration_frame(self):
```

```
    """
```

```
    Show the first frame of the GUI, allowing users to input server details.
```

```
    """
```

```
    if self.current_frame:
```

```
        self.current_frame.destroy() # Destroy the current frame
```

```
    self.current_frame = tk.Frame(self.root)
```

```
    # Labels and entry widgets for server details
```

```
    self.server_label = tk.Label(self.current_frame, text="Server Address:")
```

```
    self.server_entry = tk.Entry(self.current_frame, width=30)
```

```
    self.server_entry.insert(tk.END, "lead.libera.chat") # Set default value
```

```
    self.port_label = tk.Label(self.current_frame, text="Server Port:")
```

```
    self.port_entry = tk.Entry(self.current_frame, width=30)
```

```
self.port_entry.insert(tk.END, "6667") # Set default value
```

```
self.channel_label = tk.Label(self.current_frame, text="Channel to Join:")
```

```
self.channel_entry = tk.Entry(self.current_frame, width=30)
```

```
self.channel_entry.insert(tk.END, "#your_channel") # Set default value
```

```
self.nickname_label = tk.Label(self.current_frame, text="Your Nickname:")
```

```
self.nickname_entry = tk.Entry(self.current_frame, width=30)
```

```
self.nickname_entry.insert(tk.END, "your_name") # Set default value
```

```
self.connect_button = tk.Button(self.current_frame, text="Connect",  
command=self.send_server_info)
```

```
self.server_label.grid(row=0, column=0, padx=5, pady=5, sticky="w")
```

```
self.server_entry.grid(row=0, column=1, padx=5, pady=5)
```

```
self.port_label.grid(row=1, column=0, padx=5, pady=5, sticky="w")
```

```
self.port_entry.grid(row=1, column=1, padx=5, pady=5)
```

```
self.channel_label.grid(row=2, column=0, padx=5, pady=5, sticky="w")
```

```

self.channel_entry.grid(row=2, column=1, padx=5, pady=5)

self.nickname_label.grid(row=3, column=0, padx=5, pady=5, sticky="w")

self.nickname_entry.grid(row=3, column=1, padx=5, pady=5)

self.connect_button.grid(row=4, column=0, columnspan=2, pady=10)

# Center the frame within the root window

self.current_frame.pack(expand=True, fill="both", padx=20, pady=150)

self.current_frame.pack()


def send_server_info(self):
    """
    Validate server details, connect to the server, and switch to the second frame.
    Display error messages for invalid configurations or connection issues.
    """

    server = self.server_entry.get().strip()

    port = self.port_entry.get().strip()

    channel = self.channel_entry.get().strip()

    nickname = self.nickname_entry.get().strip()


    validated_config = self.irc_client.connect_to_server(server, port, channel, nickname)

    if validated_config is not None:

```

```
self.chatting_frame()
```

```
else:
```

```
messagebox.showerror("Error", "Invalid configuration. Please enter valid details.")
```

```
def chatting_frame(self):
```

```
    """
```

```
    Show the second frame of the GUI, displaying chat functionality.
```

```
    """
```

```
    if self.current_frame:
```

```
        self.current_frame.destroy() # Destroy the current frame
```

```
        self.current_frame = tk.Frame(self.root)
```

```
        # Button to disconnect from the server
```

```
        disconnect_button = tk.Button(self.current_frame, text="Disconnect from server",
                                       command=self.disconnect_from_server, width=29)
```

```
        disconnect_button.grid(row=0, column=0, pady=0, sticky="w")
```

```
        # Button to get chat history
```

```
        history_button = tk.Button(self.current_frame, text="History",
                                    command=self.get_chat_history)
```

```
        history_button.grid(row=0, column=0, pady=0, sticky="e")
```

```
# Scrollable text area for displaying messages
```

```
self.text_area = scrolledtext.ScrolledText(self.current_frame, wrap=tk.WORD, width=50,  
height=35)
```

```
self.text_area.grid(row=1, column=0, padx=0, pady=0)
```

```
self.text_area.configure(state="disabled")
```

```
# Entry widget for typing messages
```

```
self.message_entry = tk.Entry(self.current_frame, width=35)
```

```
self.message_entry.grid(row=2, column=0, padx=1, pady=0, sticky="w")
```

```
# Button to send messages
```

```
send_button = tk.Button(self.current_frame, text="Send", command=self.send_message,  
width=3, height=1)
```

```
send_button.grid(row=2, column=0, padx=0, pady=0, sticky="e")
```

```
self.current_frame.pack()
```

```
# Start a thread to continuously receive data and update the text area
```

```

output_thread = threading.Thread(target=self.give_output)

output_thread.start()

def disconnect_from_server(self):
    """
    Disconnect from the IRC server and show a message to the user.
    """
    try:
        if self.irc_client:
            self.irc_client.disconnect_from_server()

            messagebox.showinfo("Disconnected", "Disconnected from the server.")

            self.registration_frame()

    except Exception as e:
        messagebox.showerror("Error", f"An unexpected error occurred: {e}")

def send_message(self):
    """
    Send a message to the IRC server based on user input.

    Validate the message, delegate it to the message handler, and update the text area.

    Display an error message for unexpected issues during message handling.
    """
    try:

```

```

message = self.message_entry.get().strip()

if message:

    validated_message = self.irc_client.send_message(message)

    if validated_message is not None:

        # Set the text area to normal to enable insertion

        self.text_area.configure(state="normal")

        self.text_area.insert(tk.END, f"You: {validated_message}\n")

        self.text_area.yview(tk.END) # Ensure the text_area is scrolled to the end

        # Set the text area back to disabled

        self.text_area.configure(state="disabled")

        self.message_entry.delete(0, tk.END)

    except Exception as e:

        messagebox.showerror("Error", f"An unexpected error occurred: {e}")


def give_output(self):
    """
    Continuously receive data from the server and display it in the text area.

    Handle disconnection events and attempt to reconnect.

    """
    try:
        while True:

```



```

# Receive and display output from the server

output = self.irc_client.receive_data()

self.text_area.configure(state="normal")

self.text_area.insert(tk.END, output)

self.text_area.yview(tk.END) # Ensure the text_area is scrolled to the end

self.text_area.configure(state="disabled")

if "Connection closed by server." in output:

    # Handle disconnection by showing a message and reconnecting

    messagebox.showinfo("Disconnected", "Disconnected from the server.")

    self.reconnect_to_server()

except Exception as e:

    messagebox.showerror("Error", f"An unexpected error occurred: {e}")


def reconnect_to_server(self):

    """

    Attempt to reconnect to the IRC server and show a message to the user.

    """

    try:

        if self.irc_client:

            self.irc_client.reconnect_to_server()

            messagebox.showinfo("Reconnecting", "Reconnecting to the server...")

```

```

self.chatting_frame(self.irc_client.validated_config)

except Exception as e:

messagebox.showerror("Error", f"An unexpected error occurred: {e}")

```

```

def chat_history_frame(self, chat_history):

    """

    Show a frame with a scrolled text area to display chat history.

    """

    if self.current_frame:

        self.current_frame.destroy() # Destroy the current frame

        self.current_frame = tk.Frame(self.root)


    # Button to go back to the chatting frame

    back_button = tk.Button(self.current_frame, text="Back to Chat",
        command=self.chatting_frame, width=38)

    back_button.grid(row=0, column=0, pady=0, sticky="w")

```

```

# Scrollable text area for displaying chat history

history_text_area = scrolledtext.ScrolledText(self.current_frame, wrap=tk.WORD,
width=50, height=35)

history_text_area.grid(row=1, column=0, padx=0, pady=0)

history_text_area.insert(tk.END, chat_history)

history_text_area.configure(state="disabled")


self.current_frame.pack()


def get_chat_history(self):
    """
    Retrieve and display chat history.
    """
    try:
        chat_history = self.irc_client.get_chat_history()

        self.chat_history_frame(chat_history)

    except Exception as e:
        messagebox.showerror("Error", f"An unexpected error occurred: {e}")

```

```

def start(self):
    """
    Start the Tkinter main loop for the GUI.
    """
    try:
        self.root.mainloop()
    except Exception as e:
        messagebox.showerror("Error", f"An unexpected error occurred: {e}")
    finally:
        # Ensure proper cleanup when the application is closed
        if hasattr(self, 'irc_client'):
            self.irc_client.disconnect_from_server()

```

history_server_handler.py

```
import socket
```

```
import time
```

```
import threading

MAX_BUFFER_SIZE = 2048

# Global variable to store chat history
chat_history = []


class HistoryServerHandler:

    def __init__(self, server_address, server_port):

        self.server_address = server_address

        self.server_port = server_port

        self.history_socket = None


    def connect_to_server(self):

        """

        Connect to the server to establish a socket for history communication.

        """

        try:

            self.history_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

            self.history_socket.connect((self.server_address, self.server_port))
```

```

self.start_history_thread()

return True

except Exception as e:

print(f'Error connecting to history server: {e}')

return False

```

```

def post_history(self, history_data):
    """
    Send chat history data to the server.
    """
    try:
        if self.history_socket:
            self.history_socket.sendall(history_data.encode('utf-8'))
    except Exception as e:
        print(f'Error posting history to server: {e}')

```

```

def get_history_from_server(self):
    """
    Continuously receive chat history data from the server.
    """

```

```

while True:

    try:

        if self.history_socket:

            data = self.history_socket.recv(MAX_BUFFER_SIZE).decode('utf-8')

            chat_history.append(data)


            time.sleep(1) # Add a short delay to avoid excessive CPU usage

        except Exception as e:

            print(f"Error getting history from server: {e}")

```

```

def get_history(self):

    """

    Give history to the Client.

    """

    return chat_history

def start_history_thread(self):

    """

    Start a thread to continuously get information from the server.

```

```
"""
```

```
history_thread = threading.Thread(target=self.get_history_from_server)
```

```
history_thread.daemon = True # Set the thread as a daemon so it will exit when the main
program exits.
```

```
history_thread.start()
```

```
def close_connection(self):
```

```
"""
```

```
Close the socket connection to the server.
```

```
"""
```

```
try:
```

```
if self.history_socket:
```

```
self.history_socket.close()
```

```
except Exception as e:
```

```
print(f'Error closing history socket: {e}')
```

```
input_validation.py
```

```
class InputValidation:
```

```
def validate_input(user_input):
```



```
"""
```

Validate user input before sending it to the server.

Parameters:

- user_input: The user input to be validated.

Returns:

- The validated user input or None if input is invalid.

```
"""
```

```
# Remove leading and trailing whitespaces
```

```
user_input = user_input.strip()
```

```
# Check if the input is None
```

```
if user_input is None:
```

```
    return None
```

```
# Modify user input based on specific commands
```

```
if user_input[0:5].lower() == "/join":
```

```
    new_channel = user_input.split()[1]
```

```
    if new_channel.startswith("#"):
```

```

pass

else:

new_channel = "#" + new_channel

user_input = "/join " + new_channel

elif user_input[0:5].lower() == "/part":

user_input = user_input.split()[0]

elif user_input[0:6].lower() == "/names":

user_input = user_input.split()[0]

elif user_input[0:5].lower() == "/info":

user_input = user_input.split()[0]

elif user_input.lower().startswith("/nick"):

new_nickname = ""

if user_input != "/nick":

new_nickname = user_input.split()[1]

if new_nickname is None:

new_nickname = "not_given"

user_input = "/nick " + new_nickname

elif user_input[0:5].lower() == "/help":

# Help Messages

help_messages = {

"join": "Join a channel/create new channel",

"part": "Leave the current channel",

```

```

"names": "List users in the current channel",
"info": "Display information about the current channel",
"nick": "Change your nickname. Usage: /nick <new_nickname>",
"quit": "Gracefully disconnect from the server",
"help": "Display this help message"
}

```

```

helping = "_____Available commands:_____\\n"

```

```

for command, description in help_messages.items():

```

```

    helping += f"/{command} - {description}\\n"

```

```

    helping += "\\n"

```

```

    user_input = helping

```

```

elif user_input[0:5].lower() == "/quit":

```

```

    user_input = user_input.split()[0]

```

```

    return user_input

```

```

irc_client.py

```

```

from Client.irc_server_handler import IRCSocketConnection

```

```

from Client.input_validation import InputValidation

```

```

from Client.output_validation import OutputValidation

from Client.message_handler import MessageHandler

from Client.history_server_handler import HistoryServerHandler


# to improve:

#

# “/help” command fix

#

# fix bug when input validation don't allow to execute multiple commands with parameters

#

# split registration and chatting windows from gui class

#

# implement “factory method” pattern to each validation

#

# solve bug when after registration can't request “/info”, “/names” command because
channel is not initialized in IRCClient class only in SocketBridge class

#

# when opening chat_history_frame in gui "history_text_area.insert(tk.END,
chat_history)" cannot be performed

# bcs "history_text_area.configure(state="disabled

# ")" and it show error in message box (as i think)

```

```
class IRCClient:
```

```
def __init__(self):
```

```
"""
```

Initialize the IRCClient class.

Attributes:

- validated_config (tuple): A tuple containing validated server configuration.
- server_connection (ServerHandler): An instance of the ServerHandler class for server communication.

```
"""
```

```
self.server_config_validator = None
```

```
self.validated_config = None
```

```
self.bridge = None
```

```
self.command_handler = None
```

```
self.history_manager = HistoryServerHandler('127.0.0.1', 12340)
```

```
# Connect to the history server
```

```
history_connected = self.history_manager.connect_to_server()
```

```
if not history_connected:
```

```
print("Failed to connect to the history server.")
```

```
def set_server_config_validator(self, server_config_validator):
```

```
    """
```

Set the server configuration validator for the IRC client.

Parameters:

- server_config_validator (InputValidation): An instance of the InputValidation class for server configuration validation.

```
    """
```

```
self.server_config_validator = server_config_validator
```

```
def connect_to_server(self, server, port, channel, nickname):
```

```
    """
```

Connect to the IRC server with the provided configuration.

Parameters:

- server (str): The server address.
- port (str): The server port.
- channel (str): The channel to join.
- nickname (str): The client's nickname.

Returns:

- validated_config (tuple): Validated server configuration if successful, otherwise None.

"""

```
validated_config = self.server_config_validator.validate_config(server, port, channel,
nickname)
```

```
if validated_config is not None:
```

```
self.bridge = IRCSocketConnection(*validated_config)
```

```
self.bridge.start()
```

```
self.command_handler = MessageHandler(self)
```

```
self.validated_config = validated_config
```

```
return validated_config
```

```
def disconnect_from_server(self):
```

"""

Disconnect from the IRC server.

"""

```
if self.bridge:
```

```
self.bridge.disconnect()
```

```
def reconnect_to_server(self):
```

```
"""
```

Attempt to reconnect to the IRC server.

```
"""
```

```
if self.bridge:
```

```
    self.bridge.reconnect()
```

```
def send_message(self, message):
```

```
"""
```

Send a message to the IRC server.

Parameters:

- message (str): The message to send.

Returns:

- validated_message (str): Validated message if successful, otherwise None.

```
"""
```

```
    validated_message = InputValidation.validate_input(message)
```

```
    if validated_message is not None and self.bridge:
```

```
        self.message_handler(validated_message)
```



```
self.post_chat_history(validated_message)

return validated_message
```

```
def receive_data(self):
```

```
    """
```

Receive and validate data from the IRC server.

Returns:

- validated_data (str): Data received from the server, or an empty string if no data.

```
    """
```

```
    if self.bridge:
```

```
        data = self.bridge.receive_data()
```

```
        validated_data = OutputValidation.validate_output(data)
```

```
        if data is not None:
```

```
            self.post_chat_history(validated_data)
```

```
        return validated_data
```

```
    return ""
```

```
def message_handler(self, message):
```

```
    """
```

Delegate message handling to the MessageHandler.

Parameters:

- message: The message to be handled.

"""

Choose and execute IRC commands

message_to_send = self.command_handler.command_choice(message)

Send the processed message to the IRC server

if message_to_send:

self.bridge.send_data(message_to_send)

def post_chat_history(self, history_data):

"""

Send chat history to the history server.

"""

self.history_manager.post_history(history_data)

def get_chat_history(self):

"""

Request chat history from the history_server instance.

```

"""

history_data = self.history_manager.get_history()

# Process the received history data as needed

if history_data:

    return history_data

def disconnect_from_history_server(self):

    # Close the connection to the history server

    self.history_manager.close_connection()

```

irc_server_handler.py

```

import socket

import threading

import sys

from abc import ABC, abstractmethod

```

```

MAX_BUFFER_SIZE = 2048

```

```

class IRCCConnection(ABC):

```

```

"""

```

Abstract base class for IRC bridges.

Attributes:

- None

Methods:

- connect(): Connect to the IRC server.
- disconnect(): Disconnect from the IRC server.
- reconnect(): Reconnect to the IRC server.
- send_data(message): Send data to the IRC server.
- receive_data(): Receive data from the IRC server.

"""

@abstractmethod

def connect(self):

pass

@abstractmethod

def disconnect(self):

pass

```
@abstractmethod
def reconnect(self):
    pass
```

```
@abstractmethod
def send_data(self, message):
    pass
```

```
@abstractmethod
def receive_data(self):
    pass
```

```
class IRCSocketConnection(IRCCConnection):
    def __init__(self, server, port, channel, nickname):
        """
```

Concrete implementation of IRCBridge for socket-based IRC connection.

Attributes:

- server (str): IRC server address.
- port (int): IRC server port.
- channel (str): Initial channel to join.
- nickname (str): User's nickname.
- irc_socket (socket.socket): Socket object for IRC communication.

Methods:

- connect(): Connect to the IRC server, send initial USER and NICK commands, and join the specified channel.
- disconnect(): Disconnect from the IRC server.
- reconnect(): Reconnect to the IRC server.
- send_data(message): Send data to the IRC server.
- receive_data(): Receive data from the IRC server.
- start(): Start the IRC client by connecting to the server.

```
"""
```

```
self.server = server
```

```
self.port = port
```

```
self.channel = channel
```

```
self.nickname = nickname
```

```
self.irc_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
def connect(self):
```

```
    """
```

Connect to the IRC server, send initial USER and NICK commands (to register user), and join the specified channel.

Returns:

- True if the connection is successful, False otherwise.

```
    """
```

```
    try:
```

```
        # Establish a connection to the IRC server
```

```
        self.irc_socket.connect((self.server, self.port))
```

```
        # Send initial USER and NICK commands to REGISTER the user
```

```
        self.send_data(f"USER {self.nickname} 0 * :{self.nickname}\r\n")
```

```
        self.send_data(f"NICK {self.nickname}\r\n")
```

```
        # Join the specified channel
```

```
        self.send_data(f"JOIN {self.channel}\r\n")
```

```
    except socket.error as e:
```

```
print(f'Error connecting to the server: {e}')
```

```
return False
```

```
return True
```

```
def disconnect(self):
```

```
    """
```

```
    Disconnect from the IRC server.
```

```
    """
```

```
    try:
```

```
        self.send_data("QUIT :Disconnected\r\n")
```

```
        self.irc_socket.close()
```

```
    except Exception as e:
```

```
        print(f'Error disconnecting from the server: {e}')
```

```
def reconnect(self):
```

```
    """
```

```
    Reconnect to the IRC server.
```

```
    """
```

```
    try:
```

```
        # Reestablish a connection to the IRC server
```



```

self.irc_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Send initial USER and NICK commands to ReREGISTER the user

self.irc_socket.connect((self.server, self.port))

self.send_data(f"USER {self.nickname} 0 * :{self.nickname}\r\n")

self.send_data(f"NICK {self.nickname}\r\n")

self.send_data(f"JOIN {self.channel}\r\n")

except socket.error as e:

print(f"Error reconnecting to the server: {e}")

```

```

def send_data(self, message):
    """
    Send data to the IRC server.

```

Parameters:

- message: The message to be sent.

```

    """

    try:

self.irc_socket.send(bytes(message, 'UTF-8'))

except socket.error as e:

print(f"Error sending data: {e}")

```

```
def receive_data(self):
```

```
    """
```

```
    Receive data from the IRC server.
```

```
    Returns:
```

```
    - The received data.
```

```
    """
```

```
    try:
```

```
        # Receive data from the IRC server
```

```
        data = self.irc_socket.recv(MAX_BUFFER_SIZE).decode('UTF-8')
```

```
        if not data:
```

```
            return "Connection closed by server."
```

```
        if data.startswith("PING"):
```

```
            # Respond to PING messages to keep the connection alive
```

```
            self.send_data("PONG {} \r\n".format(data.split()[1]))
```

```
        elif data:
```

```
            return data
```

```
except socket.error as e:

    print(f'Socket error while receiving data: {e}')

    sys.exit(0)
```

```
except Exception as e:

    print(f'An unexpected error occurred while receiving data: {e}')

    sys.exit(1)
```

```
def start(self):

    """

    Start the IRC client by connecting to the server.

    """

    if not self.connect():

        receive_data_thread = threading.Thread(target=self.receive_data, daemon=True)

        receive_data_thread.start()

    return
```

```
class IRCWebSocketConnection(IRCCConnection):
```

```
"""
```

Concrete implementation of IRCBridge for WebSocket-based IRC connection.

Attributes:

- Implementation for WebSocket connection

Methods:

- connect(): Implementation for WebSocket connection.
- disconnect(): Implementation for WebSocket disconnection.
- reconnect(): Implementation for WebSocket reconnection.
- send_data(message): Implementation for sending data over WebSocket.
- receive_data(): Implementation for receiving data over WebSocket.

```
"""
```

```
def __init__(self, server, port, channel, nickname):
```

```
# Implementation for WebSocket connection
```

```
pass
```

```
def connect(self):
```

```
# Implementation for WebSocket connection
```

```
pass
```

```
def disconnect(self):
```

```
# Implementation for WebSocket disconnection
```

```
pass
```

```
def reconnect(self):
```

```
# Implementation for WebSocket reconnection
```

```
pass
```

```
def send_data(self, message):
```

```
# Implementation for sending data over WebSocket
```

```
pass
```

```
def receive_data(self):
```

```
# Implementation for receiving data over WebSocket
```

```
pass
```

message_handler.py

from Client.Commands.command import Command

from Client.Commands.composite_command import CompositeCommand

from Client.Commands.join_command import JoinChannelCommand

from Client.Commands.leave_command import LeaveChannelCommand

from Client.Commands.list_command import ListUsersCommand

from Client.Commands.info_command import ChannelInfoCommand

from Client.Commands.nick_command import ChangeNicknameCommand

from Client.Commands.help_command import DisplayHelpCommand

from Client.Commands.quit_command import QuitCommand

class MessageHandler:

```
"""
```

Handles the execution of IRC commands based on user input.

```
"""
```

```
def __init__(self, irc_client):
```

```
"""
```

Initialize the MessageHandler class.

Parameters:

- irc_client: The associated IRC client instance.

```
"""
```

```
self.irc_client = irc_client
```

```
self.commands = {
```

```
    "/join": JoinChannelCommand,
```

```
    "/part": LeaveChannelCommand,
```

```
    "/names": ListUsersCommand,
```

```
    "/info": ChannelInfoCommand,
```

```
    "/nick": ChangeNicknameCommand,
```

```
    "/help": DisplayHelpCommand,
```

```
    "/quit": QuitCommand
```

```
}
```

```
def command_choice(self, message):
```

```
    """
```

Execute the IRC command based on the user input.

Parameters:

- message: The user input to be chosen and executed.

Returns:

- The generated IRC message string.

```
    """
```

```
    try:
```

```
        # Check if the message contains multiple commands
```

```
        if ';' in message:
```

```
            composite_command = CompositeCommand()
```

```
            command_strings = message.split(';')
```

```
            for command_string in command_strings:
```

```
                command, *args = command_string.split()
```

```
                command_class = self.commands.get(command.lower())
```

```
                if command_class and issubclass(command_class, Command):
```



```

command_instance = command_class(self.irc_client, *args)

composite_command.add_command(command_instance)

return composite_command.execute()

else:

# Process single command

command, *args = message.split()

command_class = self.commands.get(command.lower())

if command_class and issubclass(command_class, Command):

command_instance = command_class(self.irc_client, *args)

return command_instance.execute()

else:

return f"PRIVMSG {self.irc_client.channel} :{message}\r\n"

except Exception as e:

print(f"Error executing command: {e}")

return ""

```

output_validation.py

```
import datetime
```

```
class OutputValidation:
```

```
def validate_output(server_message):
```

```
"""
```

Validate messages received from the server before displaying them to the user.

Parameters:

- server_message: The message received from the server.

Returns:

- The validated server message or None if the message is invalid.

```
"""
```

```
if "ERROR" in server_message:
```

```
    return None # Discard messages containing "ERROR"
```

```
# Add a timestamp to the received data for display
```

```
if server_message is not None:
```

```
    timestamp = datetime.datetime.now().strftime("[%Y-%m-%d %H:%M:%S]")
```

```
    server_message = f"{timestamp} {server_message}"
```

```
# If no validation conditions are met, timestamp + the original message is returned
```

```
return server_message
```

```
server_config_validation.py
```

```
import socket
```

```
from abc import ABC, abstractmethod
```

```
class ConfigValidation(ABC):
```

```
    """
```

```
    Abstract base class for server configuration validation.
```

```
    """
```

```
    @abstractmethod
```

```
    def validate_config(self, server, port, channel, nickname):
```

```
        pass
```

```
class BasicConfigValidation(ConfigValidation):
```

```
    """
```

```
    Basic implementation of server configuration validation.
```

```
    """
```

```
def is_valid_ip(self, ip):
```

```
    """
```

Check if the given string is a valid IP address.

Parameters:

- ip: The string to check.

Returns:

- True if the string is a valid IP address, False otherwise.

```
    """
```

```
    try:
```

```
        socket.inet_pton(socket.AF_INET, ip)
```

```
        return True
```

```
    except socket.error:
```

```
        try:
```

```
            socket.inet_pton(socket.AF_INET6, ip)
```

```
            return True
```

```
        except socket.error:
```

```
            return False
```

```
def is_valid_domain(self, domain):
```

```
    """
```

Check if the given string is a valid domain name.

Parameters:

- domain: The string to check.

Returns:

- True if the string is a valid domain name, False otherwise.

```
    """
```

```
    try:
```

```
        socket.gethostbyname(domain)
```

```
        return True
```

```
    except socket.error:
```

```
        return False
```

```
def validate_config(self, server, port, channel, nickname):
```

```
    """
```

Validate the server configuration before using it in the IRC client.

Parameters:

- server: IRC server address
- port: IRC server port
- channel: Initial channel to join
- nickname: User's nickname

Returns:

- A tuple containing validated server address, port, channel, and nickname.

Returns None if the configuration is invalid.

```
"""
```

```
# Check if the server is a valid IP or domain
```

```
if not (self.is_valid_ip(server) or self.is_valid_domain(server)):
```

```
    return None
```

```
# Check if the channel starts with #
```

```
if not channel.startswith("#"):
```

```
    channel = "#" + channel
```

```
# Check if the port is an integer
```

```
try:
```

```
    port = int(port)
```

```
except ValueError:
```

```
    return None
```

```
# Check if the port is within a valid range
```

```
if not (0 < port < 65535):
```

```
    return None
```

```
# If no validation conditions are met, return the validated configuration
```

```
return server, port, channel, nickname
```

```
class AdvancedConfigValidation(ConfigValidation):
```

```
    """
```

```
    Advanced implementation of server configuration validation.
```

```
    """
```

```
def validate_config(self, server, port, channel, nickname):
```

```
    """
```

Validate the server configuration using advanced logic.

Parameters:

- server: IRC server address
- port: IRC server port
- channel: Initial channel to join
- nickname: User's nickname

Returns:

- A tuple containing validated server address, port, channel, and nickname.

Returns None if the configuration is invalid.

```
    """
```

```
    # Advanced validation logic
```

```
    if server and port and channel and nickname and len(nickname) > 3:
```

```
        return server, port, channel, nickname
```

```
    else:
```

```
        return None
```



```
history_server.py
```

```
import socket
```

```
import threading
```

```
class ServerConfig:
```

```
"""
```

```
Configuration class for the Server.
```

```
Attributes:
```

- address (str): The IP address to bind the server to.
- port (int): The port number to bind the server to.
- max_buffer_size (int): The maximum buffer size for receiving data.

```
"""
```

```
def __init__(self, address, port, max_buffer_size):
```

```
    self.address = address
```

```
    self.port = port
```

```
self.max_buffer_size = max_buffer_size
```

```
class Server:
```

```
    """
```

Simple server class that handles communication with a single client.

```
    """
```

```
    def __init__(self, config):
```

```
        """
```

Initialize the server with the provided configuration.

Parameters:

- config (ServerConfig): The configuration for the server.

```
        """
```

```
        self.config = config
```

```
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
        self.server_socket.bind((config.address, config.port))
```

```
        self.server_socket.listen(1)
```

```
        print(f"Server listening on {config.address}:{config.port}")
```

```
self.client_socket = None
```

```
self.chat_history = []
```

```
def start(self):
```

```
    """
```

```
    Start the server and continuously accept incoming client connections.
```

```
    """
```

```
    try:
```

```
        while True:
```

```
            self.client_socket, client_address = self.server_socket.accept()
```

```
            print(f"Accepted connection from {client_address}")
```

```
            # Start a thread to handle the client
```

```
            client_handler_thread = threading.Thread(target=self.handle_client)
```

```
            client_handler_thread.start()
```

```
        except Exception as e:
```

```
            print(f"Server error: {e}")
```

```

def handle_client(self):
    """
    Handle communication with a connected client.
    Receive data from the client, update the chat history,
    and send back the updated history to the client.
    """
    try:
        while True:
            data = self.client_socket.recv(self.config.max_buffer_size).decode('utf-8')
            if not data:
                break

            # Store received data in chat_history
            self.chat_history.append(data)

            # Send back the updated chat_history to the client
            history_data = '\n'.join(self.chat_history)
            self.client_socket.sendall(history_data.encode('utf-8'))

```

```
except Exception as e:
```

```
print(f"Error handling client: {e}")
```

```
finally:
```

```
# Close the client socket when the client disconnects
```

```
self.client_socket.close()
```

```
print("Client disconnected")
```

```
main_script.py
```

```
from multiprocessing import Process
```

```
from Client.gui import GUI
```

```
from Client.irc_client import IRCCClient
```

```
from Client.server_config_validation import BasicConfigValidation,  
AdvancedConfigValidation
```

```
from Server.history_sever import Server, ServerConfig
```

```
def run_client():  
    irc_client = IRCClient()  
    irc_client.set_server_config_validator(BasicConfigValidation())  
    gui = GUI(irc_client)  
    gui.start()
```

```
def run_server():  
    server_config = ServerConfig("127.0.0.1", 12340, 2048)  
    server = Server(server_config)  
    server.start()
```

```
if __name__ == "__main__":  
    # Create separate processes for client and server  
    client_process = Process(target=run_client)  
    server_process = Process(target=run_server)  
  
    # Start both processes  
    client_process.start()  
    server_process.start()  
  
    # Wait for both processes to finish
```

```
client_process.join()
```

```
server_process.join()
```