PROJECT

Software Engineering Department

SOFTWARE CONSTRUCTION & DEVELOPMENT

Submitted by

Syeda Aqsa Sohail| 64947

Lab Instructor

MAAM LAILA

Session Fall – 2023

BALOCHISTAN UNIVERSITY OF INFORMATION TECHNOLOGY, ENGINEERING AND MANAGEMENT SCIENCES, QUETTA.

1. Introduction:

1.1 Project Description

This project is a **Simple Chat / Messaging Simulator** developed for desktop platforms. The goal of the project is to demonstrate the practical application of **Software Design Patterns** in a GUI-based messaging system.

The system allows a user to type, send, and view messages inside a chat window. It simulates incoming messages, applies message styling (decorators), and manages chat logs using a centralized engine (singleton).

The main patterns implemented are:

- **Factory Method** → To create different message types (text, emoji, system message)
- **Builder** → For constructing chat sessions
- **Decorator** → To apply message styling (e.g., bold text, emoji-style messages)
- **Observer** → For notifications and incoming messages
- **Singleton** → For Chat-Engine that manages chat logs and message routing

2. Design Pattern Usage:

2.1 ***Factory Method Pattern***

Problem It Solves:
The system must create different types of messages (text, emoji) without repeating code everywhere.

How Implemented:
A MessageFactory class creates messages using a method like: create_message(type, content)
Based on the type, it returns the correct message object.

Class-Level Explanation:

- Message → Base class
- TextMessage, EmojiMessage → Subclasses
- MessageFactory → Chooses and returns the correct message type

UML (Simple Text):

```
MessageFactory → Message
              ├── TextMessage
              └── EmojiMessage
```

2.2 Builder Pattern

Problem It Solves:
A chat session has many parts (sender, receiver, theme). Creating it in one constructor becomes confusing.

How Implemented:
ChatSessionBuilder builds the chat session step-by-step.

Class-Level Explanation:

- ChatSession → Final chat session
- ChatSessionBuilder → Sets sender, receiver, theme
- build() → Returns the ready ChatSession

2.3 Decorator Pattern

Problem It Solves:
Messages need different styles (bold, italic, emoji), but creating a new class for every combination would be messy.

How Implemented:
Decorators wrap the base message and add styling dynamically.

Class-Level Explanation:

- MessageDecorator → Base decorator
- BoldDecorator, ItalicDecorator, EmojiDecorator → Add style to messages
  - show() → Returns the message with applied decoration.

2.4 Observer Pattern

Problem It Solves:
The chat window should update automatically when new messages arrive.

How Implemented:
Observers watch the ChatEngine.
When a new message comes, all observers are notified.

Class-Level Explanation:

- Observer → Interface
- ChatWindow → Observer
- ChatEngine → Subject that notifies observers

2.5 Singleton Pattern

Problem It Solves:
Only one engine should control chat logs, routing, and notifications.

How Implemented:
ChatEngine has one instance using get_instance().

Class-Level Explanation:

- Only one ChatEngine object exists
- Manages all chat data and processes
- Avoids duplication and confusion

3. Implementation Details

Technologies Used

Programming Language:

- Python

GUI Framework:

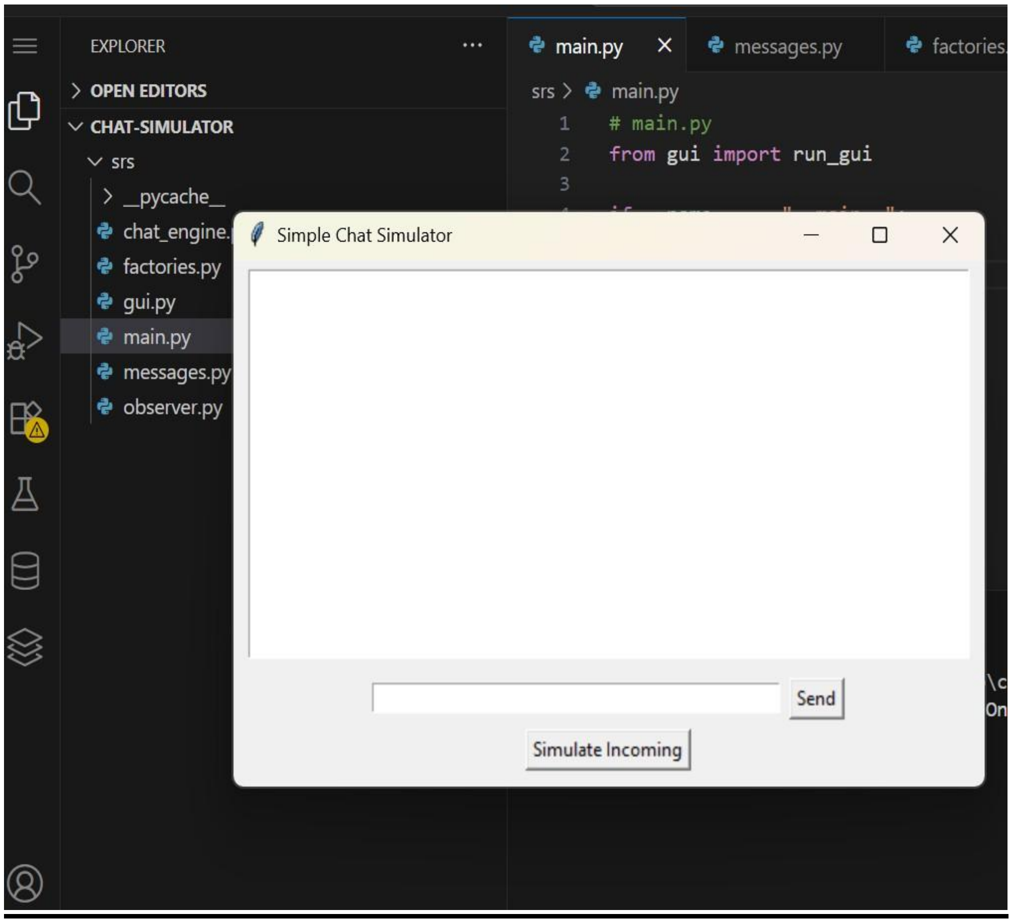
- Tkinter

Libraries Used:

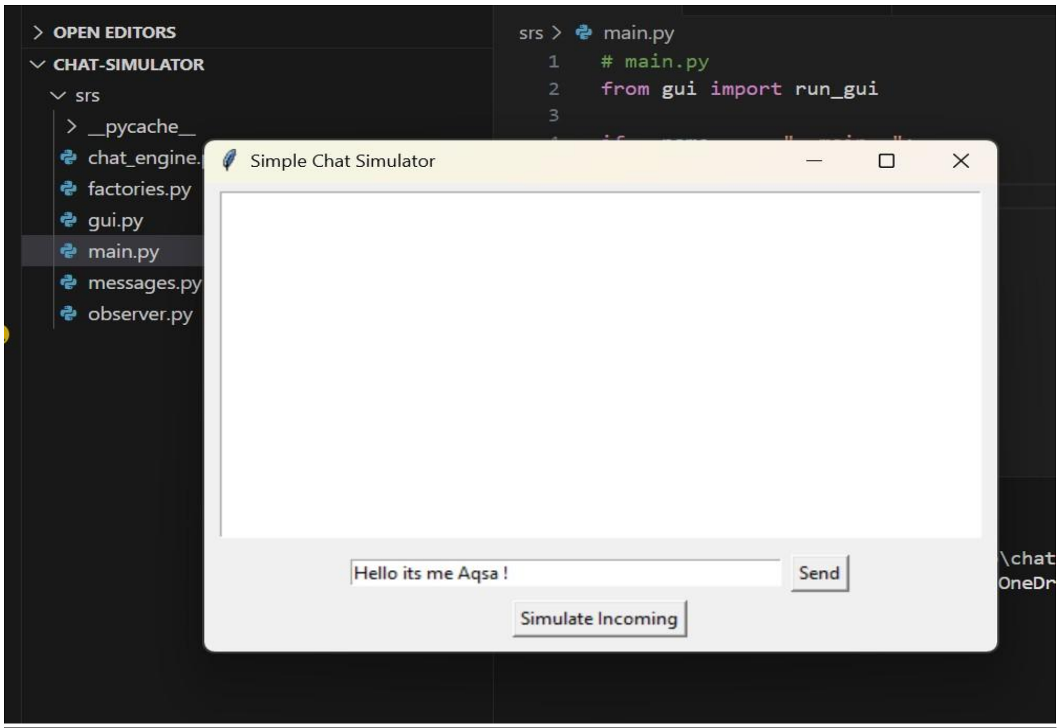
- **tkinter** – used to build the chat window and buttons
- **datetime** – used for simple time functions
- **threading** – used to simulate incoming messages

- **time** – used for delays during message simulation
- **abc** – used to create abstract classes
- **random** – used for simple random incoming message text (optional).

4. Screenshots / Output:

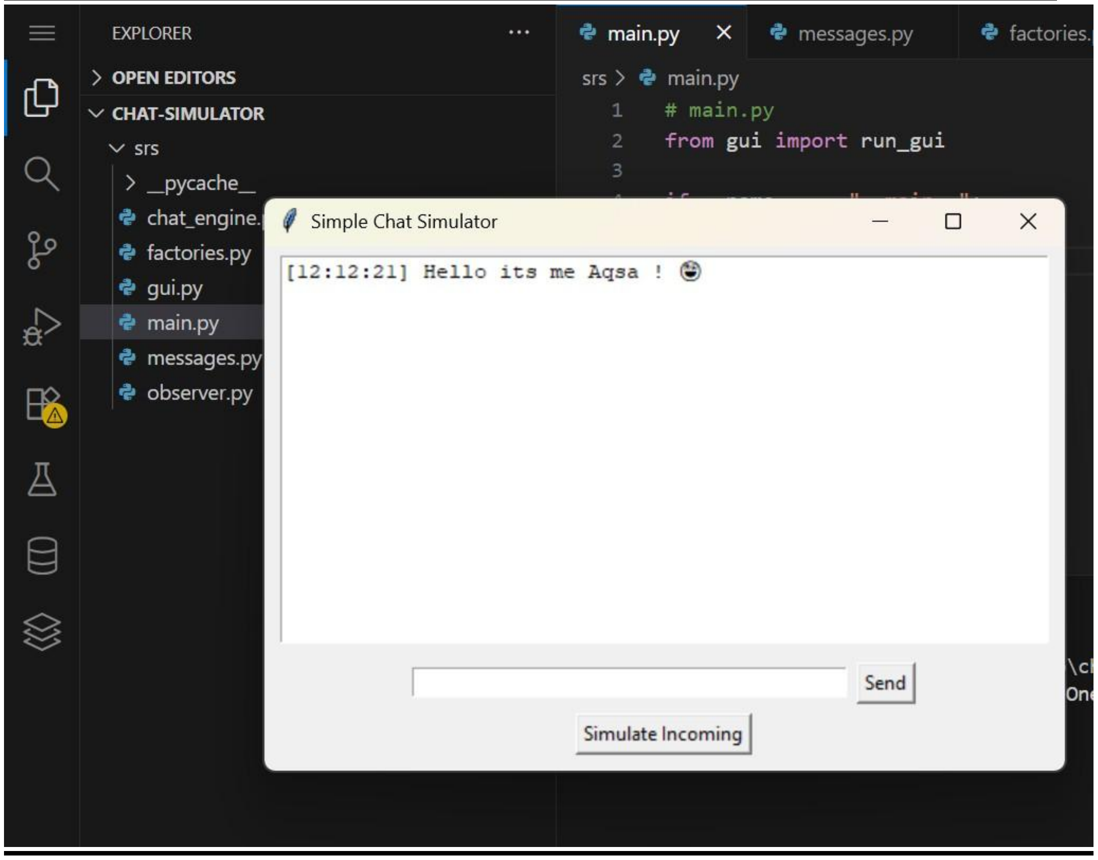


```
# main.py
from gui import run_gui
```



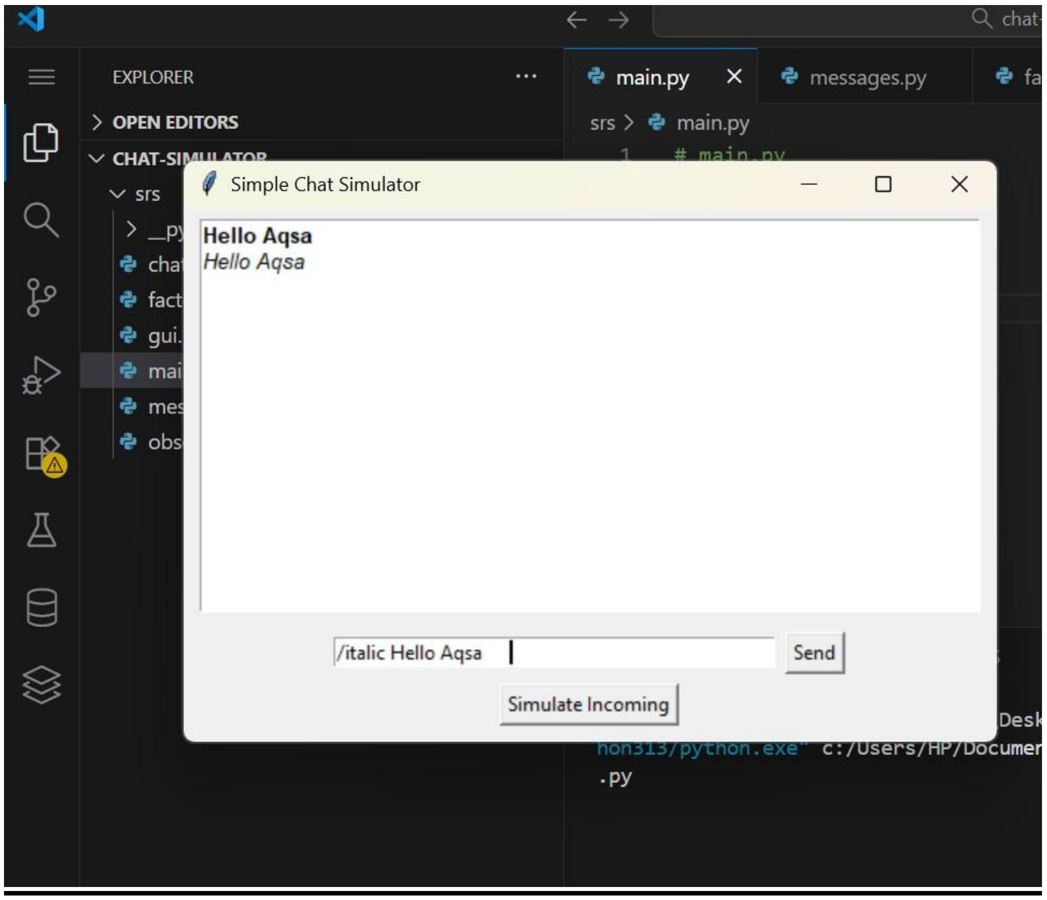
```
# main.py
from gui import run_gui
```

Hello its me Aqsa !



```
# main.py
from gui import run_gui
```

```
[12:12:21] Hello its me Aqsa ! 😀
```

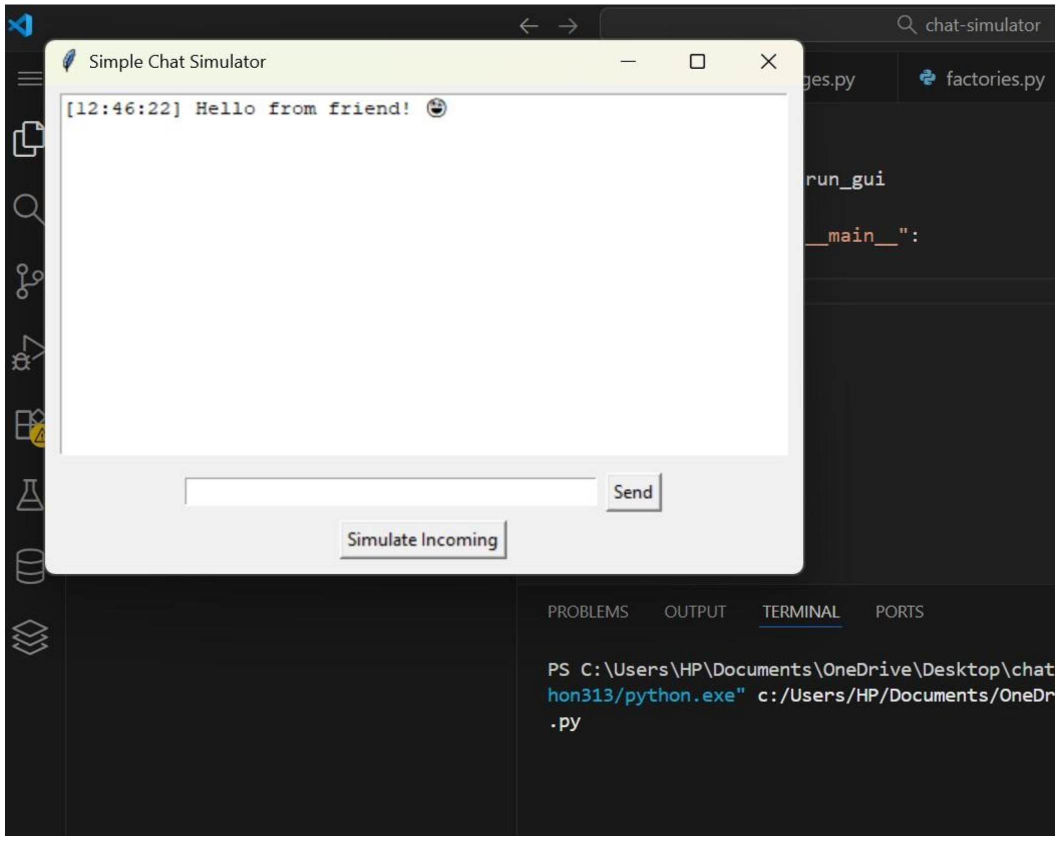


Hello Aqsa

Hello Aqsa

```
/italic Hello Aqsa
```

```
hon313/python.exe" c:/Users/HP/Documen
.py
```



```
[12:46:22] Hello from friend! 😀
```

```
PS C:\Users\HP\Documents\OneDrive\Desktop\chat
hon313/python.exe" c:/Users/HP/Documents/OneDr
.py
```

Problems Faced and Solutions

Problem 1: The GUI was freezing because incoming messages made the window stop responding.
Solution: I used the after() function so the updates happen smoothly without freezing.

Problem 2: It was difficult to handle different message types like text and emoji because each needed separate code.
Solution: I used the Factory Pattern to automatically create the correct message type.

Problem 3: Adding styles like bold and italic became messy when written directly inside the main code.
Solution: I used the Decorator Pattern to apply styles in a cleaner way.

Problem 4: Managing chat logs was hard because many parts of the program needed access to the same data.
Solution: I used the Singleton Pattern so only one engine stores and manages all the messages.

Conclusion:

This project successfully demonstrates how major software design patterns can be applied in a real GUI-based chat simulator. By using Factory, Builder, Decorator, Observer, and Singleton patterns, the system becomes easier to extend, maintain, and understand. The GUI works smoothly, messages are organized properly, and styling features are added cleanly. Overall, the project shows how structured design patterns improve both functionality and code quality.