



UniMe
1548

Object Oriented Programming

Project Report

Project Title: Telegram Messenger

Professor: Salvatore Distefano

Student: Arman Khademi – 556026

Semester: Spring 2025

Table of Contents

Project Introduction and User Interaction	3
Startup Flow.....	3
Main Menu Options	3
Chat Interaction Flow	5
Role-Based Access	5
User Plans	5
Message Types	6
Additional Features	6
Project Overview :	7
"models" – Core Data Models	7
"manager" – Application Logic and Use Case Handling.....	7
"app" – User Interface (Terminal Menus)	8
"exception" – Custom Exceptions.....	8
"util" – Utility Classes	8
OOP Concepts in more Detail with Code Snippets	10
Inheritance.....	10
Encapsulation.....	11
Information hiding	12
Polymorphism	13
Abstraction	18
Interface Implementation:	18
Composition.....	20
Subtyping	20
Exception Handling	21
Extensibility:	23

Project Introduction and User Interaction

The goal of this project is to create a simple and interactive messenger that runs in the terminal. It works like Telegram, but it's fully built in Java, without using any graphical interface or database.

The project was developed to follow key object-oriented programming principles, including inheritance, abstraction, encapsulation, polymorphism, and modular design.

The application allows users to create an account, log in, send and receive messages, create group chats or channels, manage permissions, and publish posts just like social media platforms (inspired by Instagram). All features are text-based and designed to work within a single computer environment (no networking).

Startup Flow

When the program launches, it immediately clears the terminal screen (via a utility method) and presents a **Welcome Menu** with two primary options:

1. **Register**
2. **Login**

Users who choose to register are prompted to enter a **username**, **nickname**, **phone number**, and a **password**. Duplicate usernames are not allowed and are handled using a custom exception.

After successful registration or login, users are redirected to the **Main Menu**, which contains the four key functional sections of the messenger.

Main Menu Options

1. View Chats

Displays a list of all chats (private, group, or channel) the user is currently a member of. Users can select a chat by its ID and enter it to read and send messages.

2. Create New Chat

Allows the user to create a new chat. The app asks for the type of chat:

- ❖ **Private Chat (PV)**: between two users
- ❖ **Group**: multi-user chat with admins
- ❖ **Channel**: broadcast-only platform where only admins can post

Username of participants are entered, and roles are assigned. The creator of a group or channel becomes the **owner**.

3. Instagram Section

Simulates a social media experience. Users can enter a username to view that person's:

- ❖ Profile information (nickname, username, phone)
- ❖ Posts (image path + caption + Like counts)

Each post is a combination of a **caption** and a **file path** representing the image. Since this is a terminal project, the image is not shown, only the filename is displayed. Users can **like a post**.

4. My Profile

Displays the current user's:

- ❖ Username, nickname, phone number
- ❖ Biography (bio)
- ❖ List of posts

Users can:

- ❖ Add a new post (caption + image path)
- ❖ Remove a post from their profile
- ❖ Edit their personal Info
- ❖ Upgrade their UserPlan (Basic , Silver , Gold)

Chat Interaction Flow

Inside a chat session, users see the full message history (in order), and depending on their **role**, they can:

- ❖ Send new messages (text or media)
- ❖ Edit their own messages (text only)
- ❖ Delete their own messages.
- ❖ If they are admins/owners: edit or delete others' messages.
- ❖ See Chat Details

The system dynamically checks whether the user has permission to perform each action.

Role-Based Access

Each user has a role inside a chat:

- ❖ **NormalUser**: can send/edit/delete their own messages
- ❖ **Admin**: can manage members and control messages
- ❖ **Owner**: the creator of the chat; has all permissions
- ❖ **Viewer** (channels): can only read messages

Roles are implemented using the **Role interface**, and permissions are checked dynamically during interaction.

User Plans

Each user has a **User Plan**:

- ❖ **Basic** : can post 10 images , only can create private chat
- ❖ **Silver** : can post 20 images , can create group chat also
- ❖ **Gold** : can post 30 images . can create channel too

Each plan defines how many posts a user can publish and can they create group or channel or not .

Message Types

There are 3 types of **messages** in chats :

- ❖ **TextMessage**: contain Text messages
- ❖ **MultiMediaMessage**: show by path of that file
- ❖ **InfoMessage**: some messages related to add member , create a chat or ...

Roles are implemented using the **Role interface**, and permissions are checked dynamically during interaction.

Additional Features

Each screen clears the console before displaying new content for a clean UI

Message timestamps are formatted for readability (e.g. "2025-04-28 21:03:22").

```
○ Arman - Mahya
-----
[INFO] armannella Created This Chat (2025-05-03 20:42:11)
-2 | Arman : Hi there (2025-05-03 20:42:22)
-3 | Arman : its me Arman (2025-05-03 20:42:31)
-4 | Mahya : Hello Arman (2025-05-03 20:43:04)
-5 | Mahya : whats up ?? (2025-05-03 20:43:13) (Edited)
-----
Options:
1. Send Text Message
2. Send Media Message
3. Edit Message
4. Delete Message
5. Chat Details
0. Back
-----
Choose an option: █
```

An Example
from a chat
menu

Project Overview :

The Telegram Messenger project is designed following a layered architecture using clean OOP principles. The goal is to keep the logic modular, maintainable, and easy to scale or extend in future versions. The codebase is divided into several clearly named packages, each with its own responsibility:

"models" – Core Data Models

This package contains all the essential classes that represent real-world entities in the system.

- ❖ **User**: Represents a registered user, with fields like username, password, phone number, and an associated **Profile** and **UserPlan**.
- ❖ **Profile**: Manages the user's posts and bio.
- ❖ **Post**: Contains a caption, image path, and timestamp.
- ❖ **UserPlan (abstract)**: subclasses include **BasicPlan**, **SilverPlan**, and **GoldPlan**.
- ❖ Messaging system:
- ❖ **Message (abstract)**: Superclass for **TextMessage**, **MultimediaMessage**, and **InfoMessage**.
- ❖ **Chat (abstract)**: Parent class for **PV (private)**, **Group**, and **Channel**.
- ❖ **ChatMember**: Represents a user's membership and role in a specific chat.
- ❖ **Role (interface)**: Implemented by **NormalUser**, **Admin**, **Owner**, **Viewer**.

"manager" – Application Logic and Use Case Handling

All core **logic**, **user validation**, **permission control**, and chat/message **operations** are handled here.

- ❖ **UserManager**: Handles registration, login, user lookup.
- ❖ **ChatManager**: Responsible for chat creation, user addition/removal, role assignment.
- ❖ **MessageManager**: Manages sending, editing, deleting, and displaying messages.
- ❖ **PostManager**: Handles post creation, deletion, and enforcement of post limits.

These classes serve as a controller layer between the raw data (models) and the terminal interface.

"app" – User Interface (Terminal Menus)

Handles user interaction via command-line menus and collects input from the user. Each menu represents a specific screen or feature:

- ❖ **WelcomeMenu**: Entry point for Register/Login.
 - ❖ **MainMenu**: Main dashboard with options for chat, profile, posts.
 - ❖ **ChatMenu / ChatSessionMenu**: Displays chat list and chat interaction view.
 - ❖ **ProfileMenu**: User's own profile and post management.
 - ❖ **InstagramMenu**: Allows searching for and viewing another user's profile.
 - ❖ **ChatDetails / AccountDetails**: Views and manages chat/user-specific info.
-

"exception" – Custom Exceptions

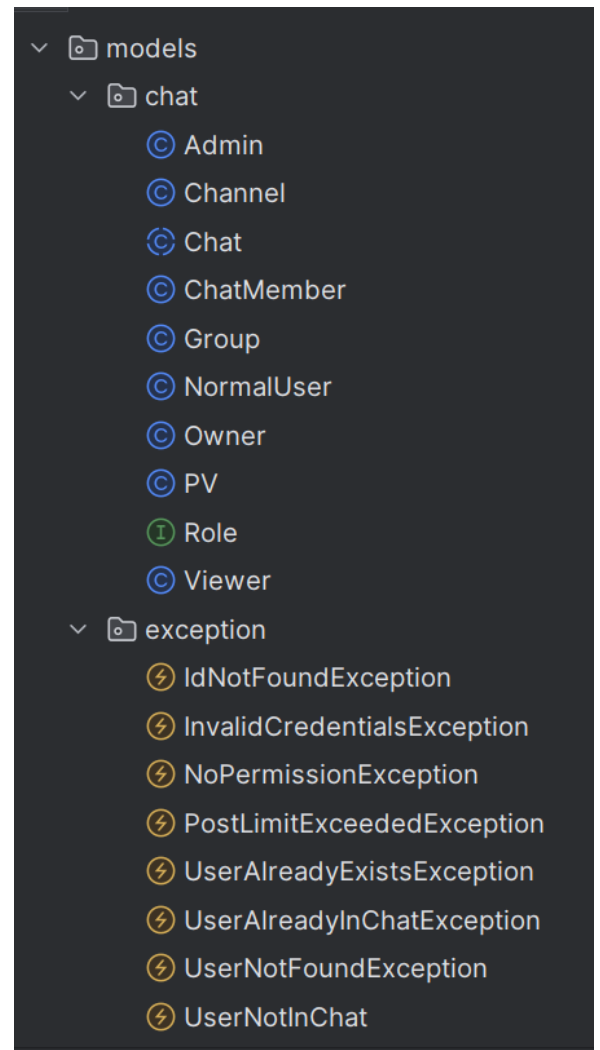
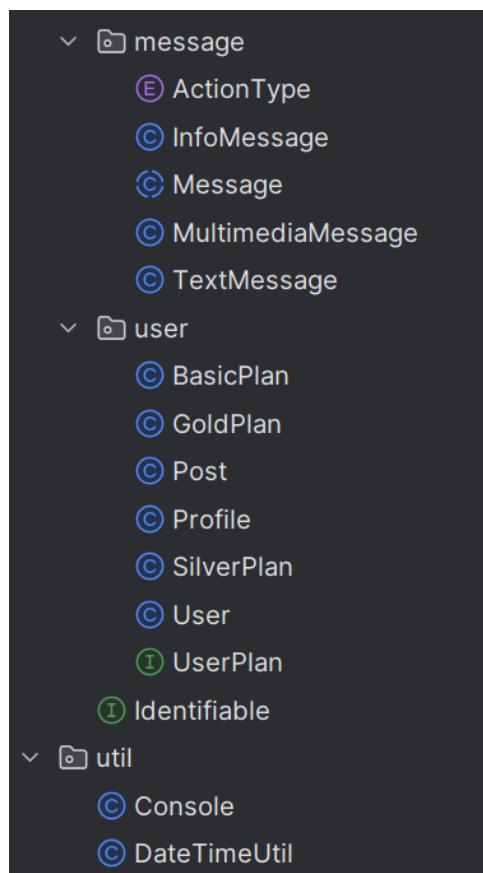
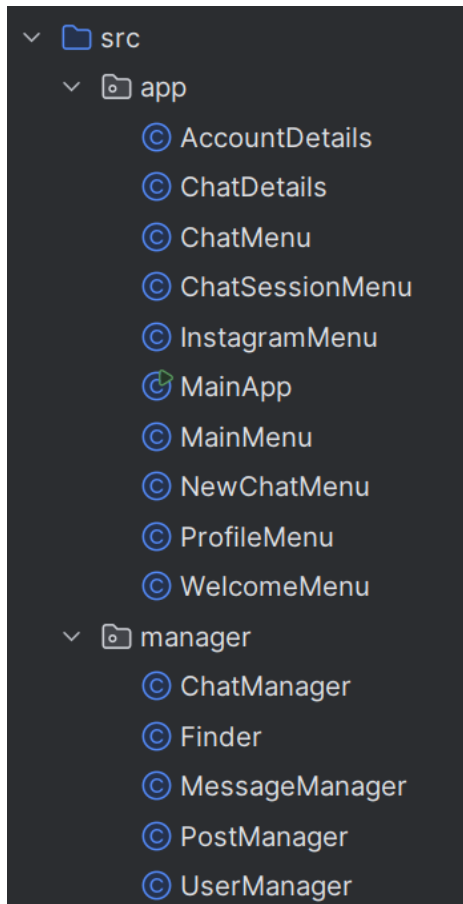
Instead of basic conditionals, the program uses custom exceptions to handle errors more cleanly:

- ❖ **UserNotFoundException**
- ❖ **InvalidCredentialsException**
- ❖ **UserAlreadyExistsException**
- ❖ **NoPermissionException**
- ❖ ...

This keeps logic modular and separates concerns.

"util" – Utility Classes

Small helper classes used throughout the system for repetitive tasks.



OOP Concepts in more Detail with Code Snippets

In this below I mentioned some OOP concepts with some examples. Inside the project is used way more OOP concepts then here in the report.

Inheritance

Inheritance is a way to create a new class (called a subclass or child class) by using the features of an existing class (called a superclass or parent class). It helps you reuse code and build a structure where classes are related to each other.

For example in my code we have **"Chat"** class and inherit from it to create **"PV"** and **"Group"** and **"Channel"** classes. Or also you can see in the below picture for Class **"Message"**.

```
public abstract class Chat implements Identifiable {}  
public class PV extends Chat{}  
public class Group extends Chat{}  
  
public abstract class Message implements Identifiable{}  
public class TextMessage extends Message {}  
public class MultimediaMessage extends Message {}  
public class InfoMessage extends Message{}  
|
```

Encapsulation

Encapsulation is a form of data abstraction that enables the creation of Abstract Data Types (ADTs) by **bundling together** both the **attributes** (i.e., the structure or **state** of an object) and the **methods** that operate on those attributes (i.e., the **behavior**). Essentially, an ADT is composed of two main parts: its **state** and its **behavior**.

In my project, each class is designed as a self-contained ADT. For example, the **"User" class** encapsulates the **username, password, user plan, and profile information**, along with **behaviors** such as **credential validation and profile access**. That you can see full details in the below picture.

By encapsulating related data and functionality within the same class, the system design becomes more robust, modular, and easier to maintain. This approach also significantly reduces the cost and complexity of fixing bugs.

```
public class User {  
    private String username ;  
    private String password ;  
    private String nickname ;  
    private String phonenumber ;  
    private Profile profile ;  
    private UserPlan userplan ;  
  
    public User (String username , String password , String nickname , String phonenumber) { ...  
  
    public boolean checkPassword(String password) {  
        return this.getPassword().equals(password) ;  
    }  
  
    public String getUsername() {  
        return username ;  
    }  
  
    public void setUsername(String username) {  
        this.username = username ;  
    }  
  
    private String getPassword() {  
        return password ;  
    }  
}
```

States

Methods
(behavior)

Information hiding

Information hiding helps separate the **interface** and **implementation** in an ADT. An ADT has **two parts**: **public or external**, which includes the conceptual architecture (what is a specific ADT) and conceptual operations (what can be done with ADT), and **private or internal**, which includes data representation and implementation of operations. we always have implementation hiding, and we may have information hiding .

In this project, information hiding effectively applied, especially in the **"User"** class. All the class attributes such as **username**, **password**, **nickname**, and **phoneNumber** are declared as **private**, which is an example of information hiding — these fields are not directly accessible from outside the class.

To allow controlled access, the class provides **public getter and setter methods**,. This way, other classes can interact with user data safely, without breaking internal structure or security.

Or for example Fields like **password** are completely hidden; there is no public **getPassword()** method. Instead, only a method like **checkPassword(String input)** is exposed, which simply returns **true/false**. The user of the class doesn't need to (and shouldn't) know how the password is stored and validated.

```
3 public class User {
4     private String username ;
5     private String password ;
6     private String nickname ;
7     private String phonenumber ;
8     private Profile profile ;
9     private UserPlan userplan ;
10
11
12 > public User (String username , String password , String nickname , String phonenumber) { ...
13
14
15
16
17
18
19
20
21
22
23 > public boolean checkPassword(String password) {
24     return this.getPassword().equals(password) ;
25 }
26
27 > public String getUsername() {
28     return username;
29 }
30
31 > public void setUsername(String username) {
32     this.username = username;
33 }
34
35 > private String getPassword() {
36     return password;
37 }
```

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables flexibility and extensibility in code. some common forms of polymorphism are (and how I used them in my project) :

- * Method Overloading:

This occurs when multiple methods in a class have the same name but different parameter lists. In my Project, this can be seen in the overloaded constructors of **"InfoMessage"** class, such as:

```
public InfoMessage(User user1 , ActionType action , User user2)
{
    super(user1);
    Info = user1.getUsername() + " " + action.getText() + " " + user2.getUsername() + " (" + DateTimeUtil
}

public InfoMessage(User user , ActionType action)
{
    super(user);
    Info = user.getUsername() + " " + action.getText()+ " (" + DateTimeUtil.format(timestamp)+ ")";
}
```

We have 2 kind of constructor because some times info message is related to 2 user and an action (like user1 added user2 to chat)
sometimes we have 1 user and an action (user1 changed group title)

-** Polymorphism by Inclusion:

Polymorphism by inclusion allows using a superclass or interface reference to refer to objects of its subclasses. This enables runtime polymorphism where the actual method implementation called is determined by the runtime type of the object. In my project, I implemented this concept using abstract classes like **Message** and **Chat**, and an interface **Role**.

1. Base Abstract Class: Message

The abstract class **"Message"** defines a method called **display()**, which is overridden by its subclasses. This sets the foundation for polymorphism by inclusion:

```
public abstract class Message implements Identifiable{
    protected User sender ;
    protected LocalDateTime timestamp ;
    protected static int counter = 1 ;
    protected int id ;

    public Message(User sender) ...

    public abstract void display();
}
```

2. Derived Classes: TextMessage, MultimediaMessage, InfoMessage

Each subclass overrides the display method differently. Below is an example from TextMessage:

```

@Override
public void display()
{
    String info = "-" + this.getID() + " | " + this.getSender().getNickname() + " : " + this.content + "
    if(edited)
    {
        info += " (Edited)";
    }
    System.out.println(info);
}

```

3. Usage Example in MessageManager :

Inside the **"MessageManager"** class, we iterate over a list of Message objects, and although each object may be a TextMessage, MultimediaMessage, or InfoMessage, we treat them as **Message** and still achieve correct behavior:

```

public void displayChatMessages(Chat chat)
{
    for (Message msg : chat.getMessages()) {
        msg.display();
    }
}

```

This is a clear example of runtime polymorphism. Even though msg is declared as Message, the overridden **display()** method of the actual subclass is executed at runtime.

4. Second Example: Role Interface

The Role interface defines permission-checking methods, and multiple implementations (Admin, Owner, Viewer, NormalUser) provide their own logic:


```

public interface Role {
    boolean canSendMessage();
    boolean canEditMessage(Message message, User editor);
    boolean canDeleteMessage(Message message, User deleter);
    boolean canAddMember();
    boolean canDeleteMember();
    boolean canManageAdmins();
    boolean canChangeTopic();
    String getRoleName();
}

```

Each class implements these methods differently. In **ChatManager** or **MessageManager**, we refer to role as the interface type:

```

public void sendMessage(Chat chat, Message message) throws NoPermissionException
{
    ChatMember sender = getChatMember(chat, message.getSender()) ;
    if (!sender.getRole().canSendMessage()) 
    {
        throw new NoPermissionException(action:"send Message");
    }
    chat.addMessage(message);
}

```

The actual behavior depends on whether the role is Admin, Viewer, etc., demonstrating polymorphism by inclusion.

*** Parametric Polymorphism (Generics):

Generics allow methods or classes to operate on different types while maintaining type safety. It is useful when writing reusable utility classes or methods.

I used a generic method in the project **"Finder"** class:

```
public class Finder<T extends Identifiable > {  
  
    public T findById(ArrayList<T> list , int id) throws IdNotFoundException  
    {  
        for (T item : list)  
        {  
            if(item.getID() == id)  
            {  
                return item ;  
            }  
        }  
        throw new IdNotFoundException(id);  
    }  
}
```

Classes like **Message, Chat, or Post** can implement the **Identifiable** interface, and this method helps us retrieve items from a list based on ID without duplicating code.

Abstraction

Abstraction is the process of simplifying complex reality by modeling classes based on their essential properties and behaviors. It hides internal complexity and allows users to interact with high-level interfaces.

In my project, abstraction is applied via abstract classes such as **"Message"**, **"Chat"**.

```
public abstract class Chat implements Identifiable{  
    public abstract String getType();  
    public abstract Role getDefaultJoinRole();  
}
```

Each abstract class defines common behavior, which is then concretely implemented in subclasses like **"PV , Group, Channel"**

Interface Implementation:

An interface defines a contract of behaviors that a class must implement. In simpler terms, it specifies a set of methods (and constants) that a class must provide, but it doesn't provide the implementation details

In my project I implemented 3 interfaces.

- ❖ **UserPlan** → BasicPlan , SilverPlan , GoldPlan
- ❖ **Role** → Viewer , NormalUser , Admin , Owner
- ❖ **Identifiable** → Message , Chat , Post

for example for user plans :

```
public interface UserPlan {  
  
    int getMaxPosts();  
    boolean canCreateChannel();  
    boolean canCreateGroup();  
    String getPlanName();  
}
```

```
public class SilverPlan implements UserPlan{  
  
    @Override  
    public int getMaxPosts() {  
        return 20 ;  
    }  
  
    @Override  
    public boolean canCreateChannel(){  
        return false;  
    }  
  
    @Override  
    public boolean canCreateGroup(){  
        return true ;  
    }  
  
    @Override  
    public String getPlanName() {  
        return "Silver" ;  
    }  
}
```

Composition

Composition is a design principle where one class includes references to other classes as fields. This project uses composition to create complex systems by combining simpler classes.

Examples include:

- A **User** "has-a" **Profile**
- A **Profile** "has-a" **ArrayList<Post>**
- A **Chat** "has-a" **ArrayList<ChatMember>**
- A **ChatMember** "has-a" **Role** and **User**

This approach allows for better code reuse and modular design. For instance, the **Post** class is not defined inside **Profile**, but Profile holds a **List<Post>** which it manages.

Subtyping

Subtyping is a key OOP concept that allows an object of a derived class to be treated as an instance of its superclass or interface.

In my project, this is achieved through both class inheritance and interface implementation:

-Subtyping through Inheritance:

```
public abstract class Chat implements Identifiable{  
public class Channel extends Chat{  
public class Group extends Chat{
```

-Subtyping through Interfaces:

```
public interface Role {  
public class Admin implements Role{  
public class Viewer implements Role{
```

Exception Handling

Exception handling is a critical part of any robust software system. In our Telegram Terminal project, we incorporated two levels of exception handling:

System-Level Exceptions:

"InputMismatchException" :

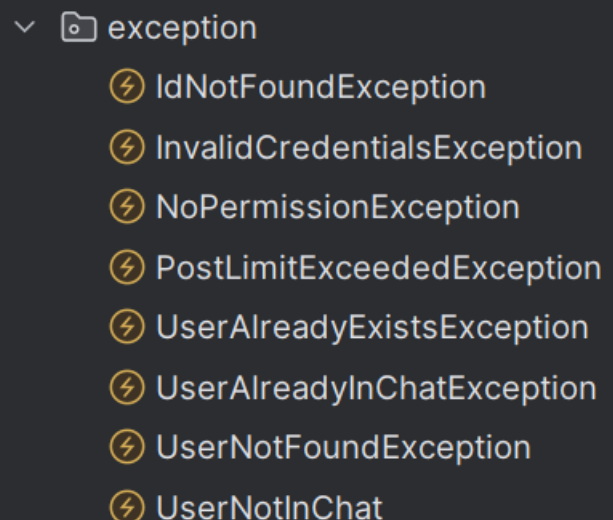
When the user is expected to enter an integer but types a string, the program catches this using a try-catch block:

```
while (true) {  
    try {  
        int input = scanner.nextInt();  
        scanner.nextLine();  
        return input;  
    } catch (InputMismatchException e) {  
        System.out.println(x:"Invalid input. Please enter a number.");  
        scanner.nextLine();  
    }  
}
```

Custom exceptions

help model the business logic of the application more cleanly and precisely. We've defined multiple domain-specific exceptions, including:

These are thrown and handled throughout the application to enforce rules clearly and separate responsibilities.



```
exception  
  ⚡ IdNotFoundException  
  ⚡ InvalidCredentialsException  
  ⚡ NoPermissionException  
  ⚡ PostLimitExceededException  
  ⚡ UserAlreadyExistsException  
  ⚡ UserAlreadyInChatException  
  ⚡ UserNotFoundException  
  ⚡ UserNotInChat
```

```

public class NoPermissionException extends Exception {
    public NoPermissionException(String action)
    {
        super("You dont have permission to " + action);
    }
}

```

```

public void addMember(Chat chat, User actor, User target) throws NoPermissionException, UserAlreadyInC

    Role actorRole = chat.getRoleofUser(actor);
    if (!actorRole.canAddMember()) {
        throw new NoPermissionException(action:"add member");
    }

    if (chat.isUserinChat(target)) {
        throw new UserAlreadyInChatException();
    }
    if (chat instanceof PV) {
        throw new NoPermissionException(action:"add members to PV chat");
    }

    ChatMember newMember = new ChatMember(target, chat.getDefaultJoinRole());
    chat.addMember(newMember);
}

```

Extensibility:

Our project is built to support future extension and changes. This is achieved through:

- ❖ **Interface-based design:** Chat roles like Admin, Owner, and Viewer implement a shared Role interface. New roles can be added easily. Or UserPlans now are Basic, Silver and Gold and we can add more plans .

```
public interface Role {  
    boolean canSendMessage();  
    boolean canEditMessage(Message message, User editor);  
    boolean canDeleteMessage(Message message, User deleter);  
    boolean canAddMember();  
    boolean canDeleteMember();  
    boolean canManageAdmins();  
    boolean canChangeTopic();  
    String getRoleName();  
}
```

```
✓ public interface UserPlan {  
  
    int getMaxPosts();  
    boolean canCreateChannel();  
    boolean canCreateGroup();  
    String getPlanName();  
  
}
```

- ❖ **Abstract base classes:** **Chat and Message** are abstract classes extended by specific types like Group, PV, Channel, and TextMessage, MultimediaMessage and we can easily add new type of messages or chat without any other parts of project.

```
public abstract class Message implements Identifiable{
    protected User sender ;
    protected LocalDateTime timestamp ;
    protected static int counter = 1 ;
    protected int id ;

    public Message(User sender)
    {
        this.id = counter++;
        this.sender = sender ;
        this.timestamp = LocalDateTime.now();
    }

    public abstract void display();

    public User getSender()
    {
        return this.sender;
    }

    public LocalDateTime getTimestamp()
    {
        return this.timestamp;
    }
}
```

```
public abstract class Chat implements Identifiable{
    private static int counter = 1 ;
    private final int id ;
    private ArrayList<ChatMember> members = new ArrayList<>();
    private ArrayList<Message> messages = new ArrayList<>();
    private String Title ;

    public Chat() ...

    public void addMember(ChatMember user) ...

    public void removeMember(User user) ...

    public ArrayList<ChatMember> getMembers() ...

    public int countMembers() ...

    public ArrayList<Message> getMessages() ...

    public int countMessages() ...

    public void addMessage(Message message) ...

    public void deleteMessage(Message message) ...
}
```


- ❖ **Enums and constants:** Enums like `ActionType` allow for easy addition of new action types.

```
public enum ActionType {  
    ADDED(text:"added"),  
    REMOVED(text:"removed"),  
    RENAMED(text:"changed the Chat name"),  
    CREATED(text:"Created This Chat");  
  
    private final String text;  
  
    ActionType(String text) {  
        this.text = text;  
    }  
  
    public String getText() {  
        return text;  
    }  
}
```

- ❖ **Decoupled architecture:** The logic is split across Manager and Model classes, allowing UI or persistence changes with minimal impact.

This modular architecture ensures that adding new features (e.g., voice messages, new user roles,...) requires minimal code changes.