UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS214

Practical 3 Specifications

Release Date: 07-09-2025

Due Date: 30-09-2025 at 11:00

Total Marks: 135

Contents

1 General Instructions **3**

2 Plagiarism **3**

3 Mark Distribution **4**

4 Overview and Background **4**

4.1 Mediator 5

4.1.1 Classes and Attributes 5

4.1.2 Implementation Details 6

4.2 Command 7

4.2.1 Classes and Attributes 7

4.2.2 Implementation Details 9

4.3 Iterator 9

4.4 Your Choice of Design Pattern 10

4.5 Doxygen 10

5 Task 1: UML Diagrams **10**

5.1 Class Diagrams (20 Marks) 10

5.2 Activity Diagrams (10 Marks) 11

6 Task 2: Implementation **11**

7 Task 3: FitchFork Testing Coverage **11**

8 Task 4: Demo Preparation **12**

9 Submission Instructions **12**

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed in pairs.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as **no extension will be granted.**
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- You can use any version of C++.
- You can import the following libraries:
  - vector
  - string
  - iostream
  - map
  - list
- You will upload your code with your main to FitchFork as proof that you have a working system.
- **If you meet the minimum FitchFork requirements (working code with at least 60% testing coverage), you will be marked by tutors during your practical session. Please book in advance (Instructions will follow on ClickUp).**
- **You will not be allowed to demo if you do not meet the minimum FitchFork requirements.**

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with permission) and copying material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to `http://www.library.up.ac.za/plagiarism`. If you have any questions regarding this, please ask one of the lecturers to avoid misunderstanding.

3 Mark Distribution

Activity	Mark
Task 1: UML Diagrams	30
Task 2: Implementing patterns	60
Task 3: FitchFork Testing Coverage	10
Task 4: Demo preparation	20
Task: Doxygen	10
Task: Github	5
Total	**135**

Table 1: Mark Allocation

4 Overview and Background

After a fruitful discussion about your pets and your love for Computer Science, you and your friend decided to create **PetSpace**, a place for animal lovers to rave about their favourite animals and tech-related things (even better if it's about animals doing tech-related activities!). You decided that the platform should have multiple themed rooms where users can join and interact with one another. Knowing that this platform could be a massive hit, you kept **extensibility and maintainability** in mind from day one. You decided to start off with two main chat rooms: **CtrlCat** and **Dogorithm**. Since many people share similar interests, the chat rooms can hold multiple users. You also allow users to be part of multiple chat rooms, as some love both cats and dogs.

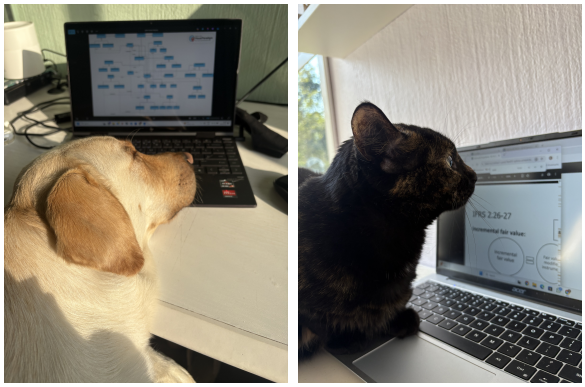
People are creative and send very funny messages, so to keep your users happy, you decided to store conversations so that users can scroll back and enjoy a good laugh on a bad day.

Organisation is key, and system expansion is inevitable, so you designed the platform using the **Command pattern** to separate sending and saving messages. This will make it easy to add even more commands as the system grows (or even now, if needed).

To keep interactions smooth and manageable, each chat room acts as a **Mediator**, handling all message delivery between users. This ensures that users never need to communicate directly with each other — the rooms manage everything behind the scenes.

Since chat rooms can have multiple users and a long history of messages, you implemented the **Iterator pattern** to allow easy traversal of both users and messages. This way, admins, moderators, or even future features can browse through users or scroll through message history without ever needing to access the internal lists directly. Combined with the Command pattern, this design allows for clean, flexible, and easily maintainable interactions throughout the platform.

To keep you motivated and inspired while coding and chatting, here's Bentley and Charlie, our Dogorithm and CtrlCat mascots, hard at work, showing that even our furry friends can be tech enthusiasts!

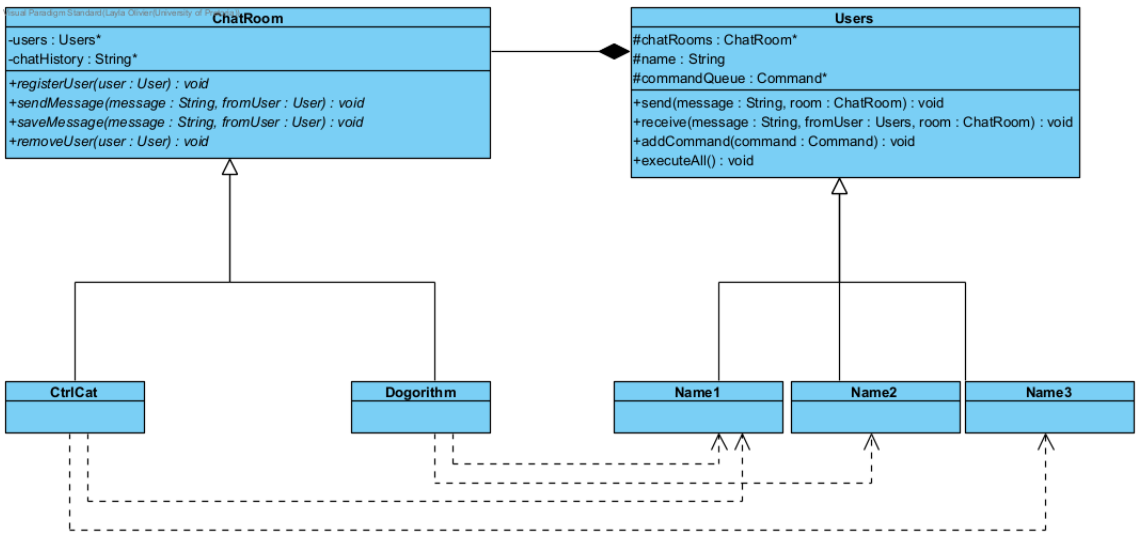


Read the following sections carefully and complete the tasks that follow.

4.1 Mediator

In PetSpace, users should be able to interact with one another in chat rooms. Without a mediator, every user would need to know about all the other users in a room and directly send messages to them, creating a messy web of dependencies. You can imagine how messy it would be if we had even more chat rooms and hundreds of users. Not to mention that a single user can be a part of multiple chat rooms. However, by introducing a Mediator, the chat room itself handles all communication between users.

Two main chat rooms are initially available: **CtrlCat** and **Dogorithm**. Each room manages its own list of users and chat history, ensuring that users only ever interact with the mediator, and not with one another directly.



4.1.1 Classes and Attributes

1. **ChatRoom**

   - *Attributes*
     - users: User*
       Keeps track of all users in this chat room.

- chatHistory: String*
  Stores all messages sent in the room.

- *Methods*
  - registerUser(user: User): void
    Abstract Method.
  - removeUser(user: User): void
    Abstract Method.

2. **CtrlCat, Dogorithm**

   - *Methods*
     - registerUser(user: User): void
       Registers a user with the chat room.
     - removeUser(user: User): void
       Removes a user from the chat room.

3. **User**

   - *Attributes*
     - name: String
     - chatRooms: ChatRoom*
       Remember Users can be a part of multiple chat rooms.

   - *Methods*
     - send(message: String, room: ChatRoom): void
       Sends a message to a chosen chat room.
     - receive(message: String, fromUser: User, room: ChatRoom): void
       Receives a message from the chat room.

4. **Name1, Name2, Name3**

   - Instances of User with specific names. Each one participates in one or more chat rooms.
   - Give your users names of your choosing.

4.1.2 Implementation Details

Mediated Communication

Users never communicate directly with one another. Instead, they send messages to the chat room, which forwards the messages to all other registered users. This keeps the system loosely coupled and easy to maintain.

Extensibility

New chat rooms can be added by simply creating new ConcreteMediators. Users can also be extended without impacting the mediator logic.

Constraints

You need to implement at least 3 users. You can decide if you want users to all be the same, i.e have the same capabilities or if you want users with different capabilities such as an Admin user, a normal user, etc.

There has to be at least **1** user that is a part of more than one chat room, i.e Name1 must be a part of CtrlCat and Dogorithm.

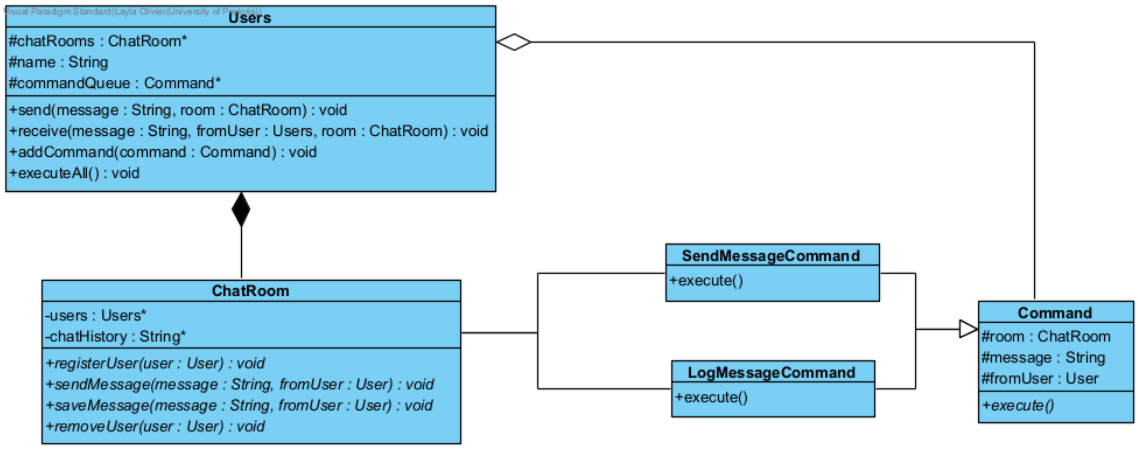
You can add more chat rooms, be creative with the names! You can also add more users, but 3 is enough to show the functionality that is being tested.

The send and receieve methods can either be abstract in the User class or they can be implemented directly in the User class. It is your choice to decide on the implementation. You must then be able to explain why you chose that approach and if the advantages/ disadvantages. **Hint:** Think in terms of flexibility and extensibility.

4.2 Command

In PetSpace, when a user sends a message, more than one action may need to occur. The message should be delivered to all users in the chat room, but it should also be stored in the chat history for later retrieval. Instead of hard coding this logic directly into the User or ChatRoom classes, the Command pattern encapsulates each action as a separate object.

This allows new commands (e.g., logging, moderation, notifications) to be added later without modifying the existing system.



4.2.1 Classes and Attributes

1. **Command**

   - *Attributes*
     - chatRoom: ChatRoom
     - fromUser: User

- message: String

- *Methods*
  - execute(): void
    Abstract Method.

2. **SendMessageCommand**

- *Methods*
  - execute(): void
    Uses the chatRoom to deliver the message to all users.

3. **SaveMessageCommand**

- *Methods*
  - execute(): void
    Appends the message to the chatRoom's history.

4. **User (Invoker)**

- *Attributes*
  - commandQueue: Command*
    List of all the commands..

- *Methods*
  - addCommand(command: Command): void
    Adds a command to the queue.
  - executeAll(): void
    Executes all queued commands in sequence.

5. **ChatRoom (Receiver)**

- *Attributes*
  - users: User*
    Keeps track of all users in this chat room.
  - chatHistory: String*
    Stores all messages sent in the room.

- *Methods*
  - sendMessage(message : String, fromUser : User) : void
    Sends the message to all users in the room.
  - saveMessage(message : String, fromUser : User) : void
    Appends the message to the chat history for later retrieval.

4.2.2 Implementation Details

Message Actions

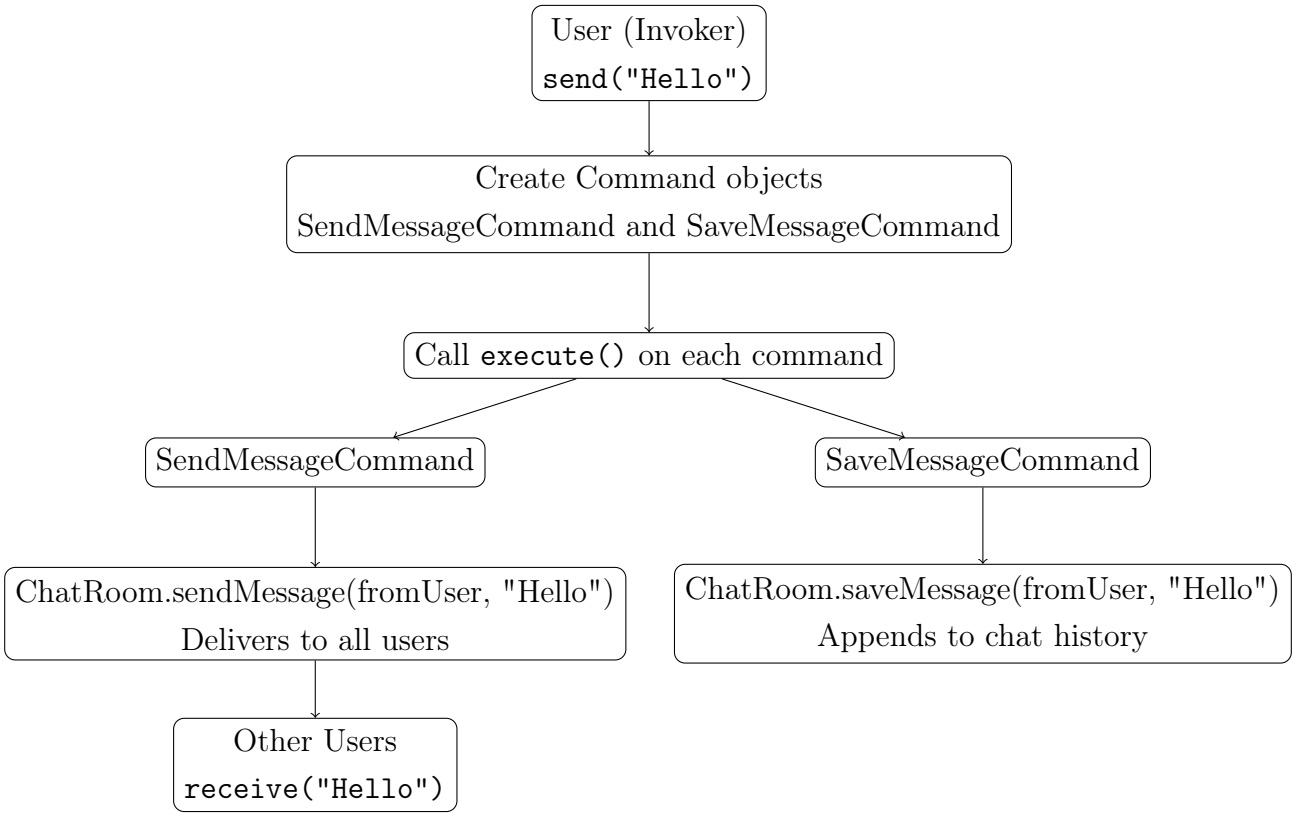
When a user sends a message, two commands can be created: SendMessageCommand (to deliver the message to all users) and SaveMessageCommand (to add the message to history). The User then executes these commands in order.

Extensibility

Future commands can easily be added, such as LogMessageCommand, ModerateMessageCommand, or SendNotificationCommand, without altering the existing logic. You may add these if you would like to.

Flow

Below is a basic flow to help you understand how the message gets from the User to the Chat Room.



4.3 Iterator

Currently there is no Iterator design pattern that is a part of the solution. Your task is to identify a suitable place to use the Iterator pattern and implement it.
You can add the pattern anywhere that is appropriate. Make sure that you can justify your choice to the tutors.
When deciding where to add the Iterator pattern, consider the following:

- Ensure the intent of the pattern is clearly demonstrated in your chosen context.
- Think about where there is a **list** of items to be traversed.

4.4 Your Choice of Design Pattern

In the past practicals you have used the following patterns: Memento, Template Method, Prototype, Factory Method, Abstract Factory, Strategy, State, Composite, Decorator and Observer.
Your task now is to **pick one of these patterns** and integrate it into the current system. Let your creativity run wild but make sure that the pattern you choose and where you choose to implement it, makes sense. You will be required to explain your reasoning to the tutors in your demo.
When choosing which pattern to use, consider the following:

- Has the intent of the pattern been demonstrated?
- Is the pattern creational, behavioural or structural? How will this fit with the current patterns in the system? (Think about Mediator, Command and Iterator and if they are creational, behavioural or structural.)
- How will the pattern positively contribute to the system?
- How will the pattern increase the extensibility of the system?

4.5 Doxygen

For this practical, all code that you submit must be documented using Doxygen. This includes:

- File-level documentation (brief description of purpose, author, date).
- Function/method documentation (parameters, return values, purpose, and notes).
- Class/struct documentation (where applicable, describing attributes and responsibilities).

5 Task 1: UML Diagrams

In this task, you need to construct UML Class and Activity diagrams for the given scenario.

5.1 Class Diagrams (20 Marks)

- Create a comprehensive UML class diagram that shows how the classes in the scenario interact and participate in various patterns.
- Ensure it includes all relevant classes, their attributes, and methods.
- Identify and indicate the design patterns each class is involved in, as well as their roles within these patterns. (Remember a class can participate in more than one pattern)
- Display all relationships between classes.
- Add any additional classes that are necessary for a complete representation.
- Do not forget to add your Iterator design pattern and your other choice of design pattern to the UML Class Diagram.

5.2 Activity Diagrams (10 Marks)

- Develop a UML Activity diagram for your system showing a User sending and saving a message.
- Use the correct notation.

6 Task 2: Implementation

In this task, you need to implement the design patterns described in the scenario.

- Use your UML class diagram as a guide to integrate the design patterns. Note that a **single class** can be part of **multiple patterns**.
- The provided scenario outlines the minimum requirements for this practical. Feel free to add any additional classes or functionalities to better demonstrate your understanding of the patterns and tasks.
- Thoroughly test your code (more details will be provided in the next task). Tutors will only mark code that runs.
- You may use arrays, vectors, or other relevant containers. Avoid using any libraries not mentioned in the general instructions, as your code must be compatible with FitchFork.
- Tutors may require you to explain specific functions to ensure you understand the pattern being implemented.

7 Task 3: FitchFork Testing Coverage

In this task, you are required to upload your completed practical to FitchFork. The FitchFork submission slot will open on 9 September 2025.

- Ensure that your code has at least 60% coverage in your testing main. This is necessary for demonstrating your work to the tutors.
- You will need to show your FitchFork submission with sufficient coverage to the tutor before being allowed to demo.
- You will be required to download your code from FitchFork for demonstration purposes.
- It might be useful to have two main files: a testing main for FitchFork and a demo main with a more user-friendly interface (e.g., a terminal menu) for demoing. This is just a suggestion, not a requirement. You are required to have at least a testing main. If you choose to have a demo main, upload it to FitchFork as well, but ensure it does not compile and run with make run.
- Name your testing main file **TestingMain.cpp**.

- If you create a demo main, name it **DemoMain.cpp** so it can be excluded from the coverage percentage. Note that you do not have to test your demo main.
- Additionally, upload a makefile that will compile and run your code with the command **make run** and **make coverage**.
- Please note that all submissions will be checked using valgrind, so please ensure your memory management is in check.

8 Task 4: Demo Preparation

You will have 10 minutes to demo your system and answer any questions from the tutors. Proper preparation is crucial.

- Ensure you have all relevant files open, such as UML diagrams and source code.
- Set up your environment so that you can easily run the code during the demo after downloading it.
- Be ready to demonstrate specific function implementations upon request.
- Practice your demo to make sure it fits within the allotted time and leaves time for questions.
- Prepare a brief overview (30 seconds) of your system to quickly explain its functionality and design.
- Make sure your testing and demo mains (if applicable) are ready to show their respective functionalities.
- Have a clear understanding of the design patterns used and be prepared to discuss how they are implemented in your code. Also, think about other potential use cases for the patterns.
- Be prepared to answer questions. If you do not know the answer, inform the marker and offer to return to the question later to avoid running out of time.

9 Submission Instructions

You will submit on FitchFork.

- FitchFork submission:
  - Zip all files.
  - Ensure you are zipping the files and not the folder containing the files.
  - Upload to the appropriate slot on FitchFork well before the deadline, as no extensions will be granted.
  - Ensure that you have at least 60% coverage.