ADVANCED PROGRAMMING (COMP-1549)

Coursework Report (Group No: 76)

Introduction

For this project, our team teamed on developing an effective and adaptable Java-based chat application, including significant design patterns like Factory, Singleton, and Observer. These patterns were essential in ensuring consistent resource management, permitting efficient and efficient communication throughout several associated customers, and delivering real-time updates throughout meeting with clients. Furthermore, even in the event of unplanned coordinator reassignments or disconnections, the system operated well because to the efficient implementation of fault tolerance techniques. To verify the accuracy as well as reliability of our system, extensive JUnit testing was conducted, carefully confirming all essential functionalities such coordinator management, private messaging, message broadcasting, and active client handling.

Table 1: **Design Pattern Implementation**

Design Pattern Name	Involved classes/methods	Justification
Singleton	**Server.java** getInstance()	We chose the Singleton pattern for the Server class because our chat application needs a single central point to manage everything effectively. By allowing only one instance of the server, we ensure that all connected clients share the same synchronized state, preventing conflicts and keeping communication consistent across the application.
Factory	**ClientHandlerFactory.java** createClientHandler(Socket socket)	We decided to use the Factory pattern because it cleanly separates the logic for creating client-handler objects from the server's main operational logic. This approach makes our system easier to maintain and more modular, allowing our team to update or change how clients are handled without impacting the overall server management.

Throughout our implementation, the team ensured clear separation of responsibilities through appropriate use of design patterns. The Singleton pattern was selected collectively to keep our server state synchronized across multiple client connections, a decision we unanimously agreed simplified state management. Similarly, adopting the Factory pattern allowed us to independently manage client handler instantiation, thus providing modularity.

Table 2: **JUnit Testing Implementation**

Test Name	Involved classes/methods	Description / Justification
Singleton Instance Verification	**ServerTest.java** testSingletonInstance()	This test verifies that our server fully adheres to the Singleton pattern. By checking that multiple calls to getInstance() always return the exact same server object, we can confidently ensure there's just one central place managing all active clients. This consistency helps our team avoid unexpected

		behaviors and maintain reliable client management.
Coordinator Assignment Test	**ServerTest.java** testCoordinatorAssignment()	This test validates our coordinator assignment logic. It ensures the first client who joins automatically becomes the coordinator, and if that client disconnects, the role is smoothly and predictably transferred to the next client in line. This test helps our team confirm that chat management remains reliable and uninterrupted, even when clients leave unexpectedly.
Private Message Delivery Test	**ClientHandlerTest.java, TestClientHandler.java** testSendPrivateMessage_Success()	We created this test to make sure our private messaging feature works correctly, verifying that messages go only to their intended recipients. By using a mock handler (TestClientHandler), we can simulate sending messages and accurately check delivery without needing actual network connections. This helps our team confidently confirm the accuracy and reliability of private messaging.
Broadcast Message Delivery Test	**ClientHandlerTest.java, verifyClientHandler.java** testSendBroadcastMessage()	This test was crucial for making sure group communication works reliably. It verifies that when one client sends a broadcast message, it successfully reaches every active client in the group. By using a mock handler, we could accurately simulate and confirm the exact delivery and content of these broadcast messages, helping us confidently ensure reliable communication.

Our team took a collaborative approach to testing, allowing us to thoroughly verify all critical features. For instance, we placed special emphasis on testing private and broadcast messaging, as it was important to make sure messages were accurate and reached the correct recipients. By using mock client handlers, we could focus purely on logic and correctness without worrying about real network conditions, giving us greater confidence that our application is robust and reliable.

Table 3: **Fault Tolerance Implementation**

Fault Tolerance Feature	Involved classes/methods	Description / Justification
Coordinator reassignment upon Coordinator leaving	**Server.java** removeClient(String clientId), setNewCoordinator()	Our implementation makes sure that if the current coordinator disconnects or leaves the application, the coordinator role automatically moves to the next client in the order they joined. This clear sequence helps keep our app stable and minimizes any disruption or confusion for users.
Handling Non-coordinator Client Disconnection	**Server.java** inspectActiveClients(), removeClient(String clientId)	To keep our group membership accurate, we set up periodic checks every 20 seconds to identify and remove inactive clients. This helps our application manage resources efficiently and ensures that the list of active clients stays reliable and up to date.

Prevention of Duplicate Client IDs	**Server.java** addClient(String clientId, ClientHandler handler)	Our team realized that duplicate client IDs could lead to confusion and misdirected messages. To avoid this, our server actively checks for and rejects duplicate IDs. By enforcing unique client names, we maintain clarity and ensure that messages always reach the correct recipient.

Fault tolerance was a major priority for our team. We agreed that predictable coordinator reassignment was essential to providing a smooth user experience, so we implemented a clear, ordered approach rather than assigning roles randomly. Additionally, by rejecting duplicate client IDs and routinely removing inactive clients, we ensured the application remained consistent and efficient. These decisions highlight our focus on making the system reliable and robust.

Table 4: **Usage of AI**

AI Program	Classes and/or Methods	Contribution
ChatGPT	**We took AI assistance for these:** **Server.java:** setNewCoordinator(), inspectActiveClients() **ClientHandler.java:** sendPrivateMessage(), sendClientList() **JUnit Tests:** ServerTest.java: (testCoordinatorAssignment()), ClientHandlerTest.java: (testSendBroadcastMessage())	**20%** ChatGPT was used by us to as a supportive tool to assist with some specific, challenging implementations related to fault tolerance (coordinator reassignment, checking active clients), certain methods for message handling, selected JUnit tests, and general understanding of advanced design patterns.

Our team has built the core logic and functionality of the chat application (socket handling, threading, client interaction) by leveraging our personal knowledge, team collaboration, and practical demonstrations from multiple YouTube tutorials such as:
https://youtu.be/gLfuZrrfKes?si=DKVdgM_YeLP3aVfX
https://youtu.be/cRfsUrU3RjE?si=Wcf7jJi37TedtOBB

Conclusion: During this project, our team successfully created a complete Java-based chat application. We carefully chose design patterns that made the app modular, easy to maintain, and efficient in resource management. We also carried out thorough testing with JUnit, which helped us confirm that coordinator assignments worked correctly, messages reached the right users (both broadcast and private), and the system handled errors smoothly. In the end, our application provided stable and user-friendly communication among clients, meeting all the original project requirements.

Future Work: Looking ahead, our team wants to enhance the chat app by improving the GUI interface, making it easier and more intuitive to use, with clear visuals showing who's online or offline. We're also thinking about adding features like automatic reconnection for dropped users and stronger security, including message encryption to protect privacy. Plus, we're exploring options like distributed servers and load balancing to smoothly handle more users as the app grows, keeping things fast and reliable.