Advanced Programming Report

1. Design Patterns

To increase scalability and maintainability, design patterns are used in the project implementation. The main design patterns are listed in the table below:

Design Pattern Name	Involved Classes/Methods	Justification
Factory Method	DistributedChatServer, ClientHandler class	The server creates a product called the ClientHandler class each time a new client connects. This architecture isolates client connection management and allows each client to be processed independently.
Observer	DistributedChatServer, ClientHandler, PrintWriter map	The system implements a notification mechanism where client sockets are observers. The system broadcasts notifications to all clients whenever events such as join, leave or messages occur.

2. JUnit Testing

The project includes unit tests to verify correct operation of individual components. The important test cases are shown in the table below.

Test Name	Involved Classes/Methods	Description
TestCoordinator	testAddMember() testRemoveMember()	It serves to verify that a member of the coordinator list can be successfully added or removed. Also causes the coordinator to be correctly reassigned after removal.
TestDistributedChatClientGUI	testClientConnection()	Verifies that the client can connect to the server via a socket and send messages. Evaluates the basic communication channel that makes chat possible.
TestDistributedChatServer	testServerConnection() testMessageBroadcast()	Assesses that the server can manage broadcasting messages to all clients and accepts incoming client connections.
TestMember	testMemberCreation() testMemberToString()	Verifies that the Member object was instantiated correctly and that the member information (toString() output) were formatted correctly in strings.

3. Fault Tolerance

The system is designed to operate without interruption in case of unexpected failures. The following table describes fault tolerance strategies:

Fault Tolerance Feature	Involved Classes/Methods	Description
Basic Member Verification	Coordinator.addMember()	Checks for duplicate IDs to prevent conflicts in the group state and guarantee consistency.
Graceful Exit Handling	DistributedChatClientGUI.quitChat(), DistributedChatServer.ClientHandler.run()	Makes certain that resources are released appropriately in the event of an unexpected client disconnect or quit. Keeps the group stable and stops the server from crashing.
Periodic Check Stub	Coordinator.startPeriodicCheck()	Active members are currently logged every 20 seconds. In future versions, it can be extended to check for client responsiveness using heartbeat messages and automatically remove inactive members or trigger coordinator re-election.
Coordinator Reassignment	Coordinator.removeMember(), DistributedChatServer.broadcastNewCoordinator()	Another active member gets promoted when a coordinator leaves. To preserve group integrity, all clients are informed about the new coordinator.

4. AI Usage

AI Program	Classes and/or Methods	Contribution
Claude.ai (https://claude.ai)	DistributedChatServer.ClientHandler class Coordinator.removeMember()	10% of the class logic 15% of method logic

5. Conclusion

The project's fault-tolerant features, sound architecture, and appropriate unit testing enable it to successfully deploy a distributed chat system. Design patterns improve code maintainability, and JUnit tests guarantee accuracy. The system handles client disconnections and coordinator transitions as some of the failures. The report is academically honest as it clearly states the AI-assisted contributions.

The periodic check mechanism should be extended to implement heartbeat messages as mentioned in the fault tolerance section for automatic detection and removal of non-responsive clients. Also, the system could be improved with end to end encryption for private messages and a more sophisticated coordinator election algorithm.

The structured approach guarantees that the project meets coursework requirements efficiently.