Assignment: Modular Java Packet-Based Server Framework

Project Purpose

This project involves developing a modular Java TCP server framework that communicates using structured packets. The server mimics real-world backends such as multiplayer game servers, chat apps, and protocol testing environments. It handles multiple persistent TCP connections, supports packet types like login, chat, and pings, and demonstrates SOLID design principles and real-world multithreaded server behavior.

TCP Networking Concepts

TCP is a connection-oriented, stream-based protocol guaranteeing reliable and ordered delivery. The connection lifecycle includes a 3-way handshake (SYN, SYN-ACK, ACK), data transmission, and 4-way termination (FIN, ACK, FIN, ACK). Connections in this server are persistent. Unlike short-lived HTTP/1.0 connections, persistent TCP allows real-time interactions.

What Is a Packet?

A packet is a structured message representing a single client-server operation, such as login, chat, or ping. They are self-contained and handled using the Command pattern.

Buffering and Framing

TCP streams may split or merge packets, requiring a framing strategy. Use length-prefixing or delimiters like newline '\n'. Java does not provide built-in framing; use BufferedReader or DataInputStream for manual parsing.

Threads in Context

Threads are units of concurrent execution. One thread listens for new connections, and a new thread is spawned per client for communication. This enables the server to handle multiple clients concurrently.

Java Code Sample

```
ServerSocket serverSocket = new ServerSocket(12345);
while (true) {
    Socket clientSocket = serverSocket.accept();
    new Thread(() -> {
        try (BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()))) {
            String packetStr;
            while ((packetStr = in.readLine()) != null) {
                Packet packet = PacketRegistry.fromJson(packetStr);
                CommandExecutor.execute(packet);
            }
        } catch (IOException e) {
            System.out.println("Client disconnected.");
        }
    }).start();
}
```

Assignment: Modular Java Packet-Based Server Framework

Modules and Design Patterns

- ServerManager: Singleton - Controls lifecycle
- ClientConnection: Observer - Notifies on packet arrival
- Packet: Factory/Command - Create and process packets
- PacketRegistry: Factory - Dynamically instantiate packets
- CommandExecutor: Command/Strategy - Decoupled logic execution
- Logger: Decorator - Dynamic logging enhancement
- ProtocolAdapter: Adapter - Byte stream to object
- FrontendStub: MVC/Observer - UI abstraction and event listening

Expected Packet Types

- LoginPacket / LoginSuccessPacket
- PingPacket / PongPacket
- ChatMessagePacket
- ErrorPacket
- ServerAnnouncementPacket

Packet Flow Example

Client sends: {"type": "LoginPacket", "username": "mitch", "password": "123"}
Server reads -> PacketRegistry creates -> CommandExecutor handles -> Server replies: {"type": "LoginSuccessPacket"}

Deliverables

1. Java codebase with modular structure
2. UML class and sequence diagrams
3. Markdown answers to design/OOP questions
4. Optional: CLI frontend for testing

Design Pattern Justification

1. Why Singleton for ServerManager?
2. How does Factory help with packet extension?
3. Command vs if-else chains?
4. Observer for loose coupling?
5. Decorator for logging enhancement?

OOP Principle Justification

1. SRP in ClientConnection?
2. OCP in packet addition?
3. DIP in CommandExecutor?
4. DRY in packet structures?
5. LSP challenges and fixes?