1. Singleton Pattern:

Implement a singleton for managing a network connection pool. This ensures that multiple parts of your application can share a single pool of network connections efficiently.

java

public class NetworkConnectionPool {

private static NetworkConnectionPool instance;

private NetworkConnectionPool() {

// Initialize connection pool

}

public static synchronized NetworkConnectionPool getInstance() {

if (instance == null) {

instance = new NetworkConnectionPool();

}

return instance;

}

// Other methods to manage connections

}

2. Factory Pattern:

Create a factory for generating different types of network protocols (TCP, UDP) based on client requirements. This can help abstract the creation of network protocol objects.

java

public interface Protocol {

void sendData(byte[] data);

void receiveData();

}

public class TCPProtocol implements Protocol {

// Implement TCP protocol methods

}

public class UDPProtocol implements Protocol {

// Implement UDP protocol methods

}

public class ProtocolFactory {

public Protocol createProtocol(String type) {

if (type.equalsIgnoreCase("TCP")) {

return new TCPProtocol();

} else if (type.equalsIgnoreCase("UDP")) {

return new UDPProtocol();

} else {

throw new IllegalArgumentException("Unknown protocol type");

}

}

}

3. Observer Pattern:

Use the observer pattern to notify multiple parts of your application when network events (like connection established, data received) occur.

java

public interface NetworkObserver {

void update(String event);

}

public class NetworkObservable {

private List<NetworkObserver> observers = new ArrayList<>();

public void addObserver(NetworkObserver observer) {

observers.add(observer);

}

public void notifyObservers(String event) {

for (NetworkObserver observer : observers) {

observer.update(event);

}

}

// Methods to trigger network events and notify observers

}

4. Strategy Pattern:

Implement the strategy pattern to dynamically change the algorithm used for data encryption or compression over the network.

java

public interface DataStrategy {

void apply(byte[] data);

}

public class EncryptionStrategy implements DataStrategy {

// Implement encryption logic

}

public class CompressionStrategy implements DataStrategy {

// Implement compression logic

}

public class DataProcessor {

private DataStrategy strategy;

public void setStrategy(DataStrategy strategy) {

this.strategy = strategy;

}

public void processData(byte[] data) {

strategy.apply(data);

}

}

5. Proxy Pattern:

Use the proxy pattern to control access to a remote object or manage the connection to a remote server.

java

public interface Network {

void sendData(byte[] data);

void receiveData();

}

public class RealNetwork implements Network {

// Implement network operations

}

public class NetworkProxy implements Network {

private RealNetwork realNetwork;

public NetworkProxy() {

// Initialize realNetwork as needed

}

@Override

public void sendData(byte[] data) {

// Add logging or security checks

realNetwork.sendData(data);

}

@Override

public void receiveData() {

// Add logging or security checks

realNetwork.receiveData();

}

}

Exercise:

Implement a scenario where you need to establish a secure communication channel using TLS/SSL in a client-server architecture. Utilize appropriate design patterns such as Factory, Strategy, and Proxy to handle various aspects like protocol selection, encryption, and client-server interaction.