**5. Real-time Chat Application Programming Exercise**

**Singleton: Chat Room Manager:**

class ChatRoomManager:

\_instance = None

def \_\_init\_\_(self):

if not ChatRoomManager.\_instance:

self.chat\_rooms = {}

ChatRoomManager.\_instance = self

@staticmethod

def get\_instance():

if not ChatRoomManager.\_instance:

ChatRoomManager()

return ChatRoomManager.\_instance

def get\_chat\_room(self, room\_id):

if room\_id not in self.chat\_rooms:

self.chat\_rooms[room\_id] = ChatRoom(room\_id)

return self.chat\_rooms[room\_id]

**Observer Pattern: Chat Room and User:**

class ChatRoom:

def \_\_init\_\_(self, room\_id):

self.room\_id = room\_id

self.users = []

self.messages = []

def join\_room(self, user):

self.users.append(user)

self.notify\_users(f"{user.username} has joined the room.")

def leave\_room(self, user):

self.users.remove(user)

self.notify\_users(f"{user.username} has left the room.")

def send\_message(self, user, message):

formatted\_message = f"{user.username}: {message}"

self.messages.append(formatted\_message)

self.notify\_users(formatted\_message)

def notify\_users(self, message):

for user in self.users:

user.update(message)

class User:

def \_\_init\_\_(self, username):

self.username = username

def update(self, message):

print(f"New message for {self.username}: {message}")

def send\_message(self, chat\_room, message):

chat\_room.send\_message(self, message)

**Adapter Pattern: WebSocket Adapter:**

class WebSocketAdapter:

def \_\_init\_\_(self, connection):

self.connection = connection

def send(self, message):

self.connection.send(message)

def receive(self):

return self.connection.receive()

class HTTPAdapter:

def \_\_init\_\_(self, http\_request):

self.http\_request = http\_request

def send(self, message):

self.http\_request.post('/send\_message', data={'message': message})

def receive(self):

return self.http\_request.get('/receive\_message')

**Private Messaging**

class PrivateMessage:

def \_\_init\_\_(self, sender, receiver, message):

self.sender = sender

self.receiver = receiver

self.message = message

def send(self):

self.receiver.update(f"Private message from {self.sender.username}: {self.message}")

**Persistent Chat History**

class ChatRoom:

def \_\_init\_\_(self, room\_id):

self.room\_id = room\_id

self.users = []

self.messages = self.load\_history(room\_id) # Load previous messages from storage

def load\_history(self, room\_id):

return []

def send\_message(self, user, message):

formatted\_message = f"{user.username}: {message}"

self.messages.append(formatted\_message)

self.save\_history(formatted\_message)

self.notify\_users(formatted\_message)

def save\_history(self, message):

**Design Patterns**: The solution leverages the Observer pattern for real-time updates to users, the Singleton pattern to manage chat rooms, and the Adapter pattern for communication protocol flexibility.

**OOP Principles:** Encapsulation (handling users and messages), polymorphism (different protocols through adapters), and inheritance (if more complex user roles are added).

**Scalability:** This setup is highly scalable, and the architecture allows for the addition of private messages, chat history, and different communication protocols without major modifications.