1 Testat-Übung: Chat

In dieser Übung geht es darum, sich in existierenden Code einzulesen, Anforderungen daraus abzuleiten und die noch fehlenden Klassen entsprechend zu implementieren. Wir arbeiten hier mit einem Beispiel eines verteilten Systems, welches als einfacher Chat funktioniert.

1.1 Vorgehen

Das entsprechende Projekt ist bereits vorbereitet. Ebenfalls finden Sie die Interface-Definition in `serializer.h` und `chat_client.h`. Sie sollen die entsprechend geforderte Funktionalität in `serializer.cpp` und `chat_client.cpp` implementieren.

Die dazu passenden Unit-Tests sind bereits implementiert und erlauben ihnen, ihren `Serializer` zu testen. Zudem existiert die Server-Applikation `uebung_chat_core_app`, welche Sie für den Test ihres `ChatClient` brauchen.

Öffnen Sie das `workspace`-Repo in CLion als Root-Projekt und navigieren Sie zu `uebungen`/`chat`. Hier finden sie das vorbereitete Projekt. Aktivieren Sie es indem Sie in `uebungen`/`CMakeLists.txt` die folgende Zeile aktivieren:

```
add_subdirectory(chat)
```

Die Aufgaben sind Schritt für Schritt abzuarbeiten. Alle Teile sind zwingend korrekt zu implementieren.

1.2 Testat

Geben Sie die geforderten Implementationen bis zum vereinbarten Datum ab. Senden Sie die Dateien `serializer.h`, `serializer.cpp`, `chat_client.h` und `chat_client.cpp` per Mail an christian.lang@fhnw.ch. Die Lösung muss einerseits alle Unit-Tests, anderseits auch bestimmte Qualitäts-Merkmale erfüllen. Diese Qualitätsmerkmale werden automatisch durch cpplint und manuell durch den Dozenten geprüft. Sie erhalten dementsprechend Feedback zu ihrer Abgabe.

1.3 Ziele

- Sie repetieren den Stoff aus der Vorlesung: Streams, Templates, STL, Containers, Lambdas.
- Sie stellen vorgegebenen Code fertig indem Sie diesen analysieren und ergänzen.
- Sie implementieren einen Serializer/Parser für die Kommunikation.
- Sie integrieren die existierenden Bausteine und bauen damit den Chat-Client.

2 Einführung in das Thema

Ähnlich wie in modernen Kommunikationsmitteln (wie z.B. WhatsApp) läuft die komplette Kommunikation nicht direkt zwischen den Kommunikationspartnern, sondern zentral über einen Server (Message-Broker). Dieser leitet die Nachrichten einerseits weiter, andererseits speichert er diese zentral und stellt so sicher, dass jeder Client alle Nachrichten wieder abfragen kann, ohne dass dieser selbst einen Zustand speichern muss. Zudem erlaubt diese Vorgehensweise, dass Clients auch Nachrichten an momentan nicht aktive Partner senden können und diese erst beim Anmelden des Partners zugestellt werden.

2.1 Architektur-Überblick

Hier erhalten Sie einen Einblick in die Architektur und das Design des existierenden Codes. Dies ist vor allem für die zweite Teilaufgabe wichtig, wo Sie den Hauptteil des Clients implementieren. Der `ChatServer` ist bereits komplett implementiert und erlaubt Ihnen, den `ChatClient` zu testen.

Abbildung 1 zeigt eine grobe Übersicht über die beteiligten Klassen in der Kommunikation. Dabei ist immer genau ein Server beteiligt, aber beliebig viele Clients. Dementsprechend muss der Server eine Liste von aufgebauten `Connection`s und dazugehörenden `Conversation`s verwalten. Der Client dagegen baut genau eine Verbindung zum Server auf und speichert nur diese. Der Client kennt entsprechend nur den Server und keine anderen Clients.

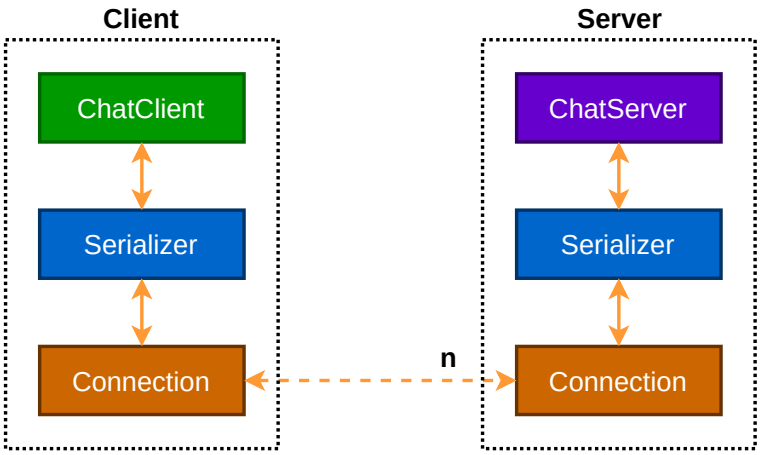

Abbildung 1: Client-Server-Beziehung

Neben dem zentralen Element der `Connection` ist auch der `Serializer` sehr wichtig. Er transformiert Chat-Nachrichten in einen Byte-String aus ASCII-Zeichen und diesen zurück in die gleiche Chat-Nachricht. Dieser Byte-String kann dann mittels einer `Connection` per Websockets über eine

TCP/IP-Verbindung versendet werden. Um eine Übersicht über die Datenobjekte zu erhalten, finden Sie nachfolgend ein abstrahiertes Klassendiagramm, inklusive den öffentlichen Methoden.

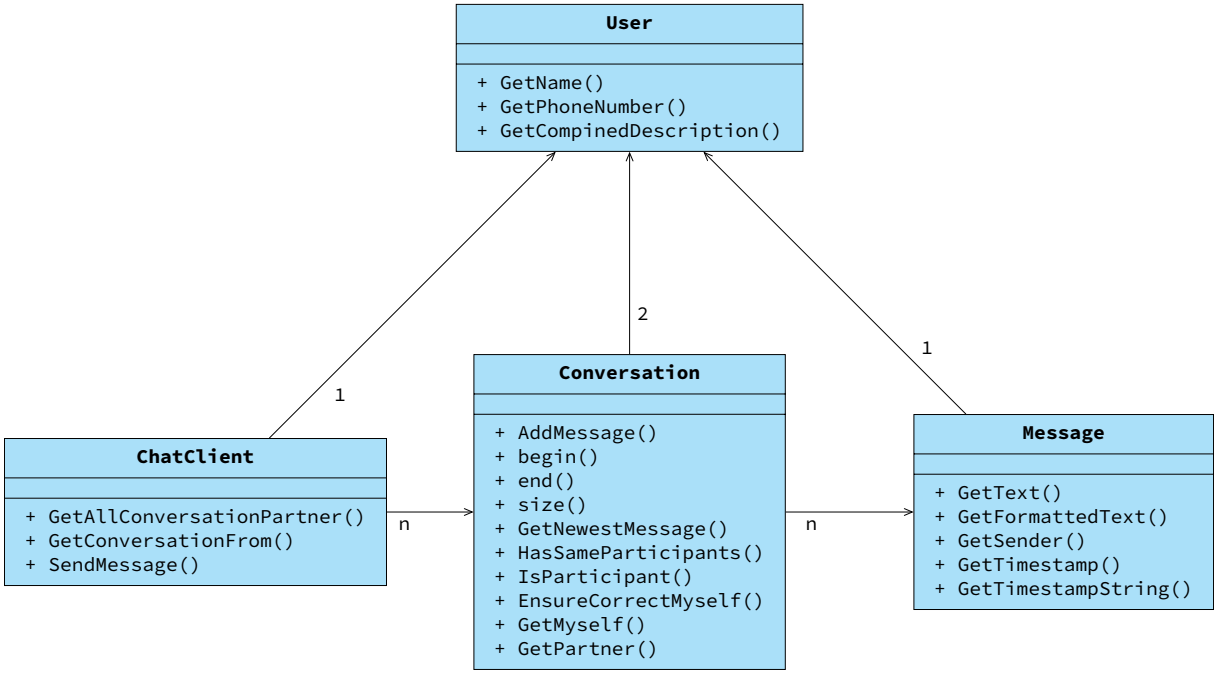

Abbildung 2: Klassendiagramm

Die Klasse `User` wird verwendet, um die beteiligten Personen im Chat zu identifizieren. Sie besteht aus einem `std::string` für den Namen und einem `std::string` für die Telefonnummer. Beachten Sie dabei, dass die eindeutige Identifizierung einer Person immer nur über die Telefonnummer stattfindet. Der Namen ist nur zur verbesserten Identifizierung gedacht.

Die Klasse `Message` aggregiert jeweils einen Sender des Typs `User`, eine Textnachricht und einen Zeitstempel. Der Zeitstempel wird für die Sortierung der Nachrichten verwendet. Als Hilfe existieren zwei Methoden, welche eine formatierte Ausgabe der Nachricht inklusive Zeitstempel und Absender ermöglichen.

Die Klasse `Conversation` aggregiert einerseits die beiden Partner eines Chats, bezeichnet mit `myself` und `partner`, und andererseits eine sortierte Liste von beliebig vielen Messages. Zudem beinhaltet Sie bereits diverse Hilfsmethoden. Die wichtigste ist `EnsureCorrectMyself()`, welche sicherstellt, dass `myself` nach der Synchronisierung der kompletten `Conversation` mit dem Server richtig eingestellt ist. Auf der Server-Seite hat die Zuordnung von `myself` und `partner` keinerlei Bedeutung.

Die Klasse `ChatClient` verwaltet alle Daten auf der Client-Seite. Sie verwaltet den Chat-Benutzer und eine Liste aller seiner `Conversation`s.

3 Serializer

In der ersten Teilaufgabe sollen Sie die Serialisierung und Deserialisierung im vorgegebenen Rumpf der Klasse `Serializer` implementieren. Programmieren Sie dazu die Methoden in der nachfolgenden Tabelle aus. Die Methoden `GetTypeString()` und `EvaluateMessageType()` sind für die Transformation und Evaluierung des Meldungstyps zuständig. Beachten Sie, dass der Serializer eine statische Klasse ist, welche keinerlei Instanzmethoden und Attribute enthält. Alle öffentlichen Methoden, welche Sie zwingend bereitstellen müssen, sind bereits deklariert. Natürlich dürfen und sollen Sie private Hilfsmethoden hinzufügen.

MessageType	Methoden
`kUser`	`SerializeUserMessage()` `ParseUserMessage()`

`kConversationUpdate`	`SerializeConversationUpdateMessage()`
	`ParseConversationUpdateMessage()`
	`GetTypeString()`
	`EvaluateMessageType()`

Damit Ihre eigene Client-Implementierung später mit der zentralen Server-Instanz kommunizieren kann, muss sich ihr Serializer genau an das vorgegebene Protokoll halten. Die Einhaltung des Protokolls wird durch die bestehenden Unittests überprüft. Die nachfolgend aufgelisteten Meldungsbeispiele sollen Ihnen bei der Implementierung helfen. Weitere Beispiele und genauere Definitionen können Sie aus den Unittests entnehmen.

Da der Serializer mit ASCII-Strings arbeitet, macht es Sinn, `std::string` für die Implementierung zu verwenden. Besonders hilfreich in diesem Kontext sind die Methoden: `std::string::find()` und `std::string::substr()`.

```
UserMessage:
 kUser|anna|0791111111|
            ^---------- PhoneNumber
       ^--------------- UserName
 ^--------------------- MessageType
```

```
ConversationUpdateMessage: (1 text message)
 kConversationUpdate|a|1|b|2|text_a|a|1|10|
                                      ^-- Timestamp of first message
                                    ^--PhoneNumber of sender in first message
                                  ^-- UserName of sender in first message
                              ^-- Text of first message
```

```
                            ^-- PhoneNumber of partner in conversation
                          ^-- UserName of partner in conversation
                        ^-- PhoneNumber of myself in conversation
                      ^-- UserName of myself in conversation
 ^-- MessageType
```

```
ConversationUpdateMessage: (2 text messages)
 kConversationUpdate|a|1|b|2|text_a|a|1|10|text_b|b|2|20|
                             ^------------^^------------^
                             each text message consists always of 4 parts
```

4 Chat-Client

Im zweiten Teil sollen Sie den Hauptteil der Client-Applikation selber entwickeln. Der `ChatClient` verwaltet den User, alle seine `Conversation`s und auch die geöffneten Websocket-Connections. Diese Daten erhält er über den Konstruktor. Der Verbindungsaufbau und das einfache Text-User-Interface (TUI) sind bereits im Hauptprogramm `client/src/main.cpp` und der `Manager`-Klasse implementiert.

Wie bereits in der ersten Aufgabe sollen Sie keinerlei anderen Klassen verändern. Implementieren Sie den `ChatClient` in der cpp-Datei mit den vorgegebenen Methoden im Header.

Ihre Aufgaben sind:

- Im Konstruktor:
  - Richten Sie das Empfangen von beliebigen Nachrichten von der Server-Seite ein, indem Sie den Empfangshändler der Verbindung mit SetReceiveHandler setzen.
  - Melden Sie sich mit der `UserMessage` beim Server an. Benutzen Sie dazu `Connection::Send()`.
- Im `MessageHandler` deserialisieren Sie die eingehenden Meldungen vom Typ `kConversationUpdate`. Alle empfangenen `Conversation`s sollen in einem geeigneten Container gespeichert werden. Aktualisierte `Conversation`s sollen nicht neu zum Container hinzugefügt, sondern nur aktualisiert werden. Achten Sie dabei immer auf die richtige Identifizierung von «gleichen» `Conversation`s.
- Der `MessageHandler` signalisiert jeweils mit einem `return true;`, dass er die `message` hinter dem Pointer verändert hat und als Antwort zurücksenden will. Der Client hier antwortet allerdings nie auf Nachrichten des Servers.
- In `SendMessage()` suchen Sie die entsprechende `Conversation` (mittels Hilfsmethode), fügen dieser die neue Nachricht hinzu und senden sie an den Server.

Als Hilfswerkzeuge können Sie alle anderen Klassen verwenden. Hauptsächlich `Serializer`, `Connection` und `Conversation`. Für Debugging und Ausgabe auf die Konsole kann die `Logger`-Klasse eingesetzt werden. Mittels `Logger::SetDebug(true);` im `client/src/main.cpp` können Sie alle Debug-Logs im Client aktivieren.