Programmentwurf
Chat

Name: Rest, Bjarne und Schrader, Maren
Matrikelnummer: 4399673 und 1234726

Abgabedatum: 31.05.2022

Allgemeine Anmerkungen:

es darf nicht auf andere Kapitel als alleiniger Leistungsnachweis verwiesen werden (z.B. in der Form “XY wurde schon in Kapitel 2 behandelt, daher hier keine Ausführung”)

alles muss in UTF-8 codiert sein (Text und Code)

sollten mündliche Aussagen den schriftlichen Aufgaben widersprechen, gelten die schriftlichen Aufgaben (ggf. an Anpassung der schriftlichen Aufgaben erinnern!)

alles muss ins Repository (Code, Ausarbeitung und alles was damit zusammenhängt)

die Beispiele sollten wenn möglich vom aktuellen Stand genommen werden

- *finden sich dort keine entsprechenden Beispiele, dürfen auch ältere Commits unter Verweis auf den Commit verwendet werden*
- *Ausnahme: beim Kapitel “Refactoring” darf von vorne herein aus allen Ständen frei gewählt werden (mit Verweis auf den entsprechenden Commit)*

falls verlangte Negativ-Beispiele nicht vorhanden sind, müssen entsprechend mehr Positiv-Beispiele gebracht werden

- *Achtung: werden im Code entsprechende Negativ-Beispiele gefunden, gibt es keine Punkte für die zusätzlichen Positiv-Beispiele*
- *Beispiele*
  - *“Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt.” (2P)*

    Antwort: Es gibt keine Klasse, die SRP verletzt, daher hier 2 Klassen, die SRP einhalten: [Klasse 1], [Klasse 2]

    Bewertung: falls im Code tatsächlich keine Klasse das SRP verletzt: 2P ODER falls im Code mind. eine Klasse SRP verletzt: 1P

verlangte Positiv-Beispiele müssen gebracht werden

Code-Beispiel = Code in das Dokument kopieren

Gesamt-Punktzahl: 60P

- *zum Bestehen (mit 4,0) werden 30P benötigt*

Kapitel 1: Einführung (4P)

Übersicht über die Applikation (1P)

[Was macht die Applikation? Wie funktioniert sie? Welches Problem löst sie/welchen Zweck hat sie?]

Die Anwendung ist ein Chatraumserver mit dazugehörigem Client. Auf einem Computer wird der Server-Teil ausgeführt und im Netzwerk freigegeben. Mehrere Nutzende können sich dann mittels Client unter Angabe von IP-Adresse und Port in den Chatraum einwählen und untereinander per Textnachrichten kommunizieren. Die Nachrichten werden dabei serverseitig nicht dauerhaft gespeichert. Zweck der Anwendung ist somit die Schaffung eines Austausches im Netzwerk.

Wie startet man die Applikation? (1P)

[Wie startet man die Applikation? Was für Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]

Zum Start der Anwendung muss vorab die Java-Runtime auf dem ausführenden Rechner installiert werden.

Die Anwendung besteht aus einem Client- und Serverteil. Zum Start sind getrennte Aufrufe notwendig. Dazu können die bereitgestellten Skripte start_server.sh und start_client.sh in der CLI ausgeführt werden. Die Anwendung bindet sich dann an tcp/0.0.0.0:25531.

Ein „gesicherter“ Server kann mit einem Aufruf von start_secure_server.sh gestartet werden. Das Kennwort zur Eingabe im Client lautet „secure“

Zum manuellen Aufruf der Applikation mit weitergehender Konfiguration können Kommandozeilenparameter verwendet werden:

java -jar AseChat.jar Server

Optionale Argumente:

Argument	Alias	Bedeutung
--key [String]	-k [String] --password [String]	Kennwort des Chatraumes
--port [int]	-p [int]	TCP-Port
--host [IP-Addr]	-h [IP-Addr]	IP-Adresse

java -jar AseChat.jar Client

Optionale Argumente:

Argument	Alias	Bedeutung
--key [String]	-k [String] --password [String]	Kennwort des Chatraumes
--port [int]	-p [int]	TCP-Port des Servers
--host [IP-Addr]	-h [IP-Addr]	IP-Adresse des Servers
–username [String]	-u [String]	Username des Clients

Falls eines der Argumente nicht gesetzt ist, werden Werte aus der Konfigurationsdatei verwendet.

Technischer Überblick (2P)

[Nennung und Erläuterung der Technologien (z.B. Java, MySQL, …), jeweils Begründung für den Einsatz der Technologien]

Wir haben uns für die Verwendung von Java 11 entschieden, da die Sprache uns bekannt ist und den Anforderungen dieser Aufgabe genügte.

Große Hilfe bei der Entwicklung stellte die Verwendung von Picocli dar. Die Library hilft beim Parsing der Kommandozeilenargumente. So ist es möglich mit wenig Zusatzaufwand eine vollständig POSIX/MS-DOS/GNU-konforme CLI-Anwendung zu erstellen.

Die Kommunikation zwischen Clients und Server erfolgt über TCP-Sockets. Über die Schnittstelle werden “Instructions” ausgetauscht. Das sind strukturierte Befehle, die mit einem zusätzlichen Payload versehen werden können (Eigenentwicklung).

Eine persistente Datenhaltung erfolgt nicht. Daher ist auch keine Anbindungen von DBMS oÄ. vorgesehen.

Kapitel 2: Clean Architecture (8P)

Was ist Clean Architecture? (1P)

[allgemeine Beschreibung der Clean Architecture in eigenen Worten]

Clean Architecture bezeichnet eine haltbare Architektur einer Software, die klar definiert, welche Komponenten bestimmte Rollen übernehmen. Dies führt zu einer klaren Aufteilung, die auch langfristig ohne große Änderungen (Refactoring) lauffähig ist.

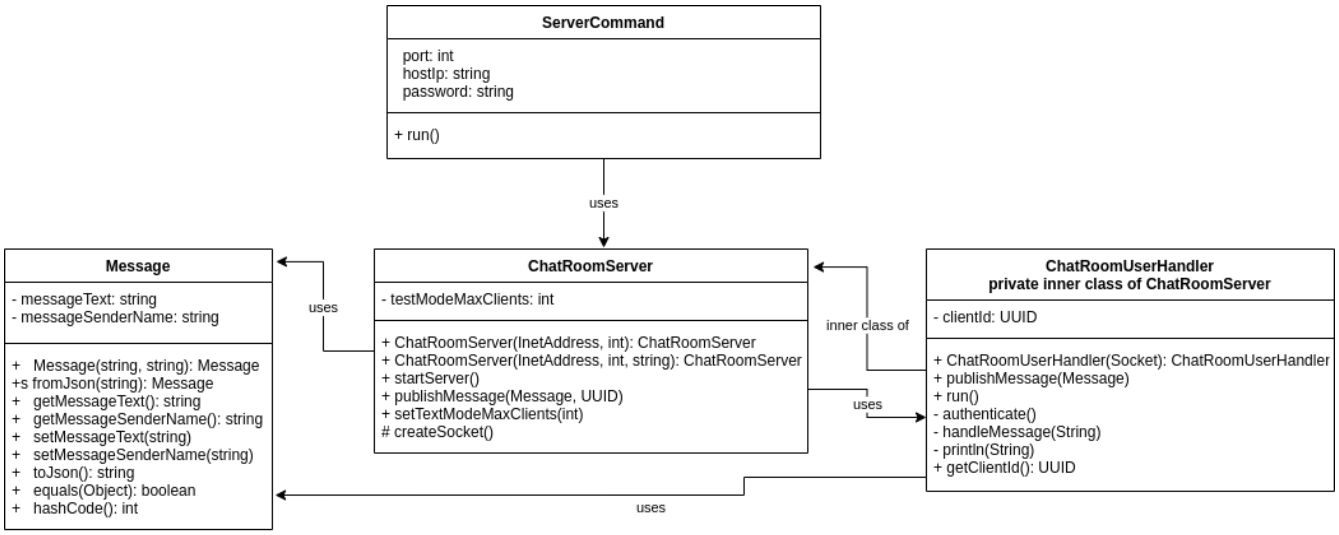
Die eigentliche Anwendung besitzt im Besten Fall einen technologieunabgängigen Kern, welcher Abhängigkeiten als austauschbar betrachtet. Man unterscheidet also zwischen langlebigem Code (dem domänenspezifischen Code) und kurzlebigen, austauschbaren Code (dem “Drumherum”).

Analyse der Dependency Rule (2P)

[1 Klasse, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML der Klasse und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

Positiv-Beispiel: Dependency Rule

Klasse `ChatRoomServer`

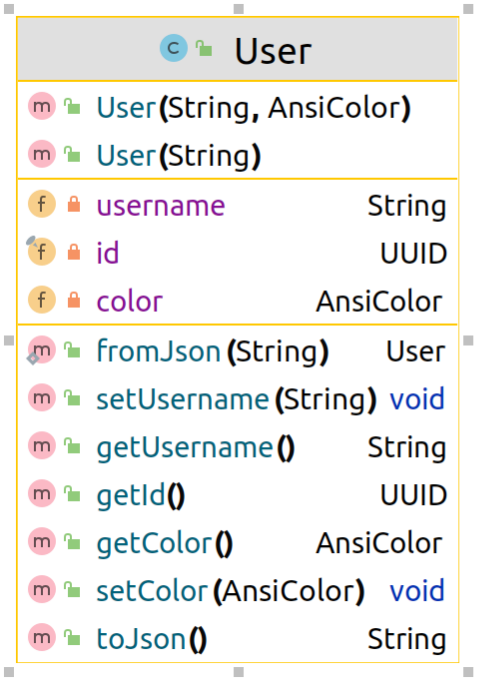


Abhängigkeiten von `ChatRoomServer`:
Die Klasse `ChatRoomServer` ist Bestandteil des innersten Layers der Software-Architektur. Daher sind Abhängigkeiten zu Klassen auf äußeren Schichten zu vermeiden.
Zur strukturierten Verarbeitung von Chat-Nachrichten wird die Klasse `Message` verwendet.
Es werden zudem Inhalte aus java.net, java.io und java.util importiert (nicht im UML zu sehen).
Natürlich ist ChatRoomServer auch von der eigenen Sub-Klasse `ChatRoomUserHandler` abhängig.
Weitere Abhängigkeiten bestehen nicht. Somit bestehen nur Abhängigkeiten zu Hilfsklassen außerhalb der geschichteten Programmarchitektur oder zu Klassen auf selber oder untergeordneter Ebene.

Andere Klassen, die von `ChatRoomServer` abhäging sind:
Die innere Klasse `ChatRoomUserHandler` hängt vollständig von `ChatRoomServer` ab.

Der `ServerCommand` nutzt den `ChatRoomServer` und ist damit von diesem abhängig.

Negativ-Beispiel: Dependency Rule



Die Klasse User ist ein Negativbeispiel der Dependency-Rule. Unter Einhaltung der Richtlinien darf User von keiner Klasse aus darüberliegenden Ebenen abhängig sein. Da es sich um domänen-Code in den “innersten” Schicht handelt, muss User nahezu vollständig unabhängig sein. Lediglich anderer Code in selber Schicht darf verwendet werden.

Das Feld color stellt jedoch ein Problem dar. Im Code sieht die Definition folgendermaßen aus:

```
private AnsiColor color = UserNameHelper.generateColor();
```

Die Klasse User ist somit vom UserNameHelper abhängig. Ohne Vorliegen würde der Programmcode nicht funktionieren. Dies ist eine Verletzung der Dependency Rule, da der UserNameHelper kein Domänencode ist.

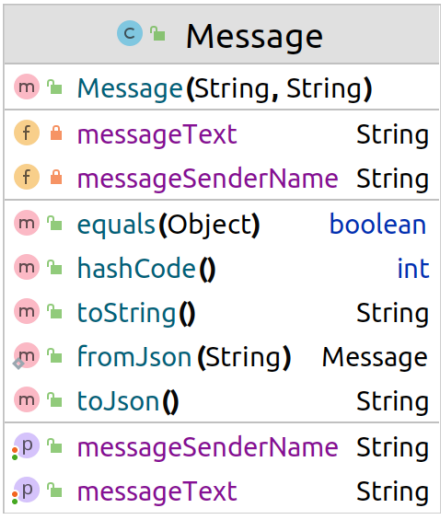
Besser wäre es, das Feld color bei der Initialisierung über den Konstruktor zu befüllen. Das Objekt, welches einen User erstellt müsste die Auswahl der Farbe (möglicherweise mithilfe eines Helpers) übernehmen.

Analyse der Schichten (5P)

[jeweils 1 Klasse zu 2 unterschiedlichen Schichten der Clean-Architecture: jeweils UML der Klasse (ggf. auch zusammenspielenden Klassen), Beschreibung der Aufgabe, Einordnung mit Begründung in die Clean-Architecture]

Schicht: Domain

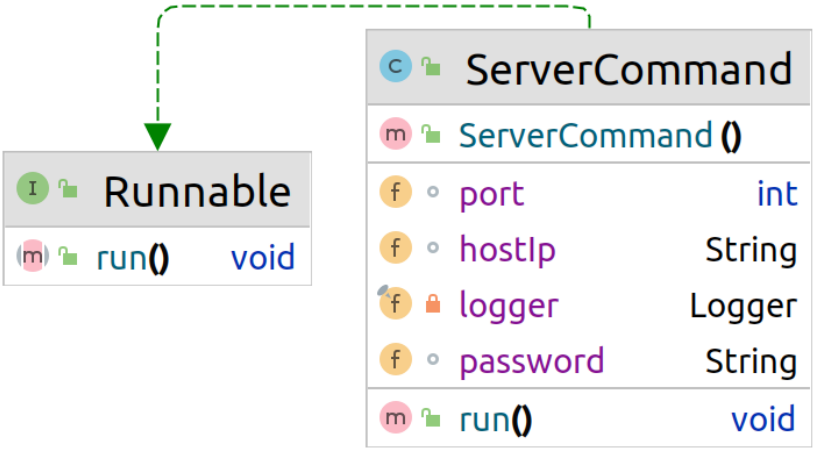
Klasse: Message



Die Klasse Message ist Teil des domänenspezifischen Codes. Im Umfeld einer Chat-Anwendung gehört eine “Nachricht” zum essentiellen Kern der Applikation. Die Klasse besitzt in diesem Fall keine Abhängigkeiten zu äußeren Schichten und ist somit universell einsetzbar, selbst wenn der äußere Code geändert wird.
Aufgabe der Klasse ist die Repräsentierung einer Chatnachricht im Code. So gehörten zu einer solchen Nachricht die Attribute Text und Sender.

Schicht: Presentation

Klasse: ServerCommand



Die Ausgestaltung der Klasse ServerCommand hat direkten Einfluss auf die Bedienung des CLI-Interface. Die mit @Option annotierten Felder werden vom Nutzer direkt verändert. Durch diese direkte “Greifbarkeit” der Klasse, muss sie der Präsentationsschicht zugeschrieben werden.
Aufgabe der Klasse ist die Strukturierung des CLI-Befehls “Server” mitsamt der Argumente. Bei Aufruf wird ein ChatRoomServer erstellt und gestartet.
Im ServerCommand selbst befindet sich keinerlei Geschäftslogik. Diese ist voll in den ChatRoomServer ausgelagert.

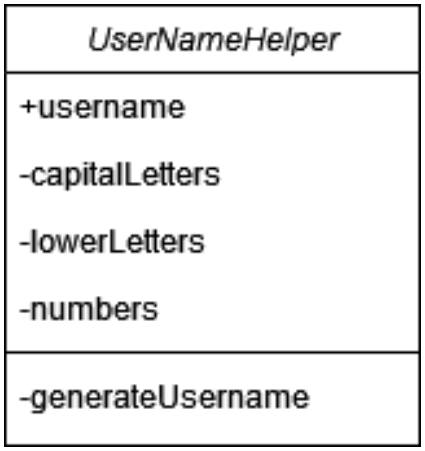
Kapitel 3: SOLID (8P)

Analyse SRP (3P)

[jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML der Klasse und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

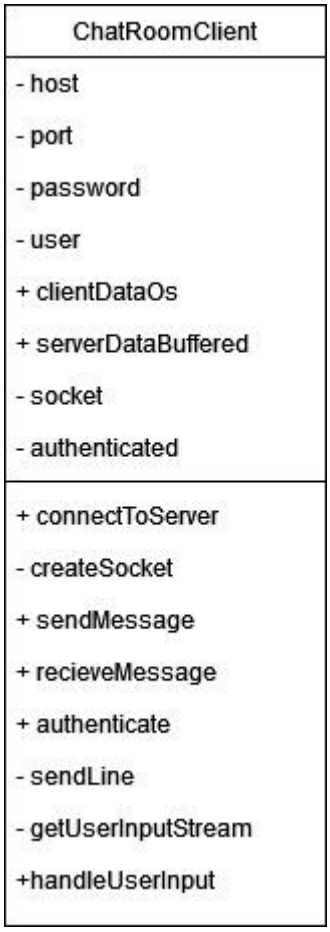
Positiv-Beispiel

Die Klasse `UserNameHelper` generiert einen Usernamen, für den Fall, dass beim Starten der Client Anwendung keiner Angegeben worden ist. Der generiert Username besteht aus einem Großbuchstaben, zwei Kleinbuchstaben und zwei Zahlen.

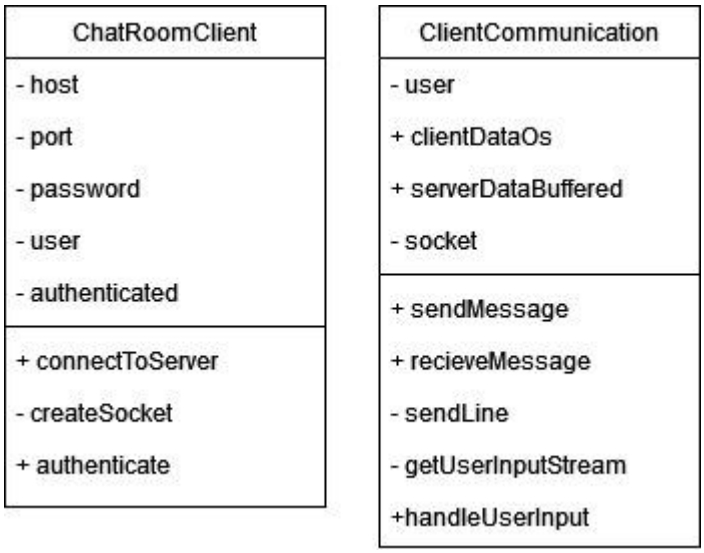


Negativ-Beispiel

Die Klasse *ChatRoomClient* hat nicht nur die Aufgabe den Clienten aufzusetzen, also ihn zu erstellen und mit dem Server zu verbinden, sondern ist auch dafür da die Kommunikation des Users über den Clienten zu verarbeiten, also die Nachrichten des Servers entgegenzunehmen und für den User sichtbar abzubilden oder direkt zu verarbeiten und die Eingaben des Users nach JSon zu formatieren und an den Server zu schicken.



Ein möglicher Lösungsweg wäre die Funktionen, die zur Erstellung und Verbindung des Clients dienen in der Klasse *ChatRoomClient* zu lassen und alle Funktionen, die dazu da sind die Kommunikation zu gewährleisten, in eine neue Klasse, bspw. *ClientCommunication*, auszulagern.



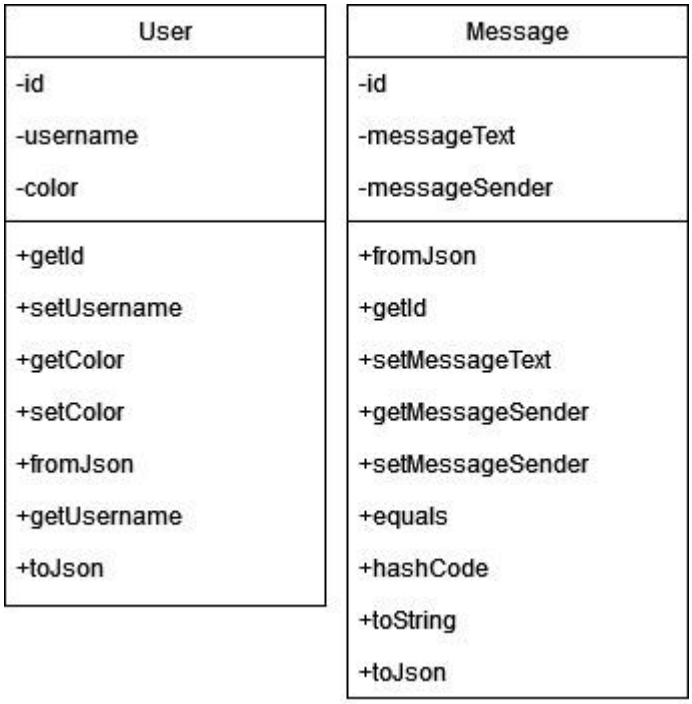
Analyse OCP (3P)

\\offen für erweiterungen geschlossen für Änderungen
[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML der Klasse und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?]

Positiv-Beispiel

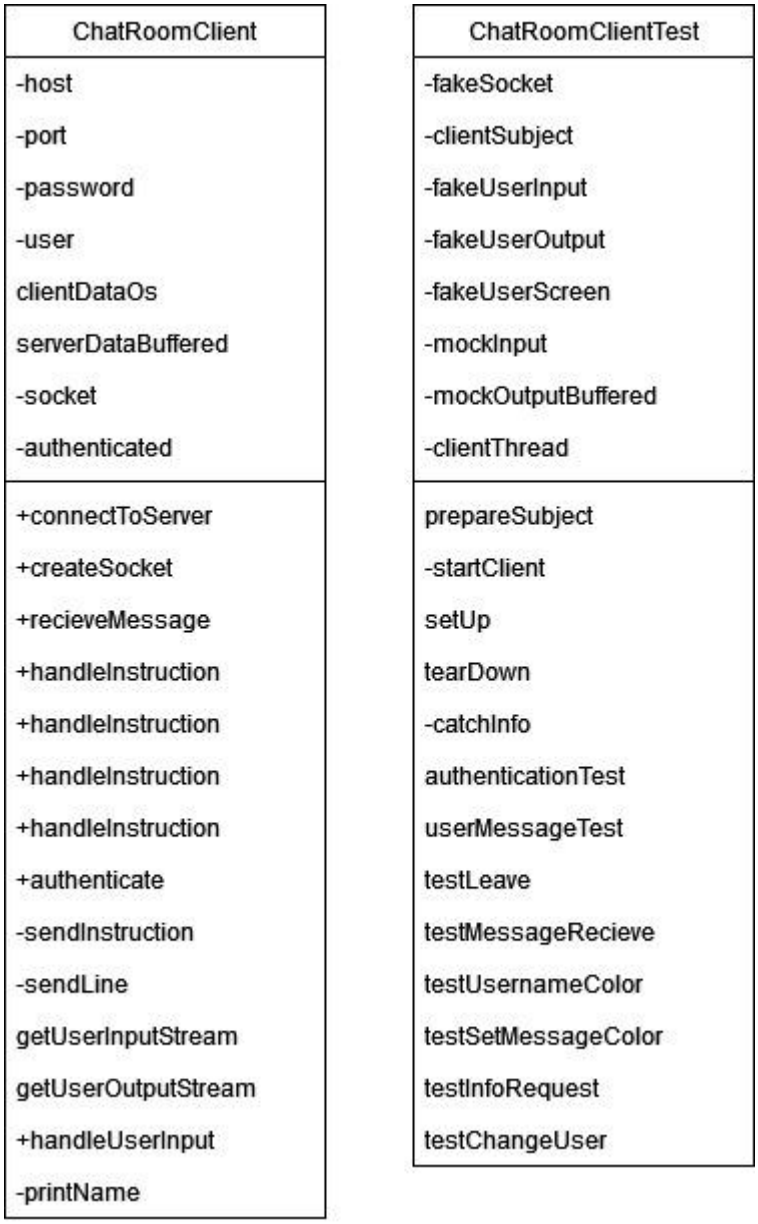
Die Klasse Message verwendet ein Objekt der Klasse User. Allerdings müssen, wenn in der Klasse User Veränderungen vorgenommen werden keine Anpassungen in der Klasse Message vorgenommen werden, da diese das Objekt lediglich in ihre Methoden einbindet, ansonsten aber keinerlei Veränderungen am Objekt vornimmt. Das Objekt dient nur zur weiteren Information.

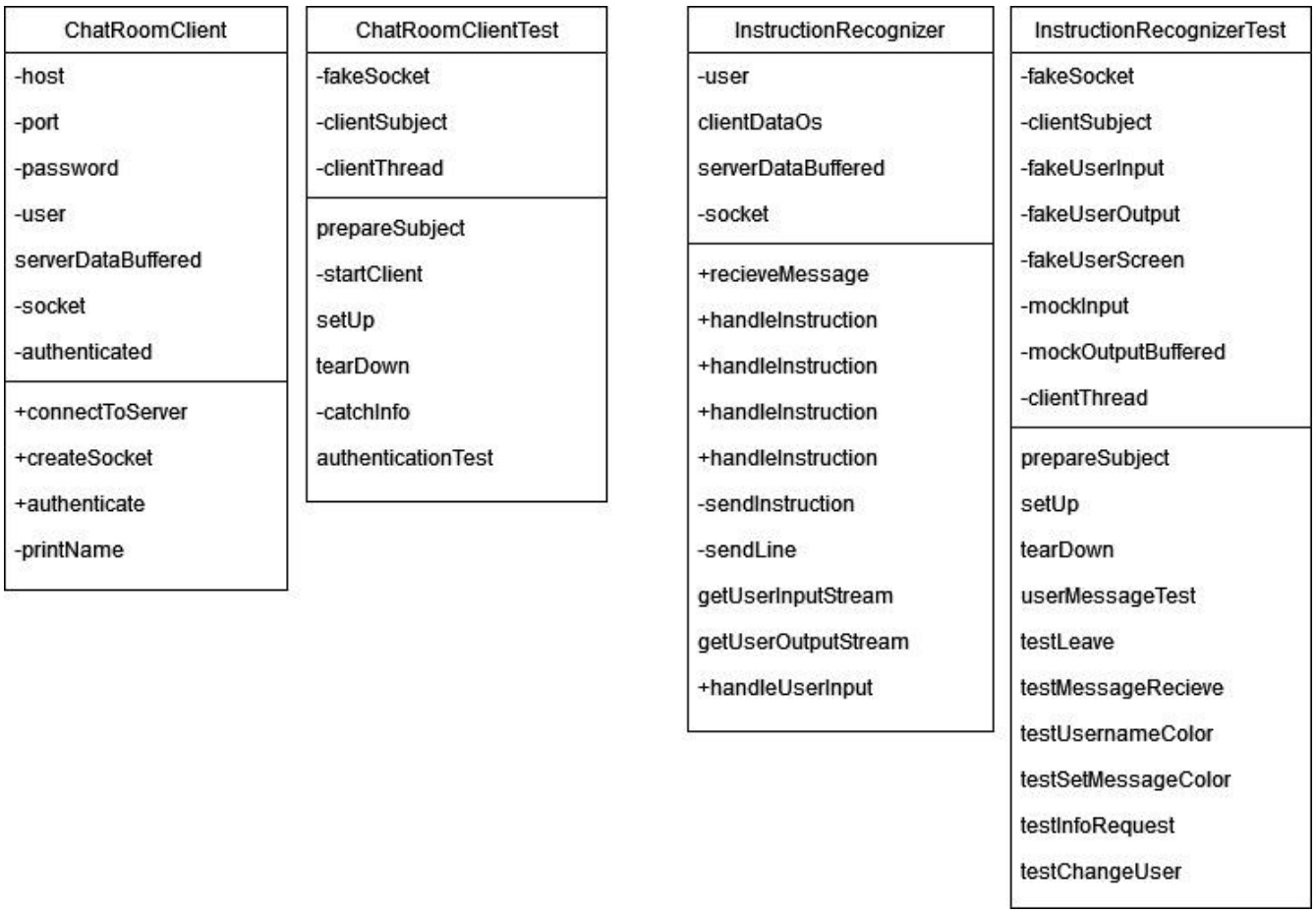
Die Klasse message hat ein Objekt der Klasse User, um es als Informations zugewinn für weitere Klasse zu verwenden. Die Klasse Messag muss nicht wissen, welchen Inhalt das Objekt der Klasse User hat, weil das für sie nicht relevant ist.



Negativ-Beispiel

Von der Klasse ChatRoomClient hängt die Testklasse ChatRoomClientTest ab. Jedes mal, wenn ein neuer Befehl zur Klasse ChatRoomClient hinzugefügt wird, müssen die Tests für die Befehle angepasst werden. Das kann umgangen werden, indem zum Einen die befehle in eigenständige Klassen ausgelagert werden, was bereits geschehen ist, allerdings wird die Erkennung der Befehle immer noch in der Kasse ChatRoomClient durchgeführt. Diese Erkennung kann ebenfalls in eine andere Klasse ausgelagert werden. Dann würde bei einer Einführung neuer Befehle weder die Klasse ChatRoomClient noch die Klasse ChatRoomClientTest angepasst werden müssen.




ChatRoomClient
-host
-port
-password
-user
serverDataBuffered
-socket
-authenticated
+connectToServer
+createSocket
+authenticate
-printName
ChatRoomClientTest
-fakeSocket
-clientSubject
-clientThread
prepareSubject
-startClient
setUp
tearDown
-catchInfo
authenticationTest
InstructionRecognizer
-user
clientDataOs
serverDataBuffered
-socket
+recieveMessage
+handleInstruction
+handleInstruction
+handleInstruction
+handleInstruction
-sendInstruction
-sendLine
getUserInputStream
getUserOutputStream
+handleUserInput
InstructionRecognizerTest
-fakeSocket
-clientSubject
-fakeUserInput
-fakeUserOutput
-fakeUserScreen
-mockInput
-mockOutputBuffered
-clientThread
prepareSubject
setUp
tearDown
userMessageTest
testLeave
testMessageRecieve
testUsernameColor
testSetMessageColor
testInfoRequest
testChangeUser

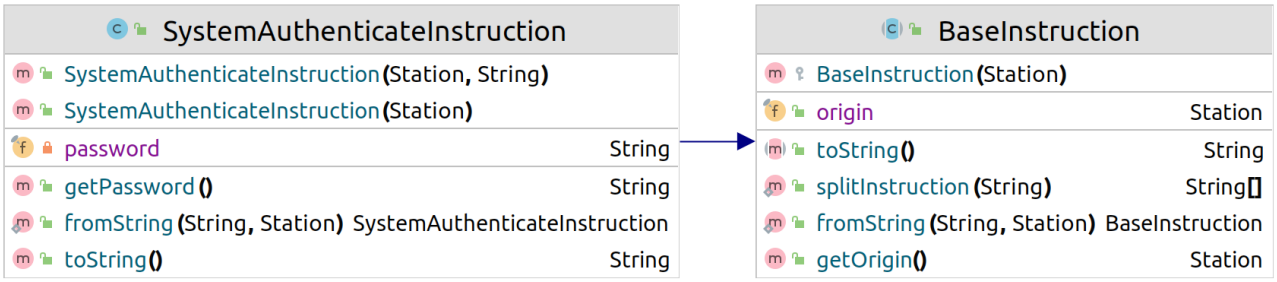
Analyse [LSP/ISP/DIP] (2P)

[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP); jeweils UML der Klasse und Begründung, warum man hier das Prinzip erfüllt/nicht erfüllt wird]

[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

Positiv-Beispiel

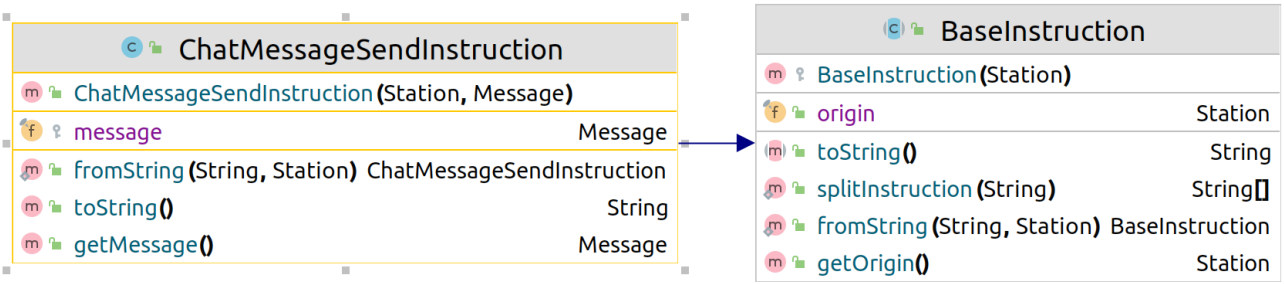
Abgeleitete Klasse: SystemAuthenticateInstruction
Stammklasse: BaseInstruction



Die Klasse SystemAuthenticateInstruction ist ein vollwertiger Abkömmling der BaseInstruction. Erwartet eine Methode eine Instanz der Basisklasse, so entstehen keine Nachteile, wenn stattdessen eine Instanz der “Spezialklasse” SystemAuthenticateInstruction verwendet wird. Alle in der Stammklasse aufgeführten Methoden sind aufrufbar. Auch die Erstellung des Objektes kann in selber Weise erfolgen, da ein identischer Konstruktor angeboten wird.

Negativ-Beispiel

Abgeleitete Klasse: ChatMessageSendInstruction
Stammklasse: BaseInstruction



Die abgeleitete Klasse bietet grundlegend die selbe Funktionalität wie die Stammklasse. Unterschiede ergeben sich jedoch im Konstruktor: So wird für die hypothetische Erstellung eines BaseInstruction-Objekts nur die Station benötigt. Eine ChatMessageSendInstruction benötigt jedoch zusätzlich noch die zu Übermittelnde Nachricht (Message).
Daher ist es nicht möglich die beiden Klassen vollumfänglich miteinander auszutauschen.

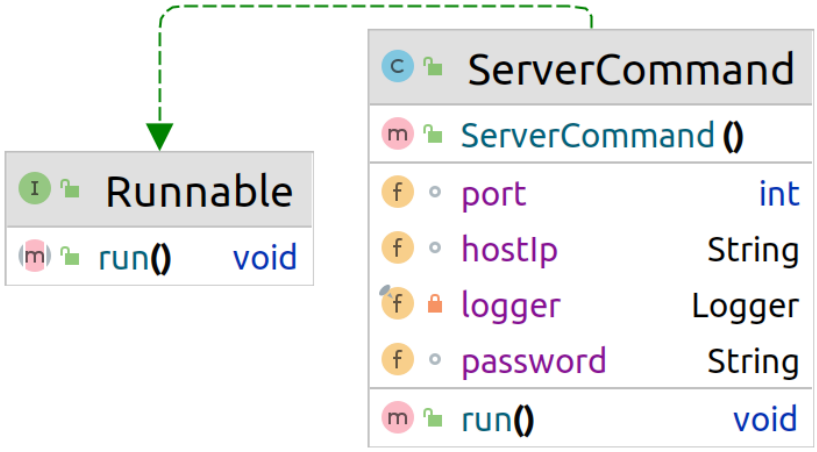
Kapitel 4: Weitere Prinzipien (8P)

Analyse GRASP: Geringe Kopplung (4P)

[jeweils eine bis jetzt noch nicht behandelte Klasse als positives und negatives Beispiel geringer Kopplung; jeweils UML Diagramm mit zusammenspielenden Klassen, Aufgabenbeschreibung der Klasse und Begründungwarum hier eine geringe Kopplung vorliegt bzw. Beschreibung, wie die Kopplung aufgelöst werden kann]

Positiv-Beispiel

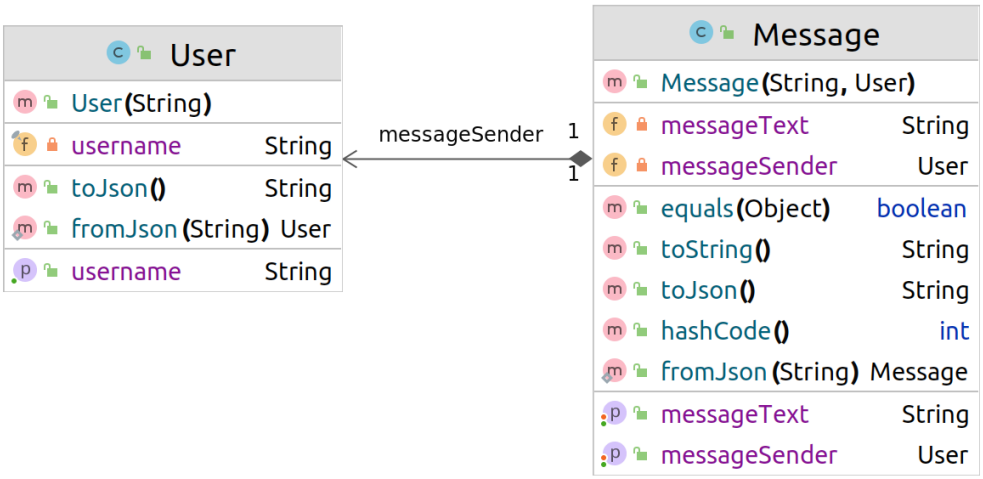
Klasse: ServerCommand und ChatManager



Die Klasse ServerCommand implementiert das Interface Runnable. Die Klasse ChatManager referenziert auf diesen ServerCommand als Sub-Command.
Für Picocli-Subcommands können alle Klassen verwendet werden, die Runnable (ohne Rückgabewert) oder Callable<T> (mit Rückgabewert) implementieren.

Negativ-Beispiel

Klasse: Message und User



Die Klasse Message ist stark von der Klasse User abhängig. Das Feld messageSender referenziert auf eine User-Instanz. Wird später eine Differenzierung der Benutzer vorgenommen, wäre eine Implementierung eines User-Interface (oder einer abstrakten Klasse) sinnvoller. Von dieser könnten User-Derivate abgeleitet werden. Für die Klasse Message würde sich die API nicht ändern.

Analyse GRASP: Hohe Kohäsion (2P)

[eine Klasse als positives Beispiel hoher Kohäsion; UML Diagramm und Begründung, warum die Kohäsion hoch ist]

In der Klasse Message gibt es drei Instanzvariablen. Zwei davon, messageText und MessageSender, werden zusammen in mehreren Methode verwendet. Nämlich in den Methoden hashCode, equals und toString. Dadurch ergibt sich eine Hohe Kohäsion, da zwei Drittel der Variablen in derselben Methode verwendet werden. Da die letzte Variable nicht gleichzeitig in derselben methode verwendet wird ergibt sich allerdings keine maximale Kohäsion.

DRY (2P)

[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher); begründen und Auswirkung beschreiben]

Die Auswirkungen einer solchen Aufhebung von Doppelter Logik führt dazu, dass bei einer Änderung und/oder beim Bugfixing nur noch eine Stelle im Code angepasst werden muss anstatt von einer. Da außerdem nur eine Stelle im Code entwickelt werden muss, ist der Code weniger anfällig für Bugs. Durch das Auflösen der doppelten Logik wird das Bugfixing außerdem weiter vereinfacht, da nun klar ist, an welcher Stelle im Code nach dem Fehler gesucht werden muss, anstatt an zwei möglichen Ursprüngen.

Kapitel 5: Unit Tests (8P)

10 Unit Tests (2P)

[Nennung von 10 Unit-Tests und Beschreibung, was getestet wird]

Unit Test	Beschreibung
Klasse#Methode	
ChatRoomServerTest #serverMessagingTest	Es wird getestet, dass die von einem simulierten Client versendeten Nachrichten, bei einem anderen Simulierten Client ankommen und umgekehrt. Als “Beifang” wird auch der eigentliche Verbindungsaufbau getestet.
ChatRoomServerTest #serverPasswordProtectionTest	Testet den Passwortschutz eines Chatraumes mittels zwei simulierten Clients.
ConfigHelperTest #testDefaultConfig	Prüft, ob beim Erstellen einer neuen Konfigurationsdatei die richtigen Standardwerte vorliegen.
ConfigHelperTest #testDummyConfig	Prüft, ob vorhandene Konfigurationsdateien richtig eingelesen werden.
ConfigHelperTest #readyCheck	Prüft, ob der ConfigHelper vor Initialisierung isReady = false und nach Initilaisierung isReady = true zurückgibt.
UserNameHelperTest #testUserNameGeneratorProducesUniqueNames	Prüft anhand einer Stichprobe von 1000 Benutzernamen, ob Duplikate generiert werden.
HashingHelperTest #testHashing	Ein String wird mithilfe des HashingHelper gehasht. Es wird geprüft, ob der Helper den String hinterher erfolgreich verifizieren kann.
HashingHelperTest #testVerification	Vorgefertigte Hash-Secret-Paare werden mithilfe des Hashing-helpers getestet.
UserNameHelperTest #testUserNameNotEmpty	Testet ob die generierten Usernames nicht leer sind
UserNameHelperTest #testUserNameBeginning	Testet ob ein generierter Usernam mit “User_” beginnt

ATRIP: Automatic (1P)

[Begründung/Erläuterung, wie 'Automatic' realisiert wurde]

Beim Durchlaufen aller Tests müssen keine zusätzlichen Daten oder Werte eingegeben werden, diese sind entweder von vornerein im Test enthalten oder werden währenddessen erzeugt.

Zum Durchlaufen eines Tests muss die Testklasse lediglich gestartet werden. Es müssen keine weiteren Vorkehrungen getroffen werden.

ATRIP: Thorough (1P)

[Code Coverage im Projekt analysieren und begründen]

Eine Testklasse testet immer nur die Funktionen einer anderen Klasse. Dadurch ist eine Übersicht dessen gegeben, was alles bereits getestet wird.
Die wichtigsten Funktionen eines Server-Client Chats, ist die Kommunikation zwischen Server und Client. Diese wird aus beiden Richtungen getestet, durch die Klassen *ChatRoomServerTest* und *ChatRoomClientTest*. Ein weiterer wichtiger Aspekt ist, dass die Funktionen zur Unterstützung des Aufbau des Chats funktionieren, damit der Nutzer das programm einfach bedienen kann. Dies wird durch die vier Klassen *ConfigHelperTest*, *HashingHelperTest*, *InstructionNameHelperTest* und *UserNameHelperTest* gewährleistet.

ATRIP: Professional (1P)

[jeweils 1 positves und negatives Beispiele zu 'Professional'; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell an den Beispielen ist]

Positiv-Beispiel

Die Klasse *InstructionNameHelperTest* besitzt für jede Funktionalität eine eigene Methode. Die Funktionalität besteht aus zwei unterschiedlichen Tests. Es müssen keine zusätzlichen Ressourcen erzeugt oder angefordert werden. Daher ergeben sich zwei Methoden als ausreichende Anzahl. Die erste Methode testet, ob durch den Namen der Klasse der richtige Name der Instruktion erzeugt werden kann. Die zweite Methode testet, ob dies umgekehrt funktioniert, also ob aus dem Namen der Instruktion der richtige Klassenname folgt.

```
void testClassToName() {
        assertEquals("system:ready",
InstructionNameHelper.getNameForInstruction(SystemReadyInstruction.class));
        assertEquals("system:authenticate",
InstructionNameHelper.getNameForInstruction(SystemAuthenticateInstruction.class));

void testNameToClass() throws Exception {
```

```
        assertEquals(ChatLeaveinstruction.class,
InstructionNameHelper.parseInstruction("chat:leave", Station.CLIENT).getClass());
        assertEquals(SystemReadyInstruction.class,
InstructionNameHelper.parseInstruction("system:ready", Station.SERVER).getClass());
        assertEquals(SystemErrorInstruction.class,
InstructionNameHelper.parseInstruction("system:error=parsing",
Station.SERVER).getClass());}
```

Negativ-Beispiel

Die Testklasse *UserNameHelperTest* testet in einer Methode, ob die von der Klasse *UserNameHelper* generierten Strings nicht leer sind. In einer anderen Methode der Testklasse wird jedoch bereits getestet, ob die generierten Strings ungleich sind, also ob die Erstellung von zufälligen Werten funktioniert und am Ende nicht mehrere User mit demselben zufälligen Username erstellt werden würden. Da die Strings ungleich sind wird indirekt bereits getestet, ob diese nicht leer sind, denn wenn zwei Strings leer wären, wären sie gleich. Dadurch ergibt sich doppelter Code, der im Produktivcode so nicht vorkommen soll.

Methode, die die Strings auf Leere prüft:

```
void testUserNameNotEmpty() {
        assertFalse(UserNameHelper.generateUserName().isEmpty());}
```

Methode, die die Strings auf Gleichheit prüft:

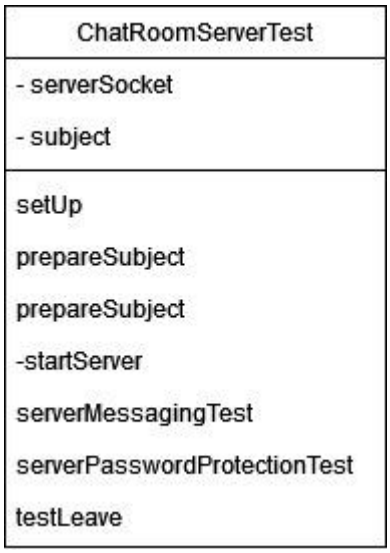
```
void testUserNameGeneratorProducesUniqueName() {
        Set<String> userNames = new HashSet<>();
        for(int i = 0; i<1000; i++) {
                usernames.add(UserNameHelper.generateUserName());}
        assertEquals(1000, userNames.Size());}
```

Fakes und Mocks (3P)

[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten; zusätzlich jeweils UML Diagramm der Klasse]

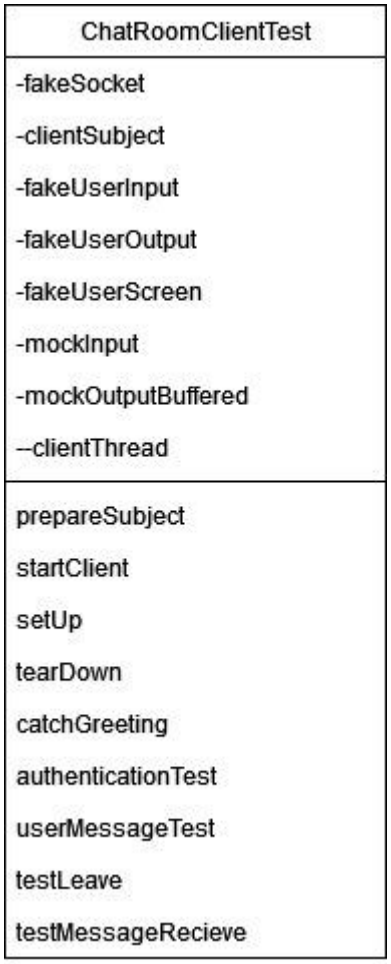
Mock-Objekt 1

Ein eingesetztes Mock-Objekt dient zum simulieren eines *ServerSocket*. Dieser befindet sich in der Klasse *ChatRoomServerTest*, welche zum Testen der Klasse *ChatRoomServer* dient. Der simulierte ServerSocket stellt lediglich die Funktion zur Verfügung, dass durch ihne ClientSockets erstellt werden können, über die eine simulierte Kommunikation mit verbundenen Clients getestet werden kann. Allerdings können ohne einen ServerSocket keine ClientSockets zur Verfügung gestellt werden und somit auch keine Tests zur Kommunikation in diesem Sinne stattfinden.



Mock-Objekt 2

Ein weiteres Mock-Objekt ist ein einfacher *Socket*. Dieser befindet sich in der Klasse *ChatRoomClientTest*, welche dazu dient die Klasse *ChatRoomClient* zu testen. Der Socket simuliert die Verbindung zum Server. Daher kann die Kommunikation des Clients mit dem Server von der Seite des Clients aus getestet werden.



Kapitel 6: Domain Driven Design (8P)

Ubiquitous Language (2P)

[4 Beispiele für die Ubiquitous Language; jeweils Bezeichung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört]

Bezeichung	Bedeutung	Begründung
Message	Inhalt einer Nachricht, die vom Nutzer versendet wird	Bezeichnet einheitlich alle Inhalte von Nachrichten, die vom Nutzer versendet werden und an alle mit dem Server verbundenen Clients weitergeleitet wird
Instruction	Instruktion einer Nachricht, die an den Server bzw. den Client geht	Bezeichnet einheitlich alle Instruktionen von Nachrichten, die vom Server an einen Client oder umgekehrt, gesendet werden. So z.B. die Aufforderung zur Authentifizierung.
Server	Server, der einen Chat verwaltet	Bezeichnet einheitlich den Knotenpunkt und Verwalter eines Chats
Client	Teil der Anwendung, die auf der Seite des Nutzers(Users) gestartet wird und die Kommunikation mit dem Server übernimmt	Bezeichnet einheitlich alle Aspekte des Teils der Anwendung, die vom Nutzer(User) dazu verwendet wird mit dem Server zu kommunizieren

Repositories (1,5P)

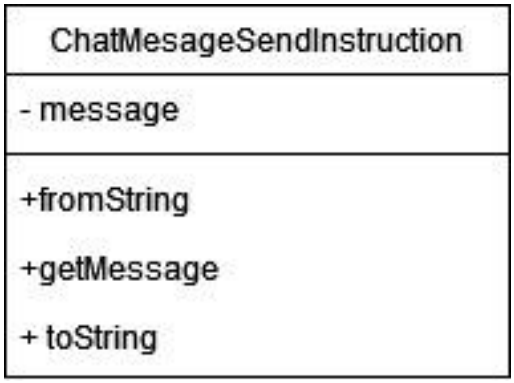
[UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

Wir haben uns für ein Chatprogramm ohne Speicherung der Nachrichten entschieden, da zum Einen keinen Datenbanken verwendet werden sollen und zum Anderen die Speicherung der Nachrichten als nicht sinnvoll erachtet worden ist. Dies folgt daraus, dass das Chatprogramm als Unterhaltungsmöglichkeit für Personen, die sich im selben Netzwerk befinden, sich aber aus anderweitigen Gründen nicht persönlich unterhalten können gedacht. Der Inhalt des Chats soll auch nur von Personen gesehen werden, die zum Zeitpunkt des Verschickens der Nachricht mit dem Server verbunden sind. Eine Speicherung auf Seiten des Servers ist also nicht sinnvoll. Eine Speicherung auf Seiten des Clients wurde ebenfalls nicht umgesetzt, da die Notwendigkeit hierfür nicht gesehen wurde und der Speicher auf dem Endgerät des Clients dadurch nur unnötig belegt werden würde.
Es werden zwar Entities, Value Objects und Aggregates im Laufe des Programmes erzeugt, da es aber keine Persistierung von Daten gibt, gibt es auch keine Repositories, um die Logiken dahinter zu Kapseln.

Aggregates (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

Als Aggregate kann die *ChatMessageSendInstruction* angesehen werden, da diese ein Value Object ist, welches eine Entity, eine Message, enthält. Dies wurde hier umgesetzt, da somit der Inhalt einer Nachricht über die Unveränderbare Hülle der ChatMessageSendInstruction sicherer vor Veränderungen durch Dritte versendet werden kann.



Entities (1,5P)

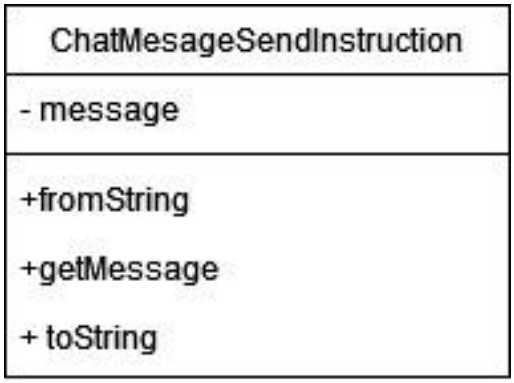
[UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

Als Entity könne Objekte der Klasse User angesehen werden, da dieser eine eindeutige, unveränderbare ID besitzt und andere Aspekte des Users können sich während seiner Lebenszeit verändern. Es kann beispielsweise der Username geändert werden. Dadurch begründet sich auch der Einsatz einer Entity, da der Nutzer sämtliche Erkennungsmerkmale ändern kann, aber durch die ID kann dennoch nachvollzogen werden, wer die Nachricht versendet hat.

Value Objects (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Value Objects; falls kein Value Object vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

Als Value Objects können Objekte der Instruction Klasse angesehen werden, da diese durch ihren Namen und Wert beschrieben werden, aber keine eigene Identität besitzen. Außerdem sind die Objekte unveränderlich. Die unveränderlichkeit ergibt sich aus Sicherheitsgründen, da dadurch eine vom Server bzw. Client erstellte Instruction nicht durch einen Man-in-the-Middle Angriffe abgeändert werden können.



Kapitel 7: Refactoring (8P)

Code Smells (2P)

[jeweils 1 Code-Beispiel zu 2 unterschiedlichen Code Smells aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]

1. Duplicated Code:

Klasse: ChatRoomClientTest

Commit:https://github.com/BjarneRest/ase-chat/commit/2bd1c14586abbfeab4befe91ddac8c6aa244b861

Der authentifizierte Client wird vom Server mit einem “system:ready”/SystemReadyInstruction über die erfolgreiche Authentifizierung informiert.

Da wir im Client mehrere Funktionalitäten testen wollen, die erst nach Erreichen dieses State möglich sind, ist es nötig für jeden dieser Tests den “system:ready”-Handshake zu simulieren. Dieser Code fand sich somit dupliziert in mehreren Tests wieder (Zeile 120-131 und 153-164 im Commit 2835cde8).

Der duplizierte Code wurde in eine Methode catchGreeting() ausgelagert. Diese wird an den vorherigen Stellen aufgerufen.

2.Large Class

Die Klasse *ConfigHelper* Hat mit Leerzeilen einen Umfang von über 100 Zeilen. Zudem besitzt sie sieben Methoden, was Ihr einen großen Umfang an Aufgaben gibt.

```
public class ConfigHelper {
	private ConfigHelper() {}
	public static ConfigHelper getInstance() {}
	public static ConfigHelper forceNewInstance() {}
	public boolean isReady() {}
	public void initTemp() {}
	public void init() {}
	private Configuration generateDefaultConfigFile() {}
	public Configuration getConfig() {}
```

Ein Möglicher Lösungsansatz wäre einen Teil der Methoden in eine neue Klasse auszulagern. So könnten die Methoden *isReady*, *initTemp* und *init* in eine Klasse *ConfigCreator* ausgelagert werden, die dazu da ist die Erstellung einer Konfiguration umzusetzen, während die verbliebenen Methoden lediglich zur Unterstützung dienen. So hätten beide Klasse nur drei bzw. vier Methoden und damit auch einen eingeschränkten Zuständigkeitsbereich.

```
public class ConfigHelper {
	private ConfigHelper() {}
	public static ConfigHelper getInstance() {}
	public static ConfigHelper forceNewInstance() {}
	private Configuration generateDefaultConfigFile() {}
	public Configuration getConfig() {}

public class ConfigCreator {
	private ConfigCreator() {}
	public boolean isReady() {}
	public void initTemp() {}
	public void init() {}
```

2 Refactorings (6P)

[2 unterschiedliche Refactorings aus der Vorlesung anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen]

1. Replace Conditional with Polymorphism

Commit:
https://github.com/BjarneRest/ase-chat/commit/a225c923171f23a7a3cb87b48ba7ab1b6af59317

In der Klasse *ChatRoomClient* gibt es ein if-else-Konstrukt, welches eine Instruktion nach ihrem Typen filtert und dementsprechend Code ausführt. Dies wurde durch polymorphe Methoden *handleInstruction*, welches je nach Instruktions Typ anderen Code ausführt. Allerdings wird trotzdem ein If-else-Konstrukt benötigt, um im vorhinein die *BaseInstruction* ihrem richtigen Typ zuzuordnen.

2. Extract Method

In der Klasse *ChatRoomClientTest* gibt es eine Methode *authenticationTest*, die mehr als eine Aufgabe hat und lang geraten ist. Um die Methode einfacher verständlich zu machen wurden die Funktionalitäten in unterschiedliche Methoden ausgelagert. Eine Methode zum Aufsetzen der Testumgebung, eine Methode zum Schließen der Testumgebung und eine Methode, die den Test durchführt.
Commit: Refactor ChatRoomClientTest

Methode authentcationTest alt:

```
void authenticationTest() throws Exception {
        //Set up test environment
        Mockito.reset(faeSocket);
        prepareSubject();

        //Fake i/o
        final pipedOutputStream pipedOutoutStream = new PipedOutputStream();
        final PipedInputStream pipedInputStream = new PipedInputStream();
        Mockito.when(fakeSocket.getOutputStream()).thenReturn(pipedOutputStream);
        Mockito.when(fakeSocket.getInputStream()).thenReturn(pipedInputStream);

        final PipedInputStream mockOutput = new PipedInputStream();
        pipedOutputStream.connect(mockOutput);
        final BufferedReader mockOutputBuffered = new BufferedReader(new
InputStreamReader(mockOutput));

        final PipedOutputStream mockInput = new PipedOutputStream();
        pipedInputStream.connect(mockInput);

        Thread clientThread = startClient();

        mockInput.write("system:authenticate\n".getBytes(StandardCharsets.UTF_8));
        await().atMost(Duration.ofSeconds(2)).until(mockOutputBuffered::ready);
        String line = mockOutputBuffered.readLine();

        assertEquals("system:authenticate=password", line);

        clientThread.interrupt();}
```

Neue Methoden:

```
void setUp() throws Exception {
        Mokito.reset(fakeSocket);
        prepareSubject();

        final pipedOutputStream pipedOutoutStream = new PipedOutputStream();
```

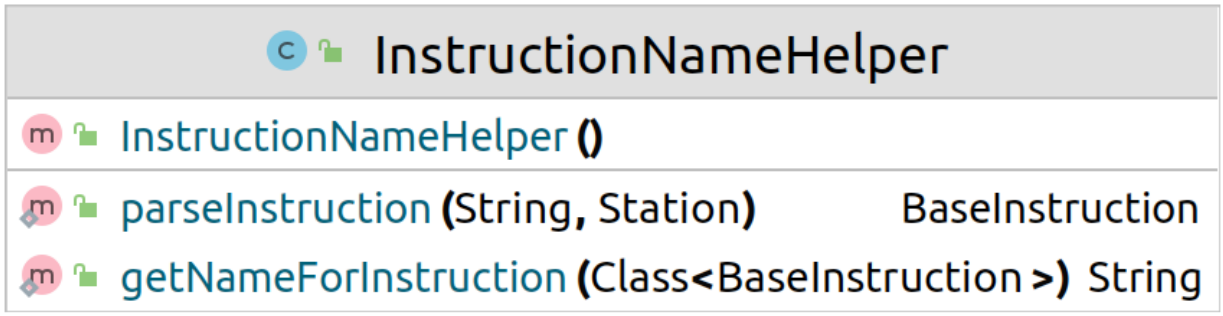
```
        final PipedInputStream pipedInputStream = new PipedInputStream();
        Mockito.when(fakeSocket.getOutputStream()).thenReturn(pipedOutputStream);
        Mockito.when(fakeSocket.getInputStream()).thenReturn(pipedInputStream);

        final PipedInputStream mockOutput = new PipedInputStream();
        pipedOutputStream.connect(mockOutput);
        mockOutputBuffered = new BufferedReader(new InputStreamReader(mockOutput));

        mockInput = new PipedOutputStream();
        pipedInputStream.connect(mockInput);

        clientThread = startClient();}

void tearDown() {
        clientThread.interrupt();}

void authenticationTest() throws Exception {
        mockInput.write("system:authenticate\n".getBytes(StandardCharsets.UTF_8));
        await().atMost(Duration.ofSeconds(2)).until(mockOutputBuffered::ready);
        String line = mockOutputBuffered.readLine();
        assertEquals("system:authenticate=password", line);}
```

Kapitel 8: Entwurfsmuster (8P)

[2 unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils sinnvoll einsetzen, begründen und UML-Diagramm]

Entwurfsmuster: Factory-Method (4P)

Methode: InstructionNameHelper#parseInstruction



Die Instructions werden zwischen Server und Clients in strukturierter Textform ausgetauscht. Beispiele:

1. system:authenticate=SehrGeheimesPassword
2. chat:message:send={"messageText":"GutenMorgen","messageSender":{"id":"d13745b0-c252-45a8-80c8-7efe254e12e2","username":"Heinz","color":"CYAN"}}

Diese Instruktionen müssen zur besseren Verarbeitung in Java-Objekte umgewandelt werden. Hierfür ist die Methode parseInstruction(String, Station) zuständig.
Die Methode sucht über Java-Reflection nach Subklassen von der Stammklasse BaseInstruction. Wird eine passende Klasse zum Instruction-Text gefunden, so wird diese mit der fromString(String, Station)-Methode instanziiert.
Vor dem Aufruf von parseInstruction(...) steht nicht fest, welcher Objekt-Typ zurückgegeben wird. Lediglich festgestellt werden kann, dass ein Subtyp von BaseInstruction geliefert wird.

Die Methode könnte auch als Interpreter der “Instruction-Sprache” verstanden werden.

Entwurfsmuster: Prototyp (4P)

In der Klasse ChangeUserInstruction gibt es eine Methode copyWithInstance, die eine Instruction kopiert und dabei Sender und Empfänger vertauscht. Also, wenn der Client an den Server eine Instruction sendet den Usernamen zu ändern, dann kann der der Server die gleiche Instruction als Bestätigung an den Client zurücksenden.