Entwickler Dokumentation

Gruppe 31 Entwickler

Juni 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In diesem Dokument begleiten wir, Benjamin und Daniel, Sie durch unser Programm für das Modul Softwaretechnik. Wir beide waren die Hauptentwickler und kennen uns daher am Besten mit den technischen Details aus.

2 Ideen

Wir wollten einen kolaborativen Arbeitsplatz schaffen, auf dem Lehrer und Schüler eines Gymnasiums zusammen die UML lernen und anwenden können.
Um dies umzusetzen, brauchten wir eine Graphische Nutzerschnittstelle und auch einen funktionierenden Backend, der alles managed. Für die GUI entschieden wir uns für Java Swing. Swing ist zwar veraltet und es gibt modernere Varianten, Java GUIs zu erstellen, jedoch war Swing einsteigerfreundlicher. Als Backend entschieden wir uns einen Server aufzusetzen, der mit einer MySQL Datenbank kommuniziert und so über Anwendungssessions hinweg persistent Daten, wie z.B. Nutzer, speichern kann.
Desweiteren wollten wir eine Modularisierung bereitstellen, mit der wir verschiedene Komponenten einfach wie beim Plug and Play integrieren können, und alles funktioniert, wie man es sich wünscht.

3 Client

Der Client ist die zentrale Schnittstelle und entählt alle Klassen die benötigt werden damit dieser gestartet werden kann. Die wichtigsten werden wir folgend nun darstellen und erläutern. Dazu zählt das Graphical User Interface, kurz GUI (siehe 3.1) welche für sämtliche Anzeigemöglichkeiten verantwortlich ist. Desweiteren verwaltet der Client auch Daten eigenständig und synchronisiert sich innerhalb der Logik Klassen (siehe 3.2). Weiterhin stellt er auch Methoden für die Kommunikation zwischen ihm, dem Server und anderen Clients bereit (siehe 3.4).

3.1 GUI

Wir als Gruppe hatten uns ein Hauptziel gesetzt: Ein modernes, minimales und intuitives GUI.
Im GUI finden wir die hauptsächliche Modularität. Wie man sehen wird, kann man GUI Elemente beliebig hin und her tauschen, ohne Funktionalitäten zu verlieren.

3.1.1 Modernität

Für den modernen Aspekt entschieden wir uns ein Java Look and Feel von FlatLaf zu benutzen. Ein Java LAF bietet die Möglichkeit, Swing Applikationen

modern und ästhetisch aussehen zu lassen, ohne viel selber mit z.B. CSS zu machen. Wir entschieden uns für 'Solarized Light':

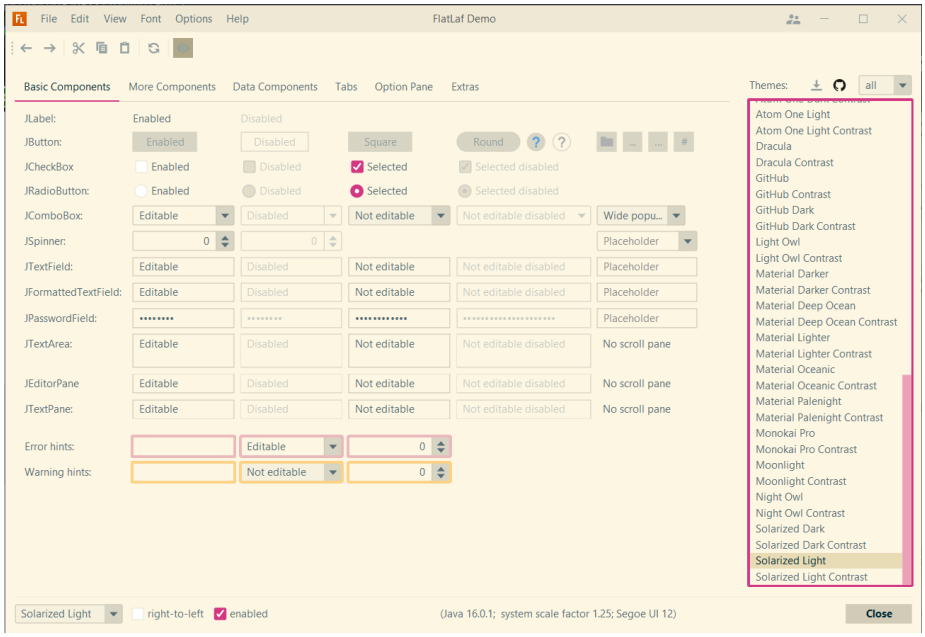

Abbildung 1: FlatLaf's Solarized Light Theme

Außerdem wollten wir nur ein einziges Fenster haben, von dem man alles was man will erreichen kann. Dadurch ergab sich dann natürlich das Problem, wie man zwischen grundlegend verschiedenen Ansichten hin und her navigieren kann, ohne Informationen zu verlieren etc.. Dazu im Abschnitt 3.1.4 mehr.

3.1.2 Minimalität

Für das Minimale wollten wir mit Popups arbeiten. Dadurch konnten wir vieles, was nicht dauerhaft gebraucht wird, verstecken, die Funktionalitäten jedoch immer noch dem Nutzer bereitstellen. Nach unzähligen Iterationen landeten wir bei dem Swing JPopupMenu, was man mit einem JPanel versehen kann und somit ein schwebendes Fenster "aufpoppen" kann.

3.1.3 Intuitivität

Minimalität fordert aber auch die Intuitivität der Anwendung. Es bringt nichts, Funktionen zu verstecken, wenn der Nutzer sie nicht ohne Umstände finden kann.
Daher haben wir viele Schnittpunkte mit dem Nutzer in Form von JButtons

geschaffen. Diese sind mit ihrer Funktionalität beschriftet und beim Betätigen des JButtons wird dann ein Popup mit der Funktionalität erzeugt.

3.1.4 Umsetzung

Wir haben unsere GUI mit dem UI Designer von IntelliJ erschaffen. Das bietet den Vorteil, dass man Klassen erzeugen kann, die verschiedene Sub-GUIs repräsentieren und diese wie beim Plug and Play einfach benutzen kann. Dafür brauchten wir aber ein paar Voraussetzungen:

1. Jedes GUI ist ein JPanel (nur JPanels können z.B. durch ein Popup gezeigt werden)
2. Jedes GUI muss bestimmte Eigenschaften haben (Hat das GUI Buttons die Listener benötigen?, Hat das GUI Popups?,...)
3. Jedes GUI sollte einheitlich benannte Komponenten haben, damit man per Plug and Play tatsächlich nicht sonderlich nachdenken muss
4. Jedes GUI muss sich aktualisieren können und auf Datenbankupdates reagieren können

Und dafür eignet sich am Besten eine Abstrakte Klasse in Java. Diese haben wir auch erzeugt und 'GUI' genannt (vgl. Quellcode). Diese Klasse besitzt nun also abstrakte Methoden (updateGUI, setupListeners), vorimplementierte Methoden (setupPopup[left,right,top,down]) und einen Standardkonstruktor. Diese GUI Klasse bildet also das gemeinsame aller GUIs ab.
Nun haben wir also ganz viele JPanels, brauchen aber noch ein JFrame was alles hält. Dafür benutzen wir einen sogenannten 'Manager', genauer: den 'GUIManager'. Für uns sind Manager Klassen, die grundlegende Funktionalitäten der Anwendung managen (Intuitiv, oder?). Der GUIManager ist also etwas, der alles rund ums GUI managed. Und da wir es für unsinnig hielten, extra noch eine Klasse zu erstellen, die das JFrame repräsentiert, machten wir den GUIManager zu unserem Fenster!
Der GUIManager zeigt dem Nutzer also immer ein bestimmtes GUI. Um genauer zu sein, zeigt der GUIManager dem Nutzer immer ein von drei GUIs, alle anderen GUI Klassen (vgl. Quellcode/Klassendiagramm) sind Popups oder Sub-GUIs von den drei Haupt-GUIs. Der GUIManager löst das Umschalten der Anzeige recht elegant durch simpler Neuvergabe der ContentPane. Dadurch wird der Status von allen anderen Haupt GUIs erhalten und man verliert keine Informationen, wenn der Nutzer zwischen Ihnen hin und her schaltet.
Außerdem hält der GUIManager viele Funktionen, die für die Sub-GUIs oder auch andere außenstehenden Klassen von Wichtigkeit sind. Wieso hält der GUIManager Funktionen für Sub-GUIs? Können diese die Funktionen nicht selber direkt implementieren? Nein! Denn: Wir verwenden eine Baumstruktur bei den GUIs. Jedes GUI kennt nur seine direkt untergeordneten GUIs UND den GUIManager. Dadurch kann es z.B. nicht auf sein Eltern-GUI zugreifen und wäre in vielen Situationen dadurch Handelsunfähig. Gar zu schweigen davon, wenn ein

GUI etwas in einem anderen GUI eines anderen Astes verändern möchte. Da der GUIManager jedoch an der Wurzel des Baumes situiert ist, kann dieser sein Kind, dieses wiederrum sein Kind, etc. ansprechen und somit überall agieren. Wir schicken 'Impulse' durch den Baum. Da nun jedes GUI den GUIManager kennt, kann jedes GUI solche Impulse durch den Baum senden, noch besser: genau dahin, wo es möchte. Diese Baumstruktur spiegelt sich auch im Klassendiagramm wider. Jeder '→' ist immer vom GUIManager weg gerichtet. Das bedeutet, dass der GUIManager alles GUI spezifische kontrolliert. Das Konzept des Manager wird uns noch öfter begegnen.

3.1.5 Popups

Ein zentraler Eckpfeil unseres Programmes sind die Popups. Es gibt natürlich die Möglichkeit für jedes Popup in Swing ein seperates Popup Fenster (JDialog, JOptionPane) zu haben. Das sieht aber nicht modern aus, und ist clunky, wenn man viele Popups gleichzeitig hat. Daher entschieden wir uns dazu, mit Panel Popups zu arbeiten.
Anfangs benutzten wir dazu noch einen eher umständlichen Ansatz. Diesen hatten wir auch im Code Review vorgestellt: Wir benutzen Swing's PopupFactory und Popup Klassen um JPanel-Popups zu erzeugen. Das Problem hierbei war, dass wir so gut wie alles selber implementieren mussten. Sprich Schließung des Popups beim erneuten Klicken, Schließung des Popups, wenn man außerhalb davon klickt, etc.. Es ergab sich, dass man pro Popup ca. 30 Lines of Code benötigte. Also ein eher suboptimaler Ansatz in Bezug auf Einfachheit.
Dann entdeckten wir den heiligen Gral in Swing's Popup Ecosystem: JPopupMenu. Diesem JPopupMenu kann man dann z.B. sagen, ein GUI (JPanel) zu zeigen. JPopupMenus sind eigentlich für (wer hätte es gedacht) Menus gedacht, jedoch kann man sie missbrauchen und sie einfach jedem erdenklichen JComponent zuordnen und 'popuppen' lassen. Denn jedes JComponent (also auch z.B. JButton) hat eine Eigenschaft namens 'ComponentPopup'. Dieser Eigenschaft kann man z.B. ein JPopupMenu zuordnen und dann komfortabel über einen Listener (z.B. ActionListener bei JButton) abrufen. Und das Beste: JPopupMenu regelt die ganze Klicklogik allein! Doppelklick zum Schließen, irgendwo anders klicken zum Schließen, alles von Haus aus implementiert! So war es uns möglich von ca. 30 Lines of Code pro Popup zu gerade einmal 5 Lines of Code zu kommen. Und da das JPopupMenu zu viel alleine erledigt und sich dadurch die Deklarationen alle ähneln, war es möglich, die Methoden zu vereinheitlichen und kollektiviert in der abstrakten GUI Klasse zu haben. Somit sind die meisten Popups nur noch eine line of code. (vgl. z.B. setupPopupBelow() in GUI: erzeugt ein Popup unterhalb eines Referenzbuttons und weist diesem die Popupfunktionalität hinzu).

3.1.6 Modularität

Wie aufgezeigt, können also Popups alle möglichen GUIs beinhalten, man könnte also auch theoretisch eine unendliche GUI-Verschachtelung haben. Dies bezeugt die modulare Arbeitsweise unserer GUI Lösung.

Andere Klassen, wie z.B. die folgenden Manager in 3.2 können auch einfach modular ausgetauscht werden, um verschiedene Funktionalitäten bereitzustellen.

3.2 Logik

Die Logik des Clients umfasst die Informationsverwaltung und Synchronisationsmechanismen von den Userdaten, den Klassenraumdaten, den Gruppendaten und den Sessiondaten. Alle vier Bereiche ähneln sich stark, daher ist es ausreichend wenn wir uns nur den User und den Usermanager genauer anschauen.

Für alle vier Kategorien existieren Objekte mit passenden Attributen. So hat das Userobjekt zum Beispiel eine userid, einen Accounttyp, Vor- und Nachname sowie eine Variable die bestimmt, ob dieser Online oder Offline ist.
Jedes dieser Objekte hat eine Managerklasse welche gemeinsame Eigenschaften von einer übergestellten abstrakten Managerklasse erben, beziehungsweise dessen Funktionalitäten bezogen auf das Objekt für welches sie stehen, implementieren. Die übergestellte Managerklasse nennt sich simpel 'Manager', die daruntergestellten, am Beispiel des Userobjektes, UserManager. Die abstrakte Klasse 'Manager', beinhaltet eine HashMap welche die Ids eines Objektes auf das Objekt selber mapt. Dies ermöglicht uns einen schnellen und effizienten Zugriff auf das Objekt anhand seiner Id. Desweiteren beinhaltet sie einige abstrakte Methoden wie load(int id) und cacheAllData() welche in den Kinderklassen genauer spezifiziert und implementiert sind. Grundlegend ermöglichen sie das Laden und das Aktualisieren aller Informationen in die bereits erwähnte HashMap. Über die Usermanager Klasse kann der Client nun sämtliche Informationen aller registrierten Userobjekte erhalten, die Beispielsweise für die Aktualisierung der GUI benötigt werden. Das bezieht sich auch auf die anderen drei Objekte.

3.3 UML Arbeit

Unsere Hauptaufgabe war das kollaborative Unterrichten der UML Anwendungsspezifikationen. Dafür brauchten wir erstmal zeichenbare UML Objekte, die man beliebig verschieben, bearbeiten und ggf auch speichern kann.
Da wir mit Swing arbeiten und auch unsere UML Objekte in der GUI irgendwie darstellen möchten, lag es nahe, zur Realisierung der Objekte JComponents zu verwenden. Diese kann man nämlich einfach in das GUI einbinden (da es ja Swing Komponenten sind) und außerdem: JComponents sind serialisierbar! Für das Synchronisieren des Arbeitsplatzes, was ja ein fundamentaler Teil des kollaborativen Arbeitens ist, ist dieser Aspekt von allerhöchster Wichtigkeit.
Wir entschieden uns, die Basics der UML Anwendungsspezifikation zu implementieren: Use Case, Actor, Arrow, Includes Arrow, System Box. Damit hat man ein mächtiges Repertoir, um UML Anwendungsfalldiagramme zu erstellen. Da wir eine Bearbeitbarkeit der UML Komponenten fordern (wir möchten unseren Use cases etc. ja Namen geben und sie ggf. skalieren), kann man keine Bildeinbindung verwenden, denn Bilder sind statisch. Also entschieden wir uns dazu, mithilfe von paintComponent zu arbeiten. Diese Methode ist durch

Swing nativ bereitgestellt, um mit einem Graphics Context Dinge zu malen. Bemerkenswert ist die Parameterabhängigkeit der Graphics-Arbeit. Heißt, man kann das gepaintete tatsächlich in der Runtime verändern. Und damit kommen wir zu den Graphics-GUIs (DrawableObjectGUIs). Diese beinhalten je nach UML-Komponenten-Art verschiedene Möglichkeiten, das Objekt zu modifizieren. Besonders fordernd waren hier die Arrow-GUIs, denn einen Pfeil können wir natürlich rotieren, strecken, stauchen, die Pfeilspitze swappen etc.. Da musste ich (Daniel) erstmal meine Trigonometrie-Kenntnisse wieder auffrischen, um z.B. die Pfeilspitze mit rotieren zu lassen (das setzte komplexe Tangens Kalkulationen voraus).
Zusammenfassend haben wir unsere UML Objekte also durch leere JComponents, in denen mithilfe von Java Graphics(2D) gemalt wurde, dargestellt. An diese wurde dann per MouseListener ein Popup GUI rangehangen, welches die Eigenschaften des Objektes modifizieren kann.

3.3.1 Synchronisation der UML Objekte

Folgend greifen wir dem Abschnitt 3.4 'Kommunikation' ein wenig voraus.
Zum Synchroniseren der UML Objekte müssen diese Serialisierbar sein. Das sind sie auch, bis auf ein paar Eigenschaften (Listener, Popupmenu...). Daher müssen diese Eigenschaften mit dem Java keyword 'transient' versehen werden und die Clients müssen demzufolge die Listener etc. wieder initialisieren, da diese bei den übertragenen Objekten fehlen.
Wann wird die Synchronisation getriggert? Wir wollen ja den Server nicht unnötig belasten, beziehungsweise generell das Internet. Daher liegt jedem UML Objekt ein TimerTask zu Grunde, der nur genau dieses Objekt mit anderen Client synchronisieren will. Das aber auch nur wenn eine bestimmte Bedingung gilt: das UML Objekt möchte sich synchronisieren. Wann möchte es sich synchronisieren? Immer genau dann, wenn es verschoben und geklickt (Eigenschaften verändert) wird. Dadurch synchronisieren wir nur genau das jenige Objekt, welches gerade durch den Lehrer z.B. verändert wurde.

3.4 Kommunikation

Um Schüler und Lehrer in einer kollaborativen Umgebung effizient verknüpfen zu können, haben wir uns für eine Client-Server Architektur entschieden. Hierbei werden viele Informationen zwischen Clients und dem Server ausgetauscht. Im Allgemeinen funktioniert der Ablauf der Kommunikation wie folgt.
Der Client lädt eine Kopie der gesamten Informationen über User, Kurse, Gruppen und Sessions beim einloggen. Ändert der Client nun eine Variable dieser Informationen so wird automatisch eine Update Nachricht an den Server übermittelt welcher das zu ändernde in die Datenbank überträgt. Dabei werden auch alle Clients informiert, sich bezüglich dieser Änderung zu aktualisieren. Wir erhalten so eine allgemeine Konsistenz zwischen Clients, Server und der Datenbank.

3.4.1 Rolle des RMI

RMI steht für Remote Method Invocation und erlaubt es uns, Methoden die eigentlich auf dem Server liegen, auch im Client zu benutzen. Damit können wir unter anderem Objekte zum Server übertragen und umgekehrt. Genau darauf basiert unsere Client-Server Kommunikation. Wir haben uns für diese Kommunikationsmethode entschieden da TCP zu viele Probleme mit der Firewall aufgeworfen hat.
Der Client besitzt eine RMI Klasse namens 'RMIClient' welcher sich mit RMI Klasse des Servers verbindet (mehr dazu unter 4.3), sowie eine Interface Klasse 'RMIServerInterface' mit sämtliche Methoden des Servers, welche der Client benutzen kann. Dazu zählen update Methoden für User, Klassen, Sessions und Gruppen sowie für das aktualisieren des Arbeitsplatzes.

3.4.2 Kommunikation zwischen Client und Server

Betrachten wir nun eine detailliertere Darstellung der Client- und Serverkommunikation am Beispiel des Anmeldeprozesses. Grundlegend ist der Client als Offline durch eine Variable in der Datenbank hinterlegt. Meldet dieser sich an, und ist der Anmeldevorgang erfolgreich, so setzt der Client sein eigenes Userobjekt auf Online.
Damit besteht jedoch eine Datendiskrepanz zwischen dem eigenen Client, allen verbundenen Clients und der Datenbank. In allen anderen Clients, dem Server und der Datenbank ist man selber noch als Offline vermerkt. Daher sendet der Client per Java RMI (siehe 3.4.1 oder 4.3) eine update Nachricht mit dem eigenen Userobjekt an den Server.
Diese Clientnachricht wird nun im Server in eine Warteschlange (siehe 4.2.2) mit anderen Nutzeranfragen geladen. Das geschiet, um den RMI Prozess nicht zu lange zu blockieren. Wird diese update Anfrage nun verarbeitet, so wird der Server versuchen, das vom Client erhaltene, aktualisierte Userobjekt in die Datenbank (siehe 5) zu speichern. Sobald das erledigt ist, lädt der Server das veränderte Userobjekt erneut aus der Datenbank und speichert es in seiner eigenen Hashmap ab (siehe 4).
Die veränderte Information erhalten die anderen Clients nun dadruch, das sie alle zehn Millisekunden eine RMI update Anfrage an den Server schicken, um sich mit diesem zu synchronisieren. Sie holen sich dabei also sämtliche Hashmaps des Servers.

3.4.3 Kommunikation zwischen Server und Client

Der Server muss in der Lage sein, direkt mit dem Client zu kommunizieren. Das ist notwendig, wenn beispielsweise der User eine Anfrage zum Beitritt an den Lehrer stellt, sowie wenn dieser darauf eine Antwort gibt.
Das erreichen wir, in dem der Server für jeden angemeldeten Client eine Warteschlange mit Befehlen bereit hält (siehe 4.2.2). Diese wird nach erfolgreichem Anmelden jede Sekunde vom Client selber über RMI abgefragt. Daraus erhält der Client eine Liste von Befehlsobjekten. Ein Befehlsobjekt besteht aus einer

id, einem Befehlsprefix als String, einem optionalen Objekt, sowie zwei weiterer, hier unwichtigen, Parameter. Anhand des Befehlsprefix wertet der Client das Befehlsobjekt in der 'ReadCommandList' Klasse aus.

3.5 Utils

Ein letzter Teil des Clients besteht aus den util Klassen. Hierbei handelt es sich teilweise um kleine Helferklassen die einige Funktionalitäten bereitstellen und das Programmieren ein wenig vereinfachen, aber auch Enum Klassen. Helferklassen sind unter anderem die 'NextDate' Klasse und die 'Synchronizer' Klasse. Erstere erlaubt es uns bestimmte Operationen auf Daten ausführen zu können, die zweite um den Client aktuell zu halten. Zu den Enums zählen Klassen wie 'AccountType' welcher angibt ob es sich bei dem User um einen 'STUDENT', 'TEACHER' oder 'ADMIN' handelt, aber auch die 'Language' Klasse mit welcher festgelegt wird, in welcher Sprache die GUI angezeigt werden soll.

4 Server

In diesem Abschnitt befassen wir uns mit dem Server. Der Server ist eine eigenständige, seperat entwickelte Java Applikation, die verschiedene Funktionalitäten für die Clients bereitstellt.

4.1 Befehlsobjekte

Wir hatten uns mit der Frage beschäftigt, wie der Server überhaupt alles koordinieren soll. Spricht er die Clients direkt an und triggert Events? Dann bräuchten wir TCP oder ähnliches. Nimmt er nur Befehle von Clients entgegen und bearbeitet diese? Oder sollte der Server das Mastermind hinter allem sein, mit dem sich die Clients nur stetig austauschen?
Aus diesen Wahlmöglichkeiten wählten wir einen Mix aus dem 2. und 3. Punkt. Der Server nimmt Anfragen in Form von Befehlen von Clients entgegen, bearbeitet diese, passt ggf seinen Zustand und die Datenbank an und ist fertig. Die Clients wiederrum erfahren von den Veränderungen nur indirekt über eine Synchronisierung mit dem Server.
Wir benötigen also Befehlsobjekte, die hinzukommend auch noch Serialisierbar sein müssen (Wir arbeiten ja mit RMI). Letztendlich besitzt unser Befehlsobjekt 5 Aspekte:

- originId: Die User Id (Client Id) von dem der Command stammt
- command: Ein String der Form "COMMAND:ARGUMENTS", z.B.: "UU:2" → es gibt ein User Update vom User 2 (z.B. wurde der Name geändert oder ähnliches)
- updatedObject: Das Objekt was bei einem Update Command das geupdatete Objekt repräsentiert. In unserem Beispiel wäre das der neue User mit verändertem Namen

- workspaceFileBytes: Ein Byte Array für die Synchronisation des Arbeitsplatzes beim kollaborativen Arbeiten. In dem Array sind die JComponent Daten hinterlegt. (z.B. UseCase mit Breite 2 und Höhe 4 mit Text 'Bla' und etc.)
- taskBytes: Ein Nischen Byte Array. Dieser Parameter wird tatsächlich nur für eine Benutzungsmöglichkeit der Software belegt: Wenn der Lehrer den Schülern eine Aufgabe stellt und eine Aufgabenstellung schriftlich mitteilen möchte

Dieses Befehlsobjekt wird jedoch nicht direkt abgearbeitet. Wie folgend erläutert wird, wird das Befehlsobjekt nämlich nur an eine CommandQueue gehangen.

4.2 Warteschlangen

Der Server besitzt zur besseren Laufzeit und Ansprechbarkeit durch Clients mehrere Queues. Eine davon ist für die Befehle, die Clients ausführen müssen und eine tut das selbe für den Server.
Als Queue Implementierung wird hier die Java 'LinkedBlockingQueue' verwendet. Beide Queues sind Queues von in 4.1 eingeführten Befehlsobjekten

4.2.1 Server Warteschlange

Wozu überhaupt? Da wir RMI benutzen, resultieren Client RMI Methodenaufrufe in der Blockierung und exklusiven Nutzung des Servers durch den Client. Wenn die Methode nun genügend komplex ist, kann es natürlich sein, dass der Server für längere Zeit blockiert ist und andere Client Anfragen verworfen werden. Um dieses Problem zu minimieren, werden bei RMI Methodenaufrufen die Methoden nicht direkt ausgeführt, sondern nur ein Befehlsobjekt wie in 4.1 beschrieben kontextabhängig an die ServerCommandQueue rangehangen. Somit wird die Zeit des Blockierens durch Clients minimiert.
Die ServerCommandQueue wird von einem seperaten Thread kontinuierlich abgearbeitet. Somit kann tatsächlich eine extrem gute Performance und Availability erzielt werden, obwohl der Server hart am arbeiten ist.

4.2.2 Client Warteschlange

Eine analoge Queue existiert für die Clients. Hier besitzt der Server eine HashMap, die jeder Client Id (=UserId) eine Queue zuordnet, die der Client abzuarbeiten hat. Somit kann der Server die Clients koordinieren, ohne sie jemals ansprechen zu müssen. Der Client guckt also periodisch beim Server nach, ob unter seiner Client Id eine Queue vorhanden ist, die er abzuarbeiten hat. Wenn dies der Fall ist, kopiert er sich die Queue lokal, löscht sie beim Server und arbeitet die Queue ab.
Alles rund um die Queue läuft auch wiederrum seperat auf einem Thread, sodass die Performance des Clients nicht eingeschränkt wird.

4.3 RMI des Servers

Wir möchten nicht in die Tiefe des RMI Protokolls gehen und wie man das nun genau implementiert, dafür gibt es genug Tutorials. Wichtig ist nur, dass der RMIServer natürlich alle Methoden vom RMIServerInterface implementieren muss, ansonsten kann der Client die Methode ja nicht remote aufrufen (sie würde dann ja nicht existieren). Der Server stellt also die gesamte RMI Funktionalität bereit und hält dort Methoden, die Clients aufrufen möchten.

4.4 Kommunikation zwischen Server und Datenbank

Für die Datenbankkommunikation berufen wir wieder mal einen Manager: den DBManager.
Wie schon erwähnt, wird java.sql benutzt, um mit einer MySQL Datenbank zu kommunizieren, die alles Sessionübergreifend persistent speichert. Beim Starten des Servers wird der komplette sessionabhängige Datenbankteil resetted. Dieser Teil umfasst z.B. welche User gerade online sind, welche Gruppen zur Zeit erstellt wurden, etc.. Dadurch wird sichergestellt, dass es keine 'Geister' gibt. Also z.B. Gruppen, die gar nicht existieren.
Danach synct der Server sich einmalig(!) mit der Datenbank und cached alle Daten in seinen Managern. Dadurch braucht der Server nicht ständig mit der Datenbank kommunizieren, solche Datenbankanfragen sind nämlich extrem zeitaufwändig. Also ist der Server danach also asynchron zur Datenbank? Nein. Bei jedem Befehl, den der Server abzuarbeiten hat, bei dem die Datenbank verändert werden müsste, wird sie verändert und nur die betroffenen Tupel werden erneut gecached. Somit arbeiten wir minimal invasiv bezüglich der Datenbank und brauchen uns keine Gedanken darum machen, dass der Server eine schlechte Laufzeit hat, da er dauernd Datenbankanfragen stellt.

5 Datenbank

Wir benutzen die MySQL Datenbank. Hierfür stellt Java die Bibliothek java.sql zur Verfügung. Bei einer Anfrage benutzen wir entweder die executeUpdate Methode mit einem PreparedStatement um die Datenbank zu verändern, beziehungsweise die executeQuery Methode, ebenfalls mit einem PreparedStatement, um ein ResultSet mit relevanten Tupeln der Datenbank zu erhalten.
Die Datenbank wird, falls diese noch nicht existiert, automatisch beim Starten des Servers erstellt.

5.1 Datenbankstruktur

In der Datenbank speichern wir die User, Gruppen, Klassen (Course) sowie die Sessions. In Abbildung 2 kann unser ER-Diagram gesehen werden.
Weiterhin befindet sich unsere Datenbank in der 3. Normalform und ist damit robust gegenüber Inkonsistenzen. In der aktuellen Version kann es sogar vorkommen, dass der Server Fehler wirft, da die Datenbank wegen Integrität/Inkonsistenzen

meckert, falls man bestimmt Operationen darauf in der falschen Reihenfolge durchführt (z.B. Tupel in groupInSession erstellen, der auf eine Session referiert, die nicht existiert, da diese noch nicht erstellt wurde)

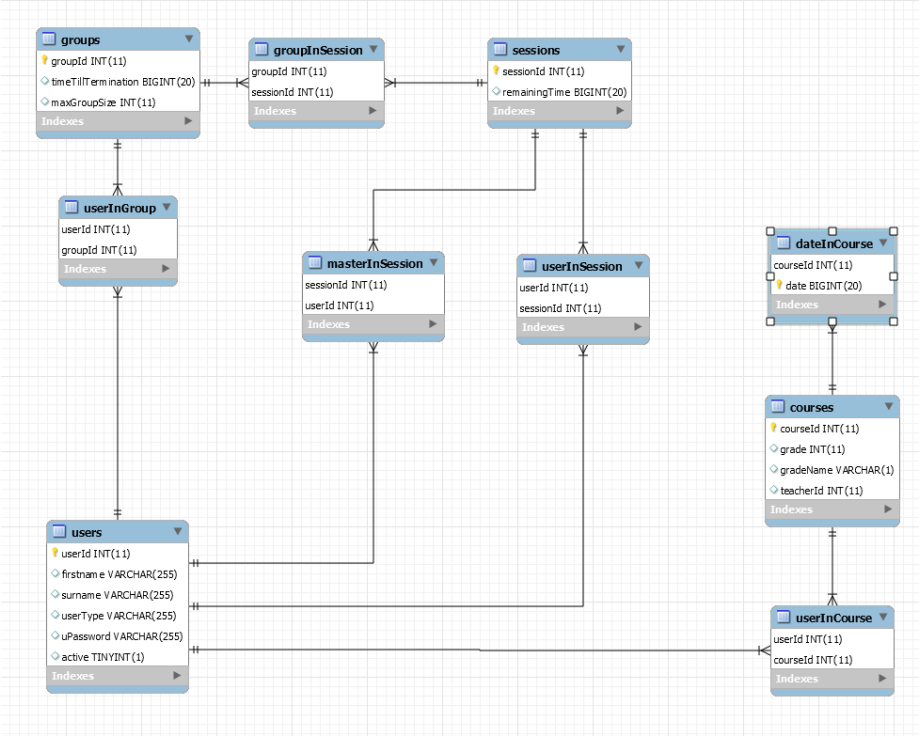

Abbildung 2: ER-Diagram

6 Probleme

Leider funktioniert nicht alles so wie wir es uns wünschen würden. Es passiert oft, dass der Server abstürzt, weil die Datenbank eine Integritätsverletzung bemängelt. Der Grund dafür ist uns jedoch nicht ersichtlich. Die einzige Möglichkeit, die uns einfällt, ist, dass unsere Java Datenbank-Schnittstelle asynchron operiert. Code-wise sind die Integritäten nämlich alle eingehalten.
Außerdem haben wir ein Memory Leak Client seits. Beim Synchronisieren des Arbeitsplatzes beim kollaborativen Arbeiten werden die alten Objekte aus irgend einem Grund von Java's Garbage Collection nicht aufgenommen und freigegeben, selbst wenn man deren Referenz auf 'null' setzt! Somit sind nach ca. 15 Minuten 16 GB voll. Das ist natürlich nicht gewollt, aber wir haben da auch keinen Ansatz, das zu fixen.
Weiterhin ist die CPU Last beim kollaborativen Arbeiten enorm. Obwohl der

Client nur effektiv 30 Mal pro Sekunde den Arbeitsplatz synchronisiert (aber auch nur bei Veränderung), steigt die CPU Last dabei auf 100%. Das liegt höchstwahrscheinlich daran, dass wir das Syncen mit einem Object[Out/In]putStream aufgebaut auf einem File[Out/In]putStream realisieren. Und letzterer arbeitet natürlich auf einer tatsächlichen Datei im Root Ordner. Diese Datei-Operationen sind vermutlich zu anspruchsvoll für eine Taktrate von 30 Mal pro Sekunde. Ein Ansatz hier wäre, dass man statt auf einem File[Out/In]putStream zu arbeiten, direkt mit dem übergebenen Byte Array arbeitet und dadurch nur alles im Arbeitsspeicher speichert.

7 Was noch fehlt

Hier können wir wieder ein wenig glänzen. Es fehlt nicht viel! Im Vergleich zu unserem Pflichtenheft/Feinentwurf fehlen nur 3 Dinge. Die Admin-Übersicht, die textuelle Use Case Spezifikation und eine Signup Prozedur.
Die Admin-Übersicht wurde entfernt, da Sie keine hohe Priorität besaß und am Ende nun doch die Zeit und die Motivation gedrückt hatte! Trotzdem gibt es den Admin als Account-Typen, dieser hat dann halt nur das Alleinstellungsmerkmal Klassen (Courses) zu migrieren, zu löschen oder zu resetten.
Desweiteren fehlt eine Signup-Prozedur. Derzeit kann man neue User nur direkt über z.B. MySQL Workbench hinzufügen.

8 Begriffserklärungen

- Session - Eine temporäre Repräsentation des Klassenraumes. Wird erzeugt, wenn der Lehrer dem Klassenraum beitritt und bleibt für 120 Minuten bestehen, danach wird sie vom Server gelöscht. Wird auch gelöscht, falls sich der Lehrer abmeldet. Wird für die Synchronisation des Arbeitsplatzes beim kollaborativen Arbeitens genutzt (Synchronisiere nur zu Usern, die in einer Session sind). Eine Session ist demnach eine Kapselung von Usern.
- Course/Klasse - Ein persistentes Objekt, was einen Klassenraum repräsentiert. Hält Informationen über Schüler, Lehrer, Terminen, etc.. Beim Betreten durch den zuständigen Lehrer wird eine Session erschaffen.
- Group/Gruppe - Ein an eine Session gebundenes temporäres Objekt. Eine Gruppe enthält User, die alle das Recht haben, den Arbeitsplatz zu bearbeiten (Dieser wird dann nur an Teilnehmer der Gruppe synchronisiert!). Eine Gruppe ist also eine weitere Kapselung von Usern in einer Session (”Subsession”).