Luisenschule

Gymnasium der Stadt Mülheim an der Ruhr

Besondere Lernleistung

CaRP-Assigner

verfasst von

Fabius Mettner

Grundkurs Informatik

Betreuer: Herr Dipl. Inf. Ch. Wolf

Abgabetermin: 16.03.2020

Inhaltsverzeichnis

[1. Einleitung 2](#_Toc35081369)

[1.1. Motivation 2](#_Toc35081370)

[1.2. Vorgehensweise 5](#_Toc35081371)

[2. Der Entwicklungsprozess 7](#_Toc35081372)

[2.1. Die Neustrukturierung 7](#_Toc35081373)

[2.2. Die Ausprogrammierung 7](#_Toc35081374)

[2.3. Die größten Fehler 9](#_Toc35081375)

[2.3.1. Das Verschwinden der Projektkurs-Schüler 9](#_Toc35081376)

[2.3.2. Fehler beim Sortieren der Berechnungen nach der Güte 10](#_Toc35081377)

[3. Manual 11](#_Toc35081378)

[3.1. Importieren 11](#_Toc35081379)

[3.1.1. Kriterien, die eine Eingabedatei erfüllen muss 12](#_Toc35081380)

[3.2. Der Verteilungsprozess 15](#_Toc35081381)

[3.3. Das Exportieren der Daten 15](#_Toc35081382)

[3.4. Das Bearbeiten der Eingabedaten im CaRP-Assigner 16](#_Toc35081383)

[3.5. Das Bearbeiten der Verteilung im CaRP-Assigner 17](#_Toc35081384)

[3.6. Die Einstellungen 18](#_Toc35081385)

[4. Fazit 19](#_Toc35081386)

[4.1. Optimierungsmöglichkeiten 19](#_Toc35081387)

[4.2. Zusammenfassung 19](#_Toc35081388)

[5. Anhang 20](#_Toc35081389)

[6. Quellen- und Literaturverzeichnis 21](#_Toc35081390)

[6.1. Internet 21](#_Toc35081391)

[6.2. Bilder 21](#_Toc35081392)

[6.3. Java-Abhängigkeiten 22](#_Toc35081393)

[7. Abschlusserklärung 23](#_Toc35081394)

# Einleitung

Der Course and Research Paper Assinger (Kurs und Facharbeit Zuweiser = KuFa-Zuweiser) ist ein Programm zum Zuweisen von Schülern nach Wunschkursen, z.B. Sportkurse oder Facharbeiten, die nach einer Priorität geordnet sind. Dabei wird versucht eine Zuweisung zu erstellen, welche einer Verteilung mit der bestmöglichen Priorität entspricht und dabei das Limit eines jeden Kurses nicht überschreitet.

## Motivation

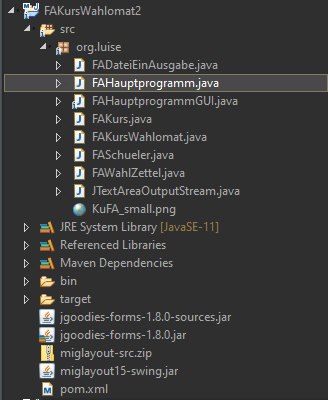
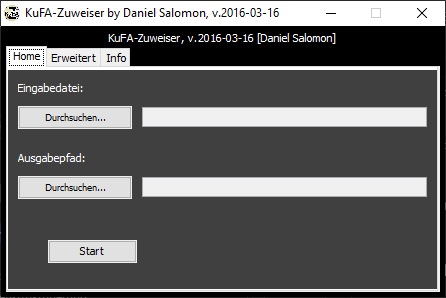
Betrachtet man die alte Version des KuFA-Zuweisers (Abb. 1), so erkennt man eine im Java „Swing“-Format angelegte Benutzeroberfläche. Beim Ausführen des Berechnungs-prozesses, friert diese graphische Oberfläche (GUI) ein, wodurch ein Beobachten des Fortschrittes nicht mehr möglich ist. Auch lässt sich die Größe des Fensters nicht verändern. Beides hat die Grundfunktionen der Applikation nicht beeinträchtigt, konnte aber beim Anwenden der Applikation als störend empfunden werden. Aus diesem Grund lässt sich zunächst eine vollständige Überarbeitung der graphischen Oberfläche als Ziel setzen. Diese muss vor allem anwenderfreundlicher, variabler und moderner werden.

Abb. 1: Alte KuFA-Anwendung

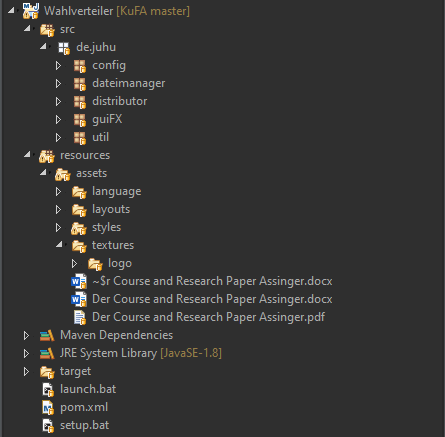
Bei der ersten Betrachtung des dahinterstehenden Quellcodes, ist bereits der nächste Verbesserungspunkt zu erkennen. Der alte KuFA-Zuweiser verfügt lediglich über wenige besonders lange Klassen, die die einzelnen Funktionen grob unterteilen. Um eine übersichtliche Struktur in das Projekt zu bringen, wurde nun mit sogenannten „packets“ (Paketen) gearbeitet, die das Projekt hierarchisch strukturieren sollten. Über diesen Weg werden die einzelnen Funktionen leichter unterscheidbar gemacht. Gleichzeitig bietet die in Abb. 3 zu erkennende Struktur einen weiteren Vorteil: Jeder Bereich kann einfacher getrennt werden und so in einem anderen Programm einfach wiederverwertet werden. Es erleichtert auch das Testen der einzelnen Programmabschnitte, durch den Fokus auf die unterschiedlichen Funktionsbereiche.

Abb. 2: Klassenstruktur des alten KuFA-Zuweisers

In Folge dieser Neustrukturierung ist der gesamte vorangegangene Quellcode aus mehreren Gründen verworfen worden:

Abb. 3: Klassenstruktur des CaRP-Assigners

1. Zum einen war es nicht möglich, mit ihm die einzelnen Funktionen entsprechend zu trennen.
2. Teilweise erschien der Quellcode sehr unübersichtlich und das Einarbeiten in diesen hätte ähnlich viel Zeit in Anspruch genommen, wie das strukturierte Neuerstellen des Quellcodes.
3. Durch diese komplette Neuauflage des Projektes wird es zudem möglich, einfacher neue Funktionen zu integrieren.

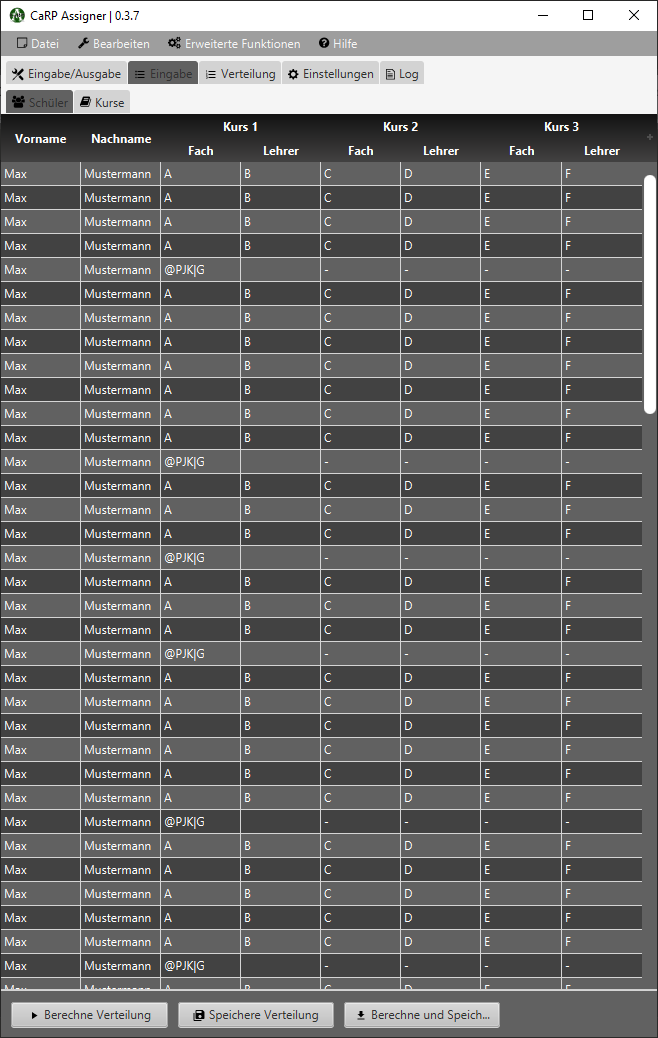


Abb. 4: Eingabedatenvorschau des CaRP-Assigners

Im Verlauf des Erstellens der besonderen Lernleistung kommt zudem ein weiterer Punkt hinzu:

Eine Voransicht der Daten im Programm sowohl vor dem Zuweisungsprozess als auch nach diesem, erleichtert dem Endnutzer das Bedienen des Programms. So wird eine Vorschau für die Eingabe- und für die Ausgabedaten eingefügt. Um die flexible Nutzung des Programms hierbei zu steigern, wird im Anschluss noch die Möglichkeit der Bearbeitung der Daten in den Voransichten hinzugefügt.

## Vorgehensweise

Mit diesen Verbesserungspunkten beginne ich die Erstellung eines komplett neuen Kurs- und Facharbeit Zuweisers. Zunächst einmal müssen die bereits vorhandenen Funktionen des alten Programms gesichtet und neu ausprogrammiert werden, um im Folgenden dann nach neuen Ansätzen zur Überarbeitung des Programmes und neuen Funktionen suchen zu können. Nach einiger Überlegungszeit und vielem Ausprobieren entsteht ein komplett neues Java-Projekt in Eclipse, einer Programmierumgebung für Java.

Hierbei werden schon die ersten großen Änderungen im Grundaufbau vorgenommen. Anstatt der flachen Hierarchie des Klassenbaumes wird nun auf einen hierarchisch durchstrukturierten Klassenbaumaufbau, wie er auch in Abb. 3 zu sehen ist, gesetzt. Dies ermöglicht nicht nur einen einfacheren Überblick über die einzelnen Bereiche, sondern erweist sich auch als sinnvoll, wenn man nur Teile des Programms verändern oder in eine neue Umgebung (z.B. in ein neues Projekt) überführen will.

Des Weiteren wird nun auch die Einbindung von Libraries erneuert. Statt die einzelnen benötigten Libraries manuell hinzuzufügen wird auf das Build Management Tool „Maven“ gesetzt. Neben einer vereinfachten Möglichkeit zum Exportieren des Projektes in das „.exe“ und „.jar“-Format ist es so auch möglich, einfacher neue Referenzen, sogenannte „Dependencies“, im Programm einzufügen. Diese Referenzen werden von Maven selbstständig verwaltet und es ist nicht mehr nötig, manuell Libraries in das Projekt zu importieren.

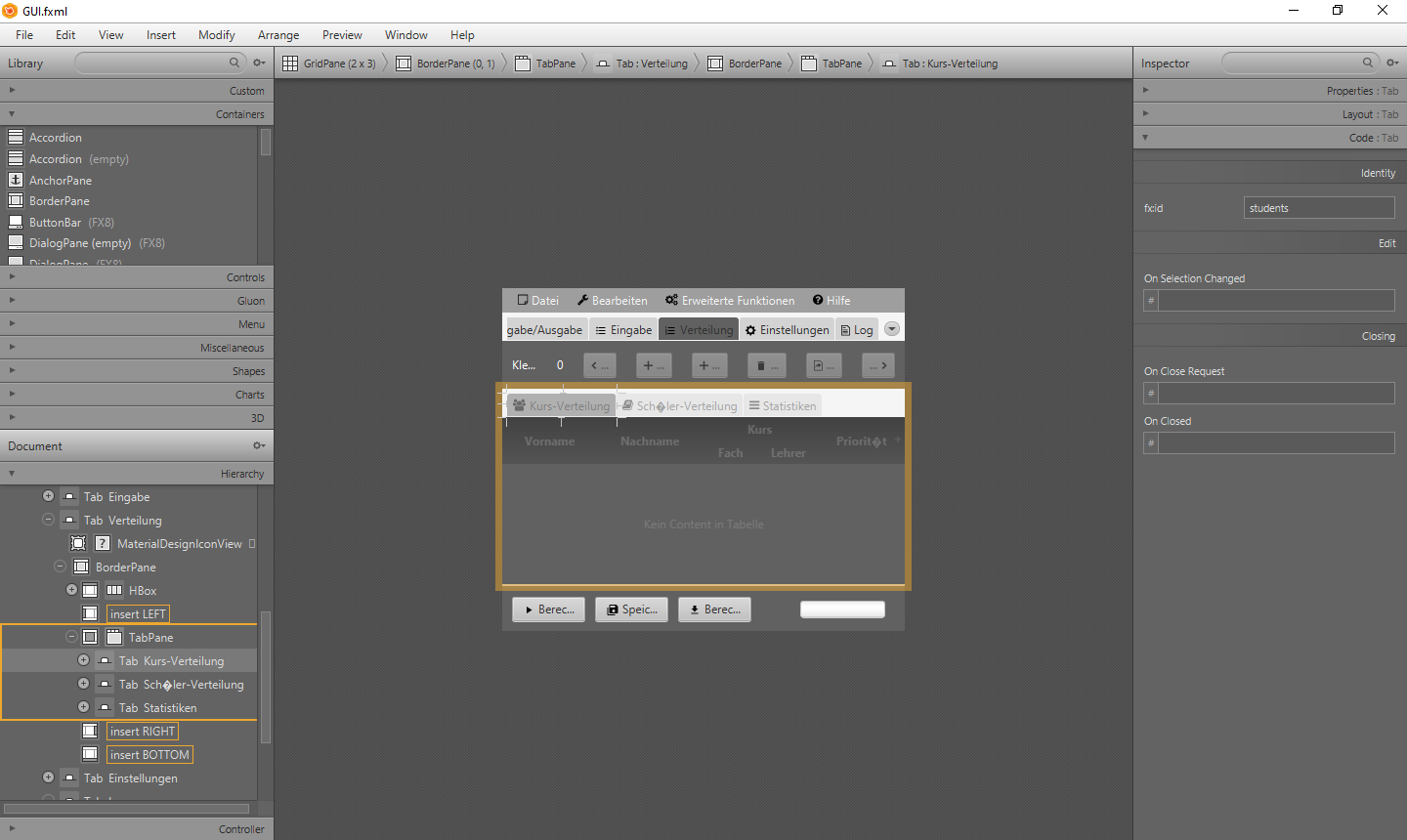
Ebenso hat sich die graphische Oberfläche grundlegend verändert. Die genutzte „Swing“ Oberfläche wird verworfen und durch eine neue, auf „JavaFX“ basierende Benutzeroberfläche, ersetzt. Diese gibt der Anwendung nicht nur einen „moderneren Look“, sondern ermöglichte auch das erleichterte Darstellen neuer Funktionen in das graphische Gesamtbild. Um eine passende Oberfläche zu schaffen, arbeitet man mit dem für diesen Zweck entwickelten „Scene Builder“, welcher die Erstellung neuer Benutzeroberflächen durch eine bearbeitbare Voransicht weitgehend erleichtert.

Abb. 5: Ansicht des GUIs im „Scene Builder“

Im weiteren Entwicklungsprozess werden neben neuen Features auch weitere Referenzen und Designs eingefügt. So ist das Wechseln zwischen einem hellen und einem dunklen Oberflächendesign, sowie das Wechseln der Sprache zwischen Deutsch und Englisch möglich geworden. Dies kann durch das Einbinden von „css“, zum Verändern des Oberflächendesigns, sowie das Einbinden von Sprachdateien über sogenannte „Internationalized Strings“ realisiert werden.

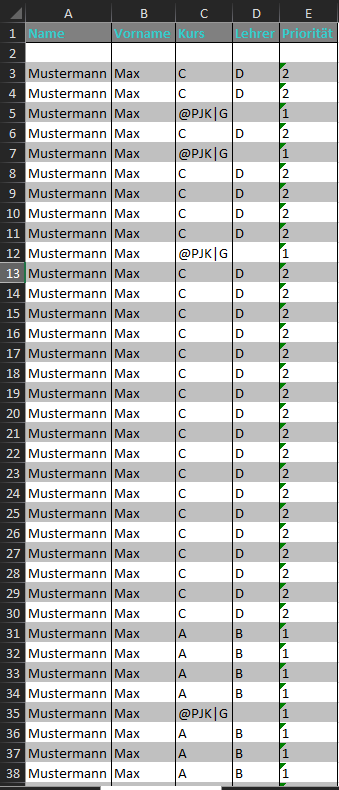
# Der Entwicklungsprozess

Nachdem die grundlegenden Ideen strukturiert und der Aufbau des Projekts geklärt ist, kann nun der eigentliche Entwicklungsprozess gestartet werden. Hierbei wird viel Zeit zum Testen und Verbessern benötigt und in die Implementierung der einzelnen Ideen mithilfe der vorangegangenen Vorgehensweisen investiert. Um einen Datenverlust oder einen andersgearteten Rückschlag zu vermeiden, wird nun ein „GitHub-Repository“ angelegt, auf dem die letzte funktionierende Version stets hinterlegt ist.

Abb. 6: „Github Repositoriy“ des CaRP-Assigners

## Die Neustrukturierung

Zuerst ist durch die neu aufgesetzte Struktur auch eine vollständige Erneuerung des alten „KuFA-Zuweisers“ entstanden. Dabei werden folgende Baumpfade erstellt: Unter dem Pfad „distributor“ ist alles für den Zuweisungsprozess Relevante eingearbeitet, während das Einlesen und Schreiben von Dateien in den Reiter „dateimanager“ eingefügt wird. Der Pfad „config“ enthält alle relevanten Dateien zum Erstellen der Konfiguration und unter „guiFX“ werden die Klassen der graphischen Oberfläche gesammelt. Alle weiteren Klassen fasst man schließlich im Reiter „util“ zusammen.

Um eine weitreichende Struktur zu wahren, wird zudem ein eigener Ordner für alle nicht Java-Dateien erstellt. Hier finden sich neben der „Hilfe“ Datei auch die „Stylesheets“ (css Dateien zum Bearbeiten der Ansicht der graphischen Oberfläche) und die Sprachdateien, sowie die „fxml“ Dateien, die die Struktur der einzelnen GUIs beinhalten.

## Die Ausprogrammierung

Begonnen hat die Ausprogrammierung zunächst mit dem Im- und Exporter für Dateien. Im gleichen Zug entsteht auch die Klasse „References“ mit einem den Umständen angepassten Logger. Um die Exportdatei graphisch aufzuwerten, wird die Klasse „CellStyles“ erstellt, die eine kleine Auswahl an unterschiedlichen Ansichten für die Tabellenzellen von Excel bereitstellt (siehe Abb.7).

Abb. 7: Exporttabelle der Kursverteilung mit Beispieldaten

Der Berechnungsprozess und alle zugehörigen Klassen werden nachfolgend in einem Schritt eingefügt. Um diese Funktionen zu testen, wird zunächst auf ein einfaches GUI auf „Swing“-Basis zurückgegriffen, bevor schließlich im nächsten Schritt unter Zuhilfenahme des „SceneBuilders“ ein JavaFX GUI aufgesetzt wird.

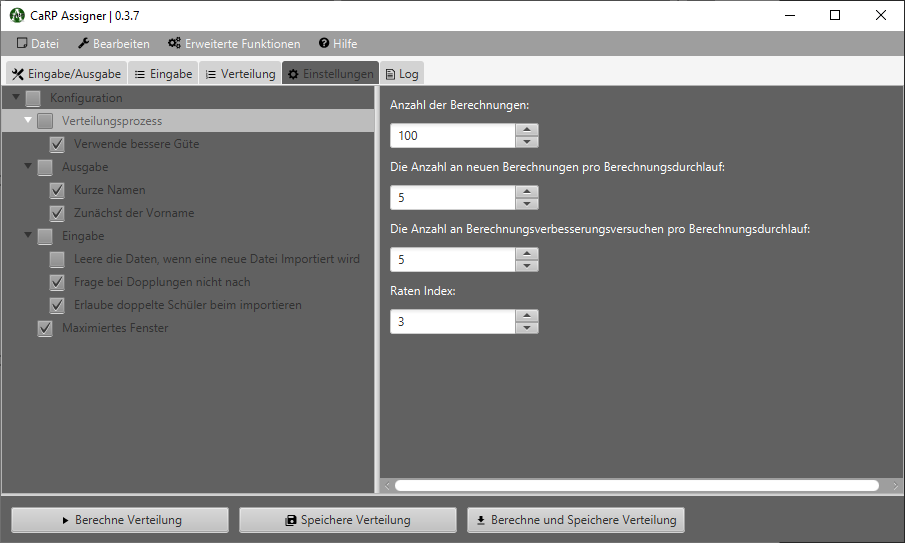
 Nach Erstellen des neuen GUIs kommen nun vor allem in dem Bereich der Berechnung und in der graphischen Darstellung immer wieder kleinere und größere Neuerungen hinzu. So wird die im Benutzerverzeichnis hinterlegte, speicherbare Konfiguration, der speicherbare Log und die Möglichkeit, eine einlesbare „.carp“-Datei zu exportieren, entwickelt. Über einen längeren experimentellen Zeitraum mit vielen Änderungen im graphischen Design, entstehen die neuen Vorschauen für die Voransicht der Verteilung und der einzulesenden Daten.

Abb. 8: Überarbeitete Einstellungsansicht

Schließlich werden die Shortcuts, sowie die Icons in das Programm eingeführt und das „Über“ Fenster optimiert. Zuletzt wird die in Abb. 8 zu erkennende Darstellung der Einstellungen überarbeitet und damit endet der eigentliche Programmierprozess. Zum Schluss müssen noch bestehende Bugs (Fehler) behoben und Klassen ausdokumentiert werden.

## Die größten Fehler

Während der Programmierung haben sich in den Programmcode immer wieder kleinere und größere Fehler eingeschlichen. Einige von ihnen sind sofort aufgefallen, bei anderen ist ein genaues Verfolgen der vorgehenden Prozesse nötig gewesen, um sie im Code selbst ausfindig zu machen und dort zu beheben. Dafür ist zunächst ein kleinschrittiges Auffinden der Fehlerquellen in kleinsten Programmabschnitten erforderlich. Mithilfe von hunderten „Print-Messages“ und unter Nutzung des in Eclipse verfügbaren Debug-Modus, können so viele der Fehlerquellen identifiziert werden. Nach dem Auffinden der Fehlerquelle ist das eigentliche Fehlerbeheben nicht mehr problematisch, da die meisten Fehler lediglich kleinen logischen Aussetzern (des Programmierers 😉) oder dem Übersehen von Zusammenhängen zu verdanken sind. Bei komplexeren Fehlern ist es sinnvoll, testweise die Wirkung einer Veränderung einzelner Programmabschnitte auszuprobieren, um so schließlich doch die Fehlerquelle zu entdecken. Im Folgenden sind einige dieser Fehler beispielhaft aufgeführt.

### Das Verschwinden der Projektkurs-Schüler

Die Behebung des ersten Fehlers, die ungewollte Verminderung der Anzahl der Projektkursschüler während des Berechnungsprozesses, hat besonders viel Zeit in Anspruch genommen. Jedoch ist die Behebung des Problems nach Entdecken des Programmfehlers relativ simpel gewesen. Durch Betrachten unterschiedlicher Verteilungsvorgänge, kann der Fehler schließlich in folgender Schleifenkonstruktion lokalisiert werden:

For (int i = 0; i < this.allStudents.size(); i++) {

If ((this.allStudents.get(i).getActiveCourse() != null &&

this.allStudents.get(i).getActiveCourse().equals(this.ignore())) ||

this.allStudents.get(i).getCoursesAsList()

.contains(this.ignore()))

this.ignoredStudents.add(this.allStudents.remove(i));

}

Wie zu sehen ist, wird jedes Mal beim Durchlaufen der Schleife der Iterator i eine Position weitergezählt. Wird nun ein Schüler aus der Liste „allStudents“ entfernt, rücken die nachfolgenden Schüler eine Position auf. Stehen nun zwei Projektkursschüler hintereinander in der Liste, so wird der erste erkannt und richtig verschoben, der dahinterstehende Schüler rückt dadurch einen Listenplatz auf, d.h., er nimmt nun den gerade behandelten Index des Iterator an. Da dieser nun aber eine Position weitergezählt wird, kann dieser Schüler nicht mehr behandelt werden und nicht mehr in die Liste „ignoredStudents“ eingefügt werden. Somit verschwindet er aus dem Verteilungsprozess und ist letztendlich gelöscht.

Beheben kann man diesen Fehler, in dem man den Index nur erhöht, wenn die if-Bedingung nicht zutrifft. Also ist die Lösung dieses Fehlers folgende:

For (int i = 0; i < this.allStudents.size();) {

If ((this.allStudents.get(i).getActiveCourse() != null &&

this.allStudents.get(i).getActiveCourse().equals(this.ignore()))

|| this.allStudents.get(i).getCoursesAsList().contains(this.ignore()))

this.ignoredStudents.add(this.allStudents.remove(i));

else

i++;

}

### Fehler beim Sortieren der Berechnungen nach der Güte

Dieser zweite Fehler, der erläutert wird, entsteht durch das falsche Interpretieren von den Rückgabewerten der „comparteTo(Object o)“-Methode. Nur durch Ausprobieren wird diesen Fehler schließlich behoben. Da der Ursprung des Fehlers bekannt war, nämlich die selbst geschriebene „Save.compareTo(Save s)“ Methode, kann durch Ausprobieren der Reihung und Verrechnung der verglichenen Werte stets schnell eine Lösung gefunden werden. Hierzu wird teilweise das Vorzeichen des zurückgegebenen Wertes vertauscht oder auch if-Bedingungen für Randfälle eingeführt, um so die Speicherung in der richtigen Reihenfolge anordnen zu können. Dabei bewirkt der entsprechende Vorzeichenwechsel entweder, dass die beste oder dass die schlechteste Berechnung vorne in der Liste der Speicher steht und zuerst angezeigt wird. Dieser Fehler ist hier vor allem aus dem Grund aufgeführt, weil er durch zwischenzeitliches Umstellen von den Berechnungen der Güte und anderen zu berücksichtigenden Werten immer wieder aufgetreten ist.

# Manual

Um den Einstieg in die Nutzung des CaRP-Assigners zu erleichtern, werden im Folgenden die wichtigsten Funktionen erklärt und Schritt für Schritt der Umgang mit dem Programm erläutert.

## Importieren

Abb. 9: Der Reiter „Eingabe/Ausgabe“

Gestartet wird zunächst mit dem Importieren (Einfügen) einer Datei in den CaRP-Assigner. Der CaRP-Assigner unterstützt sowohl .csv, .xls und .xlsx Dateien. Um eine Datei zu laden, begeben Sie sich in den Reiter „Eingabe/Ausgabe“. Das Importieren einer Datei kann nun über drei unterschiedliche Wege geschehen:

1. Über Drag and Drop: Nehmen Sie Ihre Datei und ziehen Sie diese über das weiße Textfeld neben dem oberen „Suche“ Knopf. Wenn Sie die Datei nun loslassen, wird sie automatisch in den Assigner geladen.
2. Über die „Suche“ Funktion: Betätigen Sie den oberen „Suche“ Knopf und wählen Sie Ihre Datei aus dem Dateisystem aus. Mit „Öffnen“ können Sie die Datei in das Programm laden.
3. Über das Menü: Sie können auch über den Menü-Reiter „Datei“ und dann über die Funktion „Suche Eingabedatei“ den Dateiexplorer zum Importieren der gewünschten Tabellendatei aufrufen.

Nachdem Sie Ihre Datei importiert haben, können Sie sich über den Knopf „Zeige Importierte Daten“, die von dem Programm eingelesenen Daten anzeigen lassen.

**Hinweis:** Es ist möglich, Daten zu bereits importierten Daten hinzuzufügen. Sollten sich bei diesem Vorgang Daten doppeln, wird Ihnen eine Warnung angezeigt. Die Optionen in diesem Warnfenster können auch in der Konfiguration geändert werden. Wenn sie diese Warnung ausstellen, wird sie Ihnen jedoch nach jedem Neustart wieder angezeigt werden!

### Kriterien, die eine Eingabedatei erfüllen muss

Damit eine Datei von dem neuen CaRP-Assigner eingelesen werden kann, muss sie folgende Kriterien erfüllen:

1. Zum Einfügen eines Schülers
   1. Die Datei muss die unter den Konfigurationseinstellungen zu findende Config „Schüler Kennzeichnung“ beinhalten, um einen Schüler hinzuzufügen. Diese „Schüler Kennzeichnung“ kann entweder in der Spalte „A“ vor den Angaben zum Schüler stehen, sodass dessen Werte erst in der darauffolgenden Spalte „B“ beginnen, oder sie kann als Tabellenbenennung angegeben werden. Dann steht der Nachname des Schülers schon in der Spalte „A“.
   2. Dabei ist die maximale Kursanzahl zu beachten. Der Schüler wird wie folgt in der Tabelle aufgeschrieben: {Nachname, Vorname, Kurs 1: Fach, Kurs 1: Lehrer, Kurs 2: Fach, Kurs 2: Lehrer, …, Kurs („maximale Kursanzahl“): Fach, Kurs („maximale Kursanzahl“): Lehrer}
   3. Wenn mehr Fächer angegeben werden als die „maximale Kursanzahl“, werden auch sie eingelesen, aber sie sind in der Berechnung nicht verfügbar, solange die „maximale Kursanzahl“ kleiner ist als die Anzahl der Fächer. Daher ist es zu empfehlen, dass die „maximale Kursanzahl“ auf die Kursanzahl des Schülers mit den meisten Kursangaben gesetzt wird!
2. Zum Einfügen eines Kurses:
   1. Die bei den Schülern angegebenen Kurse werden automatisch mit dem Schülermaximum aus der Config Datei initialisiert.
   2. Die Datei muss die unter den Konfigurationseinstellungen zu findende Config „Kurs Kennzeichnung“ beinhalten, um einen Kurs hinzuzufügen. Diese „Kurs Kennzeichnung“ kann entweder in der Spalte „A“ vor den Angaben zum Kurs stehen, sodass dessen Werte erst in der darauffolgenden Spalte „B“ beginnen, oder sie kann als Tabellen Benennung angegeben werden. Dann beginnen die Angaben zum Kurs in Spalte „A“. Der Kurs muss wie folgt in der Tabelle stehen: {Fach, Lehrer, Maximale Schülerzahl}
3. Um einen Kommentar zu kennzeichnen, muss in der Spalte „A“ als erstes das Zeichen aus der Konfiguration, welches unter „Kommentar Kennzeichnung“ zu finden ist, angegeben werden.

**Hinweis:** Alle Einstellungen, die hier erwähnt werden, sind unter „Konfiguration>Eingabe“ zu finden.

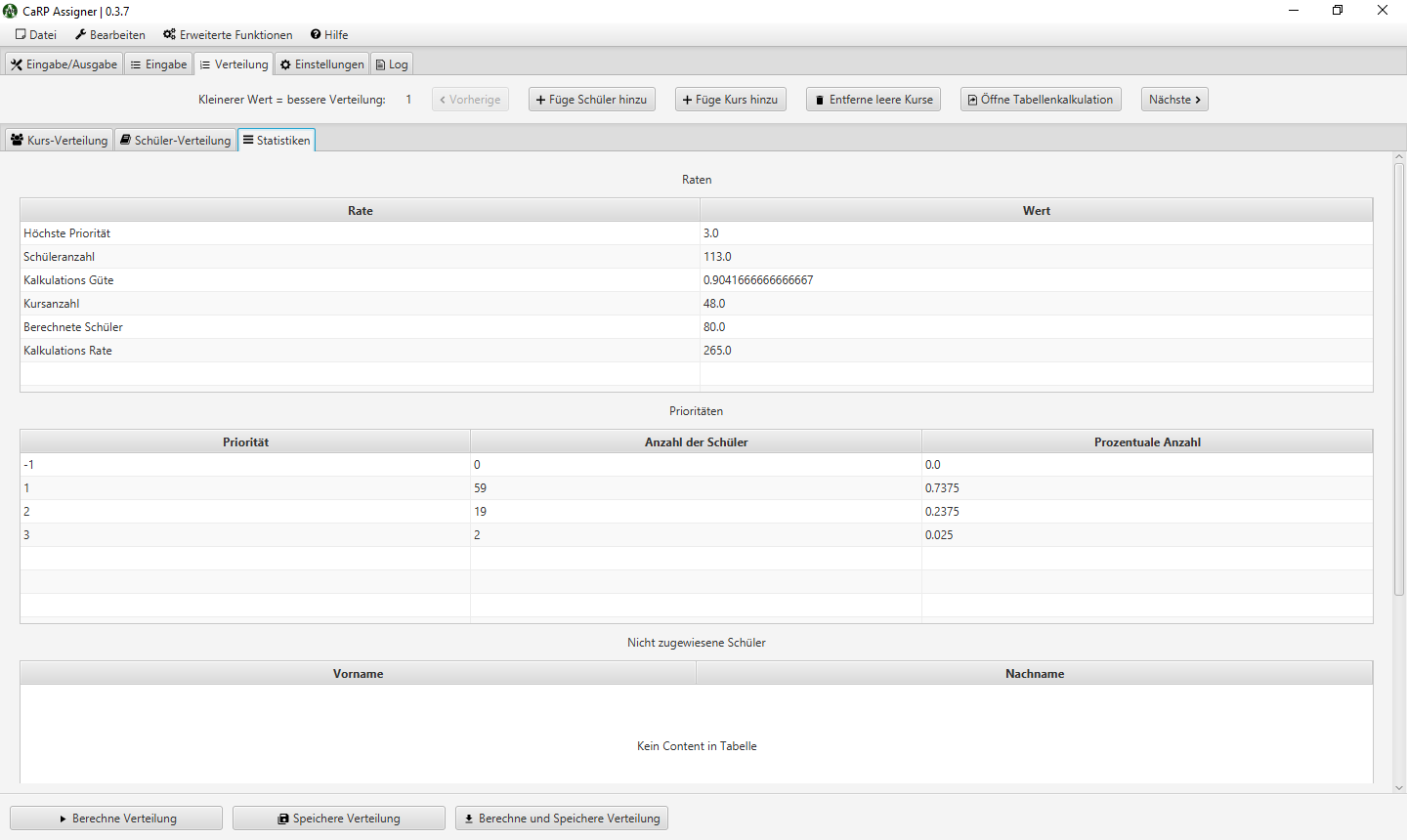


Abb. 10: Vorschau der Statistik einer Verteilung im light-Theme

## Der Verteilungsprozess

Um eine Verteilung zu berechnen müssen, bereits Daten eingelesen worden sein. Sollte dass der Fall sein, reicht ein Einfaches betätigen des Knopfes „Berechne Verteilung“ aus, um den Verteilungsprozess zu starten. Den Knopf „Berechne Verteilung“ finden Sie unten links auf der graphischen Oberfläche. Wenn unten rechts eine Prozessanzeige erscheint, welche Ihnen anzeigt, wie weit das Programm mit dem Verteilungsprozess ist, haben Sie den Zuweisungsprozess erfolgreich gestartet.

Sobald die Zuweisung abgeschlossen ist, wird eine Vorschau der Daten angezeigt. Diese gliedert sich in drei Unteransichten: Die „Kurs-Verteilung“, die die Schüler anzeigt mit dem ihnen zugeordneten Kurs. Die „Schüler-Verteilung“, die die Kurse mit den entsprechenden Schülern, die ihnen zugewiesen wurden, anzeigt und die in Abb. 10 zu sehenden „Statistiken“, welche weitere relevante Daten aufführen.

## Das Exportieren der Daten

Das Exportieren ist wieder, sowohl in das .xls, .xlsx als auch .csv Format möglich. Um den Ort auszuwählen, an den Sie Ihre Daten exportiert haben wollen, gehen Sie wieder in den Reiter „Eingabe/Ausgabe“ aus Abb. 9. Hier betätigen Sie den unteren „Suchen“ Knopf, um einen Ausgabepfad auszuwählen. Wählen Sie nun den gewünschten Ausgabeordner und bestätigen Sie mit „Ordner Auswählen“. Um das Dateiformat festzulegen, in welches die Daten exportiert werden sollen, kann der rechts in der gleichen Reihe zu findende Knopf verwendet werden, mit dem zwischen den drei Formaten gewählt werden kann. Die ausgewählte Möglichkeit wird dann als Text des Knopfes angezeigt.

Nun kann die Verteilung über den Knopf „Speichere Verteilung“ gespeichert werden. In dem gewünschten Ordner finden Sie nun zum einem eine „.log“-Datei, in der sich der Log befindet, dann eine „.carp“-Datei, die Sie mit dem KuFA-Zuweiser wieder öffnen können, und zuletzt die exportierte Datei im gewünschten Format.

## Das Bearbeiten der Eingabedaten im CaRP-Assigner

Unter dem Reiter „Eingabe“ finden Sie eine Vorschau, aller der von Ihnen importierten Daten. Um diese Daten zu bearbeiten, wählen Sie einfach den Eintrag in der Liste der Schüler oder der Liste der Kurse aus, den Sie gerne bearbeiten möchten. Mit einem Rechtsklick werden Ihnen nun die Bearbeitungsfunktionen angezeigt. Alternativ finden Sie im Menü unter dem Reiter „Bearbeiten“ ebenfalls für den ausgewählten Kurs oder Schüler die folgenden Bearbeitungsfunktionen:

Abb. 11: Bearbeitungsfenster für einen Schüler mit Daten

1. „Füge hinzu“: Mit dieser Funktion kann ein neuer Schüler oder Kurs in das Programm eingefügt werden. Dabei erscheint eine grafische Oberfläche, in die Sie die gewünschten Daten eintragen können. Wenn Sie fertig sind, betätigen Sie einfach den Knopf „Speichern“ und der neue Schüler wird automatisch eingefügt.
2. „Bearbeite“: Mit dieser Funktion ist es möglich, einen Schüler oder Kurs zu bearbeiten. Hier erscheint dasselbe GUI wie beim Einfügen des gewünschten Schülers oder Kurses. Die Felder sind nun dem Kurs oder Schüler entsprechend ausgefüllt und können daher beliebig verändert werden. Indem Sie „Speichern“ drücken, werden die Daten überschrieben.
3. „Entferne“: Mit dieser Funktion ist es möglich, den aktuell ausgewählten Kurs oder Schüler zu löschen.

**Hinweis:** Groß und Kleinschreibung wird bei den Kursen nicht berücksichtigt.

## Das Bearbeiten der Verteilung im CaRP-Assigner

Auch an den verteilten Daten können noch Veränderungen vorgenommen werden. Dazu kann der Schüler, oder Kurs, der bearbeitet werden soll, ausgewählt werden, und dann die gewünschte Aktion ausgeführt werden:

Abb. 12: Fenster zum Bearbeiten eines Schülers in einer Verteilung

* 1. „Bearbeite“: Ermöglicht es bei Schülern, diesen in einen anderen seiner gewählten Kurse zu setzen oder seinen Namen zu verändern, und bei Kursen, deren Namen zu verändern. Dazu erscheint, wie auch beim Bearbeiten der Eingabedaten, eine graphische Oberfläche, an der die Veränderungen vorgenommen werden können.
  2. „Entferne“: Entfernt den Schüler oder Kurs aus der Berechnung. Dabei können lediglich leere Kurse entfernt werden!

Um einen Kurs oder Schüler hinzuzufügen, können die über den Tabellen zu findenden Knöpfe „Füge Schüler hinzu“ und „Füge Kurs hinzu“ genutzt werden.

Da bei einem Verteilungsprozess mehrere Verteilungen erstellt werden ist es mit Hilfe der Knöpfe „Vorherige“ und „Nächste“ auch möglich, zwischen den besten fünf Verteilungen zu wechseln.

Um nur die Kurse mit Schülern in der Verteilung zu haben, können auch direkt alle leeren Kurse über den Knopf „Entferne leere Kurse“ gelöscht werden.

Wenn Sie eine Vorschau der Datei in Excel sehen möchten, ist dies über den Knopf „Öffne Tabellenkalkulation“ möglich.

## Die Einstellungen

Abb.13: Konfigurationsfenster mit rechts eingeblendetem Beschreibungstext für die links ausgewählte Einstellung

Abschließend wird nun der Reiter „Einstellungen“ erklärt. Hier sind alle Konfigurationsmöglichkeiten aufzufinden. Dabei können im Baum (links), die gewünschten Pfade ausgewählt werden, wodurch auf der rechten Hälfte die zugeordneten Konfigurationsmöglichkeiten angezeigt werden. Einstellungen, die nur an- und ausgeschaltet werden können, sind im Baum mitaufgeführt. Werden diese ausgewählt, erscheint rechts eine Beschreibung über deren Funktion. Allen anderen Einstellungen sind Hovertexte, also Texte, die beim Darüberfahren mit der Maus erscheinen, beigefügt, die deren Funktion erläutern.

# Fazit

Abschließend sollen noch einmal der CaRP-Assigner betrachtet werden und einige weitere Verbesserungsmöglichkeiten aufgeführt werden, sowie die gesammelten Erfahrungen herausgestellt werden.

## Optimierungsmöglichkeiten

Viele Verbesserungen wurden in dieser Version des CaRP-Assigners bereits umgesetzt. Dennoch lassen sich immer wieder weitere Verbesserungen und Vereinfachungen finden.

Zuerst will ich hier noch einen weiteren Möglichen Weg zum Einlesen der Daten erwähnen. So könnte man das Importieren von Daten aus einer Datenbank ermöglichen. Dies würde später auch die Kurswahlen der Schüler mithilfe eine Web-Applikation erlauben.

Auch bei Betrachtung der graphischen Oberfläche können noch kleine Änderungen zum Verbessern des Aussehens erkannt werden, wie Beispielsweise das Hinzufügen weiterer Designarten, um das graphische Aussehen auf jeden Nutzer anzupassen. Auch wäre eine Erweiterung der zur Auswahl stehenden Sprachen möglich, als auch eine Überarbeitung der englischen Sprachversion.

## Zusammenfassung

Abschließend lässt sich die Arbeit als weitreichende Bereicherung meiner Fähigkeiten in Java sehen. Neben dem Erlernen vom Arbeiten mit „css-Styling“ und Maven konnte ich auch meine Programmierkenntnisse, beispielsweise durch das Erlernen der Funktion von Lambda Ausdrücken, weitreichend verbessern. Auch im Umgang mit GitHub konnte ich so meine Kenntnisse weiter schulen. In diesem Sinne ist festzuhalten, dass diese besondere Lernleistungen mich inner- als auch außerunterrichtlich stark gefordert, sowie gefördert hat.

# Anhang

Hier ist zuletzt der gesamte Quellcode noch einmal nach den Klassen geordnet aufgelistet:

## Konfiguration

package de.juhu.config;

import static java.lang.annotation.ElementType.***FIELD***;

import static java.lang.annotation.RetentionPolicy.***RUNTIME***;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;

/\*\*

\* Diese Annotation verifiziert ein Attribut, als Konfigurationsattribut.

\* Dabei muss das Attribut {@code public static}, also öffentlich und

\* statisch sein. Das Config Elements wird über die Klasse {@link

\* ConfigManager Konfigurations Manager} registriert.

\* Konfigurationsattribute, die ein {@code String}, {@code Integer}, oder

\* {@code Boolean} als Wert aufweisen, werden automatisch im GUI unter dem

\* Reiter Einstellungen zu finden sein. Dabei ist {@link #description() die

\* Beschreibung} als Hovertext und {@link #name() der

\* Name} als Benennung eingefügt. Hierbei werden diese beiden eingegebenen

\* Strings durch den String aus der verwendeten Sprachdatei ersetzt. Alle

\* registrierten Konfigurationselemente werden in der Config-Datei unter

\* "%localappdata%/CaRP/config.cfg" gespeichert.

\*

\* **@author** Juhu1705

\* **@category** Config

\* **@version** 1.0

\*/

*@Retention*(***RUNTIME***)

*@Target*(***FIELD***)

public @interface *ConfigElement* {

/\*\*

\* **@return** Den Standartmäßig gesetzte Initialwert.

\* **@implNote** Nur Strings werden automatisch richtig initialisiert.

\* Bitte den gewünschten Wert standartmäßig einprogrammieren. Dieser

\* wird überschrieben, sobald die Konfigurationsdatei geladen wird.

\*/

public String defaultValue();

/\*\*

\* **@return** Den Objekttypen dieser Klasse. Wenn primäre Datentypen wie

\* int benutzt werden, dann kann hier die Klasse Integer.class

\* verwendet werden.

\*/

public Class elementClass();

/\*\*

\* **@return** Den in den Sprachdateien hinterlegten Übersetzungsstring

\* für die Beschreibung der Konfiguration

\* **@implNote** Der hinterlegte String muss, damit dass Programm läuft,

\* in den Sprachdateien hinterlegt sein.

\*/

public String description();

/\*\*

\* **@return** Den in den Sprachdateien hinterlegten Key zum Übersetzten

\* des Namens.

\* **@implNote** Der hinterlegte String muss, damit dass Programm läuft in

\* den Sprachdateien hinterlegt sein.

\*/

public String name();

/\*\*

\* **@return** Die Position im Baumsystem, unter der die Config zu finden

\* ist.

\* **@implNote** Der hinterlegte String muss, damit dass Programm läuft in

\* den Sprachdateien hinterlegt sein. Ein "." trennt die

\* Strings. Jeder Einzelstring wird mit "String".location gesucht.

\*/

public String location();

} //Ende des @interfaces *ConfigElement*

package de.juhu.config;

import static de.juhu.util.References.***LOGGER***;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.lang.reflect.Field;

import java.util.ArrayList;

import java.util.logging.Level;

import org.xml.sax.InputSource;

import org.xml.sax.SAXException;

import org.xml.sax.XMLReader;

import org.xml.sax.helpers.XMLReaderFactory;

import de.juhu.distributor.Distributor;

import de.juhu.guiFX.GUIManager;

import de.juhu.util.Config;

import de.juhu.util.References;

import javafx.event.ActionEvent;

import javafx.scene.control.CheckBoxTreeItem;

import javafx.scene.control.Label;

import javafx.scene.control.Spinner;

import javafx.scene.control.SpinnerValueFactory;

import javafx.scene.control.TextArea;

import javafx.scene.control.TextField;

import javafx.scene.control.Tooltip;

import javafx.scene.control.TreeItem;

import javafx.scene.control.TreeView;

import javafx.scene.control.cell.CheckBoxTreeCell;

import javafx.scene.input.KeyEvent;

import javafx.scene.input.MouseEvent;

import javafx.scene.layout.VBox;

/\*\*

\* Hier werden alle {@link ConfigElement Konfigurations Elemente} gespeichert

\* und verwaltet. Diese Klasse ermöglicht das Laden der initialisierten Werte,

\* einer Konfigurations-Datei, sowie das Schreiben einer Konfigurationsdatei.

\*

\* **@author** Juhu1705

\* **@category** Config

\* **@version** 1.0

\*/

public class ConfigManager {

/\*\*

\* Die Instanz des {@link ConfigManager Konfigurations Managers}.

\*/

private static ConfigManager *instance*;

/\*\*

\* **@return** Die {@link #instance aktive Instanz} des Konfigurations-Managers.

\*/

public static ConfigManager getInstance() {

return *instance* == null ? *instance* = new ConfigManager() : *instance*;

}

/\*\*

\* Alle {@link ConfigElement Konfigurations-Elemente}, die registriert wurden.

\*/

private ArrayList<Field> fields = new ArrayList<>();

/\*\*

\* Gibt das erste {@code Field} aus {@link #fields der Liste aller

\* Konfigurations-Elemente} aus, dessen Name mit dem mitgegebenen {@link String}

\* übereinstimmt zurück.

\*

\* **@param** name Die Bennenung des {@link ConfigElement Konfigurations-Element}

\* **@return** Das gleichnamige {@link ConfigElement Konfigurations-Element}, oder

\* {@code null}, falls keines Vorhanden.

\*/

public Field getField(String name) {

for (Field f : fields) {

if (f.getName().equals(name))

return f;

}

return null;

}

/\*\*

\* Fügt das zu registrierende {@link ConfigElement} in {@link #fields die Liste

\* aller Konfigurations-Elemente} ein, wenn es über die Annotation

\* {@link ConfigElement} verfügt.

\*

\*

\* **@param** configElement Das zu registrierende {@link ConfigElement}

\* **@throws** IOException Sollte die Datei, die zu registrieren versucht wird,

\* nicht die Annotation {@link ConfigElement} besitzen, wird

\* eine IOException mit der Nachricht: "Not the right

\* annotation argument.", ausgegeben.

\*/

public void register(Field configElement) throws IOException {

if (configElement.getAnnotation(*ConfigElement*.class) == null)

throw new IOException("Not the right annotation argument.");

fields.add(configElement);

}

/\*\*

\* Registriert alle {@link Field Felder} der {@link Class Klasse}, die über die

\* Annotation {@link ConfigElement} verfügen über die Methode

\* {@link #register(Field)}.

\*

\* **@param** c Die {@link Class Klasse}, deren {@link Field Felder}, welche die

\* Annotation {@link ConfigElement} tragen, registriert werden sollen.

\* **@throws** IOException Sollte ein Fehler beim Registrieren der Felder auftreten.

\*/

public void register(Class c) throws IOException {

for (Field f : c.getFields()) {

if (f.getAnnotation(*ConfigElement*.class) != null)

this.register(f);

}

}

/\*\*

\* Lädt die unter {@link ConfigElement#defaultValue() dem Standartwert}

\* mitgegebenen Werte in die jeweiligen Felder, sollten diese nicht

\* standartmäßig über einen Wert verfügen.

\*

\* **@implNote** Diese Methode Funktioniert nur bedingt. Daher ist es ratsam, die

\* Felder direkt zu initialisieren.

\*/

public void loadDefault() {

this.fields.forEach(r -> {

boolean a = r.isAccessible();

r.setAccessible(true);

try {

if (r.get(this) == null) {

*ConfigElement* e = r.getAnnotation(*ConfigElement*.class);

String dv = e.defaultValue();

if (e.elementClass().equals(Level.class))

r.set(this, Level.*parse*(dv));

else if (e.elementClass().equals(Boolean.class))

r.set(this, Boolean.*parseBoolean*(dv));

else if (e.elementClass().equals(Integer.class))

r.set(this, Integer.*parseInt*(dv));

else if (e.elementClass().equals(String.class))

r.set(this, dv);

else

r.set(this, e.elementClass().cast(dv));

}

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

}

r.setAccessible(a);

});

}

/\*\*

\* <p>

\* Lädt eine Konfigurations Datei ein. Dabei werden die {@link #fields Elemente

\* aus der Liste aller Konfigurations-Elemente} auf den für sie vermerkten Wert

\* gesetzt.

\* </p>

\*

\* <p>

\* Die Konfigurationsdatei ist in {@code XML} zu schreiben. Dabei umschließt der

\* Parameter {@code config} die gesammte Konfigurationsdatei. Unter dem

\* Parameter {@code fields} können mithilfe des Parameters {@code field} und dem

\* folgenden Parameter {@code parameter} die einzelnen Konfigurationswerte

\* gesetzt werden. Dabei ist {@code name} der Name des zu setztenden Feldes.

\* {@code value} gibt den Wert an, auf den es gesetzt wird. Unter

\* {@code default} kann der Standartwert als orientierung angegeben werden und

\* unter {@code type} ist die Klasse vermerkt, in welche das {@code value}

\* konvertiert wird.

\* </p>

\*

\* **@param** input Der Pfad zu der einzulesenden Datei.

\* **@throws** SAXException Sollte ein Fehler in der Struktur der Datei vorliegen.

\* **@throws** IOException Sollte ein Fehler beim Einlesen der Datei aufgtreten

\*/

public void load(String input) throws SAXException, IOException {

XMLReader xmlReader = XMLReaderFactory.*createXMLReader*();

InputSource inputSource = new InputSource(new FileReader(input));

xmlReader.setContentHandler(new FieldHandler());

xmlReader.parse(inputSource);

}

/\*\*

\* Exportiert die Daten der {@link #fields Elemente aus der Liste aller

\* Konfigurations-Elemente} in eine Datei nach dem unter {@link #load(String)}

\* erklärten Aufbau. Diese Datei ist von der Methode {@link #load(String)}

\* wieder einlesbar.

\*

\* **@param** output Der Pfad zu dem Exportiert wird.

\* **@throws** IOException Sollte es nicht möglixh sein an den angegebenen Pfad zu

\* Schreiben, oder sollte output gleich {@code null} sein

\*/

public void save(File output) throws IOException {

if (output == null)

throw new IOException("No file to write to!");

FileWriter fw;

BufferedWriter bw;

fw = new FileWriter(output);

bw = new BufferedWriter(fw);

bw.append("<config>");

bw.newLine();

bw.append(" <fields>");

bw.newLine();

fields.forEach(e -> {

try {

bw.append(" <field>");

bw.newLine();

bw.append(" <parameter>");

bw.newLine();

bw.append(" <name>" + e.getName() + "</name>");

bw.newLine();

bw.append(" <value>" + e.get(this) + "</value>");

bw.newLine();

bw.append(" <default>" + e.getAnnotation(*ConfigElement*.class).defaultValue() + "</default>");

bw.newLine();

bw.append(" <type>" + e.getAnnotation(*ConfigElement*.class).elementClass() + "</type>");

bw.newLine();

bw.append(" </parameter>");

bw.newLine();

bw.append(" </field>");

bw.newLine();

} catch (IllegalArgumentException | IOException | IllegalAccessException e1) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Erstellen der Config datei!", e1);

}

});

bw.append(" </fields>");

bw.newLine();

bw.append("</config>");

bw.close();

}

public void createMenuTree(TreeView<String> tree, VBox configurations) {

CheckBoxTreeItem root = new CheckBoxTreeItem(References.*language*.getString("config.location"));

tree.setCellFactory(CheckBoxTreeCell.<String>*forTreeView*());

tree.setRoot(root);

for (Field f : this.fields) {

if (f.getAnnotation(*ConfigElement*.class) == null)

continue;

*ConfigElement* e = f.getAnnotation(*ConfigElement*.class);

CheckBoxTreeItem actual = null;

for (String s : e.location().split("\\.")) {

if (actual == null) {

actual = root;

continue;

}

boolean found = false;

for (Object ti : actual.getChildren()) {

if (((String) ((TreeItem) ti).getValue())

.equalsIgnoreCase(References.*language*.getString(s + ".location"))) {

actual = (CheckBoxTreeItem) ti;

found = true;

break;

}

}

if (!found) {

TreeItem nti = new CheckBoxTreeItem<String>(References.*language*.getString(s + ".location"));

actual.getChildren().add(0, nti);

actual = (CheckBoxTreeItem) nti;

}

}

if (e.elementClass().equals(Boolean.class)) {

CheckBoxTreeItem cb = new CheckBoxTreeItem(References.*language*.getString(e.name()));

// cb.setTooltip(new Tooltip(References.language.getString(e.description())));

cb.addEventHandler(ActionEvent.***ANY***, r -> {

this.onConfigChanged();

});

try {

cb.setSelected(f.getBoolean(null));

} catch (IllegalArgumentException e2) {

e2.printStackTrace();

} catch (IllegalAccessException e2) {

e2.printStackTrace();

}

cb.selectedProperty().addListener((obs, oldValue, newValue) -> {

try {

f.setBoolean(null, newValue);

} catch (IllegalArgumentException | IllegalAccessException e1) {

***LOGGER***.severe("Error while updating config!");

}

});

cb.addEventHandler(ActionEvent.***ACTION***, event -> {

try {

f.setBoolean(null, cb.isSelected());

} catch (IllegalArgumentException e1) {

e1.printStackTrace();

} catch (IllegalAccessException e1) {

e1.printStackTrace();

}

});

// configurationTree.setCellFactory(CheckBoxTreeCell.<String>forTreeView());

actual.getChildren().add(cb);

}

}

tree.addEventHandler(MouseEvent.***MOUSE\_CLICKED***, (event -> {

CheckBoxTreeItem selected = (CheckBoxTreeItem) tree.getSelectionModel().getSelectedItem();

if (selected == null)

return;

String location = (String) selected.getValue();

CheckBoxTreeItem<String> actual = selected;

while (actual != null) {

actual = (CheckBoxTreeItem<String>) actual.getParent();

if (actual != null)

location = actual.getValue() + "." + location;

}

if (selected != null) {

configurations.getChildren().clear();

for (Field f : this.fields) {

if (f.getAnnotation(*ConfigElement*.class) == null)

continue;

*ConfigElement* e = f.getAnnotation(*ConfigElement*.class);

String fieldlocation = "";

for (String s : e.location().split("\\."))

if (!s.isEmpty())

fieldlocation = fieldlocation + (fieldlocation == "" ? "" : ".")

+ References.*language*.getString(s + ".location");

if (location.equalsIgnoreCase(fieldlocation + "." + References.*language*.getString(e.name()))) {

TextArea ta = new TextArea(References.*language*.getString(e.description()));

ta.setEditable(false);

ta.setWrapText(true);

configurations.getChildren()

.addAll(new Label(References.*language*.getString("description.text") + ":"), ta);

}

if (!fieldlocation.equalsIgnoreCase(location))

continue;

if (e.elementClass().equals(Integer.class)) {

Spinner cb = new Spinner();

cb.setTooltip(new Tooltip(References.*language*.getString(e.description())));

cb.setEditable(true);

try {

if (f.getName().equals("runs") || f.getName().equals("newCalculating")

|| f.getName().equals("improvingOfCalculation"))

cb.setValueFactory(new SpinnerValueFactory.IntegerSpinnerValueFactory(1,

Integer.***MAX\_VALUE***, f.getInt(null)));

else

cb.setValueFactory(new SpinnerValueFactory.IntegerSpinnerValueFactory(-1,

Integer.***MAX\_VALUE***, f.getInt(null)));

cb.getValueFactory().valueProperty().addListener((o, oldValue, newValue) -> {

try {

f.set(null, newValue);

} catch (IllegalArgumentException e1) {

e1.printStackTrace();

} catch (IllegalAccessException e1) {

e1.printStackTrace();

}

this.onConfigChanged();

});

} catch (IllegalArgumentException | IllegalAccessException e3) {

e3.printStackTrace();

}

Label l = new Label((References.*language*.getString(e.name()) + ":"));

l.autosize();

configurations.getChildren().addAll(l, cb);

} else if (e.elementClass().equals(String.class)) {

TextField cb = new TextField();

cb.setTooltip(new Tooltip(References.*language*.getString(e.description())));

try {

cb.setText((String) f.get(null));

} catch (IllegalArgumentException e2) {

e2.printStackTrace();

} catch (IllegalAccessException e2) {

e2.printStackTrace();

}

cb.addEventHandler(KeyEvent.***KEY\_RELEASED***, events -> {

try {

f.set(null, cb.getText());

this.onConfigChanged();

} catch (IllegalArgumentException e1) {

e1.printStackTrace();

} catch (IllegalAccessException e1) {

e1.printStackTrace();

}

});

Label l = new Label((References.*language*.getString(e.name()) + ":"));

l.setMinWidth(1000);

l.autosize();

configurations.getChildren().addAll(l, cb);

}

}

}

this.onConfigChanged();

}));

}

public void onConfigChanged() {

GUIManager.*getInstance*().updateInputView();

if (!Distributor.*getInstance*().ignore().toString().equalsIgnoreCase(Config.*ignoreStudent* + "|"))

Distributor.*getInstance*().setIgnoreMark(Config.*ignoreStudent*);

Distributor.*getInstance*().updateStandartReaders();

}

} //Ende der Klasse ConfigManager

package de.juhu.config;

import java.lang.reflect.Field;

import java.util.logging.Level;

import org.xml.sax.Attributes;

import org.xml.sax.ContentHandler;

import org.xml.sax.Locator;

import org.xml.sax.SAXException;

/\*\*

\* Diese Klasse organisiert das Einlesen der Config Datei.

\*

\* **@author** Juhu1705

\* **@category** Config

\* **@version** 1.0

\* **@see** ContentHandler

\* **@see** ConfigManager#load(String)

\*/

public class FieldHandler implements ContentHandler {

/\*\*

\* Zwischenspeicherung für die einzulesenden Daten.

\*/

private String value, defaultValue, type, name, currentValue;

/\*\*

\* Speichert die eingelesenden Werte zur Bearbeitung.

\*/

*@Override*

public void characters(char[] arg0, int arg1, int arg2) throws SAXException {

currentValue = new String(arg0, arg1, arg2);

}

*@Override*

public void endDocument() throws SAXException {

}

/\*\*

\* Lädt die Daten in das Programm.

\*/

*@Override*

public void endElement(String arg0, String arg1, String arg2) throws SAXException {

if (arg1.equals("name"))

name = currentValue;

if (arg1.equals("value"))

value = currentValue;

if (arg1.equals("default"))

defaultValue = currentValue;

if (arg1.equals("type"))

type = currentValue;

if (arg1.equals("field")) {

Field f = ConfigManager.*getInstance*().getField(name);

if (f == null)

return;

boolean a = f.isAccessible();

f.setAccessible(true);

try {

*ConfigElement* e = f.getAnnotation(*ConfigElement*.class);

if (e.elementClass().equals(Level.class))

f.set(this, Level.*parse*(value));

else if (e.elementClass().equals(Integer.class))

f.set(this, Integer.*parseInt*(value));

else if (e.elementClass().equals(Boolean.class))

f.set(this, Boolean.*parseBoolean*(value));

else

f.set(this, e.elementClass().cast(value));

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

}

f.setAccessible(a);

}

}

*@Override*

public void endPrefixMapping(String arg0) throws SAXException {

}

*@Override*

public void ignorableWhitespace(char[] arg0, int arg1, int arg2) throws SAXException {

}

*@Override*

public void processingInstruction(String arg0, String arg1) throws SAXException {

}

*@Override*

public void setDocumentLocator(Locator arg0) {

}

*@Override*

public void skippedEntity(String arg0) throws SAXException {

}

*@Override*

public void startDocument() throws SAXException {

}

*@Override*

public void startElement(String arg0, String arg1, String arg2, Attributes arg3) throws SAXException {

}

*@Override*

public void startPrefixMapping(String arg0, String arg1) throws SAXException {

}

} //Ende der Klasse FieldHandler

## Dateimanagment (In- und Export)

package de.juhu.dateimanager;

import static de.juhu.util.References.***LOGGER***;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

import java.util.logging.Level;

/\*\*

\* Verwaltet das Exportieren eines {@link WriteableContent} nach CSV an den

\* gegebenen Pfad.

\*

\* **@author** Juhu1705

\* **@category** Export

\*/

public class CSVExporter {

protected CSVExporter() {

}

public static void writeCSV(String pathfile, WriteableContent toWrite) throws IOException {

FileWriter fileWriter = null;

BufferedWriter writer = null;

try {

fileWriter = new FileWriter(new File(pathfile + ".csv"));

writer = new BufferedWriter(fileWriter);

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Erstellen einer .csv Datei", e);

}

toWrite.writeCSV(writer);

writer.close();

}

public static void writeCSV(String pathfile, WriteableContent... toWrite) throws IOException {

FileWriter fileWriter = null;

BufferedWriter writer = null;

try {

fileWriter = new FileWriter(new File(pathfile + ".csv"));

writer = new BufferedWriter(fileWriter);

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Erstellen einer .csv Datei", e);

}

for (WriteableContent writeable : toWrite) {

writeable.writeCSV(writer);

try {

writer.newLine();

writer.newLine();

} catch (IOException e) {

***LOGGER***.log(Level.***WARNING***, "Exception caused while exporting data: ", e);

}

}

writer.close();

}

} //Ende der Klasse CSVExporter

package de.juhu.dateimanager;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.net.URISyntaxException;

import java.util.logging.Level;

import de.juhu.util.References;

/\*\*

\* Importiert eine CSV Datei aus dem mitgegebenen Pfad.

\*

\*

\* **@author** Juhu1705

\* **@category** Import

\*

\*/

public class CSVImporter {

protected CSVImporter() {

}

public static WriteableContent readCSV(String pathfile) throws IOException, URISyntaxException {

WriteableContent writeable = new WriteableContent();

InputStreamReader fileReader = new InputStreamReader(*getInput*(pathfile), "UTF8");

BufferedReader reader = new BufferedReader(fileReader);

String zeile = "";

int y = 0, x;

while (reader.ready() && (zeile = reader.readLine()) != null) {

x = 0;

String parameter = "";

for (char c : zeile.toCharArray()) {

if (c == ';') {

writeable.addCell(new Vec2i(x++, y), parameter);

parameter = "";

} else

parameter += c;

}

writeable.addCell(new Vec2i(x++, y), parameter);

y++;

}

reader.close();

return writeable;

}

private static InputStream getInput(String name) throws URISyntaxException, FileNotFoundException {

InputStream output;

output = ExcelImporter.class.getClassLoader().getResourceAsStream(name);

if (output == null) {

try {

output = new FileInputStream(new File(name));

} catch (FileNotFoundException e) {

References.***LOGGER***.log(Level.***SEVERE***, "", e);

}

}

return output;

}

} // Ende der Klasse CSVImporter

package de.juhu.dateimanager;

import static de.juhu.util.References.***LOGGER***;

import java.io.File;

import java.io.FileOutputStream;

import java.io.IOException;

import java.util.List;

import java.util.logging.Level;

import org.apache.poi.hssf.usermodel.HSSFSheet;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;

import org.apache.poi.xssf.usermodel.XSSFSheet;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;

/\*\*

\* Verwaltet das Exportieren eines {@link WriteableContent} in ein Excel Format

\* in den gegebenen Pfad.

\*

\* **@author** Juhu1705

\* **@category** Export

\*/

public class ExcelExporter {

public static void writeXLS(String pathfile, WriteableContent toWrite) {

HSSFWorkbook workbook = new HSSFWorkbook();

HSSFSheet sheet = workbook.createSheet(toWrite.getName());

int rownumber = 0;

toWrite.writeXLS(workbook, sheet, rownumber);

File file = new File(pathfile + ".xls");

file.getParentFile().mkdirs();

try {

FileOutputStream outFile = new FileOutputStream(file);

workbook.write(outFile);

outFile.close();

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Exportieren einer .xls Datei", e);

}

try {

workbook.close();

} catch (IOException e) {

e.printStackTrace();

}

}

public static void writeXLSX(String pathfile, WriteableContent toWrite) {

XSSFWorkbook workbook = new XSSFWorkbook();

XSSFSheet sheet = workbook.createSheet(toWrite.getName());

int rownumber = 0;

toWrite.writeXLSX(workbook, sheet, rownumber);

File file = new File(pathfile + ".xlsx");

file.getParentFile().mkdirs();

try {

FileOutputStream outFile = new FileOutputStream(file);

workbook.write(outFile);

outFile.close();

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Exportieren einer .xlsx Datei", e);

}

try {

workbook.close();

} catch (IOException e) {

e.printStackTrace();

}

}

public static void writeXLS(String pathfile, WriteableContent... toWrite) {

HSSFWorkbook workbook = new HSSFWorkbook();

int rownumber = 0;

for (WriteableContent writeable : toWrite) {

HSSFSheet sheet = workbook.createSheet(writeable.getName());

writeable.writeXLS(workbook, sheet, rownumber);

}

File file = new File(pathfile + ".xls");

file.getParentFile().mkdirs();

try {

FileOutputStream outFile = new FileOutputStream(file);

workbook.write(outFile);

outFile.close();

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Exportieren einer .xls Datei", e);

}

try {

workbook.close();

} catch (IOException e) {

e.printStackTrace();

}

}

public static void writeXLS(String pathfile, List<WriteableContent> toWrite) throws IOException {

HSSFWorkbook workbook = new HSSFWorkbook();

int rownumber = 0;

for (WriteableContent writeable : toWrite) {

HSSFSheet sheet = workbook.createSheet(writeable.getName());

writeable.writeXLS(workbook, sheet, rownumber);

}

File file = new File(pathfile + ".xls");

file.getParentFile().mkdirs();

try {

FileOutputStream outFile = new FileOutputStream(file);

workbook.write(outFile);

outFile.close();

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Exportieren einer .xls Datei", e);

}

workbook.close();

}

public static void writeXLSX(String pathfile, WriteableContent... toWrite) throws IOException {

XSSFWorkbook workbook = new XSSFWorkbook();

int rownumber = 0;

for (WriteableContent writeable : toWrite) {

XSSFSheet sheet = workbook.createSheet(writeable.getName());

writeable.writeXLSX(workbook, sheet, rownumber);

}

File file = new File(pathfile + ".xlsx");

file.getParentFile().mkdirs();

try {

FileOutputStream outFile = new FileOutputStream(file);

workbook.write(outFile);

outFile.close();

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Exportieren einer .xlsx Datei", e);

}

workbook.close();

}

public static void writeXLSX(String pathfile, List<WriteableContent> toWrite) throws IOException {

XSSFWorkbook workbook = new XSSFWorkbook();

int rownumber = 0;

for (WriteableContent writeable : toWrite) {

XSSFSheet sheet = workbook.createSheet(writeable.getName());

writeable.writeXLSX(workbook, sheet, rownumber);

}

workbook.setForceFormulaRecalculation(true);

File file = new File(pathfile + ".xlsx");

// file.getParentFile().mkdirs();

try {

FileOutputStream outFile = new FileOutputStream(file);

workbook.write(outFile);

outFile.close();

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Exportieren einer .xlsx Datei", e);

}

workbook.close();

}

} // Ende der Klasse ExcelExporter

package de.juhu.dateimanager;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.io.InputStream;

import java.net.URISyntaxException;

import java.util.ArrayList;

import java.util.List;

import java.util.logging.Level;

import org.apache.poi.hssf.usermodel.HSSFSheet;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;

import org.apache.poi.ss.usermodel.Cell;

import org.apache.poi.ss.usermodel.Row;

import org.apache.poi.ss.usermodel.Sheet;

import org.apache.poi.xssf.usermodel.XSSFSheet;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import de.juhu.util.References;

/\*\*

\* Importiert eine Excel Datei aus dem mitgegebenen Pfad.

\*

\*

\* **@author** Juhu1705

\* **@category** Import

\*

\*/

public class ExcelImporter {

public static WriteableContent readXLS(String pathfile) throws IOException, URISyntaxException {

WriteableContent input = new WriteableContent();

InputStream inputStream = *getInput*(pathfile);

HSSFWorkbook w;

HSSFSheet sheet = (w = new HSSFWorkbook(inputStream)).getSheetAt(0);

for (Row row : sheet) {

for (Cell cell : row) {

switch (cell.getCellType()) {

case ***BLANK***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

case ***BOOLEAN***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + cell.getBooleanCellValue());

break;

case ***ERROR***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

case ***FORMULA***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + w.getCreationHelper().createFormulaEvaluator().evaluate(cell).getNumberValue());

break;

case ***NUMERIC***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + cell.getNumericCellValue());

break;

case ***STRING***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

cell.getStringCellValue());

break;

case ***\_NONE***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

default:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

}

}

}

inputStream.close();

w.close();

return input;

}

public static WriteableContent readXLSX(String pathfile) throws IOException, URISyntaxException {

WriteableContent input = new WriteableContent();

InputStream inputStream = *getInput*(pathfile);

XSSFWorkbook w;

XSSFSheet sheet = (w = new XSSFWorkbook(inputStream)).getSheetAt(0);

for (Row row : sheet) {

for (Cell cell : row) {

switch (cell.getCellType()) {

case ***BLANK***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

case ***BOOLEAN***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + cell.getBooleanCellValue());

break;

case ***ERROR***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

case ***FORMULA***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + w.getCreationHelper().createFormulaEvaluator().evaluate(cell).getNumberValue());

break;

case ***NUMERIC***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + cell.getNumericCellValue());

break;

case ***STRING***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

cell.getStringCellValue());

break;

case ***\_NONE***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

default:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

}

}

}

inputStream.close();

w.close();

return input;

}

public static List<WriteableContent> readXLSImproved(String pathfile) throws IOException, URISyntaxException {

ArrayList<WriteableContent> dataContent = new ArrayList<>();

InputStream inputStream = *getInput*(pathfile);

HSSFWorkbook w;

for (Sheet sheetIn : (w = new HSSFWorkbook(inputStream))) {

WriteableContent input = new WriteableContent(sheetIn.getSheetName());

for (Row row : sheetIn) {

for (Cell cell : row) {

switch (cell.getCellType()) {

case ***BLANK***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

case ***BOOLEAN***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + cell.getBooleanCellValue());

break;

case ***ERROR***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

case ***FORMULA***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + w.getCreationHelper().createFormulaEvaluator().evaluate(cell).getNumberValue());

break;

case ***NUMERIC***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + cell.getNumericCellValue());

break;

case ***STRING***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

cell.getStringCellValue());

break;

case ***\_NONE***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

default:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

}

}

}

dataContent.add(input);

}

inputStream.close();

w.close();

return dataContent;

}

public static List<WriteableContent> readXLSXImproved(String pathfile) throws IOException, URISyntaxException {

ArrayList<WriteableContent> dataContent = new ArrayList<>();

InputStream inputStream = *getInput*(pathfile);

XSSFWorkbook w;

for (Sheet sheet : (w = new XSSFWorkbook(inputStream))) {

WriteableContent input = new WriteableContent(sheet.getSheetName());

for (Row row : sheet) {

for (Cell cell : row) {

switch (cell.getCellType()) {

case ***BLANK***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

case ***BOOLEAN***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + cell.getBooleanCellValue());

break;

case ***ERROR***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

case ***FORMULA***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + w.getCreationHelper().createFormulaEvaluator().evaluate(cell).getNumberValue());

break;

case ***NUMERIC***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

"" + cell.getNumericCellValue());

break;

case ***STRING***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()),

cell.getStringCellValue());

break;

case ***\_NONE***:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

default:

input.addCell(new Vec2i(cell.getAddress().getColumn(), cell.getRowIndex()), "");

break;

}

}

}

dataContent.add(input);

}

inputStream.close();

w.close();

return dataContent;

}

private static InputStream getInput(String name) {

InputStream output;

output = ExcelImporter.class.getClassLoader().getResourceAsStream(name);

if (output == null) {

try {

output = new FileInputStream(new File(name));

} catch (FileNotFoundException e) {

References.***LOGGER***.log(Level.***SEVERE***, "", e);

}

}

return output;

}

} // Ende der Klasse ExcelImporter

package de.juhu.dateimanager;

import static de.juhu.util.References.***LOGGER***;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

import java.util.logging.Level;

import de.juhu.util.References;

/\*\*

\* Schreibt den Log in den angegebenen Pfad.

\*

\* **@author** Juhu1705

\* **@category** Export

\*/

public class LogWriter {

public static void writeLog(String pathfile) {

FileWriter fileWriter = null;

BufferedWriter writer = null;

try {

fileWriter = new FileWriter(new File(pathfile + ".log"));

writer = new BufferedWriter(fileWriter);

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Erstellen einer .log Datei", e);

}

StringBuilder sw = References.***LOGGING\_HANDLER***.getLog();

String[] strings = sw.toString().split(" \n");

try {

for (String s : strings) {

writer.write(sw.toString());

writer.newLine();

}

} catch (IOException e1) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Erstellen einer .log Datei", e1);

}

try {

writer.close();

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Verarbeiten einer .log Datei", e);

}

}

} // Ende der Klasse LogWriter

package de.juhu.dateimanager;

import java.io.Serializable;

/\*\*

\* Representiert einen zweidimensionalen Vektor

\*

\* **@category** Math

\* **@version** 0.1

\* **@author** Juhu1705

\*/

public class Vec2i implements Serializable {

/\*\*

\* the x, y coordinates of the Vec2i

\*/

public int x, y;

/\*\*

\* Creates a Null-Vector

\*/

public Vec2i() {

this.x = 0;

this.y = 0;

}

/\*\*

\* x, y coordinates set to f

\*

\* **@param** f A int value

\*/

public Vec2i(int f) {

this.x = f;

this.y = f;

}

/\*\*

\* Sets x, y to x, y

\*

\* **@param** x

\* **@param** y

\*/

public Vec2i(int x, int y) {

this.x = x;

this.y = y;

}

/\*\*

\* Creates a new Vec2i with the values of the given Vec2i

\*

\* **@param** Vec2i A Vec2i

\*/

public Vec2i(Vec2i Vec2i) {

this.x = Vec2i.x;

this.y = Vec2i.y;

}

/\*\*

\*

\* **@return** Length of the vector

\*/

public int getLength() {

return (int) Math.*sqrt*(x \* x + y \* y);

}

/\*\*

\* Normalize the vector

\*

\* **@return** this

\*/

public Vec2i normalize() {

int length = this.getLength();

this.x /= length;

this.y /= length;

return this;

}

/\*\*

\* Multiplies this vector and the given Vec2i.

\*

\* **@param** Vec2i The vector to multiply this vector with.

\* **@return** A new Vec2i with the multiplied data.

\*/

public Vec2i mul(Vec2i Vec2i) {

return new Vec2i(x \* Vec2i.x, y \* Vec2i.y);

}

/\*\*

\* Multiplies this vector with the int f.

\*

\* **@param** f The int to multiply this vector with.

\* **@return** A new Vec2i with the multiplied data.

\*/

public Vec2i mul(int f) {

return new Vec2i(x \* f, y \* f);

}

/\*\*

\* Divided this with given Vec2i

\*

\* **@param** Vec2i The Vec2i to divide this Vec2i with

\* **@return** A new Vec2i with divided values

\*/

public Vec2i div(Vec2i Vec2i) {

return new Vec2i(x / Vec2i.x, y / Vec2i.y);

}

/\*\*

\* Divided this with given int

\*

\* **@param** f The int to divide this Vec2i with

\* **@return** A new Vec2i with divided values

\*/

public Vec2i div(int f) {

return new Vec2i(x / f, y / f);

}

/\*\*

\* Subtract given Vec2i from this

\*

\* **@param** Vec2i The Vec2i to subtract this Vec2i with

\* **@return** A new Vec2i with subtracted values

\*/

public Vec2i sub(Vec2i Vec2i) {

return new Vec2i(x - Vec2i.x, y - Vec2i.y);

}

/\*\*

\* Subtract given int from this

\*

\* **@param** f The int to subtract this Vec2i with

\* **@return** A new Vec2i with subtracted values

\*/

public Vec2i sub(int f) {

return new Vec2i(x - f, y - f);

}

/\*\*

\* Add given Vec2i to this

\*

\* **@param** Vec2i The Vec2i to add to this Vec2i

\* **@return** A new Vec2i with added values

\*/

public Vec2i add(Vec2i Vec2i) {

return new Vec2i(x + Vec2i.x, y + Vec2i.y);

}

/\*\*

\* Add given int to this

\*

\* **@param** f The int to add to this Vec2i

\* **@return** A new Vec2i with added values

\*/

public Vec2i add(int f) {

return new Vec2i(x + f, y + f);

}

/\*\*

\* Returns modulo

\*

\* **@param** Vec2i The Vec2i to modulo this Vec2i with

\* **@return** A new Vec2i with modulo values

\*/

public Vec2i mod(Vec2i Vec2i) {

return new Vec2i(x % Vec2i.x, y % Vec2i.y);

}

/\*\*

\* Returns modulo

\*

\* **@param** f The int to modulo this Vec2i with

\* **@return** A new Vec2i with modulo values

\*/

public Vec2i mod(int f) {

return new Vec2i(x % f, y % f);

}

/\*\*

\* Sets the x and y value

\*

\* **@param** x

\* **@param** y

\*/

public Vec2i set(int x, int y) {

this.x = x;

this.y = y;

return this;

}

*@Override*

public boolean equals(Object obj) {

return obj instanceof Vec2i ? this.x == ((Vec2i) obj).x && ((Vec2i) obj).y == this.y : super.equals(obj);

}

*@Override*

protected Vec2i clone() throws CloneNotSupportedException {

return new Vec2i(this);

}

*@Override*

public String toString() {

return "[" + x + "|" + y + "]";

}

} // Ende der Klasse Vec2i

package de.juhu.dateimanager;

import static de.juhu.util.References.***LOGGER***;

import static org.apache.poi.ss.usermodel.CellType.***STRING***;

import java.io.BufferedWriter;

import java.io.IOException;

import java.util.HashMap;

import java.util.List;

import java.util.Map.Entry;

import java.util.logging.Level;

import org.apache.poi.hssf.usermodel.HSSFCellStyle;

import org.apache.poi.hssf.usermodel.HSSFSheet;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;

import org.apache.poi.ss.usermodel.Cell;

import org.apache.poi.ss.usermodel.Row;

import org.apache.poi.xssf.usermodel.XSSFCellStyle;

import org.apache.poi.xssf.usermodel.XSSFSheet;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import de.juhu.util.CellStyles;

/\*\*

\* Diese Klasse dient als Schnittstelle zum Im- / Export.

\*

\* **@author** Juhu1705

\* **@category** Import / Export

\*/

public class WriteableContent {

private HashMap<Vec2i, String> lines = new HashMap<Vec2i, String>();

private String name;

public WriteableContent() {

this.name = "";

}

public WriteableContent(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

public WriteableContent setName(String name) {

this.name = name;

return this;

}

public WriteableContent addCell(Vec2i position, String content) {

lines.put(position, content);

return this;

}

public WriteableContent removeCell(Vec2i position) {

if (lines.containsKey(position))

lines.remove(position);

return this;

}

public WriteableContent addLine(Vec2i startPosition, String[] contents) {

for (int x = startPosition.x, i = 0; i < contents.length; x++, i++) {

lines.put(new Vec2i(x, startPosition.y), contents[i]);

}

return this;

}

public WriteableContent addGrid(Vec2i startPosition, String[][] grid) {

for (int x = startPosition.x, i = 0; i < grid.length; x++, i++) {

this.addLine(new Vec2i(x, startPosition.y), grid[i]);

}

return this;

}

public WriteableContent addLine(Vec2i startPosition, List<String> contents) {

for (int x = startPosition.x, i = 0; i < contents.size(); x++, i++) {

lines.put(new Vec2i(x, startPosition.y), contents.get(i));

}

return this;

}

public WriteableContent addListGrid(Vec2i startPosition, List<List<String>> contents) {

for (int x = startPosition.x, i = 0; i < contents.size(); x++, i++) {

this.addLine(new Vec2i(x, startPosition.y), contents.get(i));

}

return this;

}

public WriteableContent removeLine(Vec2i startPosition, int length) {

for (int x = startPosition.x, i = 0; i < length; x++, i++) {

lines.remove(new Vec2i(x, startPosition.y));

}

return this;

}

public String[][] getReverseGrid() {

Vec2i maxLength = this.getMaxLength();

String[][] cells = new String[maxLength.x + 1][maxLength.y + 1];

for (Entry<Vec2i, String> cell : lines.entrySet()) {

cells[cell.getKey().x][cell.getKey().y] = cell.getValue();

}

return cells;

}

public String[][] getGrid() {

Vec2i maxLength = this.getMaxLength();

String[][] cells = new String[maxLength.y + 1][maxLength.x + 1];

for (Entry<Vec2i, String> cell : lines.entrySet()) {

cells[cell.getKey().y][cell.getKey().x] = cell.getValue();

}

return cells;

}

public String getStringAt(Vec2i position) {

return this.lines.get(position);

}

public void writeXLS(HSSFWorkbook workbook, HSSFSheet sheet, int startingLineY) {

Vec2i maxLength = this.getMaxLength();

String[][] cells = this.getReverseGrid();

int rownumber = startingLineY + 1;

Row row;

Cell cell;

HSSFCellStyle style = CellStyles.*title*(workbook);

row = sheet.createRow(0);

for (int x = 0; x <= maxLength.x; x++) {

cell = row.createCell(x, ***STRING***);

cell.setCellValue(cells[x][0] == null ? "" : cells[x][0]);

cell.setCellStyle(style);

}

style = CellStyles.*normal2*(workbook);

row = sheet.createRow(1);

for (int x = 0; x <= maxLength.x; x++) {

cell = row.createCell(x, ***STRING***);

cell.setCellValue("");

cell.setCellStyle(style);

}

for (int y = 1; y <= maxLength.y; y++) {

if (y % 2 == 0) {

style = CellStyles.*normal2*(workbook);

row = sheet.createRow(rownumber + y);

for (int x = 0; x <= maxLength.x; x++) {

cell = row.createCell(x, ***STRING***);

cell.setCellValue(cells[x][y] == null ? "" : cells[x][y]);

cell.setCellStyle(style);

}

} else {

style = CellStyles.*normal1*(workbook);

row = sheet.createRow(rownumber + y);

for (int x = 0; x <= maxLength.x; x++) {

cell = row.createCell(x, ***STRING***);

cell.setCellValue(cells[x][y] == null ? "" : cells[x][y]);

cell.setCellStyle(style);

}

}

}

style = CellStyles.*up*(workbook);

row = sheet.createRow(maxLength.y + 1 + rownumber);

for (int x = 0; x <= maxLength.x; x++) {

cell = row.createCell(x, ***STRING***);

cell.setCellValue("");

cell.setCellStyle(style);

}

}

public void writeCSV(BufferedWriter writer) {

Vec2i maxLength = this.getMaxLength();

String[][] cells = this.getReverseGrid();

for (int y = 0; y <= maxLength.y; y++) {

try {

for (int x = 0; x <= maxLength.x; x++) {

if (cells[x][y] != null)

writer.append(cells[x][y] + ";");

}

writer.newLine();

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Failed to write CSV", e);

}

}

}

public void writeXLSX(XSSFWorkbook workbook, XSSFSheet sheet, int startingLineY) {

Vec2i maxLength = this.getMaxLength();

String[][] cells = this.getReverseGrid();

int rownumber = startingLineY + 1;

Row row;

Cell cell;

XSSFCellStyle style = CellStyles.*title*(workbook);

row = sheet.createRow(0);

for (int x = 0; x <= maxLength.x; x++) {

cell = row.createCell(x, ***STRING***);

cell.setCellValue(cells[x][0] == null ? "" : cells[x][0]);

cell.setCellStyle(style);

}

style = CellStyles.*normal2*(workbook);

row = sheet.createRow(1);

for (int x = 0; x <= maxLength.x; x++) {

cell = row.createCell(x, ***STRING***);

cell.setCellValue("");

cell.setCellStyle(style);

}

for (int y = 1; y <= maxLength.y; y++) {

if (y % 2 == 0) {

style = CellStyles.*normal2*(workbook);

row = sheet.createRow(rownumber + y);

for (int x = 0; x <= maxLength.x; x++) {

cell = row.createCell(x, ***STRING***);

cell.setCellValue(cells[x][y] == null ? "" : cells[x][y]);

cell.setCellStyle(style);

}

} else {

style = CellStyles.*normal1*(workbook);

row = sheet.createRow(rownumber + y);

for (int x = 0; x <= maxLength.x; x++) {

cell = row.createCell(x, ***STRING***);

cell.setCellValue(cells[x][y] == null ? "" : cells[x][y]);

cell.setCellStyle(style);

}

}

}

style = CellStyles.*up*(workbook);

row = sheet.createRow(maxLength.y + 1 + rownumber);

for (int x = 0; x <= maxLength.x; x++) {

cell = row.createCell(x, ***STRING***);

cell.setCellValue("");

cell.setCellStyle(style);

}

}

private Vec2i getMaxLength() {

Vec2i maxLength = new Vec2i();

for (Entry<Vec2i, String> cell : lines.entrySet()) {

maxLength.x = cell.getKey().x > maxLength.x ? cell.getKey().x : maxLength.x;

maxLength.y = cell.getKey().y > maxLength.y ? cell.getKey().y : maxLength.y;

}

return maxLength;

}

} // Ende der Klasse WriteableContent

## Verteilungsprozess

package de.juhu.distributor;

import java.io.Serializable;

import java.util.ArrayList;

import de.juhu.util.Config;

/\*\*

\* Diese Klasse bildet einen Kurs ab und beinhaltet alle diesbezüglich wichtigen

\* Informationen.

\*

\* **@version** 1.0

\* **@category** Distribution

\* **@author** Juhu1705

\* **@implNote** Ist {@link Comparable vergleichbar} mit anderen {@link Course

\* Kursen}.

\*/

public class Course implements Comparable<Course>, Serializable {

/\*\*

\* Alle Schüler, die dem Kurs zugeordnet sind.

\*/

private ArrayList<Student> students = new ArrayList<>();

/\*\*

\* Der Lehrer des Kurses / Das Lehrerkürzel

\*/

private String teacher;

/\*\*

\* Das Fach des Kurses

\*/

private String subject;

/\*\*

\* Die Maximale Schüleranzahl, die dieser Kurs beinhalten darf.

\*/

private int maxStudents;

/\*\*

\* Erzeugt einen Kurs, indem das maximale Schülerlimit, der Lehrer und das Fach

\* gesetzt sind.

\*

\* **@param** subject Das Fach des Kurses

\* **@param** teacher Der Lehrer des Kurses

\* **@param** maxStudents Die maximale Anzahl an Schülern, die in diesen Kurs passen

\*/

public Course(String subject, String teacher, int maxStudents) {

this(subject, teacher);

this.maxStudents = maxStudents;

}

/\*\*

\* Erzeugt einen Kurs, in dem der Lehrer und das Fach mitgegeben werden. Die

\* maximale Schülerzahl wird auf den in der Config mitgegebenen Standartwert

\* gesetzt.

\*

\* **@param** subject Das Fach des Kurses

\* **@param** teacher Der Schüler des Kurses

\*/

public Course(String subject, String teacher) {

this.teacher = teacher.toUpperCase();

this.subject = subject.toUpperCase();

this.maxStudents = Config.*normalStudentLimit*;

}

/\*\*

\* Erzeugt einen Kurs aus einem zwei langen Array, indem der Kurs und der Lehrer

\* gesetzt wird und die maximale Schülerzahl auf den in der Config mitgegebenen

\* Standartwert gesetzt wird.

\*

\* **@param** split An Position 0 das Fach und an Position 1 der Lehrer

\* **@throws** NullPointerException Wenn das gegebene Array weniger als zwei Strings

\* beinhaltet.

\*/

public Course(String[] split) throws NullPointerException {

this(split[0], split[1]);

}

/\*\*

\* **@return** Der Lehrer des Kurses / Das Lehrerkürzel

\*/

public String getTeacher() {

return this.teacher;

}

/\*\*

\* **@return** Das Fach des Kurses

\*/

public String getSubject() {

return this.subject;

}

/\*

\* **TODO** Bitte Überprüfen

\*/

*@Override*

public boolean equals(Object obj) {

if (obj instanceof String) {

return ((String) obj).equals(this.toString());

}

if (obj instanceof Course) {

return ((Course) obj).toString().equals(this.toString());

}

return super.equals(obj);

}

/\*\*

\* Überprüft, ob mehr Schüler im Kurs sind, als das {@link #maxStudents

\* Schülerlimit} zulässt. Ist das {@link #maxStudents Schülerlimit} {@code -1}

\* besitzt der Kurs kein Schülerlimit.

\*

\* **@return** {@code true}, wenn mehr Schüler im Kurs sind, als das Schülerlimit

\* erlaubt.

\*/

public boolean isFull() {

return ((this.maxStudents < this.students.size()) && (this.maxStudents != -1));

}

/\*\*

\* Gibt den Kurs als String zurück. Dazu wird das {@link #subject Fach} durch

\* einen "|" vom {@link #teacher Lehrer} getrennt. Der zurückgegebene String

\* sieht dann wiefolgt aus: {@link #subject

\* Fach}{@code + "|" + }{@link #teacher Lehrer}.

\*/

*@Override*

public String toString() {

return this.subject + "|" + this.teacher;

}

/\*\*

\* **@return** Gibt eine Liste aller Schüler im Kurs zurück.

\*/

public ArrayList<Student> getStudents() {

return this.students;

}

/\*\*

\* Überprüft, ob dieser Schüler diesem Kurs zugeordnet ist.

\*

\* **@param** student Der Schüler, bei dem überprüft werden soll, ob er im Kurs ist.

\* **@return** Ob der Schüler im Kurs ist.

\* **@implNote** Verwendet die Methode {@link ArrayList#contains(Object)}

\*/

public boolean contains(Student student) {

return this.students.contains(student);

}

/\*\*

\* Fügt einen Schüler hinzu, wenn er nicht bereits im Kurs ist, oder der

\* hinzuzufügene Schüler nicht {@code null} ist.

\*

\* **@param** student Der Schüler, der hinzugefügt werden soll.

\* **@return** Ob der Schüler erfolgreich hinzugefügt werden konnte.

\* **@implNote** Verwendet die Methode {@link ArrayList#add(Object)}

\*/

public boolean addStudent(Student student) {

if (student == null || this.contains(student))

return false;

return this.students.add(student);

}

/\*\*

\* Entfernt einen Schüler aus diesem Kurs.

\*

\* **@param** student Der Schüler, der entfernt werden soll.

\* **@return** Der Schüler, der entfernt wurde.

\* **@implNote** Verwendet die Methode {@link ArrayList#remove(int)}

\*/

public Student removeStudent(Student student) {

return this.students.remove(this.students.indexOf(student));

}

/\*\*

\* Gibt den Schüler zurück, der dem mitgegebene Index in der {@link #students

\* Liste der Schüler des Kurses} zugeordnet ist.

\*

\* **@param** i Der Index von dem der Schüler zurückgegeben werden soll.

\* **@return** Der Schüler, der an der Position i der Liste steht.

\*/

public Student getStudent(int i) {

return this.students.get(i);

}

/\*\*

\* **@return** {@link ArrayList#size()} der {@link #students Liste der Schüler des

\* Kurses}.

\*/

public int size() {

return this.students.size();

}

/\*\*

\* Gibt die Liste als String zurück. Die Schüler werden über

\* {@link Student#toString()} in einen String überführt.

\*

\* **@return** {@link ArrayList#toString()} der {@link #students Liste der Schüler

\* des Kurses}.

\*/

public String studentsToString() {

return this.students.toString();

}

*@Override*

public int compareTo(Course c) {

int i = this.subject.compareTo(c.subject);

if (i == 0)

i = this.teacher.compareTo(c.teacher);

return 0;

}

*@Override*

protected Object clone() throws CloneNotSupportedException {

return new Course(subject, teacher, maxStudents);

}

/\*\*

\* **@return** Die maximale Anzahl an Schülern, die diesem Kurs zugewiesen werden

\* dürfen.

\*/

public int getMaxStudentCount() {

return this.maxStudents;

}

/\*\*

\* Setzt den {@link #teacher Lehrer} des Kurses.

\*

\* **@param** teacher Der Lehrer des Kurses.

\*/

public void setTeacher(String teacher) {

this.teacher = teacher.toUpperCase();

}

/\*\*

\* Setzt das {@link #subject Fach} des Kurses.

\*

\* **@param** subject Das Fach des Kurses.

\*/

public void setSubject(String subject) {

this.subject = subject.toUpperCase();

}

/\*\*

\* Setzt das {@link #maxStudents Schülerlimit} des Kurses

\*

\* **@param** maxStudents Die maximale Anzahl an Schülern, die diesem Kurs

\* zugeordnet werden dürfen.

\*/

public void setStudentMax(int maxStudents) {

this.maxStudents = maxStudents;

}

} // Ende der Klasse Course

package de.juhu.distributor;

import java.io.Serializable;

import java.util.ArrayList;

import de.juhu.util.Config;

/\*\*

\* Diese Klasse bildet einen Kurs ab und beinhaltet alle diesbezüglich wichtigen

\* Informationen.

\*

\* **@version** 1.0

\* **@category** Distribution

\* **@author** Juhu1705

\* **@implNote** Ist {@link Comparable vergleichbar} mit anderen {@link Course

\* Kursen}.

\*/

public class Course implements Comparable<Course>, Serializable {

/\*\*

\* Alle Schüler, die dem Kurs zugeordnet sind.

\*/

private ArrayList<Student> students = new ArrayList<>();

/\*\*

\* Der Lehrer des Kurses / Das Lehrerkürzel

\*/

private String teacher;

/\*\*

\* Das Fach des Kurses

\*/

private String subject;

/\*\*

\* Die Maximale Schüleranzahl, die dieser Kurs beinhalten darf.

\*/

private int maxStudents;

/\*\*

\* Erzeugt einen Kurs, indem das maximale Schülerlimit, der Lehrer und das Fach

\* gesetzt sind.

\*

\* **@param** subject Das Fach des Kurses

\* **@param** teacher Der Lehrer des Kurses

\* **@param** maxStudents Die maximale Anzahl an Schülern, die in diesen Kurs passen

\*/

public Course(String subject, String teacher, int maxStudents) {

this(subject, teacher);

this.maxStudents = maxStudents;

}

/\*\*

\* Erzeugt einen Kurs, in dem der Lehrer und das Fach mitgegeben werden. Die

\* maximale Schülerzahl wird auf den in der Config mitgegebenen Standartwert

\* gesetzt.

\*

\* **@param** subject Das Fach des Kurses

\* **@param** teacher Der Schüler des Kurses

\*/

public Course(String subject, String teacher) {

this.teacher = teacher.toUpperCase();

this.subject = subject.toUpperCase();

this.maxStudents = Config.*normalStudentLimit*;

}

/\*\*

\* Erzeugt einen Kurs aus einem zwei langen Array, indem der Kurs und der Lehrer

\* gesetzt wird und die maximale Schülerzahl auf den in der Config mitgegebenen

\* Standartwert gesetzt wird.

\*

\* **@param** split An Position 0 das Fach und an Position 1 der Lehrer

\* **@throws** NullPointerException Wenn das gegebene Array weniger als zwei Strings

\* beinhaltet.

\*/

public Course(String[] split) throws NullPointerException {

this(split[0], split[1]);

}

/\*\*

\* **@return** Der Lehrer des Kurses / Das Lehrerkürzel

\*/

public String getTeacher() {

return this.teacher;

}

/\*\*

\* **@return** Das Fach des Kurses

\*/

public String getSubject() {

return this.subject;

}

/\*

\* **TODO** Bitte Überprüfen

\*/

*@Override*

public boolean equals(Object obj) {

if (obj instanceof String) {

return ((String) obj).equals(this.toString());

}

if (obj instanceof Course) {

return ((Course) obj).toString().equals(this.toString());

}

return super.equals(obj);

}

/\*\*

\* Überprüft, ob mehr Schüler im Kurs sind, als das {@link #maxStudents

\* Schülerlimit} zulässt. Ist das {@link #maxStudents Schülerlimit} {@code -1}

\* besitzt der Kurs kein Schülerlimit.

\*

\* **@return** {@code true}, wenn mehr Schüler im Kurs sind, als das Schülerlimit

\* erlaubt.

\*/

public boolean isFull() {

return ((this.maxStudents < this.students.size()) && (this.maxStudents != -1));

}

/\*\*

\* Gibt den Kurs als String zurück. Dazu wird das {@link #subject Fach} durch

\* einen "|" vom {@link #teacher Lehrer} getrennt. Der zurückgegebene String

\* sieht dann wiefolgt aus: {@link #subject

\* Fach}{@code + "|" + }{@link #teacher Lehrer}.

\*/

*@Override*

public String toString() {

return this.subject + "|" + this.teacher;

}

/\*\*

\* **@return** Gibt eine Liste aller Schüler im Kurs zurück.

\*/

public ArrayList<Student> getStudents() {

return this.students;

}

/\*\*

\* Überprüft, ob dieser Schüler diesem Kurs zugeordnet ist.

\*

\* **@param** student Der Schüler, bei dem überprüft werden soll, ob er im Kurs ist.

\* **@return** Ob der Schüler im Kurs ist.

\* **@implNote** Verwendet die Methode {@link ArrayList#contains(Object)}

\*/

public boolean contains(Student student) {

return this.students.contains(student);

}

/\*\*

\* Fügt einen Schüler hinzu, wenn er nicht bereits im Kurs ist, oder der

\* hinzuzufügene Schüler nicht {@code null} ist.

\*

\* **@param** student Der Schüler, der hinzugefügt werden soll.

\* **@return** Ob der Schüler erfolgreich hinzugefügt werden konnte.

\* **@implNote** Verwendet die Methode {@link ArrayList#add(Object)}

\*/

public boolean addStudent(Student student) {

if (student == null || this.contains(student))

return false;

return this.students.add(student);

}

/\*\*

\* Entfernt einen Schüler aus diesem Kurs.

\*

\* **@param** student Der Schüler, der entfernt werden soll.

\* **@return** Der Schüler, der entfernt wurde.

\* **@implNote** Verwendet die Methode {@link ArrayList#remove(int)}

\*/

public Student removeStudent(Student student) {

return this.students.remove(this.students.indexOf(student));

}

/\*\*

\* Gibt den Schüler zurück, der dem mitgegebene Index in der {@link #students

\* Liste der Schüler des Kurses} zugeordnet ist.

\*

\* **@param** i Der Index von dem der Schüler zurückgegeben werden soll.

\* **@return** Der Schüler, der an der Position i der Liste steht.

\*/

public Student getStudent(int i) {

return this.students.get(i);

}

/\*\*

\* **@return** {@link ArrayList#size()} der {@link #students Liste der Schüler des

\* Kurses}.

\*/

public int size() {

return this.students.size();

}

/\*\*

\* Gibt die Liste als String zurück. Die Schüler werden über

\* {@link Student#toString()} in einen String überführt.

\*

\* **@return** {@link ArrayList#toString()} der {@link #students Liste der Schüler

\* des Kurses}.

\*/

public String studentsToString() {

return this.students.toString();

}

*@Override*

public int compareTo(Course c) {

int i = this.subject.compareTo(c.subject);

if (i == 0)

i = this.teacher.compareTo(c.teacher);

return 0;

}

*@Override*

protected Object clone() throws CloneNotSupportedException {

return new Course(subject, teacher, maxStudents);

}

/\*\*

\* **@return** Die maximale Anzahl an Schülern, die diesem Kurs zugewiesen werden

\* dürfen.

\*/

public int getMaxStudentCount() {

return this.maxStudents;

}

/\*\*

\* Setzt den {@link #teacher Lehrer} des Kurses.

\*

\* **@param** teacher Der Lehrer des Kurses.

\*/

public void setTeacher(String teacher) {

this.teacher = teacher.toUpperCase();

}

/\*\*

\* Setzt das {@link #subject Fach} des Kurses.

\*

\* **@param** subject Das Fach des Kurses.

\*/

public void setSubject(String subject) {

this.subject = subject.toUpperCase();

}

/\*\*

\* Setzt das {@link #maxStudents Schülerlimit} des Kurses

\*

\* **@param** maxStudents Die maximale Anzahl an Schülern, die diesem Kurs

\* zugeordnet werden dürfen.

\*/

public void setStudentMax(int maxStudents) {

this.maxStudents = maxStudents;

}

} // Ende der Klasse Distributor

package de.juhu.distributor;

import java.io.Serializable;

import java.util.ArrayList;

import de.juhu.dateimanager.Vec2i;

import de.juhu.dateimanager.WriteableContent;

import de.juhu.util.Config;

import de.juhu.util.References;

/\*\*

\* Diese Klasse hällt alle weiteren für eine Verteilung interressanten von dem

\* ihr zugeordneten {@link #parent Speicher} zum abrufen bereit.

\*

\* **@version** 1.0

\* **@category** Distribution

\* **@author** Juhu1705

\* **@implements** {@link Serializable}

\* **@since** 0.0.2

\*

\*/

public class InformationSave implements Serializable {

/\*\*

\* Der diesem Informationsspeicher zugeordnete {@link Save Speicher}.

\*/

Save parent;

/\*\*

\* Die höchste Priorität, die im {@link #parent zugeordneten Speicher} vergeben

\* wurde.

\*/

private int highestPriority;

/\*\*

\* Die Rate der Berechnung des {@link #parent zugeordneten Speichers}.

\*/

private int rate;

/\*\*

\* Die Guete der Berechnung des {@link #parent zugeordneten Speicher}.

\*/

private double guete;

/\*\*

\* Die Anzahl der zugewiesenden Schüler, die im {@link #parent zugeordneten

\* Speicher} gespeichert sind. Alle Schüler, die nicht verteilt werden mussten

\* werden hier nicht mit aufgeführt.

\*/

private int studentCount;

/\*\*

\* Alle Schüler aus dem {@link #parent zugeordneten Speicher}, die nicht

\* zugewiesen werden konnten.

\*/

private ArrayList<Student> unallocatedStudents = new ArrayList<Student>();

/\*\*

\* Alle Schüler aus dem {@link #parent zugeordneten Speicher}, die die

\* schlechteste Priorität besitzen, aber zugeordnet werden konnten.

\*/

private ArrayList<Student> badPriorityStudents = new ArrayList<Student>();

/\*\*

\* <p>

\* Die Anzahl der Schüler aus dem {@link #parent zugeordneten Speicher} mit den

\* entsprechenden Prioritäten.

\* </p>

\* <ul>

\* <li>Prio 1: studentPriorities[0]</li>

\* <li>Prio 2: studentPriorities[1]</li>

\* <li>Prio n: studentPriorities[n - 1]</li>

\* <li>Anzahl der nicht zugewiesenden Schüler:

\* studentPriorities[studentPriorities.lenght - 1]</li>

\* </ul>

\*

\*/

private int[] studentPriorities;

/\*\*

\* Erzeugt einen Informationsspeicher, der an den mitgegebenen Speicher gebunden

\* ist. Nach dem erstellen dieses Speichers, kann die {@link #update()} funktion

\* genutzt werden, um alle anderen gespeicherten Werte den Werten des Speichers

\* anzugleichen.

\*

\* **@param** parent Der diesem Informationsspeicher zugeordnete {@link Save

\* Speicher}.

\*/

InformationSave(Save parent) {

this.parent = parent;

}

/\*\*

\* Schreibt die im Informationsspeicher gespeicherten Daten in eine

\* {@link WriteableContent exportierbare Tabelle} und gibt diese zurück.

\*

\* **@return** Eine exportierbare Tabelle mit den im Informationsspeicher

\* gespeicherten Daten.

\*/

public WriteableContent write() {

this.update();

WriteableContent information = new WriteableContent(References.*language*.getString("statistics.text"));

int line = 0;

information.addLine(new Vec2i(0, line++), new String[] { References.*language*.getString("informations.text"),

References.*language*.getString("value.text") });

information.addLine(new Vec2i(0, ++line), new String[] {

References.*language*.getString("highestpriority.text") + ": ", Integer.*toString*(this.highestPriority) });

information.addLine(new Vec2i(0, ++line), new String[] {

References.*language*.getString("calculationrate.text") + ": ", Integer.*toString*(this.rate) });

information.addLine(new Vec2i(0, ++line), new String[] {

References.*language*.getString("calculationgoodness.text") + ": ", Double.*toString*(this.guete) });

information.addLine(new Vec2i(0, ++line),

new String[] { References.*language*.getString("studentcount.text") + ": ",

Integer.*toString*(this.parent.getAllStudents().size()) });

information.addLine(new Vec2i(0, ++line),

new String[] { References.*language*.getString("coursecount.text") + ": ",

Integer.*toString*(this.parent.getAllCourses().size()) });

information.addLine(new Vec2i(0, ++line),

new String[] { References.*language*.getString("calculatedstudentcount.text") + ": ",

Integer.*toString*(this.getStudentCount()) });

line -= -2;

information.addLine(new Vec2i(0, line++), new String[] { References.*language*.getString("priorities.text"),

References.*language*.getString("countofstudents.text") });

for (int i = 0; i < studentPriorities.length - 1; i++) {

information.addLine(new Vec2i(0, line++),

new String[] { Integer.*toString*(i + 1), Integer.*toString*(this.studentPriorities[i]),

Double.*toString*((double) this.studentPriorities[i] / (double) this.getStudentCount()) });

}

line += 1;

if (this.studentPriorities[this.studentPriorities.length - 1] != 0) {

information.addLine(new Vec2i(0, line++),

new String[] { References.*language*.getString("unallocatedstudents.text"),

Integer.*toString*(this.studentPriorities[this.studentPriorities.length - 1]) });

information.addLine(new Vec2i(0, line++), new String[] { References.*language*.getString("student.text"),

References.*language*.getString("firstchoise.text") });

for (Student s : this.unallocatedStudents)

if (s.getCourses().length > 1)

information.addLine(new Vec2i(0, line++),

new String[] { s.toString(), s.getCourses()[0].toString() });

} else {

information.addLine(new Vec2i(0, line++),

new String[] { References.*language*.getString("nounallocated.text") });

}

line = line % 2 == 0 ? line + (line - 1) % 2 : line + line % 2;

information.addLine(new Vec2i(0, line++),

new String[] { References.*language*.getString("studentswithbadpriority") });

information.addLine(new Vec2i(0, line++), new String[] { References.*language*.getString("student.text"),

References.*language*.getString("firstchoise.text") });

for (Student s : this.badPriorityStudents)

if (s.getCourses().length > 1)

information.addLine(new Vec2i(0, line++), new String[] { s.toString(), s.getCourses()[0].toString() });

return information;

}

/\*\*

\* **@return** {@link #highestPriority Die höchste Priorität} der gespeicherten

\* Berechnung.

\*/

public int getHighestPriority() {

return this.highestPriority;

}

/\*\*

\* **@return** {@link #rate Die Rate} der gespeicherten Berechnung

\*/

public int getRate() {

return this.rate;

}

/\*\*

\* **@return** Die Anzahl der Schüler mit den zugeordneten Prioritäten.

\* **@see** Save#getStudentPriorities()

\* **@see** InformationSave#studentPriorities

\*/

public int[] getStudentPriorities() {

return this.studentPriorities;

}

/\*\*

\* **@return** Gibt alle Schüler mit der schlechtesten Priorität der gespeicherten

\* Berechnung zurück.

\*/

public ArrayList<Student> getBStudents() {

return this.badPriorityStudents;

}

/\*\*

\* **@return** Gibt alle Schüler zurück, die nicht zugewiesen werden konnten.

\*/

public ArrayList<Student> getUStudents() {

return this.unallocatedStudents;

}

/\*\*

\* **@return** Gibt die Anzahl der Berechneten Schüler zurück.

\*/

public int getStudentCount() {

return this.studentCount;

}

/\*\*

\* Aktualisiert die Daten dieses Inforationsspeichers mit den im {@link #parent

\* zugeordneten Speicher zu findenden Daten}.

\*/

public void update() {

/\*

\* Aktualisiert die Prioritäten aller Schüler.

\*/

parent.getAllStudents().forEach(s -> s.refreshPriority());

/\*

\* Aktualisiert die Liste der nicht zugewiesenden Schüler und berechnet die Zahö

\* der Schüler, die verteilt wurden.

\*/

this.unallocatedStudents.clear();

int sCount = 0;

for (Student s : this.parent.getAllStudents()) {

if (s.isMarked() || s.getPriority() > this.parent.getHighestPriorityWhithoutIntegerMax()

|| s.getPriority() < 0) {

s.mark();

this.unallocatedStudents.add(s);

}

if (s.getActiveCourse() == null)

++sCount;

if (s.getActiveCourse() != null && !s.getActiveCourse().equals(Distributor.*getInstance*().ignoredCourse))

sCount -= -1;

}

this.studentCount = sCount;

/\*

\* Aktualisiert die Anzahlen der Schüler mit den jeweiligen Prioritäten.

\*/

this.studentPriorities = this.parent.getStudentPriorities();

/\*

\* Aktualisiert die höchste Priorität dieser Verteilung.

\*/

this.highestPriority = parent.getHighestPriority();

/\*

\* Aktualisiert die Rate dieser Verteilung.

\*/

this.rate = parent.rate(this.highestPriority);

/\*

\* Aktualisiert die Liste der Schüler mit der schlechtesten Priorität.

\*/

this.badPriorityStudents = parent.getStudentsWithPriority(parent.getHighestPriorityWhithoutIntegerMax());

/\*

\* Gleicht die Anzahl der Schüler mit den Prioritäten veränderten Werten an.

\*/

if (this.unallocatedStudents.size() != this.studentPriorities[this.studentPriorities.length - 1])

this.studentPriorities[this.studentPriorities.length - 1] = this.unallocatedStudents.size();

if (!this.badPriorityStudents.isEmpty()

&& this.studentPriorities[this.studentPriorities.length - 2] != this.badPriorityStudents.size())

this.studentPriorities[this.studentPriorities.length - 2] = this.badPriorityStudents.size();

/\*

\* Aktualisiert die Güte der Verteilung.

\*/

this.updateGuete();

}

/\*\*

\* Aktualisiert die Güte dieses Speichers.

\*/

private void updateGuete() {

if (!Config.*useNewGoodness*)

this.guete = (double) this.parent.getAllStudents().size() / (double) this.rate;

else {

/\*

\* Addiert die Prioritäten der Schüler mit umgekehrten Prioritätszuordnung.

\*/

int studentPriorities = 0;

for (Student s : this.parent.getAllStudents()) {

if (s.getActiveCourse() == null) {

continue;

}

if (s.getActiveCourse() == null || !s.getActiveCourse().equals(Distributor.*getInstance*().ignoredCourse))

studentPriorities += this.translatePriority(s.getPriority());

}

/\*

\* Ermittelt die genutzte höchste Priorität.

\*/

int highest = this.parent.getHighestPriority();

if (highest > this.parent.getHighestPriorityWhithoutIntegerMax() || highest < 0)

highest = this.parent.getHighestPriorityWhithoutIntegerMax() + 10;

/\*

\* Ermittelt die Anzahl der Schüler.

\*/

int studentCount = 0;

for (Student s : this.parent.getAllStudents())

if (s.getActiveCourse() == null || !s.getActiveCourse().equals(Distributor.*getInstance*().ignoredCourse))

studentCount++;

/\*

\* Berechnet die Güte aus den ermittelten Werten.

\*/

this.guete = studentPriorities / ((double) highest \* (double) studentCount);

}

}

/\*\*

\* Wandelt die gegebene Priorität um. Die kleinste Prioritätszahl wird dabei zum

\* größten Wert und umgekehrt.

\*

\* **@param** priority Die Priorität, die umgewandelt werden soll.

\* **@return** Die umgewandelte Priorität.

\*/

private int translatePriority(int priority) {

if (priority > this.parent.getHighestPriorityWhithoutIntegerMax() || priority < 0)

return 0;

return this.parent.getHighestPriority() > this.parent.getHighestPriorityWhithoutIntegerMax()

|| this.parent.getHighestPriority() < 0

? this.parent.getHighestPriorityWhithoutIntegerMax() - priority + 2

: this.parent.getHighestPriority() - priority + 1;

}

/\*\*

\* **@return** Die Güte dieser Berechnung.

\*/

public double getGuete() {

return this.guete;

}

} // Ende der Klasse InformationSave

package de.juhu.distributor;

import java.io.Serializable;

import java.util.ArrayList;

import de.juhu.dateimanager.Vec2i;

import de.juhu.dateimanager.WriteableContent;

import de.juhu.util.Config;

import de.juhu.util.References;

/\*\*

\* Diese Klasse hällt alle weiteren für eine Verteilung interressanten von dem

\* ihr zugeordneten {@link #parent Speicher} zum abrufen bereit.

\*

\* **@version** 1.0

\* **@category** Distribution

\* **@author** Juhu1705

\* **@implements** {@link Serializable}

\* **@since** 0.0.2

\*

\*/

public class InformationSave implements Serializable {

/\*\*

\* Der diesem Informationsspeicher zugeordnete {@link Save Speicher}.

\*/

Save parent;

/\*\*

\* Die höchste Priorität, die im {@link #parent zugeordneten Speicher} vergeben

\* wurde.

\*/

private int highestPriority;

/\*\*

\* Die Rate der Berechnung des {@link #parent zugeordneten Speichers}.

\*/

private int rate;

/\*\*

\* Die Guete der Berechnung des {@link #parent zugeordneten Speicher}.

\*/

private double guete;

/\*\*

\* Die Anzahl der zugewiesenden Schüler, die im {@link #parent zugeordneten

\* Speicher} gespeichert sind. Alle Schüler, die nicht verteilt werden mussten

\* werden hier nicht mit aufgeführt.

\*/

private int studentCount;

/\*\*

\* Alle Schüler aus dem {@link #parent zugeordneten Speicher}, die nicht

\* zugewiesen werden konnten.

\*/

private ArrayList<Student> unallocatedStudents = new ArrayList<Student>();

/\*\*

\* Alle Schüler aus dem {@link #parent zugeordneten Speicher}, die die

\* schlechteste Priorität besitzen, aber zugeordnet werden konnten.

\*/

private ArrayList<Student> badPriorityStudents = new ArrayList<Student>();

/\*\*

\* <p>

\* Die Anzahl der Schüler aus dem {@link #parent zugeordneten Speicher} mit den

\* entsprechenden Prioritäten.

\* </p>

\* <ul>

\* <li>Prio 1: studentPriorities[0]</li>

\* <li>Prio 2: studentPriorities[1]</li>

\* <li>Prio n: studentPriorities[n - 1]</li>

\* <li>Anzahl der nicht zugewiesenden Schüler:

\* studentPriorities[studentPriorities.lenght - 1]</li>

\* </ul>

\*

\*/

private int[] studentPriorities;

/\*\*

\* Erzeugt einen Informationsspeicher, der an den mitgegebenen Speicher gebunden

\* ist. Nach dem erstellen dieses Speichers, kann die {@link #update()} funktion

\* genutzt werden, um alle anderen gespeicherten Werte den Werten des Speichers

\* anzugleichen.

\*

\* **@param** parent Der diesem Informationsspeicher zugeordnete {@link Save

\* Speicher}.

\*/

InformationSave(Save parent) {

this.parent = parent;

}

/\*\*

\* Schreibt die im Informationsspeicher gespeicherten Daten in eine

\* {@link WriteableContent exportierbare Tabelle} und gibt diese zurück.

\*

\* **@return** Eine exportierbare Tabelle mit den im Informationsspeicher

\* gespeicherten Daten.

\*/

public WriteableContent write() {

this.update();

WriteableContent information = new WriteableContent(References.*language*.getString("statistics.text"));

int line = 0;

information.addLine(new Vec2i(0, line++), new String[] { References.*language*.getString("informations.text"),

References.*language*.getString("value.text") });

information.addLine(new Vec2i(0, ++line), new String[] {

References.*language*.getString("highestpriority.text") + ": ", Integer.*toString*(this.highestPriority) });

information.addLine(new Vec2i(0, ++line), new String[] {

References.*language*.getString("calculationrate.text") + ": ", Integer.*toString*(this.rate) });

information.addLine(new Vec2i(0, ++line), new String[] {

References.*language*.getString("calculationgoodness.text") + ": ", Double.*toString*(this.guete) });

information.addLine(new Vec2i(0, ++line),

new String[] { References.*language*.getString("studentcount.text") + ": ",

Integer.*toString*(this.parent.getAllStudents().size()) });

information.addLine(new Vec2i(0, ++line),

new String[] { References.*language*.getString("coursecount.text") + ": ",

Integer.*toString*(this.parent.getAllCourses().size()) });

information.addLine(new Vec2i(0, ++line),

new String[] { References.*language*.getString("calculatedstudentcount.text") + ": ",

Integer.*toString*(this.getStudentCount()) });

line -= -2;

information.addLine(new Vec2i(0, line++), new String[] { References.*language*.getString("priorities.text"),

References.*language*.getString("countofstudents.text") });

for (int i = 0; i < studentPriorities.length - 1; i++) {

information.addLine(new Vec2i(0, line++),

new String[] { Integer.*toString*(i + 1), Integer.*toString*(this.studentPriorities[i]),

Double.*toString*((double) this.studentPriorities[i] / (double) this.getStudentCount()) });

}

line += 1;

if (this.studentPriorities[this.studentPriorities.length - 1] != 0) {

information.addLine(new Vec2i(0, line++),

new String[] { References.*language*.getString("unallocatedstudents.text"),

Integer.*toString*(this.studentPriorities[this.studentPriorities.length - 1]) });

information.addLine(new Vec2i(0, line++), new String[] { References.*language*.getString("student.text"),

References.*language*.getString("firstchoise.text") });

for (Student s : this.unallocatedStudents)

if (s.getCourses().length > 1)

information.addLine(new Vec2i(0, line++),

new String[] { s.toString(), s.getCourses()[0].toString() });

} else {

information.addLine(new Vec2i(0, line++),

new String[] { References.*language*.getString("nounallocated.text") });

}

line = line % 2 == 0 ? line + (line - 1) % 2 : line + line % 2;

information.addLine(new Vec2i(0, line++),

new String[] { References.*language*.getString("studentswithbadpriority") });

information.addLine(new Vec2i(0, line++), new String[] { References.*language*.getString("student.text"),

References.*language*.getString("firstchoise.text") });

for (Student s : this.badPriorityStudents)

if (s.getCourses().length > 1)

information.addLine(new Vec2i(0, line++), new String[] { s.toString(), s.getCourses()[0].toString() });

return information;

}

/\*\*

\* **@return** {@link #highestPriority Die höchste Priorität} der gespeicherten

\* Berechnung.

\*/

public int getHighestPriority() {

return this.highestPriority;

}

/\*\*

\* **@return** {@link #rate Die Rate} der gespeicherten Berechnung

\*/

public int getRate() {

return this.rate;

}

/\*\*

\* **@return** Die Anzahl der Schüler mit den zugeordneten Prioritäten.

\* **@see** Save#getStudentPriorities()

\* **@see** InformationSave#studentPriorities

\*/

public int[] getStudentPriorities() {

return this.studentPriorities;

}

/\*\*

\* **@return** Gibt alle Schüler mit der schlechtesten Priorität der gespeicherten

\* Berechnung zurück.

\*/

public ArrayList<Student> getBStudents() {

return this.badPriorityStudents;

}

/\*\*

\* **@return** Gibt alle Schüler zurück, die nicht zugewiesen werden konnten.

\*/

public ArrayList<Student> getUStudents() {

return this.unallocatedStudents;

}

/\*\*

\* **@return** Gibt die Anzahl der Berechneten Schüler zurück.

\*/

public int getStudentCount() {

return this.studentCount;

}

/\*\*

\* Aktualisiert die Daten dieses Inforationsspeichers mit den im {@link #parent

\* zugeordneten Speicher zu findenden Daten}.

\*/

public void update() {

/\*

\* Aktualisiert die Prioritäten aller Schüler.

\*/

parent.getAllStudents().forEach(s -> s.refreshPriority());

/\*

\* Aktualisiert die Liste der nicht zugewiesenden Schüler und berechnet die Zahö

\* der Schüler, die verteilt wurden.

\*/

this.unallocatedStudents.clear();

int sCount = 0;

for (Student s : this.parent.getAllStudents()) {

if (s.isMarked() || s.getPriority() > this.parent.getHighestPriorityWhithoutIntegerMax()

|| s.getPriority() < 0) {

s.mark();

this.unallocatedStudents.add(s);

}

if (s.getActiveCourse() == null)

++sCount;

if (s.getActiveCourse() != null && !s.getActiveCourse().equals(Distributor.*getInstance*().ignoredCourse))

sCount -= -1;

}

this.studentCount = sCount;

/\*

\* Aktualisiert die Anzahlen der Schüler mit den jeweiligen Prioritäten.

\*/

this.studentPriorities = this.parent.getStudentPriorities();

/\*

\* Aktualisiert die höchste Priorität dieser Verteilung.

\*/

this.highestPriority = parent.getHighestPriority();

/\*

\* Aktualisiert die Rate dieser Verteilung.

\*/

this.rate = parent.rate(this.highestPriority);

/\*

\* Aktualisiert die Liste der Schüler mit der schlechtesten Priorität.

\*/

this.badPriorityStudents = parent.getStudentsWithPriority(parent.getHighestPriorityWhithoutIntegerMax());

/\*

\* Gleicht die Anzahl der Schüler mit den Prioritäten veränderten Werten an.

\*/

if (this.unallocatedStudents.size() != this.studentPriorities[this.studentPriorities.length - 1])

this.studentPriorities[this.studentPriorities.length - 1] = this.unallocatedStudents.size();

if (!this.badPriorityStudents.isEmpty()

&& this.studentPriorities[this.studentPriorities.length - 2] != this.badPriorityStudents.size())

this.studentPriorities[this.studentPriorities.length - 2] = this.badPriorityStudents.size();

/\*

\* Aktualisiert die Güte der Verteilung.

\*/

this.updateGuete();

}

/\*\*

\* Aktualisiert die Güte dieses Speichers.

\*/

private void updateGuete() {

if (!Config.*useNewGoodness*)

this.guete = (double) this.parent.getAllStudents().size() / (double) this.rate;

else {

/\*

\* Addiert die Prioritäten der Schüler mit umgekehrten Prioritätszuordnung.

\*/

int studentPriorities = 0;

for (Student s : this.parent.getAllStudents()) {

if (s.getActiveCourse() == null) {

continue;

}

if (s.getActiveCourse() == null || !s.getActiveCourse().equals(Distributor.*getInstance*().ignoredCourse))

studentPriorities += this.translatePriority(s.getPriority());

}

/\*

\* Ermittelt die genutzte höchste Priorität.

\*/

int highest = this.parent.getHighestPriority();

if (highest > this.parent.getHighestPriorityWhithoutIntegerMax() || highest < 0)

highest = this.parent.getHighestPriorityWhithoutIntegerMax() + 10;

/\*

\* Ermittelt die Anzahl der Schüler.

\*/

int studentCount = 0;

for (Student s : this.parent.getAllStudents())

if (s.getActiveCourse() == null || !s.getActiveCourse().equals(Distributor.*getInstance*().ignoredCourse))

studentCount++;

/\*

\* Berechnet die Güte aus den ermittelten Werten.

\*/

this.guete = studentPriorities / ((double) highest \* (double) studentCount);

}

}

/\*\*

\* Wandelt die gegebene Priorität um. Die kleinste Prioritätszahl wird dabei zum

\* größten Wert und umgekehrt.

\*

\* **@param** priority Die Priorität, die umgewandelt werden soll.

\* **@return** Die umgewandelte Priorität.

\*/

private int translatePriority(int priority) {

if (priority > this.parent.getHighestPriorityWhithoutIntegerMax() || priority < 0)

return 0;

return this.parent.getHighestPriority() > this.parent.getHighestPriorityWhithoutIntegerMax()

|| this.parent.getHighestPriority() < 0

? this.parent.getHighestPriorityWhithoutIntegerMax() - priority + 2

: this.parent.getHighestPriority() - priority + 1;

}

/\*\*

\* **@return** Die Güte dieser Berechnung.

\*/

public double getGuete() {

return this.guete;

}

} // Ende der Klasse Reader

package de.juhu.distributor;

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

import de.juhu.dateimanager.Vec2i;

import de.juhu.dateimanager.WriteableContent;

import de.juhu.util.Config;

import de.juhu.util.MergeSort;

import de.juhu.util.References;

import de.juhu.util.Util;

/\*\*

\* Diese Klasse dient zur Speicherung berechneter Zuweisungen des

\* {@link Distributor Berechners} im Arbeitsspeicher und stellt zudem noch

\* Methoden zur Bearbeitung der gespeicherten Daten, sowie zum Exportieren der

\* Daten bereit.

\*

\*

\* **@version** 1.0

\* **@category** Distribution

\* **@author** Juhu1705

\* **@implements** {@link Comparable}, {@link Serializable}

\* **@since** 0.0.2

\*/

public class Save implements Comparable<Save>, Serializable {

/\*\*

\* Der {@link InformationSave Informationsspeicher} dieser Klasse, hier werden

\* interressante Zusatzinformationen hinterlegt.

\*/

private InformationSave informations;

/\*\*

\* Die Liste aller Schüler einer Berechnung inklusive der ignorierten Schüler.

\*/

private List<Student> allStudents;

/\*\*

\* Die Liste aller Kurse einer Berechnung.

\*/

private List<Course> allCourses;

/\*\*

\* Erzeugt einen neuen Speicher, indem die "editedStudents" mit den

\* "ignoredStudents" gekreuzt und dann in die {@link #allStudents Liste aller

\* Schüler} dieses Speichers gespeichert werden. Dabei wird die mitgegebenen

\* Listen über einen MergeSort allgorithmus sortiert. Dem {@link #informations

\* Informations Speicher} wird dieser Speicher als {@link InformationSave#parent

\* Elternklasse} gesetzt und dieser wird noch einmal

\* {@link InformationSave#update() aktualisiert}.

\*

\* **@param** editedStudents Alle berechneten Schüler -

\* {@link Distributor#allStudents}

\* **@param** ignoredStudents Alle nicht mitberechneten Schüler -

\* {@link Distributor#ignoredStudents}

\* **@param** allCourses Alle Kurse - {@link Distributor#allCourses}

\* **@param** informations Weitere Informationen über die Berechnung -

\* {@link InformationSave#InformationSave(int, int, int[], ArrayList, ArrayList)}

\*/

public Save(List<Student> editedStudents, List<Student> ignoredStudents, List<Course> allCourses) {

this.allCourses = Save.*sortCourse*((allCourses));

this.allStudents = new ArrayList<>(editedStudents);

this.allStudents.addAll(ignoredStudents);

this.allStudents = Save.*sortStudents*((this.allStudents));

(this.informations = new InformationSave(this)).update();

}

/\*\*

\* **@return** Die höchste Priorität der in diesem Speicher gespeicherten

\* Berechnung.

\*/

public int getHighestPriority() {

int highest = 0;

for (Student s : this.allStudents)

if (s.getActiveCourse() != null && !s.getActiveCourse().equals(Distributor.*getInstance*().ignore()))

highest = highest >= s.getPriority() ? highest : s.getPriority();

else if (s.getActiveCourse() == null)

highest = highest >= s.getPriority() ? highest : s.getPriority();

return highest;

}

/\*\*

\* **@return** Die Anzahl der Schüler mit den einzelnen Prioritäten der in diesem

\* Speicher gespeicherten Berechnung.

\*/

public int[] getStudentPriorities() {

int[] studentPriorities = new int[this.getHighestPriorityWhithoutIntegerMax() + 1];

int i1 = 0;

for (Student s : this.getStudentsWithPriority(1))

if (s.getActiveCourse() != null && !s.getActiveCourse().equals(Distributor.*getInstance*().ignoredCourse))

i1++;

studentPriorities[0] = i1;

for (int i = 1; i < studentPriorities.length - 1; i++) {

studentPriorities[i] = this.getStudentsWithPriority(i + 1).size();

}

studentPriorities[studentPriorities.length - 1] = this.getInformation().getUStudents().size();

return studentPriorities;

}

/\*\*

\* Bewertet die hier gespeicherte Berechnung indem die Summe aller Schüler

\* {@link Student#getRate(int) Raten} zurückgegeben wird. Sollte diese aus dem

\* zugelassenden Bereich herausfallen, wird {@link Integer#MAX\_VALUE}

\* zurückgegeben.

\*

\* **@param** highestPriority Die höchste Priorität dieser Berechnung - Über

\* {@link #getHighestPriorityWhithoutIntegerMax()} zu

\* erhalten.

\* **@return** Die Rate dieser Berechnung.

\*/

public int rate(int highestPriority) {

int count = 0;

for (Student s : this.allStudents) {

if (s.getActiveCourse() != null && s.getActiveCourse().equals(Distributor.*getInstance*().ignore()))

continue;

count += s.getRate(highestPriority);

}

if (count > Integer.***MAX\_VALUE*** || count < 0)

return Integer.***MAX\_VALUE***;

return count;

}

/\*\*

\* Gibt den Kurs aus der {@link #allCourses Liste aller Kurse} zurück, der

\* gleich benannt ist. Dazu wird {@link Course#toString()} mit dem mitgegebenen

\* Namen verglichen.

\*

\* **@param** name Der Name des Kurses, der gesucht wird.

\* **@return** Der Kurs mit dem entsprechenden Namen oder {@code null}, wenn der

\* Kurs nicht gefunden werden konnte.

\*/

public Course getCourseByName(String name) {

for (Course c : this.allCourses) {

if (name.equalsIgnoreCase(c.toString()))

return c;

}

return null;

}

/\*\*

\* **@return** Die in diesem Speicher vorhandende höchste Schülerpriorität zurück,

\* wobei nicht zugewiesenden Schüler nicht beachtet werden.

\*/

public int getHighestPriorityWhithoutIntegerMax() {

int highest = 0;

for (Student s : this.allStudents)

if (s.getPriority() != Integer.***MAX\_VALUE***)

highest = highest >= s.getPriority() ? highest : s.getPriority();

return highest;

}

/\*\*

\* Sucht alle Schüler mit der gegebenden {@link Student#priority Priorität}

\* heraus und gibt diese als Liste zurück.

\*

\* **@param** priority Die Priorität nach der gesucht wird.

\* **@return** Eine Liste aller Schüler mit dieser Priorität.

\*/

public ArrayList<Student> getStudentsWithPriority(int priority) {

ArrayList<Student> pStudents = new ArrayList<>();

for (Student s : this.allStudents)

if (s.getPriority() == priority)

pStudents.add(s);

return pStudents;

}

/\*\*

\* Sortiert die mitgegebende Liste der Kurse nach ihren Namen über die

\* {@link Course#compareTo(Course)} Methode des Kurses.

\*

\* **@param** courseToSort Die Liste der Kurse, die Sortiert werden soll.

\* **@return** Die sortierte Liste der Kurse.

\*/

private static List<Course> sortCourse(List<Course> courseToSort) {

ExecutorService pool = Executors.*newFixedThreadPool*(courseToSort.size() / 2 + 10);

Future<ArrayList<Course>> sortedStudents = pool

.submit(new MergeSort<Course>((ArrayList<Course>) courseToSort, pool));

try {

ArrayList<Course> compute = new ArrayList<Course>(sortedStudents.get());

pool.shutdownNow();

return compute;

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

pool.shutdownNow();

return null;

}

/\*\*

\* Sortiert die mitgegebende Liste der Schüler nach ihren Namen über die

\* {@link Student#compareTo(Student)} Methode des Schülers.

\*

\* **@param** studentsToSort Die Liste der Schüler, die sortiert werden soll.

\* **@return** Die sortierte Liste der Schüler.

\*/

public static List<Student> sortStudents(List<Student> studentsToSort) {

ExecutorService pool = Executors.*newFixedThreadPool*(studentsToSort.size() / 2 + 10);

Future<ArrayList<Student>> sortedStudents = pool

.submit(new MergeSort<Student>((ArrayList<Student>) (studentsToSort), pool));

try {

ArrayList<Student> compute = new ArrayList<Student>(sortedStudents.get());

pool.shutdownNow();

return compute;

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

pool.shutdownNow();

return null;

}

/\*\*

\* **@return** Die {@link #allStudents Liste aller Schüler} dieses Speichers zurück.

\*/

public List<Student> getAllStudents() {

return this.allStudents;

}

/\*\*

\* **@return** Die {@link #allCourses Liste aller Kurse} dieses Speichers zurück.

\*/

public List<Course> getAllCourses() {

return this.allCourses;

}

/\*\*

\* **@return** Gibt die {@link #allCourses Liste aller Kurse} als Array zurück.

\*/

public Course[] getAllCoursesAsArray() {

Course[] courses = new Course[this.allCourses.size()];

int i = 0;

for (Course c : this.allCourses)

courses[i++] = c;

return courses;

}

/\*\*

\* **@return** Der mit diesem Speicher verknüpfte {@link #informations

\* Informationsspeicher}.

\*/

public InformationSave getInformation() {

return this.informations;

}

/\*\*

\* Erstellt {@link WriteableContent Tabellen}, indenen die unterschiedlichen

\* Informationen geschrieben werden. Unter dem Element 1 sind die Informationen

\* von der Schülerseite aus, in dem zweiten Listenelement befinden sich die

\* Informationen von der Kursseite aus und an letzter Stelle findet man die

\* weiteren Informationen zu dieser gespeicherten Berechnung.

\*

\* **@return** Liste der Tabellen

\*/

public List<WriteableContent> writeInformation() {

List<WriteableContent> export = new ArrayList<WriteableContent>(3);

// Write students

WriteableContent students = this.writeStudentInformation();

export.add(students);

WriteableContent courses = this.writeCourseInformation();

export.add(courses);

WriteableContent information = this.informations.write();

export.add(information);

return export;

}

/\*\*

\* Schreibt alle Informationen von Schülerseite aus in eine

\* {@link WriteableContent Tabelle}.

\*

\* **@return** Eine Tabelle mit den schülerseitigen Informationen dieses Speichers.

\*/

public WriteableContent writeStudentInformation() {

WriteableContent students = new WriteableContent(References.*language*.getString("coursedistribution.text"));

students.addLine(new Vec2i(0, 0), new String[] { "Name", "Vorname", "Kurs", "Lehrer", "Priorität" });

for (int i = 0; i < this.allStudents.size(); i++) {

String[] line = new String[5];

line[0] = this.allStudents.get(i).getName();

line[1] = this.allStudents.get(i).getPrename();

if (this.allStudents.get(i).getActiveCourse() != null

&& !Util.*isBlank*(this.allStudents.get(i).getActiveCourse().toString())) {

line[2] = this.allStudents.get(i).getActiveCourse().getSubject();

line[3] = this.allStudents.get(i).getActiveCourse().getTeacher();

line[4] = this.allStudents.get(i).getPriority() + "";

} else

line[2] = "@PJK";

students.addLine(new Vec2i(0, i + 1), line);

}

return students;

}

/\*\*

\* Schreibt alle Informationen von Kursseite aus in eine {@link WriteableContent

\* Tabelle}.

\*

\* **@return** Eine Tabelle mit den kursseitigen Informationen dieses Speichers.

\*/

public WriteableContent writeCourseInformation() {

WriteableContent courses = new WriteableContent(References.*language*.getString("studentdistribution.text"));

courses.addLine(new Vec2i(0, 0), new String[] { "Kurs", "Lehrer", "Anzahl Schüler", "Schüler" });

int line = 1;

for (Course c : this.allCourses) {

String[] parameter = new String[c.size() + 3];

References.***LOGGER***.info("Kursname: " + c.toString() + "; Schülerzahl: " + Integer.*toString*(c.size())

+ "; Parameterlänge: " + Integer.*toString*(parameter.length));

parameter[0] = c.getSubject();

parameter[1] = c.getTeacher();

int lineAdd = 3;

int anzahl = 0;

for (int i = 0; i < c.size(); i++, anzahl++) {

References.***LOGGER***.info(i + "");

if (Config.*firstPrename*)

if (Config.*shortNames*)

parameter[i + lineAdd] = c.getStudent(i).getPrename().toCharArray()[0] + ". "

+ c.getStudent(i).getName();

else

parameter[i + lineAdd] = c.getStudent(i).getPrename() + " " + c.getStudent(i).getName();

else if (Config.*shortNames*)

parameter[i + lineAdd] = c.getStudent(i).getName() + ", "

+ c.getStudent(i).getPrename().toCharArray()[0];

else

parameter[i + lineAdd] = c.getStudent(i).getName() + ", " + c.getStudent(i).getPrename();

}

parameter[2] = Integer.*toString*(anzahl);

courses.addLine(new Vec2i(0, line++), parameter);

}

return courses;

}

/\*\*

\* Vergleicht diesen Speicher mit einem anderen. Dazu wird zunächst die

\* {@link InformationSave#getGuete() Guete} der beiden Speicher miteinander

\* verglichen. Wird 0 zurückgegeben, beinhalten die beiden Speicher die selben

\* Informationen.

\*/

*@Override*

public int compareTo(Save s) {

if (s.getInformation().getGuete() > 1)

return 1;

if (this.informations.getGuete() > 1)

return -1;

if (this.informations.getGuete() < 0)

return -1;

if (s.getInformation().getGuete() < 0)

return 1;

if (this.getInformation().getGuete() == s.getInformation().getGuete())

return 0;

if (this.sameCalculation(s))

return 0;

return this.informations.getGuete() - s.informations.getGuete() >= 0 ? 1 : -1;

}

/\*\*

\* Überprüft, ob es sich bei diesem Speicher und dem gegebenen Speicher, um

\* Speicher mit den selben Informationen handelt.

\*

\* **@param** s Der Speicher, der mit diesem verglichen werden soll.

\* **@return** Ob die Speicher Identische Werte aufweisen.

\*/

private boolean sameCalculation(Save s) {

if (this.getHighestPriority() == s.getHighestPriority()) {

int[] thispriorities = this.getStudentPriorities();

int[] spriorities = s.getStudentPriorities();

if (thispriorities.length != spriorities.length)

return false;

for (int i = 0; i < thispriorities.length; i++)

if (thispriorities[i] != spriorities[i])

return false;

return true;

}

return false;

}

/\*\*

\* Verqleicht die gegebene Guete mit der {@link InformationSave#getGuete()

\* Guete} dieses Speichers. Es wird auf gleiche Weise verglichen wie in der

\* Methode {@link #compareTo(Save)} die Guete verglichen wurde.

\*

\* **@implNote** Es ist besser die Methode {@link #compareTo(Save)} anstelle von

\* dieser hier zu verwenden.

\*/

public int compareTo(int guete) {

return this.informations.getGuete() - guete >= 0 ? -1 : 1;

}

/\*\*

\* Fügt einen Kurs zu diesem Speicher hinzu. Existiert dieser Kurs bereits, wird

\* der existierende Kurs gelöscht und der mitgegebende anschließend eingefügt.

\*

\* **@param** c Der Kurs der hinzugefügt werden soll.

\*/

public void addCourse(Course c) {

if (this.allCourses.contains(c))

this.allCourses.remove(c);

this.allCourses.add(c);

}

/\*\*

\* Ermittelt den Schüler mit der mitgegebenen ID

\*

\* **@param** studentID Die ID nach der gesucht werden soll.

\* **@return** Der Schüler mit dieser ID oder {@code null}, wenn kein Schüler mit

\* dieser ID gefunden werden konnte.

\*/

public Student getStudentByID(int studentID) {

for (Student s : this.allStudents)

if (s.idequals(studentID))

return s;

return null;

}

/\*\*

\* Fügt einen Schüler zu diesem Speicher hinzu. Existiert dieser Schüler

\* bereits, wird der vorgang abgebrochen und {@code false} zurückgegeben.

\*

\* **@param** student Der Schüler der hinzugefügt werden soll.

\* **@return** Ob der Schüler erfolgreich hinzugefügt werden konnte.

\*/

public boolean addStudent(Student student) {

if (student == null || this.allStudents.contains(student))

return false;

return this.allStudents.add(student);

}

/\*\*

\* Löscht einen Schüler aus der {@link #allStudents Liste aller Schüler}.

\*

\* **@param** student Der Schüler der gelöscht werden soll.

\*/

public void removeStudent(Student student) {

if (student == null)

return;

if (this.allStudents.contains(student)) {

student.getActiveCourse().removeStudent(student);

this.allStudents.remove(this.allStudents.indexOf(student));

}

}

/\*\*

\* Löscht einen Kurs aus der {@link #allCourses Liste aller Kurse}.

\*

\* **@param** course Der Kurs der gelöscht werden soll.

\*/

public void removeCourse(Course course) {

if (course == null)

return;

if (this.allCourses.contains(course)) {

for (Student s : course.getStudents()) {

s.setActiveCourse(null);

s.refreshPriority();

s.mark();

}

this.allCourses.remove(this.allCourses.indexOf(course));

}

}

} // Ende der Klasse Save

package de.juhu.distributor;

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

import de.juhu.util.Config;

/\*\*

\* Diese Klasse bildet einen Schüler ab und beinhaltet alle diesbezüglich

\* wichtigen Informationen.

\*

\* **@version** 1.1

\* **@category** Distribution

\* **@author** Juhu1705

\* **@implNote** Ist {@link Comparable vergleichbar} mit anderen {@link Student

\* Schülern}.

\*/

public class Student implements Comparable<Student>, Serializable {

// **INFO**: Attribute

/\*\*

\* Speichert alle vom {@link Student Schüler} gewählten {@link Course Kurse}

\* nach ihrer Priorität. An Position 0 steht der Kurs mit der geringsten

\* Priorität.

\*/

private ArrayList<Course> courses = new ArrayList<>();

/\*\*

\* Der Kurs, in dem sich der Schüler momentan befindet. Ist {@code null}, wenn

\* der Schüler in keinem Kurs ist.

\*/

private Course activeCourse;

/\*\*

\* <p>

\* Die Priorität die dem {@link #activeCourse aktiven Kurs} nach der

\* {@link #courses Liste der Kurse} zugeordnet ist.

\* </p>

\* Kann über {@link #refreshPriority()} aktualisiert werden.

\*/

protected int priority;

/\*\*

\* Die ID des Schülers. Sie ist für jeden unterschiedlichen Schüler einzigartig.

\*/

private int id;

/\*\*

\* Der Nachname des Schülers.

\*/

private String name;

/\*\*

\* Der Vorname des Schülers.

\*/

private String prename;

/\*\*

\* Ein Indikator, der anzeigt, ob dieser Schüler einem Kurs zugeordnet ist.

\*/

private boolean mark;

// **INFO**: Konstruktoren

/\*\*

\* Erzeugt einen Schüler mit einer neuen ID, der noch keinen Namen besitzt.

\*/

protected Student() {

this.generateID();

this.unmark();

}

/\*\*

\* Erzeugt einen neuen Schüler, mit einer neuen ID. Der Name und der Vorname

\* werden gespeichert und die mitgegebenen Kurse werden in gleicher Reihenfolge

\* in die {@link #courses Liste der gewünschten Kurse} gespeichert. Der erste

\* Kurs der Liste wird dabei zum {@link #activeCourse aktiven Kurs} des

\* Schülers.

\*

\* **@param** name Der Nachname des Schülers

\* **@param** prename Der Vorname des Schülers

\* **@param** courses Die Kurswünsche des Schülers

\*/

public Student(String name, String prename, Course... courses) {

this();

for (Course courseName : courses) {

this.addCourse(courseName);

}

if (this.courses.size() == 1 && this.courses.get(0).equals(Distributor.*getInstance*().ignore()))

this.activeCourse = this.courses.get(0);

this.setName(name);

this.setPrename(prename);

this.priority = 0;

}

/\*\*

\* Erstellt einen Schüler mit dem mitgegebenen Vor- und Nachnamen, sowie der

\* mitgegebenen ID. Nur zum Kopieren verwendet. Kurse müssen manuell eingefügt,

\* oder beim kopieren synchronisiert werden. Hierzu eignet sich die Methode

\* {@link Distributor#copyData(ArrayList, ArrayList, Course)}

\*

\* **@param** name Der Nachname des Schülers

\* **@param** prename Der Vorname des Schülers

\* **@param** id Die ID des Schülers

\*/

private Student(String name, String prename, int id) {

this.setName(name);

this.setPrename(prename);

this.priority = 0;

this.id = id;

this.unmark();

}

// **INFO**: Methoden

/\*\*

\* Generiert eine ID für den Schüler. Wird immer beim erstellen eines Schülers

\* aufgerufen. Greift auf die Methode {@link Distributor#getStudentID()} zurück

\* um eine ID zu generieren und setzt die ID auf den von dieser Methode

\* zurückgegebenen Wert.

\*/

private void generateID() {

this.id = Distributor.*getStudentID*();

}

/\*\*

\* <p>

\* Setzt den {@link #activeCourse aktive Kurs} auf den nächsten noch freien

\* {@link Course Kurs} aus der {@link #courses Liste der gewünschten Kurse} des

\* Schülers.

\* </p>

\* <ul>

\* <li>Dazu wird zunächst geguckt, ob der Schüler {@link #courses gewünschten

\* Kurse} besitzt. Sollte dies nicht der Fall sein, so wird {@code false}

\* zurückgegeben, der Prioritätswert durch Aufruf der Methode

\* {@link #refreshPriority()} aktualisiert und die Methode beendet.</li>

\*

\* <li>Danach wird geschaut, ob der {@link #activeCourse aktive Kurs} momentan

\* den Wert {@code null} beinhaltet. Sollte dies der Fall sein, so wird der

\* aktive Kurs, auf den ersten Kurs aus der Liste der {@link #courses

\* gewünschten Kurse} gesetzt und der Schüler dem Kurs hinzugefügt.</li>

\*

\* <li>Im folgenden wird in einer {@code while}-Schleife geschaut, ob der aktive

\* Kurs bereits mit seiner maximalen Schüleranzahl besetzt ist. Wenn ja, so wird

\* der nächste Kurs aus der {@link #courses Liste der gewünschten Kurse}

\* ermittelt und bei diesem das gleiche ermittelt. Sollten alle Kurse die der

\* Schüler sich wünscht an dieser Stelle voll sein, so wird die Methode mit dem

\* Rückgabewert {@code false} abgebrochen, sowie der {@link #activeCourse aktive

\* Kurs} auf {@code null} gesetzt.</li>

\*

\* <li>Sollte ein Kurs gefunden werden, der noch frei ist, so wird dieser

\* Schüler der {@link Course#students Liste der Schüler des Kurses} hinzugefügt.

\* Anschließend wird der {@link #priority Prioritätswert} des Schülers durch die

\* Methode {@link #refreshPriority()} aktualisiert. Dann wird die Methode mit

\* der Rückgabe {@code true} beendet.</li>

\* </ul>

\*

\* **@return** Ob ein neuer Kurs zugewiesen werden konnte. Wenn {@code false}

\* zurückgegeben wird, so ist der {@link #activeCourse aktive Kurs} dann

\* {@code null}.

\*/

public boolean next() {

if (this.courses.isEmpty())

return false;

if (this.activeCourse == null) {

this.activeCourse = this.courses.get(0);

this.activeCourse.addStudent(this);

}

while (this.activeCourse.isFull()) {

this.setNextCourse();

if (this.activeCourse == null) {

this.refreshPriority();

return false;

}

}

this.refreshPriority();

return true;

}

/\*\*

\* Setzt den {@link #activeCourse Aktiven Kurs} auf den nächsten Kurs aus der

\* {@link #courses Liste der gewünschten Kurse} des Schülers.

\*

\* **@see** #next() Hier ist die Funktion genau beschrieben. Anstatt der

\* {@code while}-Schleife wird hier lediglich ein {@code if}-Case genutzt,

\* der mit einer {@code else} Bedingung mit der vorgeschalteten Überprüfung

\* verbunden ist. Ansonsten sind die Methoden identisch aufgebaut.

\* **@implNote** Die Methode fügt einen Schüler auch zu einem Bereits vollen Kurs

\* hinzu und lässt ihn dort verweilen.

\* **@return** Ob der Schüler einem Kurs zugeordnet werden konnte.

\*/

public boolean onlyNext() {

if (this.courses.isEmpty())

return false;

if (this.activeCourse == null) {

this.activeCourse = this.courses.get(0);

this.activeCourse.addStudent(this);

} else if (this.activeCourse.isFull()) {

this.setNextCourse();

if (this.activeCourse == null) {

this.refreshPriority();

return false;

}

}

this.refreshPriority();

return true;

}

// **INFO**: Getter und Setter

/\*\*

\* Setzt den {@link #name Nachnamen} des Schülers auf den mitgegebenen String.

\*

\* **@param** name Den Nachnamen, den der Schüler annehmen soll.

\*/

public void setName(String name) {

this.name = name;

}

/\*\*

\* Gibt den {@link #name Nachnamen} des Schülers zurück.

\*

\* **@return** Der Nachname des Schülers.

\*/

public String getName() {

return this.name;

}

/\*\*

\* Setzt den {@link #prename Vornamen} des Schülers auf den mitgegebenen String.

\*

\* **@param** prename Den Vornamen, den der Schüler annehmen soll.

\*/

public void setPrename(String prename) {

this.prename = prename;

}

/\*\*

\* Gibt den {@link #prename Vornamen} des Schülers zurück.

\*

\* **@return** Der Vorname des Schülers.

\*/

public String getPrename() {

return this.prename;

}

/\*\*

\* Fügt einen Kurs ans Ende der {@link #courses Liste der gewünschten Kurse} des

\* Schülers hinzu.

\*

\* Wenn der Kurs den Wert {@code null} darstellt, so wird dieser Ignoriert.

\*

\* **@param** course Der Kurs, der hinzugefügt werden soll.

\*/

public void addCourse(Course course) {

if (course == null)

return;

if (this.courses.contains(course)) {

this.courses.remove(course);

}

this.courses.add(course);

}

/\*\*

\* Fügt einen Kurs an der gewünschten Stelle zu den {@link #courses

\* Wunschkursen} hinzu. Dabei werden die nachfolgenden Kurse und der Kurs an

\* diesem Index um einen Listenplatz nach hinten verschoben.

\*

\* **@param** index Der Listenplatz, an den der Kurs eingefügt werden soll.

\* **@param** course Der Kurs, der eingefügt werden soll.

\* **@throws** IndexOutOfBoundsException Wenn der Index nicht innerhalb der Liste

\* liegt.

\*/

public void addCourse(int index, Course course) throws IndexOutOfBoundsException {

if (course == null)

return;

if (this.courses.contains(course)) {

this.courses.remove(course);

}

this.courses.add(index, course);

}

/\*\*

\* Gibt die Liste aller {@link #courses gewünschten Kurse} des Schülers zurück.

\*

\* **@return** Ein Array der gewünschten Kurse des Schülers.

\*/

public Course[] getCourses() {

return this.courses.toArray(new Course[this.courses.size()]);

}

/\*\*

\* Setzt den {@link #activeCourse aktiven Kurs} auf den nächsten Kurs aus der

\* {@link #courses Liste der gewünschten Kurse}.

\*

\* **@return** Der neu gesetzte {@link #activeCourse Kurs}.

\*/

private Course setNextCourse() {

return this.setCourse(this.getNextCourse());

}

/\*\*

\* Setzt den {@link #activeCourse aktiven Kurs} auf den gewünschten Kurs. Dabei

\* wird der letzte aktive Kurs in einen entfernt.

\*

\* **@param** course Der Kurs der zum aktiven Kurs werden soll.

\* **@return** Der aktive Kurs.

\*/

private Course setCourse(Course course) {

if (this.activeCourse.contains(this))

this.activeCourse.removeStudent(this);

if (course == null)

return this.activeCourse = null;

this.activeCourse = course;

this.activeCourse.addStudent(this);

return course;

}

/\*\*

\* Gibt denjenigen Kurs aus der {@link #courses Liste der gewünschten Kurse des

\* Schülers} zurück, welcher dem {@link #activeCourse aktiven Kurs} folgt.

\*

\* **@return** Der dem aktiven Kurs folgende Kurs.

\* **@see** #getNextCourse(Course, int)

\* getNextCourse({@link #activeCourse}{@link #getNextCourse(Course, int) ,

\* 1)}

\*/

public Course getNextCourse() {

return this.getNextCourse(this.activeCourse, 1);

}

/\*\*

\* Gibt den Kurs zurück der iterator viele Positionen hinter dem

\* {@link #activeCourse aktiven Kurs} in der {@link #courses Liste der

\* gewünschten Kurse} steht.

\*

\* **@param** iterator Anzahl der Positionen die zwischen dem aktiven Kurs und dem

\* zu ermittelnden Kurs liegen sollen.

\* **@return** Den iterator-viele Positionen hinter dem aktiven Kurs liegende Kurs.

\* **@see** #getNextCourse(Course, int)

\* getNextCourse({@link #activeCourse}{@link #getNextCourse(Course, int) ,

\* iterator)}

\*/

public Course getNextCourse(int iterator) {

return this.getNextCourse(this.activeCourse, iterator);

}

/\*\*

\* Ermittelt den Kurs, der iterator viele Positionen hinter dem mitgegebenen

\* Kurs in der {@link #courses Liste der gewünschten Kurse} liegt.

\*

\* **@param** course Der Kurs, von dem gezählt wird.

\* **@param** iterator Die Anzahl an Positionen, die vom mitgegebenen Kurs

\* weitergegangen werden soll.

\* **@return** Der Kurs der entsprechend viele Positionen hinter dem mitgegebenen

\* Kurs liegt. Gibt {@code null} zurück wenn der Kurs kein Element der

\* {@link #courses Liste der gewünschten Kurse} ist. Gibt den ersten

\* Kurs der Liste zurück wenn der mitgegebene Kurs {@code null} ist.

\*/

public Course getNextCourse(Course course, int iterator) {

if (course == null)

return this.courses.get(0);

if (this.courses.contains(course)) {

int i = this.courses.indexOf(course);

if (i + iterator >= this.courses.size())

return null;

else

return this.courses.get(i + iterator);

}

return null;

}

/\*\*

\* Ermittelt den Index an dem der Kurs in der {@link #courses Liste der

\* gewünschten Kurse} steht multipliziert mit 2.

\*

\* **@param** course Der Kurs dessen Wert ermittelt werden soll

\* **@return** Der Index multipliziert mit 2, oder {@link Integer#MAX\_VALUE}, wenn

\* der Kurs nicht in der {@link #courses Liste der gewünschten Kurse}

\* existiert.

\*/

public int getCourseAmount(Course course) {

for (int i = 0; i < this.courses.size(); i++) {

if (this.courses.get(i).equals(course)) {

// References.LOGGER.info(this.toString() + ": " + i);

return i \* 2;

}

}

return Integer.***MAX\_VALUE***;

}

/\*\*

\* **@return** Den {@link #activeCourse aktiven Kurs} des Schülers

\*/

public Course getActiveCourse() {

return activeCourse;

}

/\*\*

\* Setzt den aktiven Kurs des Schülers auf den gewünschten Kurs, wenn dieser in

\* der {@link #courses Liste der Wunschkurse} vorhanden ist. Danach sorgt die

\* Methode dafür, das die {@link #priority Priorität} des Kurses aktualisiert

\* wird.

\*

\* **@param** activeCourse Der Kurs, welcher zum aktiven Kurs gemacht werden soll.

\* Wenn er {@code null} entspricht wird der

\* {@link #activeCourse aktive Kurs} auf {@link null}

\* gesetzt.

\*/

public void setActiveCourse(Course activeCourse) {

if (this.isMarked())

this.unmark();

if (activeCourse == null) {

if (this.activeCourse != null && this.activeCourse.contains(this))

this.activeCourse.removeStudent(this);

this.activeCourse = activeCourse;

this.refreshPriority();

return;

}

if (this.courses.contains(activeCourse)) {

if (this.isMarked())

this.mark = false;

if (this.activeCourse != null && this.activeCourse.contains(this))

this.activeCourse.removeStudent(this);

this.activeCourse = activeCourse;

if (!this.activeCourse.contains(this))

this.activeCourse.addStudent(this);

this.refreshPriority();

}

}

/\*\*

\* Setzt den {@link #priority Prioritätswert} auf den, durch die Methode

\* {@link #calculatePriority()} ermittelten Wert.

\*/

public void refreshPriority() {

this.priority = this.calculatePriority();

}

/\*\*

\* Berechnet die Priorität die der {@link #activeCourse aktive Kurs} besitzt.

\*

\* **@return** Die Priorität die der aktive Kurs besitzt, oder

\* {@link Integer#MAX\_VALUE} wenn der aktive Kurs {@code null} ist, oder

\* nicht in der {@link #courses Liste der Wunschkurse} existiert.

\*/

private int calculatePriority() {

if (this.activeCourse == null)

return Integer.***MAX\_VALUE***;

if (this.courses.contains(this.activeCourse))

return (this.courses.indexOf(this.activeCourse) + 1);

return Integer.***MAX\_VALUE***;

}

/\*\*

\* **@return** Gibt die zuletzt ermittelte {@link #priority Priorität} zurück.

\*/

public int getPriority() {

return this.priority;

}

/\*\*

\* Ermittelt die Rate, die dieser Schüler besitzt: {@link #priority Priorität}

\* hoch {@link Config#powValue}. Sollte die Priorität {@link Integer#MAX\_VALUE}

\* entsprechen, wird die durch

\* {@link Distributor#getHighestPriorityWhithoutIntegerMax()} ermittelte

\* Priorität plus eins als Priorität angenommen.

\*

\* **@return** Die Rate des Schülers

\*/

public int getRate() {

return (int) (this.priority == Integer.***MAX\_VALUE***

? Math.*pow*(Distributor.*getInstance*().getHighestPriorityWhithoutIntegerMax() + 1, Config.*powValue*)

: Math.*pow*(this.priority, Config.*powValue*));

}

/\*\*

\* Ermittelt die Rate, die dieser Schüler besitzt: {@link #priority Priorität}

\* hoch {@link Config#powValue}. Sollte die Priorität {@link Integer#MAX\_VALUE}

\* entsprechen, wird die durch highestPriority plus eins ersetzt.

\*

\* **@param** highestPriority Die höchste Priorität der Berechnung

\* **@return** Die Rate des Schülers

\*/

public int getRate(int highestPriority) {

return (int) (this.priority == Integer.***MAX\_VALUE*** ? Math.*pow*(highestPriority + 1, Config.*powValue* + 3)

: Math.*pow*(this.priority, Config.*powValue*));

}

/\*\*

\* Setzt die {@link #mark Markierung} des Schülers auf {@code true}.

\*/

public void mark() {

this.mark = true;

}

/\*\*

\* Setzt die {@link #mark Markierung} des Schülers auf {@code false}.

\*/

public void unmark() {

this.mark = false;

}

/\*\*

\* **@return** Die {@link #mark Markierung} des Schülers.

\*/

public boolean isMarked() {

return this.mark;

}

*@Override*

public int compareTo(Student s) {

int i = this.getName().compareToIgnoreCase(s.getName());

if (i == 0)

i = this.getPrename().compareToIgnoreCase(s.getPrename());

return i;

}

/\*\*

\* Schaut ob das zu vergleichende Objekt ein Schüler ist, wenn nicht wird der

\* Rückgabewert der {@link Object#equals(Object) super-Methode} zurückgegeben.

\* Zum vergleichen der beiden Schüler wird die {@link #id ID} der Schüler

\* miteinander verglichen.

\*/

*@Override*

public boolean equals(Object obj) {

if (!(obj instanceof Student))

return false;

Student student = (Student) obj;

return student.id == this.id;

}

/\*\*

\* Gibt als representativen String den {@link #prename Vornamen} verbunden mit

\* dem {@link #name Nachnamen} zurück.

\*/

*@Override*

public String toString() {

return this.prename + " " + this.name;

}

*@Override*

protected Object clone() throws CloneNotSupportedException {

Student s = new Student(this.name, this.prename, this.id);

return s;

}

/\*\*

\* Der Index des Kurses in der {@link #courses Liste der Wunschkurse}

\*

\* **@see** {@link ArrayList#indexOf(Object)}

\*/

public int getPosition(Course c) {

return this.courses.indexOf(c);

}

/\*\*

\* Überprüft, ob die ID mit der ID dieses Schülers übereinstimmt.

\*

\* **@param** studentID Die ID, deren Übereinstimmung geprüft wird.

\* **@return** Ob die ID mit der {@link #id ID} dieses Schülers übereinstimmt.

\*/

public boolean idequals(int studentID) {

return this.id == studentID;

}

/\*\*

\* **@return** Die ID dieses Schülers

\*/

public int getID() {

return this.id;

}

/\*\*

\* Fügt die mitgegebene Kurse in gleicher Reihenfolge ans Ende der

\* {@link #courses Liste der Wunschkurse} des Schülers an.

\*

\* **@param** c Die Kurse sie eingefügt werden sollen.

\*/

public void setCourses(Course... c) {

this.courses.clear();

for (Course course : c)

this.addCourse(course);

}

/\*\*

\* **@return** Die {@link #courses Wunschkurse} dieses Schülers.

\*/

public List<Course> getCoursesAsList() {

return this.courses;

}

/\*\*

\* Aktualisiert die Marke des Schülers in Bezug auf dessen {@link #priority

\* Priorität}. Ist die {@link #priority Priorität} des Schülers kleiner 0, oder

\* größer als die highestPriority wird der Schüler {@link #mark() markiert}, ist

\* der Schüler {@link #mark markiert}, obwohl die vornestehenden Bedingungen

\* nicht erfüllt sind, wird die {@link #unmark() markierung zurückgezogen}.

\*

\* **@param** highestPriority Die höchste Priorität, die in dem zu vergleichenden

\* Prozess vorliegt.

\*/

public void checkMarkt(int highestPriority) {

if (this.getPriority() > highestPriority || this.getPriority() < 0)

this.mark();

else if (this.mark)

this.unmark();

}

} // Ende der Klasse Student

## Graphische Oberfläche

package de.juhu.guiFX.lists;

import de.juhu.distributor.Course;

import de.juhu.distributor.Distributor;

import de.juhu.util.References;

import javafx.collections.FXCollections;

import javafx.scene.control.TableView;

/\*\*

\* Behandelt die Tabelle zur Ansicht der importierten Kursliste.

\*

\* @author Juhu1705

\* @category GUI

\*/

public class CourseView {

TableView inputTable;

public CourseView(TableView<Course> inputTable) {

this.inputTable = inputTable;

}

public void fill() {

References.LOGGER.config("Loading Input Data to the Preview!");

this.inputTable.getItems().clear();

this.inputTable.setItems(FXCollections.observableArrayList(Distributor.getInstance().getCourses()));

this.inputTable.sort();

}

} // Ende der Klasse CourseView

package de.juhu.guiFX.lists;

import java.util.ArrayList;

import de.juhu.distributor.Distributor;

import de.juhu.distributor.Student;

import de.juhu.util.References;

import javafx.collections.FXCollections;

import javafx.scene.control.TableView;

/\*\*

\* Behandelt die Tabelle zur Ansicht der importierten Schülerliste.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class InputView {

TableView inputTable;

public InputView(TableView<Student> inputTable) {

this.inputTable = inputTable;

}

public void fill() {

ArrayList calcStud = Distributor.*getInstance*().getCalcStudents();

ArrayList ignStud = Distributor.*getInstance*().getIgnoreStudents();

ArrayList allStudents = new ArrayList<>(calcStud);

allStudents.addAll(ignStud);

References.***LOGGER***.config("Loading Input Data to the Preview!");

this.inputTable.getItems().clear();

this.inputTable.setItems(FXCollections.*observableArrayList*(allStudents));

this.inputTable.sort();

}

} // Ende der Klasse InputView

package de.juhu.guiFX.lists;

import java.util.ArrayList;

import de.juhu.distributor.Course;

import de.juhu.guiFX.GUIManager;

import de.juhu.util.References;

import de.juhu.util.Util;

import javafx.beans.property.SimpleStringProperty;

import javafx.collections.FXCollections;

import javafx.scene.control.TableColumn;

import javafx.scene.control.TableView;

/\*\*

\* Behandelt die Tabelle zur Ansicht der verteilten Kurse.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class OutputCourseView implements Runnable {

TableView tv;

ArrayList<TableColumn<Course, String>> students = new ArrayList<>();

public OutputCourseView(TableView<Course> inputTable) {

this.tv = inputTable;

}

public void fill() {

References.***LOGGER***.config("Loading Output Data to the Preview!");

this.tv.setItems(FXCollections.*observableArrayList*(GUIManager.*actual*.getAllCourses()));

this.tv.sort();

}

private void loadMaxStudents() {

for (TableColumn<Course, String> t : students) {

tv.getColumns().remove(t);

}

students.clear();

int maxCount = Util.*maxStudentCount*(GUIManager.*actual*.getAllCourses());

for (int i = 0; i < maxCount; i++) {

TableColumn<Course, String> prename = new TableColumn<>(References.*language*.getString("prename.text"));

TableColumn<Course, String> name = new TableColumn<>(References.*language*.getString("name.text"));

TableColumn<Course, String> student = new TableColumn<>(

References.*language*.getString("student.text") + " " + Integer.*toString*(i + 1));

final int position = i;

student.getColumns().addAll(prename, name);

prename.setCellValueFactory(s -> {

if (s.getValue().getStudents().size() > position)

return new SimpleStringProperty(s.getValue().getStudent(position).getPrename());

return new SimpleStringProperty("-");

});

name.setCellValueFactory(s -> {

if (s.getValue().getStudents().size() > position)

return new SimpleStringProperty(s.getValue().getStudent(position).getName());

return new SimpleStringProperty("-");

});

this.students.add(student);

this.tv.getColumns().addAll(student);

}

}

*@Override*

public void run() {

this.loadMaxStudents();

fill();

GUIManager.*getInstance*().teachers.setDisable(false);

}

} // Ende der Klasse OutputCourseView

package de.juhu.guiFX.lists;

import java.util.ArrayList;

import java.util.HashMap;

import de.juhu.distributor.Student;

import de.juhu.guiFX.GUIManager;

import de.juhu.util.Config;

import de.juhu.util.References;

import javafx.beans.property.SimpleStringProperty;

import javafx.collections.FXCollections;

import javafx.scene.control.TableColumn;

/\*\*

\*

\* Behandelt alle unter Statistiken angezeigten Tabellen.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*

\*/

public class OutputInformationView implements Runnable {

public ArrayList<TableColumn<Student, String>> scourses = new ArrayList<>();

public void fill() {

GUIManager.*getInstance*().bStudents

.setItems(FXCollections.*observableArrayList*(GUIManager.*actual*.getInformation().getBStudents()));

GUIManager.*getInstance*().bStudents.sort();

HashMap<String, Double> ratess = new HashMap<>();

ratess.put(References.*language*.getString("highestpriority.text"),

Double.*valueOf*(GUIManager.*actual*.getInformation().getHighestPriority()));

ratess.put(References.*language*.getString("calculationrate.text"),

Double.*valueOf*(GUIManager.*actual*.getInformation().getRate()));

ratess.put(References.*language*.getString("calculationgoodness.text"),

Double.*valueOf*(GUIManager.*actual*.getInformation().getGuete()));

ratess.put(References.*language*.getString("studentcount.text"),

Double.*valueOf*(GUIManager.*actual*.getAllStudents().size()));

ratess.put(References.*language*.getString("calculatedstudentcount.text"),

Double.*valueOf*(GUIManager.*actual*.getInformation().getStudentCount()));

ratess.put(References.*language*.getString("coursecount.text"),

Double.*valueOf*(GUIManager.*actual*.getAllCourses().size()));

GUIManager.*getInstance*().rates.getItems().clear();

GUIManager.*getInstance*().rates.setItems(FXCollections.*observableArrayList*(ratess.entrySet()));

GUIManager.*getInstance*().rates.sort();

int[] priorities = GUIManager.*actual*.getInformation().getStudentPriorities();

HashMap<Integer, Integer> p = new HashMap<>();

for (int i = 0; i < priorities.length - 1; i++)

p.put(Integer.*valueOf*(i + 1), Integer.*valueOf*(priorities[i]));

p.put(Integer.*valueOf*(-1), Integer.*valueOf*(priorities[priorities.length - 1]));

GUIManager.*getInstance*().priorities.setItems(FXCollections.*observableArrayList*(p.entrySet()));

GUIManager.*getInstance*().priorities.sort();

if (!GUIManager.*actual*.getInformation().getUStudents().isEmpty()) {

if (!this.scourses.isEmpty()) {

this.scourses.forEach(c -> GUIManager.*getInstance*().unallocatedStudents.getColumns().remove(c));

this.scourses.clear();

}

for (int i = 1; i <= Config.*maxChooses*; i++) {

TableColumn<Student, String> k = new TableColumn<>("Course " + i), s = new TableColumn<>("Subject"),

t = new TableColumn<>("Teacher");

k.getColumns().addAll(s, t);

this.scourses.add(k);

this.scourses.add(s);

this.scourses.add(t);

final int number = i;

s.setCellValueFactory(e -> {

if (e.getValue().getCourses().length > number - 1)

return new SimpleStringProperty(e.getValue().getCourses()[number - 1].getSubject() == null ? "-"

: e.getValue().getCourses()[number - 1].getSubject());

else

return new SimpleStringProperty("-");

});

t.setCellValueFactory(e -> {

if (e.getValue().getCourses().length > number - 1)

return new SimpleStringProperty(e.getValue().getCourses()[number - 1].getTeacher() == null ? "-"

: e.getValue().getCourses()[number - 1].getTeacher());

else

return new SimpleStringProperty("-");

});

GUIManager.*getInstance*().unallocatedStudents.getColumns().add(k);

}

GUIManager.*getInstance*().unallocatedStudents.getItems().clear();

// GUIManager.getInstance().unallocatedStudents.setItems(FXCollections.observableArrayList(new

// ArrayList<>()));

References.***LOGGER***.fine(GUIManager.*actual*.getInformation().getUStudents().toString());

GUIManager.*getInstance*().unallocatedStudents

.setItems(FXCollections.*observableArrayList*(GUIManager.*actual*.getInformation().getUStudents()));

GUIManager.*getInstance*().unallocatedStudents.sort();

} else

GUIManager.*getInstance*().unallocatedStudents.getItems().clear();

}

*@Override*

public void run() {

References.***LOGGER***.info("Update Statistics!");

fill();

GUIManager.*getInstance*().statistics.setDisable(false);

GUIManager.*getInstance*().p0.setVisible(false);

GUIManager.*getInstance*().r1.setDisable(false);

GUIManager.*getInstance*().r2.setDisable(false);

GUIManager.*getInstance*().r3.setDisable(false);

// GUIManager.getInstance().b1.setDisable(false);

GUIManager.*getInstance*().b2.setDisable(false);

GUIManager.*getInstance*().b3.setDisable(false);

// GUIManager.getInstance().b4.setDisable(false);

GUIManager.*getInstance*().b5.setDisable(false);

GUIManager.*getInstance*().b6.setDisable(false);

}

} // Ende der Klasse OutputInformationView

package de.juhu.guiFX.lists;

import de.juhu.distributor.Student;

import de.juhu.guiFX.GUIManager;

import de.juhu.util.References;

import javafx.collections.FXCollections;

import javafx.scene.control.TableView;

/\*\*

\* Behandelt die Tabelle zur Ansicht der verteilten Schüler.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class OutputStudentsView implements Runnable {

TableView tv;

public OutputStudentsView(TableView<Student> inputTable) {

this.tv = inputTable;

}

public void fill() {

References.***LOGGER***.config("Loading Output Data to the Preview!");

this.tv.setItems(FXCollections.*observableArrayList*(GUIManager.*actual*.getAllStudents()));

this.tv.sort();

}

*@Override*

public void run() {

fill();

GUIManager.*getInstance*().students.setDisable(false);

}

} // Ende der Klasse OutputStudentsView

package de.juhu.guiFX.lists;

import java.util.ArrayList;

import de.juhu.distributor.Course;

import de.juhu.distributor.Student;

import de.juhu.util.References;

import de.juhu.util.Util;

import javafx.beans.property.SimpleStringProperty;

import javafx.collections.FXCollections;

import javafx.scene.control.TableColumn;

import javafx.scene.control.TableView;

/\*\*

\* Behandelt die Tabelle zum bearbeiten / hinzufügen von Schülern zu einer

\* Verteilung.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class SwitchCourseView implements Runnable {

TableView tv;

ArrayList<TableColumn<Course, String>> students = new ArrayList<>();

Student student;

public SwitchCourseView(TableView<Course> inputTable, Student student) {

this.tv = inputTable;

this.student = student;

}

public void fill() {

References.***LOGGER***.config("Loading Courses Data!");

this.tv.setItems(FXCollections.*observableArrayList*(this.student.getCoursesAsList()));

this.tv.sort();

}

private void loadMaxStudents() {

for (TableColumn<Course, String> t : students) {

tv.getColumns().remove(t);

}

students.clear();

int maxCount = Util.*maxStudentCount*(this.student.getCoursesAsList());

for (int i = 0; i < maxCount; i++) {

TableColumn<Course, String> prename = new TableColumn<>(References.*language*.getString("prename.text"));

TableColumn<Course, String> name = new TableColumn<>(References.*language*.getString("name.text"));

TableColumn<Course, String> student = new TableColumn<>(

References.*language*.getString("student.text") + " " + Integer.*toString*(i + 1));

final int position = i;

student.getColumns().addAll(prename, name);

prename.setCellValueFactory(s -> {

if (s.getValue().getStudents().size() > position)

return new SimpleStringProperty(s.getValue().getStudent(position).getPrename());

return new SimpleStringProperty("-");

});

name.setCellValueFactory(s -> {

if (s.getValue().getStudents().size() > position)

return new SimpleStringProperty(s.getValue().getStudent(position).getName());

return new SimpleStringProperty("-");

});

this.students.add(student);

this.tv.getColumns().addAll(student);

}

}

*@Override*

public void run() {

this.loadMaxStudents();

fill();

}

} // Ende der Klasse SwitchCourseView

package de.juhu.guiFX;

import java.awt.Desktop;

import java.io.IOException;

import java.net.URI;

import java.net.URISyntaxException;

import java.net.URLEncoder;

import java.nio.charset.StandardCharsets;

import java.util.logging.Level;

import de.juhu.util.References;

import javafx.event.ActionEvent;

import javafx.fxml.FXML;

import javafx.scene.control.Hyperlink;

import javafx.scene.control.TextField;

import javafx.scene.input.MouseEvent;

/\*\*

\* Behandelt alle Aktionen des About Fensters.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class AboutManager {

*@FXML*

public TextField weg;

public void openLink(ActionEvent event) {

if (event.getSource() instanceof Hyperlink) {

Hyperlink link = (Hyperlink) event.getSource();

try {

Desktop.*getDesktop*().browse(new URI(link.getText()));

} catch (IOException | URISyntaxException e) {

References.***LOGGER***.log(Level.***SEVERE***, "Can not browse link!", e);

}

}

}

public void startMail(ActionEvent event) {

if (event.getSource() instanceof Hyperlink) {

Hyperlink link = (Hyperlink) event.getSource();

try {

Desktop.*getDesktop*().mail(new URI("mailto:" + link.getText() + "?" + "Need\_Help:CaRP-Assigner"));

} catch (IOException | URISyntaxException e) {

References.***LOGGER***.log(Level.***SEVERE***, "Can not open new mail!", e);

}

}

}

public void openLSPage(MouseEvent event) {

try {

if (Desktop.*isDesktopSupported*() && Desktop.*getDesktop*().isSupported(Desktop.*Action*.***BROWSE***)) {

Desktop.*getDesktop*().browse(new URI("https://www.luisenschule-mh.de/"));

}

} catch (IOException | URISyntaxException e) {

References.***LOGGER***.info("https://www.luisenschule-mh.de/");

}

}

public void onHelpSearch(ActionEvent event) {

try {

if (Desktop.*isDesktopSupported*() && Desktop.*getDesktop*().isSupported(Desktop.*Action*.***BROWSE***)) {

Desktop.*getDesktop*().browse(new URI("https://github.com/juhu1705/CaRP/issues?utf8=%E2%9C%93&q="

+ URLEncoder.*encode*(weg.getText(), StandardCharsets.***UTF\_8***.toString())));

}

} catch (IOException | URISyntaxException e) {

References.***LOGGER***.info("https://github.com/juhu1705/CaRP/issues?utf8=%E2%9C%93&q=" + weg.getText());

}

}

} // Ende der Klasse AboutManager

package de.juhu.guiFX;

import java.net.URL;

import java.util.ResourceBundle;

import de.juhu.distributor.Course;

import de.juhu.distributor.Distributor;

import de.juhu.util.Config;

import de.juhu.util.References;

import javafx.event.ActionEvent;

import javafx.fxml.FXML;

import javafx.fxml.Initializable;

import javafx.scene.control.Button;

import javafx.scene.control.Spinner;

import javafx.scene.control.SpinnerValueFactory;

import javafx.scene.control.TextField;

import javafx.stage.Stage;

/\*\*

\* Behandelt alle Aktionen des Fensters zum hinzufügen und bearbeiten von Kursen

\* zur Liste.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class AddCourseManager implements Initializable {

public static String *s*, *t*;

public static int *mS* = -2;

*@FXML*

private TextField subject;

*@FXML*

private TextField teacher;

*@FXML*

private Spinner<Integer> maxStudents;

public void onAdd(ActionEvent event) {

boolean missingInformation = false;

if (this.subject.getText().isEmpty()) {

this.subject.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (this.teacher.getText().isEmpty()) {

this.teacher.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (missingInformation)

return;

if (*s* != null && *t* != null) {

References.***LOGGER***.info("Try to configure Course");

Course c = Distributor.*getInstance*().getCourseByName(*s* + "|" + *t*);

c.setTeacher(teacher.getText());

c.setSubject(subject.getText());

c.setStudentMax(this.maxStudents.getValue().intValue());

Distributor.*getInstance*().addCourse(c);

} else {

References.***LOGGER***.info("Try to add new Course");

Course c = new Course(this.subject.getText(), this.teacher.getText(),

this.maxStudents.getValue().intValue());

Distributor.*getInstance*().addCourse(c);

}

GUIManager.*getInstance*().inputView.fill();

GUIManager.*getInstance*().cView.fill();

References.***LOGGER***.info("Course was added");

*s* = null;

*t* = null;

*mS* = -2;

((Stage) ((Button) event.getSource()).getScene().getWindow()).close();

}

*@Override*

public void initialize(URL location, ResourceBundle resources) {

maxStudents.setValueFactory(new SpinnerValueFactory.IntegerSpinnerValueFactory(-1, Integer.***MAX\_VALUE***,

*mS* == -2 ? Config.*normalStudentLimit* : *mS*));

if (*s* != null)

subject.setText(*s*);

if (*t* != null)

teacher.setText(*t*);}} // Ende der Klasse AddCourseManager

package de.juhu.guiFX;

import java.net.URL;

import java.util.ResourceBundle;

import de.juhu.distributor.Course;

import de.juhu.util.Config;

import de.juhu.util.References;

import javafx.application.Platform;

import javafx.event.ActionEvent;

import javafx.fxml.FXML;

import javafx.fxml.Initializable;

import javafx.scene.control.Button;

import javafx.scene.control.Spinner;

import javafx.scene.control.SpinnerValueFactory;

import javafx.scene.control.TextField;

import javafx.stage.Stage;

/\*\*

\* Behandelt alle Aktionen des Fensters zum hinzufügen von Kursen zur

\* Verteilung.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class AddCourseToSaveManager implements Initializable {

public static String *s*, *t*;

public static int *mS* = -2;

*@FXML*

private TextField subject;

*@FXML*

private TextField teacher;

*@FXML*

private Spinner<Integer> maxStudents;

public void onAdd(ActionEvent event) {

boolean missingInformation = false;

if (this.subject.getText().isEmpty()) {

this.subject.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (this.teacher.getText().isEmpty()) {

this.teacher.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (missingInformation)

return;

if (*s* != null && *t* != null) {

References.***LOGGER***.info("Try to configure Course");

Course c = GUIManager.*actual*.getCourseByName(*s* + "|" + *t*);

c.setTeacher(teacher.getText());

c.setSubject(subject.getText());

c.setStudentMax(this.maxStudents.getValue().intValue());

GUIManager.*actual*.addCourse(c);

} else {

References.***LOGGER***.info("Try to add new Course");

Course c = new Course(this.subject.getText(), this.teacher.getText(),

this.maxStudents.getValue().intValue());

GUIManager.*actual*.addCourse(c);

}

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

References.***LOGGER***.info("Course was added");

*s* = null;

*t* = null;

*mS* = -2;

((Stage) ((Button) event.getSource()).getScene().getWindow()).close();

}

*@Override*

public void initialize(URL location, ResourceBundle resources) {

maxStudents.setValueFactory(new SpinnerValueFactory.IntegerSpinnerValueFactory(-1, Integer.***MAX\_VALUE***,

*mS* == -2 ? Config.*normalStudentLimit* : *mS*));

if (*s* != null)

subject.setText(*s*);

if (*t* != null)

teacher.setText(*t*);}} // Ende der Klasse AddCourseToSaveManager

package de.juhu.guiFX;

import java.net.URL;

import java.util.ArrayList;

import java.util.ResourceBundle;

import de.juhu.distributor.Course;

import de.juhu.distributor.Distributor;

import de.juhu.distributor.Student;

import de.juhu.util.Config;

import de.juhu.util.References;

import de.juhu.util.Util;

import javafx.event.ActionEvent;

import javafx.fxml.FXML;

import javafx.fxml.Initializable;

import javafx.geometry.Insets;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.control.TextField;

import javafx.scene.input.KeyEvent;

import javafx.scene.layout.HBox;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;

/\*\*

\* Behandelt alle Aktionen des Fensters zum hinzufügen und bearbeiten von

\* Schülern zur Liste.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class AddStudentManager implements Initializable {

public ArrayList<TextField> cources = new ArrayList<>((Config.*maxChooses*) \* 2);

public static int *studentID* = -1;

*@FXML*

public TextField prename, name, c1f, c1t;

*@FXML*

public VBox vBox;

public TextField last;

public int i = 1;

public void onAdd(ActionEvent event) {

boolean missingInformation = false;

if (prename.getText().isEmpty()) {

prename.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (name.getText().isEmpty()) {

name.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (c1f.getText().isEmpty()) {

c1f.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (c1t.getText().isEmpty() && !Util.*isIgnoreCourse*(c1f.getText())) {

c1t.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (missingInformation)

return;

if (*studentID* == -1) {

Course[] c = new Course[cources.size() / 2 + 1];

c[0] = Distributor.*getInstance*().getOrCreateCourseByName(c1f.getText() + "|" + c1t.getText());

for (int i = 0, n = 1; i + 1 < cources.size(); i += 2, n++) {

if (Util.*isBlank*(cources.get(i).getText() + cources.get(i + 1).getText()))

continue;

c[n] = Distributor.*getInstance*()

.getOrCreateCourseByName(cources.get(i).getText() + "|" + cources.get(i + 1).getText());

}

Student s = new Student(name.getText(), prename.getText(), c);

if (c1f.getText().contains(Config.*ignoreStudent*))

s = new Student(name.getText(), prename.getText(), Distributor.*getInstance*().ignore());

Distributor.*getInstance*().addStudent(s);

} else {

Student s = Distributor.*getInstance*().getStudentByID(*studentID*);

s.setName(name.getText());

s.setPrename(prename.getText());

Course[] c = new Course[cources.size() / 2 + 1];

c[0] = Distributor.*getInstance*().getOrCreateCourseByName(c1f.getText() + "|" + c1t.getText());

for (int i = 0, n = 1; i + 1 < cources.size(); i += 2, n++) {

if (Util.*isBlank*(cources.get(i).getText() + cources.get(i + 1).getText()))

continue;

c[n] = Distributor.*getInstance*()

.getOrCreateCourseByName(cources.get(i).getText() + "|" + cources.get(i + 1).getText());

}

s.setCourses(c);

Distributor.*getInstance*().addStudent(s);

}

GUIManager.*getInstance*().inputView.fill();

GUIManager.*getInstance*().cView.fill();

((Stage) ((Button) event.getSource()).getScene().getWindow()).close();

*studentID* = -1;

}

public void filled(KeyEvent event) {

if (!last.getText().equals("") && i < Config.*maxChooses*) {

i++;

HBox hBox = new HBox(10);

hBox.setPadding(new Insets(10));

Label label = new Label(References.*language*.getString("course.text") + " " + i + ":"),

subject = new Label(References.*language*.getString("subject.text") + ":"),

teacher = new Label(References.*language*.getString("teacher.text") + ":");

TextField st = new TextField(), tt = new TextField();

this.cources.add(st);

this.cources.add(tt);

tt.setOnKeyReleased(k -> {

filled(k);

});

last = tt;

hBox.getChildren().addAll(subject, st, teacher, tt);

vBox.getChildren().addAll(label, hBox);

}

}

*@Override*

public void initialize(URL location, ResourceBundle resources) {

References.***LOGGER***.info("Initialize Add Student Manager");

this.last = c1t;

Student s = Distributor.*getInstance*().getStudentByID(*studentID*);

int i = 0;

if (s != null) {

prename.setText(s.getPrename());

name.setText(s.getName());

boolean first = true;

for (Course c : s.getCourses()) {

if (first) {

first = false;

c1f.setText(c.getSubject());

c1t.setText(c.getTeacher());

filled(null);

} else {

if (this.cources.size() < i + 1)

break;

if (this.cources.get(i) == null || this.cources.get(i + 1) == null)

break;

this.cources.get(i++).setText(c.getSubject());

this.cources.get(i++).setText(c.getTeacher());

filled(null);

}

}

}

}

} // Ende der Klasse AddStudentManager

package de.juhu.guiFX;

import java.net.URL;

import java.util.ResourceBundle;

import de.juhu.distributor.Course;

import de.juhu.distributor.Student;

import de.juhu.guiFX.lists.SwitchCourseView;

import de.juhu.util.References;

import de.juhu.util.Util;

import javafx.application.Platform;

import javafx.beans.property.SimpleStringProperty;

import javafx.event.ActionEvent;

import javafx.fxml.FXML;

import javafx.fxml.Initializable;

import javafx.scene.control.Button;

import javafx.scene.control.TableColumn;

import javafx.scene.control.TableView;

import javafx.scene.control.TextField;

import javafx.stage.Stage;

/\*\*

\* Behandelt alle Aktionen des Fensters zum hinzufügen und bearbeiten von

\* Schülern zur Verteilung.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class AddStudentToSaveManager implements Initializable {

public static Student *student*;

*@FXML*

private TableView<Course> courses;

*@FXML*

private TextField prename;

*@FXML*

private TextField name;

*@FXML*

private TableColumn<Course, String> teacher, subject;

private SwitchCourseView scw;

public void onSetActive(ActionEvent event) {

Course c = this.courses.getSelectionModel().getSelectedItem();

if (c == null || (c.getSubject() == null || c.getTeacher() == null)

|| (c.getSubject().isEmpty() && c.getTeacher().isEmpty()))

return;

*student*.setActiveCourse(c);

this.scw.run();

}

*@FXML*

public void onFinished(ActionEvent event) {

boolean missingInformation = false;

if (this.prename.getText().isEmpty()) {

this.prename.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (this.name.getText().isEmpty()) {

this.name.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (missingInformation)

return;

*student*.setName(this.name.getText());

*student*.setPrename(this.prename.getText());

GUIManager.*actual*.addStudent(*student*);

GUIManager.*actual*.getInformation().update();

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

((Stage) ((Button) event.getSource()).getScene().getWindow()).close();

}

*@Override*

public void initialize(URL location, ResourceBundle resources) {

References.***LOGGER***.info("Initializing Add Student Window");

AddStudentToSaveManager.*student* = new Student("", "", GUIManager.*actual*.getAllCoursesAsArray()) {

*@Override*

public void refreshPriority() {

priority = 1;

}

};

References.***LOGGER***.info("Student: " + *student*.toString());

this.teacher.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getTeacher());

});

this.subject.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getSubject());

});

if (!Util.*isBlank*(*student*.getName()))

this.name.setText(*student*.getName());

if (!Util.*isBlank*(*student*.getPrename()))

this.prename.setText(*student*.getPrename());

this.scw = new SwitchCourseView(courses, *student*);

this.scw.run();

}

} // Ende der Klasse AddStudentToSaveManager

package de.juhu.guiFX;

import java.net.URL;

import java.util.ResourceBundle;

import javafx.event.ActionEvent;

import javafx.fxml.FXML;

import javafx.fxml.Initializable;

import javafx.scene.control.Button;

import javafx.scene.control.TextArea;

import javafx.stage.Stage;

/\*\*

\* Behandelt alle Aktionen des Fehler Fensters.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class ErrorGuiController implements Initializable {

public static String *headline*, *information*;

*@FXML*

public TextArea tinformation, theadline;

public void onOK(ActionEvent event) {

((Stage) ((Button) event.getSource()).getScene().getWindow()).close();

}

*@Override*

public void initialize(URL location, ResourceBundle resources) {

this.theadline.setText(*headline*);

this.tinformation.setText(*information*);

}

} // Ende der Klasse ErrorGuiController

package de.juhu.guiFX;

import javafx.beans.property.DoubleProperty;

import javafx.beans.property.SimpleDoubleProperty;

/\*\*

\* Behandelt alle Aktionen der Prozessleiste.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class FullProgress {

private static FullProgress *instance*;

public static FullProgress getInstance() {

return *instance* == null ? *instance* = new FullProgress() : *instance*;

}

private DoubleProperty progress;

public final double getProgress() {

return progress == null ? 0 : progress.get();

}

public final void setProgress(double progress) {

this.progress.set(progress);

}

public final DoubleProperty progressProperty() {

return this.progress == null ? this.progress = new SimpleDoubleProperty(0) : this.progress;}} // Ende der Klasse FullProgress

package de.juhu.guiFX;

import java.net.URL;

import java.util.ResourceBundle;

import de.juhu.util.Config;

import de.juhu.util.References;

import javafx.event.ActionEvent;

import javafx.fxml.FXML;

import javafx.fxml.Initializable;

import javafx.scene.control.Button;

import javafx.scene.control.CheckBox;

import javafx.scene.control.TextField;

import javafx.stage.Stage;

/\*\*

\* Behandelt alle Aktionen des Fensters, dass aufploppt, wenn ein doppelter

\* Schüler beim Import auftritt.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class GUIDoubleStudentManager implements Initializable {

public static String *sName* = "", *sPrename* = "";

public static boolean *finished* = false;

*@FXML*

public TextField name, prename;

*@FXML*

public CheckBox shouldMemorice;

public void skip(ActionEvent event) {

Config.*rememberDecision* = shouldMemorice.isSelected();

Config.*allowDoubleStudents* = false;

GUIDoubleStudentManager.*finished* = true;

((Stage) ((Button) event.getSource()).getScene().getWindow()).close();

}

public void add(ActionEvent event) {

Config.*rememberDecision* = shouldMemorice.isSelected();

References.***LOGGER***.info(Config.*rememberDecision* + "");

Config.*allowDoubleStudents* = true;

GUIDoubleStudentManager.*finished* = true;

((Stage) ((Button) event.getSource()).getScene().getWindow()).close();

}

*@Override*

public void initialize(URL location, ResourceBundle resources) {

if (!*sName*.isEmpty())

name.setText(*sName*);

if (!*sPrename*.isEmpty())

prename.setText(*sPrename*);}} // Ende der Klasse GUIDoubleStudentManager

package de.juhu.guiFX;

import static de.juhu.util.References.***LOGGER***;

import static de.juhu.util.References.***PROJECT\_NAME***;

import static de.juhu.util.References.***VERSION***;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.Reader;

import java.nio.file.FileSystems;

import java.nio.file.Files;

import java.nio.file.LinkOption;

import java.util.PropertyResourceBundle;

import de.juhu.util.References;

import javafx.application.Application;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.scene.image.Image;

import javafx.stage.Stage;

import javafx.stage.StageStyle;

/\*\*

\* Diese Klasse stellt die Hauptklasse des CaRP-Assigners da. Von hier werden

\* die Startprozesse eingeleitet und die Sprache geladen.

\*

\* **@author** Juhu1705

\* **@category** GUI

\* **@version** 1.0

\*/

public class GUILoader extends Application {

private static Stage *primaryStage*;

public static Stage *secondaryStage*;

public static Scene *scene*;

private static File *toLoad*;

*@Override*

public void start(Stage primaryStage) throws Exception {

// LOGGER.info(getClass().getResource("/de/juhu/guiFX/GUI.fxml") + "");

// LOGGER.info(new File("./de/juhu/guiFX/GUI.fxml").toURI() + "");

// LOGGER.info(new File("./de/juhu/guiFX/GUI.fxml").toURI().toURL() + "");

GUILoader.*primaryStage* = primaryStage;

try {

if (!Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"),

*LinkOption*.***NOFOLLOW\_LINKS***)) {

if (!Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER***), *LinkOption*.***NOFOLLOW\_LINKS***))

new File(References.***HOME\_FOLDER***).mkdir();

Files.*copy*(getClass().getResourceAsStream("/assets/language/de.properties"),

FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"));

}

} catch (Exception e) {

FileReader reader;

References.*language* = new PropertyResourceBundle(reader = new FileReader("/assets/language/de.properties"));

reader.close();

}

this.loadLanguage();

if (References.*language*.containsKey("version")) {

FileReader reader = null;

InputStreamReader reader2 = null;

if (!References.*language*.getString("version").equalsIgnoreCase(

new PropertyResourceBundle((new File("./resources/assets/language/de.properties").exists()

? reader = new FileReader("./resources/assets/language/de.properties")

: (reader2 = new InputStreamReader(

References.class.getResourceAsStream("/assets/language/de.properties")))))

.getString("version"))) {

if (reader != null)

reader.close();

if (reader2 != null)

reader2.close();

if (!Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER***), *LinkOption*.***NOFOLLOW\_LINKS***))

new File(References.***HOME\_FOLDER***).mkdir();

if (Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"),

*LinkOption*.***NOFOLLOW\_LINKS***))

Files.*delete*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"));

Files.*copy*(getClass().getResourceAsStream("/assets/language/de.properties"),

FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"));

Reader reader3;

References.*language* = new PropertyResourceBundle(

reader3 = new FileReader(References.***HOME\_FOLDER*** + "language.properties"));

reader3.close();

}

} else {

if (!Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER***), *LinkOption*.***NOFOLLOW\_LINKS***))

new File(References.***HOME\_FOLDER***).mkdir();

if (Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"),

*LinkOption*.***NOFOLLOW\_LINKS***))

Files.*delete*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"));

Files.*copy*(getClass().getResourceAsStream("/assets/language/de.properties"),

FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"));

Reader reader;

References.*language* = new PropertyResourceBundle(

reader = new FileReader(References.***HOME\_FOLDER*** + "language.properties"));

reader.close();

}

Image i;

if (new File("./resources/assets/textures/logo/KuFA.png").exists())

i = new Image(new File("./resources/assets/textures/logo/KuFA.png").toURI().toString());

else

i = new Image("/assets/textures/logo/KuFA.png");

Parent root = FXMLLoader.*load*(getClass().getResource("/assets/layouts/GUI.fxml"), References.*language*);

Scene s = new Scene(root);

s.getStylesheets().add("/assets/styles/dark\_theme.css");

primaryStage.setMinWidth(400);

primaryStage.setMinHeight(310);

primaryStage.setTitle(***PROJECT\_NAME*** + " | " + ***VERSION***);

primaryStage.setScene(s);

primaryStage.setOnCloseRequest(c -> {

GUIManager.*getInstance*().close(null);

});

primaryStage.centerOnScreen();

primaryStage.initStyle(*StageStyle*.***DECORATED***);

primaryStage.getIcons().add(i);

primaryStage.show();

*scene* = s;

if (GUILoader.*toLoad* != null) {

GUIManager.*getInstance*().load(GUILoader.*toLoad*.getPath());

GUIManager.*getInstance*().inputView.fill();

GUIManager.*getInstance*().cView.fill();

}

}

public void starting2() throws IOException {

*secondaryStage*.show();

*primaryStage*.close();

GUILoader.*primaryStage* = *secondaryStage*;

}

public static Stage getPrimaryStage() {

return *primaryStage*;

}

private void loadLanguage() {

if (References.*language* != null)

return;

try {

Reader reader;

References.*language* = new PropertyResourceBundle(

reader = new FileReader(References.***HOME\_FOLDER*** + "language.properties"));

reader.close();

} catch (IOException e) {

try {

Reader reader;

References.***LOGGER***.info("Failed to load language, backup language was loaded");

References.*language* = new PropertyResourceBundle(

reader = new FileReader("/assets/language/de.properties"));

reader.close();

} catch (IOException e1) {

e1.printStackTrace();

}

}

}

public static void main(String[] args) {

***LOGGER***.info("Starte: " + ***PROJECT\_NAME*** + " | Version: " + ***VERSION***);

for (String arg : args) {

if (new File(arg).exists())

GUILoader.*toLoad* = new File(arg);

}

*launch*(args);

}

} // Ende der Klasse GUILoader

package de.juhu.guiFX;

import static de.juhu.util.References.***LOGGER***;

import static de.juhu.util.References.***LOGGING\_HANDLER***;

import java.awt.Desktop;

import java.io.File;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.URL;

import java.nio.file.FileSystems;

import java.nio.file.Files;

import java.nio.file.LinkOption;

import java.sql.Timestamp;

import java.util.ArrayList;

import java.util.Map.Entry;

import java.util.ResourceBundle;

import java.util.logging.Level;

import org.xml.sax.SAXException;

import de.juhu.config.ConfigManager;

import de.juhu.dateimanager.CSVExporter;

import de.juhu.dateimanager.ExcelExporter;

import de.juhu.dateimanager.LogWriter;

import de.juhu.distributor.Course;

import de.juhu.distributor.Distributor;

import de.juhu.distributor.Save;

import de.juhu.distributor.Student;

import de.juhu.guiFX.lists.CourseView;

import de.juhu.guiFX.lists.InputView;

import de.juhu.guiFX.lists.OutputCourseView;

import de.juhu.guiFX.lists.OutputInformationView;

import de.juhu.guiFX.lists.OutputStudentsView;

import de.juhu.util.Config;

import de.juhu.util.PrintFormat;

import de.juhu.util.References;

import de.juhu.util.Util;

import javafx.animation.FadeTransition;

import javafx.animation.ScaleTransition;

import javafx.animation.TranslateTransition;

import javafx.application.Platform;

import javafx.beans.property.SimpleStringProperty;

import javafx.collections.FXCollections;

import javafx.event.ActionEvent;

import javafx.event.Event;

import javafx.fxml.FXML;

import javafx.fxml.Initializable;

import javafx.scene.CacheHint;

import javafx.scene.control.Button;

import javafx.scene.control.CheckMenuItem;

import javafx.scene.control.ComboBox;

import javafx.scene.control.Label;

import javafx.scene.control.ListView;

import javafx.scene.control.Menu;

import javafx.scene.control.ProgressBar;

import javafx.scene.control.ProgressIndicator;

import javafx.scene.control.Tab;

import javafx.scene.control.TabPane;

import javafx.scene.control.TableColumn;

import javafx.scene.control.TableView;

import javafx.scene.control.TextArea;

import javafx.scene.control.TextField;

import javafx.scene.control.TreeView;

import javafx.scene.image.ImageView;

import javafx.scene.input.DragEvent;

import javafx.scene.input.Dragboard;

import javafx.scene.input.MouseButton;

import javafx.scene.input.MouseEvent;

import javafx.scene.input.TransferMode;

import javafx.scene.layout.AnchorPane;

import javafx.scene.layout.VBox;

import javafx.stage.DirectoryChooser;

import javafx.stage.FileChooser;

import javafx.stage.FileChooser.ExtensionFilter;

import javafx.util.Duration;

/\*\*

\* Diese Klasse verwaltet alle Aktionen des Haupt-GUIs.

\*

\* **@author** Juhu1705

\* **@category** GUI

\* **@version** 2.1

\*/

public class GUIManager implements Initializable {

private static GUIManager *instance*;

public static GUIManager getInstance() {

return *instance*;

}

public static Save *actual*;

*@FXML*

public ImageView i0;

*@FXML*

public ProgressBar p0;

*@FXML*

public TextArea ta1;

*@FXML*

public TreeView<String> configurationTree;

*@FXML*

public Label counter;

*@FXML*

public TextField t1, t2;

*@FXML*

public Button r1, r2, r3, r4, r5, b1, b2, b3, b4, b5, b6;

*@FXML*

public ComboBox<Level> cb1;

*@FXML*

public ComboBox<String> cb2;

*@FXML*

public ComboBox<PrintFormat> cb0;

*@FXML*

public TableView<Student> tv0, tv1, unallocatedStudents, bStudents;

*@FXML*

public TableView<Entry<String, Double>> rates;

*@FXML*

public TableView<Entry<Integer, Integer>> priorities;

*@FXML*

public TableView<Course> tvc, tv2;

*@FXML*

public TableColumn<Student, String> vtc, ntc, k1tc, k1stc, k1ttc, cvtc, cntc, cptc, ckstc, ckttc, unallocatedName,

unallocatedPrename, bName, bPrename, bSubject, bTeacher, bPriority;

*@FXML*

public TableColumn<Course, String> subject, teacher, oSubject, oTeacher, maxStudentCount;

*@FXML*

public TableColumn<Entry<String, Double>, String> rate, rateV;

*@FXML*

public TableColumn<Entry<Integer, Integer>, String> priority, swpriority, percentualPriorities;

*@FXML*

public Tab students, teachers, statistics, tabStudents, tabCourses, tabInput, tabOutput;

*@FXML*

public TabPane masterTabPane;

*@FXML*

public Menu menuStudent, menuCourse;

public ArrayList<TableColumn<Student, String>> atci = new ArrayList<TableColumn<Student, String>>();

public InputView inputView;

public CourseView cView;

public OutputStudentsView outputSView;

public OutputCourseView outputCView;

public OutputInformationView outputIView;

*@FXML*

public VBox config;

*@FXML*

public ListView<String> lv0;

public void onDragOver(DragEvent event) {

event.acceptTransferModes(*TransferMode*.***COPY\_OR\_MOVE***);

}

*@FXML*

public CheckMenuItem mb0, mb1;

private CheckMenuItem last, lastSwitch;

public void onFullScreen(ActionEvent event) {

GUILoader.*getPrimaryStage*().setFullScreen(!GUILoader.*getPrimaryStage*().isFullScreen());

}

public void onDeleteStudentFromActualSave(ActionEvent event) {

if (*actual* == null)

return;

Student student = this.tv1.getSelectionModel().getSelectedItem();

this.*actual*.removeStudent(student);

GUIManager.*actual*.getInformation().update();

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

}

public void onRemoveUnusedCourses(ActionEvent event) {

if (*actual* == null)

return;

for (Course c : *actual*.getAllCoursesAsArray())

if (c.getStudents().isEmpty())

this.*actual*.removeCourse(c);

GUIManager.*actual*.getInformation().update();

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

}

public void onDeleteCourseFromActualSave(ActionEvent event) {

if (*actual* == null)

return;

Course course = this.tv2.getSelectionModel().getSelectedItem();

if (!course.getStudents().isEmpty()) {

this.startErrorFrame("Cannot remove a course with students inside!",

"Please ensure that there are no students in this course. Remove them, or replace their course with some other course!");

return;

}

this.*actual*.removeCourse(course);

GUIManager.*actual*.getInformation().update();

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

}

public void onAddStudentToActualSave(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot add student while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

AddStudentToSaveManager.*student* = null;

***LOGGER***.config("Starting Add Student Window");

Util.*openWindow*("/assets/layouts/AddStudentToCalculation.fxml",

References.*language*.getString("addstudent.text"), GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

public void onEditCourseActualSave(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot modify course data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Course course = this.tv2.getSelectionModel().getSelectedItem();

if (course == null || (course.getSubject() == null || course.getTeacher() == null)

|| (course.getSubject().isEmpty() && course.getTeacher().isEmpty()))

return;

***LOGGER***.config("Starting Add Course Window");

AddCourseToSaveManager.*s* = course.getSubject();

AddCourseToSaveManager.*t* = course.getTeacher();

AddCourseToSaveManager.*mS* = course.getMaxStudentCount();

Util.*openWindow*("/assets/layouts/AddCourseToCalculation.fxml", References.*language*.getString("addcourse.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

public void onAddCourseToActualSave(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot add course while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

AddCourseToSaveManager.*s* = null;

AddCourseToSaveManager.*t* = null;

AddCourseToSaveManager.*mS* = Config.*normalStudentLimit*;

***LOGGER***.config("Starting Add Course Window");

Util.*openWindow*("/assets/layouts/AddCourseToCalculation.fxml", References.*language*.getString("addcourse.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

public void onNextSave(ActionEvent event) {

GUIManager.*actual* = Distributor.*calculated*.next(*actual*);

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

if (GUIManager.*actual*.equals(Distributor.*calculated*.next(*actual*)))

this.b4.setDisable(true);

this.b1.setDisable(false);

this.counter.setText(Integer.*toString*((Distributor.*calculated*.indexOf(*actual*) + 1)));

}

public void onPreviousSave(ActionEvent event) {

GUIManager.*actual* = Distributor.*calculated*.previous(*actual*);

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

if (GUIManager.*actual*.equals(Distributor.*calculated*.previous(*actual*)))

this.b1.setDisable(true);

this.b4.setDisable(false);

this.counter.setText(Integer.*toString*((Distributor.*calculated*.indexOf(*actual*) + 1)));

}

public void addCourse(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot add course while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

***LOGGER***.config("Starting Add Student Window");

AddCourseManager.*mS* = -2;

AddCourseManager.*s* = null;

AddCourseManager.*t* = null;

Util.*openWindow*("/assets/layouts/AddCourse.fxml", References.*language*.getString("addcourse.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

public void onClearCalculations(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot delete calculation while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Distributor.*getInstance*().*calculated*.clear();

boolean hold = Config.*clear*;

Distributor.*getInstance*().reset();

GUIManager.*getInstance*().teachers.setDisable(true);

GUIManager.*getInstance*().students.setDisable(true);

GUIManager.*getInstance*().statistics.setDisable(true);

GUIManager.*getInstance*().b1.setDisable(true);

GUIManager.*getInstance*().b2.setDisable(true);

GUIManager.*getInstance*().b3.setDisable(true);

GUIManager.*getInstance*().b4.setDisable(true);

GUIManager.*getInstance*().b5.setDisable(true);

GUIManager.*getInstance*().b6.setDisable(true);

}

public void onThemeChange(ActionEvent event) {

if (last == null)

last = mb1;

if (lastSwitch == null)

lastSwitch = mb1;

last.setSelected(false);

if (mb0.isSelected()) {

GUILoader.*scene*.getStylesheets().clear();

// GUILoader.scene.getStylesheets().add(StyleSheet.DEFAULT\_STYLE);

last = mb0;

} else if (mb1.isSelected()) {

GUILoader.*scene*.getStylesheets().add("/assets/styles/dark\_theme.css");

last = mb1;

} else {

if (last.equals(mb0)) {

GUILoader.*scene*.getStylesheets().clear();

// GUILoader.scene.getStylesheets().add(StyleSheet.DEFAULT\_STYLE);

last = mb0;

mb1.setSelected(true);

} else if (last.equals(mb1)) {

GUILoader.*scene*.getStylesheets().add("/assets/styles/dark\_theme.css");

last = mb1;

mb0.setSelected(true);

}

}

}

public void moveBadStudentToFirst(MouseEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot move student while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Student s = this.bStudents.getSelectionModel().getSelectedItem();

if (s == null)

return;

if (s.getCourses().length <= 0)

return;

Distributor.*calculated*.peek().getAllStudents()

.get(Distributor.*getInstance*().*calculated*.peek().getAllStudents().indexOf(s))

.setActiveCourse(Distributor.*calculated*.peek().getAllCourses()

.get(Distributor.*calculated*.peek().getAllCourses().indexOf(s.getCourses()[0])));

Distributor.*calculated*.peek().getInformation().getBStudents().remove(s);

Distributor.*calculated*.peek().getInformation().update();

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

}

public void moveUnallocatedStudentToFirst(MouseEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot move student while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Student s = this.unallocatedStudents.getSelectionModel().getSelectedItem();

if (s == null)

return;

if (s.getCourses().length > 0)

s.setActiveCourse(s.getCourses()[0]);

Distributor.*calculated*.peek().getInformation().getUStudents().remove(s);

Distributor.*calculated*.peek().getInformation().update();

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

}

public void onDeleteCourse(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot modify course data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Course course = this.tvc.getSelectionModel().getSelectedItem();

if (course == null || (course.getSubject() == null || course.getTeacher() == null)

|| (course.getSubject().isEmpty() && course.getTeacher().isEmpty()))

return;

Distributor.*getInstance*().removeCourse(course);

GUIManager.*getInstance*().inputView.fill();

GUIManager.*getInstance*().cView.fill();

}

public void onEditCourse(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot modify course data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Course course = this.tvc.getSelectionModel().getSelectedItem();

if (course == null || (course.getSubject() == null || course.getTeacher() == null)

|| (course.getSubject().isEmpty() && course.getTeacher().isEmpty()))

return;

***LOGGER***.config("Starting Add Student Window");

AddCourseManager.*s* = course.getSubject();

AddCourseManager.*t* = course.getTeacher();

AddCourseManager.*mS* = course.getMaxStudentCount();

Util.*openWindow*("/assets/layouts/AddCourse.fxml", References.*language*.getString("addcourse.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

public void onCourseChangeRequest(MouseEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot modify course data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

if (event.getButton().equals(*MouseButton*.***MIDDLE***)) {

Course course = this.tvc.getSelectionModel().getSelectedItem();

if (course == null || (course.getSubject() == null || course.getTeacher() == null)

|| (course.getSubject().isEmpty() && course.getTeacher().isEmpty()))

return;

Distributor.*getInstance*().removeCourse(course);

GUIManager.*getInstance*().inputView.fill();

GUIManager.*getInstance*().cView.fill();

} else if (event.getButton().equals(*MouseButton*.***SECONDARY***)) {

Course course = this.tvc.getSelectionModel().getSelectedItem();

if (course == null || (course.getSubject() == null || course.getTeacher() == null)

|| (course.getSubject().isEmpty() && course.getTeacher().isEmpty()))

return;

***LOGGER***.config("Starting Add Student Window");

AddCourseManager.*s* = course.getSubject();

AddCourseManager.*t* = course.getTeacher();

AddCourseManager.*mS* = course.getMaxStudentCount();

Util.*openWindow*("/assets/layouts/AddCourse.fxml", References.*language*.getString("addcourse.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

}

public void onDeleteStudent(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot modify student data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Student student = this.tv0.getSelectionModel().getSelectedItem();

if (student == null || (student.getName() == null) || student.getPrename() == null

|| (student.getName().isEmpty() && student.getPrename().isEmpty()))

return;

Distributor.*getInstance*().removeStudent(student);

GUIManager.*getInstance*().inputView.fill();

GUIManager.*getInstance*().cView.fill();

}

public void onSwitchCourse(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot modify student data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Student student;

if (this.statistics.isSelected())

student = this.bStudents.getSelectionModel().getSelectedItem();

else

student = this.tv1.getSelectionModel().getSelectedItem();

if (student == null || (student.getName() == null) || student.getPrename() == null

|| (student.getName().isEmpty() && student.getPrename().isEmpty()))

return;

SwitchCourseManager.*student* = student;

***LOGGER***.config("Starting Switch Course Window");

Util.*openWindow*("/assets/layouts/SwitchCourse.fxml", References.*language*.getString("addstudent.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

public void onSwitchCourseUnallocatedStudent(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot modify student data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Student student = this.unallocatedStudents.getSelectionModel().getSelectedItem();

if (student == null || (student.getName() == null) || student.getPrename() == null

|| (student.getName().isEmpty() && student.getPrename().isEmpty()))

return;

SwitchCourseManager.*student* = student;

***LOGGER***.config("Starting Switch Course Window");

Util.*openWindow*("/assets/layouts/SwitchCourse.fxml", References.*language*.getString("addstudent.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

public void onEditStudent(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot modify student data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Student student = this.tv0.getSelectionModel().getSelectedItem();

if (student == null || (student.getName() == null) || student.getPrename() == null

|| (student.getName().isEmpty() && student.getPrename().isEmpty()))

return;

AddStudentManager.*studentID* = student.getID();

***LOGGER***.config("Starting Add Student Window");

Util.*openWindow*("/assets/layouts/AddStudent.fxml", References.*language*.getString("addstudent.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

public void onStudentChangeRequest(MouseEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot modify student data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Student student = this.tv0.getSelectionModel().getSelectedItem();

if (student == null || (student.getName() == null) || student.getPrename() == null

|| (student.getName().isEmpty() && student.getPrename().isEmpty()))

return;

if (event.getButton().equals(*MouseButton*.***MIDDLE***)) {

Distributor.*getInstance*().removeStudent(student);

GUIManager.*getInstance*().inputView.fill();

GUIManager.*getInstance*().cView.fill();

} else if (event.getButton().equals(*MouseButton*.***SECONDARY***)) {

***LOGGER***.config("Starting Add Student Window");

AddStudentManager.*studentID* = student.getID();

Util.*openWindow*("/assets/layouts/AddStudent.fxml", References.*language*.getString("addstudent.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

}

public void onDragDroppedInput(DragEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot import data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Dragboard d = event.getDragboard();

if (d.hasFiles()) {

for (File f : d.getFiles()) {

if (Util.*endsWith*(f.getPath(), ".csv", ".xlsx", ".xls")) {

new Thread(() -> {

new Distributor(f.getPath());

this.inputView.fill();

this.cView.fill();

t1.setText(f.getPath());

}).start();

}

}

}

}

public void addStudent(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot add student while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

***LOGGER***.config("Starting Add Student Window");

AddStudentManager.*studentID* = -1;

Util.*openWindow*("/assets/layouts/AddStudent.fxml", References.*language*.getString("addstudent.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected());

}

public void onFileTypeChanged(ActionEvent event) {

Config.*outputFileType* = cb2.getValue();

}

boolean tabS = true;

public void onSelectionChangedCourse(Event event) {

if (tabCourses == null) {

return;

}

if (((Tab) event.getSource()).isSelected()) {

menuCourse.setDisable(true);

tabS = true;

} else {

tabS = false;

menuCourse.setDisable(false);

}

}

public void onTabIn(Event event) {

if (tabInput == null)

return;

if (((Tab) event.getSource()).isSelected()) {

if (tabS)

menuStudent.setDisable(false);

else

menuCourse.setDisable(false);

} else {

menuCourse.setDisable(true);

menuStudent.setDisable(true);

}

}

public void onShowImportedData(ActionEvent event) {

masterTabPane.getSelectionModel().select(tabInput);

}

public void onSelectionChangedStudent(Event event) {

if (tabStudents == null)

return;

if (!((Tab) event.getSource()).isSelected())

menuStudent.setDisable(false);

else

menuStudent.setDisable(true);

}

public void searchActionInput(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot import data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

FileChooser fc = new FileChooser();

fc.getExtensionFilters().addAll(new ExtensionFilter("Grid Data", "\*.csv", "\*.xls", "\*.xlsx"));

fc.setTitle(References.*language*.getString("choosefile.text"));

File toAdd = new File(t1.getText());

if (!Util.*isBlank*(t1.getText()) && new File(t1.getText()).exists()

&& new File(new File(t1.getText()).getParent()).exists())

fc.setInitialDirectory(new File(new File(t1.getText()).getParent()));

File selected = fc.showOpenDialog(null);

if (selected == null)

return;

if (selected.exists() && Util.*endsWith*(selected.getPath(), ".csv", ".xlsx", ".xls")) {

Config.*inputFile* = selected.getPath();

new Thread(() -> {

new Distributor(selected.getPath());

this.inputView.fill();

this.cView.fill();

t1.setText(selected.getPath());

}).start();

}

}

public void onAbout(ActionEvent event) {

***LOGGER***.config("Starting About Window");

Util.*openWindow*("/assets/layouts/About.fxml", References.*language*.getString("about.text"),

GUILoader.*getPrimaryStage*(), mb1.isSelected()).setResizable(false);

}

public void clearConsole(ActionEvent event) {

***LOGGING\_HANDLER***.clear();

}

public void printFormatChanged(ActionEvent event) {

Config.*printFormat* = cb0.getValue().toString();

***LOGGING\_HANDLER***.updateLog();

}

public void levelChanges(ActionEvent event) {

Config.*maxPrintLevel* = cb1.getValue();

***LOGGING\_HANDLER***.updateLog();

}

// @FXML

// public RadioButton cf1, cf2;

public void searchActionOutput(ActionEvent event) {

// if (cf1.isSelected()) {

DirectoryChooser fc = new DirectoryChooser();

fc.setTitle(References.*language*.getString("choosedirectory.text"));

if (!Util.*isBlank*(t2.getText()) && new File(t1.getText()).exists()

&& new File(new File(t2.getText()).getParent()).exists())

fc.setInitialDirectory(new File(t2.getText()));

File selected = fc.showDialog(null);

if (selected == null)

return;

t2.setText(selected.getParent() + "\\" + selected.getName());

Config.*outputFile* = selected.getParent() + "\\" + selected.getName();

// }

// if (cf2.isSelected()) {

// FileChooser fc = new FileChooser();

// fc.getExtensionFilters().addAll(new ExtensionFilter("Grid Data", "\*.csv", "\*.xls", "\*.xlsx"));

// fc.setTitle("Choose File");

//

// File toAdd = new File(t2.getText());

//

// if (!Util.isBlank(t2.getText()) && new File(t1.getText()).exists()

// && new File(new File(t2.getText()).getParent()).exists())

// fc.setInitialDirectory(new File(new File(t2.getText()).getParent()));

//

// File selected = fc.showSaveDialog(null);

//

// if (selected.getParentFile().exists() && Util.endsWith(selected.getPath(), ".csv", ".xlsx", ".xls")) {

// t2.setText(selected.getParent());

//

// LOGGER.config(selected.getName());

// String[] strings = selected.getName().split("\\.");

// t2.setText(selected.getParent() + "\\" + strings[0]);

// cb2.setValue("." + strings[1]);

// }

// }

}

public void onDragDroppedOutput(DragEvent event) {

Dragboard d = event.getDragboard();

if (d.hasFiles()) {

for (File f : d.getFiles()) {

if (f.isDirectory()) {

t2.setText(f.getParent() + "\\" + f.getName());

// cb2.setValue("FOLDER");

Config.*outputFile* = f.getParent() + "\\" + f.getName();

// Config.outputFileType = "FOLDER";

}

/\*

\* else { if (f.exists() && Util.endsWith(f.getPath(), ".csv", ".xlsx", ".xls"))

\* { // t2.setText(f.getParent());

\*

\* LOGGER.config(f.getName()); String[] strings = f.getName().split("\\.");

\*

\* t2.setText(f.getParent() + "\\" + strings[0]); cb2.setValue("." +

\* strings[1]); } }

\*/

}

}

}

public void runAction(ActionEvent event) {

***LOGGER***.info("Start Distributor");

p0.setVisible(true);

GUIManager.*getInstance*().r1.setDisable(true);

GUIManager.*getInstance*().r2.setDisable(true);

GUIManager.*getInstance*().r3.setDisable(true);

Thread t = new Thread(Distributor.*getInstance*(), "Calculator");

t.start();

}

public void startErrorFrame(String headline, String message) {

***LOGGER***.config("Starting Error Window");

ErrorGuiController.*headline* = headline;

ErrorGuiController.*information* = message;

Util.*openWindow*("/assets/layouts/Error.fxml", "ERROR", GUILoader.*getPrimaryStage*(), mb1.isSelected());

***LOGGER***.info("Error Window Started");

}

public void addCourseToActual(ActionEvent event) {

if (GUIManager.*actual* == null) {

GUIManager.*getInstance*().startErrorFrame("Cannot Add Course to non existing Save!",

"Please calculate something, or load a save.");

return;

}

}

public void onClearDistributor(ActionEvent event) {

Distributor.*getInstance*().clear();

GUIManager.*getInstance*().inputView.fill();

GUIManager.*getInstance*().cView.fill();

}

public void saveAction(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot save calculation while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Timestamp timestamp = new Timestamp(System.*currentTimeMillis*());

***LOGGER***.info("Start Saving files");

***LOGGER***.info("Try to save to " + Config.*outputFile* + Config.*outputFileType*);

Save save = this.*actual*;

if (save == null) {

this.startErrorFrame("No calculation found!",

"There are no data in the distributor. \n Please ensure, to run the calculator and assign the given data, before exporting them!");

return;

}

Distributor.*calculate* = true;

Platform.*runLater*(new Runnable() {

*@Override*

public void run() {

p0.setVisible(true);

GUIManager.*getInstance*().r1.setDisable(true);

GUIManager.*getInstance*().r2.setDisable(true);

GUIManager.*getInstance*().r3.setDisable(true);

de.juhu.guiFX.ProgressIndicator.*getInstance*().setfProgressMax(Config.*runs*).setfProgressValue(-1);

}

});

String s;

if (Util.*isBlank*((s = Config.*outputFile*)) || s.endsWith("/") || s.endsWith("\\"))

return;

boolean xls = false, xlsx = false, csv = false;

switch (Config.*outputFileType*) {

case ".xls":

xls = true;

break;

case ".xlsx":

xlsx = true;

break;

case ".csv":

csv = true;

break;

// case "FOLDER":

// ExcelExporter.writeXLS(s + "/Excel\_OLD", save.writeInformation());

// ExcelExporter.writeXLSX(s + "/KuFA-Zuweiser Ergebnisse", save.writeInformation());

// CSVExporter.writeCSV(s + "/course", save.writeCourseInformation());

// CSVExporter.writeCSV(s + "/student", save.writeStudentInformation());

// LogWriter.writeLog(s + "/logging");

// break;

default:

xlsx = true;

break;

}

try {

if (xls) {

ExcelExporter.*writeXLS*(s + "/calculation" + timestamp.getTime(), save.writeInformation());

}

if (xlsx) {

ExcelExporter.*writeXLSX*(s + "/calculation" + timestamp.getTime(), save.writeInformation());

}

if (csv) {

CSVExporter.*writeCSV*(s + "/course" + timestamp.getTime(), save.writeCourseInformation());

CSVExporter.*writeCSV*(s + "/student" + timestamp.getTime(), save.writeStudentInformation());

}

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim Exportieren der Datei", e);

}

LogWriter.*writeLog*(s + "/logging" + timestamp.getTime());

this.save(s + "/save" + timestamp.getTime());

***LOGGER***.info("Finished Saving files");

Distributor.*calculate* = false;

Platform.*runLater*(new Runnable() {

*@Override*

public void run() {

de.juhu.guiFX.ProgressIndicator.*getInstance*().setfProgressMax(Config.*runs*).setfProgressValue(0);

p0.setVisible(false);

GUIManager.*getInstance*().r1.setDisable(false);

GUIManager.*getInstance*().r2.setDisable(false);

GUIManager.*getInstance*().r3.setDisable(false);

}

});

}

public void saveActualAction(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot save calculation while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Timestamp timestamp = new Timestamp(System.*currentTimeMillis*());

***LOGGER***.info("Start Saving files");

***LOGGER***.info("Try to save to " + Config.*outputFile* + Config.*outputFileType*);

Save save = *actual*;

if (save == null) {

this.startErrorFrame("No calculation found!",

"There are no data in the distributor. \n Please ensure, to run the calculator and assign the given data, before exporting them!");

return;

}

Distributor.*calculate* = true;

Platform.*runLater*(new Runnable() {

*@Override*

public void run() {

p0.setVisible(true);

GUIManager.*getInstance*().r1.setDisable(true);

GUIManager.*getInstance*().r2.setDisable(true);

GUIManager.*getInstance*().r3.setDisable(true);

de.juhu.guiFX.ProgressIndicator.*getInstance*().setfProgressMax(Config.*runs*).setfProgressValue(-1);

}

});

String s;

if (Util.*isBlank*((s = Config.*outputFile*)) || s.endsWith("/") || s.endsWith("\\"))

return;

//

// boolean xls = false, xlsx = false, csv = false;

//

// switch (Config.outputFileType) {

// case ".xls":

// xls = true;

// break;

// case ".xlsx":

// xlsx = true;

// break;

// case ".csv":

// csv = true;

// break;

// case "FOLDER":

// ExcelExporter.writeXLS(s + "/Excel\_OLD", save.writeInformation());

// ExcelExporter.writeXLSX(s + "/KuFA-Zuweiser Ergebnisse", save.writeInformation());

// CSVExporter.writeCSV(s + "/course", save.writeCourseInformation());

// CSVExporter.writeCSV(s + "/student", save.writeStudentInformation());

// LogWriter.writeLog(s + "/logging");

// break;

// default:

// xlsx = true;

// break;

// }

//

// if (xls) {

// ExcelExporter.writeXLS(s + "/calculation" + timestamp.getTime(), save.writeInformation());

// }

// if (xlsx) {

// ExcelExporter.writeXLSX(s + "/calculation" + timestamp.getTime(), save.writeInformation());

// }

// if (csv) {

// CSVExporter.writeCSV(s + "/course" + timestamp.getTime(), save.writeCourseInformation());

// CSVExporter.writeCSV(s + "/student" + timestamp.getTime(), save.writeStudentInformation());

// }

LogWriter.*writeLog*(s + "/logging" + timestamp.getTime());

this.save(s + "/save" + timestamp.getTime());

***LOGGER***.info("Finished Saving files");

Distributor.*calculate* = false;

Platform.*runLater*(new Runnable() {

*@Override*

public void run() {

de.juhu.guiFX.ProgressIndicator.*getInstance*().setfProgressMax(Config.*runs*).setfProgressValue(0);

p0.setVisible(false);

GUIManager.*getInstance*().r1.setDisable(false);

GUIManager.*getInstance*().r2.setDisable(false);

GUIManager.*getInstance*().r3.setDisable(false);

}

});

}

public void runAndSaveAction(ActionEvent event) {

new Thread(new Runnable() {

*@Override*

public void run() {

runAction(event);

try {

Thread.*sleep*(1000);

} catch (InterruptedException e1) {

e1.printStackTrace();

}

while (Distributor.*calculate*) {

try {

Thread.*sleep*(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

saveAction(event);

}

}, "R and S Progress").start();

}

public void onSaveLog(ActionEvent event) {

DirectoryChooser fc = new DirectoryChooser();

fc.setTitle("Choose Directory");

if (!Util.*isBlank*(t2.getText()) && new File(new File(t2.getText()).getParent()).exists())

fc.setInitialDirectory(new File(t2.getText()));

File selected = fc.showDialog(null);

if (selected == null)

return;

LogWriter.*writeLog*(selected.getPath() + "/logging");

}

public void getInformation(ActionEvent event) {

if (t1.isVisible())

t1.setVisible(false);

else

t1.setVisible(true);

}

public void onSaveConfig(ActionEvent event) {

***LOGGER***.info("Start saving config");

if (!Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER***), *LinkOption*.***NOFOLLOW\_LINKS***))

new File(References.***HOME\_FOLDER***).mkdir();

try {

ConfigManager.*getInstance*().save(new File(References.***HOME\_FOLDER*** + "config.cfg"));

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim schreiben der Config datei!", e);

}

***LOGGER***.info("Finished saving config");

}

public void close(ActionEvent event) {

ConfigManager.*getInstance*().onConfigChanged();

this.onSaveConfig(null);

***LOGGER***.info("Close " + References.***PROJECT\_NAME***);

System.*exit*(0);

}

public void onHelp(ActionEvent event) {

try {

if (!Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER***), *LinkOption*.***NOFOLLOW\_LINKS***))

new File(References.***HOME\_FOLDER***).mkdir();

if (Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "help.pdf"),

*LinkOption*.***NOFOLLOW\_LINKS***))

Files.*delete*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "help.pdf"));

Files.*copy*(getClass().getResourceAsStream("/assets/Der Course and Research Paper Assinger.pdf"),

FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "help.pdf"));

Desktop.*getDesktop*().browse(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "help.pdf").toUri());

} catch (IOException e) {

}

}

public void onShowInExcel(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot show data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

Save save = GUIManager.*actual*;

if (save == null) {

GUIManager.*getInstance*().startErrorFrame("Cannot show data without calculating before!",

"Please first run a calculation.");

return;

}

try {

ExcelExporter.*writeXLSX*("test", save.writeInformation());

Desktop.*getDesktop*().browse(new File("test.xlsx").toURI());

} catch (IOException e) {

***LOGGER***.log(Level.***SEVERE***, "Fehler beim öffnen der Prewiev Datei.");

}

}

public void updateInputView() {

while (Config.*maxChooses* > this.atci.size() / 3) {

int number = (this.atci.size() / 3 + 1);

TableColumn<Student, String> k = new TableColumn<>(

References.*language*.getString("course.text") + " " + number),

s = new TableColumn<>(References.*language*.getString("subject.text")),

t = new TableColumn<>(References.*language*.getString("teacher.text"));

k.getColumns().addAll(s, t);

this.atci.add(k);

this.atci.add(s);

this.atci.add(t);

s.setCellValueFactory(e -> {

if (e.getValue().getCourses().length > number - 1)

return new SimpleStringProperty(e.getValue().getCourses()[number - 1].getSubject() == null ? "-"

: e.getValue().getCourses()[number - 1].getSubject());

else

return new SimpleStringProperty("-");

});

t.setCellValueFactory(e -> {

if (e.getValue().getCourses().length > number - 1)

return new SimpleStringProperty(e.getValue().getCourses()[number - 1].getTeacher() == null ? "-"

: e.getValue().getCourses()[number - 1].getTeacher());

else

return new SimpleStringProperty("-");

});

this.tv0.getColumns().add(k);

}

while (Config.*maxChooses* < this.atci.size() / 3 && Config.*maxChooses* != -1 && Config.*maxChooses* != 0) {

this.tv0.getColumns().remove(tv0.getColumns().size() - 1);

this.atci.remove(this.atci.size() - 3);

this.atci.remove(this.atci.size() - 2);

this.atci.remove(this.atci.size() - 1);

}

if (Config.*maxChooses* == -1) {

while (3 > this.atci.size() / 3) {

int number = (this.atci.size() / 3 + 1);

TableColumn<Student, String> k = new TableColumn<>(

References.*language*.getString("course.text") + " " + number),

s = new TableColumn<>(References.*language*.getString("subject.text")),

t = new TableColumn<>(References.*language*.getString("teacher.text"));

k.getColumns().addAll(s, t);

this.atci.add(k);

this.atci.add(s);

this.atci.add(t);

s.setCellValueFactory(e -> {

if (e.getValue().getCourses().length > number - 1)

return new SimpleStringProperty(e.getValue().getCourses()[number - 1].getSubject() == null ? "-"

: e.getValue().getCourses()[number - 1].getSubject());

else

return new SimpleStringProperty("-");

});

t.setCellValueFactory(e -> {

if (e.getValue().getCourses().length > number - 1)

return new SimpleStringProperty(e.getValue().getCourses()[number - 1].getTeacher() == null ? "-"

: e.getValue().getCourses()[number - 1].getTeacher());

else

return new SimpleStringProperty("-");

});

this.tv0.getColumns().add(k);

}

}

}

*@Override*

public void initialize(URL location, ResourceBundle resources) {

ProgressIndicator pi = new ProgressIndicator();

GUIManager.*instance* = this;

this.ta1.setCacheHint(*CacheHint*.***SPEED***);

cb0.setItems(FXCollections.*observableArrayList*(*PrintFormat*.*values*()));

cb1.setItems(FXCollections.*observableArrayList*(Level.***ALL***, Level.***FINEST***, Level.***FINER***, Level.***FINE***, Level.***CONFIG***,

Level.***INFO***, Level.***WARNING***, Level.***SEVERE***, Level.***OFF***));

cb2.setItems(FXCollections.*observableArrayList*(".csv", ".xlsx", ".xls"));

p0.progressProperty().bind(FullProgress.*getInstance*().progressProperty());

p0.setVisible(false);

// Config Stuff

try {

ConfigManager.*getInstance*().register(Config.class);

} catch (IOException e4) {

***LOGGER***.log(Level.***SEVERE***, "Error while register Configuration Elements", e4);

}

if (!Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "config.cfg"),

*LinkOption*.***NOFOLLOW\_LINKS***))

ConfigManager.*getInstance*().loadDefault();

else

try {

ConfigManager.*getInstance*().load(References.***HOME\_FOLDER*** + "config.cfg");

} catch (SAXException | IOException e4) {

***LOGGER***.log(Level.***SEVERE***, "Error while loading Config", e4);

}

ConfigManager.*getInstance*().createMenuTree(this.configurationTree, this.config);

GUILoader.*getPrimaryStage*().setMaximized(Config.*shouldMaximize*);

t2.setText(Config.*outputFile*);

***LOGGING\_HANDLER***.bindTextArea(ta1);

// bp\_preview.setCenter(new HTMLEditor());

// PrintStream ps = new PrintStream(System.out) {

//

// private StringBuilder buffer = new StringBuilder();

//

// @Override

// public void print(String s) {

// if (s == null)

// return;

//

// buffer.append(s);

//

// if (buffer.length() >= 1000) {

// ta1.appendText(buffer.toString());

// ta1.end();

// buffer.setLength(0);

// }

//

// }

// };

// System.setOut(ps);

// System.setErr(ps);

// this.startPicture();

// **INFO**: Tabellen Initierung

// Input Tabelle

this.inputView = new InputView(this.tv0);

this.cView = new CourseView(this.tvc);

this.vtc.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getPrename());

});

this.ntc.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getName());

});

this.k1stc.setCellValueFactory(s -> {

if (s.getValue().getCourses().length > 0)

return new SimpleStringProperty(s.getValue().getCourses()[0].getSubject());

return new SimpleStringProperty("-");

});

this.k1ttc.setCellValueFactory(s -> {

if (s.getValue().getCourses().length > 0)

return new SimpleStringProperty(s.getValue().getCourses()[0].getTeacher());

return new SimpleStringProperty("-");

});

this.atci.add(k1tc);

this.atci.add(k1stc);

this.atci.add(k1ttc);

this.subject.setCellValueFactory(c -> {

return new SimpleStringProperty(c.getValue().getSubject());

});

this.teacher.setCellValueFactory(c -> {

return new SimpleStringProperty(c.getValue().getTeacher());

});

this.maxStudentCount.setCellValueFactory(c -> {

return new SimpleStringProperty(Integer.*toString*(c.getValue().getMaxStudentCount()));

});

// this.k1tc.setCellValueFactory(s -> {

// if (s.getValue().getCourseNames().length < 1)

// return new SimpleStringProperty("-");

// return new SimpleStringProperty(s.getValue().getCourseNames()[0]);

// });

//

// this.k2tc.setCellValueFactory(s -> {

// if (s.getValue().getCourseNames().length < 2)

// return new SimpleStringProperty("-");

// return new SimpleStringProperty(s.getValue().getCourseNames()[1]);

// });

//

// this.k3tc.setCellValueFactory(s -> {

// if (s.getValue().getCourseNames().length < 3)

// return new SimpleStringProperty("-");

// return new SimpleStringProperty(s.getValue().getCourseNames()[2]);

// });

// **INFO**: Output Tabellen

// Output Student View

this.outputSView = new OutputStudentsView(this.tv1);

this.cvtc.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getPrename());

});

this.cntc.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getName());

});

this.ckstc.setCellValueFactory(s -> {

if (s.getValue().getActiveCourse() == null)

return new SimpleStringProperty("No course found");

return new SimpleStringProperty(s.getValue().getActiveCourse().getSubject());

});

this.ckttc.setCellValueFactory(s -> {

if (s.getValue().getActiveCourse() == null)

return new SimpleStringProperty("");

return new SimpleStringProperty(s.getValue().getActiveCourse().getTeacher());

});

this.cptc.setCellValueFactory(s -> {

if (s.getValue().getActiveCourse() != null

&& Util.*isIgnoreCourse*(s.getValue().getActiveCourse().getSubject().split("|")))

return new SimpleStringProperty("-");

return new SimpleStringProperty(Integer.*toString*(s.getValue().getPriority()));

});

// Output Course View

this.outputCView = new OutputCourseView(this.tv2);

this.oSubject.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getSubject());

});

this.oTeacher.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getTeacher());

});

// Output Information View

this.outputIView = new OutputInformationView();

this.bPrename.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getPrename());

});

this.bName.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getName());

});

this.bSubject.setCellValueFactory(s -> {

if (s.getValue().getActiveCourse() != null)

return new SimpleStringProperty(s.getValue().getActiveCourse().getSubject());

return new SimpleStringProperty("-");

});

this.bTeacher.setCellValueFactory(s -> {

if (s.getValue().getActiveCourse() != null)

return new SimpleStringProperty(s.getValue().getActiveCourse().getTeacher());

return new SimpleStringProperty("-");

});

this.bPriority.setCellValueFactory(s -> {

return new SimpleStringProperty(Integer.*toString*(s.getValue().getPriority()));

});

this.rate.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getKey());

});

this.rateV.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getValue().toString());

});

this.priority.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getKey().toString());

});

this.swpriority.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getValue().toString());

});

this.percentualPriorities.setCellValueFactory(s -> new SimpleStringProperty(Double

.*toString*(s.getValue().getValue().doubleValue() / this.*actual*.getInformation().getStudentCount())));

this.unallocatedPrename.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getPrename());

});

this.unallocatedName.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getName());

});

// Update Config

ConfigManager.*getInstance*().onConfigChanged();

t1.setText(Config.*inputFile*);

File file = new File(Config.*inputFile*);

if (file.exists()) {

new Distributor(Config.*inputFile*);

this.inputView.fill();

this.cView.fill();

}

}

*@FXML*

public AnchorPane ap0;

FadeTransition ft;

TranslateTransition tt;

ScaleTransition st;

public void startPicture() {

if (ft == null) {

ft = new FadeTransition(Duration.*millis*(5000 / 2), i0);

ft.setFromValue(0);

ft.setToValue(1);

ft.setOnFinished(e -> {

tt.playFromStart();

st.playFromStart();

});

}

if (tt == null) {

tt = new TranslateTransition(Duration.*millis*(10000 / 2), i0);

tt.setFromX(0);

tt.setFromY(0);

tt.setByX(0);

tt.setByY(0);

tt.setToX(-180);

tt.setToY(-65);

}

if (st == null) {

st = new ScaleTransition(Duration.*millis*(10000 / 2), i0);

st.setFromX(1);

st.setFromY(1);

st.setToX(0.3);

st.setToY(0.3);

}

tt.stop();

st.stop();

ft.stop();

i0.setScaleX(1);

i0.setScaleY(1);

ft.playFromStart();

}

public void save(String location) {

ObjectOutputStream objOut;

try {

Save next = GUIManager.*actual*;

objOut = new ObjectOutputStream(new FileOutputStream(location + ".carp"));

objOut.writeObject(next);

objOut.writeObject(next = Distributor.*calculated*.next(next));

objOut.writeObject(next = Distributor.*calculated*.next(next));

objOut.writeObject(next = Distributor.*calculated*.next(next));

objOut.writeObject(next = Distributor.*calculated*.next(next));

objOut.close();

} catch (IOException e) {

References.***LOGGER***.warning("Can not save the save, please try another location!");

}

}

public void onNew(ActionEvent event) {

this.onClearCalculations(event);

this.onClearDistributor(event);

}

public void onLoad(ActionEvent event) {

if (Distributor.*calculate*) {

GUIManager.*getInstance*().startErrorFrame("Cannot import data while calculating!",

"Please wait until the actual running calculation is finished.");

return;

}

FileChooser fc = new FileChooser();

fc.getExtensionFilters().addAll(new ExtensionFilter("Grid Data", "\*.carp"));

fc.setTitle(References.*language*.getString("choosefile.text"));

File toAdd = new File(t1.getText());

if (!Util.*isBlank*(t1.getText()) && new File(t1.getText()).exists()

&& new File(new File(t1.getText()).getParent()).exists())

fc.setInitialDirectory(new File(new File(t1.getText()).getParent()));

File selected = fc.showOpenDialog(null);

if (selected == null)

return;

if (selected.exists() && Util.*endsWith*(selected.getPath(), ".carp")) {

this.load(selected.getPath());

this.inputView.fill();

this.cView.fill();

}

}

public void load(String location) {

ObjectInputStream objIn;

try {

objIn = new ObjectInputStream(new File(location).toURI().toURL().openConnection().getInputStream());

new Distributor((GUIManager.*actual* = (Save) objIn.readObject()), (Save) objIn.readObject(),

(Save) objIn.readObject(), (Save) objIn.readObject(), (Save) objIn.readObject());

Platform.*runLater*(new Runnable() {

*@Override*

public void run() {

GUIManager.*getInstance*().counter

.setText(Integer.*toString*(Distributor.*calculated*.indexOf(GUIManager.*actual*)));

}

});

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

objIn.close();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

public void onSetEnglish(ActionEvent event) {

try {

if (Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"),

*LinkOption*.***NOFOLLOW\_LINKS***))

Files.*delete*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"));

Files.*copy*(getClass().getResourceAsStream("/assets/language/en.properties"),

FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"));

} catch (IOException e) {

e.printStackTrace();

}

GUIManager.*getInstance*().startErrorFrame("You have to restart the App!",

"The language will be updatet after restarting the App.");

}

public void onSetGerman(ActionEvent event) {

try {

if (Files.*exists*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"),

*LinkOption*.***NOFOLLOW\_LINKS***))

Files.*delete*(FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"));

Files.*copy*(getClass().getResourceAsStream("/assets/language/de.properties"),

FileSystems.*getDefault*().getPath(References.***HOME\_FOLDER*** + "language.properties"));

} catch (IOException e) {

e.printStackTrace();

}

GUIManager.*getInstance*().startErrorFrame("You have to restart the App!",

"The language will be updatet after restarting the App.");}} // Ende der Klasse GUIManager

package de.juhu.guiFX;

import javafx.beans.property.DoubleProperty;

import javafx.beans.property.SimpleDoubleProperty;

/\*\*

\* Behandelt alle Prozesse der Prozessleiste, die nie angezeigt wird. (Kann

\* später noch eingefügt werden, ist aber nicht notwendig)

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class PartProgress {

private static PartProgress *instance*;

public static PartProgress getInstance() {

return *instance* == null ? *instance* = new PartProgress() : *instance*;

}

private DoubleProperty progress;

public final double getProgress() {

return progress == null ? 0 : progress.get();

}

public final void setProgress(double progress) {

// this.progress.set(progress);

}

public final DoubleProperty progressProperty() {

return this.progress == null ? this.progress = new SimpleDoubleProperty(0) : this.progress;

}

} // Ende der Klasse PartProgress

package de.juhu.guiFX;

/\*\*

\* Verwaltet die Fortschrittsanzeige des Prozesses.

\*

\*

\* **@version** 1.0

\* **@category** Distribution, GUI

\* **@author** Juhu1705

\* **@since** 0.0.1

\*/

public class ProgressIndicator {

private int fProgressMax = 0, fProgressValue = 0, aProgressMax = 0, aProgressValue = 0;

private static ProgressIndicator *instance*;

public static ProgressIndicator getInstance() {

return *instance* == null ? new ProgressIndicator() : *instance*;

}

protected ProgressIndicator() {

// References.LOGGER.info(p0 + "|" + p1);

*instance* = this;

}

public ProgressIndicator setfProgressMax(int value) {

this.fProgressMax = value;

return this;

}

public ProgressIndicator setfProgressValue(int value) {

this.fProgressValue = value;

if (this.fProgressValue == -1)

FullProgress.*getInstance*().setProgress(-1);

else

FullProgress.*getInstance*().setProgress((double) this.fProgressValue / (double) this.fProgressMax);

return this;

}

public ProgressIndicator addfProgressValue(int value) {

return this.setfProgressValue(this.fProgressValue + value);

}

public ProgressIndicator setaProgressMax(int value) {

this.aProgressMax = value;

return this;

}

public ProgressIndicator setaProgressValue(int value) {

this.aProgressValue = value;

PartProgress.*getInstance*().setProgress((double) this.aProgressValue / (double) this.aProgressMax);

return this;

}

public ProgressIndicator addaProgressValue(int value) {

return this.setaProgressValue(this.aProgressValue + value);

}

/\*\*

\* **@return** the fProgressMax

\*/

public int getfProgressMax() {

return fProgressMax;

}

/\*\*

\* **@return** the fProgressValue

\*/

public int getfProgressValue() {

return fProgressValue;

}

/\*\*

\* **@return** the aProgressMax

\*/

public int getaProgressMax() {

return aProgressMax;

}

/\*\*

\* **@return** the aProgressValue

\*/

public int getaProgressValue() {

return aProgressValue;

}} // Ende der Klasse ProgressIndicator

package de.juhu.guiFX;

import java.net.URL;

import java.util.ResourceBundle;

import de.juhu.distributor.Course;

import de.juhu.distributor.Distributor;

import de.juhu.distributor.Student;

import de.juhu.guiFX.lists.SwitchCourseView;

import de.juhu.util.References;

import de.juhu.util.Util;

import javafx.application.Platform;

import javafx.beans.property.SimpleStringProperty;

import javafx.event.ActionEvent;

import javafx.fxml.FXML;

import javafx.fxml.Initializable;

import javafx.scene.control.Button;

import javafx.scene.control.TableColumn;

import javafx.scene.control.TableView;

import javafx.scene.control.TextField;

import javafx.stage.Stage;

/\*\*

\* Behandelt alle Aktionen des Fensters zum bearbeiten von Kursen zur

\* Verteilung.

\*

\* **@author** Juhu1705

\* **@category** GUI

\*/

public class SwitchCourseManager implements Initializable {

public static Student *student*;

*@FXML*

private TableView<Course> courses;

*@FXML*

private TextField prename;

*@FXML*

private TextField name;

*@FXML*

private TableColumn<Course, String> teacher, subject;

private SwitchCourseView scw;

public void onSetActive(ActionEvent event) {

Course c = this.courses.getSelectionModel().getSelectedItem();

if (c == null || (c.getSubject() == null || c.getTeacher() == null)

|| (c.getSubject().isEmpty() && c.getTeacher().isEmpty()))

return;

*student*.setActiveCourse(c);

this.scw.run();

}

*@FXML*

public void onFinished(ActionEvent event) {

boolean missingInformation = false;

if (this.prename.getText().isEmpty()) {

this.prename.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (this.name.getText().isEmpty()) {

this.name.setPromptText(References.*language*.getString("valuemissing.text"));

missingInformation = true;

}

if (missingInformation)

return;

*student*.setName(this.name.getText());

*student*.setPrename(this.prename.getText());

Distributor.*calculated*.peek().getInformation().update();

Platform.*runLater*(GUIManager.*getInstance*().outputSView);

Platform.*runLater*(GUIManager.*getInstance*().outputCView);

Platform.*runLater*(GUIManager.*getInstance*().outputIView);

((Stage) ((Button) event.getSource()).getScene().getWindow()).close();

}

*@Override*

public void initialize(URL location, ResourceBundle resources) {

if (*student* == null)

Thread.*currentThread*().interrupt();

References.***LOGGER***.info("Student: " + *student*.toString());

this.teacher.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getTeacher());

});

this.subject.setCellValueFactory(s -> {

return new SimpleStringProperty(s.getValue().getSubject());

});

if (!Util.*isBlank*(*student*.getName()))

this.name.setText(*student*.getName());

if (!Util.*isBlank*(*student*.getPrename()))

this.prename.setText(*student*.getPrename());

this.scw = new SwitchCourseView(courses, *student*);

this.scw.run();

}

} // Ende der Klasse SwitchCourseManager

## Utils

package de.juhu.util;

import org.apache.poi.hssf.usermodel.HSSFCellStyle;

import org.apache.poi.hssf.usermodel.HSSFFont;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;

import org.apache.poi.hssf.util.HSSFColor.HSSFColorPredefined;

import org.apache.poi.ss.usermodel.BorderStyle;

import org.apache.poi.ss.usermodel.FillPatternType;

import org.apache.poi.xssf.usermodel.XSSFCellStyle;

import org.apache.poi.xssf.usermodel.XSSFFont;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;

/\*\*

\* Beinhaltet alle Zell Formatierungsformen, die benutzt werden.

\*

\* **@author** Juhu1705

\* **@category** Import / Export

\*/

public class CellStyles {

public static HSSFCellStyle title(HSSFWorkbook workbook) {

HSSFFont font = workbook.createFont();

font.setBold(true);

font.setColor(*HSSFColorPredefined*.***AQUA***.getIndex());

HSSFCellStyle style = workbook.createCellStyle();

style.setBorderBottom(*BorderStyle*.***THIN***);

style.setBorderRight(*BorderStyle*.***THIN***);

style.setBorderLeft(*BorderStyle*.***THIN***);

style.setFont(font);

style.setFillPattern(*FillPatternType*.***SOLID\_FOREGROUND***);// (HSSFColorPredefined.RED.getIndex());

style.setFillForegroundColor(*HSSFColorPredefined*.***GREY\_50\_PERCENT***.getIndex());

return style;

}

public static HSSFCellStyle normal2(HSSFWorkbook workbook) {

HSSFFont font = workbook.createFont();

font.setBold(false);

font.setColor(*HSSFColorPredefined*.***BLACK***.getIndex());

HSSFCellStyle style = workbook.createCellStyle();

style.setFont(font);

// style.setBorderBottom(BorderStyle.THIN);

style.setBorderRight(*BorderStyle*.***THIN***);

style.setBorderLeft(*BorderStyle*.***THIN***);

// style.setBorderTop(BorderStyle.THIN);

style.setFillPattern(*FillPatternType*.***SOLID\_FOREGROUND***);// (HSSFColorPredefined.RED.getIndex());

style.setFillForegroundColor(*HSSFColorPredefined*.***WHITE***.getIndex());

return style;

}

public static HSSFCellStyle normal1(HSSFWorkbook workbook) {

HSSFFont font = workbook.createFont();

font.setBold(false);

font.setColor(*HSSFColorPredefined*.***BLACK***.getIndex());

HSSFCellStyle style = workbook.createCellStyle();

style.setFont(font);

// style.setBorderBottom(BorderStyle.THIN);

style.setBorderRight(*BorderStyle*.***THIN***);

style.setBorderLeft(*BorderStyle*.***THIN***);

// style.setBorderTop(BorderStyle.THIN);

style.setFillPattern(*FillPatternType*.***SOLID\_FOREGROUND***);// (HSSFColorPredefined.RED.getIndex());

style.setFillForegroundColor(*HSSFColorPredefined*.***GREY\_25\_PERCENT***.getIndex());

return style;

}

public static HSSFCellStyle up(HSSFWorkbook workbook) {

HSSFFont font = workbook.createFont();

font.setBold(false);

font.setColor(*HSSFColorPredefined*.***BLACK***.getIndex());

HSSFCellStyle style = workbook.createCellStyle();

style.setFont(font);

// style.setBorderBottom(BorderStyle.THIN);

// style.setBorderRight(BorderStyle.THIN);

// style.setBorderLeft(BorderStyle.THIN);

style.setBorderTop(*BorderStyle*.***THIN***);

style.setFillPattern(*FillPatternType*.***SOLID\_FOREGROUND***);// (HSSFColorPredefined.RED.getIndex());

style.setFillForegroundColor(*HSSFColorPredefined*.***WHITE***.getIndex());

return style;

}

public static XSSFCellStyle title(XSSFWorkbook workbook) {

XSSFFont font = workbook.createFont();

font.setBold(true);

font.setColor(*HSSFColorPredefined*.***AQUA***.getIndex());

XSSFCellStyle style = workbook.createCellStyle();

style.setBorderBottom(*BorderStyle*.***THIN***);

style.setBorderRight(*BorderStyle*.***THIN***);

style.setBorderLeft(*BorderStyle*.***THIN***);

style.setFont(font);

style.setFillPattern(*FillPatternType*.***SOLID\_FOREGROUND***);// (HSSFColorPredefined.RED.getIndex());

style.setFillForegroundColor(*HSSFColorPredefined*.***GREY\_50\_PERCENT***.getIndex());

return style;

}

public static XSSFCellStyle normal2(XSSFWorkbook workbook) {

XSSFFont font = workbook.createFont();

font.setBold(false);

font.setColor(*HSSFColorPredefined*.***BLACK***.getIndex());

XSSFCellStyle style = workbook.createCellStyle();

style.setFont(font);

// style.setBorderBottom(BorderStyle.THIN);

style.setBorderRight(*BorderStyle*.***THIN***);

style.setBorderLeft(*BorderStyle*.***THIN***);

// style.setBorderTop(BorderStyle.THIN);

style.setFillPattern(*FillPatternType*.***SOLID\_FOREGROUND***);// (HSSFColorPredefined.RED.getIndex());

style.setFillForegroundColor(*HSSFColorPredefined*.***WHITE***.getIndex());

return style;

}

public static XSSFCellStyle normal1(XSSFWorkbook workbook) {

XSSFFont font = workbook.createFont();

font.setBold(false);

font.setColor(*HSSFColorPredefined*.***BLACK***.getIndex());

XSSFCellStyle style = workbook.createCellStyle();

style.setFont(font);

// style.setBorderBottom(BorderStyle.THIN);

style.setBorderRight(*BorderStyle*.***THIN***);

style.setBorderLeft(*BorderStyle*.***THIN***);

// style.setBorderTop(BorderStyle.THIN);

style.setFillPattern(*FillPatternType*.***SOLID\_FOREGROUND***);// (HSSFColorPredefined.RED.getIndex());

style.setFillForegroundColor(*HSSFColorPredefined*.***GREY\_25\_PERCENT***.getIndex());

return style;

}

public static XSSFCellStyle up(XSSFWorkbook workbook) {

XSSFFont font = workbook.createFont();

font.setBold(false);

font.setColor(*HSSFColorPredefined*.***BLACK***.getIndex());

XSSFCellStyle style = workbook.createCellStyle();

style.setFont(font);

// style.setBorderBottom(BorderStyle.THIN);

// style.setBorderRight(BorderStyle.THIN);

// style.setBorderLeft(BorderStyle.THIN);

style.setBorderTop(*BorderStyle*.***THIN***);

style.setFillPattern(*FillPatternType*.***SOLID\_FOREGROUND***);// (HSSFColorPredefined.RED.getIndex());

style.setFillForegroundColor(*HSSFColorPredefined*.***WHITE***.getIndex());

return style;

}

} // Ende der Klasse CellStyles

package de.juhu.util;

import java.util.logging.Level;

import de.juhu.config.ConfigElement;

/\*\*

\* Diese Klasse enthält alle Konfigurationsattribute des Kurs- und

\* Facharbeitszuweisers.

\*

\* **@author** Juhu1705

\* **@category** Config

\* **@version** 2.3

\*/

public class Config {

*@ConfigElement*(defaultValue = "OFF", elementClass = Level.class, description = "loglevel.description", name = "loglevel.text", location = "config")

public static Level *maxPrintLevel* = Level.***ALL***;

*@ConfigElement*(defaultValue = "", elementClass = String.class, description = "inputfile.description", name = "inputfile.text", location = "config.import")

public static String *inputFile*;

*@ConfigElement*(defaultValue = ".xlsx", elementClass = String.class, description = "outputfiletype.description", name = "outputfiletype.text", location = "config.export")

public static String *outputFileType* = ".xlsx";

*@ConfigElement*(defaultValue = "%localappdata%\\Local\\CaRP", elementClass = String.class, description = "outputdirectory.description", name = "outputdirectory.text", location = "config.export")

public static String *outputFile*;

static {

*outputFile* = (System.*getenv*("localappdata") + "\\CaRP\\");

}

*@ConfigElement*(defaultValue = "@PJK", elementClass = String.class, description = "ignoremark.description", name = "ignoremark.text", location = "config.import")

public static String *ignoreStudent*;

*@ConfigElement*(defaultValue = "@Course", elementClass = String.class, description = "coursemark.description", name = "coursemark.text", location = "config.import")

public static String *newCourse*;

*@ConfigElement*(defaultValue = "Student", elementClass = String.class, description = "studentmark.description", name = "studentmark.text", location = "config.import")

public static String *newStudent*;

*@ConfigElement*(defaultValue = "#", elementClass = String.class, description = "commentmark.description", name = "commentmark.text", location = "config.import")

public static String *commentLine*;

*@ConfigElement*(defaultValue = "ALL", elementClass = String.class, description = "printformat.description", name = "printformat.text", location = "config")

public static String *printFormat*;

*@ConfigElement*(defaultValue = "false", elementClass = Boolean.class, description = "shouldMaximize.description", name = "shouldMaximize.text", location = "config")

public static boolean *shouldMaximize* = false;

*@ConfigElement*(defaultValue = "false", elementClass = Boolean.class, description = "shortnames.description", name = "shortnames.text", location = "config.export")

public static boolean *shortNames*;

*@ConfigElement*(defaultValue = "true", elementClass = Boolean.class, description = "firstprename.description", name = "firstprename.text", location = "config.export")

public static boolean *firstPrename*;

*@ConfigElement*(defaultValue = "true", elementClass = Boolean.class, description = "usenewgoodness.description", name = "usenewgoodness.text", location = "config.calculation")

public static boolean *useNewGoodness* = true;

// @ConfigElement(defaultValue = "true", elementClass = Boolean.class, description = "newimproving.description", name = "newimproving.text", location = "config.calculation")

// public static boolean newImproving = true;

*@ConfigElement*(defaultValue = "true", elementClass = Boolean.class, description = "clearcalculationdata.description", name = "clearcalculationdata.text", location = "config.import")

public static boolean *clear* = true;

*@ConfigElement*(defaultValue = "false", elementClass = Boolean.class, description = "dontask.description", name = "dontask.text", location = "config.import")

public static boolean *rememberDecision* = false;

*@ConfigElement*(defaultValue = "true", elementClass = Boolean.class, description = "allowDoubles.description", name = "allowDoubles.text", location = "config.import")

public static boolean *allowDoubleStudents* = true;

*@ConfigElement*(defaultValue = "100", elementClass = Integer.class, description = "runcount.description", name = "runcount.text", location = "config.calculation")

public static int *runs* = 100;

*@ConfigElement*(defaultValue = "5", elementClass = Integer.class, description = "newcalculating.description", name = "newcalculating.text", location = "config.calculation")

public static int *newCalculating* = 5;

*@ConfigElement*(defaultValue = "5", elementClass = Integer.class, description = "improvecalculation.description", name = "improvecalculation.text", location = "config.calculation")

public static int *improvingOfCalculation* = 5;

*@ConfigElement*(defaultValue = "3", elementClass = Integer.class, description = "coosemaximum.description", name = "coosemaximum.text", location = "config.import")

public static int *maxChooses* = 3;

*@ConfigElement*(defaultValue = "3", elementClass = Integer.class, description = "studentlimit.description", name = "studentlimit.text", location = "config.import")

public static int *normalStudentLimit* = 3;

*@ConfigElement*(defaultValue = "3", elementClass = Integer.class, description = "rateindex.description", name = "rateindex.text", location = "config.calculation")

public static int *powValue* = 3;

} // Ende der Klasse Config

package de.juhu.util;

import static de.juhu.util.PrintFormat.***ALL***;

import static de.juhu.util.PrintFormat.***CLASS***;

import static de.juhu.util.PrintFormat.***LEVEL***;

import static de.juhu.util.PrintFormat.***LOGGER***;

import static de.juhu.util.PrintFormat.***ONLY\_MESSAGE***;

import static de.juhu.util.PrintFormat.***TIME***;

import static java.time.LocalDateTime.*now*;

import java.time.format.DateTimeFormatter;

import java.util.logging.Formatter;

import java.util.logging.LogRecord;

/\*\*

\* Formatiert den Log entsprechend den Configeinstellungen.

\*

\* **@author** Juhu1705

\* **@category** Log

\*/

public class LoggingFormatter extends Formatter {

private static final DateTimeFormatter ***timeformatter*** = DateTimeFormatter.*ofPattern*("HH:mm:ss");

*@Override*

public String format(LogRecord record) {

StringBuilder sb = new StringBuilder();

if (record == null || Util.*isBlank*(record.getSourceClassName()))

return "";

String[] className = record.getSourceClassName().split("\\.");

*PrintFormat* format = Config.*printFormat* != null ? *PrintFormat*.*valueOf*(Config.*printFormat*) : ***ALL***;

if (format.getLevel() >= ***TIME***.getLevel())

sb.append("[" + *now*().format(***timeformatter***) + "] ");

if (format.getLevel() >= ***LEVEL***.getLevel())

sb.append("[" + record.getLevel() + "] ");

if (format.getLevel() >= ***CLASS***.getLevel())

sb.append("[" + Thread.*currentThread*().getName() + ":" + className[className.length - 1] + ":"

+ record.getSourceMethodName() + "] ");

if (format.getLevel() >= ***LOGGER***.getLevel()) {

if (record.getLoggerName() != null)

sb.append("[" + record.getLoggerName() + "] ");

}

if (format.getLevel() >= ***ONLY\_MESSAGE***.getLevel())

sb.append(record.getMessage());

sb.append("\n");

Throwable thrown = record.getThrown();

if (thrown != null) {

sb.append(thrown);

sb.append("\n");

for (StackTraceElement ste : thrown.getStackTrace()) {

sb.append(" at ");

sb.append(ste);

sb.append("\n");

}

}

// sb.append("[" + now().format(timeformatter) + "] [" + record.getLevel() + "|"

// + Thread.currentThread().getName() + ":" + className[className.length - 1] + ":"

// + record.getSourceMethodName() + "] " + record.getMessage() + "\n");

return sb.toString();

}

} // Ende der Klasse LoggingFormatter

package de.juhu.util;

import static java.lang.System.***err***;

import static java.lang.System.***out***;

import static java.util.logging.Level.***WARNING***;

import java.util.ArrayList;

import java.util.logging.Handler;

import java.util.logging.Level;

import java.util.logging.LogRecord;

import javafx.scene.control.TextArea;

/\*\*

\* Behandelt Log Nachrichten und speichert sie zwischen.

\*

\* **@author** Juhu1705

\* **@category** Log

\*/

public class LoggingHandler extends Handler implements Runnable {

private StringBuilder log = new StringBuilder();

private ArrayList<LogRecord> history = new ArrayList<>();

private ArrayList<TextArea> logPrinter = new ArrayList<>();

private boolean isRunning = false, changed = false;

public StringBuilder getLog() {

return this.log;

}

public void clear() {

log.setLength(0);

changed = true;

}

public void updateLog() {

this.clear();

ArrayList<LogRecord> copy = new ArrayList<>(history);

this.history.clear();

copy.forEach(h -> this.publish(h));

}

public void bindTextArea(TextArea textArea) {

this.logPrinter.add(textArea);

if (!this.isRunning)

new Thread(this, "Log Updater").start();

}

*@Override*

public void run() {

this.isRunning = true;

while (!logPrinter.isEmpty()) {

if (changed) {

this.logPrinter.forEach(printer -> {

printer.setText(log.toString());

printer.end();

});

this.changed = false;

}

try {

Thread.*sleep*(500);

} catch (InterruptedException e) {

References.***LOGGER***.log(Level.***WARNING***, "Error while sleeping!", e);

}

}

this.isRunning = false;

}

*@Override*

public void publish(LogRecord record) {

this.history.add(record);

if (Config.*maxPrintLevel* != null && Config.*maxPrintLevel*.intValue() > record.getLevel().intValue())

return;

String output = this.getFormatter().format(record);

this.changed = true;

this.log.append(output);

if (record.getLevel().intValue() >= ***WARNING***.intValue()) {

***err***.print(output);

} else {

***out***.print(output);

}

}

*@Override*

public void flush() {

}

*@Override*

public void close() throws SecurityException {

}

} // Ende der Klasse LoggingHandler

package de.juhu.util;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Future;

/\*\*

\* Ein einfacher Threadbasierter MergeSort-Allgorithmus zum Sortieren von

\* {@link Comparable}-Objekten.

\*

\* **@author** Juhu1705

\*

\* **@param** <T> Die Objektklasse der zu sortierenden Objekte

\*/

public class MergeSort<T extends Comparable> implements Callable {

private ArrayList<T> input;

private ExecutorService pool;

/\*\*

\* Initialisierung des Mergesorts. Aufgerufen wird der Allgorithmus durch

\* {@link ExecutorService#submit(Callable)}. Als Callable wird diese Klasse

\* mitgegeben. Das zurückgegebene {@link Future}-Objekt gibt über

\* {@link Future#get()} die sortierte {@link ArrayList} zurück.

\*

\* **@param** input Die zu sortierende {@link ArrayList}.

\* **@param** pool Der {@link ExecutorService} über den die Berechnung läuft.

\* **@implNote** Der {@link ExecutorService} muss

\* {@link ArrayList#size()}{@code / 2 + 10} groß sein.

\*/

public MergeSort(ArrayList<T> input, ExecutorService pool) {

this.input = input;

this.pool = pool;

}

*@Override*

public ArrayList<T> call() throws Exception {

if (this.input.size() <= 1)

return this.input;

List<T> toSort1 = this.input.subList(0, this.input.size() / 2);

List<T> toSort2 = this.input.subList(this.input.size() / 2, this.input.size());

// Future<ArrayList<T>> fsorted1 = this.pool.submit(new MergeSort<T>(new

// ArrayList<T>(toSort1), pool));

Future<ArrayList<T>> fsorted2 = this.pool.submit(new MergeSort<T>(new ArrayList<T>(toSort2), pool));

ArrayList<T> sorted1 = /\*\* fsorted1.get() \*/

this.mergeThreadless(new ArrayList<T>(toSort1), pool);

ArrayList<T> sorted2 = fsorted2.get();

// LOGGER.info((sorted1.size() + sorted2.size()) + "");

ArrayList<T> sorted = new ArrayList<T>(this.input.size());

while (!sorted1.isEmpty()) {

if (sorted2.isEmpty())

sorted.add(sorted1.remove(0));

else if (sorted1.get(0).compareTo(sorted2.get(0)) < 0)

sorted.add(sorted1.remove(0));

else {

while (!(sorted1.get(0).compareTo(sorted2.get(0)) < 0)) {

sorted.add(sorted2.remove(0));

if (sorted2.isEmpty())

break;

}

sorted.add(sorted1.remove(0));

}

}

sorted.addAll(sorted2);

return sorted;

}

/\*\*

\* **@implNote** Diese Methode kann zu beginn aufgerufen werden. Sie öffnet jedoch

\* selbst weitere Threads im mitgegebenen {@link ExecutorService}

\* **@param** input Die zu sortierende {@link ArrayList}.

\* **@param** pool Der {@link ExecutorService} über den die Berechnung läuft.

\* **@return** Sortierte ArrayList

\* **@throws** Exception Wenn ein Fehler bei der Berechnung auftritt.

\*/

private ArrayList<T> mergeThreadless(ArrayList<T> input, ExecutorService pool) throws Exception {

if (input.size() <= 1)

return input;

List<T> toSort1 = input.subList(0, input.size() / 2);

List<T> toSort2 = input.subList(input.size() / 2, input.size());

Future<ArrayList<T>> fsorted2 = pool.submit(new MergeSort<T>(new ArrayList<T>(toSort2), pool));

ArrayList<T> sorted1 = this.mergeThreadless(new ArrayList<T>(toSort1), pool);

ArrayList<T> sorted2 = fsorted2.get();

// LOGGER.info((sorted1.size() + sorted2.size()) + "");

ArrayList<T> sorted = new ArrayList<T>(input.size());

while (!sorted1.isEmpty()) {

if (sorted2.isEmpty())

sorted.add(sorted1.remove(0));

else if (sorted1.get(0).compareTo(sorted2.get(0)) < 0)

sorted.add(sorted1.remove(0));

else {

while (!(sorted1.get(0).compareTo(sorted2.get(0)) < 0)) {

sorted.add(sorted2.remove(0));

if (sorted2.isEmpty())

break;

}

sorted.add(sorted1.remove(0));

}

}

sorted.addAll(sorted2);

return sorted;

}

} // Ende der Klasse MergeSort<T extends Comparable>

package de.juhu.util;

/\*\*

\* Das Print Format ist wichtig für die Logger Klasse und bestimmt die

\* angegebenen zusatzinformationen. Je kleiner der Wert, desto weniger

\* zusatzinformationen werden mitgegeben.

\*

\* **@author** Juhu1705

\* **@category** Log

\*/

public enum *PrintFormat* {

***LOGGER***(3), ***CLASS***(4), ***ONLY\_MESSAGE***(0), ***TIME***(1), ***LEVEL***(2), ***ALL***(100);

private int level;

PrintFormat(int level) {

this.level = level;

}

public int getLevel() {

return this.level;

}

} // Ende des Enums *PrintFormat*

package de.juhu.util;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Iterator;

import java.util.List;

import java.util.ListIterator;

import de.juhu.distributor.Save;

/\*\*

\* Eine kleine Listenklasse basierend auf der ArrayList, die zum sortierten

\* Speichern der hinzugefügten Objekte dient.

\*

\* **@author** Juhu1705

\* **@implNote** Funktioniert nur für die {@link Save}-Klasse nach aktuellem stand.

\* Es gibt keine Garantie, dass sie bei anderen Klassen als Eingabe

\* funktioniert.

\*/

public class PriorityQueue<T extends Comparable<T>> implements List<T> {

public ArrayList<T> list;

public PriorityQueue() {

this.list = new ArrayList<T>();

}

public PriorityQueue(int i) {

this.list = new ArrayList<T>();

}

public T poll() {

if (this.list == null || this.list.isEmpty())

return null;

return this.list.remove(0);

}

public T peek() {

if (this.list == null || this.list.isEmpty())

return null;

return this.get(0);

}

public T previous(T actual) {

int newPosition = this.list.indexOf(actual) - 1;

if (newPosition < 0)

return actual;

return this.list.get(newPosition);

}

public T next(T actual) {

int newPosition = this.list.indexOf(actual) + 1;

if (newPosition < 0 || newPosition >= this.list.size() || newPosition >= 5)

return actual;

return this.list.get(newPosition);

}

public int indexOf(T actual) {

return this.list.indexOf(actual);

}

*@Override*

public int size() {

return this.list.size();

}

*@Override*

public boolean isEmpty() {

return this.list.isEmpty();

}

*@Override*

public boolean contains(Object o) {

return this.list.contains(o);

}

*@Override*

public Iterator<T> iterator() {

return this.list.iterator();

}

*@Override*

public Object[] toArray() {

return this.list.toArray();

}

*@SuppressWarnings*("hiding")

*@Override*

public <T> T[] toArray(T[] a) {

return this.list.toArray(a);

}

*@Override*

public boolean add(T e) {

if (e == null)

return false;

if (this.list == null)

return (this.list = new ArrayList<>()).add(e);

if (this.list.isEmpty())

return this.list.add(e);

boolean found = false;

for (int i = 0; i < this.list.size(); i++) {

if (e.compareTo(this.get(i)) == 0) {

found = true;

break;

}

if (e.compareTo(this.get(i)) > 0) {

this.list.add(i, e);

found = true;

break;

}

}

if (!found)

this.list.add(e);

return true;

}

*@Override*

public boolean remove(Object o) {

return this.list.remove(o);

}

*@Override*

public boolean containsAll(Collection<?> c) {

return this.list.containsAll(c);

}

*@Override*

public boolean addAll(Collection<? extends T> c) {

boolean isAdded = true;

for (T t : c) {

isAdded = this.add(t) && isAdded;

}

return isAdded;

}

*@Override*

public boolean addAll(int index, Collection<? extends T> c) {

return this.addAll(c);

}

*@Override*

public boolean removeAll(Collection<?> c) {

return this.list.removeAll(c);

}

*@Override*

public boolean retainAll(Collection<?> c) {

return this.list.retainAll(c);

}

*@Override*

public void clear() {

this.list.clear();

}

*@Override*

public T get(int index) {

return this.list.get(index);

}

*@Override*

public T set(int index, T element) {

return null;

}

*@Override*

public void add(int index, T element) {

}

*@Override*

public T remove(int index) {

return this.list.remove(index);

}

*@Override*

public int indexOf(Object o) {

return this.list.indexOf(o);

}

*@Override*

public int lastIndexOf(Object o) {

return this.list.lastIndexOf(o);

}

*@Override*

public ListIterator<T> listIterator() {

return this.list.listIterator();

}

*@Override*

public ListIterator<T> listIterator(int index) {

return this.list.listIterator(index);

}

*@Override*

public List<T> subList(int fromIndex, int toIndex) {

return this.list.subList(fromIndex, toIndex);

}

} // Ende der Klasse PriorityQueue<T extends Comparable<T>>

package de.juhu.util;

import java.awt.Component;

import java.awt.Rectangle;

import java.io.File;

import java.io.IOException;

import java.util.List;

import de.juhu.distributor.Course;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.scene.image.Image;

import javafx.stage.Modality;

import javafx.stage.Stage;

import javafx.stage.StageStyle;

/\*\*

\* Beinhaltet nützliche Methoden

\*

\* **@author** Fabius

\* **@category** Util

\*/

public class Util {

*@Deprecated*

private static void setPositionCorrectly(Component c, double xRelation, double yRelation, double widthRelation,

double heightRelation, double newWindowWidth, double newWindowHeight) {

double dx = xRelation \* newWindowWidth, dy = yRelation \* newWindowHeight,

dwidth = widthRelation \* newWindowWidth, dheight = heightRelation \* newWindowHeight;

int x = (int) dx, y = (int) dy, width = (int) dwidth, height = (int) dheight;

if (isReliable(x / (int) newWindowWidth, xRelation) && isReliable(y / (int) newWindowHeight, yRelation)

&& isReliable(width / (int) newWindowWidth, widthRelation)

&& isReliable(height / (int) newWindowHeight, heightRelation))

c.setBounds(x, y, width, height);

}

*@Deprecated*

private static boolean isReliable(double d1, double d2) {

return d1 == d2 || ((d1 + 2) > d2 && (d1 - 2) < d2);

}

*@Deprecated*

public static void setPositionCorrectly(Component jC, Rectangle c, int newWindowWidth, int newWindowHeight) {

Util.setPositionCorrectly(jC, c.x, c.y, c.width, c.height, newWindowWidth, newWindowHeight);

}

*@Deprecated*

public static void setPositionCorrectly(Component c, Rectangle relationBounds, Rectangle windowBounds) {

Util.setPositionCorrectly(c, relationBounds.x, relationBounds.y, relationBounds.width, relationBounds.height,

windowBounds.width, windowBounds.height);

}

*@Deprecated*

public static void setPositionCorrectlyWithWindowSettings(Component c, Rectangle previousWindowBounds,

Rectangle windowBounds) {

Rectangle unrelatedBounds = c.getBounds();

Util.setPositionCorrectly(c, (double) unrelatedBounds.x / (double) previousWindowBounds.width,

(double) unrelatedBounds.y / (double) previousWindowBounds.height,

(double) unrelatedBounds.width / (double) previousWindowBounds.width,

(double) unrelatedBounds.height / (double) previousWindowBounds.height, windowBounds.width,

windowBounds.height);

}

*@Deprecated*

public static void setPositionWithStartPosition(Component c, Rectangle startPosition, Rectangle windowPosition,

Rectangle startWindow) {

double x = ((double) startPosition.x / (double) startWindow.width) \* windowPosition.width,

y = ((double) startPosition.y / (double) startWindow.height) \* windowPosition.height,

width = ((double) startPosition.width / (double) startWindow.width) \* windowPosition.width,

height = ((double) startPosition.height / (double) startWindow.height) \* windowPosition.height;

c.setBounds((int) x, (int) y, (int) width, (int) height);

}

*@Deprecated*

public static int quad(int i, int height) {

if (height < 0)

return 1 / quad(i, Math.*abs*(height));

if (height == 0)

return 0;

if (height == 1)

return i;

return i \* quad(i, height - 1);

}

/\*\*

\*

\* **@param** input

\* **@return**

\*/

public static boolean isBlank(String input) {

return input == null || input.isEmpty() || input.trim().isEmpty();

}

/\*\*

\*

\* **@param** toCheck

\* **@param** strings

\* **@return**

\*/

public static boolean endsWith(String toCheck, String... strings) {

for (String s : strings) {

if (toCheck.endsWith(s))

return true;

}

return false;

}

/\*\*

\*

\* **@param** line

\* **@return**

\*/

public static String[] removeFirst(String[] line) {

String[] newline = new String[line.length - 1];

for (int i = 1; i < line.length; i++) {

newline[i - 1] = line[i];

}

return newline;

}

/\*\*

\*

\* **@param** courses

\* **@return**

\*/

public static int maxStudentCount(List<Course> courses) {

int maxCount = 0;

for (Course c : courses)

maxCount = maxCount >= c.getStudents().size() ? maxCount : c.getStudents().size();

return maxCount;

}

/\*\*

\*

\* **@param** name

\* **@return**

\*/

public static boolean isIgnoreCourse(String... name) {

for (String s : Config.*ignoreStudent*.split("\\|"))

for (String s1 : name)

if (s.equalsIgnoreCase(s1))

return true;

return false;

}

/\*\*

\*

\* **@param** resourceLocation

\* **@param** title

\* **@param** parent

\* **@param** darkTheme

\* **@return**

\*/

public static Stage openWindow(String resourceLocation, String title, Stage parent, boolean darkTheme) {

Stage primaryStage = new Stage();

Image i;

if (new File("./resources/assets/textures/logo/KuFA.png").exists())

i = new Image(new File("./resources/assets/textures/logo/KuFA.png").toURI().toString());

else

i = new Image("/assets/textures/logo/KuFA.png");

Parent root = null;

try {

root = FXMLLoader.*load*(Util.class.getClass().getResource(resourceLocation), References.*language*);

} catch (IOException e) {

return null;

}

Scene s = new Scene(root);

if (darkTheme) {

s.getStylesheets().add("/assets/styles/dark\_theme.css");

}

primaryStage.setMinWidth(200);

primaryStage.setMinHeight(158);

primaryStage.setTitle(title);

primaryStage.setScene(s);

primaryStage.initModality(*Modality*.***WINDOW\_MODAL***);

if (parent != null)

primaryStage.initOwner(parent);

primaryStage.initStyle(*StageStyle*.***DECORATED***);

primaryStage.getIcons().add(i);

primaryStage.show();

return primaryStage;

}} // Ende der Klasse Util

package de.juhu.util;

import static java.util.logging.Level.***ALL***;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.PropertyResourceBundle;

import java.util.Random;

import java.util.logging.Handler;

import java.util.logging.Logger;

import java.util.logging.StreamHandler;

import org.apache.maven.model.Model;

import org.apache.maven.model.io.xpp3.MavenXpp3Reader;

import org.codehaus.plexus.util.xml.pull.XmlPullParserException;

/\*\*

\* Beinhaltet alle Referenzobjekte

\*

\* **@author** Juhu1705

\* **@category** Util

\*/

public class References {

/\*\*

\* Weiß nicht wieso es noch drinnen ist.

\*/

public static StreamHandler *sh*;

/\*\*

\* Die Klasse zur Formatierung der Log Nachrichten.

\*/

public static LoggingFormatter *lf*;

/\*\*

\* Die Sprachdatei, die Momentan benutzt wird.

\*/

public static PropertyResourceBundle *language*;

/\*\*

\* Der zum Logger zugehörige {@link LoggingHandler}.

\*/

public static final LoggingHandler ***LOGGING\_HANDLER***;

static {

***LOGGING\_HANDLER*** = new LoggingHandler();

}

/\*\*

\* Der Speicherpfad des Carp-Assigners

\*/

public static final String ***HOME\_FOLDER***;

static {

***HOME\_FOLDER*** = System.*getProperty*("user.home") + "/.CaRP/";

}

/\*\*

\* Der Logger, über den alle Log-Nachrichten behandelt werden.

\*/

public static final Logger ***LOGGER***;

static {

***LOGGER*** = Logger.*getLogger*("Wahlverteiler");

***LOGGER***.setUseParentHandlers(false);

Handler handler = ***LOGGING\_HANDLER***;

handler.setFormatter(*lf* = new LoggingFormatter());

handler.setLevel(***ALL***);

***LOGGER***.addHandler(handler);

***LOGGER***.setLevel(***ALL***);

}

/\*\*

\* Der Versionsnummer, wie sie in der pom.xml Datei angegeben wurde.

\*/

public static final String ***VERSION***;

static {

MavenXpp3Reader reader = new MavenXpp3Reader();

Model model = null;

try {

if ((new File("pom.xml")).exists())

model = reader.read(new FileReader("pom.xml"));

else

model = reader.read(new InputStreamReader(References.class

.getResourceAsStream("/META-INF/maven/de.juhu/Course\_and\_Research\_Paper-Assigner/pom.xml")));

} catch (FileNotFoundException e) {

} catch (IOException e) {

} catch (XmlPullParserException e) {

}

if (model == null)

***VERSION*** = "0.1.0";

else

***VERSION*** = model.getVersion();

// VERSION = "Snapshot-0.0.1";

}

/\*\*

\* Der Projektname, wie er in der pom.xml Datei angegeben wurde.

\*/

public static final String ***PROJECT\_NAME***;

static {

MavenXpp3Reader reader = new MavenXpp3Reader();

Model model = null;

try {

if ((new File("pom.xml")).exists())

model = reader.read(new FileReader("pom.xml"));

else

model = reader.read(new InputStreamReader(References.class

.getResourceAsStream("/META-INF/maven/de.juhu/Course\_and\_Research\_Paper-Assigner/pom.xml")));

} catch (FileNotFoundException e) {

} catch (IOException e) {

} catch (XmlPullParserException e) {

}

if (model == null)

***PROJECT\_NAME*** = "CaRP Assigner";

else

***PROJECT\_NAME*** = model.getName();

// PROJECT\_NAME = "KuFa Zuweiser";

}

/\*\*

\* Der Zufallsgenerator

\*/

public static final Random ***RAND\_GEN*** = new Random();

} // Ende der Klasse References

# Quellen- und Literaturverzeichnis

## Internet

Stack Overflow: „community of programmer helping each other“,

<http://stackoverflow.com/questions/>

Maven Repository: „Search/Browse/Explore“,

<https://mvnrepository.com/>

GitHub: „The world’s leading software development platform“,

<https://github.com/>

Eclipse,

<https://www.eclipse.org/>

Scene Builder

<https://gluonhq.com/products/scene-builder/>

## Bilder

Abb. 1: Alte KuFA-Anwendung

Quelle: Selbstangefertigter Screenshot von der alten KuFA-Anwendung

Abb. 2: Klassenstruktur des alten KuFA-Zuweisers

Quelle: Selbstangefertigter Screenshot vom Klassenbaum des alten KuFA-Zuweisers

Abb. 3: Klassenstruktur des CaRP-Assigners

Quelle: Selbstangefertigter Screenshot vom Klassenbaum des CaRP-Assigners

Abb. 4: Eingabedatenvorschau des CaRP-Assigners

Quelle: Selbstangefertigter Screenshot vom lauffähigen CaRP-Assigner

Abb. 5: Ansicht des GUIs im „Scene Builder“

Quelle: Selbstangefertigter Screenshot von der GUI Vorschau im „Scene Builder“

Abb. 6: „Github Repositoriy“ des CaRP-Assigners

Quelle: Selbstangefertigter Screenshot vom GitHub Repository des CaRP-Assigners (stand: 13.03.2020)

Abb. 7: Exporttabelle der Kurs-verteilung mit Beispieldaten

Quelle: Selbstangefertigter Screenshot von einer aus dem CaRP-Assigner exportierten Beispieldatei

Abb. 8: Überarbeitete Einstellungsansicht

Quelle: Selbstangefertigter Screenshot vom lauffähigen CaRP-Assigner

Abb. 9: Der Reiter „Eingabe/Ausgabe“

Quelle: Selbstangefertigter Screenshot vom lauffähigen CaRP-Assigner

Abb. 10: Vorschau der Statistik einer Verteilung

Quelle: Selbstangefertigter Screenshot vom lauffähigen CaRP-Assigner

Abb. 11: Bearbeitungsfenster für einen Schüler mit Daten

Quelle: Selbstangefertigter Screenshot vom lauffähigen CaRP-Assigner

Abb. 12: Fenster zum Bearbeiten eines Schülers in einer Verteilung

Quelle: Selbstangefertigter Screenshot vom lauffähigen CaRP-Assigner

Abb. 13: Konfigurationsfenster mit rechts eingeblendetem Beschreibungstext für die links ausgewählte Einstellung

Quelle: Selbstangefertigter Screenshot vom lauffähigen CaRP-Assigner

## Java-Abhängigkeiten

POI, POI-OOXML [The Apache Software Foundation]

<https://mvnrepository.com/artifact/org.apache.poi/poi>

<https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml>

XMLPULL [Stafan Haustein, Alexander Slominski]

<https://mvnrepository.com/artifact/xmlpull/xmlpull>

FontAwesomeFX [Dave Gandy, Jens Deters]

<https://bitbucket.org/Jerady/fontawesomefx/>

SAX [sax]

<https://mvnrepository.com/artifact/sax/sax/2.0.1>

maven-model [Apache]

<https://mvnrepository.com/artifact/org.apache.maven/maven-model>

launch4j-maven-plugin [Lukaszenart]

<https://github.com/lukaszlenart/launch4j-maven-plugin>

# Abschlusserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken oder Internetquellen als solche kenntlich gemacht habe.

|  |  |
| --- | --- |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Ort, Datum | Unterschrift des Schülers |