ORGANIZING TOOL

Abgabe: 10.08.2020

IntelliJ IDEA: organizing-tool

GitLab: https://gitlab.mi.hdm-stuttgart.de/mh334/organizing-tool

Autoren:

Hannes Frey (hf018)

Micha Huhn (mh334)

Madleen Willmann (mw229)

1. Kurzbeschreibung

Bei unserem SE2-Projekt haben wir uns für ein Organizing Tool entschieden. Es soll dem User helfen, sein Leben einfach und besser zu organisieren. Dafür kann er sich zum Beispiel eine To-Do-Liste anlegen, mit der er keine Fristen mehr aus dem Auge verlieren soll. Die To-Do-List-Einträge kann er sich dann in seinem persönlichen Kalender anzeigen lassen.

Eine weitere Funktion des Organizing Tools ist die Notiz-Funktion. Der Benutzer kann Notizen erstellen und bearbeiten und falls er eine Notiz im Moment nicht braucht, kann er sie ganz einfach archivieren und später wieder aus dem Archiv reaktivieren. Dasselbe gilt auch für die einzelnen To-Do-List-Einträge. Außerdem kann er die Notes und die Tasks ganz einfach mit der Suchfunktion durchsuchen oder nach gewünschten Parametern sortieren. Zudem können die Notes und Tasks als Backup in eine .txt-Datei exportiert werden.

2. Startklasse

Unsere Startklasse ist Main.class im mainpackage.

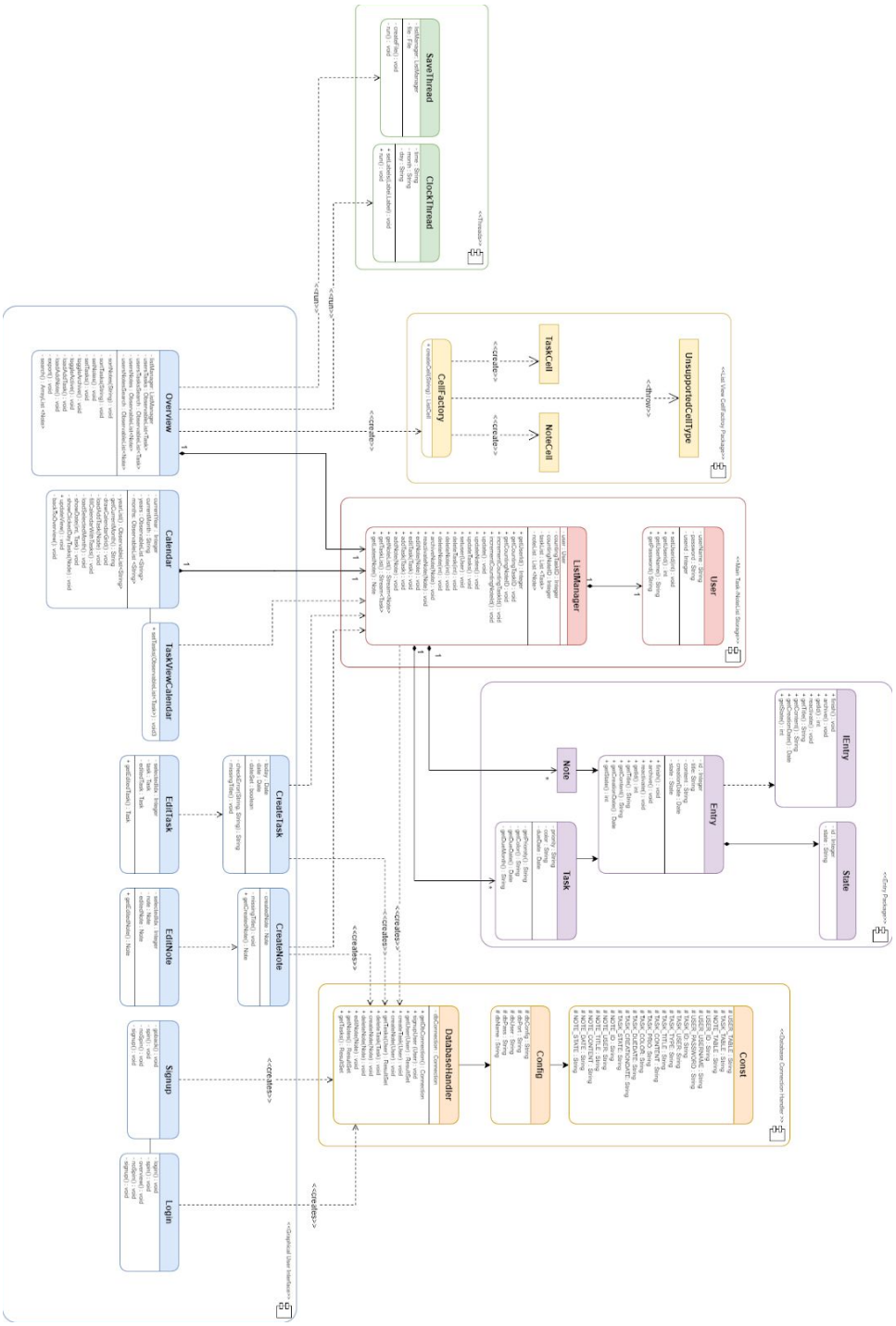
3. Besonderheiten

- Ausführbare Jar-Datei ohne aufwändige Einrichtung.
- Datenbankanbindung über JDBC, wodurch der User einen Account erstellen kann und folglich alle seine Einträge in der Cloud gespeichert werden. (Zudem Verwendung des Data Access Object Design Pattern)
- Task und Notes können exportiert werden
- Das von JavaFX zur verfügung gestellte Runnable >Task< wird für GUI Updates verwendet
- Zusätzliche Kalender Ansicht bietet die Möglichkeit besseren Überblick zu erhalten, unterstützt zudem die Möglichkeit per Rechtsklick auf einen Tag dort einen Task zu erstellen.
- Läuft unter Java 11 und JavaFX 11, und benötigt entsprechend mindestens Java 11 installiert um ohne Fehlermeldung zu starten.
- JavaDoc unter https://hfrey.de/orgadoc/apidocs/index.html

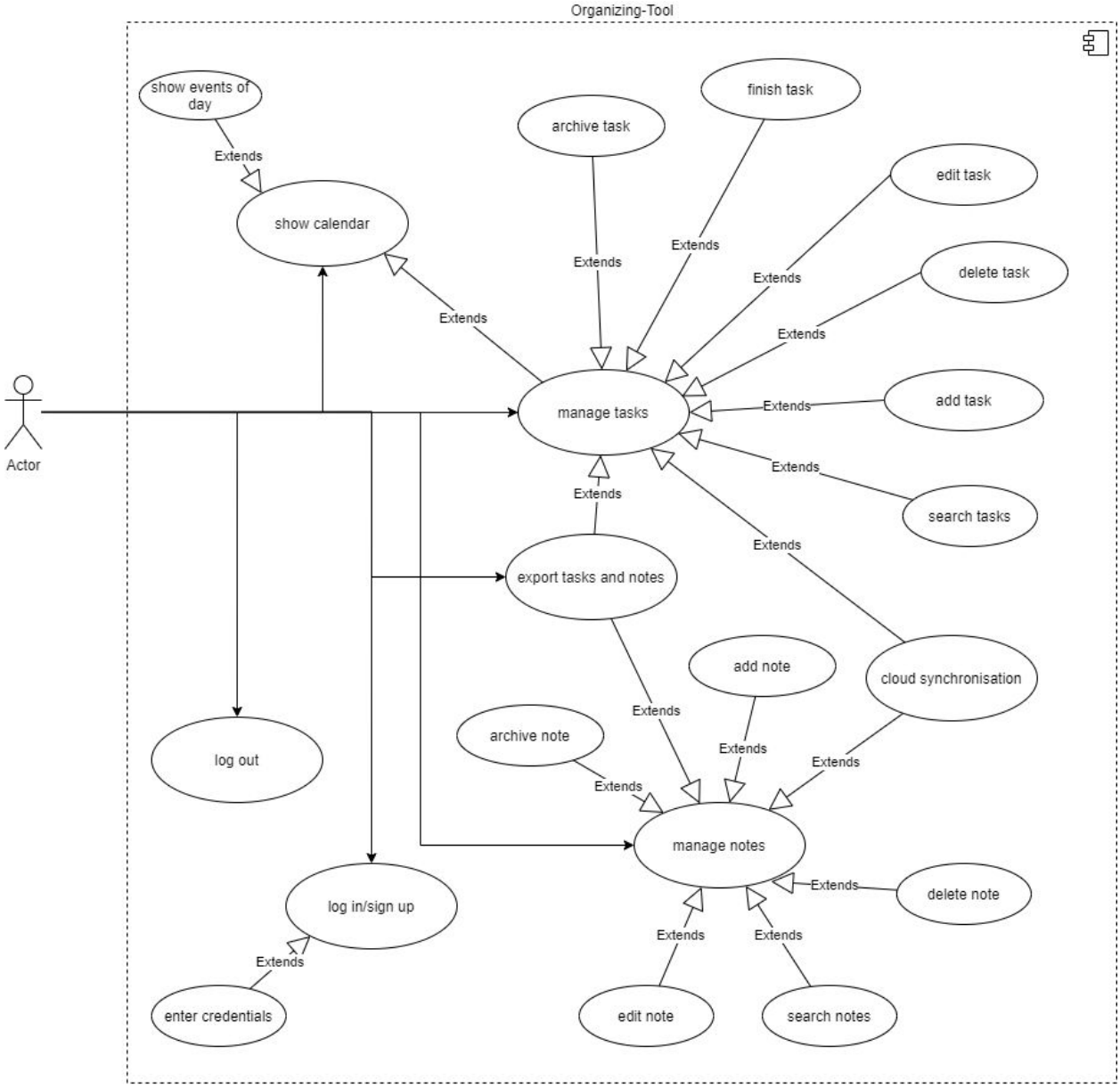
4. UML-Diagramme

https://gitlab.mi.hdm-stuttgart.de/mh334/organizing-tool/-/tree/master/src/main/resources/UML

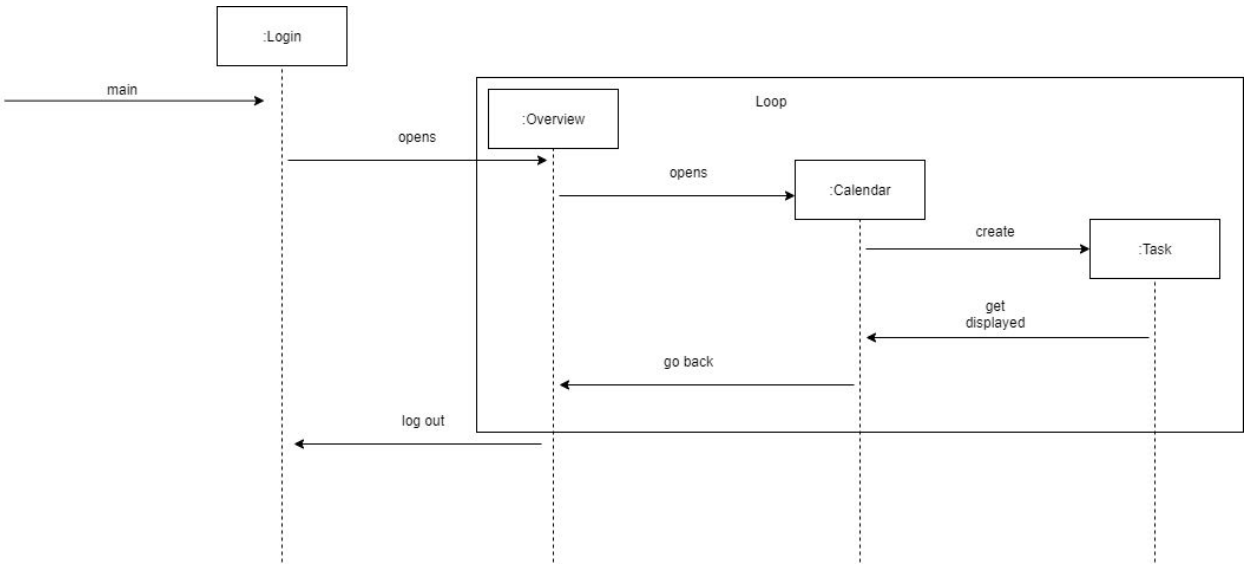
4.1 Klassendiagramm

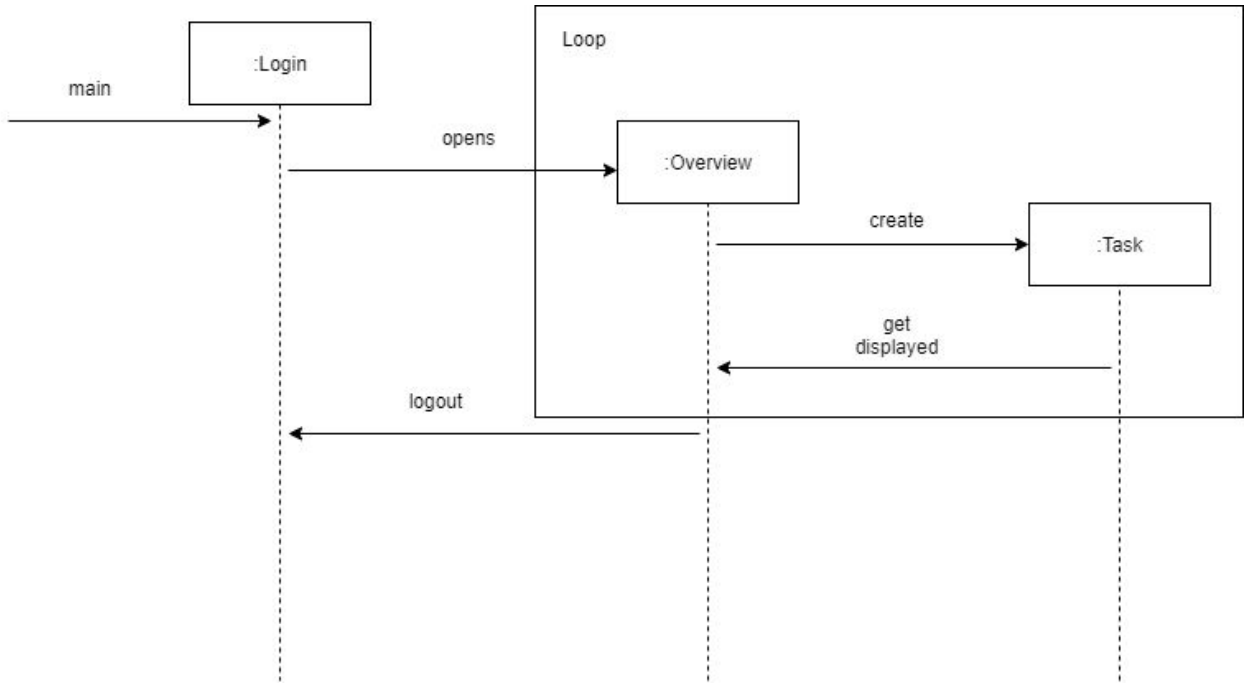


4.2 Use-Case-Diagramm



4.3 Sequenzdiagramme





5. Stellungnahme

I. Architektur

In unserem Projekt haben wir ein Interface, eine abstrakte Klasse und eine Factory verbaut. Diese drehen sich um das Objekt eines 'Eintrags' und werden derzeit als Notiz und als Task implementiert. Weitere Implementationen für die Zukunft würden beispielsweise Bilder oder handschriftliche Notizen darstellen.

- **Interface** : IEntry, wird von der abstrakten Klasse Entry implementiert
- **Abstrakte Klasse**: Entry, implementiert IEntry und vererbt an Task und Note
- **Factory**: CellFactory, erstellt Zellen für die ListView, kann eine Zelle für Task und eine für Note zurückgeben
- **Vererbung:** Viele Klassen erben von Java- oder JavaFX-Klassen, wie beispielsweise die NoteCell und TaskCell von javafx.scene.control.ListCell. Zudem erbt der DatabaseHandler seine Configurations-Konstanten und Taks und Note erben von Entry.
- **Package-Struktur:** Sehr sinnvolle Aufteilung der Klassen in Packages ermögliche einen schnellen Überblick.

II. Clean Code

In unserer Anwendung wurden in keiner Klasse public Members verwendet. Zudem haben wir nur in drei Klassen (ListManager, EditTask und EditNote) statische Methoden verwendet. Auf der linken Seite einer Objektzuweisung stehen immer die Interfaces des Objekts, und in Gettern werden nur Kopien eines Objekts zurückgegeben oder ein Stream gestartet.

III. Tests

Die Controller Klassen, für die Tests möglich waren, wurde mit Hilfe einer Dependency zum Ausführen von Tests auf JavaFX Klassen ausführlich getestet. Leider bot sich das nur für wenige der Controller an, da diese jedoch hauptsächlich unsere testbaren Klassen implementieren, sind die Tests dieser umso wichtiger.
Die Tests sind im Ordner tests zu finden und in selber Package Struktur angelegt wie das mainpackage, um Verständlichkeit zu gewährleisten.

IV. GUI (JavaFX)

Beim GUI haben wir mehrere Screens. Hier sind implementiert:

1. Buttons, die neue Fenster öffnen
2. Buttons, die Alerts auslösen
3. ein Fenster, das sich im schon geöffneten Fenster öffnet
4. ein gesplitteter Screen

Die FXML-Dateien befinden sich in resources/view. Die zugehörigen Klassen bzw. Controller befinden sich im Controller Package.

V. Logging/Exceptions

Es wurden alle Log-Levels bis auf die unterste Stufe TRACE verwendet. Exceptions werden in allen Klassen mit dem Level ERROR geloggt und alle Threads sowie deren ID werden geloggt. Es wird außerdem eine Log-Datei im Verzeichnis erstellt, in dem die ausführbare Jar liegt.

VI. UML

Unsere selbst erstellten UML-Diagramme befinden sich in resources/UML. Wir haben ein Klassendiagramm, ein Use-Case-Diagramm und zwei Sequenzdiagramme erstellt. Im Use-Case-Diagramm sind die wichtigsten Kern-Fälle enthalten.

VII. Threads

Das Programm verwendet an vielen Stellen Threads. Zu den Standard-Threads würden wir den ClockThread zählen, der im Hintergrund die Uhrzeit-Label anpasst, und den SaveThread, der in einem neuen Thread die vorhandenen Einträge eines Users in eine Text Datei exportiert.
Zusätzlich dazu existieren für alle Datenbankabfragen eigene Threads, da diese das GUI nicht responsive machen würden. Diese Threads werden mit dem Task aus dem javafx.concurrent Package ausgeführt. Da hier zwar nicht auf gemeinsam zugreifbare Members zugegriffen wird, wäre ein Standard Java Thread naheliegend gewesen, allerdings bietet der javafx Task einige vorteile die die Architektur verbesserten und Clean Code zugute kamen.
Für das Verarbeiten eines parallelen Streams jedoch wurde dieser Task voll ausgereizt und garantiert so sicheren Zugriff auf Members, auf die auch der JavaFX Thread zugreift.

VIII. Streams und Lambda-Funktionen

Die ListManager Klasse streamt alle Notes und Tasks als parallelen Stream und bietet so Performance-Vorteile gegenüber eines normalen Streams. Den synchronized Zugriff auf die zu empfangende List wurde gewährleistet mit dem entsprechenden Objekt in der Overview und Calendar Klasse.
In der Overview Klasse wird der Stream zudem sortiert und gefiltert, zwischen archivierten Entries und aktiven.

IX. Factories

Um die Erweiterung unserer Anwendung zu garantieren, haben wir zum Einen wie oben bereits beschrieben Interfaces und abstrakte Klassen. Da die ListView in der Overview Klasse jedoch eigene Zellen implementiert, war die Nutzung einer Factory für die Erstellung der Cells naheliegend und lässt sich einfach um Zellen expandieren.

6. Bewertungsbogen

https://gitlab.mi.hdm-stuttgart.de/mh334/organizing-tool/-/blob/master/src/main/resources/Bewertungsbogen.xlsx

Bewertungspunkt	Beschreibung	Punkte
Architektur	Erweiterbare Architektur mit Interfaces und Factories	3
Clean Code	Strong Encapsulation, Interfaces auf linker Seite von Zuweisungen	3
Dokumentation	Vollständige und ausführliche Dokumentation	3
Tests	Alle wichtigen Klassen mit Positiv- und Negativ-Tests	3
GUI	Komplexes und verschachteltes GUI	3
Logging/ Exceptions	Sinnvolle Logging-Struktur und korrekter Umgang mit Exceptions	3
UML Diagramme	Übersichtliches Klassen- und Use-Case-Diagramm, das dem Verständnis dient.	3
Threads	Viele Threads, über den Java Standard Thread hinaus, für Hintergrundaufgaben, die synchronisiert arbeiten.	3
Streams	Parallele Streams für schnelleres Laden der Einträge, werden synchron in eigenen Threads verarbeitet.	3
Nachdenkzettel	Alle Nachdenkzettel sind vollständig bearbeitet.	3

7. Nachdenkzettel

https://gitlab.mi.hdm-stuttgart.de/mh334/organizing-tool/-/tree/master/src/main/resources/Nachdenkzettel