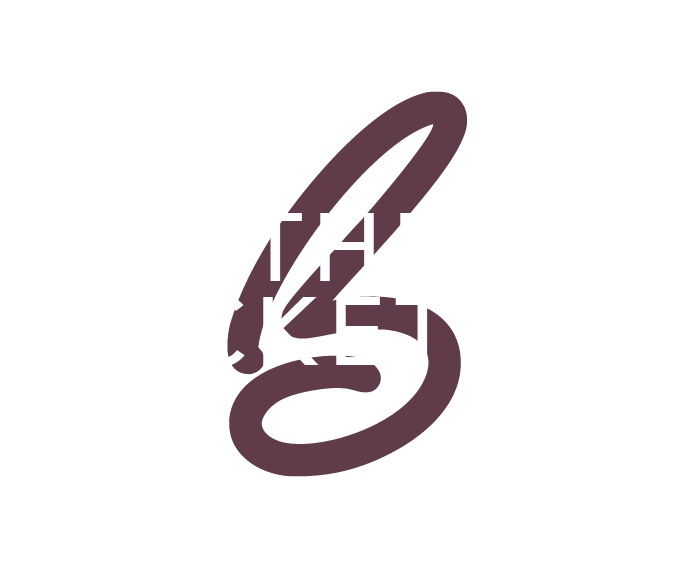
The Bucketlist

Software Entwicklung 2, Sommersemester 2021

<https://gitlab.mi.hdm-stuttgart.de/st093/bucketlist>





**Pia Schilling,** ps149@hdm-stuttgart.de

**Sara Tietze,** st093@hdm-stuttgart.de

**Merve Özdemir,** mo064@hdm-stuttgart.de

1. Kurzbeschreibung des Projekts

Unser Projekt ist, wie der Name schon verrät eine **Bucketlist** (zu Deutsch: Wunschliste).

Wenn man das Programm startet, kann man zuallererst eine **Eventliste** mit Ablaufdatum erstellen. Diese wird dann im nächsten Schritt mit **Events** gefüllt (z.B. Spanisch lernen, nach Mexico reisen, …) die man ebenfalls selbst anlegt. Man kann hier Titel und Kategorie auswählen. Wobei diese beiden Pflichtangaben sind.

Es gibt 13 verschiedene **Kategorien**: Skills, Travel, Culture, Shopping, Lifegoals, Culinary, Education, Sport, Hobby, Family, Relationship, Friends und No Category speziell für Events, die zu keiner der anderen Kategorien zugeordnet werden können.

Sobald man ein Event abgeschlossen hat, kann man dieses dementsprechend markieren. Es kann eine **abschließende Beschreibung** und ein **Bild** hinzugefügt werden.

In einer eigenen Scene kann man sich die **Events je Kategorie**, als eigene Liste anzeigen lassen.

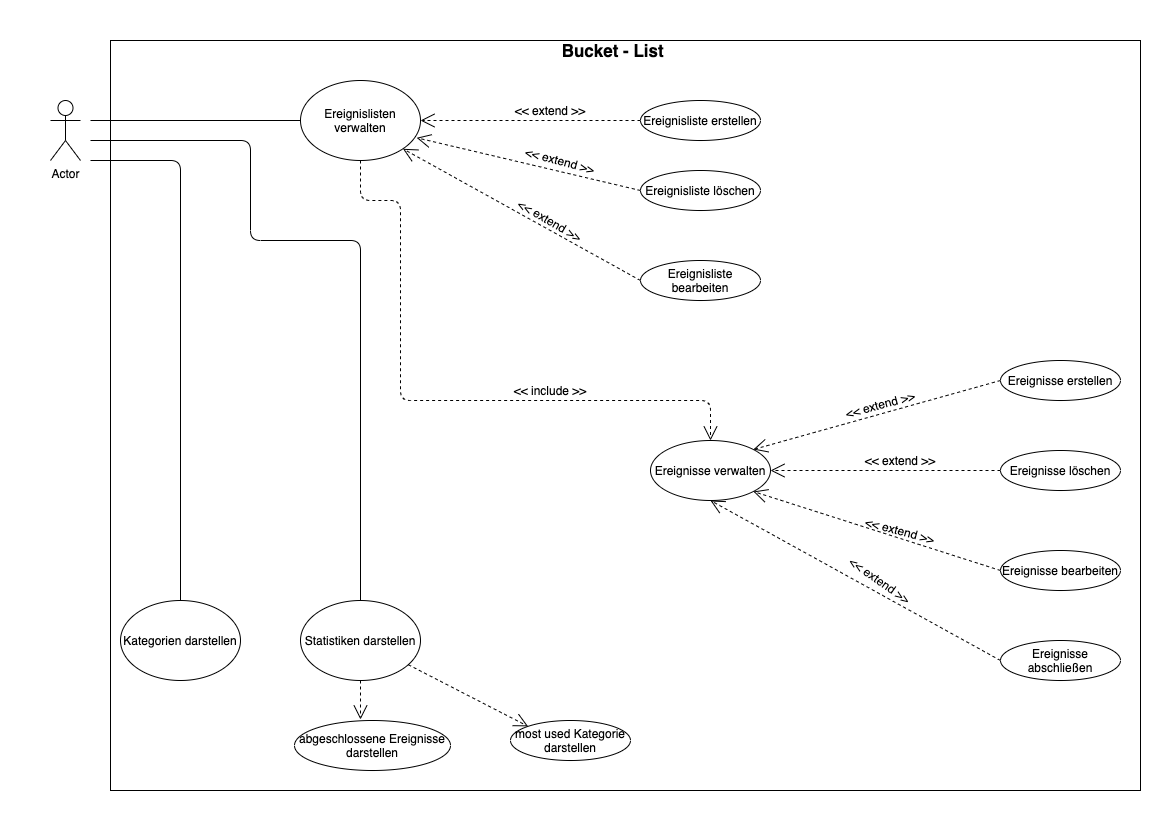
Außerdem kann man separat **Statistiken** aufrufen. Hier wird gezeigt wie viele Events angelegt oder abgeschlossen wurden, die Anzahl der angelegten Eventlisten, der Fortschritt jeder einzelnen Liste und wie viele Tage Zeit man noch hat die Liste abzuschließen. Zudem kann man sich in einer übersichtlichen **PieChart** anzeigen lassen, wie viele Events in jeder Kategorie erstellt wurden.

Die Main-Methode befindet sich in der Klasse **Main.**

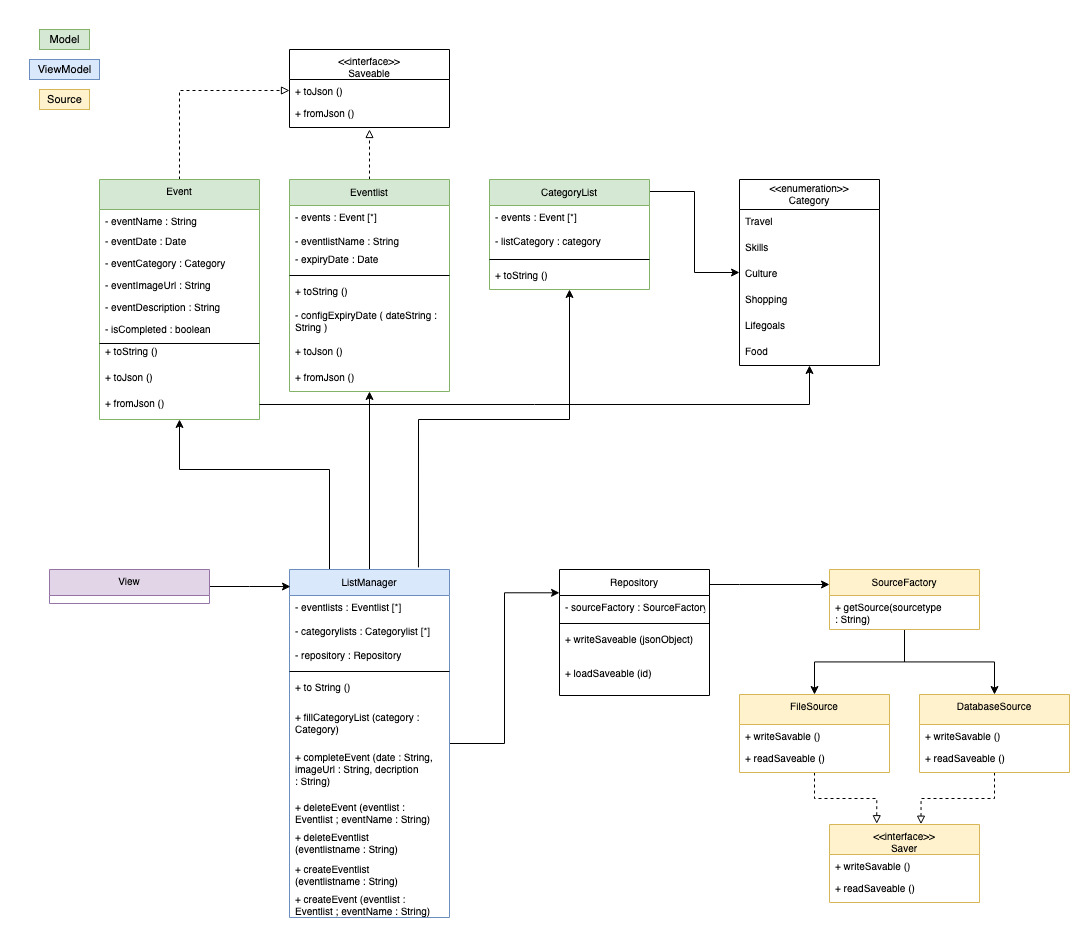
1. UML

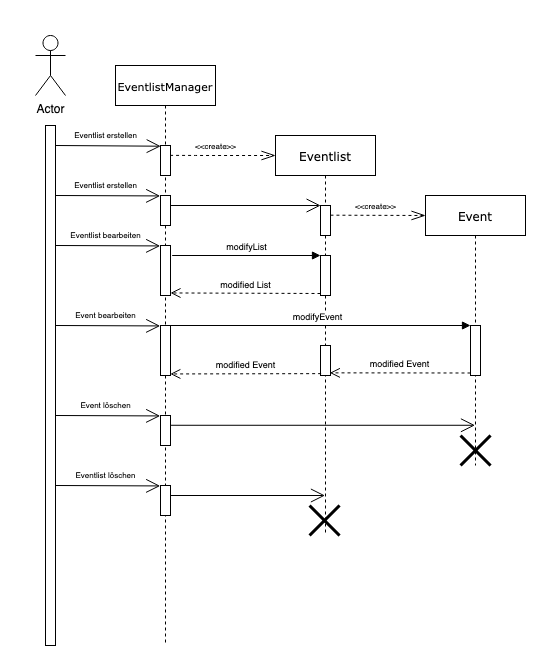
Die Diagramme befinden sich im Git-Repository.

**Use Case Diagram**

****

**Class Diagram**

****

**Sequence Diagram**

1. Stellungnahmen

Architektur

**Singelton Access: (**Dependency Injection anstatt Singleton)

Bei der Klasse ListManager hätte es sich angeboten, das Singleton-Pattern zu verwenden. In der Anwendung soll es nur eine Instanz der Klasse ListManager geben und alle Klassen sollten mit derselben Instanz arbeiten. Aufgrund verschiedener Faktoren haben wir uns jedoch dagegen entschieden. Singletons sind schwer zu testen, global zugänglich und man verliert schnell den Überblick, welche Klasse mit dem Singleton arbeitet und dieses vielleicht sogar modifizieren. Daher die Entscheidung für dependency injeciton. In der Main-Klasse wird die Instanz des ListManagers erstellt und ab da an alle Klassen „weiter gereicht“, die diese benötigen. Es entstand ein recht komplexes und teilweise unübersichtliches „hin und her gereiche“ des ListManagers. Für uns ist es schwer einzuschätzen, ob es in unserem Anwendungsfall nicht doch besser gewesen wäre, ein Singleton zu verwenden. Dieses hätte zumindest garantiert, dass es sicher nur eine Instanz des ListManagers gibt, was momentan nicht der Fall ist. Außerdem wäre der Zugriff auf ein Singleton einfacher als ständig den ListManager zu „injizieren“.

**Interface:** Wir haben mehrere Interfaces erstellt, zum Beispiel das Listener Interface, das dafür sorgt, dass alle Controller Klassen darüber informiert werden, wenn ein Model-Zustand sich ändert oder das Narrator Interface, welches dafür sorgt, dass alle Klassen, die dieses implementiert haben von GUI Klassen observiert werden können.

Das Saveable-Interface muss von allen Klassen, deren Objekte persistent gespeichert werden sollen, implementiert werden.

Das Saver-Interface muss von allen Source-Klassen implementiert werden. (FileSource und DatabaseSource)

**Factory: (SourceFactory)** Ist ein Hilfsmittel, um in unserem Fall die Datenquelle zu bestimmen. Die Methode getSource liefert die passende Quelle Filesource oder Databasesource zurück.

Jedoch wäre die Wahl einer Datenbank als Quelle im Moment nicht wirklich möglich. Saveables müssen die Methoden fromJson und toJson implementieren. Die toJson-Methode erwartet einen File als Parameter, d.h. das Speichern in eine Datenbank wäre nicht möglich ohne eine Umstrukturierung dieser Methode. (Dazu kommt noch, dass so nur Json in eine Datenbank gespeichert werden könnte, was ebenfalls fraglich ist) Kurzum: die SourceFactory ist nur eine „PseudoFactory“, die nicht wirklich Sinn macht. Um dies zu beheben müsste die Persistience so geändert werden, dass in den Modelklassen keine Quellen-Spezifischen Methoden zur Speicherung der Objekte implementiert werden.

**Enum:** Unsere Enums beinhalten die verschiedenen Event-Kategorien, die man zu jedem Event zuordnen muss und die Sourcetype der Daten. Dadurch wird eine eindeutige Wahl gewährleistet.

**Vererbung:** Alle Klassen im Unterordner Customnodes von View/Controller erben von der Klasse Anchorpane.

**Ordnerstruktur:**

Im Ordner de.hdmstuttgart.mi.bucketlist befinden sich neben der Main-Klasse, noch sechs Unterordner.

**Exceptions:** beinhaltet unsere eigenen Exceptionklassen

**Model:** beinhaltet das Enum Category, das Interface Narrator und andere Klassen die, die Attribute von Events und Eventlisten beinhalten

**ModelController:** beinhaltet die Controller Klassen, die die Eventlisten und Events steuern

**Persistence:** beinhaltet Klassen und Interfaces zur persistenten Datenspeicherung

**Threads:** beinhaltet unsere Thread Klasse

**View:** beinhaltet die Controller Klassen, die die fxml Dateien steuern und das Interface Listener

Clean Code

In den Klassen wird nirgends ein public member verwendet.

Tests

Wir haben im Ordner (test > java > de.hdmstuttgart.mi.bucketlist) mit **JUnit Tests** **Testklassen** erstellt, die die wichtigsten Methoden aus den Klassen auf ihre Funktion testen. Es gibt für eine bessere Übersicht auch hier Unterordner (Model, ModelController, Persistence).

GUI (JavaFX)

Die GUI haben wir mit dem **SceneBuilder** erstellt. Die jeweiligen fxml Dateien befinden sich im Ordner (scr > main > resources > fxml). Außerdem befinden sich die zugehörigen Controller Klassen im Ordner (scr > main > java > de.hdmstuttgart.mi.bucketlist > View).

Logging/Exceptions

**log.debug** >> Allgemeine Informationen wie z.B. der Start einer Methode

**log.error** >> Wenn z.B. kein passendes Objekt gefunden wird

**log.info** >> Informationen über den Erfolg einer Funktion

**Exception:** Wir haben einmal die **ElementAlreadyExistsException**, die von der Klasse Eventlist ausgeführt und aufgerufen wird, wenn z.B. der Titel einer Eventliste bereits vergeben ist. Und wir haben die **EmptyDirectoryException**, die von der Klasse FileSource ausgeführt und aufgerufen wird, wenn ein Verzeichnis leer ist oder nicht existiert.

UML

Wir haben mit einem **Use Case Diagramm** angefangen. Dadurch wurde uns bewusst, welche grundlegenden Funktionen wir für unsere Idee brauchen. Diese haben wir dann im Nachhinein im **Klassendiagramm** ausgearbeitet und erweitert. Außerdem haben wir ein **Sequence Diagramm** erstellt mit möglichen User Szenarien, die uns nochmal gezeigt haben, welche Funktionen eventuell noch fehlen könnten. Als wir fast fertig mit unserer Software waren, haben wir die Diagramme nochmal überarbeitet.

Threads

**PersistenceRunnable** erstellt Threads, die überprüfen, ob die persistente Datenspeicherung lauffähig ist und funktioniert.

Service (CategoryController)

Streams

Factories

Unser Programm besitzt im Ordner (scr > main > java > de.hdmstuttgart.mi.bucketlist > Persistance) eine Klasse namens **SourceFactory**. In dieser befindet sich die Methode getSource, die überprüft welchen **Sourcetype** eine Datei hat und diesen ausgibt.

1. Bewertungsbogen
2. Profiling Analyse

Stattdessen Nachdenkzettel vom Team gemeinsam ausgefüllt im Git-Repository.

**Vermischung von zwei Arten der GUI Kommunikation**

Da die Logik der Anwendung komplett unabhängig vom GUI steht, brauchten wir eine Möglichkeit, das GUI über Änderungen von Model-States zu informieren, damit sich dieses anpassen und die Änderungen für den User sichtbar machen kann.  
Da das Observable-Interface von Java als „deprecated“, also veraltet gilt, haben wir zwei eigene Interfaces erstellt.   
Zum einen das Narrator (Erzähler) Interface, welches an die Stelle des Observables tritt und von allen Klassen, die beobachtbar sein sollen, implementiert werden muss. Dieses liefert zwei Methoden, addListener und informListener, um Zuhörer anzumelden und diese dann bei Änderungen zu informieren.Zum anderen das Listener (Zuhörer) Interface, welches von allen Klassen, die über Änderungen informiert werden sollen implementiert werden muss. Dieses liefert lediglich die Methode update. Die update-Methode wird bzw. wurde von allen Klassen so verwendet, dass sich die komplette Scene neu aufbaut (GUI Elemente werden gelöscht und neu generiert), sobald die Methode durch einen Narrator (durch infromListeners) aufgerufen wurde. So sind die Änderungen der Model-Klassen nach dem Neuaufbau der Scene auch im GUI sichtbar. Aufgrund der kleinen Anzahl der Elemente funktioniert dies auch Problemlos. Trotzdem ist das ein eher unschöner und ressourcenverschwendender Ansatz. Dieser Ansatz machte uns dann auch Probleme sobald ein weiterer Thread ins Spiel kam. Der Persistence-Thread rief indirekt die update-Methode auf. Anderen Threads ist es jedoch nicht erlaubt das GUI zu manipulieren. Dieses Problem ließ sich durch den Einsatz von Platform.runLater recht einfach beheben, jedoch brachte das neue Probleme mit sich, was schließlich zu einer kompletten Umstrukturierung der GUI-Klasse ListsController führte. In dieser Klasse wird das Listener Interface nun nicht mehr benutzt. Stattdessen „hört“ das GUI einer Observable-Liste zu und aktualisiert sich selbst bei Änderungen (onChanged-Methode) Es wäre besser überall mit diesem Ansatz (also mit Properties bzw. Observable-Listen) zu arbeiten. So kann das GUI sich selbstständig updaten und es ist nicht bei jeder Änderung ein kompletter Neuaufbau der Scene nötig. Jedoch wäre hier eine Umstrukturierung der ganzen Persistence nötig, da mit der aktuellen Implementierung keine Properites serialisiert werden können. Aufgrund der Zeit war dies leider nicht mehr möglich. Die Zeit ist auch der Grund, weshalb andere GUI-Klassen, wie der EventlistController, immer noch mit der update-Methode arbeiten um das das GUI an Änderungen anzupassen.  
Diese Vermischung, der zwei Ansätze der Kommunikation zwischen GUI und Logik, ist sehr unschön und sollte auf jeden Fall noch geändert werden.

**Unnötige Methoden**

Im ListManager befinden sich einige Methoden (completeEvent, deleteEvent, addEventToList), die eigentlich nicht gebraucht werden, da die Aufgaben dieser Methoden inzwischen von der Klasse Eventlist übernommen werden. Leider haben wir es versäumt, diese Methoden frühzeitig zu löschen, weshalb diese nun in einigen Junit-Test (zum Vorbereiten anderer Tests) verwendet werden und eine Löschung nicht mehr möglich ist.