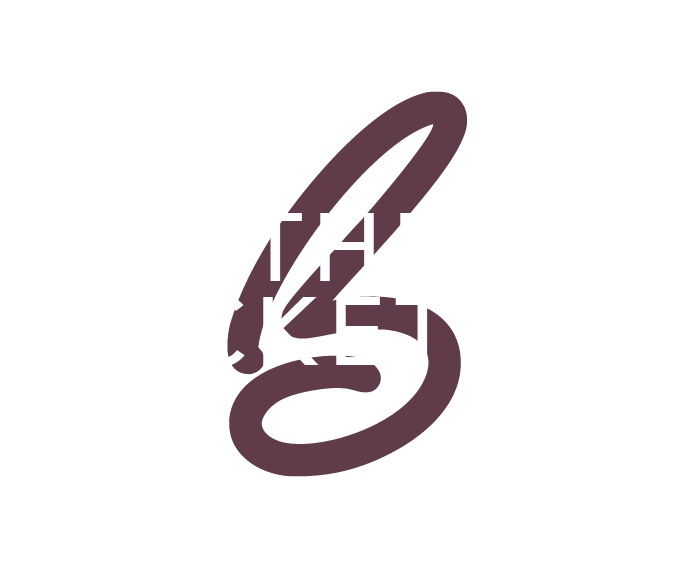
The Bucketlist

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. Demo-Video

Um einen schnellen Überblick über das Programm zu bekommen, empfehlen wir, als erstes das kurze Video anzuschauen. Es erklärt wozu das Programm gedacht ist und demonstriert alle wichtigen Funktionen.

**<LINK>**

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.

**Start des Programms**

Das Programm lässt sich über die Main-Methode starten. Diese befindet sich in der Klasse **Main** direkt im Package de.hdmstuttgart.mi.bucketlist.

**Besonderheiten**

Alle Teile des Programms funktionieren. Nur bei dem Wechsel in die CategoryScene ist leider mit einer Wartezeit von über 4 Sekunden zu rechnen.

**Ordnerstruktur:**

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

**Exceptions:** beinhaltet unsere eigenen Exception-Klassen

**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

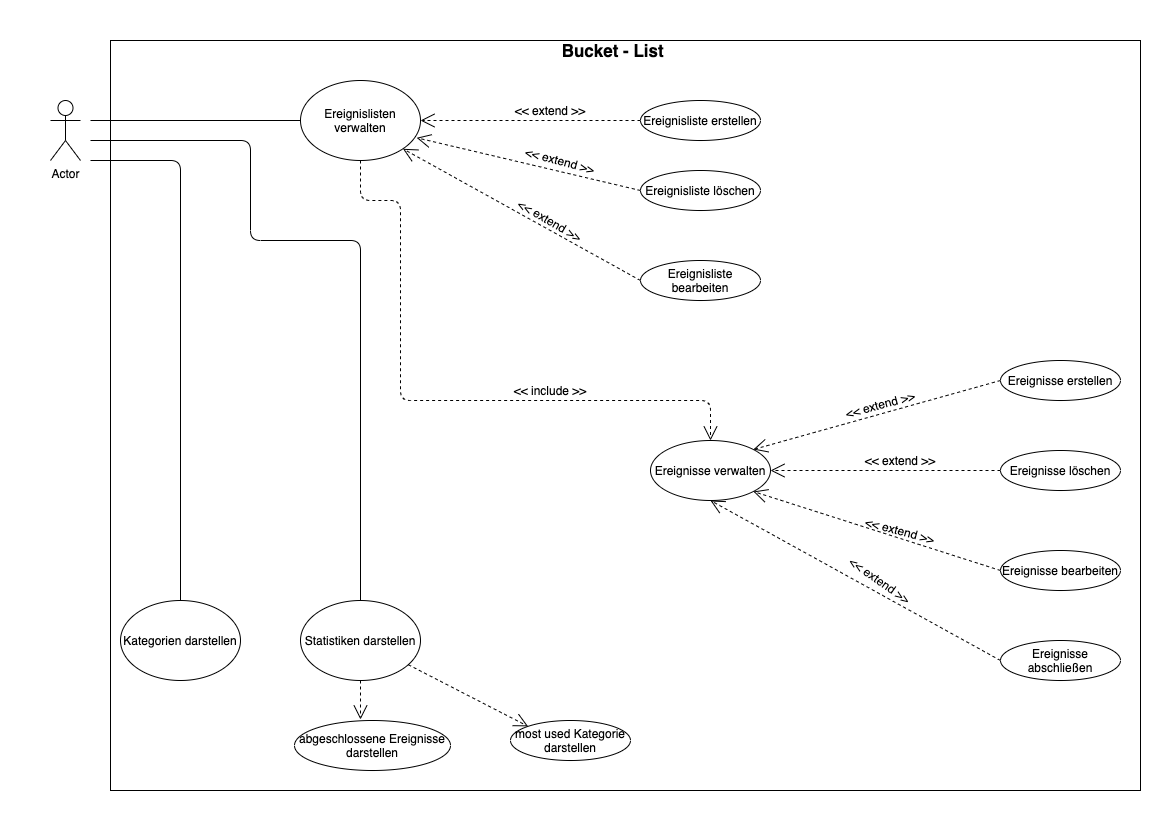
**Threads:** beinhaltet unsere Runnable Klasse

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

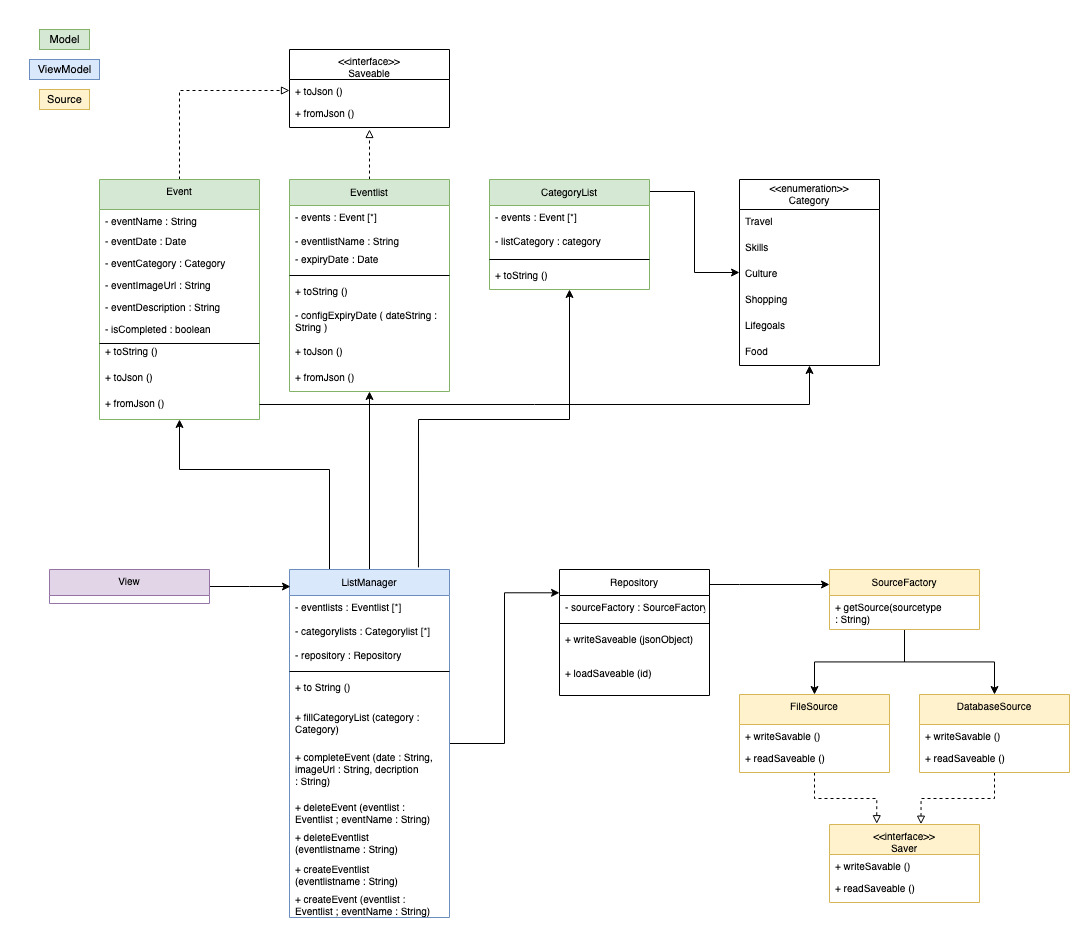
1. UML

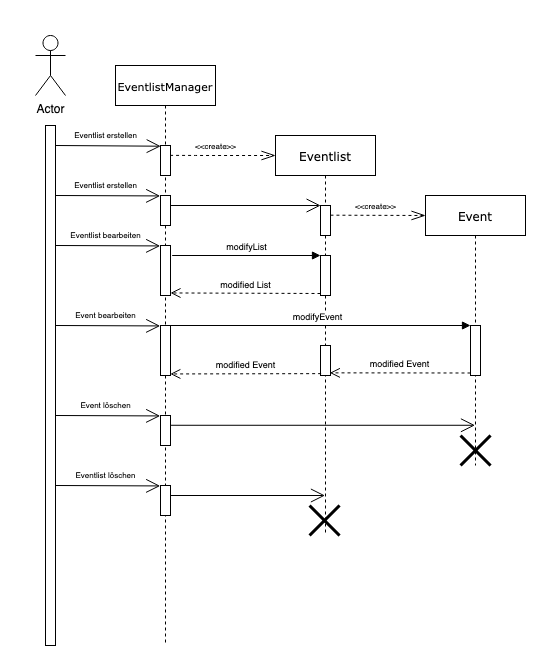
Die Diagramme befinden sich zusätzlich nochmals im Git-Repository. Dort sind auch ältere Versionen der Diagramme zu finden. So kann unser Entwicklungsprozess besser nachempfunden werden.

**Use Case Diagram**

****

**Class Diagram**

****

**Sequence Diagram**

1. Stellungnahmen

Architektur

**Singleton Access: (**Dependency injection)

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 Klassen mit dem Singleton arbeiten 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 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 weiter zu geben**.**

Auch andere Objekte, wie das BorderPane des MenuControllers werden teilweise sehr unschön durch mehrere Klassen durchgereicht, damit eine bestimmte Klasse Zugriff darauf hat. Die „zwischen“ Klassen haben dabei keine Verwendung für das Objekt, sondern dienen nur als „Überträger“. Dieses Vorgehen sollte eventuell nochmals überdacht werden.

**Interfaces:** Wir haben mehrere Interfaces erstellt.

Das Listener Interface, das dafür sorgt, dass alle Controller Klassen darüber informiert werden, wenn sich ein Model-Zustand ändert und das Narrator Interface, welches dafür sorgt, dass alle Klassen, die dieses implementiert haben von GUI Klassen observiert werden können. (darauf und auf die damit verbundenen Probleme wird später nochmals genauer eingegangen)

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)

Das Box Interface wird von einigen CustomNodes Klassen implementiert um von Polymorphismus Gebrauch machen zu können.

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

**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 im Package View erben von der Klasse Anchorpane um diesen Nodes die Eigenschaften eines Anchorpanes zukommen zu lassen, sie aber bereits mit einem eigenen Layout „füllen“ zu können.

Clean Code

In den Klassen wird nirgends ein public member verwendet.

Nur im StatisticsManager ist eine statische Methode (daysLeft) zu finden.

An einigen Stellen konnte jedoch nicht auf Getter verzichtet werden, da diese Felder im GUI dargestellt werden sollten und wir dafür keinen anderen Weg als Getter gesehen haben. Jedoch beziehen sich die meisten getter auf String Varaiblen oder primitive Datentypen. Strings sind immutable und bei primitiven Datentypen wird sowieso nicht mir Referenzen gearbeitet, deshalb stellen diese Getter kein Problem dar.

In Eventlist und im ListManager gibt es jeweils einen Getter für die ArrayListe aus Objekten. Zwar wird hier nur eine Kopie der Liste herausgegeben, jedoch sind die Objekte in der Liste dieselben. Da eine deep-copy doch recht aufwendig gewesen wäre, haben wir uns dagegen entschieden, da wir keine wirkliche Notwendigkeit dafür gesehen haben, da sowieso überall nur lesend auf die herausgegebene Liste zugegriffen wird. (Natürlich wäre eine echte Kopie besser, da nie garantiert ist, dass die Liste bzw. die Objekte der Liste nicht doch durch eine Klasse bearbeitet werden der es eigentlich nicht erlaubt ist)

Ebenfalls in Eventlist und ListManager gibt es jeweils eine Methode die get…ByName heißt. Hier wird tatsächlich die Originalreferenz auf das jeweilige Objekt herausgegeben. Uns ist kein anderer Weg eingefallen wie das GUI sonst mit den Objekten arbeiten hätte können. Hierbei handelt es sich vermutlich um ein Architekturproblem.

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).

Wir haben eigene Nodes erstellt, um das GUI an unsere Anforderungen anzupassen. Diese sind im Package CustomNodes zu finden. Zu jeder dieser Klasse gibt es einen CustomNodesController, der die Funktionalität zu der Node bereitstellt. Die Nodes werden von den SceneControllern verwaltet. In diesen Scenes ist es auch möglich weitere Nodes zu erstellen. Des Weiteren gibt es einige PopUp-Fenster. Die Klassen dazu sind im Package PupUpController zu finden.

Logging

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

**log.info** >> Informationen über den Erfolg/Misserfolg einer Funktion/Methode oder Exceptions, die aber den Programmlauf nicht beeinflussen

**log.error** >> Messages von schwerwiegenderen Exceptions

Exceptions

Wir haben einmal die **ElementAlreadyExistsException**, die von den Klassen Eventlist und ListManager in einigen Methoden geworfen wird, wenn z.B. der Titel einer Eventliste bereits vergeben ist. Diese Exception wird erst im GUI abgefangen und die Exception-Message wird für den User dargestellt.

Und wir haben die **EmptyDirectoryException**, die in der Klasse FileSource von der Methode listDirecotry geworfen wird, wenn ein Verzeichnis leer ist oder nicht existiert. Diese sorgt dafür, dass das Programm weiter läuft auch wenn das Directory leer ist oder nicht existiert. (Im zweiten Fall wird das Direcotry „Data“ erstellt).

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

Das **PersistenceRunnable** ist für das Laden der Daten aus den Files zuständig. Der Thread ist so gestaltet, dass das GUI unabhängig von dem Thread lädt, es also schon responisve ist, während der Persistence-Thread noch arbeitet.

Der **LoaderService** ist in der Klasse CategoryController zu finden. Da das Laden der CategoryBoxen leider sehr lange dauert, wird diese Aufgabe von einem JavaFx-Service übernommen. So bleibt das GUI responsive und der User kann über den Ladefortschritt informiert werden.

Streams

In vielen Klassen wird mit **Streams** gearbeitet um Listen zu filtern oder Listen auf die Existenz auf Objekte mit bestimmten Eigenschaften zu prüfen.

Im ListManager ist in der Methode createEventlist außerdem ein **parallelStream** zu finden.

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 je nach Parameter ein Objekt der gewünschten Quelle zum persistenten Speichern von Objekten zurückgibt.

1. Bewertungsbogen

**<Bewertungsbogen>**

1. Profiling Analyse

Stattdessen Nachdenkzettel vom Team gemeinsam ausgefüllt im Git-Repository.

**Zusatz: Probleme und Verbesserungsmöglichkeiten**

**Listener Interfaces und Properties**

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 (mit Hilfe der Methode 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 Properties mit der aktuellen Implementierung nicht ohne weiteres 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.

Leider wurde deshalb unser Projekt sehr unübersichtlich: manchmal werden Properties verwendet, manchmal nicht. Jede GUI-Scene arbeitet unterschiedlich, es fehlt einfach „der rote Faden“ bzw. die Konsistenz.

**.**

**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.

**SourceFactory nicht realistisch**

Wie wir bereits erwähnt haben, existiert eine SourceFactory, die je nach Parameter eine andere Quelle zum Speichern der Objekte liefert. (Datenbank oder File). 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 könnte im Moment nicht wirklich realistisch/sinnvoll genutzt werden. Es wäre eine Umstrukturierung der Persistence notwendig um dies zu erreichen.