Ostfalia - Hochschule für angewandte Wissenschaften
Fachbereich Informatik
Studiengang Informatik
Vertiefungsrichtung Medieninformatik

Implementierung eines Anwendungsfalles und Usability-Verbesserungen in einer JavaFX-Anwendung

Praxisprojektbericht

Fakultät Informatik der Ostfalia
- Hochschule für angewandte Wissenschaften

eingereicht bei	Prof. Dr. Bernd Müller
von	Jan Philipp Wiegmann Am Exer 2 38302 Wolfenbüttel Mat.-Nr. 70312216

Wolfenbüttel, den 9. Oktober 2015

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere, dass ich alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe, und dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist.

Ort, Datun Unterschrift

Abstract

This essay deals with enhancing an application in matters of new functionalities and usability. To achieve this, Code-Design Patterns are utilized and explained to the reader. Neuropsychological researches are the foundation for gestaltism, which is the base tool to analyze existing usability problems and to provide solutions for solving them. All of the implementation work is accomplished using the fairly new Java 8 linguistic devices (e.g. lambdas) and its brand new API.

Inhaltsverzeichnis

1 Einleitung

1.1 Ziele der Arbeit

Das Ziel dieser Arbeit besteht darin, dass die Anwendung *FalkoFX* um einen Anwendungsfall erweitert wird. Dies beinhaltet die Konzeption und die Umsetzung der dafür notwendigen Teillösungen. Damit der Leser einen Eindruck von der Aufgabe und den damit verbundenen Teilproblemen erhält, werden Ausschnitte des bestehenden Konzeptes erläutert. Zur Implementierungen der erarbeiteten Entwürfe werden etablierte Konzepte (Design-Patterns) der Software-Architektur verwendet, die dabei helfen, den Code zu strukturieren und die Programmierarbeit zu erleichtern.

Des Weiteren wird auf einige Probleme in der Gebrauchstauglichkeit der Anwendung eingegangen. Nach einer Analyse dieser, müssen Lösungskonzepte erarbeitet und schlussendlich implementiert werden. Anhaltspunkte für eine sinnvolle Gestaltung bieten unter Anderem allgemeingültige Konzepte, die auf Forschungen der Neuropsychologie basieren.

1.2 Motivation

Viele bestehende Softwaresysteme sind darauf ausgelegt, zu funktionieren. Das klingt erst einmal nicht verkehrt, aber sollte ein Softwaresystem, das gebraucht wird, nicht auch gebrauchstauglich sein?

Aufgrund der immer weiter verbreiteten Anforderung der Nutzerfreundlichkeit an bestehende und neue Software, müssen, je nach Komplexität des Systems und dem Budget des Projektes, neue Anwendungen entwickelt werden, die auf dem erlangtem Fachwissen fußen. Teils sind dies komplette Neuentwicklungen der bestehenden Systeme, teils Programme mit eingeschränktem Funktionsumfang. In vielen modernen Entwicklungsprozessen kommen neuartige Technologien zum Einsatz, deren Möglichkeiten immer

umfangreicher werden. Diese Möglichkeiten zum Erstellen von Software zu verwenden ist nicht immer ein einfaches, aber dennoch ein sehr interessantes Unterfangen.

1.3 Aufbau der Arbeit

Im Rahmen der theoretischen Grundlagen wird zunächst die verwendete Technologie vorgestellt. Es wird ein Einblick in die Neuerungen der Version 8 der Programmiersprache Java gewährt. Dabei wird der Fokus auf das JavaFX-Toolkit gelegt.

Um die späteren Probleme und deren Lösungen erläutern zu können, wird die Anwendung grob vorgestellt und die Ziele, die mit der Entwicklung dieser Software verfolgt werden, abgegrenzt. In der Arbeit erwähnte Konzepte der Softwarearchitektur werden erklärt und ein Basiswissen für das sinnvolle Design von Benutzeroberflächen wird geschaffen.

Es werden die Teilprobleme der Implementierungsarbeit beschrieben und dem Leser mit Hilfe von Erklärungen, Aufgabenanalyse und Konzepterstellung nähergebracht. Dabei wird auf Schwierigkeiten und deren Lösungen in der Implementierung eingegangen.

Es folgt eine Analyse von ausgewählten Barrieren in der Bedienung der Anwendung, für die mögliche Lösungen erarbeitet und im Anschluss umgesetzt werden.

2 Grundlagen

2.1 Besonderheiten in Java 8

Mit JavaSE 8 führt Oracle interessante Neuerungen in die weit verbreitete Programmiersprache Java ein [Ull14, S. 35]. Die für dieses Projekt wichtigen Änderungen werden im Folgenden dargelegt.

2.1.1 Default Implementierungen

Eine kleine, aber dennoch einflussreiche Veränderung sind Default Implementierungen für Interfaces. Dies bedeutet, dass in Interfaces Methodenkörper ausprogrammiert werden können. Wenn eine Klasse ein solches Interface implementiert, müssen die Default-Methoden des Interfaces von der implementierenden Klasse nicht überschrieben werden. Eine solche Methode wird mit dem Schlüsselwort *default* gekennzeichnet. [Ull14, S. 45f.]

Erwähnenswert ist außerdem, dass nun auch statische Methoden in Interfaces implementiert werden können. [Ull14, S. 48f.]

Durch die Einführung von Default-Methoden können Programmierschnittstellen auf eine neue Weise definiert werden. Außerdem wird das Konstrukt der Functional Interfaces ermöglicht bzw. verbessert.

2.1.2 Functional Interfaces

Functional Interfaces sind ein neues Konstrukt der Java-Sprachdefinition. Ein Functional Interface definiert sich dadurch, dass bei einem Interface nur genau eine Methode durch die implementierende Klasse überschrieben und ausprogrammiert werden muss - also genau eine abstrakte Methode vorhanden ist. Functional Interfaces alleine haben

keine besondere Bedeutung, sind jedoch Voraussetzung für ein anderes, mächtiges Sprachwerkzeug in JavaSE 8 – die Lambda-Ausdrücke. Ein oft verwendetes Beispiel für ein Functional Interface ist das *Consumer*-Interface. [Ull14, S. 63f.]

2.1.3 Stream API

Die Stream API ist kein neues Sprachfeature der Java Version 8, aber eine umfangreiche Programmierschnittstelle. Sie bietet Methoden, die auf sogenannten Streams arbeiten. Diese machen es möglich, implizit über Datenstrukturen zu iterieren, ohne dafür ein Schleifenkonstrukt (explizite Iteration) ausprogrammieren zu müssen. Streams sind mithilfe von Generics typisiert, sodass immer klar ist, auf welcher Art von Daten gearbeitet wird. [Ull14, S. 391f.]

Ein Stream kann aus einer beliebigen Java-Collection erzeugt werden, wie z.B. aus einer *ArrayList*. Dafür bietet das *Collection*-Interface die neue Methode *Collection#stream()* an, die einen Stream zurückliefert, der die aktuellen Elemente der zugrunde liegenden Datenstruktur enthält. Auf diesem Stream können nun verschiedene Operationen ausgeführt werden. [Urm14] Es gibt zwei verschiedene Operationstypen:

- Intermediate Operations
  z.B. filter(), sorted()
- Terminal Operations
  z.B. collect()

Die Besonderheit der Intermediate Operations ist, dass diese zu einer Pipeline zusammengeschlossen werden können. Das heißt, jede Intermediate Operation liefert wieder einen Stream zurück, auf der weitere Intermediate Operations oder eine Terminal Operation ausgeführt werden können. Dieser Typ der Operationen dient dem Zweck, den Stream zu manipulieren, sodass die Terminal Operation das gewünschte Ergebnis zurückliefern kann bzw. die nötigen Daten als Input bekommt. Die Stream API stellt dafür verschiedene Methoden bereit, mit denen man die Daten des Streams z.B. filtern, sortieren, reduzieren oder zusammenfassen kann. Auch das Konvertieren eines Streams in einen Stream eines anderen Typs ist möglich. [Urm14]

Eine Terminal Operation beendet die Pipeline und liefert ggf. ein Ergebnis zurück. Es ist nicht zwingend notwendig, dass die Terminal Operation einen Wert zurückgibt. Die Methode kann genauso gut mit dem Schlüsselwort void versehen sein. Es ist demnach jede Operation eine Terminal Operation, die keinen Stream zurückliefert. [Urm14]

Die neue Schnittstelle bietet außerdem eine einfache Möglichkeit zur Erzeugung von ParallelStreams, die ohne weiteren Programmieraufwand bestimmte Operationen auf den Quelldaten quasi-gleichzeitig (in mehreren Threads) ausführen. [Urm14]

Es folgt ein Beispiel für die Verwendung von Streams. Es verdeutlicht zudem, dass diese sehr gut mit Lambda-Ausdrücken synergieren, die im nächsten Abschnitt genauer beleuchtet werden.

Listing 2.1: Beispielcode ohne Stream-API und Lambdas [Urm14]

```
List<Transaction> groceryTransactions = new ArrayList<>();
for (Transaction t : transactions){
   if (t.getType() == Transaction.GROCERY) {
      groceryTransactions.add(t);
   }
}
Collections.sort(groceryTransactions, new Comparator() {
   public int compare(Transaction t1, Transaction t2){
      return t2.getValue().compareTo(t1.getValue());
   }
});
List<Integer> transactionIds = new ArrayList<>();
for (Transaction t: groceryTransactions) {
   transactionsIds.add(t.getId());
}
```

Listing 2.2: Beispielcode mit Stream-API und Lambdas [Urm14]

```
List<Integer> transactionsIds = transactions.stream()
   .filter(t -> t.getType() == Transaction.GROCERY)
   .sorted(comparing(Transaction::getValue).reversed())
   .map(Transaction::getId)
   .collect(toList());
```

2.1.4 Lambda Ausdrücke

Lambda-Ausdrücke verhalten sich in der Programmierung wie Kurzschreibweisen für Anonyme Innere Klassen, werden jedoch vom Java-Compiler in einen anderen, performanteren Bytecode übersetzt, der nicht mit einer Anonymen Inneren Klasse gleichzu-

setzen ist. Daher entstehen auch geringfügige Unterschiede im Verhalten der beiden Objektinstanzen, die aber zunächst nicht von größerer Relevanz sind. Durch einen Lambda-Ausdruck werden bei der Programmierung für den Compiler überflüssige Informationen einfach weggelassen. Lambdas können immer dort verwendet werden, wo normalerweise eine Anonyme Innere Klasse benutzt werden würde, die ein Functional Interface implementiert. Auf diese Weise wird ein Konzept der funktionalen Programmierung in die Objektorientierung übertragen und kann für prägnanteren Quellcode sorgen.

Ein Lambda-Ausdruck erscheint in zwei bzw. drei verschiedenen Ausprägungen.

Ausprägung 1: Der „normale“ Lambda-Ausdruck

„Normale“ Lambdas werden beschrieben, indem die Bezeichner der Methodenparameter der zu implementierenden Methode in runden Klammern und durch Komma getrennt definiert werden. Den Parametern folgt ein Pfeil „->“ und daraufhin der Methodenkörper in geschwungenen Klammern. Ist nur ein Parameter zu bezeichnen, können die runden Klammern weggelassen werden. Wenn der Methoden-Body nur eine Anweisung umfasst, können die geschwungenen Klammern weggelassen werden. Wenn die einzelne Anweisung eine Return-Anweisung mit einem darauffolgenden Wert ist, wird das Schlüsselwort *return* ebenfalls weggelassen. [Ora]

Listing 2.3: Anonyme Innere Klasse ohne Lambda

```
Consumer<Integer> consumer = new Consumer<Integer>() {
   public void accept(Integer i) {
      System.out.println(i);
   }
};
```

Listing 2.4: Anonyme Innere Klasse mit Lambda

```
Consumer<Integer> consumer = (i) -> {
   System.out.println(i);
};
```

Listing 2.5: Verkürzter Lambda-Ausdruck

```
Consumer<Integer> consumer = i -> System.out.println(i);
```

Ausprägung 2: Die Methodenreferenz

Mit der Methodenreferenz wird ein weiterer Operator eingeführt, der Double-Colon-Operator „*::*“. Unter Verwendung dieser Zeichenkette kann, anstatt eine Anonyme Innere Klasse auszuprogrammieren, eine Methode referenziert werden, welche die gleiche Signatur wie die eigentlich zu implementierende Methode des funktionalen Interfaces hat. [Ull14, S. 81f.]

Listing 2.6: Beispiel mit Lambda und Methodenreferenz

```
List<String> list = new ArrayList<String>();
list.add("abc");
list.add("defgh");

// Lambda
List<Integer> stringLengths = list.stream().map(string ->
      string.length()).collect(Collectors.toList());

// Methodenreferenz
List<Integer> stringLengths =
    list.stream().map(String::length).collect(Collectors.toList());
```

Auf die gleiche Weise können statische Methoden sowie Methoden von lokalen Variablen referenziert werden.

Ausprägung 3: Die Konstruktorreferenz

Eine eher untypische Variante ist die Konstruktorreferenz. Sie funktioniert analog zur Methodenreferenz. [Ull14, S. 83f.]

Listing 2.7: Beispiel - Konstruktorreferenz (analog zu vorherigem Beispiel)

```
List<Integer> stringLengths =
    list.stream().map(Integer::new).collect(Collectors.toList());
```

In diesem Fall wird der Konstruktor *new Integer(String s)* verwendet, der versucht, den String zu parsen und in einem Integer zu verpacken. Zu beachten ist, dass Lambdas keinen neuen Scope für Variablen definieren. Gibt es also in einer Methode eine lokale Variable *x*, darf ein Methodenparameter eines Lambdas, der in dieser Methode definiert wird, nicht ebenfalls *x* heißen. [Ora]

Listing 2.8: Lambda-Scope [Ora]

```
int x = 21;
Consumer<Integer> myConsumer = (x) -> {...} // Compiler-Error
```

2.2 JavaFX

Bei JavaFX handelt es sich um ein relativ neues GUI-Framework. Die letzte Release-Version (JavaFX 8) wurde zusammen mit Java 8 ausgeliefert. JavaFX wurde entwickelt, um das bewährte, aber dennoch veraltete GUI-Framework Swing abzulösen und wartet mit modernen Features auf. [Mül15]

Deklarative GUI-Definition

Die Benutzeroberfläche kann sowohl im Java-Code als auch in einer gesonderten FXML-Datei (XML-Syntax) definiert werden. Auch eine Kombination beider Möglichkeiten ist unproblematisch, da man sich durch spezielle Annotationen in den Java-Klassen die GUI-Komponenten aus den FXML-Dateien anhand einer ID injizieren lassen kann. Dazu ist nur die Annotation @FXML an einer gleichnamigen Membervariable des gleichen Komponententyps von Nöten.

Styling per CSS

Das Aussehen nahezu aller Komponenten kann durch wiederverwendbare CSS-Klassen in gesonderten CSS-Dateien definiert werden. [Mül15]

Animationen

JavaFX stellt eine Reihe von Animationen bereit, die auf verschiedene GUI-Elemente angewandt werden können und so die Eigenschaften dieser verändern. [Mül15]

Properties und Bindings

Mit JavaFX wurden Properties an UI-Komponenten eingeführt, die auf einfache Weise an andere Werte gebunden werden können. Verändert man den Wert eines Bindings, werden alle Observer (gebundene Werte) ebenfalls verändert. [Mül15] Solche Properties (ObservableValues) können auch selbst definiert und für andere Zwecke verwendet werden. Sie existieren in verschiedenen Ausprägungen mit unterschiedlicher Typisierung. [Hom13]

Die Anwendung

Die Anwendung, die im Rahmen dieser Ausarbeitung analysiert und erweitert wird, nennt sich FalkoFX. Sie entstand aus einem Kundenprojekt (Falko) eines Automobilherstellers und dient der Anzeige von länderspezifischen Produktionsfreigaben für verschiedene Ausprägungen von Fahrzeugmodellen.

Das Projekt Falko wird neben FalkoFX weiterentwickelt. Während FalkoFX für einen einfachen, möglichst benutzerfreundlichen (lesenden) Zugriff auf die Daten sorgt, bietet der weit komplexere Falko-Client noch vielfältigere Möglichkeiten zum Anlegen, Manipulieren und Pflegen der Daten an. Da die Übersichtlichkeit des FalkoFX-Clients gewährleistet bleiben soll und das Programm einen eingeschränkteren Benutzerkreis als das Ausgangsprojekt hat, bietet es dementsprechend weniger Funktionalitäten.

2.3 Die Anwendung

Derzeit sind zwei von drei der für das Release der Software vorgesehenen Anwendungsfälle implementiert. Der dritte Anwendungsfall kommt im Rahmen dieser Arbeit hinzu. Der grobe Aufbau eines jeden, derzeit beauftragten, Anwendungsfalles ist folgender:

1. Der Nutzer wechselt zu einem Anwendungsfall
2. Es öffnet sich ein Bildschirm, in dem der Nutzer die Eigenschaften einstellt, nach denen die Rohdaten gefiltert werden sollen.
3. Der Nutzer wechselt zu einer Ergebnisansicht
4. Die Daten werden in der gewählten Ansicht dargestellt

Die getroffene Filterauswahl kann in der Sidebar an der rechten Seite nachvollzogen und bearbeitet werden. Die im Filter auswählbaren Werte sind nach Attributen sortiert. Ein Attribut kann z.B. „Ländername“ und die dazugehörigen Werte „Deutschland, Kanada, Schweiz, ...“ lauten. Der Filter wird in einem späteren Abschnitt (3.1) noch genauer erläutert.

In jedem Anwendungsfall sind verschiedene Aktionen und Exportmöglichkeiten verfügbar, die allerdings für diese Ausarbeitung nicht zwangsläufig relevant sind.

Anwendungsfall 1

Der erste Anwendungsfall bezieht sich auf die Anzeige von Länderdaten. Nachdem nach bestimmten Eigenschaften, die ein Land potenziell haben kann, gefiltert wurde, kann das Ergebnis in einer tabellarischen Ansicht oder der Galerie betrachtet werden.

Die Galerie ist eine selbstentwickelte Komponente, die im oberen Bereich Icons, gleich einer Bordüre, anzeigt und je nach Selektion eines dieser Items in einem größeren Bereich eine Detailansicht zu dem selektierten Element darstellt.

Anwendungsfall 2

Der zweite Anwendungsfall ist dem ersten sehr ähnlich. Jedoch geht es hier um die Anzeige von technischen Daten einer großen Anzahl an Fahrzeugen. Bereits im Filter gibt es kleine Unterschiede, die später erläutert werden. Die Galerie, die in Anwendungsfall 1 benutzt wurde, entfällt für dieses Szenario, da sie nicht praktikabel wäre.

Anwendungsfall 3

Die grundlegende Funktionalität des dritten Anwendungsfalles ist ein Teil dieser Projektarbeit. Der Nutzer will in diesem Bereich der Anwendung Produktionsfreigaben für Fahrzeugmodelle einsehen können. Diese Freigaben sind länderabhängig und können unterschiedlich ausfallen. Die wichtigste Information dieser Freigaben sind ein Datum, das den Start der Produktion festlegt und eines, welches das Ende bestimmt.

Um in die Ergebnisansicht dieses Szenarios zu gelangen, muss der Filter sowohl für Länder als auch für Fahrzeuge konfiguriert werden. Für die Kombination der beiden Ergebnismengen werden daraufhin die Produktionsfreigaben angezeigt.

2.4 Codedesign-Patterns

2.4.1 MVC-Pattern

Das MVC-Pattern hilft dabei, den Quellcode zu strukturieren und übersichtlicher zu gestalten. Durch die Anwendung des Entwurfsmusters können Programme in Bereiche mit unterschiedlichen Zuständigkeiten eingeteilt werden. MVC steht für Model-View-Controller. In diese drei entsprechenden Bereiche kann der Code untergliedert werden. [FFSB06, S. 529ff.]

Das Model

Das Model enthält die Datenstrukturen, die für die Verarbeitung benötigt und später angezeigt werden. [FFSB06, S. 529ff.]

Die View

Dieser Bereich entspricht der Benutzeroberfläche, mit der ein Anwender interagieren kann. Es werden auf Basis des Models Daten oder Informationen visualisiert. [FFSB06, S. 529ff.]

Der Controller

Der Controller ist die Schnittstelle zwischen Model und View. Er verarbeitet eingegebene Informationen aus der View und manipuliert das Model nach Interpretation dieser. Außerdem kann er die Daten bzw. den Zustand des Models verändern. Der Controller enthält die gesamte Anwendungslogik und modifiziert das Verhalten der View gemäß dem aktuellen Zustand der Applikation. [FFSB06, S. 529ff.]

Die Anwendung in FalkoFX

Auf Codeebene kann das MVC-Pattern durch verschiedene andere Design-Patterns realisiert werden. Um die Änderungen des Models zu publizieren und andere Teile der Anwendung zu benachrichtigen, besteht die Möglichkeit, das Observer-Pattern zu verwenden. Genau dies bietet auch JavaFX an. Hier gibt es das *Observable*-Interface, welches es erlaubt, Listener an einer Instanz der implementierenden Klasse zu registrieren, die bei einer Änderung benachrichtigt werden und darauf reagieren können.

In der View wird durch JavaFX das Composite-Pattern angewandt. „*Die Anzeige besteht aus ineinander verschachtelten Fenstern, Panels, Buttons, Textlabels und so weiter*“ [FFSB06, S. 532]. In JavaFX sind die Komponenten intern in einer Baumstruktur angeordnet, dem sogenannten Szenegraphen, was die Verschachtelung gut zeigt.

Im Controller kommt zusammen mit der View das Strategy-Pattern zum Einsatz. Die View selbst kümmert sich nur darum, wie die Darstellung aufgebaut ist. Sie wird mit einer Strategie konfiguriert, die von dem Controller bereitgestellt wird. Die View selbst implementiert keine Programmlogik.

Die Patterns werden in FalkoFX zwar durchgängig verwendet, die Trennung in verschiedene Klassen jedoch ist nicht immer nach der Intention des MVC-Entwurfsmusters durchgeführt worden. Stattdessen verschwimmen in einigen benutzerdefinierten Komponenten die Grenzen zwischen Controller und View. Der Rest des Programmes wird

davon allerdings kaum beeinflusst, da die Komponenten eine gekapselte Einheit bilden. Das Model hingegen ist von View und Controller stets entkoppelt.

2.4.2 Command-Pattern

Das Command-Pattern beschreibt ein Entwurfsmuster der Softwarearchitektur, bei dem Funktionsaufrufe und Berechnungen in einem Kommando-Objekt gekapselt werden. Die Anweisungen können ausgeführt werden, ohne dass das ausführende Objekt Kenntnis von dem aufrufenden Objekt hat. [Com]

Durch die Verwendung von Kommandos können diese nach der Erstellung zu einem beliebigen Zeitpunkt ausgeführt werden. sie können in einer Rangfolge angeordnet werden und sie können Informationen zur Rücknahme (*Undo*-Operation) enthalten. Außerdem lassen sich die Kommando-Objekte durch Parameter in unterschiedlichen Ausprägungen erzeugen. [Zel00]

Eine abstrakte *Kommando*-Klasse gibt die Struktur vor. Dies ist mindestens eine *#execute()*-Methode. Diese wird in einer konkreten Implementierung des Kommandos spezifiziert. [Com] Das konkrete Kommando wird durch den Client erzeugt und enthält Informationen über den Zustand und das Verhalten. Der Client setzt ebenfalls den Receiver im Kommando-Objekt. Der Receiver stellt Methoden-Implementierungen für das Kommando bereit, auf die das Kommando in seiner *#execute()*-Methode verweist. Bei jeder Erzeugung kann ein anderer Receiver gesetzt werden, der je eine andere Implementierung der Funktionen bereitstellt und so das Verhalten anders definiert. Zum Schluss wird das Kommando an einen Invoker übergeben, der es zu einem beliebigen (definierten) Zeitpunkt ausführen kann. [Com]

2.5 Gestaltungsgrundlagen

2.5.1 Gestaltgesetze

Die Gestaltgesetze basieren auf Forschungen der Gestaltpsychologie. Es sind allgemein gültige Regelungen, die auf alle Bereiche der Wahrnehmung zutreffen. Folgend werden die für diese Arbeit relevanten Gesetze beschrieben.

Gesetz der Nähe

> *„Dinge, die räumlich nahe beieinander liegen, werden vom Auge gruppiert.“*

[Mos12, S. 185]

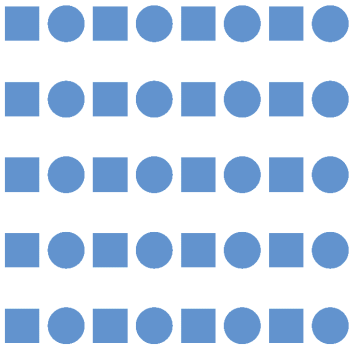


Abbildung 2.1: Gesetz der Nähe [Sch]

Die Abbildung 2.1 stellt das Gesetz der Nähe dar. Die Wahrnehmung gruppiert die Objekte zu Zeilen, obwohl die Spalten aus gleichartigen Elementen bestehen (siehe auch Gesetz der Ähnlichkeit).

Gesetz der Ähnlichkeit

> „*Visuell ähnliche Objekte werden vom Auge gruppiert. Die Ähnlichkeit kann durch Farbe, Form, Größe, Textur oder Bewegungsrichtung entstehen.*“

[Mos12, S. 185]

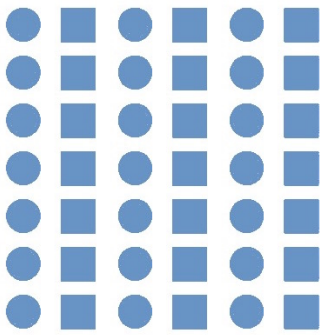


Abbildung 2.2: Gesetz der Ähnlichkeit [Gri]

Dadurch, dass die Elemente in diesem Fall zueinander den gleichen Abstand haben, die Formen sich jedoch unterscheiden, werden die Elemente durch die Wahrnehmung hier zu Spalten zusammengefasst.

3 Implementierung

3.1 Filter-Logik

Ausgangssituation

Der Filter ist der Teil der Programmlogik, der aus den Rohdaten die Ergebnismenge produziert. Dies geschieht anhand von Filterwerte, die durch den Nutzer gewählt werden. Jeder dieser Werte ist einem übergeordneten Attribut zugeordnet.

Um zu filtern, wählt der Nutzer in der Benutzeroberfläche ein Attribut aus, zu dem er dann alle vorhandenen Werte angezeigt bekommt, die noch nicht zum Berechnen der Ergebnismenge genutzt werden. Daraufhin kann er einen oder mehrere Werte anwählen. Die Werte verschwinden aus der Werteliste und tauchen im gleichen Moment in der Sidebar am rechten Bildschirmrand, unter dem entsprechenden Attribut gruppiert, wieder auf. Diesen Vorgang kann der Nutzer für verschiedene Attribute wiederholen. Hat er einen falschen Wert übernommen, kann dieser durch einen Klick auf das Element in der Sidebar wieder entfernt werden und er „wandert“ zurück in die dazugehörige Werteliste. Derzeit ist der Filter additiv implementiert. Das bedeutet, es sind immer alle Filter in der Ergebnismenge, die einem der Kriterien entsprechen. In der Mengenlehre wäre das Äquivalent dazu die Vereinigungsmenge.

Das folgende Beispiel verdeutlicht die Filterfunktionalität für den ersten Anwendungsfall:

Auswahl des Nutzers:

- Kontinent: Europa
- Klimazone: Tropische Klimazone

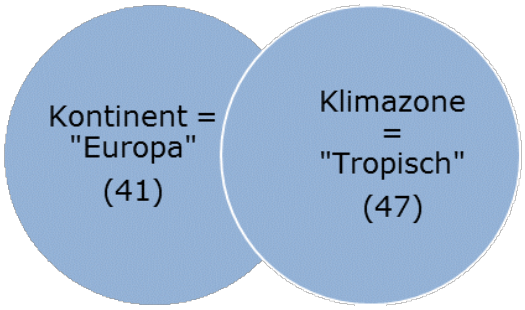


Abbildung 3.1: Vereinigungsmenge Filter

Den Wert „Europa“ unter dem Attribut Kontinent haben 41 Länder gesetzt, den Wert „Tropische Klimazone“ unter dem Attribut „Klimazone“, 47 Länder. Die Ergebnismenge besteht nun aus allen Objekten, die mindestens eine der beiden Bedingungen erfüllen, also den 47 Objekten der Klimazonenbedingung und den 41 Ländern, die in Europa liegen. Natürlich werden die Werte, die beiden Bedingungen entsprechen nur einmal in der Ergebnismenge auftauchen (äquivalent zur Vereinigungsmenge). In diesem Beispiel käme man auf 84 Länder, die den Filtereinstellungen entsprechen.

Problematik und Konzept

Der Filter soll für die Anwendungsfälle 2 und 3 angepasst werden. Der Wunsch des Kunden ist eine Mischung aus dem additiven Filter, der bereits existiert, und einem Filter, bei dem die Filterauswahl die Ergebnismenge wieder einschränkt.

Die Ergebnismenge wird erweitert, wenn Werte des gleichen Attributes ausgewählt werden. Die Einschränkung hingegen geschieht, wenn Werte eines anderen Attributes ausgewählt werden.

Auch hier hilft die Mengenlehre, das Verhalten des Filters zu verdeutlichen (Anwendungsfall 2):

Schritt 1 – Auswahl aus Attribut Motor:

- Motor A
- Motor B

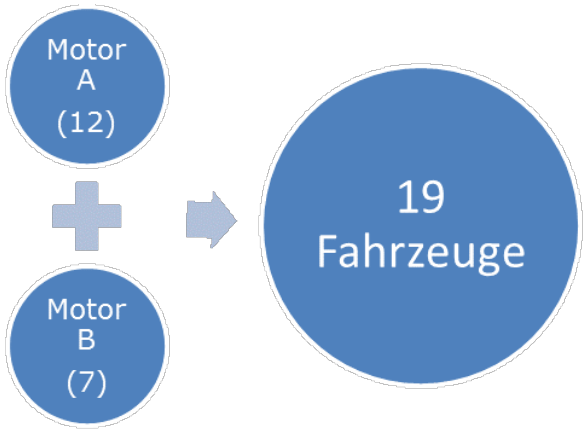


Abbildung 3.2: Teilmenge 1 - Exklusivfilter

Zu diesem Zeitpunkt besteht die Ergebnismenge aus genau diesen Fahrzeugen. Die Anzahl an Fahrzeugen der beiden Werte „Motor A“ und „Motor B“ können ohne weitere Bedenken addiert werden, da es keine Fahrzeuge mit 2 Motoren gibt.

Schritt 2 – Auswahl aus Attribut Getriebe

- Getriebe A
- Getriebe B

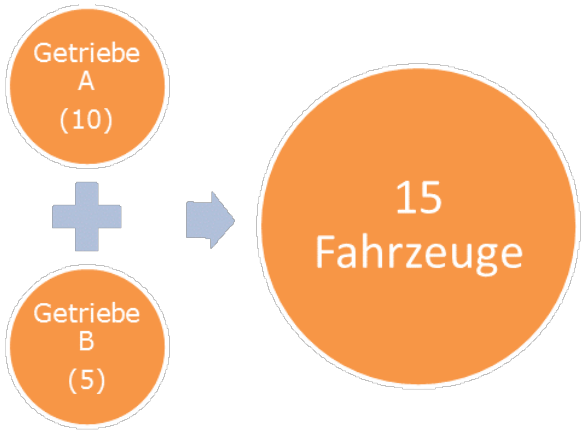


Abbildung 3.3: Teilmenge 2 - Exklusivfilter

Bei der Auswahl in Schritt 2 würden 15 Fahrzeuge in der Ergebnismenge landen. Da jetzt aber über zwei verschiedene Attribute gefiltert wird, muss die Ergebnismenge aus den beiden einzelnen Ergebnissen berechnet werden. Dazu muss die Schnittmenge der beiden Teilmengen gebildet werden.

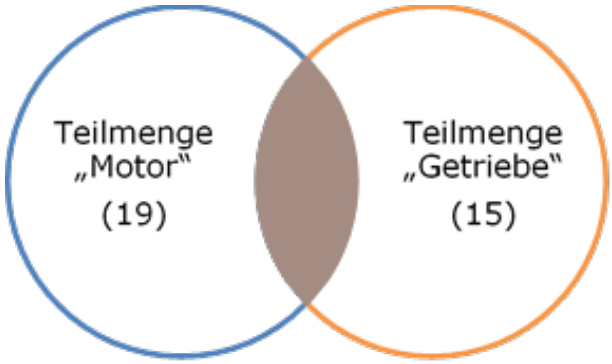


Abbildung 3.4: Schnittmenge - Exklusivfilter

Die Ergebnismenge ist in der Grafik genau der Bereich, in dem sich die beiden Teilmengen überschneiden. Sie enthält alle Elemente, die sowohl in der einen als auch in der anderen Teilmenge vorkommen.

Bei der Übernahme des Konzeptes auf die tatsächliche Implementierung des Filters, muss allerdings bedacht werden, dass die Schnittmenge aus einer Vielzahl an Attributen berechnet werden muss.

Zur Verbesserung der Gebrauchstauglichkeit des Filters müssen in diesem Fall nach Selektion von Werten eines Attributes die Wertelisten der anderen Attribute aktualisiert werden, sodass durch die alleinige Auswahl von Werten verschiedener Attribute kein leeres Ergebnis entstehen kann.

Im Umkehrschluss bedeutet das Vorhergegangene allerdings, dass die Wertelisten auch jedes Mal aktualisiert werden müssen, wenn ein Wert wieder aus der Filterselektion „rausgeworfen“ wird. Dies muss auch geschehen, wenn sich dadurch eine gerade betrachtete Werteliste ändert.

Umsetzung

Auf technischer Seite ist der Filter so implementiert, dass es definierte Datenstrukturen für alle relevanten Mengen gibt. Folgende Strukturen existieren:

- Menge aller Filterattribute

- Alle Wertelisten, zugeordnet zu einem Filterattribut
- Zuordnung von Wertelisten mit (noch) selektierbaren Werten zu jeweils einem Attribut
- Menge aller Filterattribute, von denen mindestens ein Wert selektiert wurde
- Menge selektierter Werte mit Zuordnung zu dem entsprechenden Attribut

Um die obigen Mengen aktuell zu halten, also Änderungen in allen Mengen quasigleichzeitig zu publizieren werden JavaFX-Listener benutzt. Wird die Menge der selektierten Werte geändert, werden die anderen Mengen aktualisiert.

Der Filter wird durch einen DataProvider und durch einen FilterDataProvider gestützt. Der DataProvider liefert alle existierenden Modellobjekte, die behandelt werden sollen und der FilterDataProvider stellt die Attribute sowie die verfügbaren Werte zur Verfügung.

Die Logik führt bisher die in dem Abschnitt Ausgangssituation beschriebenen Mengenoperation mit einigen Besonderheiten aus. So gibt es zum Beispiel unterschiedliche Attributtypen. Es gibt *ValueListFilterAttributes*, die die oben genannten Wertelisten beinhalten, *BooleanFilterAttributes*, die nur die Werte Ja und Nein beinhalten und es gibt *ParentFilterAttributes*, die dazu dienen, andere Filterattribute zu gruppieren und diese Filterattribute als Sublisten der ParentFilterAttribute darzustellen. Um die Beschreibung der Änderungen auf das Wesentliche zu reduzieren, wird auf diese und weitere Eigenheiten nicht näher eingegangen.

Die Änderungen am Quellcode umfassen im Groben folgendes:

Für die Methode *#isInFilter(Modelobjekt)*, die überprüft, ob ein Objekt den gewählten Filterkriterien entspricht, wurde eine Fallunterscheidung eingeführt. Diese erlaubt es, dass der Filter für den Anwendungsfall 1 wie bisher funktioniert, für den 2. Anwendungsfall jedoch das Schnittmengenkonzept umsetzt. Dazu wird für jedes selektierte Attribut überprüft, ob der Wert, den das Modelobjekt für das entsprechende Attribut gesetzt hat, in der Menge der selektierten Werte vorhanden ist. Ist dies für ein Attribut nicht der Fall, bedeutet das, dass das geprüfte Modelobjekt nicht in der Ergebnismenge vorhanden sein darf.

Auch die Wertelisten mit den auswählbaren Werten aktualisieren sich jetzt zeitgleich mit der Filterselektion. Dies erforderte mehr Aufwand, da die Wertelisten nach Definition nur durch die Selektion von Werten eingeschränkt werden dürfen, die einem anderen Attribut zugeordnet sind, als die betrachtete Werteliste. Um dies zu realisieren wurde eine neue Menge zu den oben beschriebenen Mengen hinzugefügt, die parallel

zu den selektierbaren Werten eine gefilterte Sicht auf diese Werte bietet. Um die einzelnen Listen zu erzeugen, die jeweils einem Attribut zugeordnet sind, benötigt es zum einen eine Änderung an der *#isInFilter(Modelobjekt)* Methode und zum Anderen die Aktualisierung bei Änderung der selektierten Werte. Bei der *#isInFilter(Modelobjekt)* Methode kam ein neuer Parameter hinzu, welcher ein zu ignorierendes Attribut entgegennimmt. Dieses „Exklusiv-Attribut“ wird dann bei der darauf folgenden Berechnung nicht mit einbezogen.

Listing 3.1: Auszug aus Code der Methode *#isInFilter(Modelobjekt Filterattribut)*

```
[...]
for (AbstractFilterAttribute<A> attribute : selectedAttributes) {
   if (!attribute.equals(exclusiveAttribute)) {
      Object attributeValue = dataProvider.resolve(dataObject,
          attribute.getAttributeName());
      if (!isValueSelected(attribute, value)) {
           return false;
      }
   }
}
return !selectedValues.isEmpty();
// returns true when loop passed through and false when there is no selection
[...]
```

Diese Berechnung wird für jedes Modelobjekt ausgeführt um festzustellen, ob es in der Ergebnismenge inbegriffen ist. Die einzelnen, den Attributen zugeordneten, Wertelisten der nicht-selektierten Filterwerte, begründen sich gemäß der Implementierung auf der Berechnung einer Pseudo-Ergebnismenge, die das Attribut der betrachteten Werteliste nicht berücksichtigt. Die Attributwerte aller übrig gebliebenen Modelobjekte, die nicht in der Pseudo-Ergebnismenge vorhanden sind, dienen dazu, die spezielle Werteliste zu befüllen. So wird vermieden, dass Filterwerte angewählt werden, die zu einer leeren Ergebnismenge führten.

Andersherum betrachtet ist es nicht einfach möglich, zu verhindern, dass durch das „rauswerfen“ von Attributwerten aus der Filterselektion, eine leere Ergebnismenge geschaffen wird. Um dies zu realisieren, müsste die Handlungsfreiheit des Nutzers stark eingeschränkt und verhindert werden, dass solche Attributwerte deselektiert werden. Zudem erforderte dies eine ständige Neuberechnung der möglichen Ergebnismenge, falls ein bestimmter Wert aus der Filterauswahl „geworfen“ werden würde – und dies für jeden möglichen Wert, der entfernt werden kann.

3.2 DataProvider

Struktur

Der DataProvider versorgt die einzelnen Bereiche der Anwendungslogik mit Daten aus dem Model. Es gibt für jeden Anwendungsfall einen eigenen DataProvider, da jedes Szenario auf einer eigenen Art von Modelobjekten beruht. Die Klasse stellt neben den Modellobjekten weitere Methoden zum Auflösen von Attributwerten bereit. Dazu nimmt die Methode ein Modellobjekt und ein Attribut entgegen und liefert den dazugehörigen Wert zurück.

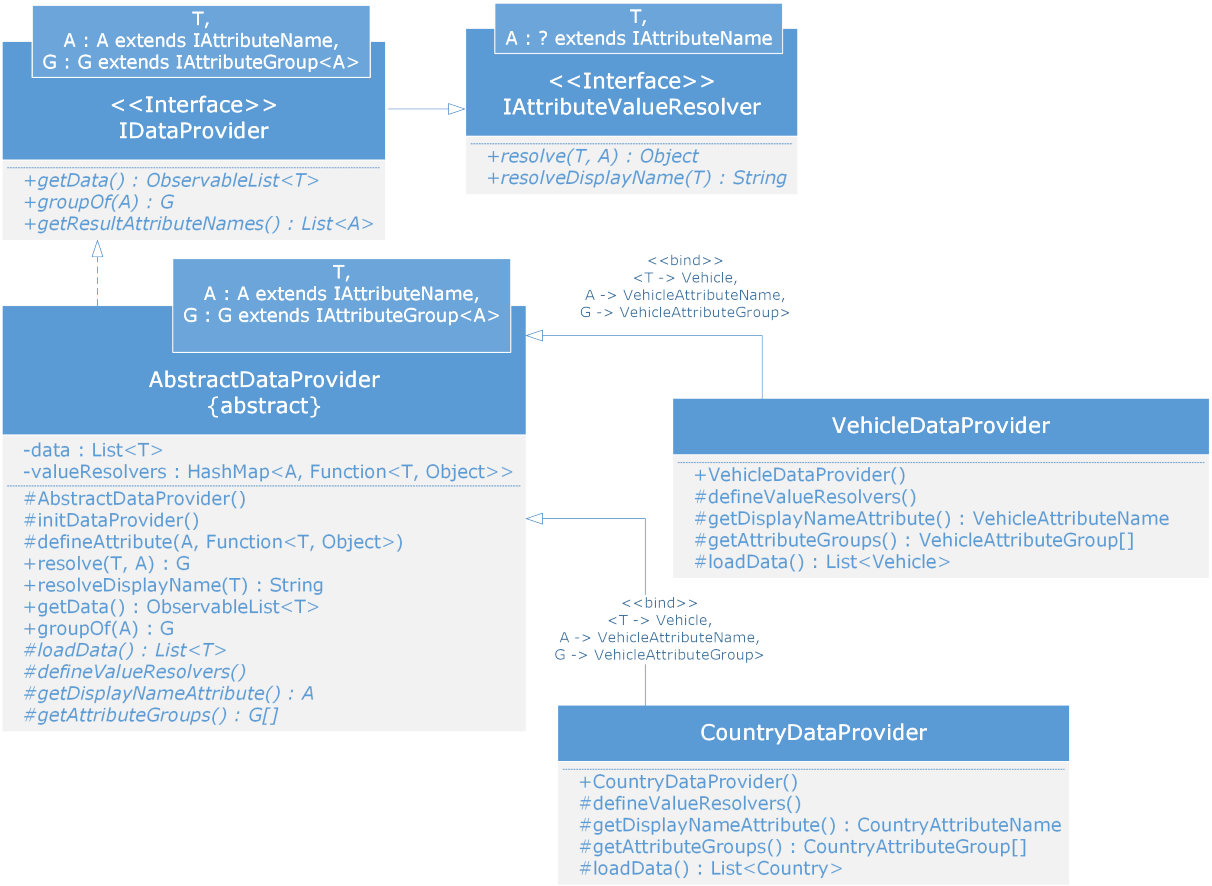


Abbildung 3.5: Klassendiagramm DataProvider

Die bestehende Klassenstruktur enthält bereits Interfaces und abstrakte Klassen, welche die Implementierungsstruktur eines DataProviders vorgeben. Die Grafik 3.5 zeigt

einen Ausschnitt der Klassenhierarchie und stellt die zum Verständnis relevanten Member dieser Klassen dar. Das hierarchisch höchste Interface *IAttributeValueResolver* liefert Funktionen, die für das Auflösen spezieller Werte zuständig sind. Anhand eines Modelobjektes T, mit dem das Interface als Typparameter versehen wurde, und einem Attribut, das das Interface *IAttributeName* implementieren muss, kann ein Wert erhalten werden (z.B. „Spanien“ für das Attribut Land). Ein weiteres Interface stellt der *IDataProvider* dar, der von *IAttributeValueResolver* ableitet. Hier werden Methoden definiert, um die Modelobjekte dieses DataProviders zu erhalten. Außerdem kann man per Methodenaufruf die Attribute und die Zuordnung zu einer Attributgruppe bekommen, die für die Ergebnisanzeige benötigt werden.

Der *AbstractDataProvider* bietet nun die ersten Implementierungen für die abstrakten Methoden der Interfaces an. Zusätzlich werden weitere abstrakte Methoden hinzugefügt, die die konkrete Implementierung durch Unterklassen erfordern. Die Methoden werden für das Laden der Daten aus der Datenbank, sowie der Definition von ValueResolvers benötigt. Ein ValueResolver beschreibt mithilfe einer Java8-*Function*, wie für ein bestimmtes Attribut bei Eingabe eines Modelobjektes T, der Wert für dieses Attributerhalten werden kann. Ein Resolver ist immer genau einem Attribut zugeordnet - und umgekehrt.

Problemstellung

Es soll, parallel zu dem *CountryDataProvider* und dem *VehicleDataProvider* der DataProvider, für den 3. Anwendungsfall implementiert werden. Dazu gehört die Definition der Attribute, die durch das Interface IAttributeValueResolver benötigt werden, sowie die jeweilige Zuordnung zu einer Attributgruppe. Zusätzlich muss eine Datenstruktur entwickelt werden, die die geladenen Informationen in einem Objekt zusammenfasst – also die kombinierten Länder und Fahrzeugdaten zusammen mit den Produktionsfreigaben.

Umsetzung

Zunächst müssen die Attribute definiert werden. Diese sind abhängig von der Ergebnispräsentation. Gemäß Anforderung sollen in der Ergebnistabelle Details zu den Fahrzeugen angezeigt werden und die Start/ Enddaten der Produktionsfreigaben. Daher entsprechen die definierten Attribute in erster Linie den Attributen aus dem Fahrzeug-Szenario. Aus diesem Grund müssen die Attribute nicht alle neu definiert werden, sondern können durch eine statische Methode *#createAttributeName(IAttributeName)* erzeugt werden. Das neu erzeugte Attribut dient nur als Kapselung für das eigentliche Attribut, welches aus der Menge der Fahrzeug- oder Länder-Attribute kommt. Die Methode wird für jedes Fahrzeug-Attribut bei der Definition der Attributgruppen aufgerufen. In

dem Enum der Attributgruppen werden dem Konstruktor die Original-Attributgruppen übergeben und die Attribute für den 3. Anwendungsfall können per Iteration über die zugeordneten Attributnamen erzeugt werden. Um zu vermeiden, dass Attribute mehrfach definiert werden, werden die gekapselten Attribute in einer statischen *Map* gespeichert, die als *Key* das Original-Attribut enthält und als Value den Wrapper zur Verfügung stellt. Zusätzlich zu den Wrapper-Attributen gibt es statische Attribute, die für die Produktionsfreigabedaten stehen.

Im DataProvider müssen, nachdem die Attribute erzeugt wurden, die ValueResolvers beschrieben werden. Es muss also für jeden Attributnamen eine *Function* implementiert werden. Allerdings kann, durch überschreiben der *#resolve(Modelobjekt, AttributeName)*-Methode doppelter Code vermieden werden. Wie bereits erläutert, stammen viele der Attribute aus den anderen beiden Anwendungsfällen und so können die jeweiligen Original-DataProvider genutzt werden, um die Werte für die meisten Attribute aufzulösen. Es müssen nur noch die Resolver für die hinzugekommenen Attribute definiert werden.

In der Theorie werden für den 3. Anwendungsfall die Daten aus dem Länder-Anwendungsfall mit den Daten aus dem Fahrzeug-Anwendungsfall kombiniert und mit zusätzlichen Daten, den Produktionsfreigaben versehen. Dies geschieht für jede Kombination aus Fahrzeugen und Ländern, für die es eine Produktionsfreigabe gibt. Da die Menge an Produktionsfreigaben mit allen daran registrierten Daten lange Ladezeiten und viel Speicher erfordern würde, können die Produktionsfreigaben vorab nicht geladen werden. Stattdessen werden, um Werte für den Filter bereitzustellen, zunächst nur die Länder- und die Fahrzeugdaten aus der Datenbank geladen. Gekapselt werden diese Daten in der Datenstruktur *FilterElement*. Diese Struktur wird im nächsten Kapitel genauer erläutert.

3.3 Multi-Filter

Problemstellung

Für den neuen Anwendungsfall ändern sich nun auch die Anforderungen an den Filter. Die Auswahl beruhte bislang nur auf Modellobjekten, die fachlich genau eine Art von anzuzeigenden Daten repräsentierten. Die Länderobjekte stellen Länder mit gewissen Eigenschaften dar und die Fahrzeugobjekte stehen für Fahrzeuge. Die Modellobjekte für die neuen Funktionen hingegen bestehen nach dem Laden der Daten aus zwei

verschiedenen Datensätzen (Land und Fahrzeug) und zur Zeit der Anzeige aus drei Datensätzen (zusätzlich die Produktionsfreigabe-Objekte).

Die Ergebnismenge, die auf dem Filter basiert, soll frei konfigurierbar sein. Das bedeutet, es sollen Produktionsfreigaben für eine beliebige Kombination von Ländern mit Fahrzeugen angezeigt werden können. Um das zu ermöglichen werden zwei voneinander unabhängig konfigurierbare Filter benötigt, die im Endeffekt jedoch zu einer einzelnen Ergebnismenge führen. Die einzelnen Filter sollen analog zu ihren jeweiligen ursprünglichen Anwendungsfällen implementiert werden. Der Länderfilter soll additiv funktionieren (siehe Abschnitt 3.1 - Ausgangssituation) und der Fahrzeug-Filter einschränkend (siehe Abschnitt 3.1 - Problematik und Konzept).

Nach der Auswahl der Filterwerte müssen die Produktionsfreigaben in die Land-Fahrzeug-Kombinationen eingefügt werden.

Umsetzung

Die erste Problematik bezieht sich darauf, wie ein solcher Filter umzusetzen ist. Eine Möglichkeit wäre es, die Klasse Filter dahingehend umzuschreiben, dass sie sowohl mit einem als auch mit 2 Modellobjekten arbeiten kann. Der Vorteil dieser Lösung läge vor Allem darin, dass nur wenige Klassen bearbeitet werden müssten. Eine andere Möglichkeit ist, die Filterlogik zu belassen und stattdessen die umgebende Struktur so zu verändern, dass 2 Instanzen des Filters gleichzeitig und unabhängig voneinander verwendet werden können.

Da die Logik der Filter-Klasse mit verschiedenen Sonderbehandlungen ohnehin schon recht komplex ist, ist die zweite Möglichkeit in diesem Falle zu bevorzugen. Folgend ist die Methodenschnittstelle der Filter-Klasse aufgeführt.

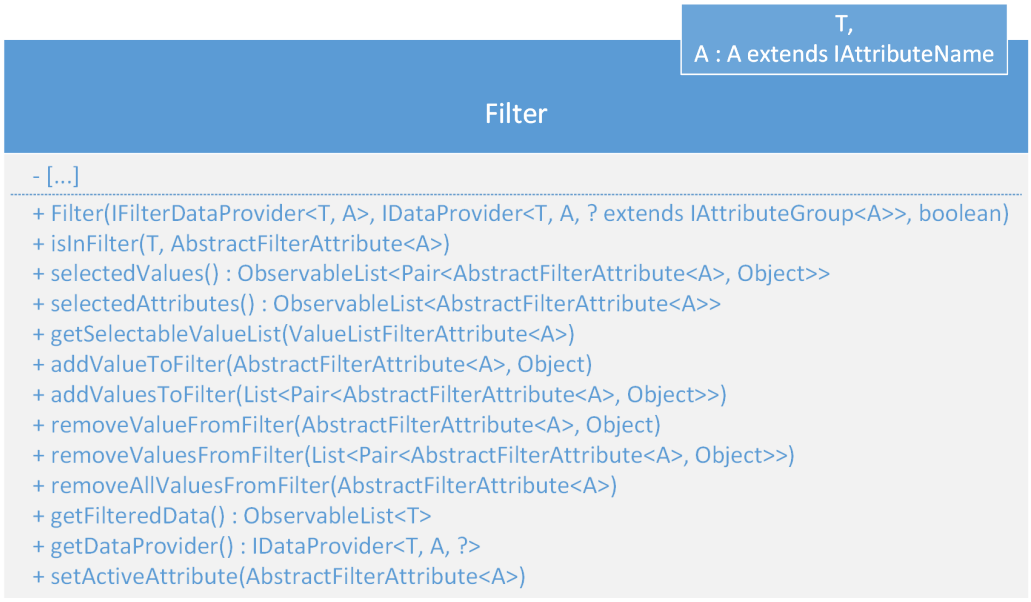


Abbildung 3.6: Klassenaufbau - Filter

Die Schnittstelle bietet Methoden, um auf die in Abschnitt 3.1 - Umsetzung definierten Mengen zuzugreifen. Es zeigt sich, dass einige Mengen nicht ganz wie erwartet implementiert sind. Im Beispiel der *selectedValues* wird eine *ObservableList* (JavaFX) vom Typ *Pair<AbstractFilterAttribute<A>, Object>* verwendet, anstatt der naheliegenden *Map* mit der Typisierung *<AbstractFilterAttribute<A>, List<Object»*. Diese Entscheidung wurde getroffen, da sonst eine Benachrichtigung per JavaFX-Listener über eine Änderung an dieser Menge nur auf umständlichem Wege möglich wäre. Dies ist bei mehreren dieser Mengen der Fall.

Eine weitere Auffälligkeit ist die häufige Verwendung von Generics. Wie bereits zuvor erwähnt, verarbeitet der Filter verschiedene Arten von Modellobjekten. Um dies zu bewerkstelligen, benötigt er nur die Informationen aus dem bereits erläuterten DataProvider und weitere Information aus einem FilterDataProvider. Neben dem Filter gibt es noch weitere Klassen, die generisch implementiert sind.

Für den FilterDataProvider gibt es ebenfalls pro Anwendungsfall eine spezifische Implementierung. Für die Aufgabe ist dieser Teil der Logik jedoch nur bedingt relevant. Im

Wesentlichen erstellt der FilterDataProvider aus den Modellobjekten des spezifischen Szenarios die Wertelisten für jedes Attribut zusammen indem er alle Modellobjekte durchläuft, die verschiedenen Werte zusammenträgt und gleiche Elemente zu einem zusammenfasst.

Um die Filteranforderungen umzusetzen muss nun eine Klasse geschrieben werden, die zwei dieser Filter verwaltet und Methodenaufrufe entsprechend delegiert. Dies kann wieder auf verschiedene Arten geschehen. Eine Möglichkeit wäre es, eine Art Provider für den Filter zu entwerfen. An diesem müssten dann Verwendungsstellen den Filter abfragen und würden, je nach Status der Anwendung, den Länder- oder den Fahrzeugfilter zurückbekommen. Da viele Verwendungsstellen den Filter jedoch zwischenspeichern und mit dem Umbau auf eine Providerstruktur auch gleichzeitig die Verwendung eines einzelnen Filters (Anwendungsfall 1 und 2) komplizierter werden würde, ist diese Lösung eher ungeeignet.

Als Alternative dazu kann man einen neuen Filter entwerfen, der die anderen beiden Filter verwaltet. Für die Realisierung muss als Erstes der bestehende Filter abstrahiert werden. Sämtliche Methoden der Schnittstelle des Filters müssen in die Oberklasse verschoben werden. Für diesen Fall bietet es sich an eine Abstrakte Klasse zu konstruieren, von der der bestehende Filter ableitet und die public Methoden der Schnittstelle überschreibt. Diese neue Klasse wird *AbstractFilter* genannt und bekommt dieselben generischen Typen zugewiesen wie die Klasse Filter. In den Verwendungsstellen muss nur (außer bei dem Konstruktoraufruf) „Filter“ durch „AbstractFilter“ ersetzt werden und das Programm funktioniert genauso wie zuvor. Für die neue Funktionalität wird eine weitere Klasse eingeführt, die ebenfalls von dem AbstractFilter ableitet und zwei konkrete Instanzen der Filter-Klasse (Länder- und Fahrzeugfilter) verwaltet.

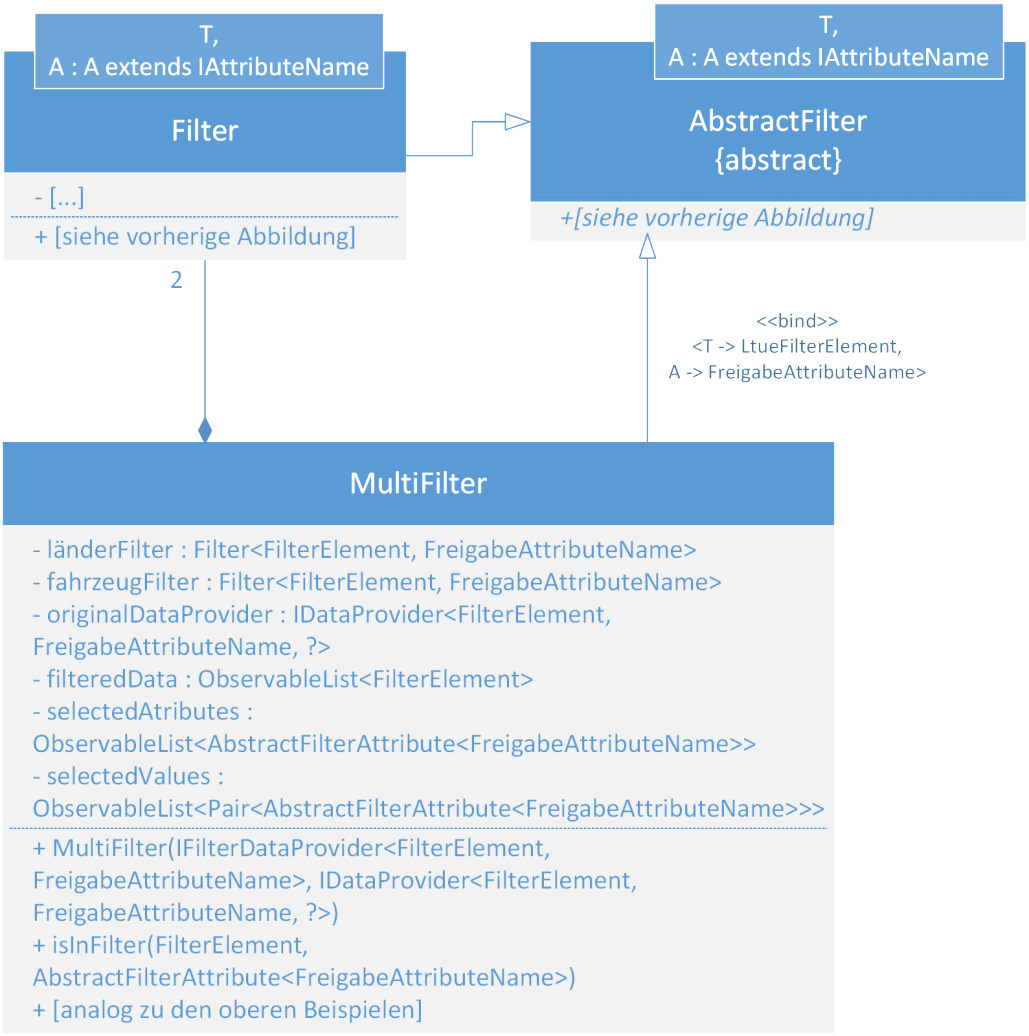


Abbildung 3.7: Klassendiagramm - Konzept Filter

In der Abbildung 3.7 sind die neuen Beziehungen zwischen den Filter-Klassen visualisiert. Um diese Struktur verwenden zu können, muss im 3. Anwendungsfall, statt eines normalen Filters, der neue MultiFilter initialisiert werden. Wie dem Entwurf der *MulitFilter*-Klasse (Abb. 3.7) entnommen werden kann, erwartet der MultiFilter als Modelobjekte die FilterElemente. Diese sind dreiteilig aufgebaut.

1. Der Länderteil
2. Der Fahrzeugteil

3. Der Freigabeteil

Mit Gettern und Settern kann auf diese zugegriffen werden. Die Problematik besteht an dieser Stelle zunächst darin, ein FilterElement einem der zwei Sub-Filter zuzuordnen. Dies wird realisiert, indem die FilterElemente, die in einer einzelnen Liste vom DataProvider zur Verfügung gestellt werden, verschiedene Ausprägungen besitzen. Wie in Kapitel 3.2 beschrieben, werden die Länder- und die Fahrzeugdaten getrennt voneinander geladen. Daher können bereits im DataProvider die FilterElemente verschieden erzeugt werden. Der eine Teil der FilterElemente hat nur die Membervariable für das Land gesetzt, der andere Teil der Objekte nur die Variable für das Fahrzeug. So kann der MultiFilter die Elemente eindeutig einem konkreten Filter zuordnen.

Die beiden einzelnen Filter können daraufhin unabhängig voneinander die Ergebnismenge bestimmen. Diese Mengen werden von dem MultiFilter zu einer Ergebnismenge zusammengeführt. Für diesen Zweck bietet die Klasse FilterElement eine statische Methode *#merge(FilterElement, FilterElement)* an, die ein FilterElement zurückliefert, das sowohl die Ländervariable als auch die Fahrzeugvariable gesetzt hat.

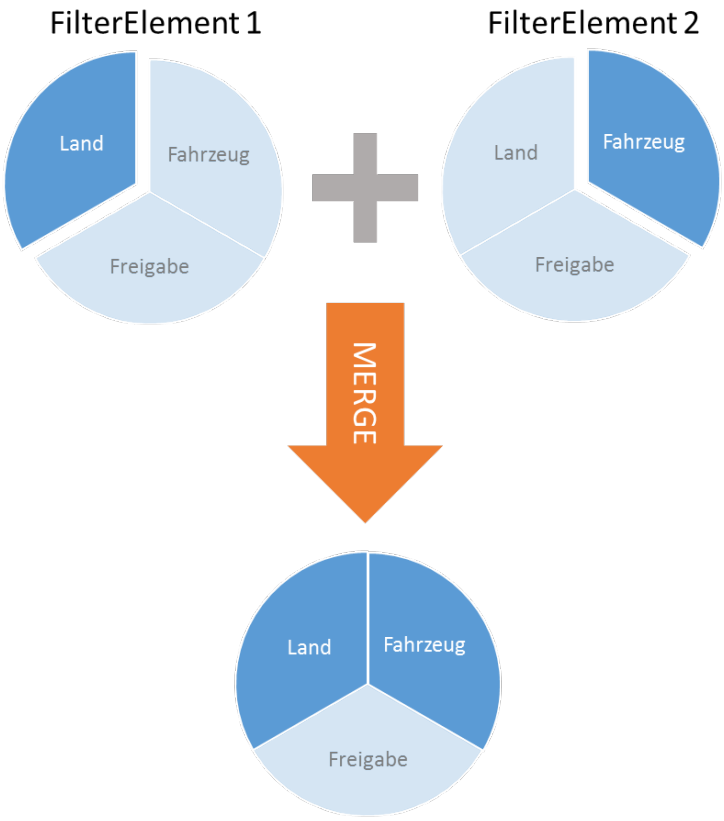


Abbildung 3.8: Schema FilterElement-Aufbau

Ist der Vereinigungsvorgang nicht erfolgreich, weil zum Beispiel zwei Länder-Filter-Elemente übergeben wurden, wird eine *IllegalArgumentException* geworfen. Dieser Vorgang wird zur Erzeugung der Ergebnismenge für die Ergebnisansicht durchlaufen. Dies geschieht für jede mögliche Kombination der Elemente aus der Länder-Ergebnismenge mit den Elementen der Fahrzeugmenge. Es wird so das Kartesische Produkt (Abb. 3.9) der Mengen gebildet.

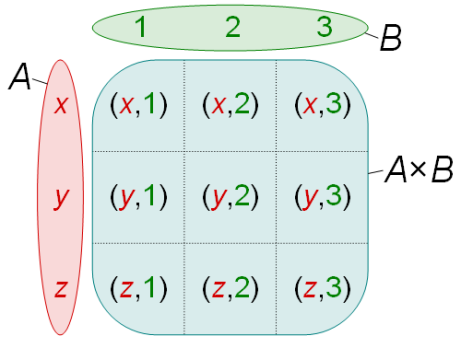


Abbildung 3.9: Kartesisches Produkt [Wik]

Es ist leicht vorstellbar, dass die Elemente der Ergebnismenge und so auch die Anzahl ausgeführter Vereinigungen schnell anwachsen, wenn die Teilmengen größer werden. Die Komplexität dieses Programmteiles ist in der Groß-O-Notation $O(n^2)$. Es gibt aufgrund der Definition des Kartesischen Produktes keinen schnelleren Algorithmus. Daher muss die *#merge(FilterElement, FilterElement)*-Methode möglichst performant implementiert sein und nur ein Minimum an Instruktionen ausführen.

Der letzte Schritt, der für die Erzeugung der endgültigen Ergebnisdaten fehlt, ist das Laden und Setzen der *FreigabeObjekte*. Anhand der erzeugten Ergebnismenge kann nun eine Datenbankabfrage durchgeführt werden, die nur eine eingeschränkte Anzahl an FreigabeObjekten lädt. Dazu wird das Command-Pattern benutzt. Es wird also ein Command erzeugt und mit den Fahrzeug- und Länder-IDs an den Server gesendet. Der Server lädt die Freigabedaten daraufhin aus der Datenbank und schickt sie an den Client zurück. Die Ausführung des Commands und das Warten auf die Server-Antwort geschehen in einem anderen Thread als dem GUI-Thread, damit die Benutzeroberfläche nicht „einfriert“. Stattdessen kann während der Ladezeit ein Ladebalken, oder wie im Falle von FalkoFX, eine benutzerdefinierte Ladeanimation angezeigt werden.

3.4 Tabellenansicht

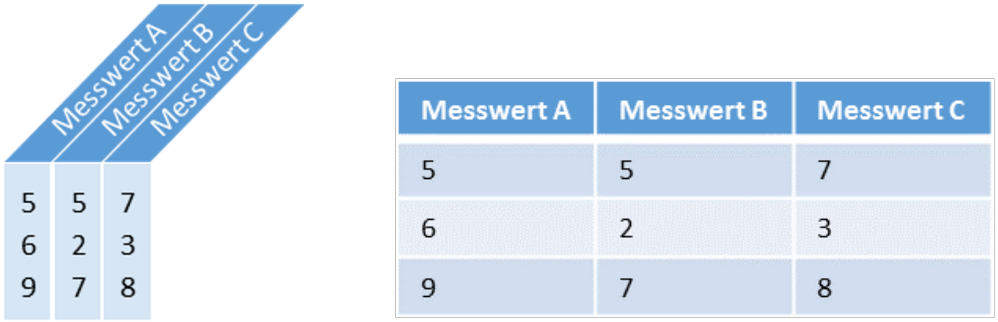
Problemstellung

Die nächste zu bewältigende Anforderung ist die Ergebnispräsentation. Diese soll in einem ersten Schritt als Tabelle erfolgen. An der linken Seite befinden sich als Zeilendeklarator die Fahrzeuge. Für die Spaltenköpfe sind die Länder vorgesehen. Im In-

haltsbereich werden die zu Spalten- und Zeilendeklarator passenden FreigabeObjekte angezeigt.

Konzept

Die einzelnen Bildschirme werden bereits über eine zentrale Klasse geladen. Damit ein Bildschirm geladen werden kann, muss eine Klasse das Interface *IScreen* implementieren. Es stellt die Methoden *#getContent()* und *#initialized()* zur Verfügung. Die erste Methode dient dem Zugriff auf ein UI-Objekt, das in den JavaFX-Szenegraphen eingehängt wird und den Content-Bereich der Anwendung füllt. Unter diesem JavaFX-*Node* können weitere Knoten hängen, die den Aufbau einer komplizierten Benutzeroberfläche ermöglichen. Die zweite Methode dient als Callback und wird ausgeführt, sobald die Benutzeroberfläche initialisiert und angezeigt wurde. In dieser Methode können Aktionen ausprogrammiert werden, die erst nach Anzeige des UIs möglich sind (z.B. das automatische Selektieren einer Zeile).



(a) Schräge Spaltenköpfe (b) Normale Tabelle

Abbildung 3.10: Vergleich: Schräge Spaltenköpfe - Normale Tabelle

Für die Benutzeroberfläche kann eine bereits bestehende Komponente angepasst verwendet werden. Bei der Komponente handelt es sich um eine JavaFX-*TableView*, die für das FalkoFX-Projekt modifiziert wurde. Eine Besonderheit dieser Tabelle ist, dass sie über schräge Spaltenköpfe verfügt. Dies ist deshalb sinnvoll, da sich so die Spaltenbreite nicht an der Länge des Textes im Spaltenkopf orientieren muss, sondern nur an dem Inhalt der Zellen. Wenn man zuvor weiß, dass es Zellen gibt, die nur sehr kurze Inhalte haben (z.B. ein Icon oder eine kurze Zahl), kann man durch die Verwendung von schrägen Spaltenköpfen den verlorenen Platz minimieren.

Wie man anhand der Abbildung 3.10 erkennen kann, nimmt zwar die Höhe der Tabelle zu, die Breite wird jedoch viel effektiver ausgenutzt. Der Platz, der am rechten

Rand durch die schrägen Spaltenköpfe entsteht, ist bei einer höheren Spaltenanzahl vernachlässigbar gering.

Ein weiterer Vorteil der Tabelle ist die mögliche Gruppierung von Spalten. Diese werden visuell von anderen Gruppen abgetrennt und bilden dementsprechend eine geschlossene Einheit (Abb. 3.11).

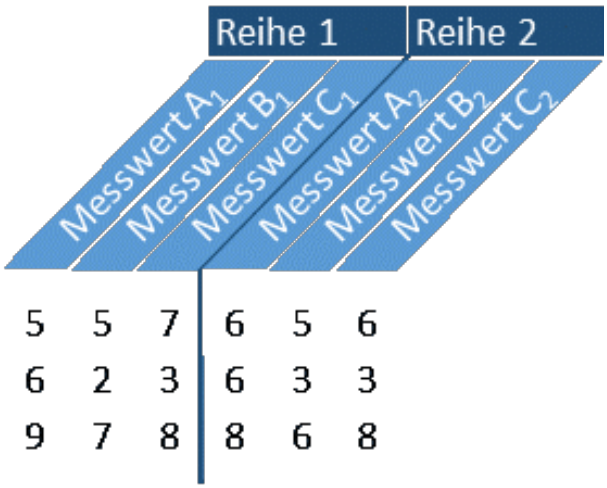


Abbildung 3.11: Schräge Spaltenköpfe mit Gruppierung

Die Tabelle unterstützt außerdem das Ein- und Ausblenden von Spaltengruppen, sowie die Umsortierung dieser. In der Anwendung ist diese Funktion in der Seitenleiste untergebracht.

Die eigentlichen Informationen, die in der Ergebnisansicht präsentiert werden sollen, sind die Freigabedaten für die Fahrzeugproduktion. Um diese Daten zuordnen zu können, muss Im Spaltenkopf das Land angezeigt werden und am Anfang der Zeile das Fahrzeug mit ausgewählten technischen Daten. Die Informationen zu den technischen Daten sind notwendig, da zu einem Fahrzeug mehrere verschiedene Varianten existieren, die sich in mindestens einem Kriterium unterscheiden. Für jedes dieser Fahrzeuge ist die Freigabe eine andere.

Wie bereits erwähnt (Abschnitt 3.2), sind die Hauptinformationen, die man aus einem FreigabeObjekt erhalten kann, das Start- und das Enddatum der Produktion. Daher ist es sinnvoll, diese beiden Daten auf den ersten Blick anzuzeigen. Dies geschieht, wie auch im Originalclient, durch die Angabe von Kalenderwoche und Jahr nach dem Schema „KW/YY“. Daher ist das folgende Layout denkbar:

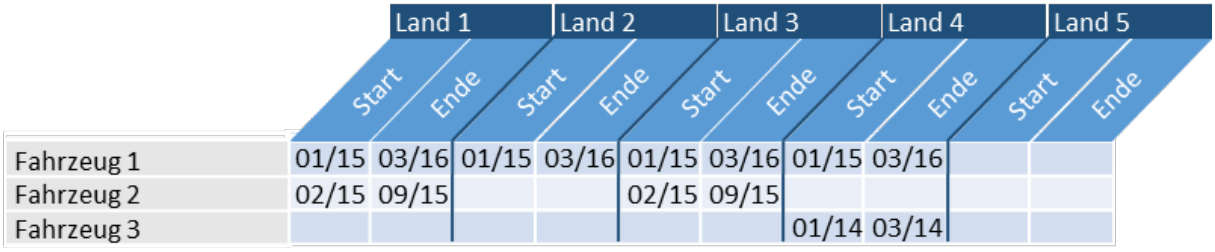


Abbildung 3.12: Design Ergebnistabelle

Umsetzung

Für die Umsetzung ist zunächst die geeignete Benutzeroberfläche erforderlich. Ein Teil der Implementierungsarbeit ist bereits durch die vorhergegangenen Anwendungsfälle absolviert worden. Auch hier gibt es in beiden Fällen eine Tabellenansicht zur ganzheitlichen Darstellung der Ergebnismenge. Dazu gibt es bislang eine mit Generics parametrisierte Klasse, welche die Ergebnisdarstellung verwaltet. So kann die Darstellung für das erste und das zweite Szenario ohne Änderungen an der Klasse genutzt werden. Auch wenn die Tabelle des dritten Anwendungsfalles denen der anderen beiden vom Aussehen stark ähnelt, kann der bereits existierende Code diese Ansicht nicht erzeugen.

Die bisherige Darstellung ist darauf ausgelegt, dass in jeder Zeile genau ein Modelobjekt mit all seinen Attributwerten angezeigt werden soll. Dabei sind die Spalten mit den entsprechenden Attributnamen betitelt. Die Problematik die hier vorliegt ist jedoch eine andere. Für den Fahrzeugbereich, der in Abbildung 3.12 noch grau angezeigt wird, kann das bisherige Prinzip weiter verwendet werden. Es werden Fahrzeugname und einige weitere technische Details angezeigt, für die wiederum jeweils eigene Spalten mit schrägen Spaltenköpfen existieren (auf Abb. 3.12 nicht abgebildet). Es ist sinnvoll, dass dieser Bereich an die Ansichtskonfiguration der Seitenleiste gekoppelt ist, da man ggf. mal mehr, mal weniger Details zu den Fahrzeugen einsehen möchte. Die Ansichtskonfiguration erlaubt es, Spalten einzublenden, auszublenden und neu zu sortieren. Für den Hauptbereich allerdings, in dem die Freigabedaten angezeigt werden, funktioniert das bisherige Prinzip nicht. Die Länder, welche die neuen Spaltengruppierungen bilden, sind keine Attribute der dargestellten Modelobjekte, sondern Werte aus der Teilergebnismenge des Filters (Länder-Subfilter), die sich als Teil der Freigabeobjekte wiederfinden.

Nun könnte man eine neue Ergebnisansicht erstellen, die genau dieses Problem behandelt, dann müsste allerdings noch die linke Seite der Tabelle erzeugt werden – die

Seite mit den Fahrzeugdaten. Um redundanten Code zu vermeiden, bietet sich hier die Möglichkeit an, die Klasse für die Ergebnisansicht zu abstrahieren und eine generische Klasse für den ersten und zweiten Anwendungsfall davon abzuleiten, sowie eine spezifische Klasse für den 3. Anwendungsfall.

Die JavaFX *TableView* lässt pro Tabelle nur eine Art von Modellobjekten zu. Diese Objekte stellen dann immer genau eine Zeile dar. Die Werte für die einzelnen Zellen werden mit Hilfe einer *CellValueFactory* bestimmt, die pro Tabellenspalte gesetzt wird und so den Wert an einem Schnittpunkt von Tabellenzeile und -spalte bestimmen kann - folglich den Wert einer Zelle. Für die Implementierung bedeutet dies, dass die zusammengesetzten FilterElemente nicht direkt aus der Ergebnismenge in die Tabelle gesetzt werden können, sondern gruppiert und für jede Zeile in einem Objekt zusammengefasst werden müssen. Zu diesem Zweck wird die neue Datenstruktur *FreigabeLineObject* eingeführt, die eine beliebige Anzahl an *FreigabeObjekten* enthält, eine Schnittstelle, um auf diese zuzugreifen, sowie eine Möglichkeit das Fahrzeug zu erhalten, nachdem die Werte gruppiert wurden. Aus technischer Sicht wurde die Datenstruktur auf Basis einer *HashMap* aufgebaut, welche mit Hilfe ID als *Key* den Zugriff darauf erlaubt. Die Werte der *HashMap* entsprechen den FilterElementen, welche jeweils ein spezifisches FreigabeObjekt enthalten.

Das Problem, das nun auftritt, ist, dass die Tabellen auf den Modellobjekten (in dem Fall FilterElemente) basieren, es sollen jedoch Elemente vom Typ FreigabeLineObject dargestellt werden. Um dieses Konzept in die vorgestellte Klassenstruktur zu integrieren, muss der *AbstractResultTableController* mit 2 generischen Parametern versehen werden (Abb. 3.13). Der erste Parameter steht für den Typ der Modelelemente, auf denen die Ansicht basiert, der zweite Parameter bestimmt den Typen der Zeilenobjekte, die in der Ergebnistabelle landen. Die Unterklassen spezifizieren diese Parameter daraufhin genauer. Es werden außerdem Methoden benötigt, um aus der Ergebnismenge der Modelobjekte die Zielergebnismenge zu generieren und umgekehrt aus einem Zeilenobjekt ein spezifisches Modelobjekt zu erhalten.

Die generische Klasse *GenericResultTableController* löst das Problem so, dass sie nur noch einen Parameter (*T*) besitzt, der sowohl für das *T* als auch das *R* der Oberklasse steht. Auf diese Weise sind die Modellobjekte gleich den Objekten, die in der Ergebnistabelle pro Zeile angezeigt werden (Abb. 3.13). Bei dem spezifischen *FreigabeResultTableController* werden die Parameter mit den konkreten Klassen FilterElement und FreigabeLineObject überschrieben (Abbildung 3.13).

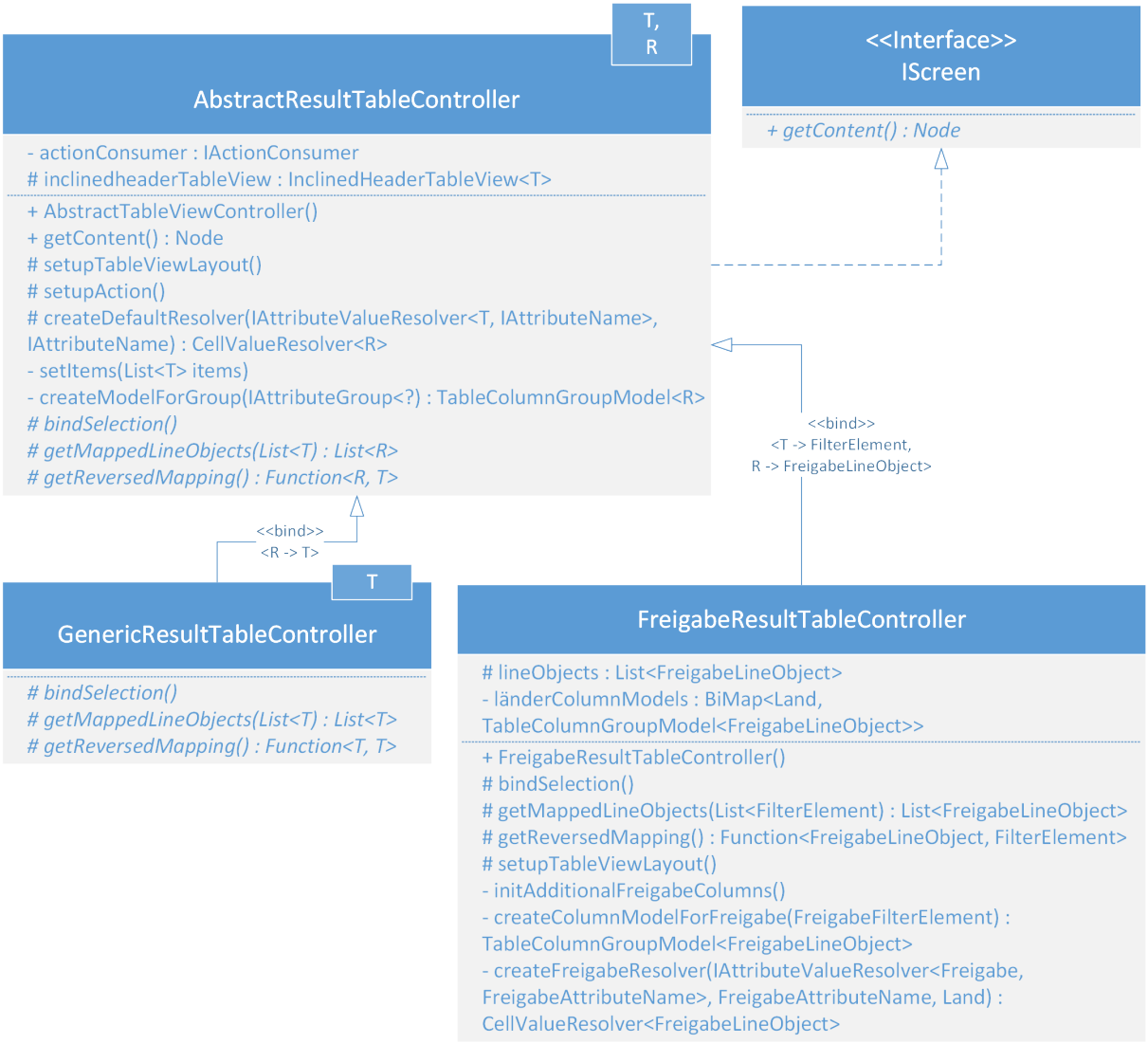


Abbildung 3.13: Klassendiagramm Ergebnisansicht

Für die zu überschreibenden abstrakten Klassen ist die Implementierung des Generic-ResultTableControllers denkbar simpel.

Listing 3.2: GenericResultTableController - Mapping-Methoden

```
public List<T> getMappedLineObjects(List<T> data) {
   return data;
}

public Function<T, T> getReversedMapping() {
   return Function.identity();
}
```

Bei dem FreigabeResultController fällt die Umsetzung etwas komplizierter aus. Bereits beim Konstruktoraufruf werden die Modelobjekte der Ergebnismenge in die Zeilenobjekte „verpackt“. Zu diesem Zweck gibt es eine Membervariable vom Typ *List<FreigabeLineObject>*. Diese Liste wird befüllt, indem für jedes Element aus der Ergebnismenge zunächst ein Zeilenobjekt erstellt wird, das dieses Element enthält. Daraufhin wird mit der *List#contains(Object o)*- Methode überprüft, ob dieses Element schon in der Liste vorhanden ist. Dafür wird die *#equals()*-Methode der Zeilenobjekte aufgerufen. Da diese überschrieben wurde, liefert sie genau dann *true* zurück, wenn die ID des zugeordneten Fahrzeuges bei beiden Zeilenobjekten gleich ist. Ist das der Fall, kann das überprüfte FilterElement dem bestehenden hinzugefügt werden, andernfalls wird das erzeugte FreigabeLineObject in die Liste der Zeilenobjekte eingegliedert. Die #getMappedLineObjects()-Methode kann diese Liste nun sortiert zurückgeben.

Der Vorteil daran, dass die Liste schon zuvor beim Konstruktoraufruf erzeugt wird, und nicht erst bei Bedarf, ist der, dass die Benutzeroberfläche nicht „einfriert“. Die Methode *#getMappedLineObjects()* wird nach dem Aufbauen der Benutzeroberfläche im GUI-Thread ausgeführt. Wenn die Liste schon vorher erzeugt wurde, fällt an dieser Stelle der Großteil des Rechenaufwandes weg und der Aufbau der Oberfläche erscheint flüssiger.

Während der Konstruktionsphase werden, neben dem Tabellenlayout für die Spalten der Fahrzeugattribute, auch die zusätzlichen Freigabe-Spalten erzeugt, die nach Ländern gruppiert sind und das Start- und Enddatum als konkrete Spalten aufweisen. Auch hier werden wieder die Ergebnisdaten benötigt.

Als erster Schritt wird eine *Map* erzeugt, die für jedes Land aus der Ergebnismenge ein *ColumnGroupModel* verwaltet. Als technische Ausprägung dieser Map wird eine *HashBiMap* aus der *Google Guava* Library verwendet. Die Besonderheit und der Zweck dieser Datenstruktur werden später erläutert. Durch Java 8 - Features kann der grundlegende Code für die Erzeugung der Map relativ gering gehalten werden.

Listing 3.3: Erzeugung der BiMap

```
countryColumnModels = HashBiMap.create(
      ucContext.getResultData().stream()
      .collect(Collectors.toMap(
            freigabe -> freigabe.getLand(), // Erzeugung des Keys
            this::createColumnModelForFreigabe, // Erzeugung des Wertes
            (o1, o2) -> o1 // Merge-Function, falls 2 Keys gleich
      ))
);
```

Die Methode *#createColumnModelForFreigabe()* erstellt auf Basis eines beliebigen FilterElements eine Spaltengruppe mit den beiden Spalten für das Start- und Enddatum der Produktionsfreigabe. Dazu gehört eine CellValueFactory, die für eine Zelle dieser Spalte anhand des darunterliegenden Zeilenobjektes den Wert dieser Zelle bestimmen kann. Beim Erstellen der CellValueFactory wird die Länderinformation aus dem FilterElement übergeben. Anhand der Länder-ID kann dann der richtige Wert für eine Zelle aus dem Zeilenobjekt ermittelt werden. Schlussendlich müssen die Spaltengruppen aus der Map nur noch sortiert werden und der angepassten TableView hinzugefügt werden.

In Abbildung 3.13 wird ersichtlich, dass der FreigabeResultTableController über eine weitere Methode verfügt, die hier von Relevanz ist – die *#bindSelection()*-Methode. Diese sorgt dafür, dass das selektierte Modelobjekt, das global (im sogenannten *Context*) verfügbar ist, aktuell gehalten wird. An diese Selektion ist beispielsweise die Sidebar gebunden, in der es eine Ergebnisvorschau gibt, die das derzeit angewählte Element farblich differenziert darstellt. Um dies in der komplexen Tabelle des 3. Anwendungsfalles zu ermöglichen, wird an die JavaFX-Property der selektierten Zellen ein Listener angehängt. Darin wird die *TablePosition* der Selektion bestimmt, durch welche die Spalte bestimmt werden kann, in der die Selektion stattgefunden hat. Problematisch ist es jetzt jedoch, das eigentliche Modellobjekt (vom Typ FilterElement) zu identifizieren, da über die Selektion nur das Zeilenobjekt erhalten werden kann. Zur Erinnerung: Um von einem FreigabeLineObject auf ein FilterElement schließen zu können, wird das Länderobjekt benötigt. An dieser Stelle kommt die zuvor erwähnte HashBiMap ins Spiel. Diese hat nämlich die Eigenheit, dass sowohl die Keys als auch die Values eindeutig zu identifizieren sind. Es entsteht also eine eineindeutige Beziehung zwischen den Objekten. Aus diesem Grund lassen sich die eigentlichen Values auch als Keys verwenden. Durch den Aufruf *#inverse()* auf der *BiMap* wird diese invertiert und die vorherigen Keys werden zu Values und umgekehrt. Durch die zuvor ermittelte Spalte kann das dazugehörige Spaltengruppenmodell identifiziert werden und dadurch das

zugehörige Land (aus der invertierten BiMap). Mit Hilfe des Landes lässt sich nun auf das Modellobjekt schließen, das daraufhin im Context aktualisiert wird.

Die letzte verbliebene Methode ist *#setupAction()*. Der Zweck dieser Methode ist das Ermöglichen von Interaktionen mit der Tabelle. Zum Beispiel kann bei einem Doppelklick eine Detailansicht zu einem selektierten Modellobjekt angezeigt werden.

Zur größeren Übersichtlichkeit bei der Navigation durch die umfangreiche Tabelle, wird neben der Zelle, die derzeit selektiert ist, auch die Spalte in einem helleren Farbton mit eingefärbt. So lässt sich schnell das zu einer Freigabe gehörende Fahrzeug identifizieren. Aufgrund der Tatsache, dass eine JavaFX TableView nur entweder Zeilenselektion oder Zellselektion nativ unterstützt, musste eine andere Lösung für das Hervorheben gefunden werden. Diese bestand in dem Setzen einer selbst definierten CSS-Pseudoklasse, die genau dann auf eine Zeile angewandt wird, wenn Zellselektion aktiviert ist und eine Zelle aus der betroffenen Zeile selektiert wurde. Andernfalls würden die Standard-CSS-Definitionen für Tabellenzeilen angewandt werden.

4 Analyse ausgewählter Usability-Probleme

4.1 Präsentation 2-teiliger Texte in ListCells

Situation

In einer Liste werden Texte angezeigt, die aus zwei Teilen bestehen. In einem Fall ist der erste Teil ein Schlüssel zur Identifizierung eines Fahrzeuges, der zweite Teil ist der Name des Fahrzeuges. Ein Fahrzeugname kann unter Umständen mehr als einmal auftreten, der Fahrzeugschlüssel hingegen ist eindeutig. Während die Schlüssel sich in der Zeichenanzahl nur um zwei bis drei Zeichen unterscheiden, können die Fahrzeugnamen ganz unterschiedliche Längen haben. Derzeit werden die beiden Textbausteine schlicht aneinandergereiht. Dies führt bei längeren Listen schnell zu Unübersichtlichkeit. Daher sollte eine sichtbare Trennung zwischen den beiden Elementen erfolgen.

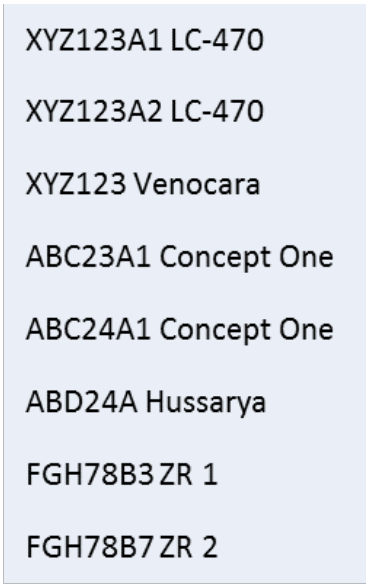


Abbildung 4.1: Liste: Ausgangszustand

(a) Design 1 (b) Design 2 (c) Design 3

Abbildung 4.2: Design-Vorschläge

Alle Design-Vorschläge sind valide Möglichkeiten zur Lösung des Problems, haben jedoch auch ihre Vor- und Nachteile.

Der erste Designvorschlag (Abb. 4.2(a)) stellt die beiden Teile des Textes sehr gut optisch getrennt dar. Beide Teile sind tabellenartig linksbündig angeordnet. Die ID-Teile aller Texte sind dabei auf einer Höhe, genauso wie die Namensteile. Auf diese Weise entstehen zwei Spalten, in denen die Textbausteine dargestellt werden.

Bei dem zweiten Designvorschlag (Abbildung 4.2(b)) ist der linke Teil des Textes fett gedruckt, der andere Teil in ganz normalem Stil. Die zweite Lösung setzt zwar die Trennung nicht so sauber wie Vorschlag 1 um, stellt aber so die Fachlichkeit besser dar. Die Fahrzeug ID und der Fahrzeugname sind eine zusammengehörige Einheit. Nach dem Gestaltgesetz der Nähe (Abschnitt 2.5.1) würden bei Designvorschlag 1 die Fahrzeugschlüssel als eine Objektgruppe und die Fahrzeugnamen als eine Objektgruppe gesehen werden. Da dies nicht die Intention der Abbildung ist, eignet sich die zweite Lösung besser für die Lösung des Problems.

Der dritte Vorschlag gruppiert die Elemente der Liste zwar ebenso wie in Abbildung 4.2(b), ist jedoch eine eher exotische Darstellungsweise. Hier sind die Elemente an einer gedachten senkrechten Linie zwischen den beiden Teilen angeordnet. Durch die unterschiedliche Bündigkeit entsteht der Eindruck der Gruppierung pro Element, ebenso wie in Vorschlag 2. Durch das Fettdrucken der linken Spalte wird diese dazu hervorgehoben. Das Problem dieser Darstellungsweise ist jedoch der Stilbruch zu den anderen Listendesigns, die im gleichen Bereich angezeigt werden können. Die anderen darstellbaren Listen beinhalten größtenteils keine 2-teiligen Texte. Auf diese Standardvariante der Listen könnte das dritte Design nicht angewandt werden. Es würde daher das Gesetz der Ähnlichkeit (Abschnitt 2.5.1) verletzt werden und sich so unter Umständen unangenehm auf das Nutzerempfinden auswirken.

4.2 Filter-Performance

Die Änderungen, die in Kapitel 3.1 eingeführt wurden, funktionieren zwar aus technischer Sicht, lassen den Filter aber bei bestimmten Operationen langsam arbeiten, was zu unbequemen Verzögerungen führt.

Analyse

Um die Laufzeitprobleme beheben zu können, mussten als erstes die Fehler gefunden werden, die zu besagten Problemen führten. Bei der Analyse wurden die folgenden Problemfaktoren untersucht:

- Zeitintensive Anweisung (möglicherweise in Schleifen ausgeführt?)

- Mehrfach hinzugefügte JavaFX-Listener an ObservableLists
- Schleifen mit hohen Durchlaufzahlen
- Konstruktion von vielen, großen Objekten

Für die Überprüfung des ersten Faktors mussten alle Anweisungen einzeln angeschaut werden. Wenn ein Funktionsaufruf dabei ist, der viele Berechnungen ausführt, kann die Ausführungszeit des Aufrufes dadurch gemessen werden, dass der Zeitpunkt vor der Ausführung gespeichert wird und nach der Ausführung von der dann aktuellen Zeit abgezogen wird. Benutzt dafür den Aufruf System*#currentTimeMillis()* erhält man die Zeitspanne in Millisekunden. Auf diese Weise ließ sich kein performancekritischer Methodenaufruf feststellen. Auch unter Berücksichtigung, dass manche Methoden in Schleifen ausgeführt werden, fand sich keine kritische Stelle.

Die zweite Überprüfung war erfolgreicher. Tatsächlich wurden zwar keine Listener mehrfach an einer ObservableList angehängt, aber pro Werteliste im Filter (in diesem Fall 13) wurden jeweils einige Listener ausgelöst, die diese aktualisieren sollten, wenn sich die ausgewählten Werte geändert haben. Diese Berechnungen erforderten einigen Aufwand und dadurch, dass sie, statt einmal, 13-mal ausgeführt wurden, sorgten sie für eine negative Laufzeitbeeinflussung.

Diese Lösung dieses Problems alleine brachte jedoch nicht den gewünschten Effekt. Eine Methode, die extrem oft ausgeführt wurde und für die meisten Filterberechnungen relevant ist, ist die Methode, die in der Menge der selektierten Werte sucht, ob eine Attribut-Werte-Kombination vorhanden ist.

Listing 4.1: Aufbau - selectedValues

```
ObservableList<Pair<AbstractFilterAttribute<A>, Object>> selectedValues;
```

Als Parameter erhält die Methode ein Attribut und einen Wert, der überprüft werden soll. Die aufgerufene benutzt die *#contains(Object)*-Methode der List-Implementierung. Dafür muss für jede Überprüfung ein Objekt vom Typ *Pair* erzeugt werden, das das Attribut und den zu überprüfenden Wert kapselt. Zudem muss danach (durch die Implementierung von *#contains(Object))* jedes Objekt der Liste durchlaufen werden und per *#equals(Object)* verglichen werden. Je größer die Menge der ausgewählten Werte wird, desto öfter muss ein solcher Vergleich ausgeführt werden. Die Laufzeit eines Durchlaufes über diese Datenstruktur $O(n)$. Bei einzelnen Aufrufen würde dies nicht weiter problematisch werden, doch durch den neuartigen Filter wird diese Methode nun weitaus häufiger ausgeführt.

4.3 „Quellcode-Usability“

Die Java 8 - Sprachfeatures sind ein gutes Mittel, vor Allem in Kombination mit JavaFX und der Stream API, Code prägnant und sehr kurz darzustellen. Doch haben die neuen Features, vornehmlich Lambdas auch die Eigenschaft, Code schnell zu verkomplizieren, wenn sie im Überfluss verwendet werden. Der durch die Definition bestimmte Nachteil eines Lambda-Ausdruckes ist, dass nicht sofort erkennbar ist, welches Interface implementiert wird und was die Parameter zu bedeuten haben. Dies lässt nicht nur ungeübte Java 8 –Programmierer an einigen Codestellen verzweifeln. Ein Konstrukt, das im Quellcode in einer noch ausgeprägteren Variante zu finden war, wird folgend umrissen.

Listing 4.2: Code-Qualität - Gegenbeispiel

```
boolen hasSubLevels = level.getListEntries().stream().filter(
      entry -> entry instanceof ILevel).count() > 0;
entryList.setCellFactory(list -> new ListCell<IListEntry>() {   //1
   { setPrefWidth(0); }                                          //2
   @Override
   protected void updateItem(IListEntry item, boolean empty) {
      super.updateItem(item, empty);
      [...]
      FalkoArrow arrow = new FalkoArrow(Direction.LEFT);
      arrow.setOpacity(item instanceof ILevel ? 1.0 : 0.0);    //3
      [...]
      HBox hbox = new HBox(5.0);
      if (hasSubLevels) {
         hbox.getChildren().add(arrow);
      }
      [...]
      setOnMouseClicked((v) -> doCallback(item));           //4
      setCursor(empty ? Cursor.DEFAULT : Cursor.HAND);
   }
});
```

Der Code ist zwar sehr knapp aber führt selbst mit Syntax-Coloring schnell zur Unübersichtlichkeit. Dieser Effekt wird dadurch verstärkt, dass das selten verwendete Konstrukt des Instanz-Initializers (2), kombiniert mit einer Anonymen Inneren Klasse als Rückgabewert eines Lambdas eingesetzt wird (1). In dieser Klasse werden zusätzlich ternäre Ausdrücke als Alternative zu If-Klauseln (3) und weitere Lambdas (4)

benutzt. Zudem hat die Anonyme Innere Klasse Referenzen in den umschließenden Block (Variable *hasSubLevels*).

Ein weiteres Beispiel hängt mit der Verwendung des Double-Colon-Operators zusammen:

Listing 4.3: Code-Qualität - Gegenbeispiel 2

```
protected StringProperty specialTextProperty = new
    SimpleStringProperty("[all values]");
protected final IListEntry specialEntry = specialTextProperty::get;
```

Auch über diesen kurzen Code-Ausschnitt kann sich schnell der Kopf zerbrochen werden. Ein JavaFX StringProperty hat nur die Funktion, einen String-Wert zu kapseln und den Zugriff darauf zu erlauben - über die Funktion *#get()* und *#set(String)*. Warum ist der obenstehende Code nicht fehlerhaft? Liefert *#get()* in diesem Falle doch keinen String zurück? Es sieht in der Tat für ungeübte Java 8 -Entwickler so aus, als würde der Variable *specialEntry* ein String zugewiesen werden. Doch hier handelt es sich um die Kurzschreibweise eines Lambda-Ausdruckes. Die betroffene Zeile würde wie folgt in einen „echten“ Lambda übersetzt werden:

Listing 4.4: Code-Qualität - Verbesserung

```
protected final IListEntry specialEntry = () -> specialTextProperty.get();
```

Um den Sinn genauer verstehen zu können, muss das Interface IListEntry genauer angeschaut werden. Hier befinden sich zwei Methoden. Eine Default Implementierung für die Methode *#getText()* und die abstrakte Methode *#getValue()*, die die String-Repräsentation von *#getValue()* zurückliefert. Es muss also die Methode *#getValue()* des funktionalen Interfaces implementiert werden, damit eine vollständige Klasse entsteht. Genau das machen beide die beiden Code-Ausschnitte 4.3 und 4.4. Dennoch ist die Schreibweise zunächst unverständlich.

Dies sind nur zwei Beispiele für schwierig lesbaren Code, im Laufe eines Java 8 -Projektes können viele solcher und ähnlicher Probleme auftreten.

5 Implementierung der Usability-Verbesserungen

5.1 Präsentation 2-teiliger Texte in ListCells

Die Umsetzung des zweiten Designvorschlages ist nicht so einfach möglich, wie die Grafik zunächst vermuten lässt. Es ist standardmäßig nicht möglich den Text, der in dem Textbereich einer ListCell eingefügt werden soll zu teilen und zwei verschiedene Styles (Per CSS-Klasse) auf die Teile des Textes anzuwenden.

Wenn man nicht die ListCell-Klasse erweitern und stark verändern möchte, bleibt nur die Möglichkeit, über die Graphic, die gesetzt werden kann, einen Text einzufügen. Die Graphic entspricht hier nicht einer Grafik im allgemein gebräuchlichen Sinne, sondern einem JavaFX Node, der eine beliebige Ausprägung haben kann. So könnten in der Theorie auch sehr komplexe Elemente als Listenelemente dargestellt werden.

Der Text, der in den Listenzellen angezeigt werden soll, erhält die Klasse aus dem DataProvider. Dieser fügt die Teile der Texte bisher zusammen und liefert sie als einen einzelnen String zurück. Bei der Präsentation werden jedoch Teilstrings benötigt. Nun könnte der Text geparst werden und daraufhin wieder aufgespalten, allerdings wäre dieses Vorgehen nicht sinnvoll, da die gleiche Listenzelle auch für Texte funktionieren soll, die nicht aufgeteilt sind. Möglicherweise würde der Text in dem Fall sogar fälschlicherweise fettgedruckt werden. Aus diesem Grund müssen die Texte von vorneherein als Mehrteiliger Text aus dem DataProvider geliefert werden. Die nächstliegende Lösung ist es, ein Array zu benutzen, das die Textteile speichert. Aufgrund der Tatsache, dass die Rückgabewerte aus dem DataProvider auch an anderen Stellen verwendet werden, muss sich das zurückgegebene Objekt nach außen hin wie ein String verhalten. Dies kann dadurch erreicht werden, dass das Array in ein neues Objekt verpackt wird, welches die *#toString()*-, *#hashCode()*- und *#equals(Object)*- Methoden überschreibt und die einzelnen Textteile zuvor zu einem String zusammenfügt.

Die Listenzelle muss jetzt nur noch überprüfen, ob es sich bei dem zurückgelieferten Wert um ein Objekt der Klasse *MultiPartString* handelt und kann daraufhin den ersten Teil des Textes anders darstellen.

Für das korrekte Layout wird eine HBox verwendet, deren innerer Abstand zwischen Elementen so eingestellt ist, dass die Teile wie ein einziger Fließtext wirken.

5.2 Filter-Performance

Behebung: Listenaktualisierung

Um die Menge an gleichzeitigen Listenaktualisierungen zu reduzieren muss zunächst definiert werden, welche Listen zu welchem Zeitpunkt aktualisiert werden müssen. Das ist zunächst die Liste, die zum aktuellen Zeitpunkt angezeigt wird - die anderen Listen sind dann zwar zeitweise veraltet, dies fällt dem Nutzer jedoch nicht auf, da er die Listen nicht sehen kann. Allerdings muss dann jedes Mal, wenn eine andere Werteliste angezeigt werden soll, diese Liste ebenfalls aktualisiert werden, bevor sie dem Nutzer angezeigt wird.

Das ganze kann nur ermöglicht werden, wenn eine Koppelung mit der Benutzeroberfläche stattfindet. Jedes Mal, wenn der Nutzer ein anderes Attribut auswählt, um die dazu passende Werteliste zu erhalten, wird eine Methode im Filter ausgeführt, die als Parameter das jetzt aktive Attribut erhält. Die dazu gehörige Werteliste wird dann einmal erneuert und nur diese durch den Listener aktuell gehalten.

Behebung: Zugriff auf selektierte Werte langsam

Das Problem der Menge der selektierten Werte musste auf Ebene der Datenstruktur angegangen werden. Je nachdem, wie viele Werte bereits selektiert waren, konnten sich die Laufzeiten für die Suche in dieser Struktur auf insgesamt bis zu mehrere Sekunden verlangsamen (kumulierter Gesamtwert aller Ausführungen, die bei einer Neuselektion eines Attributwertes nötig sind).

Für den schnelleren Zugriff wurde eine weitere Datenstruktur eingeführt, die eine andere Repräsentation der selektierten Werte darstellt. Durch eine HashBasedTable, einer Struktur aus der *Google Guava* Bibliothek, war es möglich, die Laufzeitkomplexität für einen Zugriff auf die selektierten Werte von $O(n)$ auf $O(1)$ zu verkürzen. Die Struktur ähnelt einer HashMap, arbeitet jedoch auf zwei Dimensionen und bietet die Möglichkeit, 2 Keys zu verwenden, um einen Wert zu erhalten. Da die Zeitdauer zum Auffinden

eines solchen Keys in beiden Dimensionen der HashBasedTable konstant ist, ist ein sehr schneller Zugriff auf Elemente möglich.

Listing 5.1: Datenstruktur - FastSelectedValues

```
private HashBasedTable<AbstractFilterAttribute<A>, Object, Object>
    fastSelectedValues;
```

In obigem Code-Beispiel wird die Zusammensetzung der neuen Datenstruktur gezeigt. Der erste generische Parameter steht für das Attribut, der zweite für den gesuchten Attributwert. Der dritte Parameter wird nicht verwendet und enthält nur der Boolean-Wert *true*, damit ein Objekt für das Key-Paar vorhanden ist. Die Existenz eines Key-Paares kann dann mit *#contains(Object, Object)* überprüft werden. Die Laufzeit verringert sich dadurch von anfänglich mehreren Sekunden auf wenige Millisekunden. Um unnötigen Rechenaufwand zu vermeiden, wird die Struktur nur einmal aktualisiert, sobald sich die selektierten Werte verändert haben. Ein weiterer Vorteil ist, dass die Erzeugung von vielen Objekten des Typs *Pair* wegfällt.

5.3 „Quellcode-Usability“

Um das erste in Kapitel 4.3 dargestellte Usability-Problem zu lösen, kann zum Beispiel die Anonyme Innere Klasse extrahiert werden und als eigenständige Klasse in einer eigenen Datei implementiert werden. Dadurch fiele außerdem der Instance Initializer weg, dessen Code stattdessen im Konstruktor ausgeführt werden kann. Referenzen in den übergeordneten Scope werden dem Konstruktoraufruf als Parameter übergeben.

Das zweite Problem erübrigt sich, wenn auf den Lambda-Ausdruck verzichtet und stattdessen eine Anonyme Innere Klasse implementiert werden würde.

Das Problem der Quellcode-Usability ist jedoch ein grundlegenderes. Zur Lösung müssen Richtlinien festgelegt werden, wann, in welchem Maße und in welchen Kombinationen Lambdas und andere Sprachfeatures verwendet werden dürfen.

Einige mögliche Richtlinien sind zum Beispiel folgende:

- Keine Anonymen Inneren Klassen innerhalb von Lambda-Ausdrücken verwenden
- Lambdas in maximal 2 bis 3 Ebenen schachteln

- Bei Schachtelung von Lambdas stets durch Einrückungen kennzeichnen
- DoubleColon-Operator nur in selbsterklärenden Fällen verwenden
- Parameter in Lambda-Ausdrücken nach Typ sprechend bezeichnen

Die Liste ist weit entfernt davon, komplett zu sein, aber vermittelt dennoch einen kurzen Eindruck, wie Lese- und Verständnisschwierigkeiten im Quellcode vermieden werden können.

6 Ausblick und Fazit

Durch die Entwicklung ist das Projekt einen großen Schritt in Richtung Release vorangekommen. Die Weiterentwicklung der bestehenden Funktionen verbessert nicht nur die Gebrauchstauglichkeit der Anwendung, sondern eröffnet dem Endanwender auch ganz neue Möglichkeiten, die Software zu verwenden. Die Intention hinter dem Produkt ist durch die Entwicklung weiter verfolgt worden und so wird dem Nutzer eine zwar eingeschränkte, aber weitaus übersichtlichere Möglichkeit dargeboten, Daten aus dem Falko-System zu betrachten und auszuwerten. Dennoch sind noch einige Arbeiten vorzunehmen, bevor von einem vollständigen Funktionsumfang gesprochen werden kann und gerade im Kernaspekt der Anwendung, der Gebrauchstauglichkeit, besteht noch viel Evaluations- und Optimierungsbedarf.

Literaturverzeichnis

Beispielsweise Hinweis zur Sortierung des Literaturverzeichnisses.

[Com] *Command Pattern*. http://www.oodesign.com/command-pattern.html, Abruf: 06.10.2015

[DIN] *DIN EN ISO 9241: Ergonomie der Mensch-System-Interaktion - Teil 110: Grundsätze der Dialoggestaltung*

[FFSB06] FREEMAN, Eric ; FREEMAN, Elisabeth ; SIERRA, Kathy ; BATES, Bert: *Entwurfsmuster von Kopf bis Fuß*. O'Reilly Germany, 2006

[Gri] GRIGO, Anja: *Wahrnehmungsgesetze*. http://www.orange-sinne.de/wahrnehmungsgesetze.html, Abruf: 06.10.2015

[Hau10] HAUER, Philipp: *Das Command Design Pattern*. http://philipphauer.de/study/se/design-pattern/command.php. Version: 2010, Abruf: 06.10.2015

[Hom13] HOMMEL, Scott: *Using JavaFX Properties and Binding*. https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm. Version: 06 2013, Abruf: 01.10.2015

[Jia] JIANG, Xiaoyi: *Komplexität von Algorithmen*. http://cvpr.uni-muenster.de/teaching/ws08/info1WS08/script/Kapitel6-Complexity-1.pdf, Abruf: 01.10.2015

[Mos12] MOSER, Christian: *User Experience Design: Mit erlebniszentrierter Softwareentwicklung zu Produkten, die begeistern*. Springer-Verlag, 2012

[Mül15] MÜLLER, Björn: *JavaFX im Einsatz*. http://www.informatik-aktuell.de/entwicklung/programmiersprachen/javafx-der-nachfolger-von-java-swing-im-einsatz.html. Version: 01 2015, Abruf: 06.10.2015

[Ora] *Lambda Expressions*. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html, Abruf: 06.10.2015

[Oxf] *UML, Abstract Classes and Methods, and Interfaces*. www.oxfordmathcenter.com/drupal7/node/35, Abruf: 01.10.2015

[Sch] SCHOSSMANN, Oliver: *Gestalt-Erkennungs-Gesetze*. http://www.netface.de/die-gestalt/gestalt-erkennungs-gesetze.html, Abruf: 06.10.2015

[Ull14] ULLENBOOM, Christian: *Java SE 8 Standard-Bibliothek: Das Handbuch für Entwickler*. Rheinwerk Verlag GmbH, 2014

[Urm14] URMA, Raoul-Gabriel: *Processing Data with JavaSE 8 Streams, Pt. 1*. http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html. Version: 03 2014, Abruf: 06.10.2015

[Wik] *Kartesisches Produkt*. https://de.wikipedia.org/wiki/Kartesisches_Produkt, Abruf: 06.10.2015

[Wir] WIRTH, Thomas: *Die Gesetze der Nähe und Ähnlichkeit*. http://www.kommdesign.de/texte/gestaltpsychologie1.htm, Abruf: 06.10.2015

[Zel] ZELLER, Andreas: *Entwurfsmuster*. https://www.st.cs.uni-saarland.de/edu/einst/06-patterns.pdf, Abruf: 06.10.2015

Abbildungsverzeichnis

Glossar

A

Attribut siehe Filterattribut. 9, 14–22, 24, 32

B

Binding Bindet zwei ObservableValues aneinander - bei Veränderung des einen Wertes wird der andere mit verändert. 8

C

Command-Pattern Design-Pattern zum Ausführen von Methoden zu einem beliebigen Zeitpunkt nach Erzeugung eines *Commands*. 12, 29

CSS Cascading Style Sheets: Beschreibungssprache für visuelle Präsentation von UI-Elementen. 8, 37

D

DataProvider Java-Klasse zum Bereitstellen und Laden der Rohdaten. 18, 20–22, 24, 27

Default Implementierung Standard-Implementierung für Methoden innerhalb von Interfaces. 3

Design-Pattern Etablierte Programmierkonzepte zur Lösung von spezifischen Problemen der Code-Strukturierung. 1, 11

E

Ergebnismenge Menge der Modelobjekte, die nach Filterung übrig geblieben sind. 10, 14–19, 23, 27–29, 32, 33, 35

F

Falko Eine Anwendung zur Visualisierung, Verwaltung und Editierung von länderspezifischen Fahrzeugdaten. 9

FalkoFX Eine Anwendung zur Visualisierung von länderspezifischen Fahrzeugdaten. 1, 9, 29, 30

Filter Konstrukt zum Einschränken und Verarbeiten der rohdaten zu einer Ergebnismenge. 9, 10, 14, 15, 17, 18, 22–27, 32

Filterattribut Attribut zur Gruppierung von Filterwerten im Filter. 17, 18

FilterElement Pseudo-Modelobjekte, um ein Land und ein Fahrzeug für eine Freigabe zu k apseln. 22, 26, 27, 33, 35, 36

Filterwert Wert zur Berechnung der Ergebnismenge im Filter. 14, 19, 23

FreigabeObjekt Modelobjekt der Produktionsfreigabe. 29–31, 33

Functional Interface Interface bei dem nur eine Methode implementiert werden muss. 3, 4, 6

G

Galerie Selbstentwickelte Komponente zur anschaulichen Präsentation von Daten. 10

Gebrauchstauglichkeit siehe Usability. 1, 17

Getter Funktion, die den Wert einer Membervariablen zurückgibt. 27

GUI Graphical User Interface: *Benutzeroberfläche*. 8, 29

J

JavaFX Neuartiges GUI-Toolkit zur Oberflächengestaltung von Java-Anwendungen. 2, 8, 11, 18, 24, 30, 33, 36, 37

L

Lambda-Ausdruck Kurzschreibweise für Anonyme Innere Klassen. 4–6

M

MultiFilter Filter für den 3. Anwendungsfall; Beinhaltet zwei separate Filter. 26, 27

MVC-Pattern Model-View-Controller-Pattern: Design-Pattern zur Einteilung des Codes in drei Zuständigkeitsbereiche. 10, 11

N

Neuropsychologie Teilgebiet der Psychologie, das sich mit der Funktionsweise des Gehirns beschäftigt. 1

O

ObservableValue Wert eins beliebigen Typs, dessen Änderungen per JavaFX-Listener publiziert werden können. 8

P

Property Beschreibende Eigenschaft eines Objektes, z.B. in Form einer Membervariablen des Typs ObservableValue. 8

R

Rohdaten Aus der Datenbank geladene, unverarbeitete Daten. 9, 14

S

Setter Funktion, die den Wert einer Membervariablen setzt. 27

Stream Datenstruktur, die Collection-Datentypen kapselt und komplexe Operationen darauf ermöglicht, ohne Schleifen manuell ausprogrammieren zu müssen. 4, 5

Stream API Programmierschnittstelle zu Verarbeitung von Streams. 4

V

ValueResolver Java-Klasse zum Ermitteln von Werten für ein Attribut zu einem Modelobjekt. 21, 22