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 5. 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

Zusammenfassung

Deutsche Zusammenfassung.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo

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**
1.1 Ziele der Arbeit . . . 1
1.2 Motivation . . . 1
1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **3**
2.1 Besonderheiten in Java 8 . . . 3
2.1.1 Default Implementierungen . . . 3
2.1.2 Functional Interfaces . . . 3
2.1.3 Stream API . . . 4
2.1.4 Lambda Ausdrücke . . . 5
2.2 JavaFX . . . 8
2.3 Die Anwendung . . . 9
2.4 Codedesign-Patterns . . . 10
2.4.1 MVC-Pattern . . . 10
2.4.2 Command-Pattern . . . 12

1 Einleitung

1.1 Ziele der Arbeit

Diese Arbeit hat das Ziel, die Anwendung „FalkoFX“ um einen Anwendungsfall zu erweitern. 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, teils Programme mit eingeschränktem Funktionsumfang. Eines haben aber alle gemein: Den Fortschritt.

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 und die große Neuerung, das JavaFX-Toolkit, vorgestellt.

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 einige 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 auch umgesetzt werden.

2 Grundlagen

2.1 Besonderheiten in Java 8

Mit JavaSE 8 führt Oracle einige interessante Neuerungen in die weit verbreitete Programmiersprache Java ein [ULL14, S. 35]. Die für dieses Projekt wichtigen Änderungen werde ich folgend darlegen.

2.1.1 Default Implementierungen

Eine kleinere, aber dennoch einflussreiche Veränderung sind Default-Implementierungen für Interfaces. Das bedeutet, es können in Interfaces Methoden-Bodies ausprogrammiert werden. 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 – den Lambda-Ausdrücken. 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. 391 f.]

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, das zudem verdeutlicht, 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 Konstruktor, 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. [Ora00]

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 ohne 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 der Methodenreferenz ist die Konstruktorreferenz. Sie funktioniert analog zur Methodenreferenz. [Ull14, S83f.]

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. [Ora00]

Listing 2.8: Lambda-Scope [Ora00

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

2.2 JavaFX

Bei JavaFX handelt es sich um ein relativ neues GUI-Framework. Die erste Release-Version (JavaFX 8) wurde zusammen mit Java 8 ausgeliefert. JavaFX wurde entwickelt, um das bewährte, aber dennoch veraltete GUI-Framework Swing abzulösen und watet 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 problemlos möglich, 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 des 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. [Ora01]

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 2 von 3 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 besagtem 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 (Link) noch genauer erläutert.

In jedem Anwendungsfall sind verschiedene Aktionen und Exportmöglichkeiten verfügbar, die aber 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 ich später erläutern werde. Die Galerie, die in Anwendungsfall 1 benutzt wurde, entfällt für dieses Szenario, da sie nicht praktikabel wäre.

Anwendungsfall 3

Die grobe 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 werden und die später angezeigt werden. Der Controller kann die Daten bzw. den Zustand des Models verändern. [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. Die View interagiert mit dem Controller um auf Aktionen des Nutzers zu reagieren. [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. [FFSB06, S. 529ff.]

Die Anwendung in FalkoFX

Auf Codeebene kann das MVC-Pattern durch verschiedene andere Patterns realisiert werden. Um die Änderungen des Models zu publizieren und andere Teile der Anwendung zu benachrichtigen ist es eine Möglichkeit, das Observer-Pattern zu verwenden. Genau dies bietet auch JavaFX an. Hier gibt es das Observable-Interface, das 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]. Mehr als sind in JavaFX 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 immer 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. [OOD00]

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 zum Rücknahme 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. [OOD00] 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. [OOD00]

Abbildungsverzeichnis

Tabellenverzeichnis

ethjnjdgjsfgjsrmk