



Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Newid Rahimi

**Praktische Bewertung von JavaFX zur Erstellung von
dialogorientierten Geschäftsanwendungen**

*Fakultät Technik und Informatik
Studiendepartment Informatik*

*Faculty of Engineering and Computer Science
Department of Computer Science*

Newid Rahimi

**Praktische Bewertung von JavaFX zur Erstellung von
dialogorientierten Geschäftsanwendungen**

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Ulrike Steffens
Zweitgutachter: Prof. Dr. Philipp Jenke

Eingereicht am: 10. Juni 2014

Newid Rahimi

Thema der Arbeit

Praktische Bewertung von JavaFX zur Erstellung von dialogorientierten Geschäftsanwendungen

Stichworte

GUI, JavaFX, MVC-Pattern, Eventhandling, Swing, Rich Internet Applications, FXML

Kurzzusammenfassung

Diese Bachelor-Thesis befasst sich mit der Analyse und Bewertung des Frameworks JavaFX für die Oberflächenprogrammierung. Der Schwerpunkt dieser Arbeit liegt darin, die Untersuchung von JavaFX auf Geschäftsanwendungen zu beziehen. Mit Hilfe von praktischen Beispielen wird auf die Konzepte von JavaFX eingegangen. Unter anderem erfolgt die Untersuchung über die Möglichkeit eine Swing-Anwendung in JavaFX zu migrieren. Der praktische Nutzen von JavaFX wird abschließend aufgelistet, indem die Vor- und Nachteile aufgezeigt werden.

Newid Rahimi

Title of the paper

Practical evaluation of JavaFX for the creation of a dialogue oriented business application

Keywords

GUI, JavaFX, MVC-Pattern, Eventhandling, Swing, Rich Internet Applications, FXML

Abstract

The bachelor thesis deals with the analysis and evaluation of the framework JavaFX for user interface programming. The thesis focuses on the examination of JavaFX in context with business applications. The JavaFX concepts are being presented by using practical examples. Amongst others the possibility to migrate a swing application into JavaFX will be investigated. The practical benefit of JavaFX is being depicted by highlighting the advantages and disadvantages.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ziel dieser Arbeit	1
1.2	Aufbau der Arbeit	1
2	Konzepte in JavaFX	3
2.1	Neuerungen gegenüber Swing	3
2.2	Scene Builder	4
2.3	JavaFX Architektur	5
2.4	Scene Graph	6
2.5	JavaFX Lebenszyklus	8
2.6	GUI-Elemente	10
2.7	Charts	13
2.8	Layout	16
2.9	CSS	20
2.10	FXML	23
2.11	Properties- und Databinding-Modell	27
2.11.1	Properties	27
2.11.2	Databinding	30
2.12	Events	31
2.12.1	Eventhandling	32
2.12.2	Zustellung eines Events	34
2.13	Collections	35
2.14	Nebenläufigkeit in JavaFX	38
2.14.1	Die Task Klasse	39
2.14.2	Die Service Klasse	40
2.15	Multi-Touch-Integration	40
2.16	MVC-Pattern	41
2.17	MVVM- Pattern	42
3	Migration von GUI-Anwendungen	47
3.1	Von Swing nach JavaFX	47
3.1.1	Verfahren	48
3.1.2	Fazit	51
4	Analyse	52
4.1	Vorteile von JavaFX	52

4.2	Nachteile von JavaFX	53
4.3	Gegenüberstellung von JavaFX und Swing	54
4.4	Stabilität der Anwendung	56
4.5	Application Frameworks	58
5	Schluss	59
5.1	Zusammenfassung	59
5.2	Ausblick	61
Anhang		62
A.1	Voraussetzung zum Ausführen von JavaFX	62
A.2	Laborumgebung	63

1 Einleitung

1.1 Ziel dieser Arbeit

Heutzutage werden immer höhere Anforderungen an die graphische Benutzeroberfläche (GUI) gestellt, besonders im Desktop-Bereich. Ziel dieser Arbeit ist es, die neue Softwaretechnologie JavaFX für den Einsatz von dialogorientierten Geschäftsanwendungen zu untersuchen und dem Vorgänger Swing gegenüberzustellen.

Durch diese neue Softwaretechnologie wird es Java-Anwendungen ermöglicht, auf Desktops, Smartphones und Embedded-Geräten eine graphische Oberfläche zu bieten.

Die Veröffentlichung der ersten JavaFX Version erfolgte im Jahr 2008. Bis Version 2.0 wurde in JavaFX mit einer eigenen Skriptsprache entwickelt, mittlerweile wurde JavaFX Script durch Java ersetzt und seit Version 2.0 gibt es eine Java-API.

Für die Unternehmen stellt es einen großen Anreiz dar, die alten Oberflächen ihrer Anwendungen abzulösen oder auf mobile Umgebungen umzusteigen, da *Oracle* die Entwicklung von Swing eingestellt hat. Früher oder später muss man sich also mit der Frage beschäftigen, ob man die bereits vorhandene Java-Anwendung in JavaFX migriert oder gar neu schreibt.

Interessant ist es deswegen, sich die neuen Konzepte von JavaFX einmal genauer anzuschauen und zu untersuchen, ob JavaFX als Ablösung von Swing die Erwartungen erfüllt.

1.2 Aufbau der Arbeit

Die Arbeit ist in fünf Kapitel aufgeteilt. Nach einer Einleitung beschreibt das zweite Kapitel die einzelnen Konzepte von JavaFX. Es wird auf die JavaFX Architektur eingegangen, um ein Grundverständnis für die Konzepte zu liefern. Auch wird FXML und CSS in diesem Kapitelabschnitt näher erläutert und anhand von Beispielen erklärt.

Die einzelnen Konzepte werden vorgestellt und mit Hilfe von praktischen Beispielen erläutert. Aspekte wie der Lebenszyklus und der Aufbau einer JavaFX-Anwendung werden auch näher untersucht. Anhand des neuen Konzepts von Properties und DataBinding wird auf die neue Möglichkeit eingegangen, das MVVM-Pattern zu realisieren.

Das dritte Kapitel untersucht die Möglichkeit, Swing-Anwendungen in JavaFX zu integrieren.

Im Rahmen dessen wird die *SwingNode*-Komponente vorgestellt, die erst seit *JavaSE 8* Bestandteil von JavaFX ist.

In Kapitel fünf wird dann abschließend auf den praktischen Nutzen eingegangen, den man durch die Entwicklung von JavaFX hat. Um sich einen Eindruck über die Stabilität der Anwendung zu machen, werden in diesem Abschnitt Projekte von Unternehmen vorgestellt, die sich bereits für die Entwicklung mit JavaFX entschieden haben. Das Schlusskapitel enthält eine kurze Zusammenfassung der Arbeit und einen Ausblick.

2 Konzepte in JavaFX

2.1 Neuerungen gegenüber Swing

Die Neuerungen gegenüber Swing sind sehr umfangreich. Angefangen über neue Layout-Manager, dem neuen Sprachkonstrukt FXML oder der Formatierungssprache CSS bis hin zum Properties- und Bindings-Modell.

Immer mehr Geräte verfügen über eine Multi-Touch-Steuerung. Mit JavaFX ist es auch möglich, Multi-Touch-Anwendungen zu programmieren. Auch im Bereich der Animation und Multimedia gibt es Neuerungen wie zum Beispiel *Transitions*, *Interpolation*, *Timeline*-Animation. Da sich diese Arbeit aber nicht auf diesen Aspekt bezieht, besteht die Möglichkeit, unter folgendem Link Informationen zu diesem Thema zu erhalten [Castillo \(2013a\)](#).

Es werden im Folgenden ein paar Stichpunkte zu den Neuerungen aufgelistet:

- FXML
- CSS
- Scene Builder Tool
- Transformationen und Animationen
- 2D/3D
- Support für Audio/Video
- Charts Package
- Properties und Bindings
- Multitouch support

2.2 Scene Builder

Der JavaFX Scene Builder ist ein Werkzeug von Oracle. Damit lässt sich komfortabel eine Oberfläche entwerfen. Der Scene Builder ist kein Teil des JDK, aber man kann ihn kostenlos auf der Oracle-Seite beziehen. Er lässt sich mit jeder Java-Entwicklungsumgebung kombinieren. NetBeans hat den Scene Builder schon standardmäßig integriert, deswegen ist eine Zusatzinstallation dort nicht nötig.

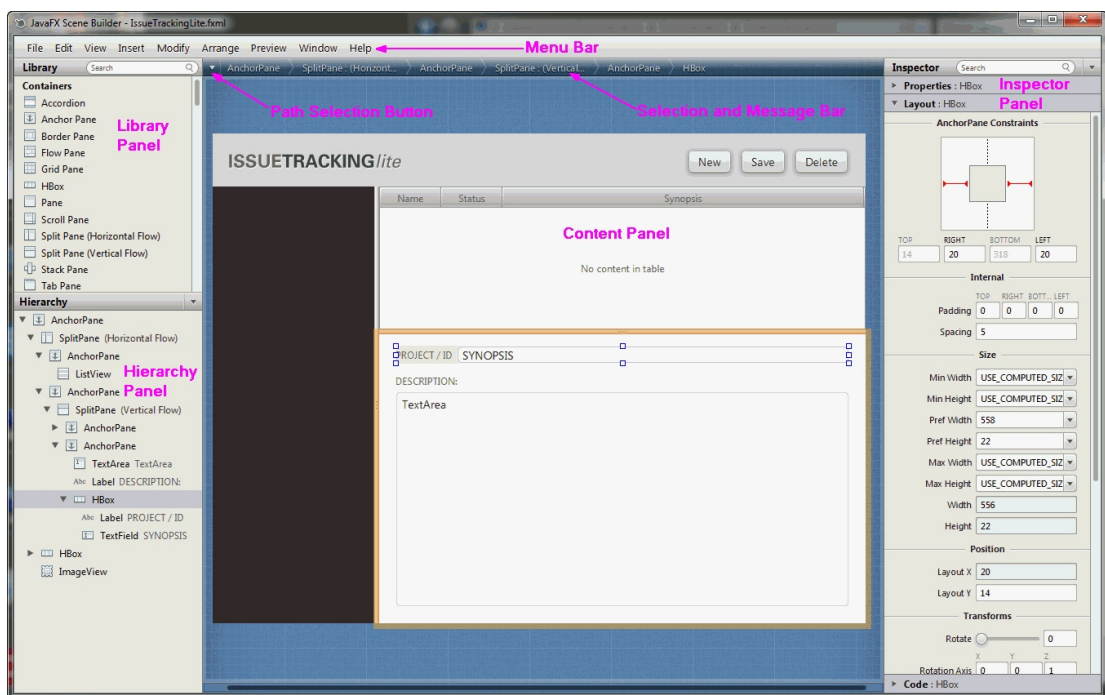


Abbildung 2.1: JavaFX Scene Builder. Quelle: [Castillo und Joan \(2013\)](#)

In [Abbildung 2.1](#) sieht man den JavaFX Scene Builder. Auf der linken Seite gibt es das *Library Panel*; hier hat man Zugriff auf die kompletten UI-Steuererelemente. Mit *Drag-and-Drop* kann man das ausgewählte Steuererelement in das *content Panel* platzieren. Unter dem *Library Panel* befindet sich der *Hierarchy Panel*. In diesem Bereich wird der Aufbau der GUI hierarchisch dargestellt. Auf der rechten Seite kann man anhand der Bedienungsfelder die Eigenschaft des jeweiligen Steuererelements spezifizieren.

2.3 JavaFX Architektur

Die Architektur von JavaFX ist von Grund auf neu und baut nicht auf die von Swing auf (Ullenboom, 2011). Die verschiedenen Bausteine der JavaFX-Plattform sind in Abbildung 2.2 dargestellt.

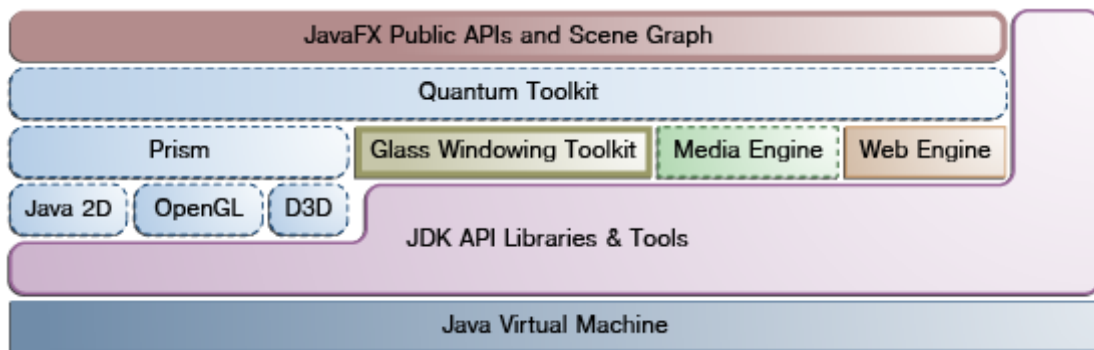


Abbildung 2.2: JavaFX Architektur Diagramm. Quelle: Castillo (2013b)

Das Glass Windowing Toolkit, das in der Abbildung 2.2 im mittleren Bereich dargestellt ist, wird im JavaFX-Graphik-Stack auf der niedrigsten Ebene angeordnet und stellt somit den Zugriff auf Low-Level-Betriebssystemroutinen zur Verfügung. Außerdem ist das Glass Windowing Toolkit für die Verwaltung der *event-queue* zuständig und läuft auch im selben Thread wie die JavaFX-Applikation - anders als beim AWT, der nur seine eigene *event-queue* verarbeitet hat. Das AWT läuft auf zwei verschiedenen Threads, einem für die Low-Level-Zugriffe und einem für die Java-Ebene.

Prism ist für das Rendering zuständig. Es kann sowohl mit Hilfe von Hardware als auch auf Software arbeiten. Die Voraussetzung für Hardwareunterstützung sind:

- DirectX 9 für Windows XP/Vista
- DirectX 11 für Windows 7
- OpenGL unter MacOS und Linux

Wenn das Rendering über Hardware möglich ist, wird dies auch genutzt. Aber wenn es die Voraussetzung dafür nicht gibt, wird auf Java2D zurückgegriffen. Dieses ist standardmäßig in allen *Java Runtime Environments* mit dabei.

Die Media Engine ist für das Einbinden von Video- und Audio-Sequenzen zuständig.

Die Web Engine unterstützt das Einbinden von Webseiten.

Das Quantum Toolkit verbindet die ganze untere Ebene, das Prism und das Glass Window Toolkit, sowie die Media- und die Web-Engine, und stellt sie der JavaFX-API zur Verfügung. Das Quantum Toolkit ist auch für die Synchronisation zwischen dem Event-Handling und dem Rendering zuständig.

2.4 Scene Graph

In JavaFX wird mit dem *Scene Graph* gearbeitet. Hierfür kommt die *retained mode API* für das rendering zum Einsatz. In Swing wird mit dem ImmediateMode-Rending gearbeitet. Der Entwickler muss sich bei der Entwicklung eigener Komponenten in Swing selber darum kümmern, wann und wo welcher Inhalt zu rendern ist. In JavaFX wird diese Aufgabe von dem Framework übernommen. [Grunwald \(2014\)](#)

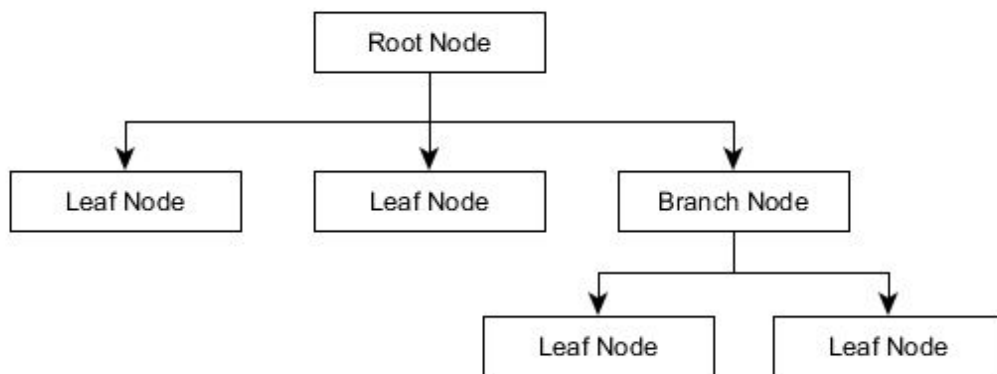


Abbildung 2.3: Der JavaFX Scene Graph

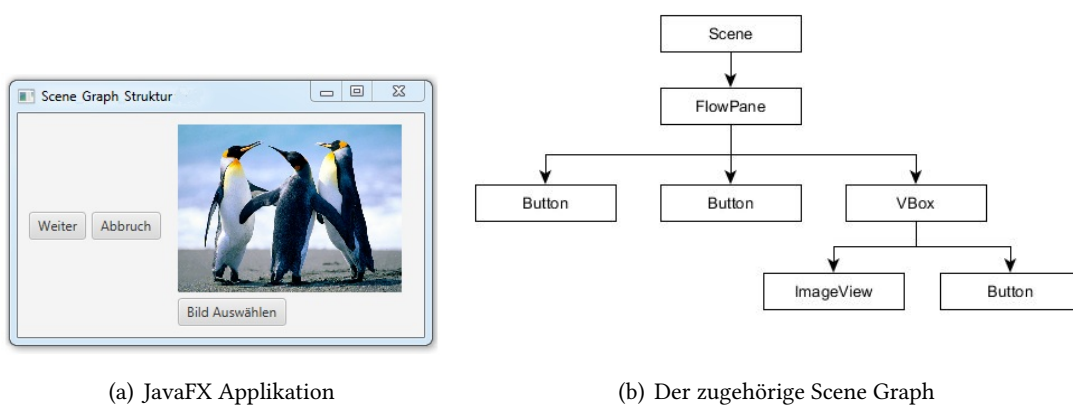
Wie man in [Abbildung 2.3](#) sehen kann, ist der JavaFX *Scene Graph* ein vorwärtsgerichteter Graph, mit dessen Hilfe Hierarchische Struktur aufgebaut wird. Der Graph kann folgende drei Node-Typen beinhalten:

Root Node: Als Root Node (Wurzelknoten) wird der erste Node im *Scene Graph* bezeichnet. Jeder Baum hat nur einen Root Node, also nur ein einziges Wurzelement.

Leaf Node: Der Leaf Node (Blattknoten) kann keine weiteren Nodes aufnehmen und besteht zumeist aus einer einfachen graphischen Komponente wie beispielsweise aus einem Steuerelement, einem Bild oder einem Media-Objekt.

Branch Node: Der Branch Node (Zweigknoten) hingegen, hat die Möglichkeit andere Nodes aufzunehmen und fungiert für diese dann als Parent (Elternknoten).

Häufig werden die JavaFX-Layoutcontainer als Branch Node verwendet, wie in Abbildung 2.4 (b) dargestellt. An dem Graphen erkennt man den Aufbau eines solchen Baumes und wie der VBox Layoutmanager als Branch Node fungiert.



(a) JavaFX Applikation

(b) Der zugehörige Scene Graph

Abbildung 2.4: Scene Graph

In einer JavaFX-Anwendung spiegelt sich der Aufbau des *Scene Graph* wider. Das Konzept des Aufbaus kann man mit einer Theaterbühne vergleichen. Es gibt eine Bühne, auf der sich die gesamte Handlung abspielt. Aus diesem Grund bezeichnet man in JavaFX den Abschnitt wo die Applikation angezeigt wird als *Stage* (Bühne). Wenn man das Beispiel mit dem Theater weiter verfolgt, gibt es auf der Bühne meistens verschiedene Szenen, die gespielt werden. Für den JavaFX Scene Graph gibt es äquivalent zum Beispiel, sogenannte *Scenes*. Die *Scene* wird dann zur Laufzeit auf die *Stage* geladen. Für die Software bedeutet das, dass man die verschiedenen Ansichten (*Views*) über mehrere *Scenes* erstellt und sie erst zum Zeitpunkt der Verwendung, auf die *Stage* lädt. In der Abbildung 2.4 (b) ist der *Scene Graph* für die danebenliegende JavaFX-Anwendung abgebildet. Man erkennt das die *Scene* einen Layoutcontainer vom Typ *FlowPane* als *Branch Node* verwendet. An dieser *Node* hängen *Leaf Nodes* eine weitere *Branch Nodes* vom Typ *VBox*.

2.5 JavaFX Lebenszyklus

Die Klasse *Application* ist für jede JavaFX-Anwendung der Eintrittspunkt. Wenn ein *Application*-Objekt mit einem *launch()* aufgerufen wird, geschieht folgendes:

1. Eine *Application*-Instanz wird erzeugt.
2. Die *init()*-Methode der erzeugten Instanz wird ausgeführt.
3. Danach wird die *start(javafx.stage.Stage)*-Methode ausgeführt.
4. Es wird solange gewartet, bis die Applikation beendet ist. Das passiert, wenn
 - die Methode *Platform.exit()* aufgerufen wurde oder
 - wenn das letzte Fenster geschlossen wurde und das *implicitExit*-Attribut von *Platform* den Wert *true* hat.
5. Zuletzt wird dann die *stop()* Methode aufgerufen.

Die *start()*-Methode ist *abstract* und muss somit überschrieben werden. Die *init()*- und die *stop()*-Methoden werden schon als *default* leer implementiert. Das Benutzen der *init()*-Methode ist dann besonders sinnvoll, wenn zu Beginn große Datenmengen geladen werden müssen. Somit kann man dem Benutzer schon einmal ein Willkommen-Fenster anzeigen, während die benötigten Daten bereitgestellt werden.

Die *start()*-Methode baut in der Zwischenzeit das eigentliche Applikationsfenster auf.

Im Listing 2.1 wird eine erste einfache Applikation gezeigt. In der *start*-Methode wird die GUI zusammengestellt. Dazu wird ein farbiges Rechteck-Objekt erstellt und ein Label-Objekt mit dem Text 'Hello World'. Als Layout-Manager wird der *StackPane* verwendet. Die erzeugten *Node*-Objekte werden in Zeile 12-13 an das Wurzelement angefügt. Durch das Anhängen der *Nodes*, baut sich der JavaFX spezifische *Scene Graph* auf. In Abbildung 2.5 ist die Anwendung zum Listing 2.1 abgebildet.

```
1 public class JavaFXBeispiel extends Application {  
2  
3     @Override  
4     public void start(Stage primaryStage) {  
5  
6         Rectangle rect = new Rectangle(150, 70, Color.LIMEGREEN);
```

```
7     Group group = new Group(rect);
8     Label label = new Label("Hello_World");
9
10    label.setFont(new Font(25));
11    StackPane root = new StackPane();
12    root.getChildren().add(group);
13    root.getChildren().add(label);
14    Scene scene = new Scene(root, 250, 250);
15
16    primaryStage.setTitle("JavaFX_Beispiel_Applikation");
17    primaryStage.setScene(scene);
18    primaryStage.show();
19 }
```

Listing 2.1: JavaFX Applikation, Hello World Beispiel

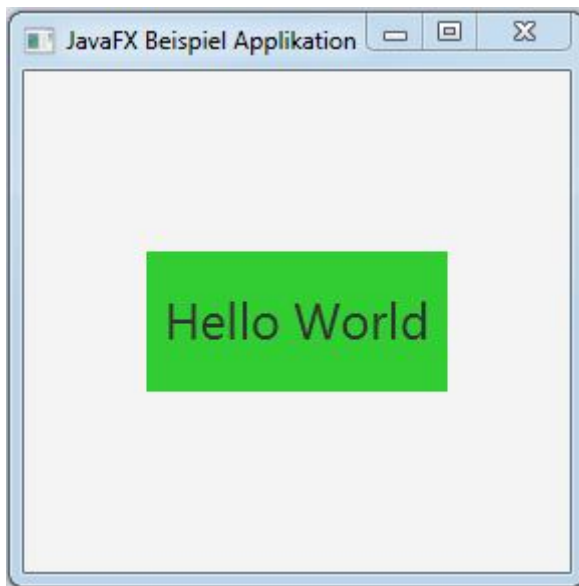
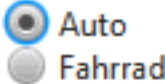



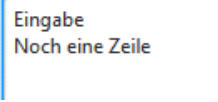
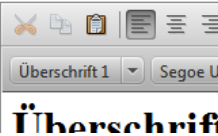

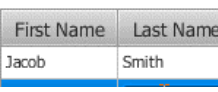

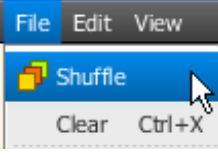
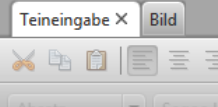






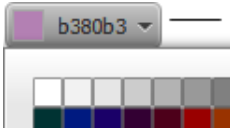

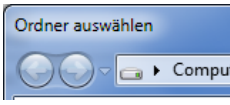
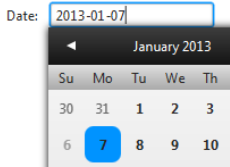
Abbildung 2.5: Abbildung zum Listing 2.1

2.6 GUI-Elemente

JavaFX besitzt viele GUI-Elemente, um eine ansprechende Applikation zu entwickeln. Es gibt eine breit gefächerte Auswahl, vom einfachen *TextField* bis hin zu komplexeren GUI-Elementen wie beispielsweise der *WebView*. Um einen Einblick in die verschiedenen Elemente zu bieten, werden einzelne Controls in einer Liste dargestellt.

Name	Verhalten
Button 	Einfacher Button, der Selektionszustand bleibt nicht erhalten
CheckBox 	Kann selektiert/deselektiert werden. Der Selektion-Zustand bleibt bis zur Änderung erhalten
RadioButton 	Hat das gleiche Verhalten wie die <i>CheckBox</i>
ToogleButton 	Kann selektiert werden. Der Selektion-Zustand bleibt erhalten, bis ein anderer Button derselben Gruppe gewählt wird
JComboBox 	Zeigt ähnlich wie eine Liste, einen ausgewählten Eintrag
ListView 	Zeigt einen ausgewählten Listeneintrag an, mit möglichen weiteren Einträgen
TextField 	Einzeiliges Texteingabefeld mit Cursor-Steuerung und Selektion

<p>Hyperlink</p> 	<p>Auswahlmöglichkeit einer URL</p>
<p>Slider</p> 	<p>Kann vertikal/horizontal verschoben werden</p>
<p>TextArea</p> 	<p>Mehrzeiliges Texteingabefeld mit Cursor-Steuerung</p>
<p>HTMLEditor</p> 	<p>Mehrzeiliges Texteingabefeld mit Cursor-Steuerung</p>
<p>PasswordField</p> 	<p>Eingegebene Zeichen werden durch * dargestellt</p>
<p>TableView</p> 	<p>Eingabe und Darstellung tabellarischer Daten</p>
<p>TreeView</p> 	<p>Baumstruktur mit aufklappbare Knoten</p>
<p>Menu</p> 	<p>Menü mit Einträgen und Untermenüs um bestimmte Aktionen zu Starten</p>
<p>TabPane</p> 	<p>Ermöglicht die Umschaltung zwischen mehreren Teildialogen</p>

<p>ToolBar</p> 	<p>Eine Leiste mit Buttons für Auswahl bestimmter Aktionen</p>
<p>SplitPane</p> 	<p>Variable Darstellung mehrerer Komponenten</p>
<p>ScrollPane</p> 	<p>Darstellung und Verschiebung eines Ausschnittes</p>
<p>TiledPane</p> 	<p>Auf-/Zu-Klappbarer Teildialog</p>
<p>Accordion</p> 	<p>Auf-/Zu-Klappbarer Teildialog, bestehend aus <i>TiledPane</i></p>
<p>ColorPicker</p> 	<p>Ein Dialog für die Farbauswahl</p>
<p>FileChooser</p> 	<p>Ein Dialog für die Dateiauswahl</p>
<p>DirectoryChooser</p> 	<p>Ein Dialog für die Verzeichnisauswahl</p>
<p>DatePicker</p> 	<p>Ein Dialog für die Datumsauswahl</p>

2.7 Charts

Eine weitere Neuheit im Vergleich zu Swing sind die JavaFX-Charts, die im *javafx.scene.chart*-*package* enthalten sind. Es gibt verschiedene Arten, zukünftig seine Massen von Daten übersichtlich und gut strukturiert darzustellen. In Abbildung 2.6 (b) sind die Balken-, Flächen-, Linien-, Blasen-, Streu- und Kuchen-Charts dargestellt. In Abbildung 2.6 (a) sieht man, wie die Struktur der Charts aufgebaut ist. Bis auf das *Pie Chart* erben alle weiteren Klassen von der Superklasse *XYChart*. Diese Klasse stellt den Zwei-Achsen-Charts Grundfunktionen zur Verfügung, um ihre Daten entsprechend anzuzeigen.

Es ist auch möglich, sich seine eigene Chart zu erstellen. Dazu muss die eigene Klasse nur von der *Chart*- oder bzw. von der *XYChart*-Klasse erben, die *@Override*-Methoden spezifizieren und für die einzelnen darzustellenden Datenelemente *Nodes* erzeugen.

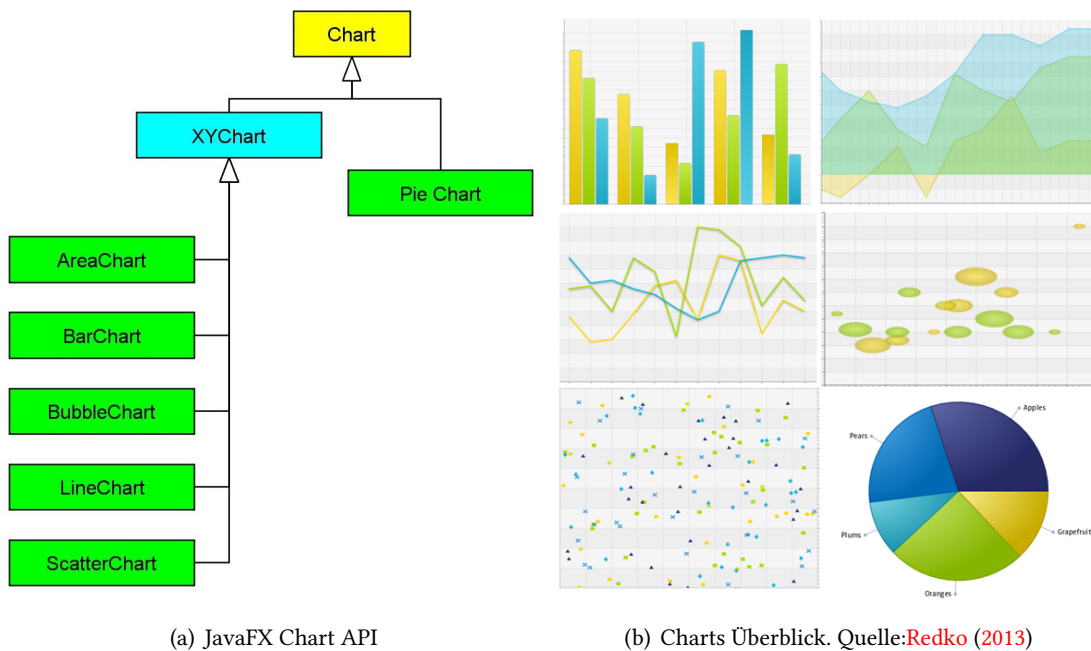


Abbildung 2.6: Chart Aufbau und Überblick

Im Listing 2.2 ist der Code, um eine Pie-Chart zu erzeugen. Der Code ist nicht besonders komplex oder umfangreich; so kann man schnell ansehnliche Diagramme erzeugen. Es wird eine Instanz der *PieChart* Klasse erzeugt. Für jedes Kuchenstück, das im Diagramm angezeigt werden soll, wird ein *PieChart.Data* Objekt erzeugt. Dieses Objekt erwartet zwei Parameter:

Erstens die Bezeichnung für das Kuchenstück und zweitens den Wert, der repräsentiert werden soll.

```
1 public class pieCharBeispiel extends Application{
2     @Override
3     public void start(Stage stage) {
4         Scene scene = new Scene(new Group());
5         stage.setTitle("Pie_Chart_Beispiel");
6         stage.setWidth(500);
7         stage.setHeight(500);
8
9         ObservableList<PieChart.Data> chartData =
10             FXCollections.observableArrayList(
11                 new PieChart.Data("Brasilien", 7),
12                 new PieChart.Data("Indonesien", 15),
13                 new PieChart.Data("Deutschland", 45),
14                 new PieChart.Data("Niederlande", 10),
15                 new PieChart.Data("USA", 28));
16         final PieChart chart = new PieChart(chartData);
17         chart.setTitle("Aktuelle_Installationen");
18         ((Group) scene.getRoot()).getChildren().add(chart);
19         stage.setScene(scene);
20         stage.show();
21     }
22     public static void main(String[] args) {
23         launch(args);
24     }
25 }
```

Listing 2.2: Pie Chart, Beispielcode

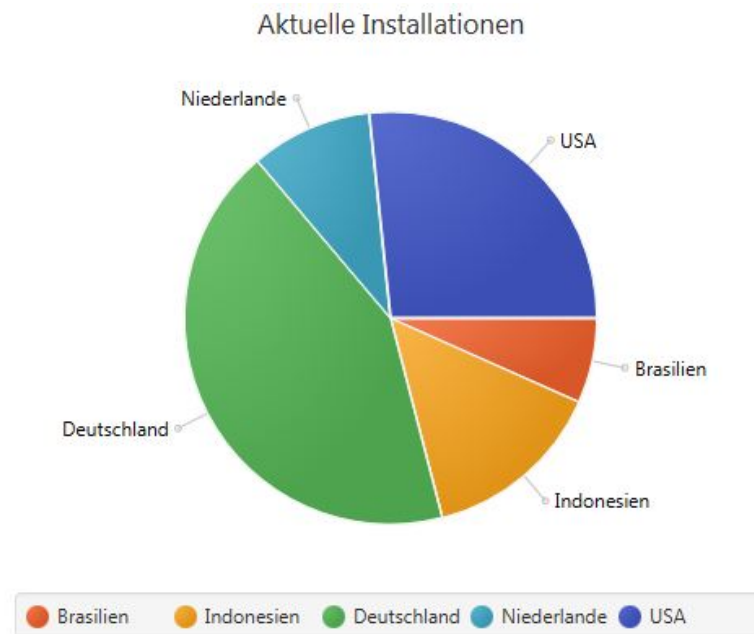


Abbildung 2.7: Pie Chart zum Codebeispiel

In Abbildung 2.7 sieht man das erzeugte Diagramm. Es veranschaulicht die zuvor definierten Daten über die Länder mit ihren jeweiligen Proportionen. Unter dem Chart ist die Legende abgebildet. Die Kuchenstücke des Charts sind mit ihren jeweiligen Ländern beschriftet, dies geht aus dem Listing 2.2 hervor.

2.8 Layout

Für die Entwicklung einer GUI gibt es die Möglichkeit, ein dynamisches oder absolutes Layout zu verwenden. Absolute Layouts sind sehr unflexibel. Will man der GUI eine weitere Komponente hinzufügen, so muss die Position der anderen Komponenten angepasst werden.

Dynamische Layouts hingegen tragen zum großen Teil dazu bei, dass eine Anwendung anscheinlich und leicht bedienbar gestaltet ist. Die Komponente kann einem bestimmten Bereich des Layouts zugeordnet werden, die sich dann auch beim Hinzufügen weiterer Komponenten anpasst. Die Anwendung ist benutzerfreundlicher, da der User die Anwendung an seine Bildschirmverhältnisse anpassen kann und das Layout sich somit dynamisch anpasst.

Da die manuelle Positionierung von Steuerelementen aufwändig, unflexibel und meistens fehleranfälliger ist, bietet JavaFX verschiedene dynamische Layout-Manager.

- **BorderPane:** Wird häufig als Standardlayout genommen.
- Es besitzt fünf Bereiche, in denen jeweils ein Element eingetragen werden kann.

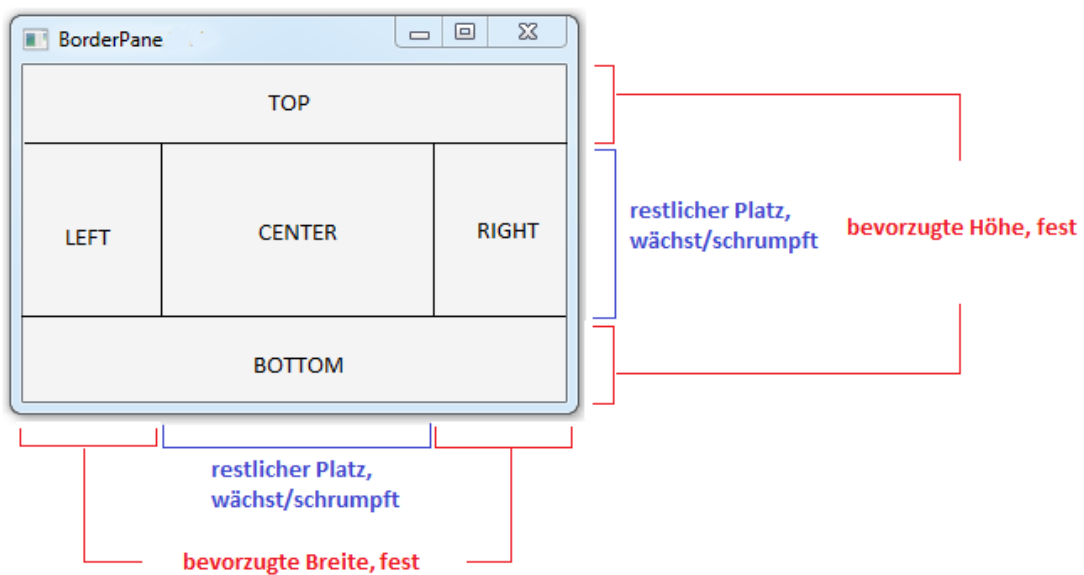
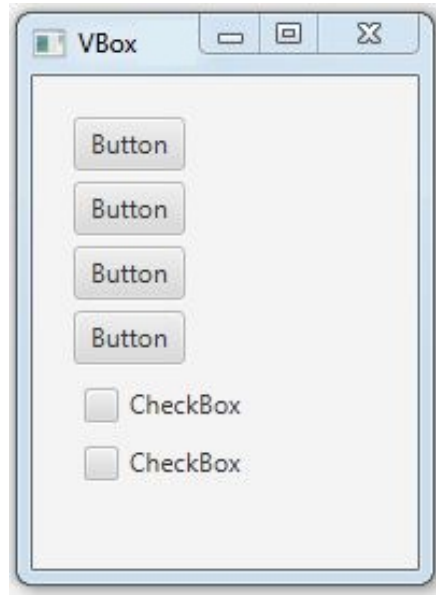
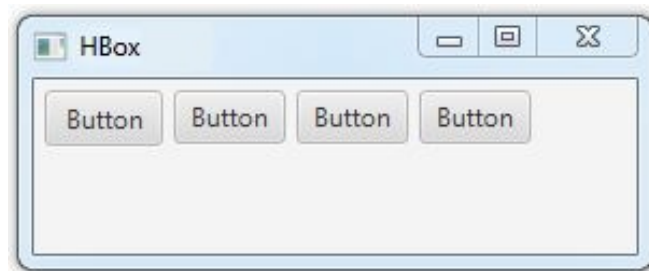


Abbildung 2.8: BorderPane Beispiel

- **VBox/HBox:** Platziert die Elemente vertikal/horizontal nacheinander in einer einheitlichen Größe.



(a) VBox Beispiel



(b) HBox Beispiel

Abbildung 2.9: Scene Graph

- Die Abstände zwischen den Elementen, die äußeren Ränder sowie Ausrichtung können explizit angegeben werden.
- Die Reihenfolge der Komponenten wird durch die Reihenfolge des Einfügens festgelegt.

- **GridPane:** Ordnet die Elemente in einem Gitter an, die aus gleich großen Gitterzellen besteht.

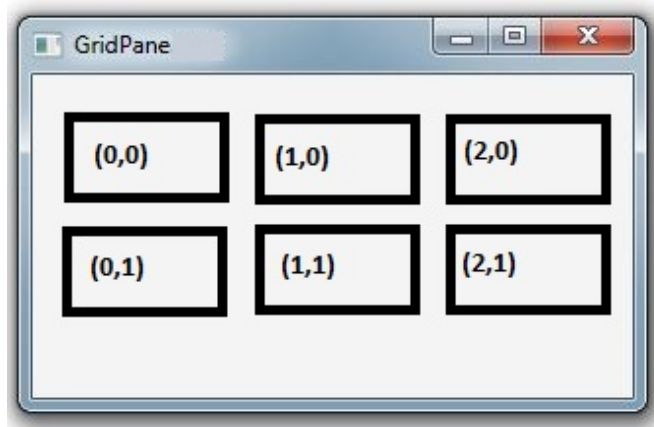


Abbildung 2.10: GridPane Beispiel

- **TabPane:** Ermöglicht das Umschalten zwischen verschiedenen *Tabs*.

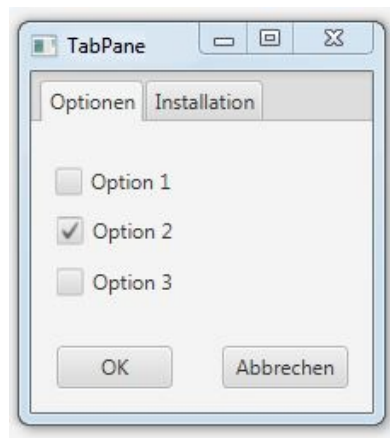


Abbildung 2.11: TabPane Beispiel

- **FlowPane:** Platziert die Elemente nacheinander in ihrer bevorzugten Größe.
- Die Abstände zwischen den Elementen sowie deren Anordnung können im Konstruktor des Layout Managers angegeben werden.

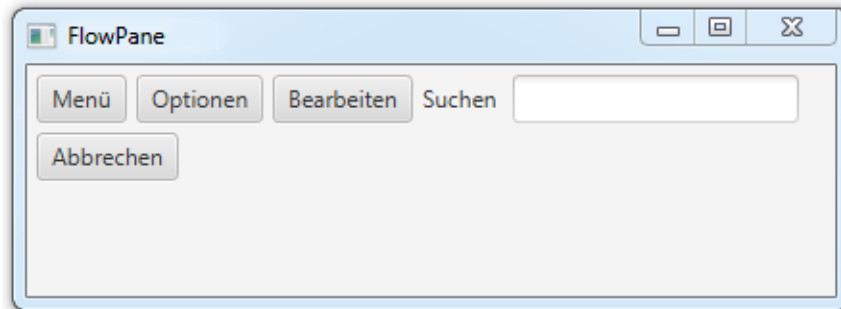


Abbildung 2.12: FlowPane Beispiel

```
1 JPanel panel = new JPanel(new FlowLayout());  
2 panel.add(komponente);
```

Listing 2.3: Layout Manager Beispiel in Swing

```
1 FlowPane pane = new FlowPane();  
2 pane.getChildren().add(komponente);
```

Listing 2.4: Layout Pane Beispiel in JavaFX

Die meisten Layouts in JavaFX sind auch schon von Swing bekannt. Zum Beispiel entspricht das *FlowLayout* in Swing, vom Verhalten her, dem *FlowPane* in JavaFX. Im Listing 2.3 ist ein Beispiel, wie man ein *FlowLayout* in Swing realisiert, aufgeführt. Dabei geht man so vor, dass man erstmal ein *JPanel*-Objekt erstellt, dem man dann einen Layout-Manager zuweist.

In JavaFX benutzt man *Layout-Panes*, wie in Listing 2.4 dargestellt. Der Vorteil der *Layout-Panes* ist der, dass die Größe und das Alignment der *Nodes* durch die *Pane* gehandhabt wird. Wenn eine Größenänderung an dem Applikationsfenster vorgenommen wird, passen sich die Steuerelemente automatisch an. Für dieses Verhalten hat jedes Steuerelement schon einen voreingestellten Wert, wie weit es sich in der Größe anpassen kann.

Wenn man mehr Kontrolle über die Größenanpassung der Steuerelemente haben will, kann man die Einstellung der Ausbreitungsreichweite selber vornehmen.

2.9 CSS

JavaFX Cascading Style Sheets (CSS) basiert auf der W3C CSS Version 2.1 [Gordon und Kouznetsov \(2013\)](#). Das Ziel von JavaFX CSS ist es, Webentwickler mit einzubeziehen, denn sie können auf ihrem existierenden Wissen zur CSS-Entwicklung aufbauen, ohne etwas komplett Neues lernen zu müssen. Die Webentwickler können mit ihren CSS Kenntnissen die Entwicklung und die Anpassung sogenannter "Themes" für die JavaFX Controls und die *Scene Graph*-Objekte umsetzen.

```
1 .root{
2     -fx-font-size: 14pt;
3     -fx-font-family: "Tahoma";
4     -fx-base: #A1D9F3;
5     -fx-background: #6776D7;
6     -fx-focus-color: #0825E0;
7 }
8 .buttonStyle {
9     -fx-text-fill: #006464;
10    -fx-background-color: #C0CCEB;
11    -fx-border-radius: 20;
12    -fx-background-radius: 20;
13    -fx-padding: 5;
14 }
```

Listing 2.5: style1.css

```
1 .root{
2     -fx-font-size: 16pt;
3     -fx-font-family: "Courier_New";
4     -fx-base: #A8E492;
5     -fx-background: #42E008;
6 }
7 .buttonStyle {
8     -fx-text-fill: red;
9     -fx-background-color: lightcyan;
10    -fx-border-color: green;
11    -fx-border-radius: 5;
12    -fx-padding: 5;
13 }
```

Listing 2.6: style2.css

Im Listing 2.5 und 2.6 wird jeweils eine CSS-Datei definiert. Die Datei enthält Informationen über die Darstellung der Anzeigeelemente. Im Gegensatz zu HTML, wo eine CSS-Datei nur einmal geladen wird und diese sich dann nicht mehr verändert, kann man in JavaFX CSS-Dateien dynamisch zur Laufzeit laden. In Abbildung 2.13 wird diese Funktionalität dargestellt. Je nach Button-Klick wird die im Code spezifizierte CSS-Datei geladen und die Benutzeroberfläche entsprechend der definierten CSS-Regel verändert.

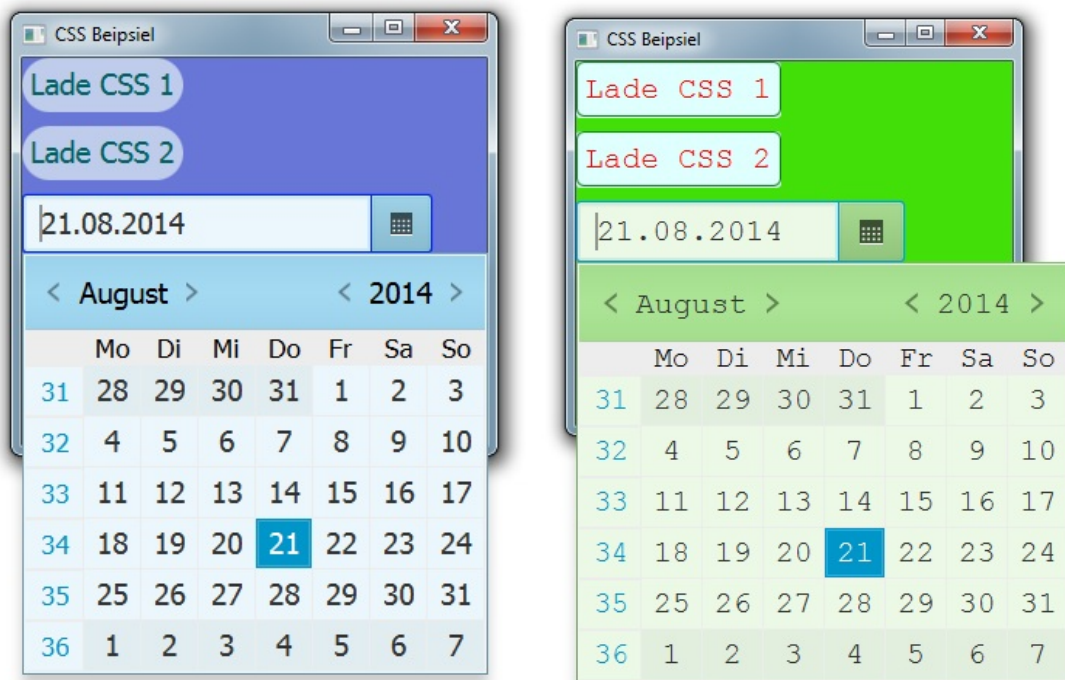
```
1 public class CSS extends Application {
2
3     private String style1Url = getClass()
4         .getResource("style1.css")
5         .toExternalForm();
6
7     private String style2Url = getClass()
8         .getResource("style2.css")
9         .toExternalForm();
10
11     @Override
12     public void start(Stage primaryStage) {
13         StackPane root = new StackPane();
14         final Scene scene = new Scene(root, 300, 250);
15         scene.getStylesheets().add(style1Url);
16
17         final Button btn1 = new Button("Lade_sytle1.css");
18         btn1.getStyleClass().add("buttonStyle");
19         btn1.setOnAction(new EventHandler<ActionEvent>() {
20             @Override
21             public void handle(ActionEvent event) {
22                 scene.getStylesheets().remove(style2Url);
23                 if(!scene.getStylesheets().contains(style1Url))
24                     scene.getStylesheets().add(style1Url);
25             }
26         });
27         final Button btn2 = new Button("Lade_sytle2.css");
28         btn2.getStyleClass().add("buttonStyle");
29         btn2.setOnAction(new EventHandler<ActionEvent>() {
30             @Override
31             public void handle(ActionEvent event) {
32                 scene.getStylesheets().remove(style1Url);
33                 if(!scene.getStylesheets().contains(style2Url))
34                     scene.getStylesheets().add(style2Url);
```

```

34     }
35     });
36     DatePicker datePicker = new DatePicker();
37     root.getChildren().add(VBoxBuilder.create()
38         .spacing(10).children(btn1, btn2, datePicker).build());
39     primaryStage.setScene(scene);
40     primaryStage.show();
41     primaryStage.setTitle("CSS_Beispiel");
42 }
43 public static void main(String[] args) {
44     launch(args);
45 }

```

Listing 2.7: Das Hauptprogramm; dynamisches Laden der CSS-Datei zur Laufzeit



(a) Darstellung mittels style1.css

(b) Darstellung mittels style2.css

Abbildung 2.13: DatePicker Beispiel

Im Listing 2.7 ist das Hauptprogramm dargestellt, wo auch die beiden zuvor definierten CSS-Dateien mit eingebunden werden. In Zeile 3 und 7 wird dafür jeweils der Pfad der CSS-Datei in einen *String* gespeichert. Als Standarddarstellung für das Programm wird die *style1.css*-Datei verwendet (Zeile 15). In den jeweiligen *EventHandler*-Methoden der beiden *Buttons* wird dann jeweils die ausgewählte Darstellung für die Oberfläche geladen.

2.10 FXML

Mit JavaFX 2.0 wurde erstmalig für den Desktop-/Rich-Client-Bereich die Möglichkeit geschaffen, die graphische Oberfläche über eine XML-Beschreibungssprache zu entwerfen. Dabei können nicht nur die graphischen Komponenten definiert, sondern auch das dazugehörige Layout und die Java-Controller-Klasse, die unter anderem Button-Events entgegennimmt, angegeben werden. Mit FXML können somit Oberflächen außerhalb des Programmcodes geschrieben werden. Das folgende Beispiel soll eine Oberfläche mit zwei Elementen darstellen: Ein *Label* und ein *Button*. Klickt der Anwender auf die Schaltfläche, erscheint eine "Hello World"-Ausgabe auf der Oberfläche.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.scene.control.*?>
4 <?import javafx.scene.layout.*?>
5
6 <HBox xmlns:fx="http://javafx.com/fxml/1"
7       fx:controller="javafxapplication2.FXMLHelloWorldController"
8       prefHeight="50"
9       prefWidth="200">
10  <children>
11    <Button text="Klick_mich!"
12           onAction="#onButtonClick" />
13    <Label fx:id="label" />
14  </children>
15 </HBox>
```

Listing 2.8: XML-Datei FXMLHelloWorld.fxml

Die hierarchische Struktur von XML passt, wie an dem Beispiel zu sehen ist, sehr gut zu dem Konzept von JavaFX, da die Zusammenstellung der GUI hierarchisch erfolgt: Ein Fenster

enthält einen Container; dieser beinhaltet wiederum Elemente. Am Ende erhält man eine Baumstruktur mit den jeweiligen Elementen, die der GUI zugeordnet sind.

1. In Zeile 3-4 stehen die `import`-Anweisungen, damit das Layout, mit dem in dem Beispiel gearbeitet wird, bekannt ist. Außerdem braucht man die folgende Anweisung `<?import javafx.scene.control.*?>` für den Import der Elemente `Button` und `Label`.
2. Als Layout wurde die `HBox` gewählt. Dieses Layout hat die Eigenschaft, GUI-Elemente horizontal aneinander zu reihen. In dem Beispiel hat die `HBox` vier Attribute: Das erste deklariert den Namensraum `fx`, das nächste eine Klasse. Diese Controller-Klasse wird benötigt, um später die Ereignisbehandlung für den Klick des `Buttons` zu übernehmen. Die letzten beiden Attribute von `HBox`, `prefHeight` und `prefWidth`, stellen im Vorfeld die bevorzugte Fenstergröße ein.

Es können auch eigene Klassen eingebaut werden, sofern sie mit `<?import>` bekannt gemacht wurden.

3. In Zeile 11-12 bekommt der `Button` zwei Attribute mit. Das erste dient nur zur Beschriftung "Klick mich!". Das Attribut `onAction` des `Buttons` referenziert den Programmcode. Bei jedem Klick auf den `Button` wird dieser Programmcode aufgerufen. Hier kann direkt Java-Quellcode stehen oder, wie in dem Beispiel, ein `#` und der Methodename, der dann in der zugehörigen Controller-Klasse deklariert werden muss. Der Klassenname, der die Methode `onButtonClick` implementiert hat, wurde bereits im Wurzelement (Zeile 7) deklariert.
4. Das `Label` bekommt mit dem Attribut `fx:id` eine ID zugewiesen. Somit lässt sich später im Code das `Label` erfragen. JavaFX macht noch einen weiteren Schritt und bildet das Objekt mit der ID automatisch auf ein Attribut der Controller-Klasse ab.

Die Ereignisbehandlung wurde gänzlich aus der FXML-Datei herausgelassen und wird in der Controller-Klasse abgehandelt. Im Listing 2.9 sieht man den Aufbau dieser Klasse.

```
1 public class FXMLHelloWorldController {
2
3     @FXML
4     private Label label;
5
6     @FXML
7     private void onButtonClick(ActionEvent event) {
8         label.setText("Hello_world!");
```

```
9 }
10 }
```

Listing 2.9: Die Controller-Klasse, FXMLLoaderHelloWorldController

Drei Eigenschaften kann man an dieser Klasse feststellen:

1. In der Controller-Klasse werden keine Schnittstellen erweitert.
2. Die Annotation `@FXML` in Zeile 3 ist dafür da, dass das *Label*-Objekt mit der ID auf diese Klassen-Variable abgebildet wird.
3. Im *Button* steht ein `onAction="#onButtonClick"`. Deswegen muss es in dieser Klasse die dazugehörige Methode geben. Die Annotation `@FXML` vor der Methode stellt die Beziehung zwischen diesen beiden Elementen her.

```
1 @Override
2   public void start(Stage stage) throws Exception {
3       Parent root = FXMLLoader.load(getClass()
4           .getResource("FXMLHelloWorld.fxml"));
5
6       Scene scene = new Scene(root);
7
8       stage.setScene(scene);
9       stage.show();
10  }
```

Listing 2.10: Das Hauptprogramm, JavaFXApplication.java

Das Hauptprogramm im Listing 2.10 ist relativ simpel aufgebaut, da es wie jede JavaFX Applikation eine *start*-Methode benutzt. Die *start*-Methode legt ein *root*-Element und eine *scene* fest. Die *scene* wird einer *stage* zugeordnet und anschließend angezeigt. Die *start*-Methode ist gegenüber einer klassischen JavaFX Applikation stark vereinfacht, da bei der klassischen Version die gesamte GUI über Java gebaut wird.

In diesem Beispiel wird letztlich nur die XML-Datei geladen, welche die Oberfläche beschreibt. Hierzu wird die Methode *load* aus der Klasse *FXMLLoader* verwendet, die dann dafür zuständig ist, die XML-Datei zu laden.

Inversion of Control

Inversion of Control (IoC) ist ein Begriff, der im Zusammenhang mit der objekt-orientierten Programmierung ein Paradigma bezeichnet, bei dem ein mehrfach verwendbares Modul ein spezifisches Modul aufruft. Mit Modul werden abgegrenzte und eigenständige Teile einer Software bezeichnet - dabei kann es sich um Objekte oder Klassen handeln. (ITWissen, 2014)

Mit *Inversion of Control* sorgt JavaFX dafür, dass die deklarativ beschriebenen Elemente in FXML in die vorgesehene Java-Klasse eingepflanzt werden. Im obigen Beispiel wurde in der FXML-Datei mit folgender Zeile

```
1 <Label fx:id="label" />
```

ein Label-Element beschrieben, mit *label* als *id*. Dieses Label-Element wird dann durch *IoC* der Klasse *FXMLHelloWorldController* (Listing 2.9) eingepflanzt.

```
1 @FXML  
2 private Label label;
```

Durch die *@FXML-Annotation* wird mit Hilfe der vorher definierten *id* die Verbindung endgültig festgelegt.

In Abbildung 2.14 ist die hier beschriebene Funktionsweise vom *IoC* graphisch dargestellt.

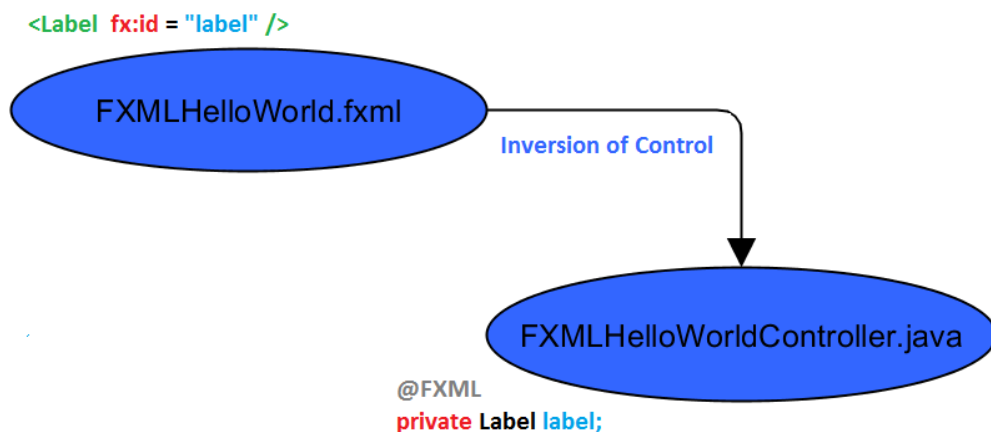


Abbildung 2.14: Inversion of Control Funktionsweise

2.11 Properties- und Databinding-Modell

Ein weiteres Feature von JavaFX sind die sogenannten Properties und Databindings. Databindings setzen das Konzept um, die Werte von Variablen an andere Variablen zu binden. Wenn der Wert der Variable A an den Wert der Variable B gebunden ist, wird Variable A automatisch den Wert von Variable B erhalten, ohne dass man dafür A explizit setzen muss. Damit dies funktioniert, braucht man die Properties.

2.11.1 Properties

Properties sind dazu da, Attribute einer Klasse zu kapseln (wrappen), damit sie an andere Daten-Objekte wie zum Beispiel GUI-Komponenten oder andere Variablen gekoppelt werden können. Durch das Kapseln eines primitiven Datentyps werden ihm zusätzliche Funktionen zur Verfügung gestellt. Somit würde beispielsweise ein Textfeld, auf dem ein Kontostand abgebildet ist, automatisch bei Transaktionen des Kontos aktualisiert werden.

Im Listing 2.11 gibt es eine Klasse *Konto* mit einer privaten Instanzvariable *kontostand*. Wie vorher erwähnt, soll die Anzeige bei jeder Veränderung des Kontostands zeitgleich aktualisiert werden.

```
1 import javafx.beans.property.DoubleProperty;
2 import javafx.beans.property.SimpleDoubleProperty;
3
4 public class Konto {
5
6     private DoubleProperty kontostand;
7
8     public final double getKontostand() {
9         if (kontostand != null){
10            return kontostand.get();
11        }
12        return 0;
13    }
14
15    public final void setKontostand(double hoehe) {
16        this.kontostandProperty().set(hoehe);
17    }
18
19    public final DoubleProperty kontostandProperty() {
```



```

20     if (kontostand == null) {
21         kontostand = new SimpleDoubleProperty(0);
22     }
23     return kontostand;
24 }
25 }

```

Listing 2.11: Properties Beispiel, Kontostand

Im Listing 2.11 sieht man, dass kein primitiver Datentyp *double* für den Kontostand gewählt wurde, sondern ein Objekt vom Typ *DoubleProperty*. Dieses Objekt ist dazu da, um primitive *double* Datentypen zu kapseln. Diese Kapslung bewirkt, dass die Variable mehr Funktionen erhält, insbesondere die Benachrichtigungsfunktionalität einer Property. Die Klassen des Packages *javafx.beans.property* erben alle vom *Interface* *Observable* oder *ObservableValue*. Dadurch wird ein Wert gekapselt. JavaFX hat solche *Wrapper*-Klassen für die Datentypen *long*, *int*, *float*, *double* und *String*. Die Klasse im Listing 2.11 enthält außerdem noch drei Methoden, von denen zwei übliche *Getter*- und *Setter*- Methoden sind. Das Besondere an dem Beispiel ist die dritte Methode vom Listing 2.11, da sie die Properties-Funktionalität gewährleistet. Die Methode gibt das *Property*-Objekt selbst zurück. Hierbei ist es wichtig, die Konvention der Methode zu einzuhalten, die sich aus dem Namen der Variablen und einem nachfolgenden *Property* zusammensetzt. Genau wie in der *Getter*-Methode wird auch hier geprüft, ob ein Objekt bereits existiert. Wenn noch kein Objekt erzeugt wurde, wird ein *SimpleDoubleProperty*-Objekt erzeugt.

Der offensichtlichste Vorteil beim Nutzen von Properties besteht darin, den primitiven Datentyp mit Funktionalität auszustatten, was vorher nur bei Referenztypen möglich war. Für die Entwickler bedeutet es etwas mehr Aufwand, da man zusätzlich die jeweiligen *Property*-Methoden schreiben muss. Von der Anwenderseite her betrachtet ermöglichen sie es jedoch, einen vergleichsweise übersichtlichen Code zu schreiben, da man keine zusätzlichen *Listener* für die Änderung der Variablen mehr benötigt.

So kann man, um bei dem Beispiel mit dem Konto zu bleiben, den Kontostand mit einem *ChangeListener* versehen. Somit wird auf aktuelle Veränderungen reagiert und diese auf einem Label angezeigt.

```

1 ...
2 public void start(Stage primaryStage) {
3     final Konto konto = new Konto();
4     konto.setKontostand(0);
5     Button geldButton = new Button("+10?_");
6     final Label label = new Label(new Double

```

```
7 (konto.getKontostand()).toString());
8
9 label.setMinSize(geldButton.getMinWidth(),
10 geldButton.getMinHeight());
11
12 label.setAlignment(Pos.CENTER);
13 konto.kontostandProperty().addListener(new ChangeListener<Object>() {
14     @Override
15     public void changed(ObservableValue<?> o, Object oldVal,
16         Object newVal) {
17         label.setText(new Double(konto.getKontostand()).toString());
18     }
19 });
20
21 geldButton.setOnAction(new EventHandler<ActionEvent>() {
22     @Override
23     public void handle(ActionEvent event) {
24         konto.setKontostand(konto.getKontostand() + 10);
25     }
26 });
27 ...
```

Listing 2.12: Properties Beispiel, Kontostand

Im Listing 2.12 wird ein JavaFX-Fenster mit einem Button und einem Label erzeugt. Das Label dient dazu, dem Benutzer den aktuellen Kontostand zu zeigen. Anhand des Buttons kann man in diesem Beispiel dem Konto 10 Euro hinzufügen. Dem *Property*-Objekt wird ein *ChangeListener* hinzugefügt, der die *setText*-Methode des Labels aufruft und diesem somit mit dem aktuellen Wert versieht. Der Button ist nur dafür da, den aktuellen Kontostand abzufragen und diesen dann um den vorgegebenen Wert zu inkrementieren. Durch dieses Beispiel wird deutlich, dass das *Property*-Objekt des Kontostandes selber für die Aktualisierung des Labels verantwortlich ist. Zusammenfassend lässt sich über Properties Folgendes sagen:

- Daten (Properties) werden in einem Objekt gekapselt
- Sie können mittels *Observe* überwacht werden
- Beim Arbeiten mit Properties sollte man sich an die Namenskonvention halten

2.11.2 Databinding

In JavaFX gibt es mehrere Möglichkeiten, das *Binding* anzuwenden. Im Allgemeinen unterscheidet man zwischen dem *Low-Level* und *High-Level-Binding*. Folgende *Binding*-Strategien sind möglich:

1. Das **High-level** binding unter Verwendung der Fluent API. Hiermit werden die meisten Anwendungsfälle für bindings abgedeckt. Im Beispiel 2.13 wird die High-Level-API verwendet.
2. Das **Low-level** binding unter Verwendung des *javafx.beans.binding.*-Packages*. Auf das Low-level binding greift man zurück, wenn die Berechnungen komplexer sind und die Fluent API dafür nicht ausreicht.

```
1 import javafx.beans.binding.NumberBinding;
2
3 public class BindingsBeispiel2 {
4
5     public static void main(String[] args) {
6         Konto konto1 = new Konto();
7         konto1.setKontostand(1000);
8         Konto konto2 = new Konto();
9         konto2.setKontostand(500);
10        NumberBinding sum = konto1.kontostandProperty()
11            .add(konto2.kontostandProperty());
12        System.out.println(sum.getValue());    //Ausgabe: 1500.0
13        konto2.setKontostand(300);
14        System.out.println(sum.getValue());    //Ausgabe: 1300.0
15    }
16 }
```

Listing 2.13: Binding Beispiel, Binding zwischen zwei Werten

Im Listing 2.13 wird das *Binding* zwischen Werten veranschaulicht. Es werden zwei Konten instanziiert und mit einem Startwert initialisiert. In Zeile 10 wird nun eine Variable *sum* vom Typ *NumberBinding* deklariert. Die beiden *Property*-Objekte werden durch *add* in das *sum*-Objekt addiert. Durch die Ausgabe kann man sehen, dass die Aktualisierung der Summe erfolgreich stattgefunden hat.

2.12 Events

Genauso wie Swing hat auch JavaFX verschiedene Event-Typen, die jeweils abhängig von der Benutzereingabe und der Event-Quelle erzeugt werden. Als Event-Quelle kommen zum Beispiel Tastatureingaben oder der Maus-Klick infrage. Ein Event ist in JavaFX eine Instanz von der *javafx.event.Event* Klasse oder einer Unterklasse von *Event*. Das Erstellen eines eigenen Events ist auch möglich, indem die eigene Klasse von der *Event*-Klasse erbt. Ein *Event* besitzt grundlegend folgende drei Eigenschaften:

Event Type ist eine Instanz der Klasse *EventType* und meldet beim Auslösen eines Events, welchen Event-Typ sie besitzt. Die Klasse *KeyEvent* beispielsweise reagiert auf Tastatureingaben und erzeugt je nach Verwendung der Tastatur eine *KEY_PRESSED*, *KEY_RELEASED* oder *KEY_TYPED* Benachrichtigung. Mit *EventType* kann man also den genauen Typ für einzelne *Event*-Klassen einordnen. *EventTypes* sind hierarchisch aufgebaut. Jeder Event-Typ hat einen Event-Namen und einen übergeordneten *Type*. Das Drücken einer Taste würde den Event-Namen *KEY_PRESSED* haben und der übergeordnete *Type* wäre *KeyEvent.ANY*, wie man anhand der Abbildung 3.4 sehen kann. Wenn man beispielsweise auf alle Events reagieren will, die durch eine Tastatureingabe generiert werden, benutzt man einfach den übergeordneten *Type* *KeyEvent.ANY* für das Eventhandling.

Als Event Source bezeichnet man die *Node*, die ein Event ausgelöst hat.

EventTarget ist ein Interface, welches jede Klasse implementiert haben muss, um einen Event zu erhalten. Die *Stage*-, *Scene*- und *Node*-Klasse haben das *EventTarget* Interface schon implementiert. Da die Unterklassen von diesen Klassen erben, ist schon eine sogenannte Benachrichtigungskette definiert. Durch diese Benachrichtigungskette (*dispatch chain*) wandert ein Event, bis es sein Ziel erreicht. Um *EventTargets* muss sich der Entwickler also nicht kümmern, da die meisten Steuerelemente schon solch eine Benachrichtigungskette besitzen. Anders sieht es aus, wenn man sein eigenes Steuerelement entwickelt und dieses auf eine Benutzerinteraktion reagieren soll. Wenn das entwickelte Steuerelement keine Unterklasse von *Stage*, *Scene* oder *Node* ist, muss das *EventTarget* Interface implementiert werden, damit das Event auch ankommt.

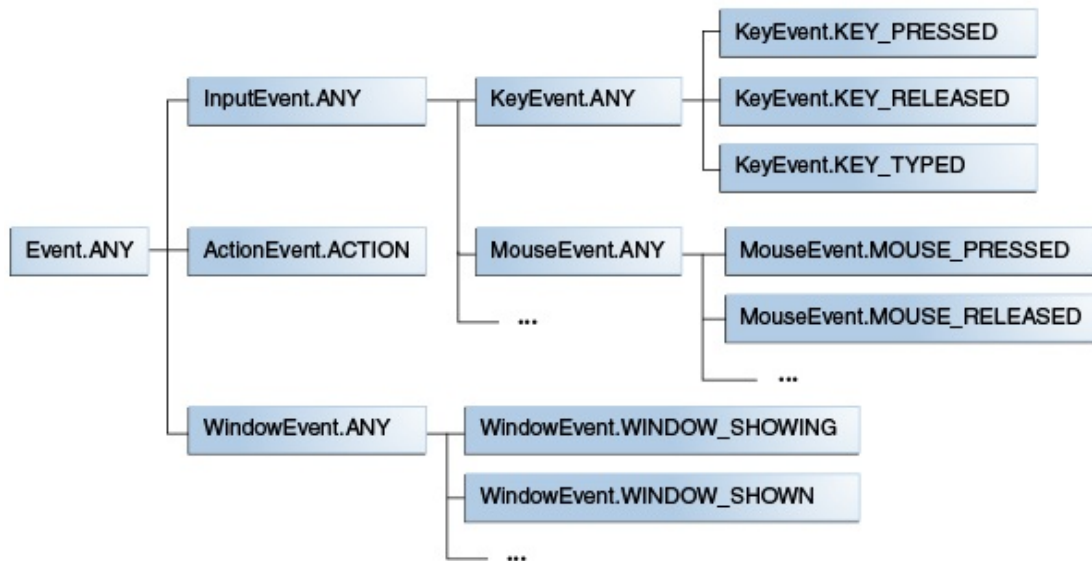


Abbildung 2.15: Event Type Hierarchy. Quelle: [Gordon \(2013\)](#)

Der Vorteil bei der neuen Event-Hierarchie liegt darin, dass ein Event-Handler mit der gleichen Methode auf die verschiedenen Events eines Teilbaums reagieren kann.

Wenn man beispielsweise eine Tabelle mit 1000 Zeilen hat, in der jede Zeile auf einen Mausklick reagieren soll, dann kann man dem übergeordneten Element des Hierarchybaums einen *Event Handler* zuweisen. Dieses Vorgehen wirkt sich positiv auf Performance und den Speicherverbrauch aus, da man nicht mehr über alle Zeilen iterieren muss und sich die entsprechende Anzahl an *Event Handler* spart.

2.12.1 Eventhandling

Das Eventhandling bietet zusätzlich zu Methoden-, Schleifen- und *if*-Konstrukten die Möglichkeit, den Programmfluss zu steuern. Es wird immer dann ein Eventhandling ausgeführt, wenn ein bestimmtes Event auftritt.

Für das Abarbeiten eines Events gibt es die Möglichkeit, einen *Event Handler* oder einen *Event Filter* zu registrieren. Beide Varianten implementieren das *EventHandler*-Interface. Der Unterschied zwischen dem *Filter* und dem *Handler* liegt darin, dass sie in verschiedenen Phasen ausgeführt werden.

Event Filter werden in der *event capturing phase* ausgeführt. Der Elternknoten kann das Eventhandling für seinen Kindknoten übernehmen und mit *consume()* den Event ver-

brauchen. Somit kann verhindert werden, dass der Event die Kindknoten erreicht. Filter, die für auftretende Events registriert sind, werden ausgeführt, sobald das Event den jeweiligen *Node* passiert.

Event Handler werden in der *bubbling phase* ausgeführt. Wenn das Event mit *consume()* nicht verbraucht wurde, wird es an den Elternknoten weitergereicht.

Eine *Node* kann einen oder mehrere Handler oder Filter registrieren. Die Reihenfolge der Abarbeitung erfolgt auf Grundlage des Hierarchie-Baums. Das Eventhandling für einen spezifizierten Event-Typ, wie beispielsweise *KEY_PRESSED*, erfolgt vor der Abarbeitung des übergeordneten Event-Typen *KeyEvent.ANY*. Die Reihenfolge für die Abarbeitung auftretender Events der selben Ebene ist nicht spezifiziert. Eine Ausnahme gibt es aber für Handler, die durch die *convenience*-Methode registriert sind. Diese werden als letztes ausgeführt.

Convenience-Methoden bieten die Möglichkeit zu spezifizieren, welches Event man verarbeiten will. Im Listing 2.14 werden die *convenience*-Methoden für die Registrierung von Tastatureingaben dargestellt.

```
1 textField.setOnKeyPressed(new EventHandler<KeyEvent>() {
2     @Override
3     public void handle(KeyEvent e) {
4         // Verarbeitung des Events !!!
5     }
6 });
7
8 textField.setOnKeyReleased(new EventHandler<KeyEvent>() {
9     public void handle(KeyEvent e) {
10        // Verarbeitung des Events !!!
11    }
12 });
```

Listing 2.14: Event Handling, convenience-Methoden Beispiel

Viele Komponenten können einen *Action Event* hervorrufen, sei es bei einem *Button* durch den Maus-Klick, bei dem *TextField* durch die Enter-Taste oder aber auch in einer *ComboBox*, in der man bestimmte Einstellungen für das Programm festlegt. Zur Behandlung solcher *Action Events* kann man mit *setOnAction* einen Event-Handler setzen, wie im Listing 2.15 gezeigt wird.

```
1 Button button = new Button("Okay");
2 button.setOnAction(new EventHandler<ActionEvent>(){
3 @Override
4 public void handle(ActionEvent event) {
```

```
5 // Verarbeitung des Events !!!  
6 }});
```

Listing 2.15: Event Handling, JavaFX Beispiel

2.12.2 Zustellung eines Events

In Abbildung 2.16 sind zwei Bilder abgebildet. Abbildung 2.16 (a) zeigt eine JavaFX Applikation, die einige Geometrische Formen beinhaltet. Daneben ist der Aufbau dieser Applikation als *Scene Graph* dargestellt. Wenn auf das Dreieck ein Maus-Klick erfolgt, wird die Zustellung eines Events eingeleitet. Anhand der weiß dargestellten Knoten in Abbildung 2.16 (b) erkennt man, wie die *Event Dispatch Chain* verläuft.

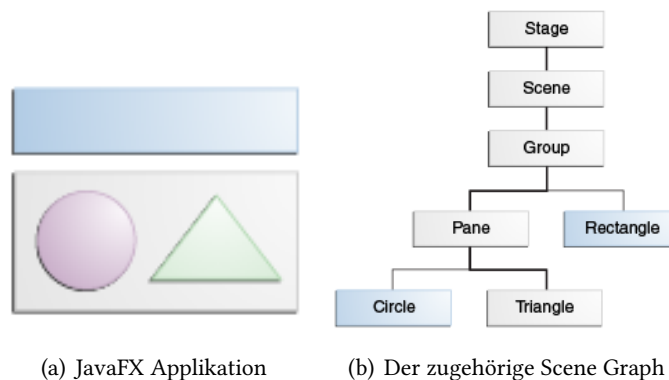


Abbildung 2.16: Beispiel für die *Event Dispatch Chain*. Quelle: (Gordon, 2013)

Der Prozess für die Zustellung eines Events beinhaltet folgende vier Schritte:

1. Im ersten Schritt wird das Ziel ausgewählt. Wenn eine Interaktion stattfindet, bestimmt das System nach internen Regeln, welche *Node* das Ziel ist. Beispielsweise wären bei Mausereignissen die *Node* das Ziel, welche mit der Position des *Cursors* übereinstimmt.
2. Im zweiten Schritt wird dann die Route erstellt. Die Initial-Route wird von der *Event Dispatch Chain* festgelegt. Diese wird mit der Auswahl des Zielevents in Schritt 1 mit der *buildEventDispatchChain()*-Methode automatisch erstellt.
3. Der dritte Schritt wird als die *Event Capturing Phase* bezeichnet. Das Event wandert vom Wurzelknoten, mittels der *Event Dispatch Chain*, bis zum Zielknoten. In dieser Phase können nur die *Event Filter* auf das Event reagieren. Wenn das Event nicht von

einem Filter mit der *consume()*-Methode aufgebraucht wurde, kommt es schließlich beim Ziel-Node an.

4. Der letzte Schritt wird als *Event Bubbling Phase* bezeichnet. Nach dem das Event die Ziel-Node erreicht hat und alle registrierten *Event Filter* benachrichtigt wurden, wird in dieser Phase der umgekehrte Weg genommen. Das Event steigt nun die Hierarchie vom Ziel-Node bis zum Wurzelknoten wieder hinauf. Jetzt haben die registrierten *Event Handler* die Möglichkeit, das Event abzuarbeiten. Wenn es auf dem Weg zum Wurzelknoten nicht von einem *Event Handler* mit der *consume()*-Methode aufgebraucht wurde, kommt es schließlich beim Wurzelknoten an.

Zusätzlich bietet JavaFX auch Event-Handling für Touch-Screens oder intelligente Trackpads, die zwischen Tippen, Rotieren, Scrollen und Zoomen anhand der Finger-Bewegung unterscheiden können.

2.13 Collections

Eine Collection (auch Container) ist in der Informatik ein abstraktes Objekt, welches Elemente des gleichen Typs speichert. Je nach Anforderungen verwendet man dabei unterschiedliche Datenstrukturen, um einen Container zu realisieren.

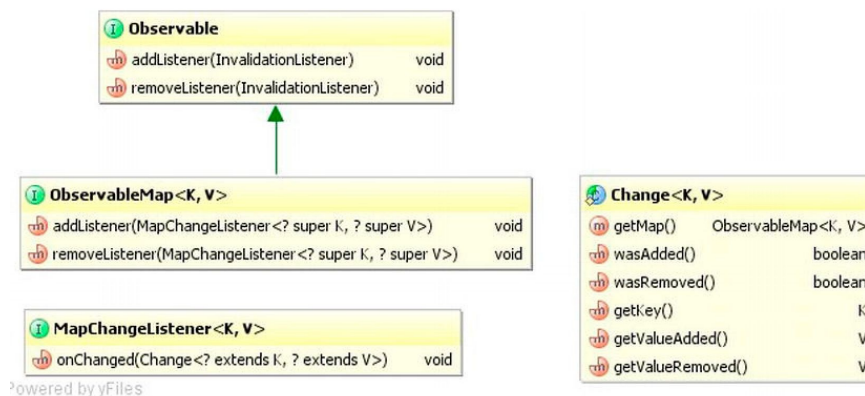


Abbildung 2.17: UML ObservableMap. Quelle: James L. Weaver und Dean Iverson (2012)

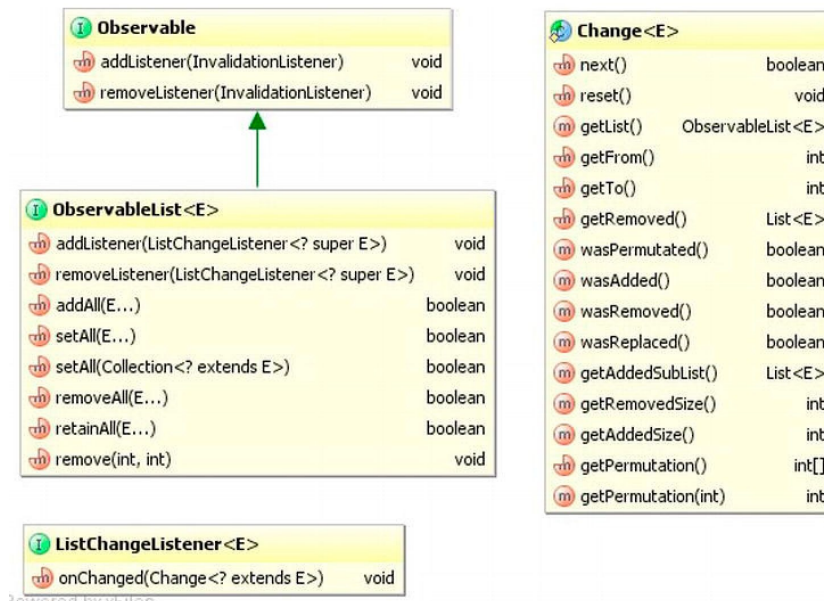


Abbildung 2.18: UML ObservableList. Quelle: [James L. Weaver und Dean Iverson \(2012\)](#)

Die Collections in JavaFX stellen eine Erweiterung der bisher vorhandenen *Java Collections Framework* dar und werden im *javafx.collection package* definiert. Dieses *package* beinhaltet folgende Klassen und Interfaces beinhalten:

Interfaces

- ObservableList (siehe: 2.18) ist eine Liste, die es *listernern* ermöglicht, auf Änderungen zu reagieren.
- ListChangeListener ist ein Interface, das bei Änderungen von der *ObservableList* benachrichtigt wird.
- ObservableMap (siehe: 2.17) ermöglicht eine Benachrichtigung des *observers* (Beobachters), wenn eine Veränderung auftritt.
- MapChangeListener ist ein Interface, das eine Benachrichtigung bei Änderungen der *ObservableMap* empfängt.

Klassen

- FXCollections ist eine *utility*-Klasse, die statische Methoden enthält. Diese Methoden entsprechen exakt den Methoden aus *java.util.Collections*.

- `ListChangeListener.Change`, stellt eine von `ObservableList` hervorgerufene Veränderung dar.
- `MapChangeListener.Change`, stellt eine von `ObservableMap` hervorgerufene Veränderung dar.

Die Interfaces `javafx.collections.ObservableList` und `javafx.collections.ObservableMap` erben beide von `javafx.beans.Observable` (und entsprechend `java.util.List` oder `java.util.Map`), um eine List oder Map bereitzustellen, die auf Änderungen reagieren kann. Wenn man sich die API dazu anschaut, stellt man fest, dass es Methoden gibt, um einen entsprechenden `Listener` hinzuzufügen oder zu entfernen. Im Listing 2.16 wird dies genauer veranschaulicht, und zwar anhand der `ObservableList`.

```
1 // Java Collection wird benutzt um eine Liste zu erzeugen
2 List<Integer> list = new ArrayList<Integer>();
3
4 // Nun wird die erzeugte list, in eine ObservableList-Wrapper gehüllt
5 ObservableList<Integer> observableList = FXCollections
6                                     .observableList(list);
7 observableList.addListener(new ListChangeListener() {
8
9 @Override
10 public void onChanged(ListChangeListener.Change change) {
11     System.out.println("Eine Änderung hat stattgefunden!");
12 }
13 });
14 // Änderungen an der observableList werden nun gemeldet.
15 // Das hinzufügen eines Wertes, wird den Listener aktivieren
16 observableList.add(1);
17 // Das hinzufügen eines Elements an die "nackte" list,
18 //wird keine Meldung bringen
19 list.add(2);
20 System.out.println("Size:_" + observableList.size());
```

Listing 2.16: Collections, ObservableList Beispiel

```
Eine Änderung hat stattgefunden !
Size: 2
```

Abbildung 2.19: Ausgabe des Listings 2.16

Im Listing 2.16 wird erst eine Standard *List* erzeugt. Diese kommt in ein *Wrapper*-Objekt vom typ *ObservableList*. Danach wird ein *ListChangeListener* registriert, der jedes Mal bei Änderungen der *ObservableList* aktiviert wird. Das gleiche Prinzip lässt sich auf die *ObservableMap* übertragen.

2.14 Nebenläufigkeit in JavaFX

Dieses Kapitel untersucht die Möglichkeiten zur Erstellung von *Multithreaded* Applikationen, die vom *javafx.concurrent package* unterstützt werden. Der JavaFX *Scene graph* repräsentiert die Grafische Benutzeroberfläche einer JavaFX-Anwendung und ist nicht *thread-safe*. Nur der *Thread* der GUI, auch bekannt als *JavaFX Application thread*, kann darauf zugreifen und ihn verändern. Das ist auch der Grund dafür, dass das Programm nicht mehr reagiert, wenn ein Codeabschnitt im *JavaFX Application thread* ausgeführt wird, der längere Zeit für die Durchführung beansprucht. Um dieses Problem zu lösen, verlagert man die zeitintensiven Codeabschnitte in einen im Hintergrund laufenden *Thread*. Somit blockiert man nicht den *JavaFX Application thread* und dieser kann sich um die Abarbeitung der Benutzer-Events kümmern, die über die *GUI* eintreffen. Der *JavaFX Application thread* ist nicht gleichzusetzen mit dem *Event Dispatch Thread (EDT)* von Swing und AWT, weshalb man bei der Integration von Swing-Komponenten in JavaFX und umgekehrt aufpassen muss. Dieses Thema wird in Kapitel 4 näher betrachtet. Neben dem *JavaFX Application thread*, gibt es den *Prism render thread* und den *Media thread*. Letzterer ist ein *Thread*, der im Hintergrund läuft und sich um die Synchronisation des aktuellen *Frames* des *Scene Graphs* kümmert, indem er den *JavaFX application thread* benutzt. Der *Prism render thread* kümmert sich um Render-Aufgaben. Er kann sowohl auf Hardware als auf Software rendern. Der *Thread* behandelt das Rendering separat vom *event dispatcher*. Somit wird ermöglicht, dass *frame N* rendert, während die Verarbeitung noch für *frame N+1* läuft. Diese Eigenschaft, nebenläufige Prozesse auszuführen, ist ein großer Vorteil. Besonders für die Systeme, die Multikern-Prozessoren besitzen.

Das *javafx.concurrent package* besteht aus einem *Worker interface* und zwei abstrakten Klassen, *Task* und *Service*. Beide Klassen implementieren das *Worker interface*. Das *Worker interface* liefert eine *API*, um die anstehenden Berechnungen im Hintergrund durchzuführen und mit der graphischen Oberfläche zu kommunizieren. Die *Task*-Klasse ermöglicht es dem Entwickler, asynchrone *tasks* (Aufgaben) in JavaFX zu implementieren. Diese *tasks* werden dann letztlich von der *Service*-Klasse ausgeführt. Der Lebenszyklus eines *Worker*-Objekts sieht wie folgt aus:

READY: Ist der Anfangsstatus des *Worker*-Objekts, den es bei Erzeugung annimmt.

SCHEDULED: Ist der Übergangstatus, wenn es eine Aufgabe gibt, die abzuarbeiten ist.

RUNNING: Wenn der *Worker* seine Aufgabe durchführt, ist er im *RUNNING* Status. Auch wenn der *Worker* gerade erst erzeugt wurde und keine anderen Aufgaben anliegen, geht er erst in den Status *SCHEDULED* und nimmt danach den *RUNNING* Status ein.

SUCCEEDED: Wenn der *Worker* seine Aufgabe erfolgreich durchgeführt hat.

FAILED: Das Objekt gerät nur dann in diesen Zustand, wenn bei der Abarbeitung eine *exception* auftritt.

CANCELLED: Wenn das *Worker*-Objekt mit der Methode *cancel()* unterbrochen wurde und es noch nicht den *SUCCEEDED* oder *FAILED* Zustand erreicht hatte, nimmt der *Worker* den *CANCELLED* Zustand ein.

Im welchen Zyklus sich das *Worker*-Objekt befindet, kann man mit der *getState()*-Methode erfragen.

2.14.1 Die Task Klasse

Die *Task* Klasse implementiert die Logik der im Hintergrund auszuführenden Aufgaben. Man muss darauf achten, dass die eigene Klasse von der *Task* Klasse erbt. Danach überschreibt man die *call()*-Methode, um darin die Aufgaben festzulegen, die im Hintergrund abgearbeitet werden sollen. Die *call()*-Methode sollte dabei nicht den Status der GUI verändern. Die Methoden *updateProgress*, *updateMessage* und *updateTitle* sind dafür zuständig, die entsprechenden Werte dem *JavaFX Application thread* mitzuteilen.

Die Klasse *java.util.concurrent.FutureTask* implementiert das *Runnable* interface. Die *Task* Klasse ist eine Unterklasse von *FutureTask*. So sieht die komplette Klassen-Signatur aus:
public abstract class Task<V> extends java.util.concurrent.FutureTask<V> implements Worker<V>, EventTarget

```
1 Thread t = new Thread(task);  
2 t.setDaemon(true);  
3 t.start();
```

Listing 2.17: Eine Task starten, Methode 1

```
1 ExecutorService.submit(task);
```

Listing 2.18: Eine Task starten, Methode 2

Wie in den Listings gezeigt wird, gibt es zwei Möglichkeiten, eine Task zu starten: Entweder einen Thread mit einer vorgegebenen Task als Parameter, oder indem man die *ExecutorService API* benutzt.

Im Listing 2.17 wird in Zeile 2 mit *t.setDaemon(true)*, dem *Thread* befohlen, solange im Hintergrund weiterzulaufen, bis die letzte *stage* geschlossen wurde. Dieser Befehl ist optional und beim Weglassen dieser Zeile standardmäßig auf *false* eingestellt.

2.14.2 Die Service Klasse

Ein Service ist eine nicht visuelle Komponente, die die erforderlichen Informationen kapselt, die es gilt, in einem oder mehreren Threads im Hintergrund zu verarbeiten. Als Teil der *JavaFX UI library* kennt der Service den *JavaFX Application thread* und ist allein dazu entworfen worden, den Anwendungsentwickler beim Managen von *Multithreaded Code* zu unterstützen, der mit der Benutzeroberfläche interagiert. Deswegen werden auch alle Methoden und Stauseigenschaften ausschließlich von dem *JavaFX Application thread* aufgerufen.

2.15 Multi-Touch-Integration

Mit JavaFX ist auch die Multi-Touch-Integration möglich. Geräte, die eine Multi-Touch-Steuerung haben, erzeugen ein bestimmtes Touch-Event. Dabei kommt es darauf an, welche Touch-Geste man mit dem Finger durchführt.

Touch- und Gesten-Events werden genauso wie andere Events gehandhabt. JavaFX erzeugt solche Touch- und Gesten-Events, wenn die Anwendung auf einem Gerät mit Touch-Screen oder einem Trackpad läuft und der Benutzer mit einem oder mehreren Fingern den Bildschirm berührt.

Geste	Beschreibung	Generierter Event
Drehen	Zwei-Finger-Drehbewegung im/gegen Uhrzeigersinn	ROTATE
Blättern	Schiebebewegung nach oben/unten	SCROLL
Swipe	Kehrbewegung ueber den Bildschrim	SWIPE
Zoom	Zwei-Finger-Kneifbewegung	ZOOM

```
1 field.setOnRotate(new EventHandler<RotateEvent>() {  
2   @Override  
3   public void handle(RotateEvent event) {  
4     Node node = (Node) event.getTarget();
```

```
5     if(node instanceof Shape) {
6         node.setRotate(node.getRotate() + event.getAngle());
7         event.consume();
8     }
9 }
10 });
```

Listing 2.19: Multi-Touch Beispiel, Geste zur Drehung der Gruppe

```
1 field.setOnZoom(new EventHandler<RotateEvent>() {
2     @Override
3     public void handle(ZoomEvent event) {
4         Node node = (Node) event.getTarget();
5         if(node instanceof Shape) {
6             node.setScaleX(node.getScaleX() * event.getZoomFactor());
7             node.setScaleY(node.getScaleY() * event.getZoomFactor());
8             event.consume();
9         }
10    }
11 });
```

Listing 2.20: Multi-Touch Beispiel, Geste zur Skalierung der Gruppe

2.16 MVC-Pattern

Das Model-View-Controller-Entwurfsmuster hat das Ziel, einen flexiblen Programmentwurf zu gestalten, der eine spätere Änderung oder Erweiterung für den Entwickler erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht. Die Applikationen werden dafür in drei Arten von Komponenten aufgeteilt. [Kaufmann \(2007\)](#)

Das Model repräsentiert die Daten einer Anwendung. Es kann ein einzelnes Objekt sein oder auch eine größere Struktur von mehreren Objekten. Es ist dabei zu beachten, dass das Model und seine Bestandteile eine Eins-zu-Eins-Verbindung zu den zu beschreibenden Objekten in der Realität haben.

Die View übernimmt die grafische Darstellung der Daten. Sie hat die Aufgabe, verschiedene Teilaspekte des Models besonders hervorzuheben und andere komplett auszublenden. Somit stellt die View eine Art visuellen Filter dar. Eine View ist immer verbunden mit einem dazugehörigen Model und erhält von diesem die Daten, die notwendig sind, um

eine Darstellung zu generieren, indem sie dem Model passende Nachrichten sendet. Somit muss die View die Semantik der Attribute des Models kennen.

Der Controller definiert die Interaktion zwischen dem Benutzer und der Applikation. Er versorgt den Benutzer mit Eingabemöglichkeiten, bei denen der Benutzer über die View Kommandos abgeben kann. Dies wird realisiert, indem die Kommandos vom Controller entgegengenommen, in Nachrichten übersetzt und an die entsprechenden Models weitergeleitet werden. Der Controller kann keinen Einfluss auf die View ausüben, also keine Veränderungen vornehmen. Andersherum sollte die View keine Kenntnis über die Eingaben des Benutzers haben, sodass es jederzeit möglich ist, im Controller Eingaben zu generieren, ohne die View verändern zu müssen.

Umsetzung in JavaFX

In JavaFX kann man durch Einsatz von FXML und Java das MVC-Entwurfsmuster realisieren. Die *View* wird in FXML dargestellt und der *Controller* besteht aus Java-Code.

2.17 MVVM- Pattern

Das *Model-View-View-Model*-Entwurfsmuster (MVVM) ähnelt dem *Model-View-Controller*-Entwurfsmuster, wie in Abbildung 2.20 dargestellt. Das Pattern definiert eine *View*, die mithilfe von Databinding an das *ViewModel* gebunden wird. Das *ViewModel* ist wiederum an eines oder mehrere *Models* gebunden.

Anhand eines kleinen Beispiels wird die Funktionsweise des MVVM-Patterns erläutert. Für das Beispiel gibt es ein Datenmodell mit dem Vor- und Nachnamen einer Person. Wie in Listing 2.21 zu sehen ist, wird das Datenmodell mit den JavaFX Properties ausgestattet. Für die Aktualisierung zwischen *Model* und *ViewModel* kann man an dieser Stelle auch das *Observer-Pattern* anwenden. Wichtig für das Pattern ist es, dass die *ViewModel* ihre Eigenschaft der *View* als JavaFX Properties bereitstellt. Casall (2013)

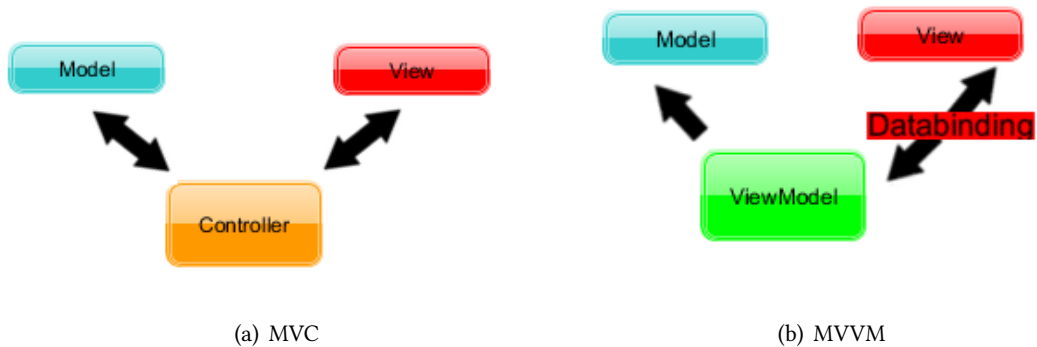


Abbildung 2.20: Unterschied zwischen MVC und MVVM

```

1 public class Person {
2
3     private final StringProperty vorname = new SimpleStringProperty();
4     private final StringProperty nachname = new SimpleStringProperty();
5
6     public Person(final String vorname, final String nachname) {
7         this.vorname.set(vorname);
8         this.nachname.set(nachname);
9     }
10
11    public StringProperty vornameProperty() {
12        return vorname;
13    }
14
15    public StringProperty nachnameProperty() {
16        return nachname;
17    }
18 }

```

Listing 2.21: Das Datenmodell, Vor-Nachname einer Person

In Abbildung 2.21 ist der Begrüßungstext abgebildet. Das Datenmodell, welches nur den Vor- und Nachnamen einer Person beinhaltet, wird von dem *ViewModel* aufbereitet (Listing 2.22). Die *View* besteht aus einer FXML-Datei und einer Java-Klasse. Die deklarativ beschriebenen Elemente in FXML werden in die Java-Klasse (Listing 2.23) geladen. Das passiert mittels *IoC*,

dessen Funktionsweise im Kapitel 2.3 erklärt wurde. Die FXML-Datei besteht nur aus einem *Label* für das Laden des Begrüßungstextes (Listing 2.24).

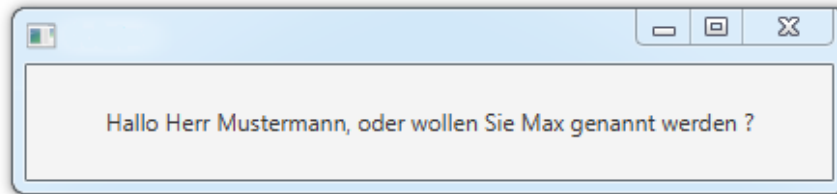


Abbildung 2.21: Begrüßungstext

```
1 public class WelcomeViewModel {
2
3     private final StringProperty welcomeString =
4         new SimpleStringProperty();
5
6     public WelcomeViewModel(final Person person) {
7         welcomeString.bind(Bindings.concat("Willkommen_Herr/Frau_",
8             person.nachnameProperty(), "_oder_wollen_Sie_",
9             person.vornameProperty(), "_genannt_werden?"));
10    }
11
12    public StringProperty welcomeStringProperty() {
13        return welcomeString;
14    }
15 }
```

Listing 2.22: Das ViewModel

```
1 public class WelcomeView implements Initializable{
2
3     @FXML
4     private Label welcomeLabel;
5
6     @Override
7     public void initialize(final URL arg0, final ResourceBundle arg1) {
8     }
9
10    public void setViewModel(final WelcomeViewModel viewModel) {
```

```
11 welcomeLabel.textProperty().bind(viewModel
12                                     .welcomeStringProperty());
13 }
14 }
```

Listing 2.23: Die WelcomeView-Klasse

```
1 <StackPane xmlns:fx="http://javafx.com/fxml"
2           fx:controller="view.WelcomeView">
3   <children>
4     <Label fx:id="welcomeLabel" text=""/>
5   </children>
6 </StackPane>
```

Listing 2.24: FXML-Datei, beinhaltet ein Labe-Element mit ID

Hier wird noch einmal auf die Gemeinsamkeiten und Unterschiede der beiden Entwurfsmuster eingegangen. Beide Architekturen sind dazu da, um die *View* vom *Model* zu trennen.

Model

- Speichert die Daten
- Kennt nur sich selbst und weiß nichts über die *Views* und *Controllers*

View

- Ist für die Präsentation der Daten zuständig
- Initialisiert und organisiert die Oberflächenelemente

Hauptunterschied zwischen MVC und MVVM:

MVVM: ViewModel

- Bidirektionale Kommunikation mit der *View*
- Die *ViewModel* repräsentiert die *View*.
- *View* Kommunikation: Die *View* ist direkt mit dem *ViewModel* verbunden, mithilfe von *Databinding*. Änderungen in der *View* werden automatisch im *ViewModel* widergespiegelt und umgekehrt.

MVC: Controller

- Der *Controller* entscheidet, welche *View* angezeigt wird
- Es kann mehrere *Views* für einen *Controller* geben
- *View* Kommunikation:
 - Der *Controller* hat eine Methode, die darüber entscheidet, welche *View* angezeigt wird.
 - Die *View* sendet *input events* an den *Controller*

Fazit

Wenn man mit JavaFX eine Applikation entwickelt, sollte man sich für das MVVM-Entwurfsmuster entscheiden, da JavaFX das *Databinding* unterstützt. Das *Databinding* ist für die Benutzung des MVVM-Entwurfsmusters eine Voraussetzung. Der Vorteil liegt darin, dass man sich für jede *View*, die Implementierung eines *Interfaces* für die Kommunikation spart, und dies bedeutet im Endeffekt weniger Code. Die Applikation ist dadurch wartbarer und deshalb nicht so fehleranfällig.

3 Migration von GUI-Anwendungen

Oracle präsentiert JavaFX als die Standard-Technologie für *Rich-Client* Java-Anwendungen. Swing wird noch weiterhin unterstützt, aber nicht mehr weiterentwickelt.

Viele Swing-Entwickler stehen also vor der Frage, ob sie mit der "veralteten" Technologie weiterarbeiten oder auf JavaFX umsteigen sollen. In der Praxis ist dies meist nicht so leicht, da in vorhandene Anwendungen bereits viel Geld investiert wurde. Eine optimale Möglichkeit wäre es, die ganze Swing-Anwendung ohne Probleme in JavaFX zu integrieren. So könnte man weiterhin seine Swing-Anwendung erhalten und zusätzlich von den Vorteilen von JavaFX profitieren. Diese Möglichkeit gibt es aktuell noch nicht.

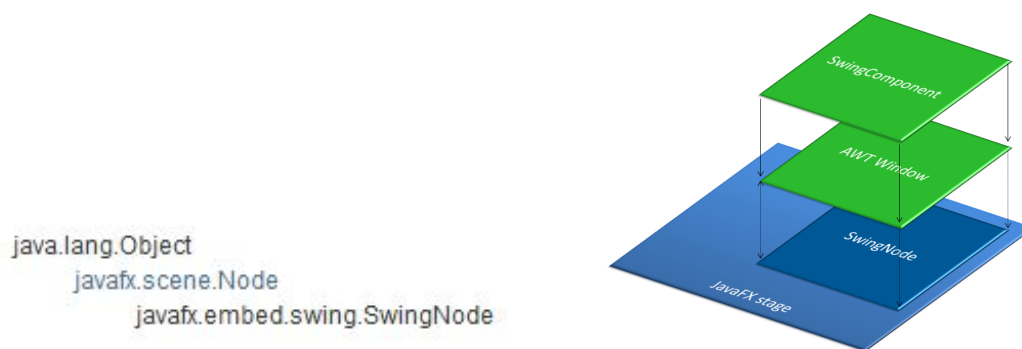
Eine schrittweise Integration ist bisher mit dem *JFXPanel* möglich, indem man einzelne Teile der Anwendung nach JavaFX portiert und integriert. *JFXPanel* hilft bei der Integration von JavaFX in Swing. Doch wie lässt sich der umgekehrte Weg realisieren? Bisher war eine Integration von Swing nach JavaFX nicht möglich, aber mit der neuen JavaFX Komponente namens *SwingNode*, die in dem *JavaSE8* enthalten ist, soll das möglich werden.

3.1 Von Swing nach JavaFX

Oracle hat auf die Nachfrage der Entwicklergemeinschaft reagiert und einen Migrationspfad zur Verfügung gestellt, den sogenannten *SwingNode*. Wie in Kapitel 3.8 bereits besprochen wurde, gibt es ein neues Rendering-Thread Konzept in JavaFX. Daher muss man auch bei der Migration von Swing und JavaFX beachten, dass man es mit zwei unabhängigen Rendering-Threads zu tun hat. JavaFX nutzt den *JavaFX Application Thread* und Swing den *Event Dispatch Thread*. Durch diese Unabhängigkeit der beiden Threads ist der Entwickler gezwungen, sich selbst um die Interaktion zwischen den beiden Threads zu kümmern. Die entsprechenden Methoden müssen in die jeweiligen *run()* Methoden implementiert werden. Für JavaFX benutzt man die *Platform.runLater(Runnable r)*-Methode und in Swing wäre es die *SwingUtilities.invokeLater(Runnable r)* Methode. [Grunwald \(2013\)](#)

3.1.1 Verfahren

Es gibt bisher noch keine Tools, die das Verfahren der Migration von Swing in JavaFX erleichtern. Der beste Weg bisher ist es, einzelne Swing Komponente schrittweise in die JavaFX-Applikation einzubinden. Für diese Aufgabe bietet Oracle ab Java 8 den *SwingNode* an. Mit diesem lassen sich Swing-Komponenten einfügen, die dann in die JavaFX Applikation eingebunden werden. Der JavaFX *SwingNode* ist ein Bestandteil der JavaFX-Hierarchie und dem *Node* untergeordnet.



(a) SwingNode Vererbungshierarchie

(b) Die verschiedenen Schichten

Quelle: [arnaud nouard \(2012\)](#)

Abbildung 3.1: Der JavaFX SwingNode

Für die Integration von Swing-Komponenten in JavaFX gibt es die *setContent*-Methode. Durch *setContent* wird der *SwingNode* eine *JComponent* hinzugefügt. Dabei muss beachtet werden, dass das Setzen der Komponente auf dem *Swing Event Dispatcher Thread* erfolgt, da sonst standardmäßig der JavaFX-Thread verwendet wird und dies zu einer *Exception* führen kann. Im Listing 3.1 wird die Verwendung der *SwingNode* Klasse an einem Beispiel veranschaulicht.

```

1 public class SwingNodeBsp1 extends Application {
2
3     private JButton swingButton;
4     private Button javaFXButton;
5     private Label javaFXLabel;
6
7     @Override
8     public void start (Stage stage) {

```

```
9      final SwingNode swingNode = new SwingNode();
10     javaFXLabel = new Label();
11     javaFXButton = new Button("JavaFX_Button");
12     javaFXButton.setOnAction(new EventHandler<ActionEvent>() {
13         @Override
14         public void handle(ActionEvent t) {
15             javaFXLabel.setText("JavaFX_Button_wurde_gedrückt!");
16         }
17     });
18
19     swingButton = new JButton("Swing_Button");
20     createSwingContent(swingNode);
21
22     FlowPane pane = new FlowPane();
23     pane.getChildren().add(javaFXButton);
24     pane.getChildren().add(swingNode);
25     pane.getChildren().add(javaFXLabel);
26
27     stage.setTitle("SwingNode");
28     stage.setScene(new Scene(pane, 200, 100));
29     stage.show();
30 }
31
32 private void createSwingContent(final SwingNode swingNode) {
33     SwingUtilities.invokeLater(new Runnable() {
34         @Override
35         public void run() {
36             swingButton.addActionListener(new ActionListener() {
37                 @Override
38                 public void actionPerformed(java.awt.event.ActionEvent e) {
39
40                     // FX Application Thread Aufruf
41                     Platform.runLater(new Runnable() {
42                         @Override
43                         public void run() {
44                             javaFXLabel.setText("Swing_Button_wurde_gedrückt!");
45                         }
46                     });
47                 }
48             });
49 }
```

```
49     swingNode.setContent(swingButton);
50     }
51     });
52     }
53
54     public static void main(String[] args) {
55         launch(args);
56     }
57 }
```

Listing 3.1: SwingNode Beispiel, Swing in JavaFX

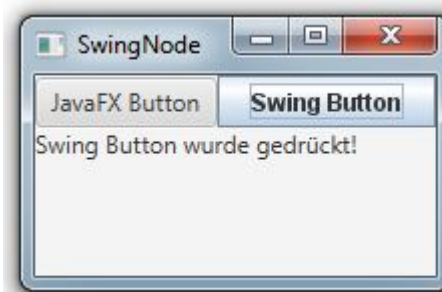


Abbildung 3.2: Integration von Swing JButton in JavaFX Applikation

In dem Codebeispiel von Listing 3.1 wird eine einfache Swing-Komponente, in diesem Fall der JButton in eine JavaFX-Applikation integriert. Außerdem wird noch ein JavaFX-Button erzeugt. Der Benutzer kann jetzt einen der beiden Buttons anklicken. Auf den ausgelösten Event wird reagiert und ein Text mit der Information, welcher Button benutzt wurde, anhand des *labels* angezeigt.

Es wird zunächst ein *SwingNode* Objekt erstellt (Zeile 9). Die Methode *createSwingContent* fügt der *SwingNode* eine Swing-Komponente hinzu. Der *setContent*-Methodenaufruf der *SwingNode* Klasse wird im *Swing Event Dispatcher Thread* ausgeführt, um mögliche *Exceptions* zu verhindern. Als letzter Schritt wird der erzeugte *SwingNode* mit der beinhalteten Swing-Komponente an den JavaFX *Scene Graph* angefügt.

3.1.2 Fazit

Die Möglichkeit, mithilfe der *SwingNode* Swing-Komponenten in JavaFX einzubinden, existiert erst seit Kurzem und steht noch am Anfang. Es wird aber nicht lang dauern, bis eine stabile Version veröffentlicht wird, da die Nachfrage groß ist. [Grunwald \(2013\)](#)

Wie gezeigt wurde, ist es generell möglich, bestehende Swing Anwendungen in JavaFX zu integrieren. Jedoch ergeben sich aktuell immer wieder noch Fehler bei der Darstellung der Swing-Komponente in JavaFX. Manchmal fehlt die *ScrollBar* oder die Komponente wurde nicht richtig gerendert. Auch bei diesem einfachen Beispiel lief das Rendering nicht ohne Probleme; manchmal wurde die Swing-Komponente mit einer kleineren Verzögerung dargestellt, oder der Bereich für Swing-Komponente war einfach nur Schwarz und ließ sich mit einem *resize* des Anzeige-Fensters erst richtig darstellen. Einen großen Aufwand stellt es dar, wenn man sein Swing-Datenmodell in JavaFX integriert und die *Listener* noch darin enthalten sind. Wenn man sich dafür entscheidet, mit den JavaFX-Properties zu arbeiten, ist eine Erweiterung des Datenmodells unvermeidbar.

4 Analyse

4.1 Vorteile von JavaFX

Grafik-System

Das Grafik-System in JavaFX verfügt über eine direkte Verbindung zu Hardwarebeschleunigungen wie zum Beispiel *DirectX* und *OpenGL* für die verschiedenen Betriebssysteme Windows, MacOS und Linux.

Wenn auf dem System keine *DirectX*- oder *OpenGL*-Bibliothek vorhanden ist, wird mit Java2D das Rendering auf Softwareebene unternommen.

Das neue *RetainedMode*-Rendering kümmert sich automatisch um das Neuzeichnen eigener GUI-Komponenten. In Swing musste der Entwickler sich noch selbst darum kümmern, wann seine selbst entwickelten GUI-Komponenten gerendert wurden.

Arbeitsteilung

Die Arbeitsteilung zwischen Design und Entwicklung lässt sich klarer trennen und ermöglicht so einen effizienteren *Workflow*. Da die Möglichkeit besteht eine GUI deklarativ zu erstellen mit Hilfe von FXML. Genauso wie man durch FXML, die GUI vom eigentlichen Java Programmcode kapselt, besteht die Möglichkeit CSS-Dateien in JavaFX einzubinden und somit die Darstellung der GUI-Elemente zu verändern. Dadurch kann der Entwickler den Aufbau der GUI, sowie die Darstellung von GUI-Elementen vom Java Programmcode trennen.

GUI-Elemente

In JavaFX gibt es ein breites Angebot an GUI-Komponenten. In Kapitel 2.6 wurde auf die *Charts* eingegangen. In *Swing* sind *Charts* kein Bestandteil der Bibliothek. Es besteht die Möglichkeit, für *Swing* externe Bibliotheken einzubinden, die es ermöglichen, *Charts* zu erstellen. Der Vorteil, dass *Charts* in JavaFX bereits zur Standardbibliothek gehören, liegt darin, dass man sich sicher sein kann, dass der *support* bzw. *updates* nicht so kurzlebig sind.

Auch mit der *WebView* oder dem *DatePicker* werden dem Entwickler komplexe GUI-Elemente

zur Verfügung gestellt, die man nur noch in die vorhandene Anwendung integrieren muss. Dazu erstellt man das gewünschte GUI-Element; diese erstellte *node* fügt man an den *scene graph*. Im Listing 2.6 wird die Implementation des *DatePicker* dargestellt.

Deployment

Auch das Deployment (Softwareverteilung) läuft einfacher ab. Nutzt man für die Entwicklung von JavaFX Applikationen die NetBeans IDE, so werden automatisch die *packages* für die verschiedenen Distributionen bereitgestellt.

Eventhandling

Das Konzept für das *Eventhandling* unterscheidet sich grundlegend von dem in *Swing*. In JavaFX gibt es eine Event-Hierarchie. Der Vorteil dieser Event-Hierarchie liegt darin, dass ein Event-Handler mit der gleichen Methode auf die verschiedenen Events eines Teilbaums reagieren kann.

Wenn man beispielsweise eine Tabelle mit 1000 Zeilen hat, in der jede Zeile auf einen Maus-Klick reagieren soll, dann kann man dem übergeordneten Element des Hierarchiebaums einen *Event Handler* zuweisen. Dieses Vorgehen wirkt sich positiv auf Performance und den Speicherverbrauch aus, da man nicht mehr über alle Zeilen iterieren muss und sich die entsprechende Anzahl an *Event Handler* spart.

Collections

Die Java Collections werden durch die JavaFX Observable List und Observable Map erweitert.

4.2 Nachteile von JavaFX

JavaFX ist zum Zeitpunkt dieser Arbeit noch relativ jung. Damit verbunden gibt es auch noch keine Application Framework von denen man anhand von Erfahrungen sagen kann, dass sie stabil unter JavaFX laufen würden.

Beim Arbeiten mit dem Scene Builder-Tool kommt es vereinzelt vor, dass sich das Tool einfach ohne eine Fehlermeldung schließt. Es gibt aber regelmäßige Update um die Software stabil laufen zu lassen.

Für die Migration von Swing-Anwendungen in JavaFX gibt es bisher noch kein Verfahren beziehungsweise Tool, die es ermöglicht eine ganze Swing-Anwendung in JavaFX zu migrieren. Mit der *SwingNode* hat man lediglich einen ersten Ansatz geschaffen, einzelne Swing-Komponenten

in JavaFX zu integrieren.

4.3 Gegenüberstellung von JavaFX und Swing

Sowohl mit Swing als auch mit JavaFX lassen sich graphische Benutzeroberflächen für Desktop-Anwendungen erstellen. *Oracle* hat die Entwicklung von Swing eingestellt und bietet JavaFX als Standard-UI-Toolkit für Java an. Im Rahmen dieser Arbeit wurde gezeigt, dass JavaFX die Möglichkeiten von Swing abdeckt und auch neue Konzepte enthält.

Während die Architektur in Swing von dem Vorgänger AWT übernommen wurde, ist der Aufbau der Architektur in JavaFX von Grund auf neu. Erfahrungen über die Architektur, die in Swing gemacht wurden, hat man versucht, in JavaFX zu optimieren. Dazu gehört zum Beispiel, dass beim *Glass Windowing Toolkit* die *event-queue* im gleichen Thread wie die JavaFX-Applikation läuft und nicht wie in Swing in zwei separaten Threads dafür zuständig ist. [Castillo \(2013b\)](#)

GUI-Steuerelemente, die man von Swing kennt, sind auch in JavaFX vorhanden und wurden noch beispielsweise um GUI-Elemente wie den *DatePicker*, *HTMLEditor*, *ColorPicker* und *Charts* erweitert.

Beim Thema Layouts liegt der Hauptunterschied darin, dass man in JavaFX, anders als in Swing, mit *Layout-Panes* arbeitet. Wie in Kapitel 2.9 gezeigt wurde, gibt es für jeden Layout-Typen eine *Pane*. Es wird also nicht mehr wie in Swing ein *JPanel*-Objekt erzeugt, dem man dann einen Layout-Manager zuweist. Für die unterschiedlichen Layout-Typen in Swing gibt es in JavaFX jeweils eine *Layout-Pane* mit dem gleichen Verhalten. Auch hier gibt es in JavaFX eine Erweiterung neuer Layout-Typen wie zum Beispiel die *TabPane*.

In Kapitel 2.13 wurde auf das Thema Events und das Eventhandling in JavaFX eingegangen. Eine solche Eventstruktur, wie sie in diesem Kapitel vorgestellt wird, ist in Swing nicht vorhanden. Somit ist JavaFX mit dem Konzept der Event-Hierarchie gegenüber Swing klar im Vorteil, weil man das Eventhandling auf eine übergeordnete Hierarchie-Ebene auslagern kann und sich somit die Abarbeitung der Events auf der darunter liegenden Ebene spart.

Die Möglichkeit, eine GUI deklarativ aufzubauen, gibt es in Swing nicht. In JavaFX wird hierzu FXML verwendet somit ist es nicht mehr notwendig, Java Programmcode für die Er-

stellung einer GUI zu verwenden. Durch das *Scene Builder*-Tool lassen sich somit grafische Oberflächen zusammenklicken. Gleichzeitig erhält man eine klare Trennung zwischen dem Layout und dem Code der Anwendung. Hinzu kommt noch die Möglichkeit, dass man in JavaFX das Aussehen der GUI-Komponenten durch CSS beeinflussen kann, wie man es aus der Webentwicklung kennt.

Die Fähigkeit der Multi-Touch-Integration ist in der Standardbibliothek von Swing nicht vorhanden; es gibt hier aber die Möglichkeit, auf Frameworks von externen Anbietern zurückzugreifen. Der Nachteil bei dem Benutzen inoffizieller Frameworks besteht darin, dass dem Entwickler keine Garantien vorliegen, dass die Software aktuell gehalten wird und sich dem neuesten Stand der Softwaretechnologie anpasst.

4.4 Stabilität der Anwendung

Um die Stabilität einer Anwendung zu beurteilen, muss man eine vorhandene Anwendung ausgiebigen Testfällen unterziehen. In diesem Abschnitt werden Projekte von Unternehmen vorgestellt, die sich bereits entschieden haben, JavaFX für ihre Anwendung zu benutzen.

eteoBoard

Das *eteoBoard* (Systems, 2013) ist ein interaktiv bedienbares *Scrum Board* für verteilte agile Softwareentwicklung. Entwickelt wurde es von dem Unternehmen *Saxsonia Systems*. Die Bedienoberfläche wurde vollständig mit JavaFX realisiert. Dabei steht die intuitive Bedienung mittels Multi-Touch im Mittelpunkt.



Abbildung 4.1: Das eteoBoard. Quelle: Systems (2013)

Celer

Die Firma *Celer Technologies* ist ein globales Finanzsoftwareunternehmen, das das Software-Framework *Celer* für Handelssysteme entwickelt hat.

Nach Auswertung anderer RIA-Technologien entschied sich *Celer* für JavaFX. Die Entwicklung mit Swing und Adobe Flex kam nicht in Frage, weil das Unternehmen nicht in Technologien mit begrenzten Wachstumsaussichten investieren wollte. Man entschied sich auch gegen das Browser-basierte Google Web Toolkit (GWT), da es nicht die gleiche Leistung wie Desktop-Anwendungen hat. [Technologies \(2013\)](#)

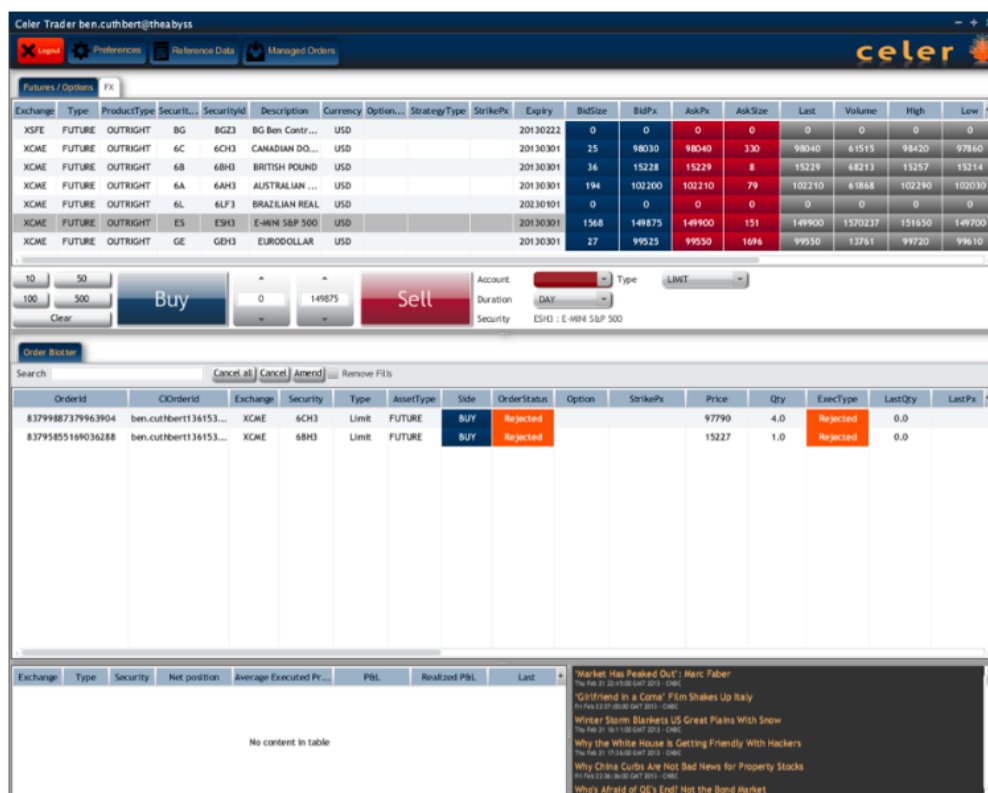


Abbildung 4.2: Celer Software. Quelle: [Technologies \(2013\)](#)

4.5 Application Frameworks

CaptainCasa ist ein *Rich Client Framework* zur effizienten Erstellung von Benutzeroberflächen für Geschäftsanwendungen. Der *Frontend-Client* ist ein JavaFX Client; die Interaktionsverarbeitung findet auf der Server-Seite statt. Die Server-Seite basiert auf *Java Server Faces (JSF)*. Der Hauptunterschied zu anderen Frameworks ist, dass der Client Java nutzt und nicht HTML5. Dieser Aspekt ist dann wichtig, wenn man seine Geschäftsanwendung an viele Firmen verkauft und garantiert, dass die *Frontends* über einen langen Zeitraum hin stabil und Performant laufen. Außerdem bekommt der Client vom Server XML-Masken. Die serverseitige Anwendungsentwicklung hat daher nichts mit Swing oder JavaFX zutun. Deswegen kann man problemlos vom Swing-Client auf den JavaFX Client umsteigen.

Das JRebirth Framework unterstützt Entwickler auf einfache Weise, anspruchsvolle und mächtige JavaFX-Applikationen zu schreiben. Das Framework bietet ein *WSC-MVC* Entwurfsmuster, ein vereinfachtes *Thread Management* und verbraucht gleichzeitig wenig Speicher.

Granite Data Services (Granite DS) stellt eine umfangreiche Entwicklungs- und Integrations-Lösung, für das Programmieren von JavaFX/Java-EE RIA Applikationen bereit. GraniteDS vereinfacht die Entwicklung und erlaubt die Wiederverwendung von existierenden Java-Services. Ein typischer Entwicklungszyklus mit GraniteDS, fängt beim Modellieren von Java-EE-Daten und -Services an. Dann wird mithilfe von Tools der Code generiert und die JavaFX Benutzeroberfläche entwickelt.

JFX Flow ist ein kostenloses *Open Source* Framework für die Entwicklung von *rich, interaktiv* und benutzerfreundlichen *Web-Style* Oberflächen für Desktops, basierend auf JavaFX. Dieses Framework kombiniert die mächtigen Features von JavaFX (styling, Animation, FXML, etc.) mit einer simplen *Web Flow* Darstellung.

Reaction ist ein flexibles, asynchrones Framework, das benutzt werden sollte, wenn man komplexe eventbasierte Applikationen schreibt. Der Fokus der *Reaction* Bibliothek liegt auf dem *Concurrency* und *callback* Modell.

JacpFX Project (Java Asynchronous Client Platform) ist ein Framework, um *Rich Clients* im MVC-style mit JavaFX und Spring zu erzeugen.

5 Schluss

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde die neue Softwaretechnologie JavaFX für den Einsatz zur Erstellung von dialogorientierten Geschäftsanwendungen untersucht. Der Schwerpunkt lag darin, die neuen Konzepte zu überprüfen und auf die Möglichkeiten, die JavaFX bietet, einzugehen.

In Kapitel zwei wurden die Konzepte in JavaFX untersucht. Für den Einstieg wurden die Grundlagen für das Verständnis einer JavaFX-Anwendung vermittelt. Anhand der JavaFX-Architektur wurde das neue Konzept des Graphik-Systems vorgestellt. Ebenso wurde der Scene Graph erläutert. Dieser ist spezifisch für den Aufbau einer JavaFX-Anwendung. Um das Konzept des Aufbaus zu verdeutlichen, erfolgte der Vergleich mit einer Theaterbühne.

Neben der Erklärung zu den einzelnen Phasen des Lebenszyklus und dem Beispiel an einer ersten Applikation wurden auch die verschiedenen GUI-Elemente vorgestellt.

Die Layouts und GUI-Elemente in JavaFX ermöglichen dieselben Funktionalitäten, die man bereits von Swing kennt. Zusätzlich dazu gibt es sowohl neue Layout-Typen als auch GUI-Elemente, die dem Entwickler Programmieraufwand ersparen. In Swing musste man sich beispielsweise für das Darstellen seiner Daten in einem *Chart*, selber die GUI-Komponente programmieren. In JavaFX gibt es für die Erstellung eines *Charts* das *javafx.scene.chart-package*, welches umfangreiche Funktionalitäten in diesem Bereich bietet.

Mit Hilfe von FXML besteht in JavaFX die Möglichkeit, eine GUI deklarativ aufzubauen. Dadurch bietet sich eine klare Trennung zwischen dem Layout und der Programmlogik. Eine wesentliche Erleichterung ist hierfür das Scene Builder-Tool, mit dem man sich eine GUI zusammenklicken kann. Das Tool generiert eine FXML-Datei, die man in sein Programm mittels *ioc* einbindet.

Das Thema Events wurde theoretisch vorgestellt. Es gibt eine ganze Event-Struktur, die dafür

sorgt, wann und wie ein Event sein Ziel erreicht. Mit Hilfe des *Scene Graphs* wurde die Zustellung eines Events erläutert und die damit verbundenen Phasen erklärt. Wichtig zu erwähnen ist der Event-Hierarchie-Baum, mit dessen Hilfe man das Eventhandling auf einer höheren Hierarchie-Ebene abarbeiten kann und sich somit einzelne Zugriffe in der darunter liegenden Hierarchie-Ebene spart.

Das neue Konzept des Properties- und Databinding-Modells wurde in Kapitel 2.12 vorgestellt. Anhand von diesem Konzept wurde auf die neue Möglichkeit eingegangen, das MVVM-Pattern zu realisieren.

Im dritten Kapitel wurde untersucht Swing-Anwendungen in JavaFX zu integrieren. Im Rahmen dessen wurde die *SwingNode*-Komponente vorgestellt, die erst seit *JavaSE 8* Bestandteil von JavaFX ist.

Im vierten Kapitel wurden abschließend die Vor- und Nachteile von JavaFX aufgelistet. Die ausgearbeiteten Informationen über die neue Softwaretechnologie wurden dem Vorgänger Swing gegenübergestellt und festgestellt, dass JavaFX umfangreiche Fähigkeiten für die Entwicklung im Bereich von Desktop-Applikationen besitzt. Da es in dieser Arbeit nicht möglich war, über die Stabilität der Anwendung zu urteilen, wurden Projekte von Unternehmen vorgestellt, die sich bereits für die Entwicklung mit JavaFX entschieden haben.

5.2 Ausblick

Die Arbeit hat gezeigt, dass JavaFX sehr gut für die Entwicklung von Desktop-Anwendungen geeignet ist. Viele GUI-Komponenten, die man bei *Swing* selber programmieren musste, sind schon standardmäßig in der Bibliothek enthalten.

Ein interessanter Aspekt zur Ergänzung dieser Arbeit wäre das Laufzeitverhalten einer JavaFX-Anwendung zu untersuchen und sie dann einer äquivalenten *Swing*-Anwendung gegenüberzustellen. Somit könnte man herausfinden, ob sich die neuen Konzepte auch tatsächlich positiv auf das Laufzeitverhalten auswirken, indem man zum Beispiel den Event-Hierarchie-Baum testet.

Im Rahmen dieser Arbeit wurde auf die verschiedenen GUI-Elemente eingegangen. Zusätzlich könnte man die Erstellung eigener GUI-Komponenten untersuchen. Anhand von Beispielen könnte man prüfen, wieviel Aufwand für die Erstellung eigener GUI-Komponenten benötigt wird.

Ein weiterer wichtiger offener Punkt ist es, Untersuchungen von JavaFX-Anwendungen durchzuführen, um konkrete Aussagen über die Stabilität zu machen. Dazu muss man eine Anwendung verschiedenen Testfällen unterziehen.

JavaFX soll sich zukünftig nicht nur auf Desktop-Geräte beschränken, sondern auch auf Smartphones und Embedded-Geräten laufen. Besonders interessant ist es, wie es in Zukunft mit der Entwicklung von JavaFX im mobilen Bereich aussieht. Mobile Anwendungen können über die *WebView-Node* als Anwendung mit Web-Technologien entwickelt werden, die dann einfach in das JavaFX-Programm integriert werden. Der Vorteil hier liegt darin, dass JavaFX CSS und die *WebKit Rendering Engine* einsetzt. Google Chrome benutzt beispielsweise die gleiche Engine. Laut dem JavaFX-Leiter Richard Bair seien JavaFX-Ports für iOS und Android auf dem Niveau guter Prototypen (Schlosser, 2013). Bis es die ersten ernsthaften Applikationen für Smartphones gibt, werden noch viele Tests nötig sein.

Anhang

A.1 Voraussetzung zum Ausführen von JavaFX

Platform	CPU Architektur	Version
Windows Vista	x86 (32 und 64 bit)	SP2
Windows XP	x86 32-bit	SP3
Windows 7	x86 (32 und 64 bit)	SP1
Windows 8	x64 (32 und 64 bit)	
Max OS X	64 bit	10.7.3 oder höher
Linux	32 und 64 bit	Ubuntu 10.4+, gtk2 2.18+

Mit Java 8 hat sich auch die Versionsnummer von JavaFX 2.x angepasst. Um die gesamten UI-Steuererelemente von JavaFX in vollem Umfang nutzen zu können, muss auf dem System mindestens die Java 8 Version installiert sein. Ab Java 8 wird die Graphikbibliothek *Swing* durch JavaFX abgelöst. [Ebbers \(2013\)](#)

JavaFX ist seit Java 8 standardmäßig sowohl im JDK (*Java Development Kit*) als auch JRE (*Java Runtime Environments*) enthalten.

Entwicklungsumgebung mit Java 8 Unterstützung

Für die Entwicklung von JavaFX-Anwendungen sollte eine geeignete Entwicklungsumgebung vorhanden sein. Folgende *IDE*-Versionen unterstützen Java 8:

NetBeans 8

Eclipse Luna

IntelliJ IDEA 13.1

A.2 Laborumgebung

Als Entwicklungsumgebung (IDE) wird die aktuellste Betaversion von NetBeans (NetBeans IDE 8.0 Beta) mit der aktuellen Beta von Java 1.8 und dem Scene Builder 2.0 benutzt. Die hier verwendeten Programmierbeispiele laufen auf einem Rechner mit dem 64-Bit Windows 7 Betriebssystem. Der Rechner besitzt einen *Core 2 Duo CPU* mit 1.83 GHz und 3 GB Arbeitsspeicher.

Abbildungsverzeichnis

2.1	JavaFX Scene Builder. Quelle: Castillo und Joan (2013)	4
2.2	JavaFX Architektur Diagramm. Quelle: Castillo (2013b)	5
2.3	Der JavaFX Scene Graph	6
2.4	Scene Graph	7
2.5	Abbildung zum Listing 2.1	9
2.6	Chart Aufbau und Überblick	13
2.7	Pie Chart zum Codebeispiel	15
2.8	BorderPane Beispiel	16
2.9	Scene Graph	17
2.10	GridPane Beispiel	18
2.11	TabPane Beispiel	18
2.12	FlowPane Beispiel	19
2.13	DatePicker Beispiel	22
2.14	Inversion of Control Funktionsweise	26
2.15	Event Type Hierarchy. Quelle: Gordon (2013)	32
2.16	Beispiel für die <i>Event Dispatch Chain</i> . Quelle: (Gordon, 2013)	34
2.17	UML ObservableMap. Quelle: James L. Weaver und Dean Iverson (2012)	35
2.18	UML ObservableList. Quelle: James L. Weaver und Dean Iverson (2012)	36
2.19	Ausgabe des Listings 2.16	37
2.20	Unterschied zwischen MVC und MVVM	43
2.21	Begrüßungstext	44
3.1	Der JavaFX SwingNode	48
3.2	Integration von Swing JButton in JavaFX Applikation	50
4.1	Das eteoBoard. Quelle: Systems (2013)	56
4.2	Celer Software. Quelle: Technologies (2013)	57

Listings

2.1	JavaFX Applikation, Hello World Beispiel	8
2.2	Pie Chart, Beispielcode	14
2.3	Layout Manager Beispiel in Swing	19
2.4	Layout Pane Beispiel in JavaFX	19
2.5	style1.css	20
2.6	style2.css	20
2.7	Das Hauptprogramm; dynamisches Laden der CSS-Datei zur Laufzeit	21
2.8	XML-Datei FXMLHelloWorld.fxml	23
2.9	Die Controller-Klasse, FXMLHelloWorldController	24
2.10	Das Hauptprogramm, JavaFXApplication.java	25
2.11	Properties Beispiel, Kontostand	27
2.12	Properties Beispiel, Kontostand	28
2.13	Binding Beispiel, Binding zwischen zwei Werten	30
2.14	Event Handling, convenience-Methoden Beispiel	33
2.15	Event Handling, JavaFX Beispiel	33
2.16	Collections, ObservableList Beispiel	37
2.17	Eine Task starten, Methode 1	39
2.18	Eine Task starten, Methode 2	39
2.19	Multi-Touch Beispiel, Geste zur Drehung der Gruppe	40
2.20	Multi-Touch Beispiel, Geste zur Skalierung der Gruppe	41
2.21	Das Datenmodell, Vor-Nachname einer Person	43
2.22	Das ViewModel	44
2.23	Die WelcomeView-Klasse	44
2.24	FXML-Datei, beinhaltet ein Label-Element mit ID	45
3.1	SwingNode Beispiel, Swing in JavaFX	48

Literaturverzeichnis

- [Casall 2013] CASALL, Alexander: FXML und Data Binding mit JavaFX. In: *Java Magazin* (2013), S. 74–77. – Ausgabe: November 2013
- [Castillo 2013a] CASTILLO, Cindy: *Introduction to JavaFX Media*. 2013. – URL <http://docs.oracle.com/javafx/2/media/overview.htm>. – Zugriffsdatum: 1. April 2014
- [Castillo 2013b] CASTILLO, Cindy: *JavaFX Architecture*. 2013. – URL <http://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm>. – Zugriffsdatum: 1. April 2014
- [Castillo und Joan 2013] CASTILLO, Cindy ; JOAN, Yves: *Getting Started with JavaFX Scene Builder*. 2013. – URL http://docs.oracle.com/javafx/scenebuilder/1/get_started/prepare-for-tutorial.htm. – Zugriffsdatum: 1. April 2014
- [Dea 2011] DEA, Carl: *JavaFX 2.0 Introduction by Example*. Apress, 2011
- [Ebberts 2013] EBBERS, Hendrik: *JavaFX 8- Was ist neu?* 2013. – URL <http://jaxenter.de/artikel/JavaFX-8-Was-ist-neu-172123>. – Zugriffsdatum: 10. April 2014
- [Gordon 2013] GORDON, Joni: *Handling JavaFX Events*. 2013. – URL <http://docs.oracle.com/javafx/2/events/processing.htm#CEGJAAFD>. – Zugriffsdatum: 18. Februar 2014
- [Gordon und Kouznetsov 2013] GORDON, Joni ; KOUZNETSOV, Alexander: *Skinning JavaFX Applications with CSS*. 2013. – URL http://docs.oracle.com/javafx/2/css_tutorial/jfxpub-css_tutorial.htm. – Zugriffsdatum: 3. März 2014
- [Grunwald 2013] GRUNWALD, Gerrit: SwingNode. In: *Java Magazin* (2013), S. 12–13. – Ausgabe: August 2013

- [Grunwald 2014] GRUNWALD, Gerrit: *heise Developer - Visualisierung in Java mit JavaFX*. 2014. – URL <http://www.heise.de/developer/artikel/Visualisierung-in-Java-mit-JavaFX-1902233.html>. – Zugriffsdatum: 17. Februar 2014
- [ITWissen 2014] ITWISSEN: *IoC (inversion of control)*. 2014. – URL <http://www.itwissen.info/definition/lexikon/IoC-inversion-of-control-Umkehrung-des-Kontrollflusses.html>
- [James L. Weaver und Dean Iversen 2012] JAMES L. WEAVER, Ph.D. Stephen C. ; DEAN IVERSON, Ph.D.: *Pro JavaFX 2 A Definitive Guide To Rich Clients with Java Technology*. Apress, 2012
- [Kaufmann 2007] KAUFMANN, Benjamin: *Das Model-View-Controller Modell*. 2007. – URL <http://fara.cs.uni-potsdam.de/~kaufmann/tuts/mvc.pdf>. – Zugriffsdatum: 24. März 2014
- [arnaud nouard 2012] NOUARD arnaud: *In-SideFX*. (2012). – URL <http://arnaudnouard.wordpress.com/2012/12/>
- [Redko 2013] REDKO, Alla: *Introduction to JavaFX Charts*. 2013. – URL <http://docs.oracle.com/javafx/2/charts/chart-overview.htm#CJAHHJCB>. – Zugriffsdatum: 15. März 2014
- [Schlosser 2013] SCHLOSSER, Hartmut: *6 Erfolgskriterien für JavaFX auf Android und iOS*. 2013. – URL <http://jaxenter.de/news/6-Erfolgskriterien-fuer-JavaFX-auf-Android-ios-168252>. – Zugriffsdatum: 3. April 2014
- [Systems 2013] SYSTEMS, Saxonia: *eteoBoard*. 2013. – URL <http://www.eteoboard.de/de/>
- [Technologies 2013] TECHNOLOGIES, Celer: *Celer Technologies Has a Winning Trading Strategy*. 2013. – URL <http://www.oracle.com/technetwork/java/javafx/celer-tech-1844342.html>
- [Ullenboom 2011] ULLENBOOM, Christian: *Java ist auch eine Insel*. Galileo Computing, 2011
- [Wikipedia 2014] WIKIPEDIA: *Container (Informatik)*. 2014. – URL [http://de.wikipedia.org/w/index.php?title=Container_\(Informatik\)&oldid=127558947](http://de.wikipedia.org/w/index.php?title=Container_(Informatik)&oldid=127558947)

