Hochschule
Ravensburg-Weingarten
Technik | Wirtschaft | Sozialwesen

Evaluierung der Eignung von JavaFX 2 als UI Technologie für das GUI-Framework Jo Widgets anhand einer prototypischen Implementierung des Jo Widgets Service Provider Interfaces

David Bauknecht
19218
Weingarten, 1. Juni 2012

Bachelorarbeit

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

an der

Hochschule Ravensburg-Weingarten
Technik | Wirtschaft | Sozialwesen

Fakultät Elektrotechnik und Informatik
Studiengang Angewandte Informatik

Thema: **Evaluierung der Eignung von JavaFX 2 als UI Technologie für das GUI-Framework Jo Widgets anhand einer prototypischen Implementierung des Jo Widgets Service Provider Interfaces**

Verfasser: David Bauknecht
Finkenweg 21
88250 Weingarten

1. Prüfer: Prof. Dr.-Ing. Silvia Keller
Hochschule Ravensburg-Weingarten
Doggenriedstraße
88250 Weingarten

2. Prüfer: Dipl.-Inf. Michael Großmann
innoSysTec GmbH
In Oberwiesen 16
88682 Salem-Neufrach

Abgabedatum: 1. Juni 2012

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Die Arbeit wurde bisher, in gleicher oder ähnlicher Form, keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Unterschrift

Ort, Datum

Abstract

Diese Bachelorabeit untersucht, ob sich JavaFX 2 für die Implementierung des Service Provider Interfaces des Jo Widgets Framework eignet. Dazu wird ein Prototyp des Service Providers implementiert. Es werden die beiden Technologien JavaFX 2 und Jo Widgets untersucht und die für die Bachelorarbeit relevanten Punkte vorgestellt. Die Grundlagen des Layoutings werden erklärt, da das Thema in der Entwicklung des Prototyps eine größere Rolle einnimmt. Danach werden die Anforderungen an den Prototyp festgelegt und die Motivation hinter dieser Arbeit beschrieben. Im Kapitel Realisierung wird dann die Implementierung des Service Providers erklärt.

Abschließend wird eine Bewertung des Prototyps abgegeben und ein Ausblick auf eine vollständige Implementierung geboten.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Nach dem *TIOBE Programming Community Index*[1] ist Java eine der meistgenutzten Programmiersprachen überhaupt, was sicher auch an der Plattformunabhängigkeit der Sprache liegt. Mit Swing bietet die Java Standard Edition bereits eine Technologie für die Entwicklung von Oberflächen. Die Firma IBM entwickelte 2001 dennoch das Standard Widget Toolkit (SWT), welches gegenüber Swing gewisse Vorteile, wie unter anderem ein natives Look and Feel, bringen soll [Fei06]. Leider sind Applikationen, welche mit SWT entwickelt werden, nur bedingt[2] mit Swing kompatibel.

Mit dem neuen JavaFX 2 will Oracle Swing langfristig ablösen [Sch12]. Die Kompatibilität zwischen JavaFX 2 und Swing/SWT ist ebenfalls nur durch ein Bridging Mechanismus[3] gegeben. Selbst wenn bei den Bridging Mechanismen technisch keine Probleme mehr bestehen, wirken Komponenten, die auf unterschiedlichen Technologien basieren, oft wie ein „Fremdkörper“, da sie ein unterschiedliches Look and Feel besitzen.

Auf Grund der Vielzahl an existierenden GUI (grafisches User Interface) Frameworks[4] wird klar, dass trotz der Plattformunabhänigkeit von Java für die Entwicklung von Oberflächen kein einheitlicher Standard existiert. Daher sind Bibliotheken oder Applikationen, welche auf einer bestimmten GUI Technologie basieren, nur bedingt in einem anderen Kontext verwendbar.

In der Regel wird die Entscheidung, welche GUI Technologie zum Einsatz kommt, schon recht früh in der Projektphase getroffen. Im Idealfall werden die Vor- und Nachteile der jeweiligen GUI Technologien in Bezug auf die Anforderungen des Projekts abgewogen. In der Praxis können für die Entscheidung jedoch auch andere Gründe relevant sein, zum Beispiel Firmenstrategie, Know How der Mitarbeiter, Wiederverwendung von bereits existierendem Code. Gerade im Bereich des Enter-

[1] s. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] Es gibt zwar die Möglichkeit der SWT/AWT Bridge, welche aber derzeit noch nicht ausgereift scheint [Hir07]

[3] Mit Hilfe der Klassen FXCanvas für SWT und JFXPanel für Swing [Wea12a, Seite 287]

[4] s. http://de.wikipedia.org/wiki/Liste_von_GUI-Bibliotheken#Java

prise Application Developments[5], wo zum Teil mehrere Mannjahre[6] für ein Projekt aufgewendet werden, ist ein Wechsel der GUI Technologie zu einer späten Projektphase mit extrem hohen Kosten verbunden. Technologische Neuerungen im GUI Bereich sind somit unter Umständen für solche Projekte nicht nutzbar.

Mit *Jo Widgets* bietet die innoSysTec GmbH ein *Single Sourcing*[7] Framework für Java an. Es erlaubt den einfachen Austausch des GUI, wenn die Anwendung mit Jo Widgets entwickelt wurde. Aktuell werden Swing, SWT und QT Jambi[8] als GUI Technologien unterstützt.

1.1 Zielsetzung

Die neue GUI Technologie JavaFX 2 der Firma Oracle bietet ein moderneres Aussehen[9] der Komponenten und weitere Neuerungen, wie zum Beispiel ein verbessertes Rendering, Charting[10] oder die Möglichkeit, die Oberflächen mit Hilfe von Cascading Style Sheets (CSS) anzupassen. Um diese Vorteile für Anwendungen, die auf auf Jo Widgets basieren, nutzbar zu machen, soll in dieser Bachelorarbeit untersucht werden, inwieweit sich JavaFX 2 als GUI Technologie für Jo Widgets eignet. Dazu muss ein Service Provider Plugin implementiert werden. Dies geschieht anhand einer prototypischen Entwicklung. Mit Hilfe des Prototypen soll eine Bewertung für eine vollständige Implementierung hinsichtlich möglicher Probleme und des Aufwands abgegeben werden.

1.2 Gliederung

In den ersten drei Kapiteln werden die Grundlagen für diese Bachelorarbeit erklärt. Darin werden die beiden Technologien JavaFX 2 und Jo Widgets vorgestellt, und auf das Thema Layouting eingegangen. Es folgt die Analyse der prototypischen Umsetzung und abschließend wird eine Bewertung der Umsetzbarkeit gegeben. Die einzelnen Kapitel dieser Arbeit bauen aufeinander auf und sollten deshalb nacheinander gelesen werden.

[5] „An enterprise application is a business application, obviously. As most people use the term, it is a big business application.“ [Mic12]

[6] Bei der Firma innoSysTec GmbH haben Projekte einen Aufwand zwischen 2 und 10 Mannjahren und bestehen aus bis zu 250.000 Zeilen Code [Gro12].

[7] Unter Single Sourcing versteht man die Möglichkeit, einen Quellcode ohne Änderungen, in verschiedenen Laufzeitumgebungen ausführen zu können (siehe Kapitel 3.1.1).

[8] QT unter Java

[9] Look and Feel

[10] Charting ≙ Diagramme

Kapitel 2

JavaFX 2

Im folgenden Kapitel wird das Framework JavaFX 2 mit den wichtigsten Eigenschaften vorgestellt.

2.1 Von F3 zu JavaFX 2

Im Jahr 2005 übernahm Sun Microsystems die Firma SeeBeyond, die eine Skriptsprache, bekannt als F3 (Form Follows Function) entwickelt hat. Hauptverantwortlich für die Sprache war Chris Oliver. Er wechselte mit der Übernahme zu Sun und stellte die Sprache im November 2006 in seinem Webblog zum ersten Mal vor[Oli06].

Im Mai 2007 präsentierte Sun Microsystems sie auf der JavaOne-Konferenz unter dem Namen JavaFX. Sie sollte eine Konkurrenz zu Silverlight aus dem Hause Microsoft und Flash von Adobe darstellen. Sun konnte der Sprache damals nicht zu einem Erfolg verhelfen, was auch daran lag, dass Anwendungen in der Skriptsprache JavaFX Script programmiert werden mussten.

Nachdem Oracle die Firma Sun Mircosystems übernommen hatte, kündigte Oracle auf der JavaOne 2010 den Ausstieg aus JavaFX Script und die Neuentwicklung als native Java-Bibliothek an [Ora11c]. Das erste Software Development Kit (SDK) für JavaFX 2 wurde am 3. Oktober 2011 veröffentlicht. Gleichzeitig wurde bekannt geben, dass es Stück für Stück dem Open Source Projekt OpenJFX übergeben werden soll [Lip11]. Dadurch erhofft sich Oracle eine schnellere Behebung von Fehlern und eine höhere Akzeptanz.

Die Laufzeitumgebung von JavaFX 2[1] ist in der aktuellen Java SE Runtime 7 enthalten. Da JavaFX sich immer noch in der Entwicklung befindet, erscheint im wöchentlichem Rhythmus eine Developer Preview[2]. Des Weiteren werden immer wieder Teile an das Open Source Projekt OpenJFX weitergegeben [Ope12]. JavaFX 3 soll Ende 2013 erscheinen und dann Bestandteil der achten Version des Java SDK werden [Ora11b].

[1] JavaFX 2 wird im Folgenden nur noch JavaFX genannt.

[2] Developer Preview - Hierbei handelt es sich um eine Vorabversion für Entwickler zum Testen, aber nicht für den produktiven Einsatz.

2.2 JavaFX Features

In diesem Kapitel werden diejenigen Eigenschaften von JavaFX näher erläutert, die für die Implementierung des Service Providers relevant sind.

2.2.1 Die Benutzeroberfläche in JavaFX

Das GUI in JavaFX besteht aus drei Komponenten, welche in Abbildung 2.1 dargestellt sind.

Stage

Die `Stage`-Klasse ist der oberste Container eines JavaFX-Programms. Hierbei wird nicht unterschieden, ob die Anwendung auf dem Desktop, im Browser oder auf einem anderem Gerät[3] gestartet wurde. Unter Windows beispielsweise hat die Stage einen Rahmen sowie eine Titelleiste, während sie beim Starten im Browser als Applet gerendert wird [Wea12a, S. 11].

Sie ist für die Größe und Position des Programms zuständig und bietet außerdem die Möglichkeit, den Titel oder die Sichtbarkeit zu setzen. Die erste Stage wird von der JavaFX Plattform beim Starten der Anwendung erzeugt.

Scene

Die `Scene`-Klasse stellt den Container für alle grafischen Elemente dar. Für die Verwaltung der Elemente innerhalb der Scene ist ein Scene Graph verantwortlich. Damit die Scene und ihre Elemente angezeigt werden können, muss sie der `Stage`-Klasse hinzugefügt werden. Außerdem ermöglicht sie, den Cursor oder die Hintergrundfarbe zu setzen.

Node

Die abstrakte `Node`-Klasse im Package `javafx.scene` ist ein Basistyp für alle Elemente im Scene Graph. Jedes `Node`-Objekt darf nur einmal im Scene Graph vorkommen. Vom Root Node, welcher der Scene hinzugefügt wurde, baut sich der gesamte Scene Graph auf.

[3] Nach heutigem Stand (08.04.2012) gibt es JavaFX für Windows, Linux und Mac. Eine Android und iOS Version sollen folgen [Pil11].

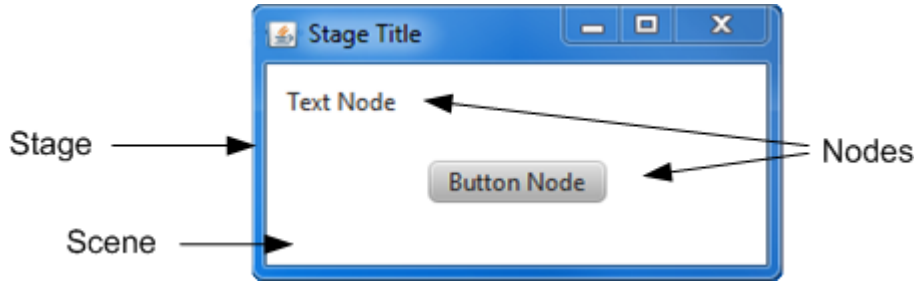

Abbildung 2.1: Erklärung Stage, Scene, Node

2.2.2 Der Scene Graph

Das Scene Graph Application Programming Interface (API)[4] in JavaFX soll die Erstellung von grafischen Benutzeroberflächen erleichtern. Der Scene Graph selbst ist eine Baumstruktur, die Daten enthält. Das Scene Graph API ist ein „Retained mode API“ [5]. Dies bedeutet, dass ein internes Modell alle grafischen Objekte der Anwendung verwaltet. Das Modell übernimmt das Zeichnen der Objekte. Dem Modell ist zu jeder Zeit bekannt, welche Objekte angezeigt, welche Bereiche neu gezeichnet werden müssen und wie die Objekte am effizientesten gerendert werden. Der Entwickler muss sich dabei nicht um das Zeichnen der Elemente kümmern. Dies reduziert die Menge an Code und deren Komplexität.

Die JavaFX Nodes in einem Scene Graph werden in die drei Typen Branch Nodes, Leaf Nodes oder Root Nodes unterteilt. Für jeden Scene Graph gibt es nur einen Root Node. Es ist der erste Knoten im Baum und dieser hat keinen Vaterknoten. Branch Nodes sind Knoten, denen weitere Kinderknoten untergeordnet sein können. Leaf Nodes können keine Kinderknoten haben und sind somit die äußersten Knoten im Baum. Die Baumstruktur ist in Abbildung 2.2 zu sehen.

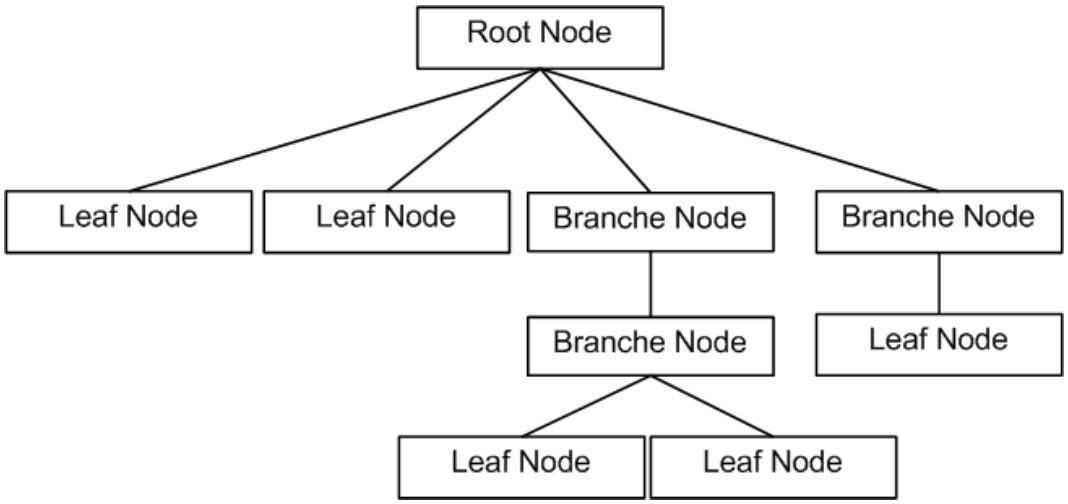

Abbildung 2.2: Scene Graph

[4] API ≙ Eine Programmanbindung auf Quelltextebene [Tho06]
[5] Hierarchische Modellierung der Szene und deren Untereinheiten bzw. Objekte. [Röt11]

2.2.3 Hello World Beispiel

In Listing 2.1 ist das Hello World Beispiel für JavaFX zu sehen.

```
1  public class HelloWorld extends Application {
2
3  @Override
4   public void start(Stage stage) {
5     Group root = new Group();
6     Label label = new Label("Hello World");
7     label.setTextFill(Color.BLUE);
8     root.getChildren().add(label);
9     Scene scene = new Scene(root,300,300, Color.WHITE);
10    stage.setTitle("Hello World");
11    stage.setScene(scene);
12    stage.show();
13  }
14
15  public static void main(String[] args) {
16    launch(args);
17  }
18 }
```

Listing 2.1: Hello World Beispiel

Eine JavaFX Anwendung muss von der Basisklasse `Application` abgeleitet werden. Die `Application`-Klasse besitzt Lebenszyklusmethoden wie `launch(String args)`, `start(Stage stage)`, `stop()` oder `init()`. Bis auf die `launch()`-Methode können alle überschrieben werden. Die `init()`-Methode wird zuerst aufgerufen. Der Entwickler kann sie überschreiben und Initialisierungen vor dem Start der Anwendung veranlassen. Da es sich hierbei um eine leere Methode handelt, wird in dieser kein Code ausgeführt, wenn sie nicht überschrieben worden ist.

Die `main()`-Methode der Anwendung ruft die `launch()`-Methode auf und übergibt ihr alle Kommandozeilenoptionen. Die `launch()`-Methode erzeugt eine Instanz der Klasse, welche sie aufgerufen hat und ruft dann die weiteren Lebenszyklusmethoden auf.

JavaFX ruft in der Anwendung nach der `init()`-Methode nun die `start(Stage stage)`-Methode auf. Aus JavaFX wird ein Stage-Objekt übergeben. Im Beispiel wird der Stage eine Scene zugewiesen und der Titel gesetzt.

Alle Knoten müssen einem Scene Graph hinzugefügt werden. Dieser Scene Graph wird innerhalb eines Container, hier ein Group-Objekt (s. Zeile 5), aufgebaut. Um Elemente hinzuzufügen, bieten Container in JavaFX über die `getChildren()`-Methode Zugriff auf eine Liste mit allen Kinderelementen. Diese liefert eine Liste mit allen Kindern zurück.

Sind alle Elemente dem Scene Graph hinzugefügt, kann dieser nicht direkt auf die Stage gelegt werden. Der Scene Graph muss noch einem Scene-Objekt zugewiesen werden (s. Zeile 11). Dieses wird dann der Stage hinzugefügt. Um das Fenster anzuzeigen wird die `show()`-Methode der Stage aufgerufen.

In Abbildung 2.3 ist die Hello World Anwendung dargestellt.

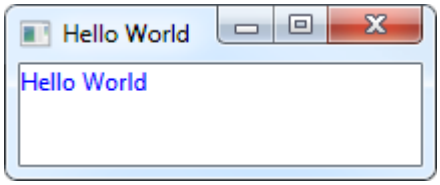

Abbildung 2.3: Einfaches JavaFX Beispiel

2.2.4 Rendering Engine JavaFX

JavaFX hat eine neue Rendering Engine, Prism genannt. Diese wird über das Quantum Toolkit dem Public API zur Verfügung gestellt. Neben Prism gibt es unterhalb des Quantum Toolkit noch das Glass Windowing Toolkit sowie die Media API und die Web Engine. Abbildung 2.4 zeigt die Architektur von JavaFX[6].

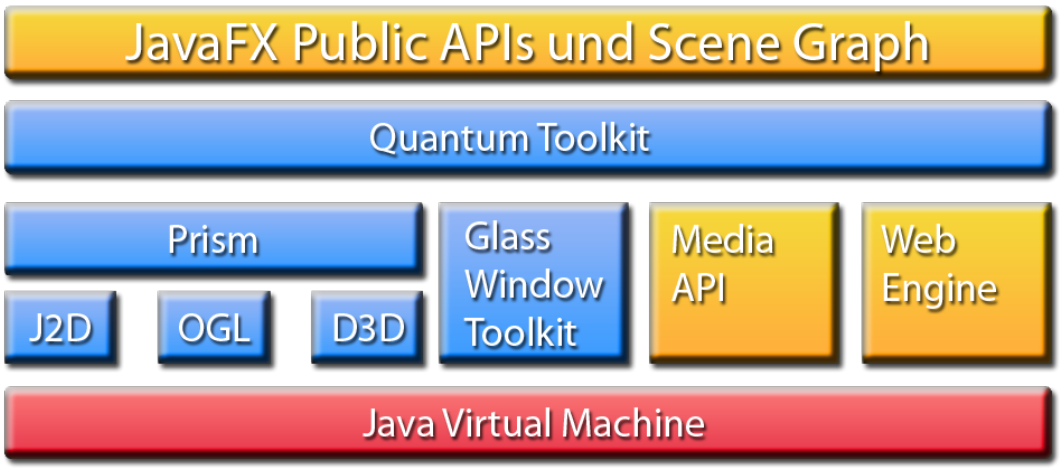

Abbildung 2.4: JavaFX Architektur

Prism

Die Prism Graphics Engine ist für das Rendern des JavaFX Scene Graph zuständig. Zum Rendern wird im Normalfall unter Windows DirectX sowie unter Mac und Linux OpenGL als Hardwarerenderer verwendet. Sollten diese nicht verfügbar sein, wird der Softwarerenderer Java2D API verwendet, der bereits in der Java Runtime Environment vorhanden ist [Cas12b]. Die Engine läuft nicht im JavaFX Application Thread. sondern in einem eigenen Prism Render Thread. Dadurch ist es möglich, dass ein Frame gerendert wird während das andere abgearbeitet wird. Aktuell laufen die beiden Threads noch sychronisiert, aber ab der Version 3.0 sollen sie parallell laufen. Dadurch erhofft man sich Performancesprünge um bis zu 40 Prozent [Ora, Aussage des Oracle Mitarbeiters Richard Bair].

[6] in Anlehnung an http://docs.oracle.com/javafx/2/architecture/img/jfx-arch-diag.gif

Glass Windowing Toolkit

Das Glass Windowing Toolkit ist für das Anzeigen von Fenstern, zum Beispiel der Stage oder von Popups, verantwortlich. Es ist plattformabhängig und verbindet JavaFX mit dem Betriebssystem. Außerdem verwaltet es die Event Queue und leitet die Events an JavaFX weiter [Cas12b].

Das Toolkit verwaltet einen Timer, der alle 16.666 ms auslöst. Für diesen Timer wird entweder ein nativer Timer des Systems verwendet oder JavaFX wartet auf die Rückmeldung des Betriebssystems, dass die Grafikkarte das Bild erneuert hat. JavaFX überprüft bei jedem Feuern ob bereits ein Pulse in der Queue vorhanden ist oder gerade abgearbeitet wird. Sollte eines von beidem zutreffen, wird solange kein neuer Pulse erzeugt. Ist das Neuzeichen der Scene nötig oder läuft gerade eine Animation ab, wird ein Pulse Event erzeugt und in die Event Queue gelegt [Ora, Aussage des Oracle Mitarbeiters Richard Bair]. Dafür ist das Quantum Toolkit verantwortlich, das im nächste Kapitel näher erläutert wird.

Quantum Toolkit

Das Quantum Toolkit stellt Prism und Glass der öffentlichen JavaFX Schicht zur Verfügung. Es hört auf den Glass Pulse Timer und erzeugt den Pulse Event. Außerdem ist es für das Bearbeiten des Pulse Event zuständig. Dort werden Animation, CSS und das Layout abgearbeitet und danach der Prism Thread für das Rendering angestoßen [Ora, Aussage des Oracle Mitarbeiters Richard Bair].

2.2.5 Aussehen von JavaFX-Objekten mit CSS

JavaFX bietet die Möglichkeit, die grafische Oberfläche mit Hilfe von CSS anzupassen. Die CSS Sprachversion, welche JavaFX verwendet, basiert auf der „W3C CSS version 2.1 specification“ [JG12]. Die JavaFX Runtime beinhaltet eine CSS-Datei[7] mit Default-Werten für das Aussehen der Komponenten. Das CSS ist so strukturiert, dass jeder CSS Selector[8] einer JavaFX-Klasse entspricht. Die Properties der Java-Klasse bekommen im CSS den Präfix `fx-`.[Ora11a]

Um das Aussehen der Komponenten anzupassen, gibt es mehrere Möglichkeiten. Möchte man, ähnlich wie in Swing das Look and Feel der Anwendung austauschen, bietet es sich an, die CSS-Datei aus der JavaFX Runtime als Vorlage zu verwenden und dort die Einstellungen anzupassen. Der Vorteil gegenüber dem Look and Feel von Swing ist, dass dazu keine Java Kenntnisse nötig sind und auch Designer diese Aufgabe übernehmen können, sofern sie die CSS-Syntax beherrschen.

Dadurch lässt sich auf einfachem Weg das Aussehen von Anwendungen an Kundenwünsche oder Betriebssysteme anpassen. So wurde bereits das Aussehen von iOS Komponenten nachgebaut [sof12].

[7] Die caspian.css im Verzeichnis **„com/sun/javafx/scene/control/skin/caspian/“ der jfxrt.jar**

[8] „Als Selektoren wird das bezeichnet, was vor den geschweiften Klammern steht. Ein Selektor wählt aus, wofür die folgenden Definitionen gelten sollen.“ Zitat aus [SEL, Abschnitt Selektor]

Um das Aussehen der Anwendung zuzuweisen, sind keine Anpassung an dem Java Code nötig. Über `scene.getStylesheets().add("aussehen.css");`, werden die Einstellungen aus der CSS-Datei in das `Scene`-Objekt geladen.

CSS-Beispiel

Das Beispiel in Abbildung 2.5 zeigt einen Login-Dialog, links mit Default Einstellungen und rechts mit einer veränderten CSS-Datei. Hierfür muss nur die CSS-Datei in Listing 2.2 geladen werden. Das Beispiel ist angelehnt an die Oracle Dokumentation zum Thema CSS unter JavaFX [Ora12a].

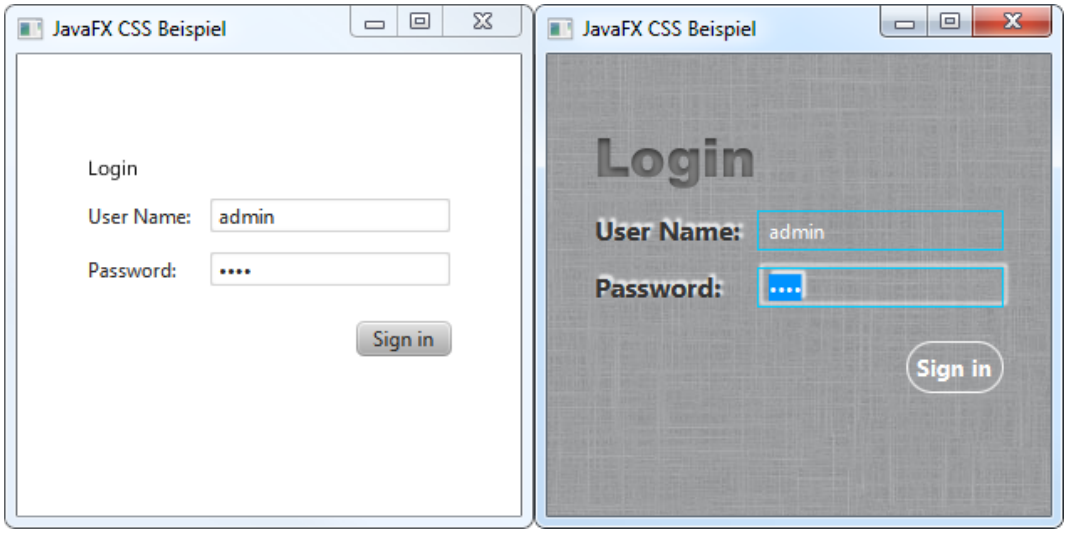

Abbildung 2.5: Styling eines Login Dialogs

In der CSS-Datei wird über den Root Selektor das Hintergrundbild[9] gesetzt und die Standardeinstellungen für die Widgets Label, TextField, PasswordField und Button überschrieben. Für das PasswordField wurde neben den Standardeinstellungen auch die Einstellungen für den „Focused“ Zustand überschrieben. Dort wurde unter anderem ein Schatten über den fx-effect Parameter hinzugefügt (s. Abbildung 2.5). Widgets, welche von der `Node`-Klasse ableiten, können neben dem „Focused“ Zustand auch den Zuständen „Disabled“, „Hover“ und „Pressed“ Einstellungen zuweisen. Unterklassen bieten auch Einstellungen für eigene Zustände, wie zum Beispiel die `Button`-Klasse den „armed“ Zustand. [Ora11a]

Der mit einer „#“ gekennzeichnete Selektor überschreibt keine Standardeinstellungen, sondern ist für Widgets mit dieser ID zuständig.

JavaFX bietet auch die Möglichkeit, CSS-Einstellungen ohne zusätzliche CSS Datei zu setzen. Die `Node`-Klasse bietet dafür die `setStyle()`-Methode. Dieser Methode kann ein String übergeben werden, welcher die Einstellung beschreibt. Die gesetzten Einstellungen gelten ausschließlich für dieses Objekt.

[9] Das Bild stammt aus der Oracle Dokumentation http://docs.oracle.com/javafx/2/get_started/background.jpg

```
.root {
     -fx-background-image: url("background.jpg");
}
.label {
    -fx-font-size: 16px;
    -fx-font-weight: bold;
    -fx-text-fill: #333333;
    -fx-effect: dropshadow( three-pass-box , rgba
       (255,255,255,0.7) , 2, 0.5 , 2 ,-2 );
}
#login-text {
   -fx-font-size: 32px;
   -fx-font-family: "Arial Black";
   -fx-fill: #818181;
   -fx-effect: innershadow( three-pass-box , rgba
      (0,0,0,0.7) , 6, 0.0 , 0 , 2 );
}
.text-field{
   -fx-background-color: transparent;
   -fx-border-color: #00CCFF;
   -fx-font-size: 12px;
   -fx-text-fill: white;
}
.password-field{
   -fx-background-color: transparent;
   -fx-border-color: #00CCFF;
   -fx-font-size: 12px;
   -fx-text-fill: white;
}
.password-field:focused{
   -fx-background-color: transparent;
   -fx-border-color: #00CCFF;
   -fx-text-fill: white;
   -fx-effect: dropshadow( three-pass-box , rgba
      (255,255,255,0.7) , 2, 0.5 , 2 ,-2 );
}
.button{
   -fx-background-color: transparent;
   -fx-border-color: white;
   -fx-background-radius: 30;
   -fx-border-radius: 30;
   -fx-text-fill: white;
   -fx-font-weight: bold;
   -fx-font-size: 14px;
   -fx-padding: 5 5 5 5;
}
```

Listing 2.2: CSS Beispiel

2.3 Weitere JavaFX Features

In diesem Kapitel werden in einem Überblick weitere Features von JavaFX vorgestellt, welche für den Prototypen nicht relevant sind.

2.3.1 Web View

JavaFX bietet ein einfacher Browser basierend auf WebKit[10] mit eigener Web Engine. Die Web Engine ist im Hintergrund tätig während die Web View der eigentlich Browser ist. Momentan wird CSS, JavaScript, DOM und HTML 5 unterstützt [Red12a].

2.3.2 Media API

Die Media API bietet Funktionen zum Abspielen von Audio und Video Dateien. Daraus lassen sich eigene Media Player entwickeln. Aktuell werden allerdings nur wenige Formate unterstützt [Cas12a].

2.3.3 Animationen

Animationen in JavaFX sind in zwei Kategorien, Timeline und Transitions, unterteilt. Sie bieten die Möglichkeiten Animationen auf bestimmten Pfade auszuführen oder sie Framegesteuert ablaufen zu lassen [Wea12a].

2.3.4 3D

Jede Node kann mithilfe von Transformation im drei dimensionalen Raum dargestellt werden. Es sind 3D Objekte wie in Abbildung 2.6[11] möglich.

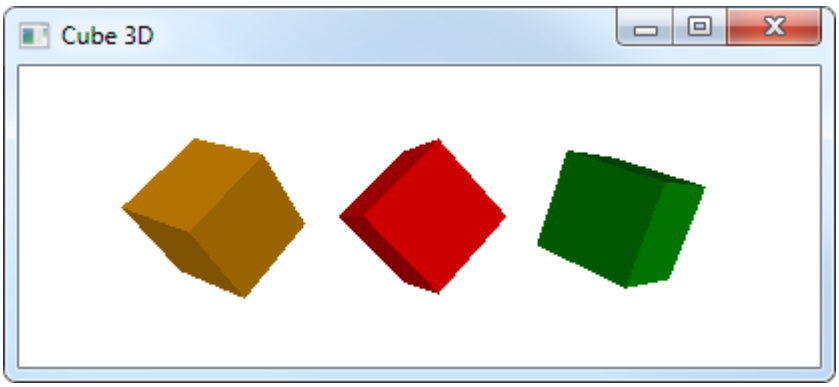

Abbildung 2.6: 3D Würfel

2.3.5 Binding und Properties

JavaFX Properties erweitern das JavaBeans Model und bieten die Möglichkeit die Werten aneinander zu binden. Den Properties können außerdem Listener gesetzt werden, die bei Änderungen reagieren [Hom12].

[10] http://www.webkit.org/
[11] Quellcode von http://fxexperience.com/2011/05/simple-3d-cubes-in-javafx-2-0/

2.3.6 Charts

JavaFX bietet die Möglichkeit Diagramme zu erstellen [Red12b]. In Abbildung 2.7 ist ein Liniendiagramm und in Abbildung 2.8 ist ein Balkendiagramm zu sehen.

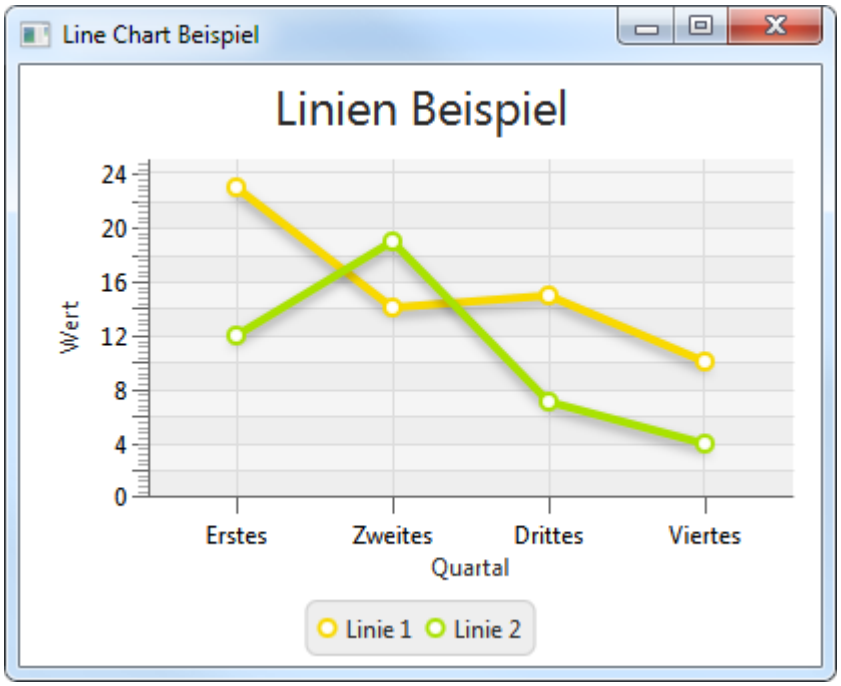

Abbildung 2.7: Liniendiagramm

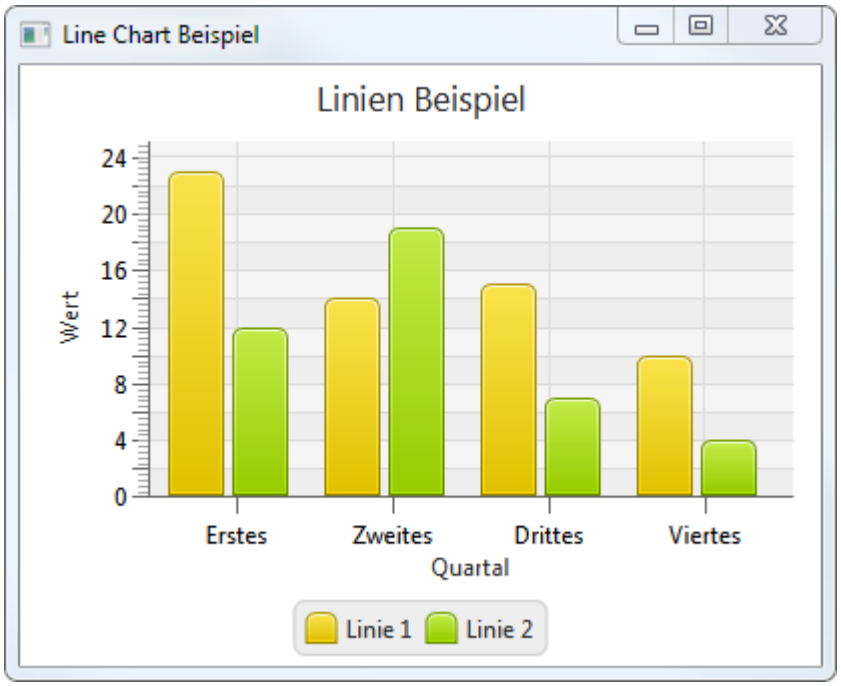

Abbildung 2.8: Balkendiagramm

Kapitel 3

Jo Widgets

Jo Widgets (Java Open Widgets) ist ein von der innoSysTec GmbH entwickeltes Open-Source GUI API für Java. Es steht unter der „New BSD License“ und wird auf Google Code gehostet[1].

3.1 Ziele von Jo Widgets

Die Ziele von Jo Widgets sind Single Sourcing und eine Enterprise GUI API. Die beiden Ziele werden im Folgenden näher beschrieben.

3.1.1 Single Sourcing

Chris Aniszczyk hat Single Sourcing in der Softwareentwicklung folgendermaßen beschrieben[Ani09, Folie 25]:

> „Single source publishing, also known as single sourcing, allows the same source to be used in different runtime environments.“

Demzufolge ist Single Sourcing die Wiederverwendung von Quellcode in unterschiedlichen Laufzeitumgebungen. Die Wiederverwendung von Quellcode ist meist mit einer modularen Entwicklung schon gegeben. Allerdings gibt es hier Einschränkungen, wenn Abhängigkeiten auf eine bestimmte GUI Technologie vorhanden sind.

Bei Single Sourcing sollte diese Abhängigkeit jedoch nicht vorhanden sein. Der Code soll nur gegen Schnittstellen entwickelt werden. An dieser Schnittstelle soll die GUI Technologie dann ansetzen. Somit ist der geschriebene Code nicht von der GUI Technologie abhängig.

Die Entwicklungszeit von großen Unternehmensanwendungen kann mehrere Jahre umfassen. Die GUI Technologie wird dabei sehr früh festgelegt. Im Entwicklungszeit-

[1] http://code.google.com/p/jo-widgets/

raum können neue Technologien erscheinen bzw. die Weiterentwicklung der gewählten Technologie eingestellt werden. Für Unternehmen bietet Single Sourcing damit Investitionssicherheit, da sie bei einem Wechsel der Technologien keinen Verlust des bisher geschriebenen Codes haben bzw. ihn nicht teuer portieren müssen. Zudem können Produkte für mehrere Plattformen angeboten werden, ohne dass sie für jede Plattform separat entwickelt werden müssen.

3.1.2 Enterprise API

Jo Widgets soll ein Enterprise GUI API bieten, das die typischen Aspekte von Unternehmensanwendungen abdeckt. Dadurch soll es helfen, den Aufwand bei der Entwicklung einer Enterprise Anwendung zu verringern, da solche Aspekte von Swing oder SWT nicht sehr weit abgedeckt werden. Zu diesen Aspekten zählen unter anderem:

- **häufig verwendete vorkonfigurierte Widgets:** In Unternehmensanwendungen werden häufig die selben Widgets immer wieder benötigt. Dazu zählen zum Beispiel Eingabefelder für bestimmte Werte wie Zahlen, Datum oder E-Mail-Adressen, Eingabe-Dialoge, Login-Dialoge, Fehler-Feedback-Dialoge und viele mehr.
- **Validierung:** Die Validierung von Eingaben soll möglichst einfach sein. So sollen zum Beispiel der Typ der Eingabe, der Wertebereich, aber auch kundenspezifische Businessregeln validierbar sein.
- **Hoch-produktiv und einfach:** Hierzu wird unter anderem der „Convention over Configuration“[2] Ansatz gewählt. Gewohnte Einstellungen sind schon vorkonfiguriert und müssen nicht immer von neuem eingestellt werden. Zum Beispiel öffnet sich ein Kindfenster standardmäßig zentriert über dem Vaterfenster.
- **Erweiterbarkeit:** Jo Widgets soll einfache und schlanke Schnittstellen bieten, um zum Beispiel leicht mit einer neuen GUI Technologie erweiterbar zu sein oder um eigene Widgets erstellen zu können.

3.2 Entstehung von Jo Widgets

Die Firma innoSysTec GmbH entwickelt vorwiegend CRUD (Create, Read, Update and Delete)[3] Anwendungen. Für diesen Zweck wurde ein auf Swing basierendes Framework entwickelt, welches fertige Lösungen für immer wiederkehrende Aufgaben bietet. Dadurch ist man in der Lage, sehr komplexe und aufwändige Anwendungen sehr schnell zu entwickeln.

Dieses Framework sollte im Rahmen eines gemeinschaftlichen Projektes mit einer Partnerfirma neu umgesetzt werden. Die ursprünglichen Ziele hierbei waren hauptsächlich die Datenunabhänigkeit, da das bisherige Framework hier nur relationale

[2] Convention over Configuration - Konvention vor Konfiguration [KE10]
[3] Die vier grundlegenden Datenoperationen sind Erstellen, Lesen , Ändern und Löschen

Datenbanken und Hibernate[4] abdeckt, sowie eine Drei-Schichten-Architektur[5]. Eine weitere Anforderung war, dass bisherige Projekte von der Neuentwicklung profitieren können. Da die Partnerfirma strategisch auf Rich Client Platform (RCP) als GUI Technologie und die innoSysTec GmbH hauptsächlich auf Swing setzt, ist keine gemeinsame GUI Technologie vorhanden. Aus diesem Grund entschied man sich für einen anderen Weg.

Es sollten die Widgets, die für das neue Framework nötig sind, ausgewählt und als Schnittstellen definiert werden. Das neue Framework wird dann gegen diese Schnittstellen implementiert. Aus dieser Idee entstand Jo Widgets. Das neuentwickelte Framework der beiden Firmen heißt JOCAP (Java Open Client Platform) und basiert auf Jo Widgets. Es ist inzwischen auch als Open Source Projekt auf Google Code verfügbar[6].

3.3 Architektur

Der Begriff GUI steht für Graphical User Interface. Demnach stellen GUIs die Schnittstellen zwischen Anwender und Programm dar. Diese Schnittstellen soll in Jo Widgets als Java Schnittstellen, also als Interfaces, abgebildet werden [Gro12].

Der Unterschied zu bekannten GUI Frameworks wie Swing, SWT oder JavaFX ist, dass sie ihre Schnittstellen als konkrete Klassen abbilden. Implementiert man gegen diese konkreten Klassen, kann man das Framework nicht mehr wechseln, ohne den GUI Code neu schreiben zu müssen. Interfaces bieten dagegen eine Lösung, um das Ziel des Single Sourcing zu erreichen.

Dieses Vorgehen ist auch vergleichbar mit der Java Persistence API, kurz JPA. Damit lassen sich Anwendungen mit Zugriff auf Datenbanken schreiben, ohne sich auf eine bestimmte Technologie festlegen zu müssen [MK06].

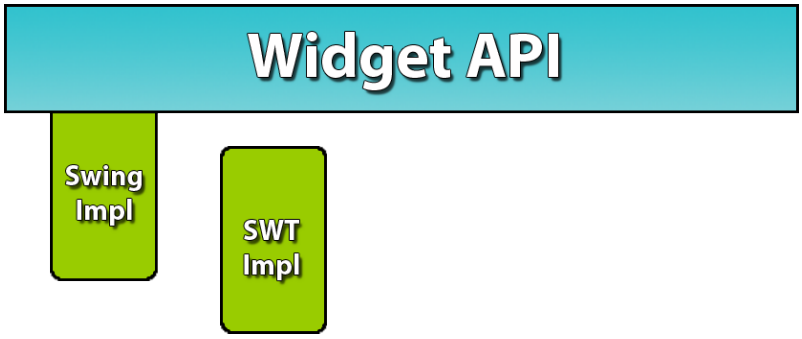

Abbildung 3.1: Jo Widgets erster Entwurf

Im ersten Entwurf der Architektur sollte der Anwendungscode vom Code des GUI Frameworks getrennt werden (s. Abbildung 3.1). Es wurden Programmierschnittstellen

[4] Hibernate ermöglicht es, gewöhnliche Java-Objekte (POJOs) mit Attributen und Methoden zu serialisieren und in relationalen Datenbanken zu speichern. Die entsprechenden Datensätze können wiederum deserialisiert werden, um das Java-Objekt zu erhalten.

[5] Präsentationsschicht, Logikschicht und Datenhaltungsschicht

[6] http://code.google.com/p/jo-client-platform/

für alle notwendigen Widgets definiert. Dafür gab es Implementierungen dieser API, welche mit Hilfe von Swing und SWT realisiert wurden.

Bei diesem ersten Entwurf wurde das Ziel des Single Sourcing bereits erreicht. Eine Anwendung wird gegen die API implementiert und es kann entweder die Swing- oder SWT-Implementierung verwendet werden. Allerdings wird hier ein hoher Grad an Code-Redundanz erschaffen, da der gesamte Umfang der API für jedes unterstützte GUI-Framework implementiert werden muss. Somit ist für die Unterstützung einer GUI Technologie ein hoher Implementierungsaufwand notwendig. Dieses Problem wird durch eine Erweiterung der Architektur um eine weitere Schicht, dem Service Provider Interface (SPI), behoben. Die Idee hierbei ist, eine schlanke SPI zu definieren, welche nur die zwingend notwendigen Funktionen für die Implementierung der API enthält.

3.3.1 Übersicht der Architektur

Die fertige Architektur ist in Abbildung 3.2 zu sehen.

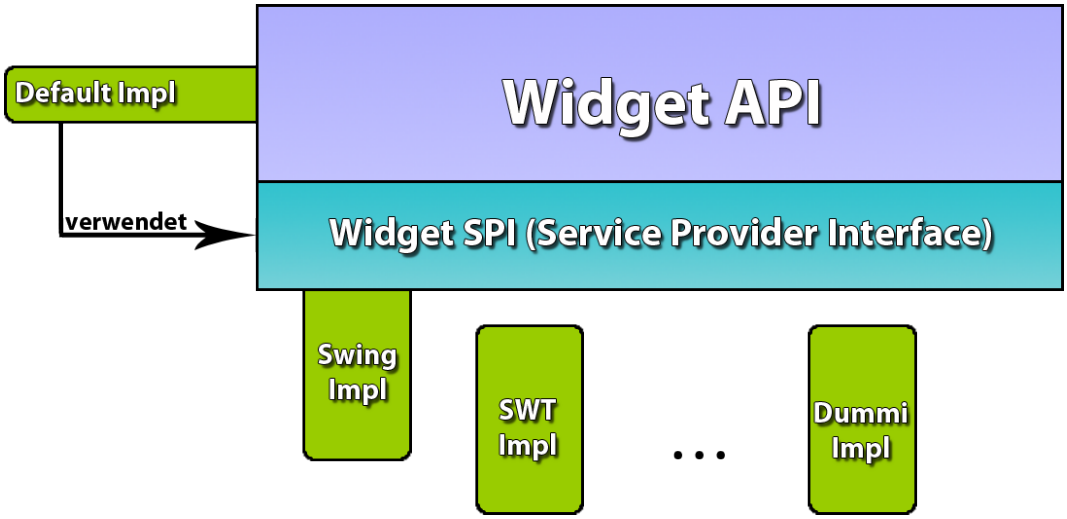

Abbildung 3.2: Die fertige Architektur von Jo Widget

API

Eine Anwendung, welche Jo Widgets verwendet, wird gegen die Jo Widgets API entwickelt. Diese besteht überwiegend aus Java Interfaces. Die meisten davon stellen Widgets und ihre dazugehörenden Setups sowie Descriptors[7] dar.

[7]Setups und Descriptors - Setups enthalten die Initialkonfiguration von Widgets. Descriptors erweitern diese um Informationen zum Widget-Typ. Mit Hilfe eines solchen Descriptors wird ein Widget erzeugt.

Default Impl

Die Default Impl stellt die Implementierung von Jo Widgets dar. Diese verwendet die SPI (Low-Level) zur Implementierung der API (High-Level).

SPI

Das SPI stellt die gemeinsame Schnittstelle der unterstützten Widgets dar. Sie wurde möglichst schlank definiert.

Service Provider Plugin

Das Konzept hinter den Plugins basiert auf dem Adapter Pattern[8]. Dadurch, dass die Plugins ohne großen Aufwand austauschbar sind, lässt sich Jo Widgets mit jeder Java UI Technologie verwenden. Ein Plugin muss dafür nur das Service Provider Interface implementieren.

3.3.2 Vorteile der Architektur

Die Entwicklung von Unternehmensanwendungen ist meist mit einem hohen Aufwand verbunden. Es kann durchaus sein, dass an einem Produkt eine Entwicklungszeit von 10 Mannjahre hat. Die Wahl der GUI Technologie wird dabei in den meisten Fällen recht früh in der Projektphase getroffen. Ein Wechsel der GUI Technologie zu einem späteren Zeitpunkt ist nur schwer möglich. Mögliche Gründe für einen Wechsel können ein neuerer Standard sein oder die Tatsache, dass man Legacy[9] Produkte erweitern möchte. Im Falle von innoSysTec war es die Zusammenarbeit mit einem Geschäftspartner. Der alte GUI Code muss in diesem Fall portiert werden oder ist wertlos.

Eine mit Jo Widgets implementierte Anwendung dagegen lässt sich ohne großen Aufwand auf ein neues GUI Framework portieren. Dies wird durch die Architektur ermöglicht, wodurch nur das SPI ausgetauscht werden muss. Ist kein Service Provider für das gewünschte Framework vorhanden, muss dieser implementiert werden. Dies lohnt sich bei kleineren Anwendungen nicht unbedingt, aber im Gegensatz zur Portierung einer Unternehmensanwendung mit mehreren Jahren Entwicklungszeit ist dieser Aufwand deutlich geringer.

Durch das schlanke SPI umfasst der Service Provider von Swing 8.429 Codezeilen und der von SWT 7.588 Codezeilen (Stand 25.05.2012). Allerdings besteht der überwiegende Teil der Implementierung aus einfach zu entwickelnden Wrapper-Methoden, welche die Aufgabe an das GUI Framework delegieren.

Für eine vollständige Implementierung wird der Umfang auf etwa 10.000 Zeilen geschätzt. Demgegenüber umfasst die Impl (also die Standard Implementierung der

[8] Adapter Pattern - Mithilfe von Adaptern können zwei inkompatible Komponenten miteinander kommunizieren.

[9] übersetzt u.a Altlast

Jo Widgets-API) bereits über 39.288 Codezeilen und es wird geschätzt, dass sie auf mindestens 50.000 Zeilen wächst. Das darauf basierende JOCAP Framework besitzt etwa 50.000 Zeilen Code [Gro12].

In der Firma innoSysTec GmBH gibt es derzeit vier größere Projekte mit jeweils bis zu 250.000 Zeilen Code. Diese bauen auf dem alten Swing-Framework auf. Dieses Framework wird derzeit noch in der Firma eingesetzt, soll jedoch von JOCAP und Jo Widgets abgelöst werden. Die vier Produkte würden dann auf JOCAP und Jo Widgets basieren [Gro12].

Der ganze Umfang zeigt sich in Abbildung 3.3.

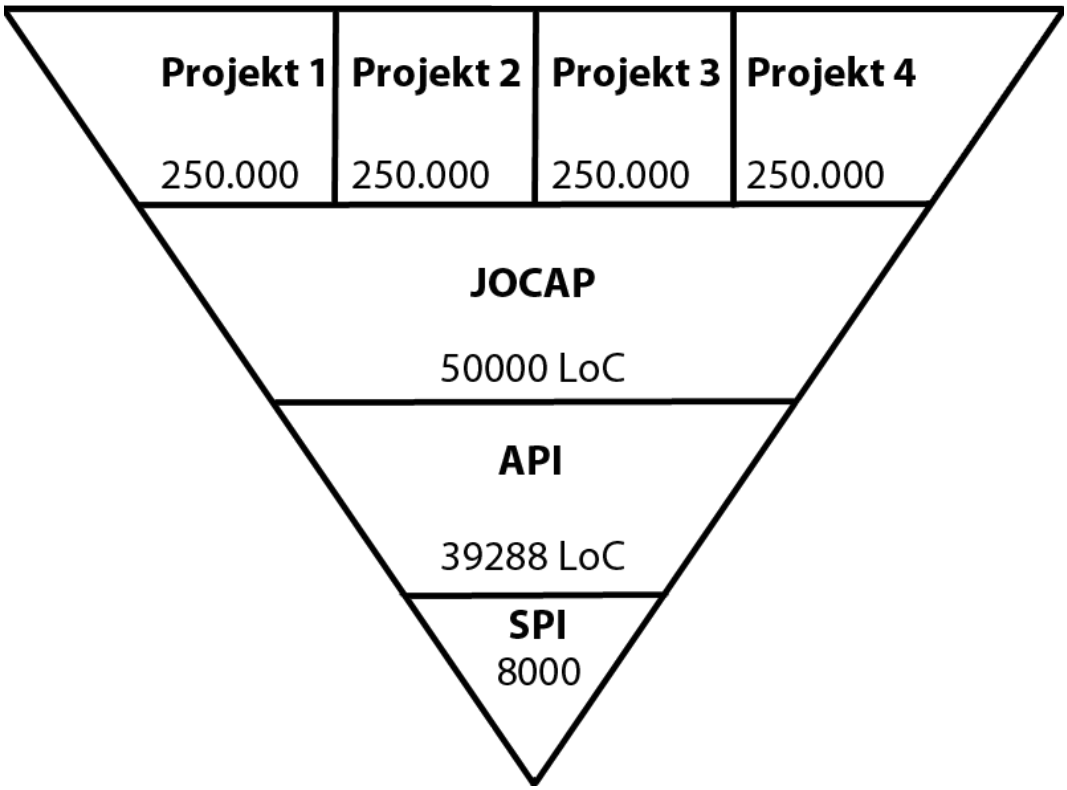

Abbildung 3.3: Der Umfang von Jo Widgets

3.4 Wichtige Klassen

In diesem Abschnitt werden die wichtigsten Klassen vorgestellt, die bei der Entwicklung mit Jo Widgets zum Einsatz kommen.

3.4.1 Toolkit

Das Toolkit ist der Einstiegspunkt für die Entwicklung mit Jo Widgets. Es bietet den Zugriff auf die Implementierung der API.

3.4.2 Blueprints

Jo Widgets trennt die Konstruktion des Widgets und das eigentliche Widget. Für jedes Widget gibt es ein passendes Blueprint Interface. Das Blueprint[10] stellt einen Bauplan für ein Widget dar und kann auch zum mehrmaligen Erstellen von Widgets wiederverwendet werden.

Auf dem Blueprint können Eigenschaften des Widgets gesetzt werden. Manche dieser Eigenschaften sind immutable[11], das heißt, sich können nach dem Erstellen des Widgets nicht mehr geändert werden. Bei diesen Eigenschaften handelt es sich zum Beispiel um die Dekoration eines Fensters oder den Rand eines Textfelds.

Blueprints sind nach dem Builder-Pattern[12] aufgebaut, sodass jede Setter-Methode den Blueprint als Rückgabewert hat. Beim Builder-Pattern wird eine Hilfsklasse verwendet, die alle Konstruktorparameter aufnimmt und die Klasse instantiiert. Dadurch wird beim Erzeugen eines Blueprints kein Konstruktor benötigt, dem mehrere Parameter zum Setzen der Properties übergeben werden. Die Konfiguration der Widgets ist somit sehr kompakt und gut lesbar.

Für die Erzeugung der Blueprints ist die Blueprint Factory verantwortlich, die über das Toolkit verfügbar ist.

3.4.3 Widgets

Die Widgets werden nach dem Bauplan des Blueprints erstellt. Dies geschieht, sobald ein Blueprint einem Container mit Hilfe der `add()`-Methode hinzugefügt wird. Im Unterschied zur add Methode eines Panels in Swing oder eines Composite in SWT wird in Jo Widgets das Widget erst aus dem Blueprint erzeugt und dann als Rückgabewert der Methode zurückgeliefert. Durch dieses Konzept ist es möglich, die Blueprints mehrfach zu verwenden, da das Blueprint immer wieder hinzugefügt werden kann, der Rückgabewert aber jeweils ein neues Widgetobjekt ist. Auf diesen Widgets lassen sich alle weiter alle Eigenschaften, welche nicht immutable sind, ändern [Gro12].

Der erste Container, der Root Frame, wird mit Hilfe der Toolkit-Klasse und eines Frame Blueprint erzeugt.

3.5 Jo Widgets Anwendung

In diesem Abschnitt wird anhand eines Hello World Beispiels erklärt, wie eine Anwendung mit Jo Widgets entwickelt wird.

[10] übersetzt u.a. Blaupause

[11] siehe Immutable Pattern unter http://www.javalobby.org/articles/immutable/index.jsp

[12] Für weitere Information, siehe http://javapapers.com/design-patterns/builder-pattern/

3.5.1 Hello World Code

Im Listing 3.1 ist ein Hello World Beispiel zu sehen. Die erste `start()`-Methode ruft mit Hilfe des Toolkits auf dem Applicationrunner der jeweiligen SPI die `run()`-Methode auf. Diese Methode ruft dann die überschriebene `start()`-Methode auf.

```
1  public class HelloWorldApplication implements
       IApplication {
2
3   public void start() {
4    Toolkit.getApplicationRunner().run(this);
5   }
6
7   @Override
8   public void start(final IApplicationLifecycle
       lifecycle) {
9
10   IBluePrintFactory bPF =Toolkit.getBluePrintFactory();
11   ILabelBluePrint labelBp = bPF.label().alignCenter().
       setFontSize(30);
12   ICompositeBluePrint compositeBp = bPF.composite();
13
14   IFrameBluePrint frameBp = bPF.frame().setTitle("Hello
       World");
15   frameBp.setSize(new Dimension(300, 100));
16
17   IFrame rootFrame = Toolkit.createRootFrame(frameBp);
18   ILayoutFactoryProvider lfp = Toolkit.
       getLayoutFactoryProvider();
19   rootFrame.setLayout(lfp.flowLayout());
20   IContainer composite = rootFrame.add(compositeBp);
21
22   ILabel hello = composite.add(labelBp);
23   hello.setText("Hello");
24   ILabel world = composite.add(labelBp);
25   world.setText("World");
26
27   rootFrame.setVisible(true);
28   }
29
30   public static void main(final String[] args) throws
       Exception {
31    new HelloWorldApplication().start();
32   }
33 }
```

Listing 3.1: Jo Widgets Hello World

Beschreibung des Hello World Codes

Über die `Toolkit`-Klasse bekommt man die Blueprintfactory zum Erzeugen der Blueprints,(s. Zeile 10). Mit der Factory werden die Blueprints für Label, Frame und Composite erzeugt. Bei der Erzeugung des Label Blueprints ist das Builder-Pattern zu erkennen(s. Zeile 11). Man kann die gewünschte Einstellungen setzen, aneinanderreihen und bekommt vom letzten Aufruf das Blueprint zurück.

Der Frame wird mit Hilfe der `Toolkit`-Klasse und dem passenden Blueprint über den Aufruf `Toolkit.createRootFrame(frameBp)` erstellt. Über die `Toolkit`-Klasse wird die Layoutfactory geholt, mit deren Hilfe das Flow Layout auf dem Frame gesetzt wird(s. Zeile 18 und 19). Das Composite Widget sowie die beiden Label Widgets werden über den Aufruf der `add()`-Methode des Frames erstellt und hinzugefügt. Aus einem Label Blueprint werden zwei Label Widget erstellt. Auf den beiden wird nach dem erzeugen noch die Beschriftung gesetzt(s. Zeile 23 und 25).

Als letzter Aufruf erfolgt das Anzeigen des Frames.

Zum Starten der Anwendung wird die Main Methode benötigt. Diese erzeugt die HelloWorldApplication und ruft dort die `start()`-Methode auf. In dieser Methode wird vom Toolkit der für die GUI Technologie zuständige ApplicationRunner aufgerufen.

Die Anwendung mit Swing, SWT

Die Hello World Anwendung aus Listing 3.1 ist unter Swing in Abbildung 3.4 und SWT in Abbildung 3.5 zu sehen.

Abbildung 3.4: Hello World in Swing

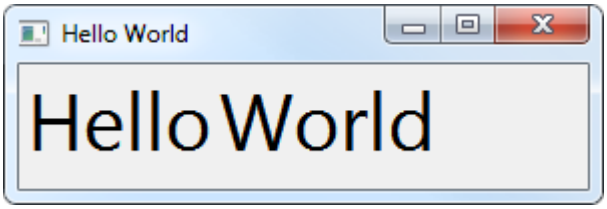

Abbildung 3.5: Hello World in SWT

Laden der GUI Technologie

Die GUI Technologie wird mit Hilfe einer Abhängigkeit, so gennante Dependency, im Maven Projekt gewählt. Maven ist ein Tool, das immer wieder anfallende Aufgaben bei der Softwareentwicklung automatisiert. Die Dependency, welcher in der Projektdatei definiert sind, werden automatisch beim Builden einer Anwendung eingebaut. Jo Widgets sucht beim Starten nach einer GUI Technologie und sollte die in der Dependency definierte finden. In Listing 3.2 ist die Dependency für die JavaFX Implementierung dargestellt.

```
1     <dependency>
2       <groupId>org.jowidgets</groupId>
3       <artifactId>org.jowidgets.spi.impl.javafx</
           artifactId>
4       <version>0.12.0-SNAPSHOT</version>
5     </dependency>
```

Listing 3.2: Dependency

3.5.2 Validierungsbeispiel

Die Jo Widget API bietet Lösungen für viele immer wieder auftauchende Probleme, weshalb diese nicht neu programmiert werden müssen. Eine Möglichkeit ist die Valdierung von Eingaben. Validierer müssen das Interface IValidator<VALUE_TYPE> implementieren. In Listing 3.3 ist ein Beispiel Valdierer zu sehen.

```
1 final IValidator<String> maxLengthValidator = new
     IValidator<String>() {
2  @Override
3  public IValidationResult validate(final String value)
       {
4   if (value != null && value.length()>50) {
5     return ValidationResult.error("Maximal 50 Zeichen
         erlaubt");
6   }
7   return ValidationResult.ok();
8  };
9 };
```

Listing 3.3: Valdierer Beispiel

Der Valdierer ist für den Datentyp String und prüft, ob die Länge des Strings nicht größer als 50 Zeichen ist (s. Zeile 4). Zurückgeben wird das Ergebnis der Validierung. Dieser Validator kann Komponenten vom Typ IInputComponent in Jo Widgets hinzugefügt werden. Um das Ergebnis des Validators anzuzeigen, kann zum Beispiel ein ValidationLabel verwendet werden. Die Verwendung eines Validators mit einem InputComponent ist in einem auf das wesentliche gekürzten Listing 3.4 zu sehen.

Mit Hilfe der BlueprintFactory werden ein Eingabefeld und ein Label für die Validierung der Eingabe erzeugt. Dem Eingabefeld wird dann der Validator aus Listing 3.3 hinzugefügt. Zuletzt werden ein Label als Hinweis für die Eingabe, das ImputComponent und das Validierungslabel, hinzugefügt.

```
1    IInputComponentValidationLabelBluePrint
        validationLabelBp;
2    validationLabelBp = bPF.
        inputComponentValidationLabel();
3
4    final IInputFieldBluePrint<String> stringFieldBp;
5    stringFieldBp =  bPF.inputFieldString();
6    stringFieldBp.setValidator(maxLengthValidator);
7
8    IInputComponent<String> eingabeFeld = rootFrame.add
        (stringFieldBp);
9    rootFrame.add(validationLabelBp.setInputComponent(
        eingabeFeld), "wrap");
```

Listing 3.4: Hinzufügen des Validators zu einem InputComponent

Das Ergebnis ist in den Abbildungen 3.6, 3.7 und 3.8 zu sehen. Das Beispiel wurde mit der Swing Implementierung gestartet und zeigt die Anwendung ohne Eingabe, mit korrekter Eingabe und einer zu langen Eingabe.

Abbildung 3.6: Ergebnis ohne Eingabe

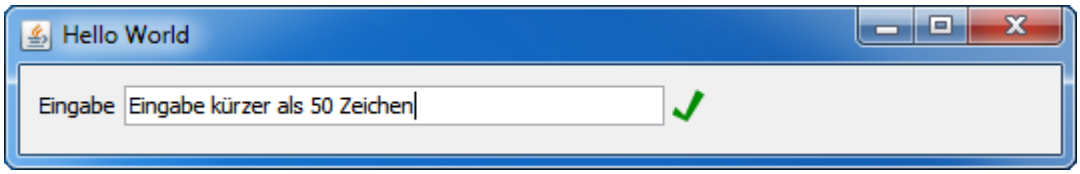

Abbildung 3.7: Ergebnis mit richtiger Eingabe

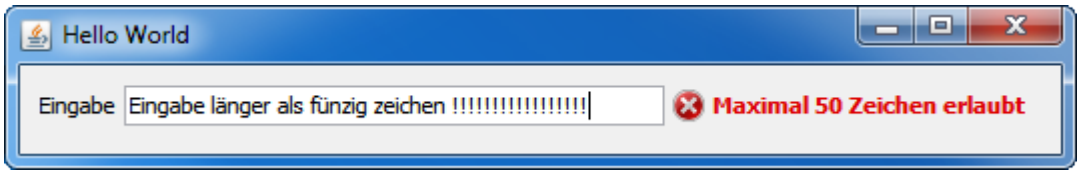

Abbildung 3.8: Ergebnis mit zu langer Eingabe

Kapitel 4

Layouting

Für die Umsetzung dieser Bachelorarbeit spielt das Layouting in JavaFX und Jo Widgets eine zentrale Rolle. Daher werden in diesem Kapitel einige Grundlagen sowie das MigLayout, welches von Jo Widgets verwendet wird, näher erläutert.

4.1 Dynamisches Layout

Elemente können in einer grafischen Oberfläche absolut und pixelgenau angeordnet werden. Da Java auf unterschiedlichen Plattformen mit unterschiedlichen Ausgabengeräten laufen soll, kann eine korrekte Anzeige bei einer absoluten Anordnung nicht garantiert werden. Schriften oder Fensterrahmen können auf jedem Gerät oder Plattform unterschiedlich sein.

Mit Swing und SWT setzen die beiden bekanntesten Java GUI-Frameworks zur Lösung dieses Problems auf Layout Manager. Sie sind für die Größenberechnung und Positionierung von Elementen innerhalb eines Containers zuständig. Diese Klassen definieren die Darstellung nicht über absolute Werte, sondern berechnen diese anhand von bestimmten Regeln. Der größte Vorteil ist die Unabhängigkeit von Fenster- und Zeichensatzgrößen[1].

In diesem Zusammenhang sind die drei Größenangaben `Preferred Size`, `Minimum Size` und `Maximum Size` der grafischen Elementen von Bedeutung.

- `Minimum Size`: Die Mindestgröße, welche für die korrekte Darstellung eines GUI-Elements nötig ist. So ist die Mimimum Size z.B. bei einem Label so groß, dass der Text auf dem Label komplett angezeigt werden kann.

- `Preferred Size`: Sie gibt die bevorzugte Größe eines GUI-Elements an, damit dieses optisch ansprechend dargestellt werden kann. Bei Komponenten mit Textinhalt wird diese Größe von der Schriftart sowie auch der Schriftgröße beeinflusst.

[1] Schriftart, Schriftgöße und Schriftstil

- `Maximum Size`: Sie entspricht der maximalen Größe die ein Element haben darf. Bei einem Button entspricht diese Größe der Preferred Size, da der Button nicht übermäßig groß dargestellt werden soll.

Layout Manager verwenden diese drei Größen sowie ihre Regeln, um die grafischen Elemente darzustellen. Welche dieser drei Größen berücksichtigt werden, kann bei jedem Layout Manager unterschiedlich sein. Das Layouten geschieht rekursiv. Das bedeutet, dass die Größenanfragen von oben nach unten laufen, solange ein Container weitere Container enthält.

4.2 Layouting in JavaFX

JavaFX bietet keine Layout Manager wie zum Beispiel Swing. Das Layout ist vielmehr direkt mit einem Container verbunden. So wird mit der Auswahl des Containers auch das Layout festgelegt. Diese Container layouten die Elemente nach ihren Regeln. In diesem Abschnitt soll kurz die Struktur hinter den Layout Containern erklärt und auf den Layout Prozess eingegangen werden. In der Abbildung 4.1 werden die wichtigsten Klassen dargestellt.

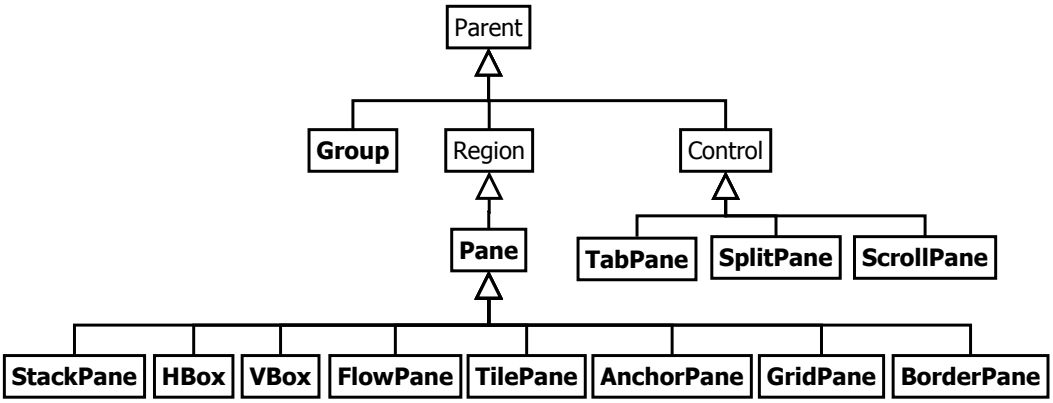

Abbildung 4.1: Klassendiagramm der JavaFX Layoutklassen

4.2.1 Parent

Die oberste Klasse, welche Elemente verwalten kann, ist die `Parent`-Klasse. Sie hält die Elemente in einer Liste und bietet Methoden zur deren Verwaltung. Die Klasse weiß, wann Elemente gelayoutet werden müssen, und führt das Layouting bei jedem Pulse Event (siehe Kapitel 2.2.4) durch.

Das Layouten der Elemente geschieht in der protected `layoutChildren`-Methode. Die `Parent`-Klasse besitzt keine besonderen Layout-Regeln, sondern setzt die Größe seiner Kinderelemente bei jedem Layout-Durchgang auf ihre Preferred Size. Dafür bietet die `Node`-Klasse[2] die `autosize`-Methode.

[2]Die oberste Klasse für alle Elemente im Scene Graph (s. Kapitel 2.2.1)

Von der `Parent`-Klasse leiten vier weitere Klassen ab, von denen auf drei im Bezug auf das Layout in JavaFX näher eingegangen wird.

4.2.2 Group

Beim Layouten der Kinder verhält sich die `Group`-Klasse ähnlich wie die Parent Klasse und setzt die Größe der Kinder ebenfalls auf ihre Preferred Size. Jedoch bietet die `Group`-Klasse die Möglichkeit, diesen Vorgang zu deaktivieren. So lassen sich Position und Größe eines jeden Kinder-Elements direkt setzen. Das `Group`-Objekt nimmt die Größe aller in ihm sichtbaren Elemente an. Diese Klasse eignet sich daher am besten für Anwendungen, welche die Nodes selbst auf eine Position und Größe setzen und diese auch nicht mehr ändern.

4.2.3 Region und Pane

Region ist die Basis-Klasse für die acht Container[3] mit besonderen Layoutregeln. Diese acht Container sind vergleichbar mit den Layout Managern aus Swing. Beispielsweise ist das GridPane dem GridLayout oder das BorderPane dem BorderLayout sehr ähnlich.[Wea12a, S. 168]

Wie zu Beginn des Kapitels erwähnt, sind Layout und Container in JavaFX fest verbunden. Möchte man ein anderes Layout verwenden, muss ein anderer Container verwendet werden. Die `Region`-Klasse layoutet nach den Regeln der `Parent`-Klasse. Die acht abgeleiteten Container layouten nach ihren jeweiligen Layout-Regeln. Außerdem benutzen sie teilweise nicht mehr die Methoden der `Parent`-Klasse zur Verwaltung der Kinder. Elemente werden z. B. dem `BorderPane`, ähnlich wie im `BorderLayout` in Swing, über `setTop` oder `setLeft` usw., hinzugefügt.

4.2.4 Control

Von der Control Klasse leiten alle JavaFX Widgets ab. Einige dieser Controls bieten auch die Möglichkeit, ihnen Elemente hinzuzufügen und diese zu layouten. Dies betrifft zum Beispiel das ScrollPane, welches einen scrollbaren Container darstellt, oder das TabPane, welches ein Container mit unterschiedlichen Tabs ist. Diese benutzen eigene Methoden zum Hinzufügen von Elementen.

[3] StackPane, HBox, VBox, TilePane, FlowPane, BorderPane, GridPane, AnchorPane

4.2.5 Layouting

Beim Layouting in JavaFX sind drei Dinge zu beachten.

Resizability

Resizability sagt aus, ob ein Node resizable[4], also in der Größe veränderbar ist. Dies bedeutet, dass ein Element seine PreferredSize, MinSize und MaxSize angibt und seinem Container erlaubt, es zu vergrößern oder zu verkleinern. Dies hängt auch mit den Regeln des jeweiligen Containers zusammen.

Content Bias

Content Bias steht für die Ausrichtung der Elemente. Bei den meisten Nodes ist die Breite unabhängig von der Höhe. Beim Layouting werden die Nodes über die `getContentBias`-Methode nach ihrer Ausrichtung gefragt. Diese wird beim Berechnen der PreferredSize, MinSize und MaxSize berücksichtigt.

Min/Pref/Max Sizes

Die Controls und Panes berechnen ihre Größe anhand ihrer Kinder bzw. ihrer Einstellungen selbst. Die Größe eines Nodes kann nicht direkt gesetzt werden. Um Einfluss auf die Größe eines Elementes zu nehmen, kann man die PreferredSize, MinSize und MaxSize überschreiben. Damit wird bei Layoutanfragen der Wert nicht neu berechnet, es wird vielmehr der überschriebene Wert übergeben [Wea12b].

Es gibt außerdem die Möglichkeit die MinSize und MaxSize fest auf die PreferredSize zu setzen. Dafür gibt es die static Variable `USE_PREF_SIZE`, welche dann als Wert beim Aufruf der `setMinSize()` oder `setMaxSize()` Methode als Parameter übergeben wird.

Die LayoutPanes berücksichtigen die Resizeability, Content Bias und Größen sowie ihre eigenen Regeln bei den Layoutdurchgängen [Wea12b].

4.3 Layouting in Jo Widgets

Für das Layouting gibt es in Jo Widgets das ILayouter Interface. Dazu gehören die Methoden `layout()`, `invalidate()`, `getMinSize()`, `getMaxSize()` und `getPreferredSize()`. Über dieses Interface lassen sich leicht eigene Layouter implementieren.

Jo Widgets bietet derzeit folgende implementierte Layout Manager.

[4] übersetzt u.a. Größenveränderlich

- **BorderLayout:** Diese ähnelt dem Layout Manager aus Swing, unterschiedlich sind die Bezeichner für die Constrainst, `TOP`, `BOTTOM`, `LEFT`, `RIGHT` und `CENTER`.
- **FillLayout:** Im FillLayout hat genau eine Komponente Platz. Sie wird bis zum Rand gestreckt. Allerdings kann man den Abstand zwischen Rand und Komponente definieren.
- **FlowLayout:** Die Komponenten werden im FlowLayout der Reihe nach angeordnet. Sollte nicht für alle Platz sein, werden weitere Reihen hinzugefügt. Die Größe der Komponenten wird auf ihre Preferred Size gesetzt.
- **ListLayout:** Dies ist Ähnlich dem FlowLayout, allerdings werden beim List-Layout die Komponenten vertikal angeordnet.
- **NullLayout:** Das NullLayout erlaubt die absolute Positionierung und das Setzen der Größe der Komponenten von Hand.
- **PreferredSizeLayout:** Das PreferredSizeLayout ist ähnlich dem NullLayout. Es benutzt für die Größe der Komponenten allerdings ihre PreferredSize.
- **MigLayout:** Hierbei handelt es sich um eine Portierung des MigLayouts (s. Kapitel 4.4) für Jo Widgets.

Zu Beginn von Jo Widgets wurde in das Layouting wenig Aufwand investiert. Es wurde das MigLayout verwendete, da dies für die beiden zu Beginn unterstützten Technologien, Swing und SWT, vorhanden war. Das Layouting fand somit auf der Seite des Servive Providers statt. Bei der Implementierung eines neuen Service Providers musste eine Portierung für die MigLayout vorhanden sein oder selbst implementiert werden. Da dies zu einem höheren Entwicklungsaufwand führte und es keine Möglichkeit gab eigene Layout Manager außer dem MigLayout zu verwenden, wurde das Layouting auf die API-Ebene verschoben [Gro12].

Dort wurde das Interface ILayouter als Schnittstelle entwickelt und eigene Layout Manager gegen dieses Interface implementiert. Die Service Provider brauchen nun keine Kenntnisse mehr über den Layout Manager, da das Layouting vollständig auf der API-Ebene stattfindet.

Da das interne MigLayout in der Entwicklung noch nicht abgeschlossen ist, besteht weiterhin die Möglichkeit, das MigLayout für die jeweilige GUI Technologie zu verwenden. Dazu wurde ein Mechanismus implementiert, womit der Entwickler selber entscheiden kann, welche Version des MigLayouts er verwendet.

4.4 MigLayout

Das MigLayout ist ein weitverbreiter Layout Manager für Java. Entwickelt wird er von der Firma MiG InfoCom AB. Er steht unter der BSD-Lizenz. Im offiziellen Google Projekt[5], in der aktuellen Version 4.2-SNAPSHOT gibt es Implementierungen für

[5] http://code.google.com/p/miglayout/

Swing, SWT und JavaFX. MigLayout ist trotz seiner Mächtigkeit und Flexibilität einfach zu nutzen. So lassen sich die Standard-Layouts aus Swing nachbauen oder komplexe, verschachtelte Layouts ohne großen Code erschaffen. Eine seiner Stärken ist das Layouten von Formularen.

4.4.1 Arbeitsweise des MigLayouts

Das Grundgerüst des MigLayouts ist ein zweidimensionales Gitter. In diesem Gitter werden alle Elemente angeordnet. Sie können im Gitter in beide Richtungen wachsen, sodass sie mehrere Zeilen oder Spalten belegen. Für die Definition des Gitters sowie für das Verhalten der Elemente werden s.g. Constraints[6] definiert. Es gibt dabei drei unterschiedliche Constraints-Klassen [AB09].

- **Layout Constraints:** Darüber lässt sich das Verhalten des Layout Managers festlegen und das Gitter definieren.
- **Column und Row Constraints:** Sie stehen für die Eigenschaften von Spalten und Zeilen im Layout. So lässt sich die Anzahl festlegen oder auch das Verhalten einzelner Zeilen oder Spalten.
- **Component Constraints:** Hier werden die Eigenschaften von Elementen, die dem Gitter hinzugefügt werden, festgelegt. So kann hier zum Beispiel definiert werden, ob ein Element bei der Veränderung der Fenstergröße wachsen soll.

[6] übersetzt u.a Bedingung. In diesem Fall sind es Anweisungen an den Layout Manager

Kapitel 5

Aufgabenstellung

Diese Bachelorabeit untersucht, ob sich JavaFX für die Implementierung des Service Provider Interfaces des Jo Widgets Frameworks eignet. Hierfür wird ein Prototyp der SPI entwickelt.

5.1 Anforderungen an den Prototyp

Ein Prototyp soll eine Anfangsversion einer Entwicklung zeigen, verschiedene Entwürfe ausprobieren und helfen, Erkenntnisse über mögliche Probleme sowie die dazu passende Lösung zu gewinnen [Som07].

In dieser Bachelorarbeit soll der Prototyp zeigen, ob sich JavaFX für die Implementierung des Service Provider Interfaces eignet. Dies soll mit der Implementierung des Application Runners beginnen. Dieser startet die Hauptkomponenten (zum Beispiel den Event-Dispatcher-Thread in Swing) der ausgewählten GUI Technologie. Im Prototypen sollen zuerst die Basiskomponenten Frame und Composite implementiert werden. Mit Hilfe des Frames kann ein Fenster angezeigt werden, mit dem Composite (ein Container) können verschachtelte Layouts entwickelt werden. Außerdem sollen für eine kleine Anwendung mindestens die Widgets für eine Button und ein Textfeld implementiert werden.

Eine Implementierung von Widgets wie dem ScrollComposite oder dem Tree, wäre wünschenswert, um zu erkennen, ob auch komplexere Widgets umsetzbar sind.

5.2 Motivation

Mit JavaFX würde Jo Widgets eine moderne GUI Technolgie unterstützen, welche möglicherweise Swing ablösen könnte [Sch12]. Obwohl die API von JavaFX noch in der Entwicklung ist und auch Jo Widgets in der Firma innoSysTec GmbH noch nicht in allen Projekten eingesetzt wird, wäre eine Implementierung des SPI mit JavaFX eine Investition in die Zukunft. Der gesamte bisher mit Jo Widgets geschriebene Code kann von den Vorteilen und Neuerungen profitieren, zum Beispiel von der

schnelleren Rendering Engine oder der Möglichkeit, das Aussehen von Anwendungen mit Hilfe von CSS anzupassen.

InnoSysTec könnte bereits Produkte mit JavaFX anbieten, ohne dass sich Entwickler erst in JavaFX einarbeiten müssten.

Kapitel 6

Realisierung des Service Providers

Dieses Kapitel befasst sich mit der Implementierung des Service Provider Prototypen. Es werden Kernprobleme beschrieben und deren Lösungen erläutert.

6.1 Application Runner

Beim Start einer mit Jo Widgets entwickelten Anwendung muss die verwendete GUI Technologie ebenfalls gestartet werden. Dafür ist unter Jo Widgets das Interface `IApplicationRunner` vorgesehen. Dieses enthält die einzige Methode `run(IApplication application)`, in der die GUI Technologie und die Jo Widgets Application gestartet werden.

Die Implementierung des Application Runner für JavaFX ist im Listing 6.1 zu sehen. Die `JavafxApplicationRunner`-Klasse erweitert die `Application`-Klasse aus JavaFX und implementiert dazu das Interface `IApplicationRunner` von Jo Widgets.

Der `run(Application application)`-Methode wird eine Application Klasse übergeben, die den Anwendungscode von Jo Widgets enthält. Der Aufruf der `run`-Methode ist zusätzlich im Hello World Listing 3.1 zu sehen.

Die `run`-Methode erzeugt mit Hilfe der `launch()`-Methode den JavaFX Application Thread, welcher die Lebenszyklenmethoden aufruft. Die in der `start()`-Methode von JavaFX übergebene Stage wird ignoriert, da Jo Widgets im Anwendungscode einen neue Stage erzeugt. Die Methode startet außerdem den Lebenszyklus von Jo Widgets. Dazu wird das Interface `IApplicationLifecycle` mit der einzigen Methode `finish()` implementiert (s. Zeile 15-25). Diese wird beim Beenden einer Jo Widgets Anwendung aufgerufen und muss den JavaFX Application Thread beenden. Dies geschieht über den Aufruf der `stop()`-Methode und indem die Jo Widgets Applikation auf `null` gesetzt wird.

Mit dem Aufruf `application.start(lifecycle)` wird die Anwendung gestartet und der Java Code ausgeführt.

```
1  public class JavafxApplicationRunner extends
       Application implements IApplicationRunner {
2
3    private static IApplication application;
4
5    //Implementierung der run()-Methode aus dem
       IApplicationRunner Interface
6    @Override
7    public synchronized void run( IApplication
       application) {
8      JavafxApplicationRunner.application = application;
9      launch();
10   }
11
12   //Die Überschriebene start()-Methode aus der
       Application Klasse von JavaFX
13   @Override
14   public void start( Stage stage) throws Exception {
15     IApplicationLifecycle lifecycle = new
         IApplicationLifecycle() {
16       @Override
17       public void finish() {
18         try {
19           stop();
20         }
21         catch ( Exception e) {
22           e.printStackTrace();
23         }
24       }
25     };
26     application.start(lifecycle);
27   }
28   //Die Überschriebene stop()-Methode aus der
       Application Klasse von JavaFX
29   @Override
30   public void stop() throws Exception {
31     super.stop();
32     application = null;
33   }
34 }
```

Listing 6.1: JavaFX ApplicationRunner

6.2 Implementierung eines Widgets

Die Hauptaufgabe bei der Entwicklung eines Service Provider Plugins besteht in der Implementierung der Interfaces aus dem Modul `org.jowidgets.spi`. Für ein Widget muss das passende Interface aus dem SPI implementiert werden. Für einen Button ist dies z.B. das Interface `IButtonSpi`. Die Interfaces in der SPI sind in einer Vererbungshierarchie aufgebaut. Die Hierarchie ist exemplarisch anhand des IButtonSpi Interfaces in Abbildung 6.1 dargestellt. In dieser Abbildung ist zusätzlich das Textfield Widget mit dem Interface `ITextControlSpi` dargestellt, um die Vererbungshierarchie zu verdeutlichen.

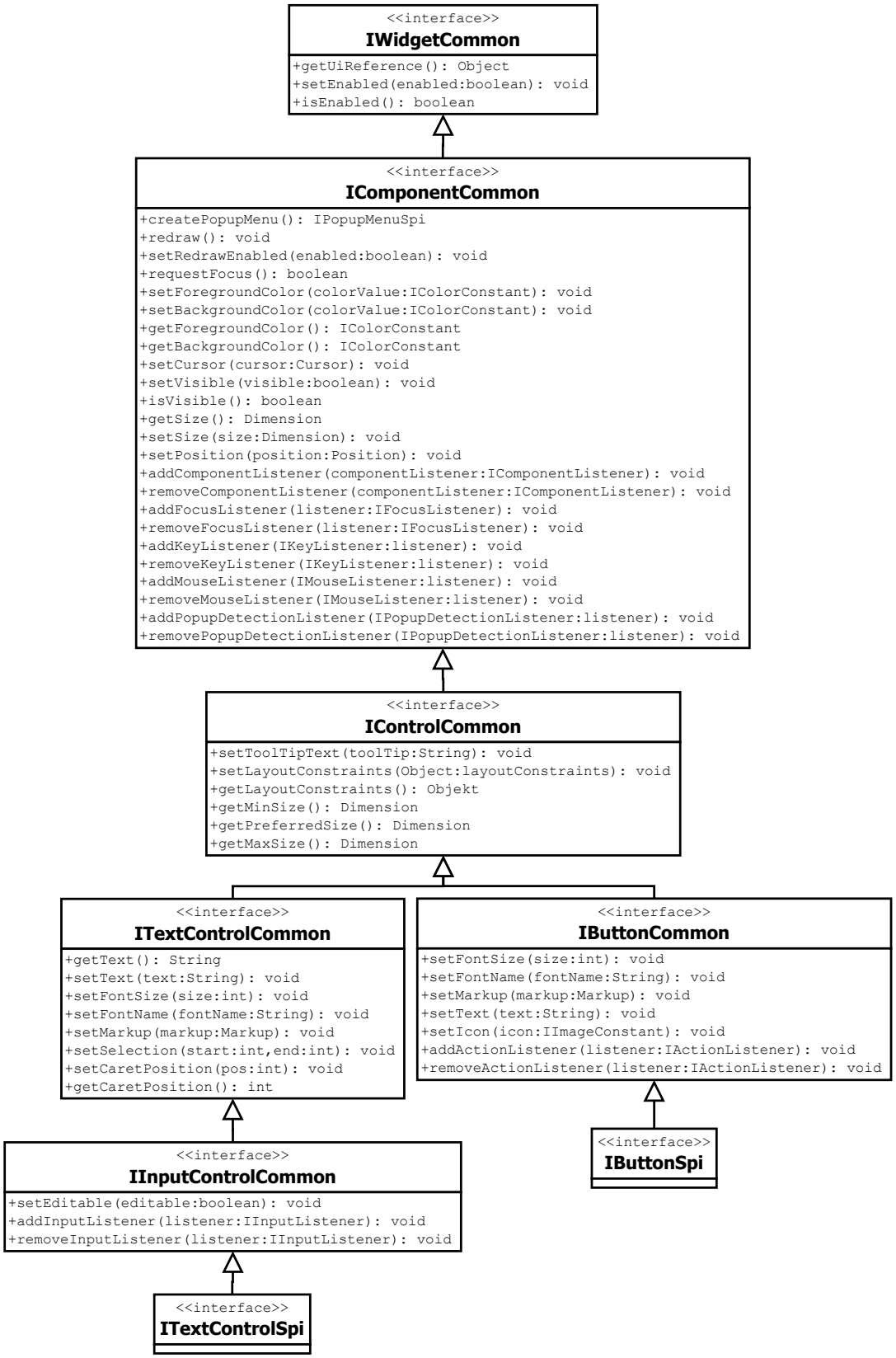

Abbildung 6.1: Jo Widgets Hierarchie anhand des Button und Textfield Widgets

Bei der Implementierung der `IButtonSpi` sind die meisten Methoden nur Wrapper-Methoden, die den Aufruf an die passende Methode von JavaFX delegieren. Da viele Widgets aus Jo Widgets dieselben Methoden implementieren müssen, wurde auch auf Seiten des Prototyps eine Hierarchie entwickelt, um redundanten Code zu vermeiden. Viele JavaFX Widgets leiten von der `Control`-Klasse im Package `javafx.scene.control` ab.

Für die Entwicklung der Hierarchie wurde eine abstrakte Klasse `JavafxControl` erstellt, welche das Interface `IControlSpi` implementiert und als JavaFX Komponente ein `Control` benutzt. Auf diese Weise kann bei der Implementierung eines Labels oder Buttons von dieser Klasse abgeleitet werden werden, wodurch Methoden nicht mehrfach implementiert werden müssen. Ein Auschnitt aus der Hierarchie, die in der Implementierung umgesetzt wurde ist in Abbildung 6.2 zu sehen.

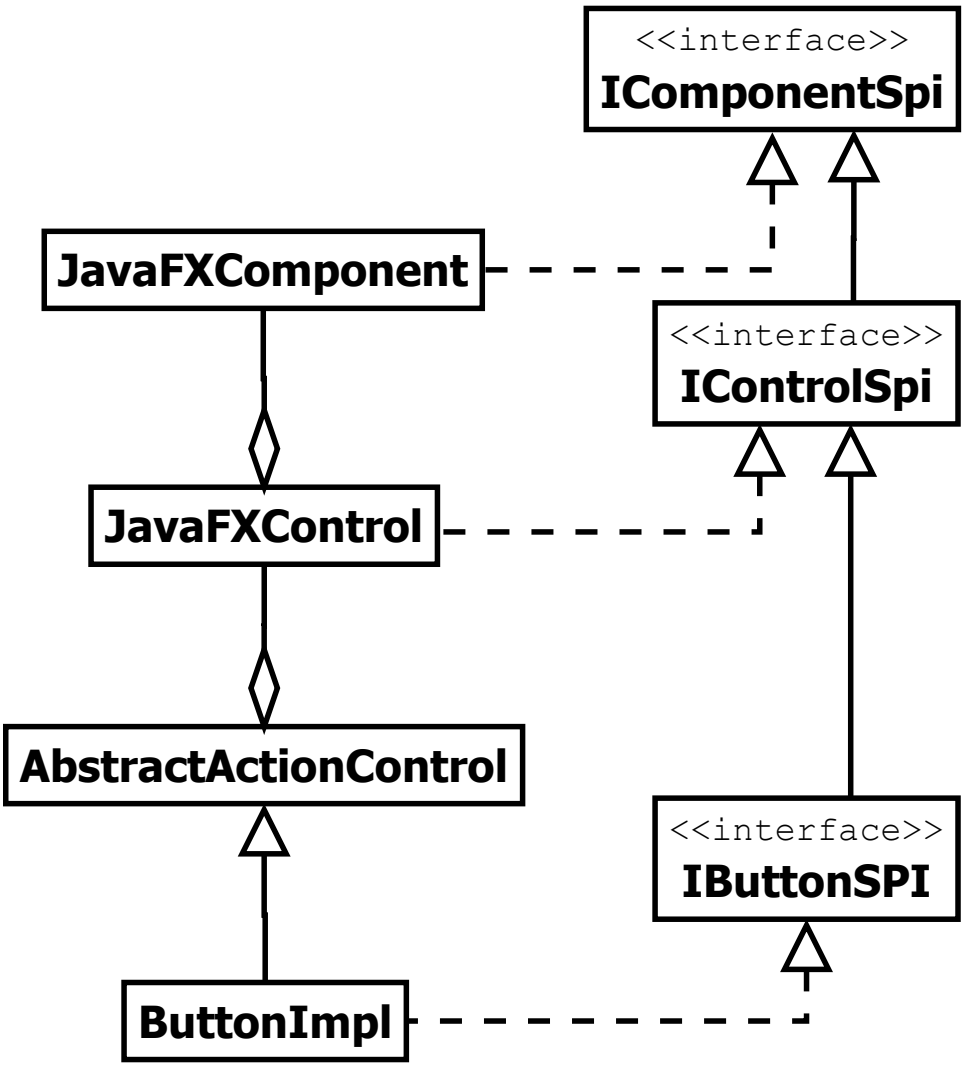

Abbildung 6.2: Ausschnitt der Hierarchie auf Seite des Service Providers

6.3 Setzen von Styleeinstellungen

JavaFX bietet Möglichkeiten, das Aussehen von Widgets zu ändern (s. Kapitel 2.2.5). Es gibt allerdings nicht für alle Einstellungen eine passende API Methode. Zum Beispiel ist es nicht möglich, auf Controls die Schrifteinstellungen zu ändern, den Rahmen oder die Hinter- bzw. Vordergrundfarbe zu setzen.

Laut Aussagen von Oracle soll es in der nächsten Version[1] ein Modell mit allen Einstellungen geben, weshalb eine vorrübergehende Lösung gefunden werden muss. Es gibt die Möglichkeit, einen String mit der CSS-Syntax auf die Widgets zu setzen. Allerdings ist es immer nur möglich, einen String zu setzen, da es keine `get()`-Methode gibt und ein erneuter Aufruf der `set()`-Methode den String mit den alten Einstellungen überschreiben würde.

Für dieses Problem wurde eine Klasse entwickelt, die in Listing 6.2 zu sehen ist. Die Implementierungen der Widgets initialisieren bei ihrer eigenen Erstellung ein Objekt der `StyleDelegate`-Klasse. Wird in der Implementierung des Widgets eine Methode zum Ändern des Aussehens aufgerufen, wird dieser Aufruf an das `StyleDelegate` delegiert.

Die `StyleDelegate`-Klasse setzt einen String, der die CSS-Syntax mit dem übergeben Wert beinhaltet. Danach wird die `setStyle()`-Methode aufgerufen. Sie fügt die einzelnen Strings zusammen und setzt diesen auf dem Widget. Initial sind die Strings leer, weshalb keine Einstellungen gesetzt werden.

Diese Klasse löst zudem das Problem, dass JavaFX keine Möglichkeit bietet die aktuell gesetzte Hinter- und Vordergrundsfarbe zu erfragen. Diese Aufrufe werden delegiert und es wird im `StyleDelegate` der aktuell gesetzte String zurückgegeben.

Die Farbwerte werden in Jo Widgets in drei dezimalen Werten, für Rot, Grün und Blau (RGB) gespeichert. JavaFX benötigt diese Farbwerte in hexadezimalen Werten. Das Konvertieren übernimmt die Klasse `ColorCSSConverter`, welche in Abschnitt 6.8 vorgestellt wird.

[1] Kommentar von Brian Beck am 13.03.2012 auf http://javafx-jira.kenai.com/browse/RT-17293

```
1  public  class StyleDelegate {
2    private String fontColorCSS = "";
3    private String backgroundColorCSS = "";
4    private String fontSizeCSS = "";
5    private String fontNameCSS = "";
6    private String borderCSS = "";
7    private String markupCSS = "";
8    private  Node node;
9
10   public StyleDelegate(Node node) {
11     this.node = node;
12   }
13   public void setForegroundColor(IColorConstant colorValue)
       {
14     fontColorCSS = "-fx-text-fill: #"+ColorCSSConverter.
         colorToCSS(colorValue)+";\n";
15     setStyle();
16   }
17   public void setBackgroundColor(IColorConstant colorValue)
       {
18     backgroundColorCSS = "-fx-background-color: #"+
         ColorCSSConverter.colorToCSS(colorValue)+";\n";
19     setStyle();
20   }
21   public void setMarkup(Markup newMarkup) {
22     if (Markup.DEFAULT.equals(newMarkup)) {
23       markupCSS = "-fx-font-style: normal;\n -fx-font-weight
           : normal;\n";
24     }
25     else if (Markup.STRONG.equals(newMarkup)) {
26       markupCSS = "-fx-font-style: normal;\n -fx-font-weight
           : bold;\n";
27     }
28     else if (Markup.EMPHASIZED.equals(newMarkup)) {
29       markupCSS = "-fx-font-style: italic;\n -fx-font-weight
           : normal;\n";
30     }
31     setStyle();
32   }
33   public IColorConstant getForegroundColor() {
34     return ColorCSSConverter.cssToColor(fontColorCSS);
35   }
36   public IColorConstant getBackgroundColor() {
37     return ColorCSSConverter.cssToColor(backgroundColorCSS);
38   }
39   public void setFontSize(int size) {
40     fontSizeCSS ="-fx-font-size:"+size*100/72+"pt;\n";
41     setStyle();
42   }
43   public void setFontName(String fontName) {
44     fontNameCSS ="-fx-font-family: "+fontName+";\n";
45     setStyle();
46   }
47   public void setNoBorder() {
48     if (backgroundColorCSS.isEmpty()) {
49       borderCSS = "-fx-border-color: #ffffff;\n"+"-fx-border
           -insets: 0;\n"+"-fx-border-width: 0;\n";
50     }
51     else {
52       borderCSS = "-fx-border-color: null;\n"+"-fx-border-
           insets: 0;\n"+"-fx-border-width: 0;\n";
53     }
54     setStyle();
55   }
56   public void setStyle() {
57     node.setStyle(fontNameCSS + fontSizeCSS + borderCSS +
         backgroundColorCSS + fontColorCSS + markupCSS);
58   }
59 }
```

Listing 6.2: JavaFX StyleDelegate

6.4 Layout

Dieses Kapitel befasst sich mit den Schwierigkeiten beim Setzen eines Layouts und den verschiedenen Lösungsansätzen.

6.4.1 LayoutPane

Jo Widgets verwendet, wie in Kapitel 4.3 beschrieben, eigene Layout Manager und greift nicht auf die Layout Mechanismen der GUI Technologie zurück. Damit Jo Widgets das Layouting übernehmen kann, muss es wissen, wann die benutzte GUI Technologie layoutet. Dazu wurde ein eigener JavaFX Layout Container geschrieben, der die Berechnungen der Größen und Positionen an die Jo Widgets Layout Manager delegiert. Um einen eigenen Layout Container in JavaFX zu schreiben, empfiehlt die Dokumentation der JavaFX API die Methoden `computePrefWidth`, `computePrefHeight` und `layoutChildren` der `Region`-Klasse zu überschreiben [Ora12b].

Leider bietet die `Region`-Klasse keine Möglichkeit um auf die Liste der Kind-Elemente zugreifen zu können. Für den eigenen Layout Container wurde deshalb die `Pane`-Klasse verwendet. Sie leitet direkt von `Region` ab und bietet diese Möglichkeit.

Die Hierarchie mit den relevanten Methoden der `Region`- und `Pane`-Klasse ist in Abbildung 6.3 dargestellt.

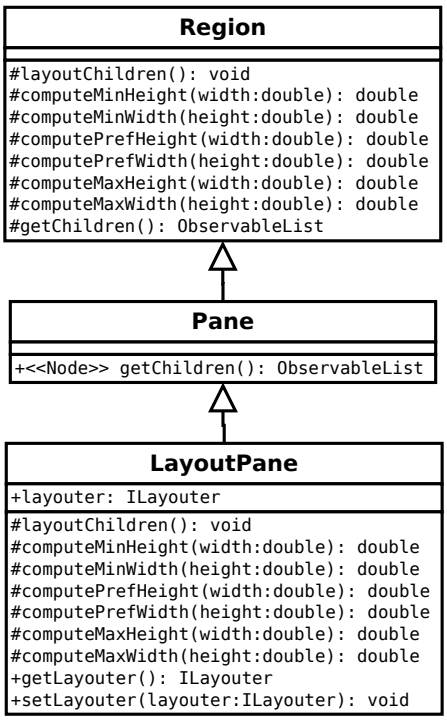

Abbildung 6.3: Die Hierarchie des LayoutPane

Der `LayoutPane`-Klasse kann beim Erzeugen ein Jo Widgets Layouter übergeben werden. Die `layoutChildren`-Methode, welche bei Layoutdurchgängen von JavaFX aufgerufen wird, wird überschrieben und ruft stattdessen den Layout Manager von Jo Widgets auf. Auch die Anfragen an die `computeMaxWidth`, `computeMaxHeight`, `computeMinWidth`, `computeMinHeight` Methoden werden an den Layouter von Jo Widgets delegiert.

6.4.2 MigPane

Da es mit MigPane[2] bereits einen Wrapper für MigLayout unter JavaFX gibt, wurde untersucht, ob dieses einsetzbar ist. Das MigPane erweitert wie das `LayoutPane` die `Pane` Klasse von JavaFX. Die Möglichkeit das MigPanes zu verwenden wurde eingebaut und wird in Abschnitt 6.4.3 genauer erklärt.

6.4.3 Ein Layout setzen

In Jo Widgets wird bei der Erstellung eines Containers ein `FlowLayout` als Standard-Layout gesetzt. Wird ein anderes Layout auf den Container gesetzt, führt dies zu Problemen (s. Kapitel 4.2.3), da ein Container in JavaFX fest mit dem Layout verbunden ist. JavaFX bietet keine `setLayout()`-Methode wie Swing, um das Layout um zusetzen. Für die Implementierung der Jo Widgets `setLayout()`-Methode musste eine Lösung gefunden werden.

Die `LayoutPane`-Klasse wurde um eine `setLayout`-Methode erweitert. Dennoch gab es ein weiteres Problem. Wenn nun statt den Jo Widgets Layout Managern die MigLayout Portierung für JavaFX verwendet wird, müssen alle bisher dem alten Container hinzugefügten Elemente ausgelesen und dem neuen Container hinzugefügt werden. Außerdem muss, falls der Container selber ein Vater-Objekt hat, der alte Container aus der Liste seines Vaters entfernt werden und der neue Container hinzugefügt werden.

Die Implementierung der `setLayout`-Methode ist in Listing 6.3 zu sehen. Dort wird anhand des `layoutDescriptor` eine Factory-Klasse für das MigPane oder für die Jo Widgets Layouter erstellt.Die Factory-Klassen implementieren beide das `IFactory<Pane>` Interface. Der `createNewPane()`-Methode wird danach die jeweilige Factory übergeben. Die Methode ist in Listing 6.4 zu sehen.

Mit Hilfe der `getUiReference()`-Methode bekommt man den aktuellen Container. Für das Vater-Objekt werden der `Parent` und die `Scene` des aktuellen Containers erfragt. Die Scene hat den Wert `null`, falls der Container nicht der Root Container ist. Die Liste der Kinder wird zuletzt zwischengespeichert. Die Stelle, an der sich der alte Container in seinem Vater befunden hat, wird über die Methode `findIndexInParent` gesucht (s. Zeile 10).

Mit Hilfe der Factory wird der neue Container erzeugt und wieder seinem Vater hinzugefügt. Die Kind-Elemente werden dem neuen Container hinzugefügt.

[2] http://java.net/projects/miglayoutfx2/pages/Home

```
1  public void setLayout(ILayoutDescriptor layoutDescriptor) {
2
3    if (layoutDescriptor instanceof ILayouter) {
4     ILayouter layouter = (ILayouter) layoutDescriptor;
5     Pane pane = getUiReference();
6    if (pane instanceof LayoutPane) {
7      ((LayoutPane) pane).setLayouter(layouter);
8    }
9    else {
10     createNewPane(new LayoutPaneFactory(layouter));
11   }
12   }
13   else if (layoutDescriptor instanceof MigLayoutDescriptor)
         {
14     MigLayoutDescriptor migLayoutManager = (
           MigLayoutDescriptor) layoutDescriptor;
15     createNewPane(new MigLayoutPaneFactory(migLayoutManager)
         );
16   }
17   else {
18     throw new IllegalArgumentException("Layout Descriptor of
         type '"
19       + layoutDescriptor.getClass().getName()
20       + "' is not supported");
21   }
22 }
```

Listing 6.3: Implementierung der setLayout() Methode

```
1  private void createNewPane(IFactory<Pane> paneFactory){
2
3    Pane pane = getUiReference();
4    Parent parent = pane.getParent();
5    Scene scene = pane.getScene();
6    ObservableList<Node> children = pane.getChildren();
7    int index = 0;
8
9    if (parent != null) {
10     index = findIndexInParent(pane);
11     ((Pane) parent).getChildren().remove(pane);
12   }
13
14   //delegate creation of the new pane
15   Pane newPane = paneFactory.create();
16
17   if (parent != null && index >= 0) {
18     ((Pane) parent).getChildren().add(index, newPane);
19   }
20   //if it was root add it again
21   else if (scene != null) {
22     scene.setRoot(newPane);
23   }
24   if (newPane instanceof MigPane) {
25     for (Node node : children) {
26       ((MigPane) newPane).add(node, (String) node.
           getUserData());
27     }
28   }
29   else {
30     newPane.getChildren().addAll(children);
31   }
32 }
```

Listing 6.4: Implementierung der setLayout() Methode

6.5 Layouting

In diesem Abschnitt werden die Probleme beim Layouting der Komponenten beschrieben.

6.5.1 Initiale Größenberechnung

Viele Probleme beim Layouting lassen sich darauf zurückführen, dass die Widgets in JavaFX vor dem ersten Zeichnen einer Stage noch keine Information über ihre Größe haben. Die Jo Widgets Layouter fragen vor dem ersten Anzeigen der GUI die Komponenten nach ihren Größen. Diese können zu diesem Zeitpunkt allerdings nur eine -1 zurückliefen, weshalb der Jo Widgets Layouter die Komponenten falsch layoutet.

Die `pack`-Methode aus Jo Widgets wurde hierfür als Workaround implementiert, da sie vor dem Anzeigen der Anwendungen aus Jo Widgets aufgerufen wird. Dabei wird Anwendung kurz für den Anwender unsichtbar gezeichnet. Da es dadurch zu keinem Flackern oder anderen Problemen kommt, wird dieser Workaround auch beibehalten, bis eventuell JavaFX eine Möglichkeit bietet die Größen vor dem Anzeigen der Anwendung zu erfragen. Die Implementierung der `pack`-Methode ist in Listing 6.5 zu sehen.

```
1     if (!stage.isShowing()) {
2       stage.setOpacity(0);
3       stage.show();
4       stage.close();
5       stage.setOpacity(1);
6     }
```

Listing 6.5: Implementierung der pack() Methode

6.5.2 Setzen von Größen und Positionen

JavaFX Nodes bieten nicht die Möglichkeit, ihnen über eine `setSize()`-Methode direkt eine Größe zu setzen. Wie in Kapitel 4.2 beschrieben, wird empfohlen, die PreferredSize, MinSize und MaxSize auf den gleichen Wert zu setzen. Dies führt aber dazu, dass beim erneuten Layouten die Größen nicht mehr neu berechnet werden. Ändert sich beispielsweise der Text eines Labels, werden die Größen nicht angepasst [Fow11].

Die Layout Manager von JavaFX verwenden für das Setzen der Größe und Position die `resize`- bzw. `relocate`-Methode. In der Dokumentation von JavaFX wird von der Verwendung dieser Methoden abgeraten, weil der Container bei Layouten die so gesetzten Werte überschreibt. In der Implementierung ist das jedoch nicht der Fall, da das LayoutPane die Anfragen direkt an einen Jo Widgets Layout Manager delegiert und dieser die Werte setzt.

6.5.3 MigPane

Die Verwendung des MigPanes ist derzeit noch fehlerhaft. Das MigPane berechnet die Mindestgröße von Labels und Buttons mit Text falsch. Das führt teilweise dazu, dass ein Label statt mit dem gesamten Text nur mit drei Punkten dargestellt wird. Die Berechnung der Fenstergröße funktioniert ebenfalls noch nicht zuverlässig.

6.6 Menüleiste

Ein Menü in JavaFX ist eine eigenständige Komponente, die gelayoutet werden muss. Anders als in Swing oder SWT bietet ein Fenster nicht die Möglichkeit, explizit eine Menüleiste hinzuzufügen. In JavaFX gibt es einen Layout Container `VBox`, der die Komponenten vertikal anordnet. Für die Lösung des Problems wird das Root-Element der `Scene` zwischengespeichert und eine `MenuBar` angelegt.

Diese beiden Elemente werden zu einer `VBox` hinzugefügt und diese wird als neues Root-Element der Scene gesetzt. Diese Vorgang ist in einem Auszug aus der Methode `createMenuBar` in Listing 6.6 zu sehen.

```
1     MenuBar bar = new MenuBar();
2     Parent oldRoot = stage.getScene().getRoot();
3     VBox newRoot = new VBox();
4     newRoot.getChildren().addAll(bar, oldRoot);
5     stage.getScene().setRoot(newRoot);
```

Listing 6.6: Menu Bar

6.7 Image Factory

Die Image Factory ist in Jo Widgets für die Verwaltung von Grafiken zuständig. Sie implementiert dazu das Interface `IImageRegistry`. Um Grafiken zu setzen ist es nötig, sie zuvor zu registrieren. Eine Implementierung dieses Interfaces ist vorhanden und kann von allen Service Providern genutzt werden. Dort können die Pfade zu der Grafik und eine Konstante gespeichert werden. Über die Konstante können die Bilder geladen und Widgets hinzugefügt werden. `JavafxImageRegistry` erweitert diese Implementierung des `IImageRegistry`-Interfaces um die `getImage`-Methode (s. Listing 6.7), die ein `ImageView`-Objekt zurückgibt. Die `ImageView`-Klasse ist in JavaFX eine Grafik und kann auf den Widgets gesetzt werden.

Neben der Möglichkeit eigene Widgets zu registrieren, bietet Jo Widgets mit dem Interface `IImageHandleFactorySpi` die Möglichkeit an, Default Icons zu setzten. JavaFX bietet dafür aktuell keine Icons. Das Interface wurde in der Klasse `JavafxImageHandleFactorySpi` implementiert, liefert momentan aber noch keine Ions zurück.

```
1  public synchronized ImageView getImage(IImageConstant
       key) {
2    if (key == null) {
3      return null;
4    }
5    ImageHandle<Image> imageHandle = getImageHandle(key);
6    if (imageHandle != null) {
7      return new ImageView(imageHandle.getImage());
8    }
9    else {
10     throw new IllegalArgumentException("No icon found
         for the image constant '" + key + "'");
11   }
12 }
```

Listing 6.7: getImage Methode aus der JavafxImageRegistry Klasse

6.8 Farbkonverter

Für die Implementierung einiger Widgets wird ein Konverter für die Farben benötigt. Der Konverter ist nötig, um die RGB[3] Farbwerte von drei dezimalen Werten aus Jo Widgets in einen hexadezimalen Wert für JavaFX und zurück zu konvertieren. Der hexadezimale Wert liegt als `String` , die drei dezimalen Werte jeweils `int` vor. In Listing 6.8 ist die Konvertierungsmethode von JavaFX nach Jo Widgets abgebildet. Der Methode `cssToColor` wird ein String übergeben, der CSS-Syntax enthält, um beispielsweise die Hintergrundfarbe über `-fx-background-color:#ffff00;` oder die Rahmenfarbe mit `-fx-border-color:#00CCFF;` zu ändern. Der hexadezimale Wert wird über die `substring`-Methode der `String`-Klasse ausgelesen. Anschließend werden die drei Buchstabenpaare in `int`-Werte über die `parseInt`-Methode umgewandelt. Daraus wird ein neues `ColorValue`-Objekt erstellt, das einen Farbwert in Jo Widgets darstellt.

```
1  public static IColorConstant cssToColor(String color) {
2    if (!EmptyCheck.isEmpty(color)) {
3      String hex =
4      color.substring(color.indexOf("#") + 1,
5                      color.indexOf(";"));
6
7      int r = Integer.parseInt(hex.substring(0, 1), 16);
8      int g = Integer.parseInt(hex.substring(2, 3), 16);
9      int b = Integer.parseInt(hex.substring(4, 5), 16);
10     return new ColorValue(r, g, b);
11   }
12   return null;
13 }
```

Listing 6.8: Farbkonvertierung von JavaFX zu Jo Widgets

Die Konvertierung von Jo Widgets nach JavaFX erfolgt über die Methode `colorToCSS` und ist in Listing 6.9 zu sehen. Der Methode wird ein `IColorConstant`-Objekt übergeben und aus diesem werden die `int`-Werte für Rot, Grün und Blau ausgelesen. Mit Hilfe der `toHexString`-Methode der `Integer`-Klasse werden die Werte konvertiert und in einem `String` zusammengefügt.

[3]RGB - Rot Grün Blau

```
1  public static String colorToCSS(IColorConstant color) {
2    if (color != null) {
3      int red = color.getDefaultValue().getRed();
4      int green = color.getDefaultValue().getGreen();
5      int blue = color.getDefaultValue().getBlue();
6
7      final String colorString =
8          Integer.toHexString(0x100 | red).substring(1)
9        + ""
10       + Integer.toHexString(0x100 | green).substring(1)
11       + ""
12       + Integer.toHexString(0x100 | blue).substring(1);
13
14     return colorString;
15   }
16   return "";
17 }
```

Listing 6.9: Farbkonvertierung von Jo Widgets zu JavaFX

6.9 Umgesetzte Widgets

Die Tabelle 6.1 bietet einen Überblick, welche Widgets bereits umgesetzt wurden. Die Widgets der SPI sind links aufgelistet und die JavaFX Klassen, mit welcher sie für diesen Prototyp implementiert wurden, rechts.

SPI Widget	JavaFX
IFrameSpi	Stage
IPopupDialogSpi	Stage
ICompositeSpi	Pane
ISplitCompositeSpi	SplitPane
IScrollCompositeSpi	ScrollPane
ITextControlSpi	TextField
ITextAreaSpi	TextArea
ITextLabelSpi	Label
IIconSpi	Label
IButtonSpi	Button
IControlSpi	Control
ICheckBoxSpi	CheckBox
IToggleButtonSpi	ToogleButton
IComboBoxSelectionSpi	bisher nicht umgesetzt
IComboBoxSpi	bisher nicht umgesetzt
IProgressBarSpi	ProgressBar
IToolBarSpi	ToolBar
ITabFolderSpi	TabFolder
ITreeSpi	Tree
ITableSpi	bisher nicht umgesetzt

Tabelle 6.1: Umgesetzte Widgets

Kapitel 7

Bewertung und Ausblick

7.1 Bewertung

Ziel dieser Bachelorarbeit war es zu untersuchen, ob mit JavaFX eine Implementierung des Service Provider Interfaces von Jo Widgets möglich ist. Dafür wurde ein Prototyp der SPI implementiert. Bei der Implementierung wurden alle Widgets der SPI bis auf `Table` und `Combobox` entwickelt (siehe Tabelle 6.1). Damit wurden die Basiscomponenten wie Frame und Composite implementiert. Hierbei sind keine Probleme aufgetreten. Mit den implementieren Widgets sind kleine Beispiel Anwendungen möglich.

Für die Implementierung der beiden fehlenden Widgets bietet JavaFX die passenden Klassen. Die Implementierung des `Table`-Widget ist zwar komplex, allerdings ist mit dem `Tree`-Widget bereits ein sehr ähnliches implementiert worden. Daher sind beim `Table`-Widget voraussichtlich keine Probleme zu erwarten.

Dass sich JavaFX noch in der Entwicklung befindet, erschwerte die Implementierung an vielen Stellen. Bisher gibt es keine Literatur, die tiefer auf die JavaFX Architektur eingeht. Ein Grund hierfür könnte sein, dass sich die Architektur noch ändern kann. Während der Entwicklung des Prototypen wurden immer wieder Anpassungen an dem Public API vorgenommen. Das führte dazu, dass bisher geschriebene Workarounds unnötig wurden oder bestimmte Methoden neu implementiert werden mussten. Diese Anpassungen sind bis zur geplanten Version 3 vermehrt zu erwarten.

Anhand der Implementierung des Prototyps kann gesagt werden, dass sich JavaFX als Service Provider für Jo Widgets eignet. Auch die Vorteile von JavaFX waren schon zu erkennen. Dank der neuen Rendering Engine hat man den Eindruck, dass Anwendungen mit komplexen Layouts schneller laufen.

Die neuen Diagramme oder die Möglichkeit, das Aussehen mit Hilfe von CSS vollständig zu ändern, wurden bisher nicht integriert. Möglichkeiten hierfür werden im folgenden Abschnitt genannt

7.2 Ausblick

Der Service Provider für JavaFX bietet noch keine Möglichkeit, eine eigene CSS-Datei zu laden. Jo Widgets ermöglicht es, spezielle Eigenschaften einer GUI Technologie zu setzen, wie es bereits für Swing und SWT vorhanden ist. Denkbar wäre eine Implementierung zum Laden einer CSS-Datei, die beim Erzeugen der Anwendung eingebunden würde.

Abkürzungsverzeichnis

API	Application Programming Interface
AWT	Abstract Window Toolkit
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
GUI	Graphical User Interface
POM	Project Object Model
RCP	Rich Client Platform
SDK	Software Development Kit
SPI	Service Provider Interface
SWT	Standard Widget Toolkit
UI	User Interface

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[AB09] MiG InfoCom AB. *MiG Layout Quick Start Guide*. MiG InfoCom AB. 2009. (Besucht am 12. 05. 2012).

[Ani09] Chris Aniszczyk. *Single Sourcing RAP and RCP*. Eclipse Source. 2009. URL: www.slideshare.net/caniszczyk/single-sourcing-rcp-and-rap (besucht am 25. 04. 2012).

[Cas12a] Cindy Castillo. *Introduction to JavaFX Media*. Apr. 2012. URL: http://docs.oracle.com/javafx/2/media/overview.htm (besucht am 23. 05. 2012).

[Cas12b] Cindy Castillo. *JavaFX Architecture and Framework*. Oracle Corporation. Apr. 2012. URL: http://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm (besucht am 25. 04. 2012).

[Fei06] Barry Feigenbaum. *SWT, Swing or AWT: Which is right for you?* Feb. 2006. URL: http://www.ibm.com/developerworks/grid/library/os-swingswt/ (besucht am 29. 05. 2012).

[Fow11] Amy Fowler. *JavaFX2.0 Layout: A Class Tour*. Juni 2011. URL: http://amyfowlersblog.wordpress.com/2011/06/02/javafx2-0-layout-a-class-tour/ (besucht am 16. 05. 2012).

[Gro12] Michael Grossmann. *Interne Dokumentation 2012*. innoSysTec GmbH. 2012.

[Hir07] Gordon Hirsch. *Swing/SWT Integration*. 2007. URL: http://www.eclipse.org/articles/article.php?file=Article-Swing-SWT-Integration/index.html (besucht am 29. 05. 2012).

[Hom12] Scott Hommel. *Using JavaFX Properties and Binding*. Apr. 2012. URL: http://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm (besucht am 23. 05. 2012).

[JG12] Alexander Kouznetsov Joni Gordon. *Skinning JavaFX Applications with CSS*. Apr. 2012. URL: http://docs.oracle.com/javafx/2/css_tutorial/jfxpub-css_tutorial.htm (besucht am 23. 05. 2012).

[KE10] Gernot Starke Karl Eilebrecht. *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*. S. 10. Gabler Wissenschaftsverlage, 2010. URL: http://books.google.de/books?id=KBbiCYea9mUC.

[Lip11] Robert Lippert. *JavaFX soll als quelloffenes OpenJFX-Projekt weitergeführt werden*. Nov. 2011. URL: http://www.heise.de/developer/meldung/JavaFX-soll-als-quelloffenes-OpenJFX-Projekt-weitergefuehrt-werden-1371310.html (besucht am 02. 05. 2012).

[Mic12] Microsoft. *What is an Enterprise Application?* 2012. URL: http://msdn.microsoft.com/en-us/library/aa267045%28v=vs.60%29.aspx (besucht am 29. 05. 2012).

[MK06] Merrick Schincariol Mike Keith. *Pro EJB 3: Java Persistence API*. Apress, 2006. URL: http://books.google.de/books?id=fVCuB_Xq3pAC.

[Oli06] Chris Oliver. *F3*. Oracle Corporation. Nov. 2006. URL: https://blogs.oracle.com/chrisoliver/entry/f3 (besucht am 02. 04. 2012).

[Ope12] OpenJFX. *OpenJFX Projekt*. OpenJFX Project. Dez. 2012. URL: http://openjdk.java.net/projects/openjfx/ (besucht am 02. 04. 2012).

[Ora12a] Oracle Corporation. *Fancy Forms with JavaFX CSS*. Oracle Corporation. 2012. URL: http://docs.oracle.com/javafx/2/get_started/css.htm (besucht am 03. 05. 2012).

[Ora12b] Oracle Corporation. *JavaFX 2 API Dokumentation*. Oracle Corporation. 2012. URL: http://docs.oracle.com/javafx/2/api/index.html (besucht am 09. 05. 2012).

[Ora11a] Oracle Corporation. *JavaFX CSS Reference Guide*. Kapitel 1: Introduction. Oracle Corporation. Mai 2011. URL: http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html (besucht am 17. 04. 2012).

[Ora11b] Oracle Corporation. *JavaFX Roadmap*. Oracle Corporation. 2011. URL: http://www.oracle.com/technetwork/java/javafx/overview/roadmap-1446331.html (besucht am 16. 04. 2012).

[Ora11c] Oracle Corporation. *Oracle Press Release - Oracle Releases JavaFX 2.0*. Okt. 2011. URL: http://www.oracle.com/us/corporate/press/512728 (besucht am 02. 04. 2012).

[Ora] *Oracle Discussion Forums - Rendering in JavaFX*. URL: https://forums.oracle.com/forums/thread.jspa?threadID=2393856 (besucht am 29. 05. 2012).

[Pil11] Peter Pilgrim. *JavaOne 2011 JavaFX 2.0 Demonstrated on iOS, Samsung Galaxy Tab and Acer Windows Tablets*. Juni 2011. URL: http://java.dzone.com/articles/javaone-2011-javafx-20 (besucht am 02. 04. 2012).

[Red12a] Alla Redko. *Adding HTML Content to JavaFX Applications*. Apr. 2012. URL: http://docs.oracle.com/javafx/2/webview/jfxpub-webview.htm (besucht am 23. 05. 2012).

[Red12b] Alla Redko. *Introduction to JavaFX Charts*. Oracle Corporation. Apr. 2012. (Besucht am 29. 05. 2012).

[Röt11] Stefan Röttger. *Retained Mode*. Ohm Hochschule Nürnberg. März 2011. URL: http://wiki.ohm-hochschule.de/roettger/index.php/Computergrafik/RetainedMode (besucht am 25. 04. 2012).

[Sch12] Hartmut Schlosser. *Swing Ade: Die Zukunft heißt JavaFX*. Jaxenter. Apr. 2012. URL: http://it-republik.de/jaxenter/news/Swing-Ade-Die-Zukunft-heisst-JavaFX-062432.html (besucht am 29. 05. 2012).

[SEL] SELFHTML. *Aufbau zentraler Formate*. SELFHTML. URL: http://de.selfhtml.org/css/formate/zentrale.htm (besucht am 07. 05. 2012).

[sof12] software4java.com. *iPhone apps with JavaFX 2: ListView*. Apr. 2012. URL: http://blog.software4java.com/?p=24 (besucht am 25. 05. 2012).

[Som07] Ian Sommerville. *Software Engineering*. 2007. URL: http://books.google.de/books/about/Software_Engineering.html?hl=de&id=B7idKfL0H64C (besucht am 27. 05. 2012).

[Tho06] Marc Oliver Thoma. *Mac OS X 10.4 Tiger*. Hanser Verlag, 2006. URL: http://books.google.at/books?id=xXHNB1Xm3UEC.

[Wea12a] James Weaver. *Pro JavaFX 2: A Definitive Guide to Rich Clients with Java Technology*. Apress, 2012. URL: http://www.apress.com/9781430268727.

[Wea12b] James L. Weaver. *Laying Out a User Interface with JavaFX 2.0*. Oracle Corporation. März 2012. URL: http://www.oracle.com/technetwork/articles/java/layoutfx-1536156.html (besucht am 07. 05. 2012).