Bachelorarbeit

AeroFX
Native Themes für JavaFX

An der Fachhochschule Dortmund
im Fachbereich Informatik
Studiengang Praktische Informatik
erstellte Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science

von:
Matthias Kurt Walter Meidinger
Geb. am 02.10.1988
Matr.-Nr. 7082 888

Betreuer: Prof. Dr. Christian Reimann
2. Prüfer: Dr. Konstantin Koll

Dortmund, 26. August 2014

Inhaltsverzeichnis

1 Einleitung **1**

2 Stand der Forschung und Technik **3**
2.1 Stand der Forschung 3
2.2 Das Windows 7 Look & Feel 6
2.2.1 Grundkonzepte 7
2.2.2 Fokusrahmen 10
2.3 Einleitung zu JavaFX 12
2.4 Stand der Technik 14
2.4.1 JavaFX-Native-Themes 14
2.4.2 JMetro 15
2.4.3 AquaFX 16
2.4.4 Abgrenzung zu Swing 16
2.4.5 Native Skins in Swing 18
2.4.6 Maven 18

3 Evaluation **20**
3.1 JavaFX Modena 20
3.2 JavaFX-Native-Themes 21
3.3 JMetro 22
3.4 AquaFX 23
3.5 Ergebnis der Evaluation 25

4 Implementierung nativer Themes **27**
4.1 Aufbau als Open Source Projekt 27
4.1.1 Was ist Open Source? 27
4.1.2 Die BSD-Lizenz 28

4.1.3 Hosting im Web 28
4.1.4 Interesse der Community 30
4.2 Aufbau von Themes in JavaFX 30
4.3 CSS Grundlagen 32
4.4 AeroFX - Die Implementierung 35
4.4.1 AeroCheckBoxSkin 38
4.4.2 AeroRadioButtonSkin 40
4.4.3 AeroButtonSkin 42
4.4.4 AeroGroupBoxSkin 49
4.4.5 TabPane 52
4.4.6 Mnemonics 54
4.5 AeroFX - Die Probleme 55
4.5.1 Textabstand 55
4.5.2 StageStyle 56
4.5.3 Fokusreihenfolge 58
4.5.4 Fokusblinken 59

5 Validierung **62**
5.1 Lizenz 62
5.2 Dokumentation 62
5.3 Aufbau 63
5.4 Verteilung und Updates 64
5.5 Windows 7 Look & Feel 65
5.6 Tabellarische Zusammenfassung 68

6 Zusammenfassung **69**
6.1 Fazit 69
6.2 Ausblick 70

Literatur **72**

Anhang A - Vergleich von Windows 7 und AeroFX **75**

Anhang B - Codelistings **81**

Anhang C - Tabellarische Zuordnung der Quellen **84**

Anhang D - Eidesstattliche Erklärung **85**

Abbildungsverzeichnis

2.1 Dialog: Windows 7 Systemeigenschaften 7

2.2 Stati eines Buttons 9

2.3 Windows 7 Fokusrahmen 11

2.4 Windows 7 Button pulse 12

2.5 JavaFX Layout 13

2.6 Beispielhafte Maven-Ordnerstruktur 19

3.1 Windows 7 Systemeigenschaften - Original und Modena 21

3.2 JavaFX-Native-Themes Beispiel 22

3.3 JMetro Beispiel 23

3.4 AquaFX Beispiel 24

3.5 Evaluierung des Technikstandes 25

4.1 Aufbau von Skins 31

4.2 Einfluss von Cascading Stylesheets 35

4.3 Wurzelprojekt 36

4.4 Überblick über die AeroFX-Paketstruktur 37

4.5 CheckBox: Windows 7 vs. AeroFX 40

4.6 RadioButton: Windows 7 vs. AeroFX 42

4.7 Aufbau des Buttonhintergrundes 48
4.8 Vergleich der Buttons: Windows 7 und AeroFX 48
4.9 JavaFX TitledPane (Modena) 49
4.10 Windows 7 Groupbox 49
4.11 Layout AeroGroupBoxSkin 50
4.12 Einfluss von Clipping auf ein Objekt 51
4.13 Vergleich der TabPane-Skins 53
4.14 Textabstand im Vergleich 56
4.15 Fensterdekorationen im Vergleich 58

5.1 Direkter Vergleich Windows 7 - AeroFX - Modena 65
5.2 Detailvergleich Windows 7 - AeroFX - Modena 66
5.3 Validierung 68

Listings

3.1 Maven-Dependency für AquaFX 24
4.1 CSS Syntax 33
4.2 CSS Klassenselektor 33
4.3 CSS Auszug - Beispiel 1 „demo.css“ 34
4.4 JavaFX Auszug - Beispiel 1 „demo.java“ 34
4.5 JavaFX - Vollständiges Beispiel 1 35
4.6 AeroCheckBoxSkin: Konstruktor 38
4.7 AeroCheckBoxSkin: Destruktor 39
4.8 AeroCheckBoxSkin: Methode layoutChildren 39
4.9 AeroRadioButtonSkin: focusListener 40
4.10 AeroRadioButtonSkin: keyListener 41
4.11 AeroRadioButtonSkin: Ausrichten des Radios 41
4.12 AeroButtonSkin: layoutChildren 43
4.13 AeroButtonSkin: FokusListener 43
4.14 AeroButtonSkin: ArmedListener 44
4.15 AeroButtonSkin: HoverListener 44
4.16 Fokusanimation - Startwerte 45
4.17 Fokusanimation - Background-ChangeListener 1/2 46

4.18 Fokusanimation - Background-ChangeListener 2/2 47
4.19 AeroButtonSkin: ResetAnimation 48
4.20 AeroGroupBoxSkin: Konstruktor 50
4.21 AeroGroupBoxSkin: Layout-Methode 52
4.22 Styling der Klasse .tab-pane 54
4.23 Mnemonic-Styling im CSS 54
4.24 Setzen eines StageStyle 57
4.25 Listener zur Fokusweiterleitung 59
4.26 Setzen des Background Vorher/Nachher 60
4.27 Zurücksetzen der Animation 61

Überblick

Kurzfassung

Durch die Ankündigung, dass Oracle eine neue Standardtechnologie zur Entwicklung grafischer Oberflächen bereitstellt, wurde das starke Interesse der Entwicklerszene für einen Nachfolger zu Swing geweckt. JavaFX soll diese Rolle übernehmen, liefert in seiner Reinform jedoch nur die Möglichkeit, Oberflächen mit einem Oracle-eigenen Aussehen zu entwickeln. Dieses Aussehen ist durch sogenannte Skins modifizierbar, die allerdings nicht von Oracle bereitgestellt werden, sondern nur durch Eigenentwicklungen der Community realisiert werden sollen. Aufgrund des jungen Alters von JavaFX, welches als Teil von Java 8 eingeführt wurde, existieren zum jetzigen Zeitpunkt nur wenige Ansätze, solche Skins zu entwickeln.
Da der von Grund auf modernisierte und durchdachte Aufbau von JavaFX als robust und zukunftssicher gilt, ist die Wahrscheinlichkeit groß, dass es in absehbarer Zeit große Verbreitung finden wird. Um die Verbreitung zu fördern, eruiert diese Arbeit den Stand der Entwicklungen im Bereich der Skins für die Windows 7-Oberfläche und zeigt Ansatzpunkte zur Entwicklung eigener Skins für JavaFX.

Abstract

Oracle's annoucement to change the standard technology for graphical development of user interfaces away from Swing raised a strong response in the developer community. JavaFX is the announced successor of Swing, but lacks the capability of developing User Interfaces with different look & feel than the default, Modena. This look & feel can be customized with so called „skins", which, as Oracle announced, will not be developed by Oracle itself, but by the Java community. As JavaFX is an emerging technology, recently introduced with the JRE 8, there are still no projects which aim for the development of skins with native Windows look & feel.
Because of the intelligent and modern design of JavaFX, there is a high probability that it will spread across Java developer communities. To support this divulgement, this thesis investigates the state of skin-projects with native Windows 7 look & feel and shows starting points for developing own skins for JavaFX.

Kapitel 1

Einleitung

JavaFX wurde von Oracle als Nachfolger des noch immer beliebten, allerdings stark veralteten, Swing-Toolkits platziert. Beide Technologien dienen zur Entwicklung grafischer Oberflächen für Java-Anwendungen und unterstützen das sogenannte „Skinning", bei dem durch Anwenden eines Skins das Aussehen der dargestellten Oberfläche verändert wird. Aufgrund seines Alters und seiner Verbreitung existieren für Swing sehr viele Skins, die es ermöglichen, Anwendungen wie native Windows- oder OS X-Anwendungen aussehen zu lassen.
Bei JavaFX ist die Verfügbarkeit von Skins sehr gering, da es erst seit wenigen Monaten als Teil von Java 8 auf dem Markt ist. Des Weiteren wurde seitens Oracle angekündigt, dass sie keine Skins mit plattformspezifischem Aussehen für JavaFX entwickeln werden. Dieser Bedarf soll alleine durch die Community gedeckt werden. Die Entwicklung eines solchen Skins von Grund auf ist ein größeres Projekt, welches selbst einen Vollzeitentwickler einige Personentage lang binden würde. Hinzu kommt, dass Kenntnisse im Umgang mit Java und CSS benötigt werden. Diese Kombination ist ungewöhnlich, da die Sprachen aus zwei verschiedenen Welten stammen. Dementsprechend ist die Motivation, einen solchen Skin zu entwickeln, mit einem hohen Arbeitsaufwand verbunden, was der Auswahl an Skins ebenfalls nicht zuträglich ist.
Skins, die es Anwendungen ermöglichen, ein betriebssystemähnliches Aussehen anzunehmen, sind jedoch eine der Voraussetzungen, damit Entwickler JavaFX verwenden. Aufgrund dessen sind sie indirekt eine Bedingung für die Marktdurchdringung von JavaFX. Da der Autor von dem zukunftssicheren und durchdachten JavaFX-Aufbau überzeugt ist, soll diese Arbeit ihren Teil dazu beitragen, die Verbreitung dessen zu steigern. Hierzu setzt sie sich die Entwicklung eines Skins mit Windows 7 Look & Feel zum Ziel.

Zunächst erfolgt die Vorstellung des Forschungsstandes, gefolgt von der Definition des Wunschzustandes, nämlich der Oberfläche des Betriebssystems Windows 7. Im Anschluss wird ein Marktüberblick über den Stand der Entwicklungen im Bereich Skins für JavaFX inklusive einer Bewertung hinsichtlich Umsetzung und Benutzbarkeit gegeben. Nach dem Überblick über existierende Lösungen wird die Implementierung von AeroFX, dem in dieser Arbeit entwickelten Skin, inklusive einiger Besonderheiten und Probleme vorgestellt. Abgeschlossen wird die Vorstellung durch einige Überlegungen zum Aufbau des Projektes und der Frage, wie der Community die Existenz dieses Projektes bekannt gemacht werden kann. Die Validierung von AeroFX gegen die Anforderungen, die aufgestellt wurden, leitet das Fazit ein, welches nochmals eine Zusammenfassung der durchgeführten Arbeiten und einen Ausblick auf weitere Möglichkeiten bietet, in Richtung derer sich das Projekt noch entwickeln kann.

Kapitel 2

Stand der Forschung und Technik

Es folgt eine Einführung in den Stand der Forschung und die Nennung verwandter Arbeiten, ergänzt durch die Vorstellung der Zielplattform und aktueller Lösungen auf dem Markt.

2.1 Stand der Forschung

UX Design (User Experience Design) ist ein Thema, das gerade in Zeiten unterschiedlichster Endgeräte immer mehr an Bedeutung gewinnt. Aus diesem Grund hat sich der Fokus der Forschung weg vom klassischen PC und hin zu Web- und Mobile-Technologien verlagert. Hierzu finden sich viele Forschungsarbeiten, die sich dem Umsatzeinfluss von Webseiten (vgl. [Dja+14]) oder auch dem Design von sozialen Netzwerken (vgl. [Hay]) widmen. Des Weiteren wurde in letzter Zeit der Frage nachgegangen, ob der kulturelle Hintergrund bei der Nutzung einer Seite eine Rolle spielt (vgl. [Naw14]). Nicht zuletzt bleibt auch die Frage nach der Verschmelzung von UX und agilen Entwicklungsmethoden ein spannendes Feld, in dem Arbeiten (vgl. [Tia+]) entstehen.
Gerade weil das Wachstum der IT-Branche im Internet und auf mobilen Plattformen prophezeit wird (vgl. [Reh13]), sinkt das Interesse an klassischen Rich Client Anwendungen und damit auch am Konzept des PCs an sich. Dieser ist nur noch eine von vielen Möglichkeiten, Zugang zu Online-Diensten zu erlangen. Nicht zuletzt ist dieser Bereich der Usability-Forschung aufgrund seines vergleichsweise hohen Alters gut erforscht und es existieren viele bekannte Werke.
Trotzdem ist das Feld des User Experience Designs gefragt wie selten, ist es doch mittlerweile fest in den Entwicklungsprozess von neuer Software integriert. Es hat

sich hierfür der Begriff „UX Guidelines" durchgesetzt. In diesen Guidelines wird beispielsweise zusammengefasst, welche Bedien- und Anzeigeelemente es gibt, wie diese aussehen und in welchen Situationen sie einzusetzen sind (vgl. [Mic]). Hierbei werden gleich zweierlei Dinge abgedeckt, die jedoch getrennt beschrieben werden sollten: das Aussehen und das Layout.
Das Aussehen wird bei nativen Anwendungen durch das Betriebssystem bestimmt. Hierzu gehören Elemente wie das verwendete Farbschema, Größe, Farbe und Verhalten von Buttons, das Verhalten bei einem Rechtsklick und Vieles mehr. Auf JavaFX übertragen übernehmen dort Skins die Definition des Aussehens: Sie verändern alle visuellen Rückmeldungen von grafischen Elementen.
Das Layout von Oberflächen ist der zweite große Punkt im Oberflächendesign. Hierbei kommen Aspekte zum Tragen, wie beispielsweise die Tatsache, dass die Fenstersteuerungen[1] bei Windows oben rechts, bei OS X oben links angebracht sind. Ist dies noch eine eher optische Unterscheidung, so fällt spätestens beim Abfragedesign auf, wie groß die Unterschiede sind: Bei Windows befindet sich die Schaltfläche zum Annehmen (OK) links, die zum Ablehnen (Abbrechen) rechts. Diese Situation ist bei Linux genau umgekehrt. Diese Unterschiede beim Design können kaum durch einen Skin kompensiert werden, sondern müssen alleine durch den Entwickler berücksichtigt und richtig umgesetzt werden.
Zu Aussehen und Layout kommt noch eine weitere, wichtige Komponente hinzu, nämlich der Benutzer. Jeder Benutzer bringt sowohl gute als auch schlechte Erfahrungen und Erwartungen mit, die die Interaktion mit dem PC beeinflussen. Da Windows das mit Abstand am häufigsten genutzte Betriebssystem ist, haben die meisten Benutzer den Umgang mit ebendiesem erlernt. Somit haben sie ganz bestimmte Erwartungen an das Aussehen und Verhalten einer Anwendung, da sie sich an das Look & Feel von Microsoft gewöhnt haben. Erfüllt man diese Erwartungen des Benutzers bei einer Anwendung, so kann ihm die Benutzung ebenjener näher gelegt werden. Gerade im Bereich User Experience verhält es sich wie zwischen Menschen: Der erste Eindruck zählt.

> „There is also first impression for the user experience of a product. If your product needs eight hours charging before you can turn it on, if the user is faced with tens of difficult questions before you can start using your web page, or if the user's impression of the sales shop she enters is far from optimal, then you have given the user a very bad first impression of your product." ([Kra12, S. 5])

[1] Steuerelemente für das Minimieren, Maximieren und Schließen

Steve Krug hat es auf den Punkt gebracht:

> „Don't make me think!" ([Kru06, S. 11])

Um dem Anwender die Benutzung des Systems so einfach und angenehm wie möglich zu gestalten, sollte der Einstieg in die Benutzung einer Anwendung so leicht wie möglich gemacht werden. Diesbezüglich sollte das Konzept der Gewohnheit berücksichtigt werden (vgl. [AMW10, S. 12]). Hierzu gehört auch, auf die innere Konsistenz zu achten. Dazu gibt es ein gutes Beispiel:

> „The internal consistency of a product should disguise a lack of consistency in the functions it's handling. If the product is interacting with a dozen different 'backend' systems, the user shouldn't have any clue that's the case - everything should all feel like the same experience." ([AMW10, S. 25])

Neben dieser inneren Konsistenz gilt es auch die äußere Konsistenz zu berücksichtigen, also die Konsistenz gegenüber dem umgebenden Betriebssystem. Durch den richtigen Aufbau der Oberfläche und das Anpassen des Aussehens lässt sich die Aufmerksamkeit des Benutzers weg von der Benutzung und hin zu seinen Aufgaben beziehungsweise Zielen verschieben, zur Erfüllung derer er die Software benutzen wollte:

> „With the goal of allowing the user to remain focused on the goal instead of having to pay attention to the microtasks of operating the product, intuitiveness allows the user to more easily slip into engagement and retain undistracted focus and productivity." ([AMW10, S. 18])

Im Idealfall wird durch konsequente Gestaltung die Benutzung der Anwendung durch einen eingewöhnten Nutzer dahingehend erleichtert, dass er positive Bestätigung bei ihrer Benutzung empfindet.

> „The goal is of course to maximize the positive moments for users when they're using your product. And ideally to make your consumers love your product—at least some or most of the time." ([Kra12, S. 2])

Wendet man nun diese Gedanken auf das mit Abstand am weitesten verbreitete Betriebssystem (vgl. [Net14]), Windows 7, an, so lässt sich auch dort ein klares Design mit hohem Wiedererkennungswert feststellen. Aufgrund der Verbreitung

kann davon ausgegangen werden, dass ein Großteil der Nutzer die Konsistenz der Oberfläche und die Bedienkonzepte von Windows so verinnerlicht hat, dass die Konfrontation mit anderem Aussehen oder Verhalten einer unter Windows gestarteten Anwendung Probleme bereiten könnte. Als Vertreter der Windows-Familie wurde Windows 7 ausgewählt, da es sowohl im privaten Umfeld weit verbreitet ist, als auch im Unternehmensbereich gerade eingeführt wird beziehungsweise wurde. Betrachtet man nun JavaFX, so fällt auf, dass nur ein plattformübergreifender Skin mitgeliefert wird. Das heißt, dass JavaFX standardmäßig nur ein Aussehen besitzt, welches auf allen unterstützten Plattform gleich aussieht, nämlich „Modena“. Dieser Skin unterscheidet sich jedoch so grundlegend vom Windows-Design, dass diese Tatsache jedem Betrachter sofort negativ auffällt. Durch diesen Umstand ist die generelle Benutzung von JavaFX-Anwendungen unter Windows zwar möglich, aber es existiert eine klare Trennung zwischen der Java-Anwendung und dem umgebenden Betriebssystem. Dies stellt einen Entwickler, der eine Windows 7-Anwendung entwickeln möchte, vor potenzielle Probleme, da alle eingangs vorgestellten Prinzipien (im negativen Sinne) auf diesen Fall zutreffen. Je nach Art der Anwendung ergibt sich sogar noch ein weiteres Problem: Mit der Verletzung von Gestaltungsprinzipien geht oft ein Verlust des Vertrauens in die Anwendung einher,[2] welcher insbesondere im Enterprise-Umfeld oft fatale Konsequenzen nach sich ziehen kann. Unter den vorgestellten Umständen ist der Einsatz von Skins mit nativem Look & Feel von Windows nicht nur sinnvoll, sondern sogar empfehlenswert.
Diese Situation wird seitens Oracle auch nicht behoben, da in einem Statement klar gemacht wurde, dass die Entwicklung von nativen Skins nur durch die Community erfolgen wird und nicht durch Oracle (vgl. [Oraa]). Hieraus lässt sich ein klarer Bedarf an nativen Skins ableiten, dessen Abdeckung durch den Markt in den folgenden Kapiteln untersucht wird. Um eine Bewertung durchführen zu können, sollte jedoch zunächst das erklärte Ziel, die Windows 7-Plattform, vorgestellt werden.

2.2 Das Windows 7 Look & Feel

Microsoft Windows 7 ist die erklärte Zielplattform, für die dieser Skin entwickelt werden soll. Die Wahl erklärt sich sowohl durch die Verbreitung als auch durch den steigenden Einsatz bei Unternehmen. Des Weiteren existiert eine umfassende

[2] vgl. „The very same issues that break engagement also have a tendency to injure the trustworthiness of the product(...)“ ([AMW10, S. 30])

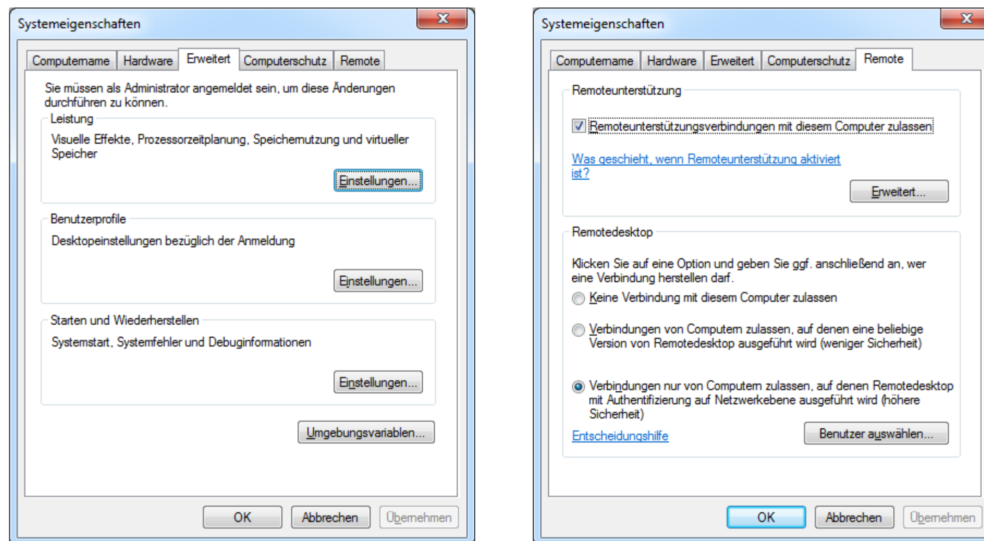


Abbildung 2.1: Dialog: Windows 7 Systemeigenschaften

Sammlung an Richtlinien, die sogenannten „User Experience Guidelines“ (vgl. [Mic]), die einen Großteil des Designkonzeptes von Microsoft vorstellen.
Als Beispiel, wie diese Guidelines umgesetzt wurden, soll hier der Dialog „Systemeigenschaften“ dienen, aus dem einige Abzüge Abbildung 2.1 zu entnehmen sind.
Das Design der Steuerelemente in Windows 7 ist konsistent und einem Großteil der Benutzer bekannt. Beim Design wurde von vornherein Wert auf klare Strukturen und gute Lesbarkeit gelegt. Im Folgenden werden einige grundlegende Designkonzepte und Steuerelemente in Windows 7 vorgestellt.

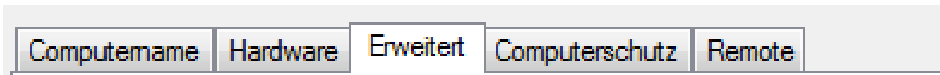
2.2.1 Grundkonzepte

Elemente

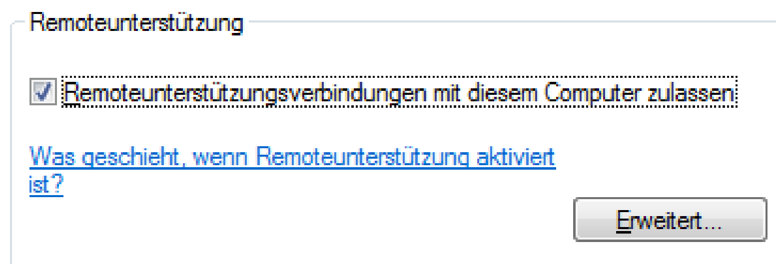
Beschäftigt man sich mit eben vorgestelltem Dialog, so sollte man zunächst einmal die Nomenklatur definieren und die verwendeten Design- und Steuerelemente vorstellen. Dies soll im Folgenden geschehen, bevor auf Details der Oberfläche eingegangen wird, um etwaige Unschärfen bei der Benennung der Elemente zu vermeiden. Die Begriffe sind aufgrund ihrer Herkunft komplett im Englischen gehalten.

Designelemente

- Tabpane - Mit diesem Element wird eine Seiten- bzw. Registerstruktur realisiert, in der per Linksklick auf die gewünschte Seite gesprungen werden kann. Bei Windows 7 wird die aktuell gewählte Registerkarte größer und ohne Abgrenzung zum Inhalt angezeigt.



- Groupbox - Dient zur Gruppierung mehrerer Steuerelemente in Funktionsgruppen. Jede Groupbox trägt einen Titel, der die Gruppe erklären soll.

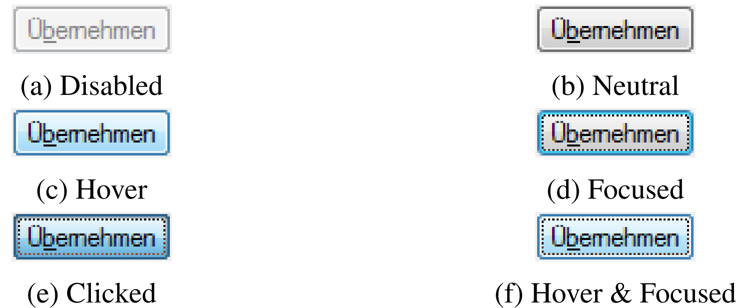


- Togglegroup - Nicht sichtbar, steuert jedoch Radiobuttons. Aus allen Radiobuttons einer Gruppe kann immer nur ein einziger gewählt werden. Bei Windows 7 kann ausschließlich der ausgewählte Radiobutton per „Tab“ erreicht werden, sprich: Nur dieser eine Radiobutton kann mit einem Fokusrahmen belegt sein. Des Weiteren kann, wenn der Fokus auf einem Radiobutton liegt, innerhalb der Gruppe mit den Pfeiltasten zwischen den Elementen gewechselt werden.

Keine Verbindung mit diesem Computer zulassen

Verbindungen von Computern zulassen, auf denen eine beliebige Version von Remotedesktop ausgeführt wird (weniger Sicherheit)

Verbindungen nur von Computern zulassen, auf denen Remotedesktop mit Authentifizierung auf Netzwerkebene ausgeführt wird (höhere Sicherheit)



(a) Disabled (b) Neutral (c) Hover (d) Focused (e) Clicked (f) Hover & Focused

Abbildung 2.2: Stati eines Buttons

Steuerelemente
Neben den Designelementen existieren noch Steuerelemente, die aufgrund ihrer Bekanntheit hier nicht nochmals grafisch vorgestellt werden. Zu den Steuerelementen gehören Buttons, Textboxen, Checkboxen und Radiobuttons, also Elemente, die zur eigentlichen Dialogsteuerung und Kommunikation genutzt werden.

Status

Jedes Steuerelement in Windows 7 kann mehrere Status annehmen. Diese werden nun am Beispiel eines Buttons vorgestellt:

- Disabled - Das Element kann nicht benutzt werden und ist ausgegraut.
- Neutral - Der Ausgangszustand, in dem sich jedes aktive Element befindet.
- Hover - Auch „Mouseover“ genannt. Die Maus steht auf dem Element.
- Focused - Kann entweder durch „Tabbing“ oder nach dem „Clicked“-Status auftreten.
- Clicked - Wird über einem Element die Auswahltaste gedrückt gehalten, so ist dies der „Clicked“-Status.

Wie bereits angedeutet, hängen bestimmte Status voneinander ab, andere hingegen schließen sich gegenseitig aus oder können sich stapeln und dadurch neue Zustände erzeugen. Der Clicked-Status kann einerseits durch das Drücken der linken Maustaste ausgelöst werden, während die Maus auf dem Element platziert ist. Jedoch kann dieser Status andererseits auch erreicht werden, wenn sich das

Element im Focused-Zustand befindet und die Leertaste gedrückt wird. Befindet sich ein Element im Clicked-Status, so kann der Hover-Status nicht angenommen werden. Als Letztes sei noch das „Stacking“ erwähnt, also das Auftürmen von Status.
Beobachtet werden kann dieser Effekt bei einer besonderen Konstellation, nämlich Focused und Hover. Diese Kombination kann logisch aus dem Statusmodell abgeleitet werden und wird durch eine leicht geänderte Farbkombination gegenüber Focused kenntlich gemacht.[3]
Vergleicht man diesen Status mit den Ausgangsstatus, so stellt man fest, dass der Fokusrahmen aus dem Focused-Status mit der Buttonfarbe aus Hover kombiniert wurde. Zur besseren visuellen Trennung wurde eine ein Pixel breite, weiße Umrandung zwischen Button- und Fokusrahmen eingefügt. Gerade solche Details wie die verschiedenen Zustände der Steuerelemente werden zwar nicht bewusst vom Benutzer wahrgenommen, erleichtern allerdings die Benutzung von UIs unter Windows 7.

Mnemonics

Hinter diesem kryptischen Begriff steht ein Konzept, das in Zeiten von Touchbedienung etwas von seiner Wichtigkeit verloren hat: Tastenkürzel. Sieht man sich die Abbildungen in den vorigen Kapiteln genauer an, so fällt auf, dass jedes Kontrollelement einen unterstrichenen Buchstaben in seinem Beschriftungstext aufweist. Dieser Unterstrich wird eingeblendet, sobald die „Alt“-Taste im aktiven Fenster gedrückt wird. Drückt man die Alt-Taste und die Taste für den unterstrichenen Buchstaben (z.B. Button übernehmen: „b“) zusammen, so entspricht das einem Linksklick auf das Steuerelement. Auch heute noch werden diese Kürzel viel genutzt, insbesondere im geschäftlichen Umfeld, also von Administratoren und professionellen Anwendern, die schnell und effizient arbeiten müssen.

2.2.2 Fokusrahmen

Wie erwähnt, sind bei allen Elementen gleiche Effekte auch optisch identisch. Es folgt die Vorstellung des Fokusrahmens, eines der zentralen Elemente des UI-Designs. Der Fokusrahmen wird als visuelle Rückmeldung genutzt, welches Element gerade per Tastatur (Tabulator) ausgewählt ist und durch Drücken der Leertaste selektiert werden würde. Es ist eine ein Pixel breite, gepunktete (dotted)

[3] vgl. Abbildung 2.2f

Linie, die einen Abstand von einem Pixel zwischen jedem Rahmenpixel aufweist. Sie findet sowohl bei Buttons als auch bei Checkboxen und Radiobuttons Verwendung, wie in Abbildung 2.3 erkennbar ist.
Trotz der auf den ersten Blick gemeinsamen Gestaltung gibt es noch Unterschiede zwischen den einzelnen Implementierungen.

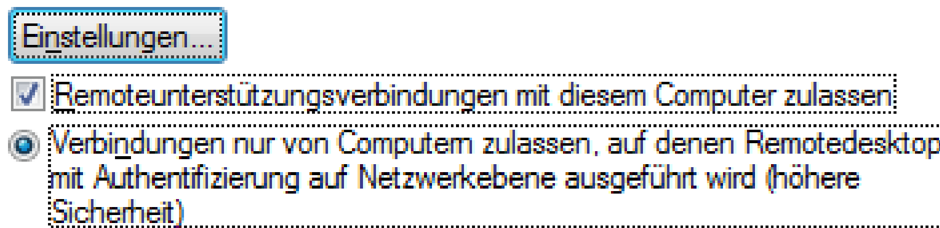


Abbildung 2.3: Windows 7 Fokusrahmen

Bei Checkboxen und Radiobuttons wird nur das Label, also die Beschriftung der Option, vom Rahmen eingefasst, das Steuerelement selbst bleibt frei.
Bei Buttons orientiert sich der Rahmen an der Außenkante des Buttons, ist aber einige Pixel nach innen versetzt. Der Stil ist, wie eingangs erwähnt, identisch gehalten. Jedoch fällt bei einem fokussierten Button eine andere Besonderheit auf:
Befindet sich ein Button in diesem Zustand, so wechselt seine Hintergrundfarbe zwischen den Farben der Status „unselektiert“ und „hover“. Abbildung 2.4 zeigt exemplarisch einen solchen Ablauf.

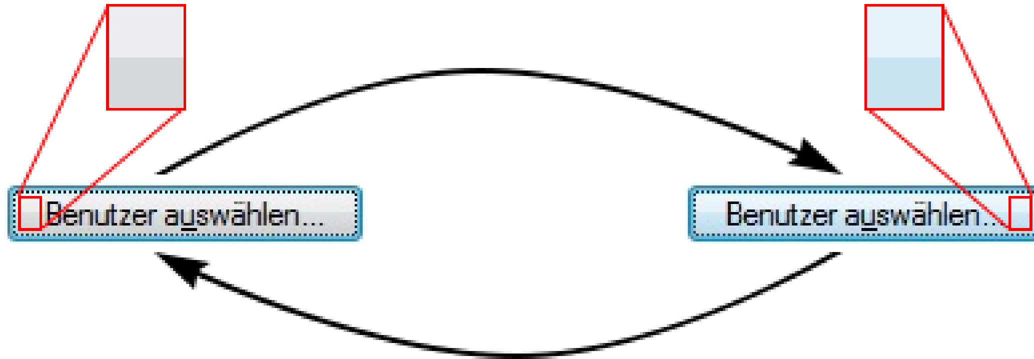


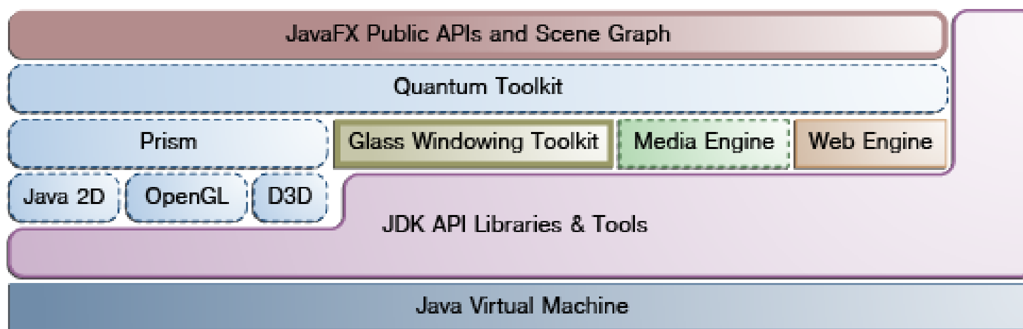
Abbildung 2.4: Windows 7 Button pulse

Das Pulsieren des Buttonhintergrundes markiert an dieser Stelle das Ende der Detailvorstellung des Windows 7 Look & Feels. Nachdem der Zielzustand nun klar definiert und bekannt ist, kann mit dem Überblick über JavaFX und die aktuelle Marktsituation fortgefahren werden.

2.3 Einleitung zu JavaFX

JavaFX wird von Oracle als Nachfolger von Swing positioniert. Geschrieben in Java und versehen mit Public-APIs ist es seit JRE7u6 (Java Runtime Environment, Version 7, Update 6) standardmäßig in jeder Installation enthalten. Seit Erscheinen von JRE8 wird es auch offiziell als stabiler Bestandteil der Java-Umgebung geführt.
Es ist ein von Grund auf neues Framework zur Entwicklung von Rich Client Applications (vgl. [Ora14a]), welches den Schwerpunkt auf grafische Programmierung (UI) legt. JavaFX wurde von Anfang an auch für die mobile Nutzung konzipiert, da es bereits jetzt Multitouch-fähig ist, obwohl iOS und Android noch nicht offiziell zu den unterstützten Plattformen zählen. Die bewusst gewählte Architektur spiegelt sich auch in der Tatsache wider, dass das Styling der Komponenten von der eigentlichen Logik entkoppelt wurde. Anwendungen können in einem deklarativen XML-Dokument erstellt werden, während das Aussehen der einzel-

nen Elemente per CSS gesteuert wird. Eine weitere Besonderheit, die mit Version 8 Einzug hielt, ist die Möglichkeit des sogenannten „nativen Packagings“. Dieser Vorgang ermöglicht es, JavaFX-Anwendungen auf Zielsystemen ausführen zu können, auf denen kein JRE installiert ist. Dies minimiert den Wartungsaufwand und potentielle Probleme, da die Zielsysteme keinerlei Vorbedingungen erfüllen müssen, um Programme auszuführen.
Im Folgenden wird ein genauerer Blick auf den generellen Aufbau geworfen, wie ihn Abbildung 2.5 illustriert.



(Entnommen aus [Ora14b])

Abbildung 2.5: JavaFX Layout

Arbeitet man sich von oben nach unten vor, so stößt man zunächst auf die Public APIs und den Scene Graph. JavaFX ist zu großen Teilen direkt in die API von Java integriert, wodurch ohne Aufwand Kernfeatures von Java wie zum Beispiel Generics, Lambdas oder auch Multithreading in JavaFX-Anwendungen genutzt werden können. Der zweite Teil der obersten Schicht ist der Scene Graph. JavaFX organisiert Anwendungen auf dieselbe Art wie Windows, nämlich in einer Baumstruktur mit Nodes. Außer dem Wurzelelement hat jede Node genau einen Vorgänger und beliebig viele Nachfolger. Im Vergleich zu Windows besitzen JavaFX-Nodes zusätzlich noch eine eindeutige ID und beliebig viele „style“-Klassen. Sowohl IDs als auch Klassen können per CSS angesprochen und so die Node beziehungsweise deren Aussehen verändert werden.
Es folgt das Quantum Toolkit, welches dem JavaFX-Layer den Zugriff auf Prism und das Glass Windowing Toolkit erlaubt und Threading-Regeln des Renderings verwaltet. Die eigentlichen Renderaufträge werden wiederum durch Prism verarbeitet, welches, je nach Plattform, die passenden Hardwarerenderer (OpenGL, D3D, ...) anspricht. Die plattformspezifische Implementierung des Glass Windowing Toolkit soll an dieser Stelle die Vorstellung des Aufbaus beenden.

Dieses Toolkit ist die Basis des Grafikstacks und für die nativen OS-Services zuständig wie beispielsweise Fenstermanagement und Timer.
Auf dieses Grundgerüst wird standardmäßig der Skin „Modena“ angewendet, der von Oracle entwickelt wird. Dieser Skin besteht aus einem CSS, welches die Style-Klassen der Nodes im Scene Graph selektiert und deren Aussehen verändert. Dies ist der Grundgedanke hinter der Nutzung eines Skins: Das Aussehen einer Anwendung ohne Einflussnahme auf die Funktionalität zu verändern. Da sich Oracle auf die Entwicklung des OS-unabhängigen Skins „Modena“ beschränkt, sind aus dem Anwenderbereich bereits erste Anstrengungen zu erkennen, Skins mit nativem Look & Feel verschiedener Betriebssysteme zu entwickeln. Hierbei sind insbesondere die nachfolgenden Projekte zu nennen.

2.4 Stand der Technik

In diesem Abschnitt wird sowohl die verwendete Kerntechnologie JavaFX als auch der Stand der Skinentwicklung am Markt vorgestellt. Abgerundet wird die Vorstellung durch eine Abgrenzung zum alten Standard, Swing, und der Vorstellung des Build-Tools Maven.

2.4.1 JavaFX-Native-Themes

JavaFX-Native-Themes ist ein JavaFX2-Projekt des Blogs software4java[4], welches als Quellcode-Repository[5] verfügbar ist. Der Quellcode liegt vollständig vor, jedoch ist unklar, unter welcher Lizenz das Projekt steht, da weder eine LICENSE-Datei noch Copyright-Vermerke im Quelltext vorhanden sind. Ferner scheint keine aktive Entwicklung an dem Projekt mehr stattzufinden, da das letzte Update des Quellcodes laut Bitbucket am 22.02.2013 erfolgte. Das Repository besteht im Wesentlichen aus einigen Demo-Anwendungen, Screenshots und CSS-Dateien. Das Projekt beinhaltet sowohl einen Max OS X- als auch einen Windows 7-Skin, wobei beide nur erste beispielhafte Implementation zeigen und somit als unfertig zu bezeichnen sind. Ein Großteil des Projektes besteht, wie erwähnt, aus Demo-Anwendungen, die zur Laufzeit ein Austauschen der Skins ermöglichen, was einen direkten Vergleich der Stile erlaubt.

[4] http://software-4-java.com
[5] https://bitbucket.org/software4java/javafx-native-themes

Die Nutzung des gesamten Projekts ist auf das händische Einbinden der CSS-Datei in die Anwendung ausgelegt. Da keine Java-Logik verwendet wurde, wird auf spezielle Methoden zum Zugriff auf eigene Objekte und Controls verzichtet. Ein wichtiger Punkt ist die Abhängigkeit des Skins vom Standard-Skin Modena. Viele Eigenschaften und Verhaltensweisen von Controls werden über das Modena CSS gesteuert. Das hier vorgestellte Projekt nutzt eine Eigenschaft von CSS-Dateien, das Stapeln von mehreren Dokumenten aufeinander, und setzt mit seinem CSS darauf auf (`Scene.add()`). Dies erspart auf den ersten Blick viel Arbeit, da das Verhalten nicht nochmals definiert werden muss, führt jedoch auch eine Abhängigkeit ein: Wird seitens Oracle etwas an der Implementierung beziehungsweise Definition der Controls im Modena-Skin geändert, so kann dies zu unerwünschtem oder unerwartetem Verhalten in der Zielanwendung führen, die den vorgestellten Windows 7-Skin einsetzt. Des Weiteren entsteht durch den Verzicht auf ein Fassaden-Entwurfsmuster ein ganz anderes Problem: Sollte der Autor sich im Laufe der Zeit entscheiden, seinen Programmaufbau entscheidend zu verändern, so muss jeder Entwickler, der den Skin benutzt, sein Programm anpassen, damit es mit der neuen Version funktioniert.

2.4.2 JMetro

JMetro ist das Projekt des Freelance Software-Engineers Pedro Duque Vieira, der die neuesten Updates hierzu in seinem Blog[6] vorstellt. Das Ziel dieses Projektes ist das Anbieten eines Look & Feels, welches Windows 8 nachempfunden ist. Die Formulierung „nachempfunden“ wurde hier gezielt gewählt, da selbst der Autor davon spricht, dass er sich von der „Metro UI“ von Windows 8 nur inspirieren lässt. Im Hinblick auf die Lizenzierung des Projekts sind Parallelen zum Projekt JavaFX-Native-Themes erkennbar, da auch hier weder eine Lizenzdatei noch ein Lizenzvermerk in der Software eingepflegt ist. Das Projekt selbst beinhaltet eine helle und eine dunkle Version des Skins, jeweils als eine CSS-Datei. Der Rest der Dateien beschränkt sich auf diverse FXML-Dateien, die Layouts und Controls für verschiedenste Demo-Programme beinhalten. Auch hier fällt auf, dass nur per CSS und nicht mit Java-Klassen gearbeitet wurde, wodurch ausschließlich die mitgelieferten Controls von JavaFX gestylt werden können. Sonderelemente wie eine Groupbox können so nicht realisiert werden. Des Weiteren soll nicht unerwähnt bleiben, dass auch bei diesem Projekt auf den Einsatz von Fassaden verzichtet wurde, was Bedenken zur Änderungssicherheit auslöst.

[6] http://pixelduke.wordpress.com/

Als letzter Punkt ist noch die Verfügbarkeit der Software hervorzuheben.
Die Softwareverteilung erfolgt über eine Dropbox-Freigabe, wodurch zwar eine Versionierung gegeben ist, allerdings keine Aussagen zu Updates oder den Unterschieden von Versionen getroffen werden können. Außerdem steht die Frage im Raum, wie sicher Dropbox als Hostingplattform ist, da die Freigabe jederzeit aufgehoben beziehungsweise das Nutzerkonto seitens Dropbox jederzeit gelöscht werden kann.

2.4.3 AquaFX

AquaFX ist ein Open Source Projekt von Claudine Zillmann, einer Softwareentwicklerin aus Dortmund, welches als Abschlussarbeit von ihr entwickelt wurde. Es existiert eine Projektseite zu dieser Arbeit, welche unter http://aquafx-project.com erreicht werden kann. Das Ziel dieses Projektes ist die Entwicklung eines nativen Look & Feels für Mac OS X. Es steht unter BSD-Lizenz, ist also sowohl privat als auch kommerziell kostenfrei nutz- und veränderbar. Das Projekt besteht aus einer zentralen Fassade zuzüglich diverser Fachklassen, in denen teilweise zusätzliche Funktionalitäten, wie zum Beispiel das Pulsieren eines Buttons, realisiert werden. Vervollständigt wird das Projekt durch einige Demos, die die Ähnlichkeit zu ausgewählten Systemdialogen aufzeigen. Der Quellcode ist direkt als Repository[7] verfügbar, darüber hinaus aber auch als fertige JAR (Java-Library) direkt auf der Projektseite oder als Maven-Projekt bei Maven Central. Die Version, die dort zum Download angeboten wird, ist Version 0.1.

2.4.4 Abgrenzung zu Swing

JavaFX wurde, wie schon erwähnt, von Oracle als Nachfolger zu Swing als Standard UI-Technologie positioniert. Daher wird in diesem Kapitel etwas näher auf die Unterschiede und Gemeinsamkeiten von Swing und JavaFX eingegangen.
Swing galt bis einschließlich JRE7 als empfohlene Technologie, um grafische Nutzeroberflächen zu realisieren. Seine Ursprünge reichen zurück bis ins Jahr 1996, als es unter dem Namen „Internet Foundation Classes“ erstmals veröffentlicht wurde. Seitdem ist die Nutzung von Swing unverändert hoch, da es bis zum jetzigen Zeitpunkt kein vergleichbares Toolkit gab, welches von Oracle (bzw. davor von Sun) selbst empfohlen wurde. Es ist plattformunabhängig, da es komplett

[7] https://bitbucket.org/czillmann/javafxaqua

in Java geschrieben ist, bietet eine sehr modulare Architektur, die „plugging“ unterstützt, also das Einbinden von eigenen Objekten, und ist, weil alle Klassen von JComponent erben, komponentenbasiert.
Ferner ist Swing eine sogenannte „Lightweight UI“, was bedeutet, dass ihm lediglich ein leeres Fenster vom OS zur Verfügung gestellt wird und es volle Kontrolle darüber hat, wie das Zeichnen durchgeführt wird. Somit ist beispielsweise ein Swing-Button nicht durch einen OS-Button repräsentiert, sondern ein reines Java-Objekt, welches in das Fenster gezeichnet wird. Jedoch erbt JComponent von der AWT-Klasse `Container`, denn Swing erweitert eigentlich AWT.
Dies erlaubt Swing die volle Kontrolle über elementare Systemfunktionen, denn AWT ist wiederum eine „Heavyweight UI“, also OS-abhängig.
Im Gegensatz hierzu stellt sich JavaFX als ein robustes und zukunftssicheres Modell vor, welches nach einer gewissen Einführungsspanne endgültig Swing ablösen wird. JavaFX wird in seiner jetzigen Version seit 2011 entwickelt. Wie bereits in Kapitel 2.3 erläutert, ist auch JavaFX eine „Lightweight UI“, da es komplett in Java geschrieben ist und nur das Glass Windowing Toolkit OS-spezifischen Code beinhaltet.
Verglichen mit Swing ist das Verändern von Elementen in JavaFX einfacher: Muss in Swing von der Klasse abgeleitet und Funktionen überschrieben werden, kann das Aussehen in JavaFX durch das Anpassen von Eigenschaften in einer CSS-Datei verändert werden[8]. Außerdem bietet JavaFX die Möglichkeit, Oberflächen in deskriptiven XML-Dateien zu definieren, wodurch eine nachhaltige Trennung von Layout und Design erreicht wird. Diese Grenzen sind bei Swing bei Weitem nicht so klar gezogen. Um Entwicklern den Umstieg von Swing zu JavaFX zu vereinfachen, besitzt JavaFX sogar eine „Swing-Node“, die eine sanfte Umstellung auf JavaFX ermöglicht.
Beim Vergleich von Swing und JavaFX soll auch die Performanz nicht unerwähnt bleiben. Hierzu hat sich Oracle in einem Technology Network-Artikel geäußert:

> „Our overall impression is that JavaFX provides the same high level of performance as Swing when it comes to assembling and rendering screens with many components. And JavaFX provides much better performance in the area of transitions and animations (Swing provides very limited functionality in this area). Overall, performance is not an issue.“ (vgl. „Experiences with JavaFX“ [Ora13b])

[8] s. Kapitel 4.3

Abschließend sei gesagt, dass die Architektur von Swing in die Jahre gekommen ist und sich trotz wiederholter Modernisierungen dem Ende ihres Lebenszyklus neigt. JavaFX hingegen bedient sich sowohl an Designideen bestehender Java-Lösungen als auch an Microsofts WPF (Windows Presentation Foundation), die die deskriptive UI-Gestaltung per X(A)ML einführte. Zusammen mit diversen Feinheiten, deren Aufzählung hier den Rahmen sprengen würde, zeichnet sich JavaFX als zukunftssicher, robust und durchdacht aus, weshalb es als Ablösung von Swing logisch und nachvollziehbar positioniert wurde.

2.4.5 Native Skins in Swing

Nachdem die theoretische Abgrenzung zu Swing bereits im vorherigen Kapitel erfolgte, soll nun darauf eingegangen werden, wie genau Skins in Swing funktionieren. Zunächst einmal sollte geklärt werden, dass Skins im Umfeld von Swing „Look and Feel“ (LAF) genannt werden.
Auf allen von Java unterstützten Plattformen stehen die LAFs mit Namen „Motif“ und „Metal“ zur Verfügung, auf den Plattformen Linux, Mac OS und Windows zusätzlich noch jeweils ein plattformspezifischer LAF, mit dem sich Anwendungen nicht vom Betriebssystem unterscheiden (vgl. [Wik14]).
Da Swing pixel- und nicht vektorbasiert ist, werden auch alle Grafiken pixelbasiert dargestellt. Als Beispiel soll hier das Zeichnen einer Combobox dienen. Soll eine solche in Swing gezeichnet werden, so übergibt Swing dem Betriebssystem ein Canvas-Objekt (eine „Zeichenfläche“) zusammen mit einigen Parametern, wie der Größe. Auf diesem Objekt zeichnet dann das Betriebssystem durch einen nativen Funktionsaufruf die angeforderte Combobox als pixelbasierte Grafik.
Dieser Vorgang wäre theoretisch auch in JavaFX möglich, jedoch ist JavaFX komplett vektorbasiert, d.h. jegliche Skalierung des Objektes würde unweigerlich Qualitätseinbußen mit sich bringen. Sollte der Fall eintreten, dass die Entwickler der Betriebssysteme die APIs um entsprechende Funktionen erweitern, bliebe noch immer die Möglichkeit, ein solches Verhalten in JavaFX zu implementieren.

2.4.6 Maven

> „Maven ist ein Build-Management-Tool der Apache Software Foundation und basiert auf Java. Mit ihm kann man insbesondere Java-Programme standardisiert erstellen und verwalten.“ ([Wik14])

Wie das Zitat bereits erläutert, ist Maven ein Tool, welches den Entwickler während des gesamten Entwicklungsprozesses vom ersten Prototypen bis zum Deployment unterstützen soll. Maven-Projekte vereinfachen das Einbinden von externen Libraries und Programmen und bieten durch eine einheitliche XML-Konfigurationsdatei auch die Reproduzierbarkeit von Ergebnissen. Wird eine Software als Maven-Projekt realisiert, so kann zum Abschluss das Projekt beispielsweise in ein Online-Repository von Maven-Projekten hochgeladen werden, um es allen Maven-Nutzern komfortabel zur Verfügung zu stellen.
Zudem zeichnen sich Maven-Projekte durch eine besondere Ordnerstruktur aus, die sich mittlerweile sogar als Standard-Projektaufbau durchgesetzt hat. Diese Struktur ist, wie viele andere Parameter auch, optional und kann in der Haupt-Konfigurationsdatei `pom.xml` angepasst werden. Maven wurde unter dem Leitsatz „Convention over Configuration“ entwickelt, weshalb viele Features bei Einhaltung der Konvention ohne explizite Konfiguration funktionieren. Abbildung 2.6 zeigt einen beispielhaften Aufbau.

```
project root
 └─src/
 │  └─main/
 │  │  └─java/ - Java-Quelltext
 │  │  └─resources/ - Weitere Ressourcen (CSS, XML,..)
 │  └─test/
 │     └─java/ - JUnit-Testfälle
 └─target/
 │  └─classes/ - Kompilierte Java-Klassen
 └─pom.xml - Maven-Konfigurationsdatei
```

Abbildung 2.6: Beispielhafte Maven-Ordnerstruktur

Durch die einheitliche Struktur ist eine automatisierte Verwaltung und Verarbeitung durch Maven gewährleistet, die Ziel beim Einsatz dieser Software ist. Selbst eine kürzlich bei Google durchgeführte Studie (vgl. [Seo+14]) kommt zu dem Schluss, dass ein Großteil der Fehler beim Kompilieren von Software auf fehlerhaft aufgelöste Projektabhängigkeiten, Klassen und Quelldateien zurückzuführen ist, was durch Maven verbessert werden kann.
Nicht zuletzt sei erwähnt, dass sich die Projektstruktur nicht nur innerhalb von Maven durchgesetzt hat, sondern immer größere Verbreitung findet. Hiervon zeugt zum Beispiel auch die Kompatibilität zu Ant und Gradle.

Kapitel 3

Evaluation

Nachdem die Grundlagen des Windows 7-spezifischen Designs, und damit der Zielzustand, bekannt sind, kann nun darauf eingegangen werden, wie gut oder schlecht dieses Ziel von den im Kapitel 2.4 vorgestellten Projekten erfüllt wird.

3.1 JavaFX Modena

Beginnen sei zunächst mit der Ausgangssituation: JavaFX mit dessen Standard-Skin Modena. Dieser wurde von Oracle als plattformübergreifender, allgemeiner Skin entwickelt und reiht sich in eine lange Folge von Cross-Platform-Skins für verschiedenste Techniken (AWT, Swing,...) ein.
Das Problem, welches alle diese Skins mit sich bringen, wurde bereits in Kapitel 2.1 eingehend erläutert. Vergleicht man den eben vorgestellten Windows 7-Standard mit diesem Skin, so fallen sofort mehrere Unterschiede ins Auge, wie Abbildung 3.1 zeigt.
Sowohl die Größen der Elemente als auch deren Farbgebung unterscheiden sich maßgeblich von Windows 7. Nutzt ein Anwender eine JavaFX-Applikation, welche auf Modena basiert, so können sich Probleme ergeben, da die vom Anwender gelernten Farbschemata und Verhaltensweisen hierauf nicht anwendbar sind.
Der Aufbau des Skins selbst besteht aus einem umfangreichen CSS, welches alle Elemente, die in JavaFX verwendet werden können, abdeckt.

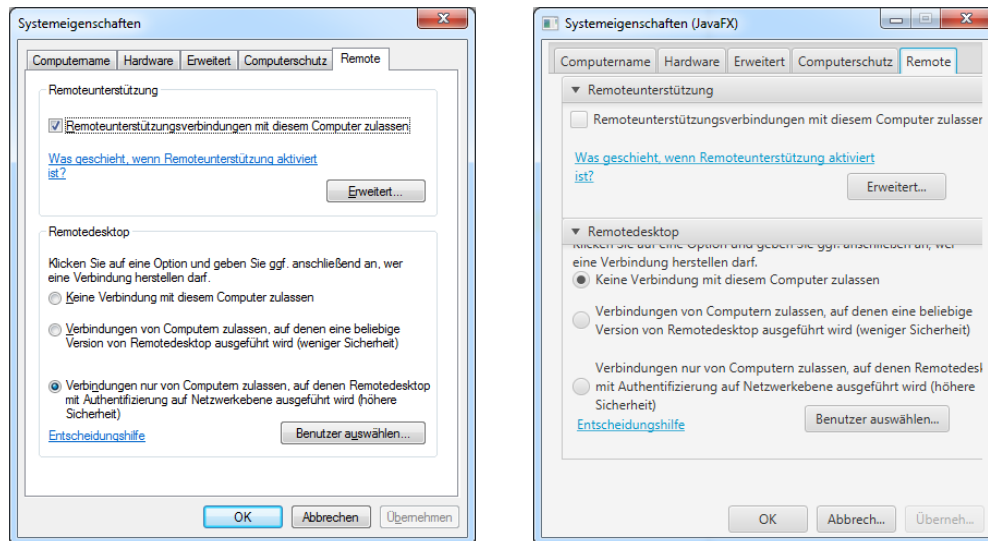


Abbildung 3.1: Windows 7 Systemeigenschaften - Original und Modena

Eine Dokumentation des Quelltextes ist faktisch nicht vorhanden, jedoch existiert eine externe Dokumentation, welche den Aufbau der Klassen und Elemente beschreibt. Bei dieser fehlt allerdings die Erklärung, wie Oracle bei der Gestaltung in Modena vorgegangen ist.
Das Modena CSS steht unter einer anerkannten Open Source Lizenz, wodurch die Verwendung klar geregelt ist.

3.2 JavaFX-Native-Themes

Der vom Projekt angebotene Skin ist eine reine CSS-Datei, die bereits viele Elemente umfasst, welche jedoch größtenteils aus der CSS-Datei des „Modena“-Skins übernommen und noch nicht angepasst wurden. Bereits implementiert wurde der Button, allerdings ausschließlich der neutrale und der default-Zustand. Der Button zeigt keine Veränderung bei „hover“- oder „click“-Ereignissen, was auf die fehlende Definition dieser Pseudoklassen zurückgeführt werden kann.
Auch die Checkbox ist bereits gut erkennbar, sie wurde einschließlich des Mark, also des „Hakens“ in der Box, implementiert. Auch hier fehlen jegliche weiteren Pseudozustände wie beispielsweise hover.
Zusammenfassend ist zu sagen, dass der Skin bisher für wenig mehr als eine Demonstration zu verwenden ist.

Von einem produktiven Einsatz ist dringend abzuraten, da auch die Codebasis weder dokumentiert ist noch sonderlich robust erscheint.
Aufgrund der hier genannten Kritikpunkte, zuzüglich zur ungeklärten Lizenzfrage eignet sich die Codebasis auch nicht als Ausgangspunkt für die Entwicklung eines eigenen Skins. Trotzdem soll an dieser Stelle ein optischer Eindruck des Entwicklungsstands nicht fehlen, daher zeigt Abbildung 3.2 die mitgelieferte JavaFX-Demo, unter Windows 7 kompiliert und ausgeführt.

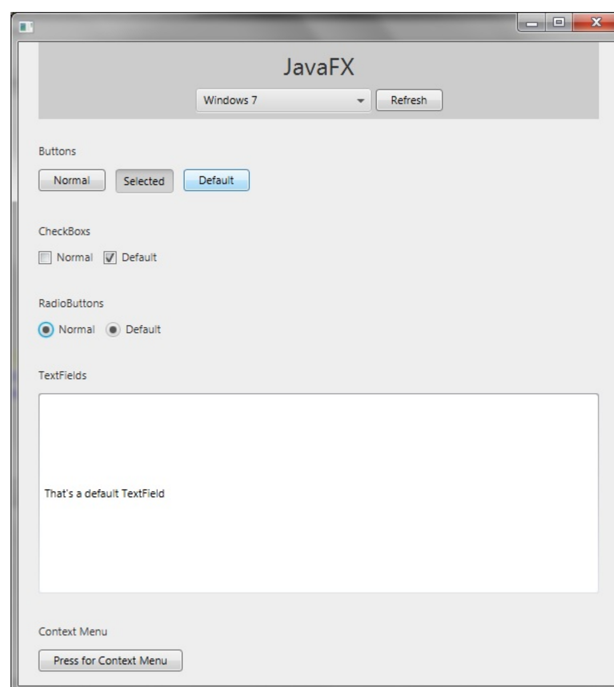


Abbildung 3.2: JavaFX-Native-Themes Beispiel

3.3 JMetro

Die Situation bei JMetro zeigt viele Parallelen zum eben vorgestellten Projekt JavaFX-Native-Themes, sowohl beim Aufbau als auch von der Dokumentation. Auch hier wurde auf eine zentrale Klasse als Interface verzichtet. Des Weiteren existieren zum Zeitpunkt der Evaluation einige Probleme in der Codebasis. Unter anderem ist zu vermuten, dass sich einige Interfaces oder Vaterklassen geändert

haben, seitdem das letzte Update des Skins veröffentlicht wurde, da sich das Projekt nicht mehr kompilieren lässt, wenn alle Klassen miteinbezogen werden.
Diese Problematik in Kombination mit der ungeklärten Lizenzfrage lassen das Projekt als Ausgangspunkt für eigene Entwicklungen ausscheiden. Hinzu kommt, dass der Skin von Windows 8 inspiriert wurde und daher nur in ganz bestimmten Szenarien sinnvoll eingesetzt werden kann, wie zum Beispiel beim Ausführen einer JavaFX-Browseranwendung in einem Browser, der aus der Modern UI (ehemals Metro UI) von Windows 8 heraus gestartet wurde. Abbildung 3.3 vermittelt einen Eindruck, wie eine solche Anwendung aussehen könnte.

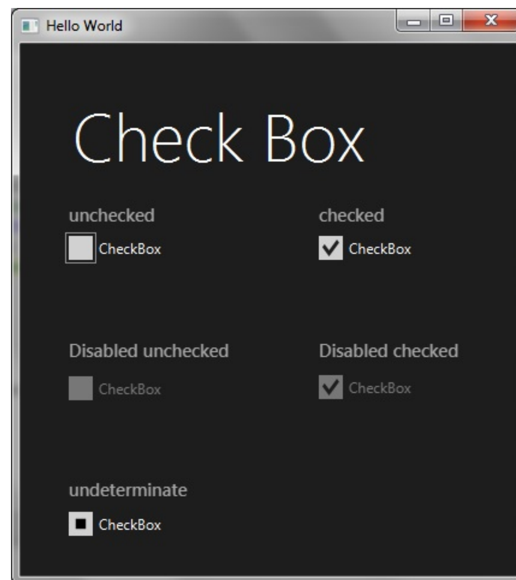


Abbildung 3.3: JMetro Beispiel

3.4 AquaFX

AquaFX nimmt eine gesonderte Position bei den hier evaluierten Projekten ein. Aufgrund der Zielplattform OS X scheidet es von vornherein als Ausgangspunkt für die Skinentwicklung aus.
Was jedoch bei der Evaluierung sofort ins Auge fiel, ist der durchdachte Aufbau des Projekts. Durch den Einsatz von Maven kann das Endprodukt sehr leicht in Anwendungen genutzt werden und die damit einhergehende klare Paketstruktur

ist selbsterklärend. Vervollständigt wird diese Situation durch die Dokumentation mittels Javadoc.
Um auch hier einen optischen Eindruck zu hinterlassen, wurde der Beispieldialog „Systemeigenschaften“ in Abbildung 3.4 mit dem AquaFX-Skin ausgestattet, um eine gewisse Vergleichbarkeit zu erzeugen.

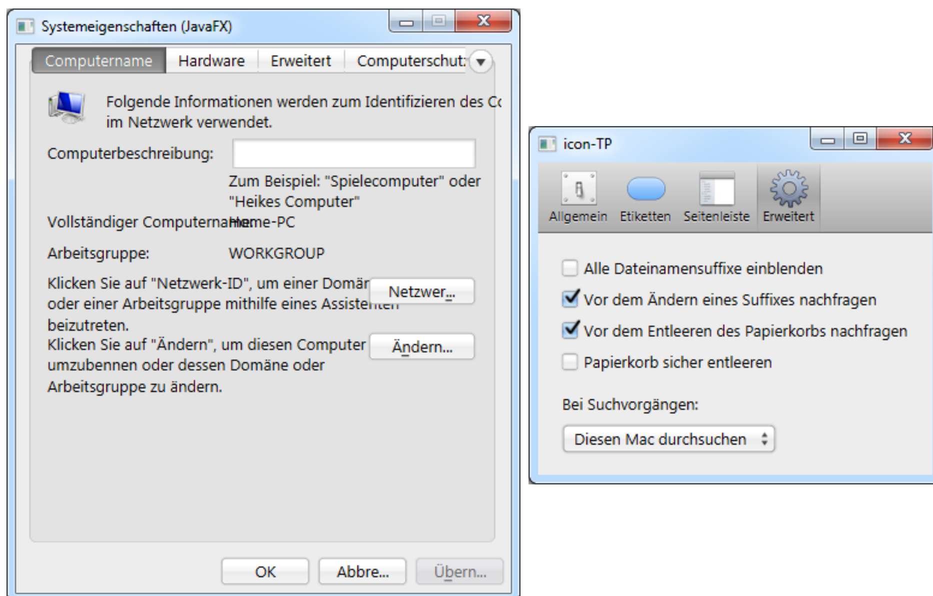


Abbildung 3.4: AquaFX Beispiel

Bei genauerer Betrachtung des Aufbaus zeigt sich die Erfahrung der Autorin mit Enterprise-Anwendungen. Durch die einfache Nutzung ergibt sich ein echter Mehrwert, da man, um AquaFX im eigenen Maven-Projekt zu nutzen, lediglich die Zeilen aus Listing 3.1 in die Konfigurationsdatei einfügen muss.

```
1 <dependency>
2     <groupId>com.aquafx_project</groupId>
3     <artifactId>aquafx</artifactId>
4     <version>0.1</version>
5 </dependency>
```

Listing 3.1: Maven-Dependency für AquaFX

Ein weiter Vorteil offenbart sich durch den Einsatz eines Fassaden-Entwurfsmusters, da die gesamte Anwendung durch den Aufruf von `AquaFX.style();` komplett in ein natives Look & Feel konvertiert werden kann. Dieses Entwurfsmuster versteckt die Details der Implementierung hinter einer Klasse, die sich auf der Benutzerseite des Skins nicht mehr oder nur in seltenen Fällen ändert. Durch Einsatz dieses Entwurfsmusters muss selbst bei einer umfassenden Umstrukturierung des Skins kein Entwickler seine Programme anpassen, um sie lauffähig zu halten.
Das Projekt überzeugt durch überlegten Aufbau, gute Dokumentation und hohe Codequalität, scheidet jedoch trotzdem aus, da es für OS X entwickelt wurde.

3.5 Ergebnis der Evaluation

Um die Evaluation zusammenzufassen und die Vergleichbarkeit greifbar zu machen, wurden die wesentlichen Punkte der Evaluation nochmals in Abbildung 3.5 festgehalten. Dabei erfolgte eine dreistufige Bewertung von gut (+) über neutral (O) zu schlecht (-). Bewertet wurde die Klarheit und Anwenderfreundlichkeit der Lizenz, die Dokumentation des Projekts, der Aufbau des Codes, die Verteilung und das Management von Updates und Downloads und die Ähnlichkeit des Skins mit Windows 7.

Projektname	JFX Modena	JFX-Native-Themes	JMetro	AquaFX
Lizenz	+	-	-	+
Dokumentation	o	-	-	+
Aufbau	+	o	o	+
Updateverteilung	+	o	-	+
Windows 7 L&F	-	o	-	-

Abbildung 3.5: Evaluierung des Technikstandes

Wie zu sehen ist, scheitern alle Kandidaten an der Ähnlichkeit zu Windows 7, daher lautet das Ergebnis der Evaluation: Die Entwicklung eines eigenen Skins ist der Weiterentwicklung eines dieser Projekte vorzuziehen. Jedoch schneidet in den anderen Kategorien AquaFX jedes Mal mit einem „Gut“ ab, weshalb einige der Entscheidungen, die während der Entwicklung von AquaFX getroffen wurden, in AeroFX übernommen werden.

Die erste Entscheidung ist die Wahl des Namens. AquaFX ist eine Kombination aus „Aqua“, der Oberfläche von OS X, und „FX“, als Kürzel für JavaFX. Der Name AeroFX wird auf ähnliche Weise gebildet, nämlich aus dem Spitznamen der Windows 7-Oberfläche, „Aero“, und dem Kürzel „FX“, um die Zugehörigkeit zu JavaFX darzustellen.
Eine weitere Idee, die auf Basis von AquaFX aufkam, ist der Aufbau als Maven-Projekt. Durch die Nutzung von Maven wird eine einfache Abhängigkeitsverwaltung garantiert, des Weiteren findet Maven aufgrund der Reproduzierbarkeit der Umgebung auch im kommerziellen Bereich große Verbreitung. Außerdem kann durch eine Veröffentlichung von AeroFX auf Maven Central, einem Online-Repository, die Verbreitung und Verfügbarkeit des Skins sehr leicht maximiert werden.
Um durch zukünftige Umbauten und Veränderungen innerhalb des Skins nicht alle Entwickler, die AeroFX in ihren Projekten einsetzen, zu zwingen, ihre Anwendungen anzupassen, erfolgt der Einsatz eines Fassaden-Entwurfsmusters. Auch dieses Entwurfsmuster findet bei AquaFX Einsatz, um den internen Aufbau und die APIs vor dem Anwender zu verbergen und einheitlichen Zugriff auf alle Funktionen des Skins zu bieten.

Durch Einbringen der zuletzt genannten Punkte entsteht ein robuster und zukunftssicherer Aufbau, der die Grundlage für ein erfolgreiches Projekt mit vielen Anwendern schafft. Hinzu kommen die Veröffentlichung des Skins unter einer anwenderfreundlichen Lizenz und das Bereitstellen des kompletten Quellcodes über eine Hostingseite, um vollständige Transparenz und Erweiterbarkeit des Projekts zu garantieren. Des Weiteren wird durch die Veröffentlichung des Quelltextes die Weiterentwicklung und Fehlerbehebung von AeroFX vereinfacht, was schlussendlich die Qualität des Gesamtproduktes verbessert.
Nicht zuletzt sei erwähnt, dass auch in Betracht gezogen wurde, AquaFX aufgrund seines guten Designs um einen Windows-Skin zu erweitern. Aufgrund vieler OSX-spezifischer Tweaks und Workarounds in AquaFX ist eine Erweiterung jedoch sehr zeitaufwendig und unübersichtlich. Daher wurde die Idee zugunsten einer Neuimplementierung verworfen. Jedoch wurden, wie eben erwähnt, viele Designentscheidungen aus AquaFX abgeleitet oder übernommen.

Kapitel 4

Implementierung nativer Themes

Im folgenden Kapitel wird die Umsetzung von Themes im JavaFX näher vorgestellt.

4.1 Aufbau als Open Source Projekt

Wenden wir uns zunächst den Strategien zu, um die Nutzung und Entwicklung des Skins durch die Community voranzutreiben. Ein Kernaspekt ist die Wahl der Lizenz, unter der das Projekt steht. Hierbei fiel die Wahl auf eine Open Source Lizenz, genauer gesagt auf eine 2-Clause BSD-Lizenz.
Bevor darauf eingegangen wird, welche Vor- und Nachteile diese Lizenz hat, sollte zunächst erläutert werden, was unter dem Begriff „Open Source" zu verstehen ist.

4.1.1 Was ist Open Source?

Oft werden die Begriffe „Free" und „Open Source" in ähnlicher Weise verwandt, jedoch bezeichnen sie unterschiedliche Konzepte.
Der Kerngedanke entstand in den 70er Jahren am MIT, wo man davon ausging, dass jegliche Verbesserungen an Software und Systemen der Allgemeinheit zur Verfügung gestellt werden sollten. Mit anderen Worten ist ein Open Source Projekt ein Projekt, bei dem jedem interessierten Menschen der komplette Quelltext einer Anwendung zur Verfügung steht.
Die Begriffsunterscheidung zwischen Free und Open Source ist auf die Ambiguität des Begriffes „Free" zurückzuführen.

Der Begriff „Free Software“ entstand nach dem Leitsatz „It’s free as in freedom—think free speech, not free beer“(vgl. [Fog05, S. 7]), also der Freiheit, Software zu benutzen und zu verändern, ohne dafür zu zahlen oder Rechte zu verletzen. Um die sprachliche Unschärfe auszuräumen, wurde der Begriff der „Open Source Software“ geprägt, der auch heute noch Verwendung findet.

4.1.2 Die BSD-Lizenz

Ausgeschrieben die „Berkeley Software Distribution“, ist eine Open Source Lizenz, die neben der GPL häufig Einsatz findet. Sie wurde von der University of California, Berkeley eingeführt (daher der Name), um darunter wichtige Teile einer Unix-Implementation zu veröffentlichen. Sie basiert zu großen Teilen auf der MIT-Lizenz, enthielt in ihrer ersten Fassung jedoch eine Werbeklausel.
Aufgrund dieser war die erste Version der Lizenz inkompatibel zur GPL, weshalb besagte Klausel in späteren Versionen entfernt wurde. Heutzutage ist eine 2-Clause BSD-Lizenz weit verbreitet, in der nur die ersten beiden Punkte der Lizenz geführt werden; es fehlen sämtliche Einschränkungen zu Werbung und Verwendung von Namen des Autors oder der Firma des Autors (vgl. [Fog05, S. 176, f.]).
Gesondert hervorzuheben ist, dass die BSD-Lizenz kein Copyleft enthält, was bedeutet, dass Projekte, die BSD-lizensierten Code nutzen, nicht zwangsweise unter BSD-Lizenz stehen müssen. Dies ermöglicht die Verwendung von Bibliotheken und Code in proprietären und geschlossenen Produkten, wodurch die Akzeptanz in der Softwareindustrie gesteigert werden kann.

4.1.3 Hosting im Web

Da aufgrund der offenen Lizenz jeder interessierte Entwickler an dem Projekt arbeiten kann, wurde der Quelltext bewusst leicht zugänglich als GitHub-Repository veröffentlicht. GitHub ist eine Hostingplattform für Softwareprojekte, die unter anderem ein Quellcodeverwaltungssystem, einen Bugtracker und ein Wiki pro Projekt zur Verfügung stellt. Sie ist kostenlos und der Quellcode sogar ohne Registrierung zugänglich. Durch den einfachen Zugang soll die Einstiegshürde minimiert werden, die Entwickler von der Beteiligung an offenen Projekten abhält. Durch die Nutzung dieser Plattform ist eine kollaborative Weiterentwicklung des Projekts auch im großen Personenkreis problemlos möglich. Um die Vorteile der Plattform vollständig zu verstehen, folgt eine kurze Einführung in den geplanten Workflow.

Der Pull-Request-Workflow

Besagter Workflow ist eine vergleichsweise einfache Methode, um kollaborativ an Softwareprojekten zu arbeiten. Gehen wir von folgendem Beispiel aus: Ein Entwickler möchte sich am Projekt „Verwaltungssoftware“ beteiligen, da ihm im Bereich der Finanzplanung die Umsetzung einiger Features nicht gefällt, beziehungsweise er eine Idee hat, wie dieser Bereich ansprechender gestaltet werden kann. Der Entwickler führt zunächst ein `git clone` des Projekts durch, hierbei wird der gesamte Entwicklungsstand, inklusive aller Änderungen seit Projektbeginn, auf sein Entwicklungssystem kopiert. Im Anschluss daran implementiert der Entwickler seine Vorschläge, entwickelt neue Features oder behebt Bugs in der Finanzplanung. Ist die Entwicklung seiner Meinung nach beendet, so kehrt er zum GitHub-Repository „Verwaltungssoftware“ zurück und stellt einen „Pull Request“. Dabei erfolgt eine Benachrichtigung des Kernprojektteams „Verwaltungssoftware“, inklusive der entwickelten Änderungen. Vereinfacht gesagt ersetzt der Pull Request die Diskussion zwischen den Entwicklern. Es kann ein Code-Review der angebotenen Änderungen durchgeführt und bei Bedarf sogar weitere Veränderungen und Patches am Code vorgenommen werden. Stimmt das Kernteam nun zu, so werden die Änderungen auf die öffentliche Codebasis angewendet. Ab diesem Zeitpunkt sind alle Änderungen fester Bestandteil des Projekts „Verwaltungssoftware“.
Da die Entwicklung des Skins ein Teil der Anforderungen zur Erlangung des akademischen Grades „Bachelor of Science“ ist, wurden bis zum jetzigen Zeitpunkt noch keinerlei Änderungen von außen akzeptiert. Des Weiteren wird das Repository, in dem das AeroFX-Projekt gehostet ist, erst nach Abschluss dieser Arbeit publik gemacht.

Als weiterer Schritt, um die Nutzung der Software zu vereinfachen und die Akzeptanz zu steigern, wurde das Projekt bereits in einer ersten Version auf Maven Central veröffentlicht, einem frei zugänglichen Repository-Service für Maven-Dependencies. Ist die Veröffentlichung durchgeführt, so reicht ein entsprechender Eintrag in der Maven-Konfigurationsdatei, damit Maven das Projekt automatisch bei Central herunterlädt und einbindet.
Außerdem wurde die Domain www.aerofx.org gekauft und eine entsprechende Projekthomepage ins Leben gerufen, die den Skin inklusive einiger Kernfeatures vorstellt und als Ausgangspunkt für weitere Informationen zum Projekt dient.

4.1.4 Interesse der Community

Das Interesse der Community wurde durch Gastbeiträge in Hendrik Ebbers' Blog, www.guigarage.com, geweckt. Dieser Blog erfreut sich großer Beliebtheit in der JavaFX-Community, unter anderem liest ein Teil des JavaFX-Entwicklerteams regelmäßig mit. Dementsprechend fielen die Reaktionen aus: Es folgten mehrere Empfehlungen auf der Oracle-nahen Newsseite www.jfxexperience.com, wo die eben erwähnten Blogbeiträge verlinkt wurden.
Hinzu kommen die sogenannten „Follower" auf GitHub. Es existiert die Möglichkeit, Projekten auf GitHub zu folgen. Dabei wird man über jede Änderung und jedes Update benachrichtigt, das bei dem entsprechenden Projekt durchgeführt wird.
Nicht unerwähnt sollen auch die „Follower" des Twitter-Accounts des Autors bleiben. Durch konsequente Verbreitung und Ankündigung von Updates und Bugfixes über diesen Account dient er ebenfalls als Schnittstelle zu interessierten Entwicklern.

4.2 Aufbau von Themes in JavaFX

Wie bereits erwähnt, hat Oracle bei der Entwicklung von JavaFX auf eine klare Trennung von Form und Funktion geachtet. Das Aussehen der Komponenten wird in erster Linie durch CSS definiert. Sollten die Möglichkeiten, die per CSS gegeben sind, nicht ausreichen, so besteht immer die Möglichkeit, von den „SkinBase"-Klassen in JavaFX zu erben und diese um eigene Formen oder Logik zu erweitern.
Zunächst folgt jedoch ein Überblick zum grundsätzlichen Aufbau, wie er bei anspruchsvollen Skins eingesetzt wird. Dieser ist Abbildung 4.1 zu entnehmen.
Zentraler Bestandteil eines jeden Skins ist eben erwähntes CSS. Im Normalfall wird ein Großteil des Designs über eine CSS-Datei realisiert. Insbesondere Definitionen wie Größe, Hintergrund- und Rahmenfarbe lassen sich problemlos anpassen. Sind anspruchsvolle Funktionen oder die volle Kontrolle über das Layout von Komponenten gewünscht, so muss von der JavaFX-Skinklasse abgeleitet werden, die zu jedem Element existiert. Diese Skinklassen definieren neben dem Aussehen der Elemente auch das Verhalten, welches ein weiterer Grund für das Ableiten sein kann.

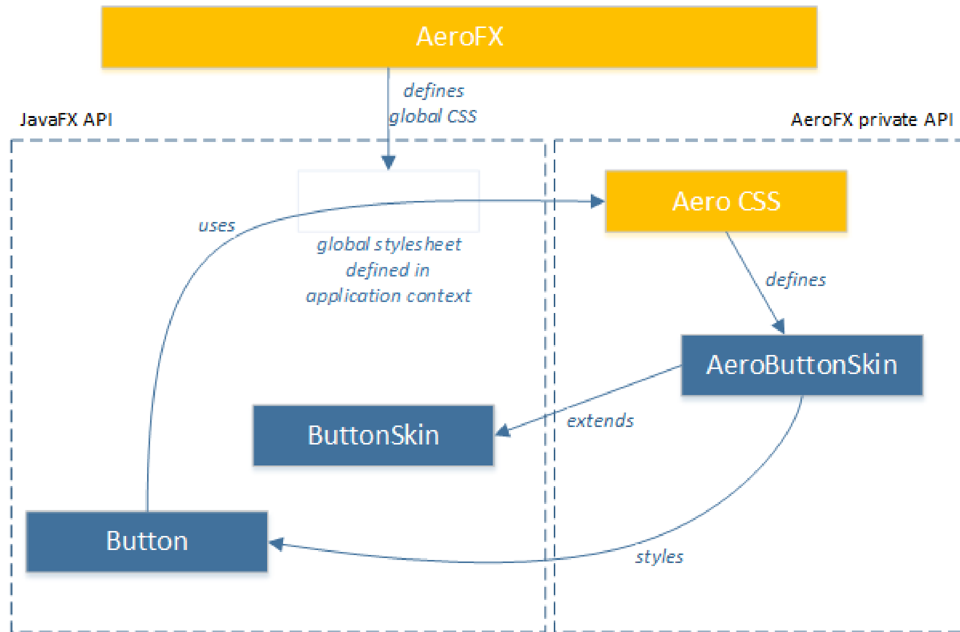


Abbildung 4.1: Aufbau von Skins

Als Beispiel sei hier die Klasse „ButtonSkin“ genannt, die das Aussehen und Verhalten eines Buttons steuert. Sie kann durch Ableiten, beispielsweise in eine Klasse „AeroButtonSkin“, erweitert werden. Durch das Überschreiben des Konstruktors und der Funktion `layoutChildren` kann das Aussehen des Buttons beliebig verändert werden.[1]
Durch die CSS-Anweisung `-fx-skin` wird dabei die Java-Klasse mit dem CSS-Pendant verknüpft, sodass beim Einbinden der CSS-Datei in eine Anwendung die abgeleitete Skinklasse genutzt wird.
Erwähnte <Control>Skinklassen sind derzeit noch nicht in der offiziellen Dokumentation von Oracle auffindbar, da sie noch nicht zur Nutzung freigegeben wurden. Diese Konstellation ergab sich, da zum Zeitpunkt dieser Arbeit noch nicht ersichtlich war, ob sich die Skinklassen zukünftig nochmals ändern.
Aus demselben Grund sind auch die Ableitungen der Controls selbst, also beispielsweise die Oracle-eigene Implementation des Buttons, als private markiert, da auch diese sich noch ändern könnten. Was jedoch bereits als öffentlich, und damit als stable, markiert wurde, sind die Interfaces, die den Controls zugrunde liegen.

[1]Details hierzu folgen in Kapitel 4.4

4.3 CSS Grundlagen

„Cascading Style Sheets“ lautet der volle Name dieser Technologie, die vor allem im Bereich des Webdesign beheimatet ist. Der Name gibt direkte Hinweise auf die wichtigsten Funktionen dieser Sprache, die in ihrer heutigen Ausprägung seit 1994 existiert. Übersetzt man ins Deutsche, so lautet der Name „stufenförmige Gestaltungsvorlagen“. CSS ist eine deklarative Sprache für Stilvorlagen, die beschreiben, wie Elemente aussehen sollen. Die besondere Eigenschaft von CSS ist das „cascading“, also das Stapeln einzelner CSS-Dateien. Bei diesem Vorgang werden mehrfach definierte Werte für bestimmte Elemente überschrieben. Dies erlaubt eine vergleichsweise flexible Einbindung von mehreren Skripten, beispielsweise einer grundlegenden CSS-Datei, welche das Farbschema definiert, und mehreren CSS-Dateien, die abhängig von aufrufenden Endgerät (PC, Tablet, Smartphone) eingebunden werden und die Größe und das Layout einer HTML-Seite bestimmen. Die aktuell als stabil eingestufte Version ist 2.1; am Nachfolger CSS 3.0 wird seit 2000 gearbeitet.
JavaFX nutzt zur Gestaltung von Elementen ebenjene deklarative Sprache, jedoch wurde der Parser, den JavaFX nutzt, um CSS-Dateien einzulesen, um einige Features erweitert. Zunächst sollte erwähnt werden, dass JavaFX den CSS-Standard in Version 2.1 verarbeitet, der einige Erweiterungen aus CSS 3.0 enthält. Zusätzlich zu diesen Erweiterungen wurde von Oracle das -fx-Präfix bei allen Eigenschaften eingeführt, die vom JavaFX CSS-Parser verarbeitet werden sollen. Alle Sprachelemente sind darauf ausgelegt, dass ein Skinning-CSS von jedem am Markt befindlichen Parser verarbeitet werden kann, selbst, wenn er die fx-Erweiterungen nicht kennt. Dahinter steht der Gedanke, dass seitens des Programmierers sowohl CSS-Code zum Styling einer Webseite, als auch einer JavaFX-Anwendung, in ein und dasselbe CSS eingetragen werden.
Nachdem die Abgrenzung zwischen Standard-CSS und der JavaFX-spezifischen Version nun erfolgt ist, wird anhand von Beispielen die Integration und Nutzung in JavaFX aufgezeigt.

Einzelne Elemente werden über einen Selektor ausgewählt, die Eigenschaften, die verändert werden sollen, folgen in geschweiften Klammern. Listing 4.1 verdeutlicht die Syntaxoptionen nochmals.

```
Klasse[:Pseudoklasse][ > Kindklasse][, Klasse2, ...] {
    Eigenschaft1: Wert1;
    Eigenschaft2: Wert2;
    ...
    Eigenschaftn: Wertn;
}
```

Listing 4.1: CSS Syntax

Jedes Element in JavaFX besitzt eine eigene Style-Klasse, die dem Namen entspricht. So kann ein Button im CSS per `.button` angesprochen werden, eine Checkbox durch `.check-box`. Diese Klassen sind standardmäßig vergeben und werden bei der Initialisierung des Objektes gebunden. Möchte man einzelnen Elementen einen anderen Stil zuweisen, so ist der Einsatz von IDs empfehlenswert. Diese müssen pro CSS-Datei einzigartig sein und werden per `#ID` selektiert.
Insbesondere auf die beiden optionalen Parameter Pseudoklasse und Kindklasse soll nun noch genauer eingegangen werden. In Kapitel 2.2.1 wurde auf die Status eingegangen, in denen sich Steuerelemente in Windows 7 befinden können. Diese Status können über die Pseudoklasse gezielt selektiert werden. Zu diesen zählen unter anderem auch `hover` und `armed`.
Alle Status, die in Kapitel 2.2.1 vorgestellt wurden, sind direkt in CSS abgebildet. Die Kindklasse, die durch den >-Operator angegeben wird, ermöglicht eine andere Art des Designs. Alle komplexen Elemente in JavaFX bestehen aus mehreren Unterelementen, auch eine Checkbox, die hier als Beispiel dient, besitzt Kinder, nämlich eine `box` und ein Label. Die Klasse box beinhaltet selbst nochmals Kinder, weswegen ein solcher Selektor wie in Listing 4.2 durchaus möglich und üblich ist.

```
.check-box:disabled > .box{
    -fx-background-color: red;
}
```

Listing 4.2: CSS Klassenselektor

Der vorgestellte Codeblock färbt den Kasten der Checkbox rot, sobald diese sich im Zustand `disabled` befindet. Generell kann über eine Verkettung durch Kommata eine beliebige Anzahl an Klassen selektiert werden, um das Zuweisen identischer Eigenschaften an eine Vielzahl von Objekten zu vereinfachen.

Um dem Leser eine Gesamtübersicht zu vermitteln, folgt ein einfaches Beispiel, in dem ein Dialog mit einem Button und einem Label per CSS gestylt wird. Hierbei wird gleichzeitig noch das Hinzufügen von Klassen und IDs zu Java-Objekten demonstriert, die im Anschluss per CSS angesprochen werden. Zunächst zeigt Listing 4.3 den Inhalt des CSS, um zu verdeutlichen, was geschehen soll.

```
1 /* Zugriff auf ID*/
2 #welcome-text{
3     -fx-fill: blue;
4 }
5
6 /*Zugriff auf Klasse*/
7 .specialButton{
8     -fx-background-color: orange;
9 }
```

Listing 4.3: CSS Auszug - Beispiel 1 „demo.css“

Das Element mit der ID `welcome-text` erhält eine blaue Schriftfarbe, wohingegen die Klasse `specialButton` die Hintergrundfarbe Orange erhält. Nun bindet man wie in Listing 4.4 die ID und die StyleClass an Java-Objekte:

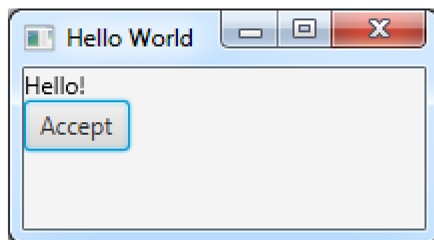
```
1 Text scenetitle = new Text("Hello!");
2 scenetitle.setId("welcome-text");
3
4 Button buttonAccept = new Button("Accept");
5 buttonAccept.getStyleClass().add("specialButton");
```

Listing 4.4: JavaFX Auszug - Beispiel 1 „demo.java“

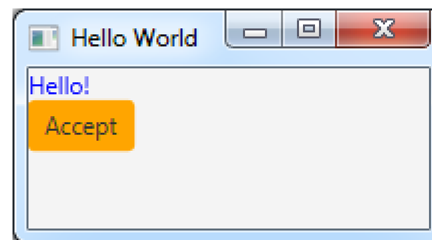
Ein vollständiges Beispiel erfordert etwas mehr Code, damit es lauffähig wird. Da JavaFX, wie erwähnt, mit einem Graphenmodell arbeitet, wird immer eine Root-Node benötigt, in die alle anderen Elemente eingehängt werden.
Des Weiteren muss der Anwendung bekannt gemacht werden, dass ein CSS vorliegt, mit dem die Anwendung gestylt werden soll. Zuletzt werden der Fenstertitel und die Szene gesetzt, nur noch gefolgt von der Anweisung, das Fenster zu zeichnen, wie in Listing 4.5 zu sehen ist. Die Auswirkungen des CSS werden in Abbildung 4.2 illustriert. Abbildung 4.2a zeigt den Beispielcode ohne, 4.2b mit angewandten Definitionen aus `demo.css`.

```
1 public class Main extends Application {
2   @Override
3   public void start(Stage primaryStage) throws Exception{
4     VBox root = new VBox();
5     Text scenetitle = new Text("Hello!");
6     scenetitle.setId("welcome-text");
7     Button buttonAccept = new Button("Accept");
8     buttonAccept.getStyleClass().add("specialButton");
9     root.getChildren().add(scenetitle);
10    root.getChildren().add(buttonAccept);
11    root.getStylesheets().add(getClass()
12      .getResource("demo.css").toExternalForm());
13    primaryStage.setTitle("Hello World");
14    primaryStage.setScene(new Scene(root, 300, 275));
15    primaryStage.show();
16  }
17  public static void main(String[] args) {
18    launch(args);
19  }
20 }
```

Listing 4.5: JavaFX - Vollständiges Beispiel 1



(a) Ohne CSS



(b) Mit CSS

Abbildung 4.2: Einfluss von Cascading Stylesheets

4.4 AeroFX - Die Implementierung

Nachdem alle Grundlagen zum Verständnis der Arbeit dargelegt wurden, kann im Folgenden das Projekt an sich mit seinen Details und Besonderheiten vorgestellt werden.
Durch den Aufbau als Maven-Projekt ist, wie in Kapitel 2.4.6 aufgezeigt, eine gewisse Ordner- und Paketstruktur bereits vorgegeben.

Daher wird nur auf die projektspezifischen Ordnerstrukturen eingegangen, die Maven-spezifischen Ordner hingegen nicht näher erläutert.
Das gesamte Projekt besteht aus 3 Maven-Projekten. Dem „Vaterprojekt“ (Abbildung 4.3), einem „Skin“-Projekt (Abbildung 4.4a) und einem „Demo“-Projekt (Abbildung 4.4b). Diese Trennung ermöglicht die Veröffentlichung des Skins auf Maven Central ohne den Ballast einer Demo-Anwendung.
Die Vorstellung wird durch einen Überblick über den Aufbau eröffnet, gefolgt von einigen Klassen und Detaillösungen.

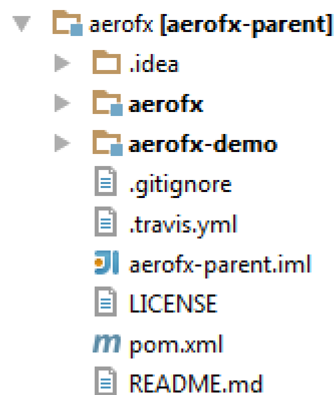
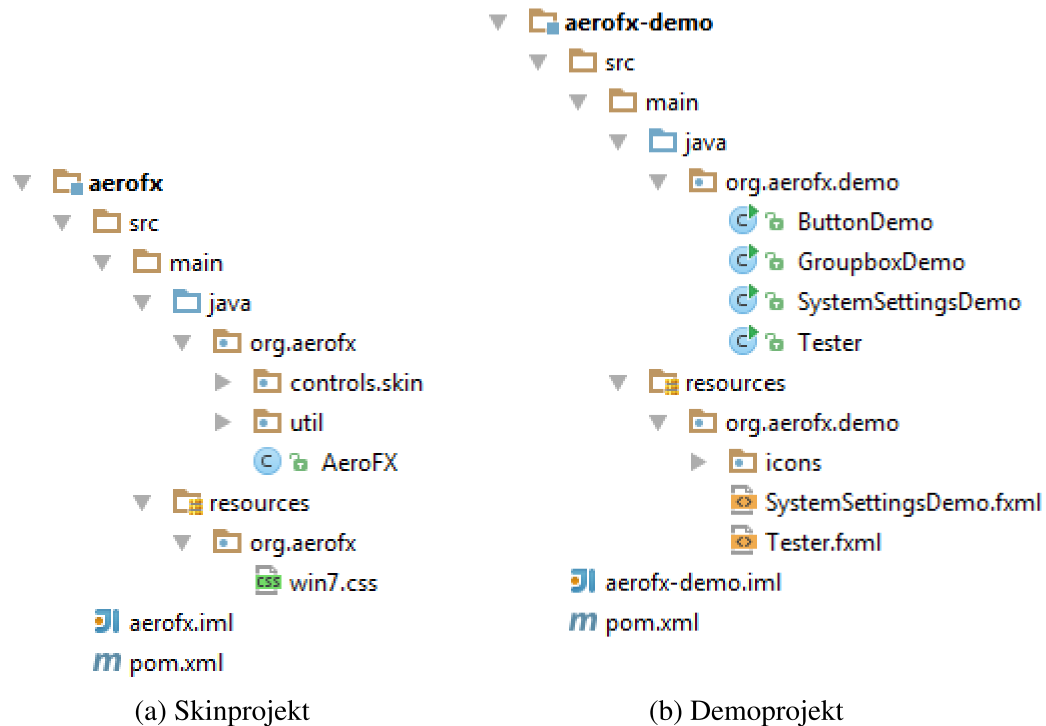


Abbildung 4.3: Wurzelprojekt

Die Vorstellung der Details wird mit der Klasse „AeroFX“ selbst begonnen, welche sich in der Datei AeroFX.java befindet. Sie bietet die Methode `style()` an, die vom zu stylenden Programm aufgerufen wird. Momentan setzt diese Funktion das Stylesheet win7.css als einzige Style-Vorlage für die Anwendung ein.
Im erwähnten Stylesheet wiederum sind alle Definitionen für Steuerelemente hinterlegt, deren Aussehen durch den Skin verändert werden. Da das Stylesheet als einzige Vorlage dient, werden nur Elemente dargestellt, die im Skin definiert sind. Wird also beispielsweise eine Progressbar in der zu stylenden Anwendung genutzt, so taucht diese nach dem Aufruf der Funktion `style()` nicht mehr auf.
Dieser augenscheinliche Mangel ist allerdings beabsichtigt, da auf diese Weise keinerlei Abhängigkeiten zu den von JavaFX zur Verfügung gestellten Skins bestehen. Wird an der Nutzung der Stylesheets nichts geändert, so ist von Anfang an ein Sheet eingebunden: Modena.css. Es existiert die Möglichkeit, über dieses CSS ein weiteres zu legen, durch diesen Vorgang werden jedoch alle Definitionen, die im obersten CSS nicht überschrieben werden, in die Anwendung übernommen.



(a) Skinprojekt (b) Demoprojekt

Abbildung 4.4: Überblick über die AeroFX-Paketstruktur

Erfolgen nun Änderungen an Modena, so könnten diese das gesamte Erscheinungsbild einer Anwendung verändern, die AeroFX verwendet.
Wie bereits im Aufbau vorgestellt, werden alle Skinklassen, die sich im Paket `controls/skin` befinden, über das CSS eingebunden. Diese Java-Klassen stellen Sonderfälle dar, die nur durch das Ableiten gelöst werden können. Alle Klassen implementieren das Interface `AeroSkin`, welches bisher noch keine Implementierungsvorgaben enthält, jedoch bereits für ein zukunftssicheres Design eingefügt wurde. Aktuell wird dieses Interface von vier Klassen genutzt, die Sonderfälle für vier Controls abbilden. Ebendiese Sonderfälle werden im Folgenden näher vorgestellt und erläutert.

4.4.1 AeroCheckBoxSkin

Die Änderungen beziehen sich hier auf das Verhalten bzw. Einfügen eines Fokusrahmens, da die Version, die von JavaFX vorgegeben wird, nicht nur das Label, sondern auch das Control selbst mit einschließt. Da bei Windows der Fokusrahmen nur um das Label gelegt wird, wurde hier ein eigener Rahmen implementiert. Um dies umzusetzen, werden nur ein Rectangle und ein Eventlistener benötigt, wie man in Listing 4.6 erkennen kann.[2] Das Rectangle wird im Konstruktor des Control als Kind des Objektes hinzugefügt, seine Hintergrundfarbe auf transparent gesetzt und die Sichtbarkeit auf false gesetzt, es wird also zunächst einmal nicht angezeigt. Des Weiteren erhält dieses Rectangle eine Style-Klasse, damit der Stil der Umrandung flexibel im CSS geändert werden kann.

```
1 public AeroCheckBoxSkin(CheckBox checkbox) {
2   super(checkbox);
3   focusBorderRect = new Rectangle(0,0, Color.TRANSPARENT);
4   getChildren().add(focusBorderRect);
5   focusBorderRect.setVisible(false);
6   focusBorderRect.getStyleClass()
7     .add("check-box-focus-border");
8   focusBorderListener = (e) ->
        focusBorderRect.setVisible(getSkinnable().isFocused());
9   getSkinnable().focusedProperty()
10    .addListener(focusBorderListener);
11 }
```

Listing 4.6: AeroCheckBoxSkin: Konstruktor

Damit ist das statische Hinzufügen abgeschlossen, jedoch fehlt das Verhalten eines Fokusrahmens: Liegt der Fokus auf dem Element, so soll der Rahmen angezeigt werden. Zu diesem Zweck wird, wie in Zeile 6 bis 8 zu sehen, ein neuer Listener erzeugt und mit dem Rectangle verknüpft. Genauer gesagt wird die Sichtbarkeit des Rectangles durch den Wert der `isFocused()`-Methode gesteuert. Zweifellos stellt sich die Frage, weshalb an dieser Stelle auf einen anonymen Listener verzichtet wurde. Die Antwort darauf liefert ein nicht unwichtiges Detail: Der Listener soll beim Zerstören des Objektes auch wieder von ihm gelöst werden. Hierzu kann die `dispose()`-Methode des Skins überschrieben werden, wie in Listing 4.7 zu sehen.

[2] Da Lambdas bei diesem Snippet das erste Mal genutzt werden, sei an dieser Stelle darauf hingewiesen, dass, wann immer Möglich, Gebrauch von ihnen gemacht wurde. Sie wurden mit Java 8 eingeführt, um das Anlegen von anonymen Klassen zu reduzieren

```
1 @Override
2 public void dispose() {
3   super.dispose();
4   getSkinnable().focusedProperty()
5     .removeListener(focusBorderListener);
6 }
```

Listing 4.7: AeroCheckBoxSkin: Destruktor

Was nun noch fehlt, ist die Anpassung der Größe des Rectangles. Da sich zur Laufzeit die Größe des Labels verändern kann, wird diese Anpassung in der Methode `layoutChildren` vorgenommen. Diese Methode wird jedes Mal aufgerufen, wenn das Element neu gezeichnet wird. In den Aufrufparametern sind Koordinaten sowie Höhe und Breite angegeben. Aus diesen Informationen lassen sich wie in Listing 4.8 alle zur Größenanpassung notwendigen Werte ableiten.

```
1 @Override
2 protected void layoutChildren(double x, double y, double w,
      double h) {
3   super.layoutChildren(x, y, w, h);
4   focusBorderRect.setX(x+17);
5   focusBorderRect.setY(y+1);
6   focusBorderRect.setWidth(w-20);
7   focusBorderRect.setHeight(h-2);
8 }
```

Listing 4.8: AeroCheckBoxSkin: Methode layoutChildren

Wie man sehen kann, wird das Rectangle um 17 Pixel nach rechts und ein Pixel nach unten verschoben, damit es die Control selbst nicht mit einschließt, sondern nur das Label. Die berechnete Breite wird entsprechend angepasst, da diese das Control ebenfalls berücksichtigt.
Die festen Zahlenwerte sind in diesem Fall allerdings als Workaround zu bezeichnen, da auf Dauer entweder eine dynamische Berechnung zur Laufzeit oder die Auslagerung in eine eigens dafür geschaffene Klasse geschehen sollte. Durch diese Veränderungen sichert man auch in Zukunft eine variable und robuste Nutzung des Skins.
Im Anschluss an die theoretische Vorstellung gibt nun noch Abbildung 4.5 einen visuellen Eindruck der Umsetzung, wobei jeweils das Original, welches als Vorbild diente, mit aufgeführt ist. Aus Gründen der Übersichtlichkeit wurde bei der Zusammenstellung der Status die Beschreibung der Control entfernt.

Die Status sind in dieser Reihenfolge aufgeführt: Neutral, Hover, Clicked (jeweils selektiert und deselektiert).

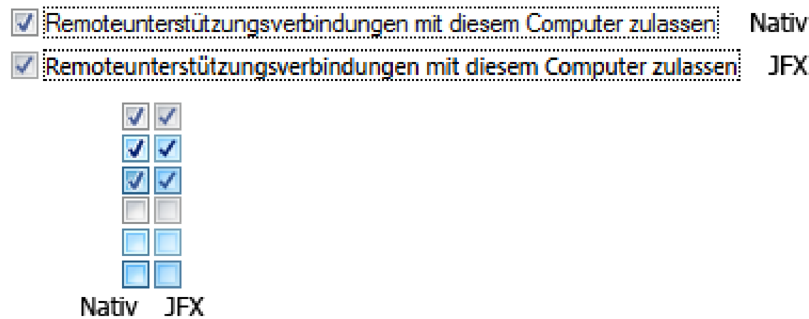


Abbildung 4.5: CheckBox: Windows 7 vs. AeroFX

4.4.2 AeroRadioButtonSkin

Die Erweiterungen in dieser Klasse decken sich größtenteils mit denen der Klasse AeroCheckBoxSkin. Darüber hinaus wurde die Fokusfunktionalität über entsprechende Listener implementiert, die in den Listings 4.9 und 4.10 vorgestellt werden.

```
1 focusTraverseListener = (observable, oldValue, newValue) -> {
2   ((RadioButton)oldValue).setFocusTraversable(false);
3   ((RadioButton)newValue).setFocusTraversable(true);
4 };
5 getSkinnable().getToggleGroup().selectedToggleProperty()
6   .addListener(focusTraverseListener);
```

Listing 4.9: AeroRadioButtonSkin: focusListener

Wie im Listing zu sehen, wird ein RadioButton durch diesen Listener anwählbar gesetzt, sobald er ausgewählt ist. Da in einer Fokusgruppe immer nur ein RadioButton ausgewählt sein kann, sind die restlichen RadioButtons somit nicht anwählbar, es sei denn, einer von ihnen wird selektiert.

```
1 keyListener = event -> {
2   ToggleGroup tg = getSkinnable().getToggleGroup();
3   Toggle sel = tg.getSelectedToggle();
4   int number = tg.getToggles().indexOf(sel);
5 
6   if (event.getCode() == KeyCode.UP) {
7     if(number <= tg.getToggles().size() && number>0) {
8       getSkinnable().getToggleGroup()
9         .selectToggle(tg.getToggles().get(number - 1));
10      ((RadioButton)getSkinnable().getToggleGroup()
11        .getSelectedToggle()).requestFocus();
12    }
13  }
14  else if (event.getCode() == KeyCode.DOWN) {
15    if(number < tg.getToggles().size()-1) {
16      getSkinnable().getToggleGroup()
17        .selectToggle(getSkinnable().getToggleGroup()
18        .getToggles().get(number + 1));
19      ((RadioButton)getSkinnable().getToggleGroup()
20        .getSelectedToggle()).requestFocus();
21    }
22  }
23 };
24 getSkinnable().setOnKeyPressed(keyListener);
```

Listing 4.10: AeroRadioButtonSkin: keyListener

Daraufhin erfolgt das Registrieren der Pfeiltasten zum Weiterschalten innerhalb der Fokusgruppe. Hierzu wird, wie in Zeile 6 und 14 zu sehen, eine entsprechende Abfrage ausgeführt, die den Index des ausgewählten „toggle“ innerhalb der Gruppe inkrementieren und dekrementieren.
Abschließend soll noch auf ein weiteres Detail eingegangen werden: Da die Positionierung der RadioButton-Control an sich nicht mit dem Windows-Pendant übereinstimmte, wurde in der Methode `layoutChildren` noch die Control umgesetzt. Hierzu wurde Gebrauch von der Methode `lookup()` gemacht, welche zu gegebener Styleklasse ein entsprechendes Objekt zurückgibt, sofern vorhanden. Den entsprechenden Befehl zeigt Listing 4.11.

```
1 getSkinnable().lookup(".radio").relocate(0, 3);
```

Listing 4.11: AeroRadioButtonSkin: Ausrichten des Radios

Um auch hier ein Ergebnis zu visualisieren, zeigt Abbildung 4.6 einen Vergleich im Stil des CheckBox-Vergleiches aus Abbildung 4.5.

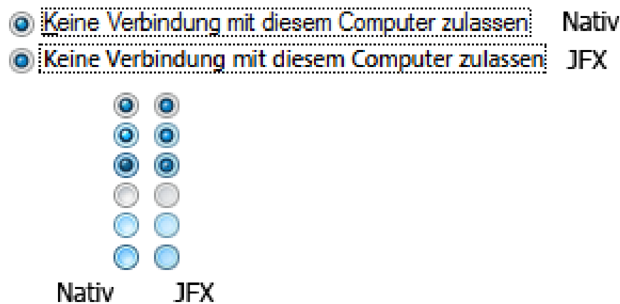


Abbildung 4.6: RadioButton: Windows 7 vs. AeroFX

4.4.3 AeroButtonSkin

Die Klasse ButtonSkin hat in ihrer abgeleiteten Version „AeroButtonSkin“ mit Abstand die meisten Änderungen erhalten. Insbesondere das in Kapitel 2.2.2 erwähnte Pulsieren bei aktivem Fokus erforderte einige Workarounds, auf die in Kapitel 4.5.4 nochmals näher eingegangen wird. Den Einstieg in die Klasse soll das Hinzufügen des Fokusrahmens bieten, der bereits aus den vorangegangenen Kapiteln bekannt ist. Da JavaFX normalerweise nicht die Möglichkeit bietet, diesen Border bei Buttons zu gestalten, wurde auch bei der Klasse AeroButtonSkin ein Rectangle hinzugefügt, welches über die CSS-Klasse `button-focus-border` gestaltet werden kann. Die Methode `layoutChildren` wurde ebenfalls überschrieben, um Veränderungen zur Laufzeit zu berücksichtigen. Die Änderungen sind Listing 4.12 zu entnehmen.
Das Rectangle wird zur Laufzeit 2 Einheiten nach unten und rechts verschoben und seine Höhe und Breite um 4 Einheiten reduziert. Somit erhält man ein Padding von 2 Einheiten an jeder Seite. Auch in dieser Klasse sind alle festen Pixelangaben als Workaround anzusehen, der in einer späteren Version ersetzt wird.

```
1 @Override
2 protected void layoutChildren(double x, double y, double w,
      double h) {
3   super.layoutChildren(x, y, w, h);
4   focusBorderRect.setX(x + 2);
5   focusBorderRect.setY(y + 2);
6   focusBorderRect.setWidth(w - 4);
7   focusBorderRect.setHeight(h - 4);
8 }
```

Listing 4.12: AeroButtonSkin: layoutChildren

Das Padding von 2 Pixeln je Seite ist notwendig, damit der Fokusrahmen nicht auf der Umrandung des Buttons liegt, sondern nach innen versetzt gezeichnet wird.[3] Es wird sich nun nach dem Aussehen der Modellierung des Verhaltens gewidmet. Diese wurde mittels EventListener gelöst. Details hierzu zeigt Listing 4.13. Dabei galt es, insgesamt drei Fälle abzubilden: Das Erlangen des Fokus bei Nutzung der „Tab“-Taste, das Abbrechen der pulsierenden Fokusanimation bei Drücken des Buttons und den Fall, dass ein hover während des focused-Zustandes erfolgt.

```
1 focusTabListener = (observable, oldValue, newValue) -> {
2   focusBorderRect.setVisible(newValue);
3   if(newValue)
4     focusedButtonTransition.play();
5   else
6     resetAnimation();
7 };
8 getSkinnable().focusedProperty().addListener(focusTabListener);
```

Listing 4.13: AeroButtonSkin: FokusListener

Zunächst wird die Sichtbarkeit des Rectangles an den aktuellen Status des Buttons angepasst, da `newValue` die boolesche Variable des `focused`-Status widerspiegelt. Des Weiteren wird abhängig von ebendieser Variablen die Animation gestartet beziehungsweise zurückgesetzt. Die Umsetzung dieser Animation beziehungsweise deren Anhalten wird im Anschluss an die Vorstellung der Logik näher erläutert. Im folgenden Listing 4.14 wird die Umsetzung des `armedListeners` vorgestellt, welcher die Animation anhält, sobald der Button ausgelöst (gedrückt) wird.

[3] Siehe hierzu auch Kapitel 2.2.2

```
armedListener = observable -> {
  if (getSkinnable().isArmed()) {
    resetAnimation();
  } else {
    if (getSkinnable().isFocused()) {
      focusedButtonTransition.play();
    } else if (!getSkinnable().isFocused()){
      resetAnimation();
    }
  }
};
getSkinnable().armedProperty().addListener(armedListener);
```

Listing 4.14: AeroButtonSkin: ArmedListener

Abschließend zeigt Listing 4.15 den `hoverListener`, welcher den Status „hover & focused“ abbildet.

```
hoverListener = observable -> {
  if(getSkinnable().isHover()) {
    resetAnimation();
  } else {
    if(getSkinnable().isFocused()) {
      focusedButtonTransition.jumpToEnd();
      focusedButtonTransition.play();
    } else if (!getSkinnable().isFocused()){
      resetAnimation();
    }
  }
};
getSkinnable().hoverProperty().addListener(hoverListener);
```

Listing 4.15: AeroButtonSkin: HoverListener

Hierbei gilt es, eine weitere Besonderheit zu beachten: Bei einem Hover verändert der Button seine Hintergrundfarbe vom grauen Hintergrund in ein pulsierendes Farbschema, welches zwischen der grauen und blauen Farbpalette wechselt. Die Transition startet bei einem normalen `play()` bei der grauen Variante. In Windows 7 beginnt die Transition jedoch bei der blauen Version, falls sie aus einem `hover` heraus gestartet wird. Für diesen Fall bietet die Transition die Methode `jumpToEnd()`, welche die Transition bei der Zielfarbe beginnen lässt.
Wie genau die Nutzung der Transition funktioniert, wird im Folgenden näher vorgestellt. Im Kern besteht sie aus einer Animation, die durch Aufruf der Methode

`setFocusedButtonAnimation()` konfiguriert wird. Es folgen einige Auszüge aus dieser Funktion, die aufgrund ihrer Länge in Anhang B ausgelagert wurde.

```
1 else{
2   final Duration duration = Duration.millis(1000);
3   focusedButtonTransition = new BindableTransition(duration);
4   focusedButtonTransition.setCycleCount(Timeline.INDEFINITE);
5   focusedButtonTransition.setAutoReverse(true);
6 
7   //starting gradient
8   final Color startColor1 = Color.rgb(242,242,242);
9   final Color startColor2 = Color.rgb(235,235,235);
10  final Color startColor3 = Color.rgb(221,221,221);
11  final Color startColor4 = Color.rgb(207,207,207);
12  //ending gradient
13  final Color endColor1 = Color.rgb(235,246,252);
14  final Color endColor2 = Color.rgb(229,243,251);
15  final Color endColor3 = Color.rgb(203,232,248);
16  final Color endColor4 = Color.rgb(184,221,242);
```

Listing 4.16: Fokusanimation - Startwerte

Das Einrichten des pulsierenden Fokus geschieht im else-Zweig der Methode. Die wichtigsten Instruktionen sind hier das Setzen der Zyklusdauer (Z.2), also wie viel Zeit ein Durchlauf vom Ausganswert zum Zielwert und wieder zurück zum Ausgangswert benötigt. Gefolgt von der Definition, dass die Anzahl der Wiederholungen unendlich ist, ergibt dies das später sichtbare Muster. Der nächste Schritt besteht in der Definition von Ausgangs- und Zielfarbe, hier durch ein graues und ein blaues Farbschema geschehen.

Die Logik, die dafür verantwortlich ist, den Hintergrund fließend vom Start- in den Zielzustand zu überführen, wird in einem Listener definiert. Hierzu bietet die Transition-Klasse ein `fractionProperty` an, welches den aktuellen Stand der Ausführung (0.0 - 1.0) repräsentiert.

```
1 focusedButtonTransition.fractionProperty().addListener(new
      ChangeListener<Number>() {
2   @Override
3   public void changed(ObservableValue<? extends Number>
      observable, Number oldValue, Number newValue) {
4     List<BackgroundFill> list = new ArrayList<>();
5     Stop[] stops = new Stop[] {
6       new Stop(0f, Color.color(
7         (endColor1.getRed() -startColor1.getRed())
8           * newValue.doubleValue()+startColor1.getRed(),
9         (endColor1.getGreen()- startColor1.getGreen())
10          * newValue.doubleValue() + startColor1.getGreen(),
11        (endColor1.getBlue()  - startColor1.getBlue())
12          * newValue.doubleValue() + startColor1.getBlue()
13         )),
14      new Stop(0.49f, Color.color(
15        //Set up second color
16        )),
17      new Stop(0.5f, Color.color(
18        //Set up third color
19        )),
20      new Stop(1f, Color.color(
21        //Set up fourth color
22        )) };
23   //         (...) - Listener continues - (...)
```

Listing 4.17: Fokusanimation - Background-ChangeListener 1/2

Als Vorbereitung wird ein Array von Stops aufgebaut. Diese Stops definieren die Hintergrundfarbe des Buttons, indem sie später in einen Gradienten umgewandelt werden. Wie zu erkennen ist, erfolgt die Berechnung in Abhängigkeit des neuen Float-Wertes `newValue`, welcher den neuen Wert der `fractionProperty` widerspiegelt.[4]

[4]Eine detaillierte Erläuterung, wie genau der Button aufgebaut ist, folgt mit Abbildung 4.7.

```
1   //          (...) - Listener continues - (...)
2           //Build up rectangles
3           BackgroundFill f1 =
4             new BackgroundFill(Color.rgb(60, 127, 177),
5             new CornerRadii(3.0), new Insets(0.0));
6           list.add(f1);
7           BackgroundFill f2 =
8             new BackgroundFill(Color.rgb(72,216,251),
9             new CornerRadii(2.0), new Insets(1.0));
10          list.add(f2);
11          LinearGradient gradient =
12            new LinearGradient(0.0,0.0,0.0,1.0,true,
13            CycleMethod.NO_CYCLE,stops);
14          BackgroundFill bgFill = new BackgroundFill(gradient,
15            new CornerRadii(1.0), new Insets(2.0));
16          list.add(bgFill);
17          ((StyleableProperty<Background>)getSkinnable()
18            .backgroundProperty()).applyStyle(
19              null, new Background(
20                list.get(0), list.get(1), list.get(2)));
21        }
22      });
23    }
24  }
```

Listing 4.18: Fokusanimation - Background-ChangeListener 2/2

In der zweiten Hälfte des Listeners wird der Background des Buttons aufgebaut. Zu diesem Zweck werden zwei Fills und ein Gradient benötigt. Der Gradient wird, wie in Zeile 13 zu sehen, aus dem in 4.17 angelegten Array von Stops gebildet. Um den Hintergrund des Buttons zu setzen, wird die berechnete Farbe inklusive aller weiteren Fills dann in Zeile 17ff. per `applyStyle` gesetzt. Das Ergebnis dieses Listeners beziehungsweise der Aufbau dieses Hintergrundes wird durch Abbildung 4.7 illustriert.
`f1` definiert die äußere Rahmenfarbe des Buttons, daher wird es mit einem Eckenradius von 3 Pixeln und einem Randabstand von 0 Pixeln initialisiert. `f2` repräsentiert hingegen die Farbe des 1 Pixel breiten Innenrandes. Da hier ein Versatz von einem Pixel nach innen vorgesehen ist, wird der Radius mit 2 Pixel Größe entsprechend kleiner gewählt. Die vier Stops setzen die Hintergrundfarbe des Buttons, indem sie für bestimmte Punkte, nämlich 0%, 49%, 50% und 100%, feste Farbwerte vorgeben, aus denen von JavaFX ein entsprechender Übergang zwischen den Werten berechnet wird.

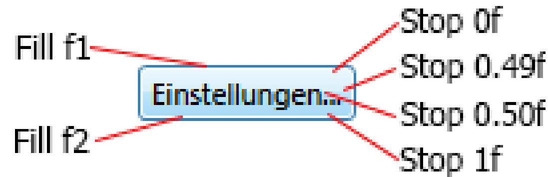


Abbildung 4.7: Aufbau des Buttonhintergrundes

```
1 private void resetAnimation(){
2     focusedButtonTransition.stop();
3     getSkinnable().impl_reapplyCSS();
4 }
```

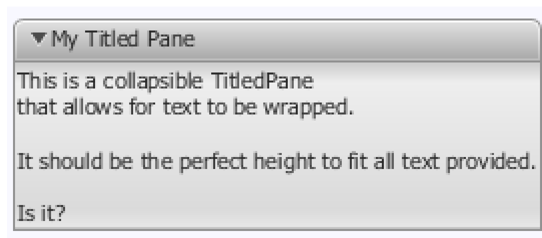
Listing 4.19: AeroButtonSkin: ResetAnimation

Die in Listing 4.19 abgebildete Methode `resetAnimation` weist die Transition an, die Berechnung abzubrechen und wendet eine interne Methode aus dem ButtonSkin an, welche dazu führt, dass das Aussehen des Buttons passend zu seinem aktuellen Status aus der CSS eingelesen und angewendet wird. Hierdurch kehrt der Button auch optisch in den richtigen Zustand zurück.
Das Ergebnis dieser Implementierung wird nachfolgend nochmals in Abbildung 4.8 visuell verglichen, wobei zur besseren Vergleichbarkeit ein Zusammenschnitt der Buttons angeführt wird. Hierbei stammt die linke Hälfte aus dem Screenshot eines nativen Buttons, die rechte Hälfte aus dem eines JavaFX-Buttons.

Abbildung 4.8: Vergleich der Buttons: Windows 7 und AeroFX

4.4.4 AeroGroupBoxSkin

Das in Kapitel 2.2.1 vorgestellte Gruppierungselement GroupBox existiert nicht nativ in JavaFX. Aus diesem Grund wurde es in der Klasse `AeroGroupBoxSkin` nachgebaut. Als Grundlage zum Vorgehen hierzu diente ein Kapitel aus dem Buch zu JavaFX-Controls von Hendrik Ebbers (vgl. [Ebb14, S. 259 ff.]).
Diese Klasse erbt von `SkinBase<TitledPane>`, der Basis eines JavaFX-TitledPanes, welches gewisse Ähnlichkeiten mit dem erklärten Ziel aufweist, wie ein Vergleich von Abbildung 4.9 und 4.10 zeigt.



(Entnommen aus [Ora13a])

Abbildung 4.9: JavaFX TitledPane (Modena)

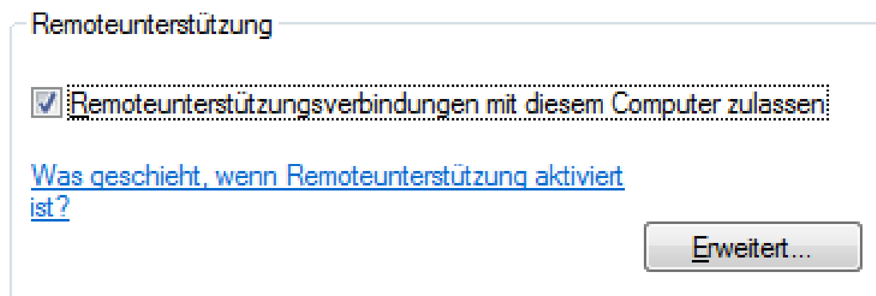


Abbildung 4.10: Windows 7 Groupbox

Auch das TitlePane ist ein Stilelement, welches weitere Elemente oder auch Text enthalten kann. Des Weiteren besitzt es bereits ein Titel-Property. Die AeroFX-Version dieser Klasse weist jedoch einige Unterschiede auf. Aufgrund der besseren Kontrolle wurde der Titel der Groupbox als Label realisiert, dessen Text-Property an den Titel des TitledPane gebunden wird (Listing 4.20, Z. 4). Hierdurch lässt sich der Titel frei bewegen, kann aber trotzdem kompatibel zur JavaFX-Spezifikation bleiben.

```
1  public AeroGroupBoxSkin(TitledPane p) {
2      super(p);
3      titleLabel = new Label("");
4      titleLabel.textProperty().bind(p.textProperty());
5      getChildren().add(titleLabel);
6      captionBg = new Rectangle();
7      p.setCollapsible(false);
8      captionBg.setStyle("-fx-fill:transparent;");
9      groupBoxBg = new Rectangle();
10     groupBoxBg.setStyle("-fx-fill:transparent;");
11     clippingRect = new Rectangle();
12     getChildren().add(groupBoxBg);
13     groupBoxBg.getStyleClass().add("group-box-border");
14     if (p.getContent() != null)
15         getChildren().add(p.getContent());
16     p.setPadding(new Insets(7, 0, 0, 0));
17 }
```

Listing 4.20: AeroGroupBoxSkin: Konstruktor

Gesondert hinzuweisen ist auf Zeile 13, in der dem Rectangle `groupBoxBg` die Styleklasse `group-box-border` hinzugefügt wird. Durch Ansprechen dieser Klasse im CSS kann später das Aussehen des Rahmens der Groupbox beeinflusst werden. Wie man sieht, werden im Konstruktor der Klasse noch einige weitere Objekte angelegt. Diese werden für einen Kunstgriff benötigt, der den Rahmen hinter dem Titel der Groupbox verschwinden lässt. Mit Hilfe von Abbildung 4.11 wird dieser im Folgenden vorgestellt.

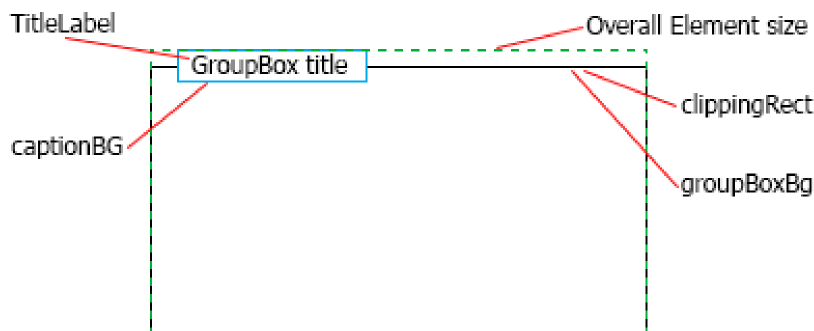


Abbildung 4.11: Layout AeroGroupBoxSkin

Die Gesamtgröße des Elements wird durch die grün gestrichelte Linie dargestellt. Das `titleLabel` liegt als unsichtbares Viereck um den Text und passt sich dessen Breite an. Das `clippingRect` besitzt dieselbe Höhe und Breite wie `groupBoxBg`, welches wiederum den eigentlichen Rahmen anzeigt. Damit nun der angezeigte Rahmen der `groupBoxBg` nicht durch den Text durchgezeichnet wird, macht man sich eine Fähigkeit von JavaFX zunutze, nämlich das „clipping“. Hierbei kann JavaFX mitgeteilt werden, dass bestimmte Bereiche im angezeigten Bildbereich nicht überlagert beziehungsweise gezeichnet werden sollen. Am besten erklärt werden kann das clipping mit der Metapher, dass man ein Blatt Papier mit einer ausgeschnittenen Form auf ein Foto legt. Analog dazu kann in JavaFX eine Form über Elemente gelegt werden, die den Bereich bestimmt, der dargestellt werden soll. Abbildung 4.12 vermittelt nochmals einen grafischen Eindruck hiervon, wobei in diesem Beispiel eine Form über einen Schriftzug gelegt wurde.

(Entnommen aus [Dea14, S. 148])

Abbildung 4.12: Einfluss von Clipping auf ein Objekt

Um die Theorie nun mit etwas Praxis anzureichern, folgt in Listing 4.21 die Vorstellung der Funktion `layoutChildren`, welche für solche Veränderungen überschrieben werden muss.

```
1 @Override
2 protected void layoutChildren(double x, double y, double w,
      double h) {
3     super.layoutChildren(x, y, w, h);
4     titleLabel.relocate(x + 9, y +1);
5     captionBg.relocate(7, -7);
6     captionBg.setWidth(titleLabel.getWidth()+4);
7     captionBg.setHeight(titleLabel.getHeight());
8     groupBoxBg.relocate(x, y + 7);
9     groupBoxBg.setWidth(w);
10    groupBoxBg.setHeight(h - 7);
11    clippingRect.relocate(0, 0);
12    clippingRect.setWidth(groupBoxBg.getWidth());
13    clippingRect.setHeight(groupBoxBg.getHeight());
14    groupBoxBg.setClip(Rectangle.subtract(clippingRect,
       captionBg));
15    if (getSkinnable().getContent() != null) {
16        getSkinnable().getContent().relocate(x, y);
17        getSkinnable().getContent().resize(w, h);
18    }
19 }
```

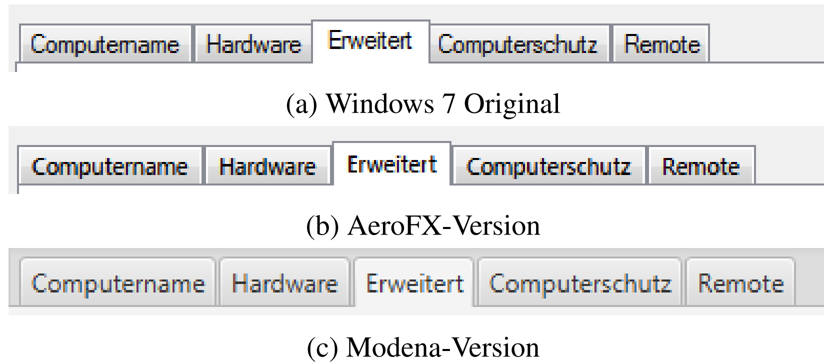
Listing 4.21: AeroGroupBoxSkin: Layout-Methode

Zunächst wird das `titleLabel` auf seine endgültige Position etwas abseits vom Rand geschoben. Im Anschluss wird das `captionBG` ebenfalls neu ausgerichtet. Durch diese Ausrichtung um 7 Pixel nach unten (Z. 5) wird der Anschein erweckt, der Titel befinde sich außerhalb des Elements. Durch Anpassung des `clippingRect` hierauf werden alle Vorbereitungen für die entscheidende Anweisung in Zeile 14 getroffen, welche das eigentliche Clipping setzt. Im Anschluss wird eventuell vorhandener Content (Buttons, Text, usw.), welcher in der GroupBox dargestellt werden soll, ebenfalls ausgerichtet und angepasst.

4.4.5 TabPane

Im Gegensatz zu allen bisher vorgestellten Elementen existiert für das TabPane keine eigene Java-Klasse, wie auch dem Titel zu entnehmen ist. Es erbringt den Beweis, dass für erfolgreiches Styling nicht zwangsweise von den Skinklassen abgeleitet werden muss, obwohl komplexe Verhaltensmuster abgebildet werden. Hierzu gehört unter anderem, dass der selektierte Tab größer und ohne Rahmen zum Inhalt der Registerkarte dargestellt wird.

Abbildung 4.13 stellt den direkten Vergleich zwischen dem Original, der AeroFX-Version und Modena dar.



(a) Windows 7 Original

(b) AeroFX-Version

(c) Modena-Version

Abbildung 4.13: Vergleich der TabPane-Skins

Eben erwähnte Veränderungen werden über den Parameter `-fx-pref-height` gesteuert. Dieser wird verändert, sobald ein Tab vom Zustand `top` in den Zustand `selected` übergeht, wie Listing 4.22 zu entnehmen ist.
Weiterhin ist gut zu sehen, dass durch die Anpassung der `-fx-border-width` die Rahmendicke für die untere Seite eines Tabs auf Null gesetzt wird, sobald ein Tab den Zustand `selected` annimmt. Hierdurch wird beim selektierten Tab die Abgrenzung zum Inhalt entfernt. Die komplette Definition aller Parameter ist nötig, um Inkonsistenzen beim Zustandswechsel zu vermeiden. Da abgesehen von den Definitionen aus dem AeroFX Stylesheet keine weiteren Stylinginformationen verfügbar sind,
Um das Verhalten des Originals so genau wie möglich zu implementieren, wurde auch beim TabPane ein Hover-Effekt implementiert, der dargestellt wird, sobald sich die Maus auf einem Tab befindet. Hierzu wurde ein entsprechender Selektor für die Pseudoklasse `.tab:hover` geschrieben, der den Hintergrund blau färbt, die Rahmenfarbe anpasst und einen ein Pixel breiten, weißen Rand einfügt.
Die hier gezeigten CSS-Regeln stellen jedoch nur einen Teil der Styling-Vorgaben dar, da die Vorstellung aller Regeln an dieser Stelle den Rahmen sprengen würden. Bei Interesse können die vollständigen Definitionen der angefügten Quellcode-DVD entnommen werden.

```
1 .tab-pane > .tab-header-area > .headers-region > .tab:top{
2     -fx-border-color: rgb(244,0,0), rgb(137,140,149),
       rgb(137,140,149), rgb(137,140,149);
3     -fx-border-width: 1 1 1 1;
4     -fx-pref-height: 1.8em;
5 }
6 .tab-pane > .tab-header-area >
7   .headers-region > .tab:selected{
8     -fx-border-color: rgb(137,140,149);
9     -fx-border-width: 1 1 0 1;
10    -fx-background-color: rgb(255,255,255);
11    -fx-pref-height: 2em;
12 }
13 .tab-pane > .tab-header-area > .headers-region > .tab:hover{
14    -fx-border-color: rgb(60,127,177);
15    -fx-background-color:
16        rgb(255,255,255),
17        linear-gradient(to bottom, rgb(234,246,253) 0%,
       rgb(217,240,252) 49%, rgb(190,230,253) 50%,
       rgb(167,217,245) 100%);
18    -fx-background-insets: 0,2 2 0 2;
19 }
```

Listing 4.22: Styling der Klasse .tab-pane

4.4.6 Mnemonics

Die Mnemonics können über eine einfache Anweisung innerhalb des CSS gestylt werden. Hierzu kann die Selektorklasse `.mnemonics` genutzt werden, wie Listing 4.23 demonstriert.

```
1 .mnemonic-underline {
2     -fx-stroke: transparent;
3 }
4
5 (...)
6
7 .radio-button:show-mnemonics > .mnemonic-underline{
8     -fx-stroke: -fx-text-base-color;
9 }
```

Listing 4.23: Mnemonic-Styling im CSS

Zunächst wird definiert, dass der Mnemonic transparent ist. Diese Definition wird durch die Pseudoklasse `show-mnemonics` des jeweiligen Elementes überschrieben, welche `-fx-text-base-color` als Farbe setzt. Diese Pseudoklasse feuert, sobald der Nutzer die Taste „Alt“ drückt.
Wichtig ist hierbei, dass die Auswahl, welcher Buchstabe unterstrichen wird, nicht automatisch, sondern durch den Entwickler erfolgt. Sie wird in der FXML-Datei festgelegt, die die Oberfläche beschreibt. Hierbei ist vom Entwickler auf die passende Internationalisierung zu achten, da in anderen Sprachen andere Tastenkombinationen genutzt werden.

4.5 AeroFX - Die Probleme

Es folgt eine Vorstellung der Probleme, die bei der Entwicklung des Skins auftraten.

4.5.1 Textabstand

Beim Design des Dialogs „Systemeigenschaften“ fiel auf, dass Optionen, die das Textrendering betreffen, in JavaFX nicht verändert werden können. Die Auswirkungen sind auch bei den direkten Vergleichen im Kapitel 4.4 zu erkennen, da die Beschriftungen der Elemente nie ganz deckungsgleich sind.
Textglättung, Schriftart und -größe sind die einzigen Parameter, die durch CSS-Definitionen beeinflusst werden können. Insbesondere fehlende, den Zeilenabstand betreffende Justiermöglichkeiten mindern die Qualität des Skins, da offensichtliche Unterschiede bei Labels mit längerem Text unvermeidbar sind. Abbildung 4.14 zeigt den Unterschied anhand eines Labels. Es ist gut zu erkennen, dass sowohl der Zeilenabstand als auch der Abstand der Buchstaben untereinander in JavaFX anders ist als in Windows 7. Hinzu kommt fehlendes Hinting, also das Ausrichten der einzelnen Buchstaben an den Pixelkanten des Bildschirms.[5]
JavaFX bietet zwar eine CSS-Option „-fx-font-smoothing“, aber hierüber ist nur eine marginale Beeinflussung des Font-Renderings möglich. Explizite Einstellungen zur Verwendung von Hinting, Subpixel-Rendering oder Font-Smoothing sind derzeit nicht möglich.

[5] Die verwendete Schriftart besitzt Hinting-Informationen, diese werden lediglich nicht von JavaFX umgesetzt, obwohl JavaFX laut Oracle den nativen Font-Renderer nutzt.

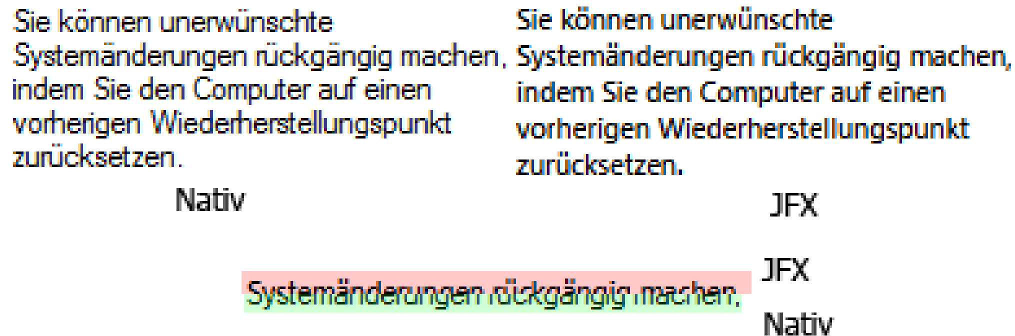


Abbildung 4.14: Textabstand im Vergleich

Diesbezüglich wurde im Bugtracker des GitHub-Projekts bereits eine Fehlerbeschreibung hinzugefügt, die durch die Eröffnung eines Bugreports im offiziellen Bugtracker von JavaFX ergänzt wurde,[6] um Oracle die Möglichkeit der Fehlerbehebung und Erweiterung von JavaFX zu geben.

4.5.2 StageStyle

Betrachtet man die Bildschirmabzüge des in JavaFX nachgebauten Dialogs, so fällt auf, dass die „Window Decoration“, also die eingeblendeten Buttons und Symbole, nicht mit dem originalen Dialog übereinstimmen. In der JavaFX-Version existieren Buttons, um das Fenster zu maxi- und minimieren, des Weiteren wird links vom Titel ein Anwendungssymbol, auch Icon genannt, eingeblendet. Entwickler können das Layout des darstellenden Fensters indirekt durch das Setzen eines `StageStyle` beeinflussen.
Dieser `StageStyle` ist ein Enum, welches verschiedene Modi bietet, nämlich `DECORATED`, `TRANSPARENT`, `UNDECORATED`, `UNIFIED` und `UTILITY`.
`DECORATED` ist das Standardlayout eines Fensters, also mit allen Dekorationen (decorations), die das Betriebssystem zur Verfügung stellt. `TRANSPARENT` wiederum stellt das Fenster ohne jegliche Dekorationen mit transparentem Hintergrund dar, was sich für Vollbild-Anwendungen und OSD-ähnliche („On Screen Display“) Projekte eignet. `UNDECORATED` stellt das Fenster ebenfalls ohne Dekorationen dar, jedoch mit weißem, undurchsichtigem Hintergrund.

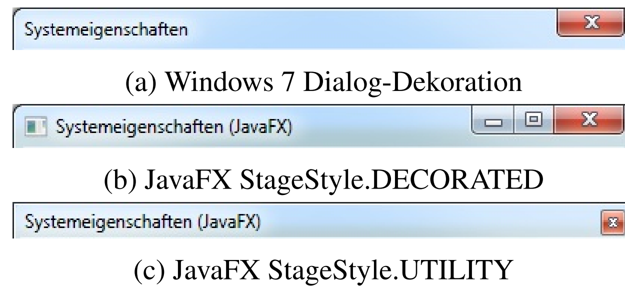
[6]https://javafx-jira.kenai.com/browse/RT-38059

Im Gegensatz dazu wird bei Wahl des `UNIFIED`-Stils das Fenster mit allen Dekorationen dargestellt, jedoch ohne Trennung von Inhalt und Rahmen, wie er sonst üblich ist. Als letzte Möglichkeit ist `UTILITY` wählbar, welches minimale Plattformdekorationen bietet, wie sie bei Tool-Dialogen üblich sind.
Bevor im Detail darauf eingegangen wird, wie die einzelnen Modi graphisch aussehen, zeigt Listing 4.24 zunächst das Einbinden des `StageStyle` in eine JavaFX-Anwendung.

```
1 public class UndecoratorStageDemo extends Application {
2
3 @Override
4 public void start(final Stage stage) throws Exception {
5  Pane root = new Pane();
6  Scene scene = new Scene(root);
7  stage.setScene(scene);
8  stage.initStyle(StageStyle.UTILITY);   //Set style to UTILITY
9  stage.show();
10 }
11
12 public static void main(String[] args) {
13  launch(args);
14 }
```

Listing 4.24: Setzen eines StageStyle

Aufgrund der Dokumentationslage würde sich die Nutzung von `UTILITY` anbieten, da hierdurch das Fenster wie gewünscht dargestellt wird, nämlich nur mit Schließen-Button, ohne Maximieren, Minimieren oder Anwendungsicon.
Setzt man den `StageStyle` jedoch, so ist gut zu erkennen, dass die erzeugte Dekoration nicht mit dem gewünschten Aussehen übereinstimmt.
Vergleicht man die Windows 7 Dialog-Dekoration (Abb. 4.15a) mit dem Standard in JavaFX (Abb. 4.15b), so fallen die Unterschiede sofort auf.
Es bliebe also noch die Wahl der `UTILITY`-Dekoration aus JavaFX (Abb. 4.15c), die allerdings die Dekoration des Tool-Dialogs repräsentiert.



(a) Windows 7 Dialog-Dekoration

(b) JavaFX StageStyle.DECORATED

(c) JavaFX StageStyle.UTILITY

Abbildung 4.15: Fensterdekorationen im Vergleich

Da die Dokumentation der Dekorationen in diesem Punkt missverständlich ist, wurde auch zu diesem Problem ein Ticket im Oracle Bugtracker eröffnet, welches die fehlerhafte Umsetzung der `UTILITY`-Dekoration zum Inhalt hatte. Dieses wurde allerdings mit dem Hinweis „Not an Issue“ geschlossen, da die Dekoration mit Absicht so umgesetzt wurde:

> Unfortunately, I'll be closing this issue as Not an Issue because this is not a bug. On Windows, JavaFX specifies the WS_EX_TOOLWINDOW native window extended style for UTILITY windows. The windows then look however the native OS renders them on the screen. This behavior is by design and is not going to change. (...)“
> (vgl. [Orab])

Es besteht trotzdem das Problem, dass JavaFX nicht alle Möglichkeiten zur Fensterdekoration abbildet, die bei Windows möglich sind.
Als Reaktion wurde in Absprache mit den Oracle-Entwicklern ein bestehendes Ticket zur Fensterdekoration erweitert. Es wird nun erörtert, ob dem Entwickler mehr Kontrolle über die Dekoration gegeben werden soll.
Hierzu kann als Beispiel genannt werden, die Anzeige der verschiedenen Steuerelemente durch eine entsprechende Kontrollstruktur zugänglich zu machen.

4.5.3 Fokusreihenfolge

Bei der Implementierung des AeroGroupBoxSkins zeigte sich ein Problem mit den Fokusrahmen, welches zwangsweise behoben werden musste. Es fiel auf, dass der Fokus nicht in eine GroupBox zurückkehrt, sobald er sie einmal verlassen hat. Das heißt, dass der JavaFX-Algorithmus, welcher das nächste Ziel für den Fokus

auswählt, zwar eine AeroGroupBox als Ziel auswählt, aber nicht die Elemente, die in ihr enthalten sind.
Durch eine Analyse der Implementierung dieses Algorithmus wurde ein Workaround entwickelt, der den Fokus in einem solchen Fall an die Elemente einer AeroGroupBox weiterreicht. Listing 4.25 zeigt diesen Workaround, welcher als Eventlistener in der Klasse `AeroGroupBoxSkin` umgesetzt wurde.

```
1 focusListener = new InvalidationListener() {
2   @Override
3   public void invalidated(Observable observable) {
4     if (p.isFocused()) {
5         Node content = p.getContent();
6       if (content != null && content instanceof Parent) {
7         if (((Parent) content).
8           getChildrenUnmodifiable().size() > 0)
9             ((Parent) content).
10              getChildrenUnmodifiable().get(0).requestFocus();
11       }
12     }
13   }
14 };
15 p.focusedProperty().addListener(focusListener);
```

Listing 4.25: Listener zur Fokusweiterleitung

Der entscheidende Schritt ist die Abfrage des Inhaltes der AeroGroupBox, welcher in Zeile 5 durchgeführt wird. Intern befindet sich als erstes Kind in der AeroGroupBox noch ein Pane, welches alle weiteren Elemente enthält. Enthält dieses Pane Kindelemente, so wird die Methode `requestFocus` des ersten Elementes aufgerufen, wodurch es den Fokus erhält. Ab diesem Punkt funktioniert der Algorithmus zur Bestimmung des nächsten Elementes wieder wie gewünscht.

4.5.4 Fokusblinken

Die Implementierung des pulsierenden Hintergrundes in JavaFX erforderte mehr Arbeit als ursprünglich geplant. Es zeigte sich ein Phänomen, welches nicht ganz leicht zu erklären, jedoch, wenn bekannt, logisch nachzuvollziehen ist.
Die Entwicklung von AeroFX erfolgte zunächst durch Hinzufügen des CSS zu Modena. Durch diese Art der Entwicklung werden alle Elemente in JavaFX angezeigt, selbst wenn sie noch nicht in AeroFX implementiert sind. Hierdurch ist ein schnelles Prototyping und eine direkte Rückmeldung möglich, da ein Element nicht erst in allen Status modelliert werden muss, um angezeigt zu werden.

Im Rahmen dieses Status wurde auch das Pulsieren des Buttons implementiert und erfolgreich getestet.
Die Einbindung des Skins auf diese Art und Weise hat allerdings einen großen Nachteil: Da viele Definitionen von Modena abhängen, ist der Skin auch allen Änderungen unterworfen, die in Modena durch Oracle vorgenommen werden. Um diese Abhängigkeiten zu eliminieren, wurde der Skin auf Standalone-Betrieb umgestellt.
In seiner jetzigen Version wird Modena beim Aufruf von `AeroFX.style()` vollständig entfernt und durch das CSS von AeroFX ersetzt.
Jedoch funktioniert die Rückkehr des Buttons aus dem Fokus-Zustand in der ursprünglichen Version nicht mehr, wenn das AeroFX-CSS als einzige Stilvorlage in ein Programm eingebunden ist. Weder hover- noch armed-Effekte werden dargestellt, sobald sich der Button einmal im focused-Zustand befunden hat. Hierzu konnte eine schnelle Nachfrage beim Entwickler von Oracle Abhilfe schaffen, die durch Hendrik Ebbers ermöglicht wurde.
Eben erwähnte Nachfrage zeigte, dass dieses Verhalten von den Entwicklern so beabsichtigt ist. In dem Moment, in dem Eigenschaften eines Elements, die im CSS definiert sind, durch eine Java-Klasse überschrieben werden, kann davon ausgegangen werden, dass der Programmierer diese Handlung mit voller Absicht ausführt. Um diese Änderungen zu erhalten, werden ab diesem Moment keinerlei Änderungen aus dem CSS auf das veränderte Element angewendet.
Der Entwickler hat das Ziel, für JavaFX 9 diese Transitions direkt im CSS zu realisieren, sodass solche Klassen, wie sie in dieser Arbeit erstellt wurden, in Zukunft in vielen Aspekten überflüssig sein werden.
Mit diesem Wissen konnte der Workaround, der vom Entwickler vorgeschlagen wurde, auch erfolgreich umgesetzt werden. Kern des Workarounds ist das Ersetzen des eigentlichen Aufrufes zum Setzen des Hintergrundes. Listing 4.26 gibt hierzu den Vorher-Nachher-Vergleich.

```
1 //Vorher:
2 getSkinnable().setBackground(new Background(list.get(0),
        list.get(1), list.get(2)));
3
4 //Nachher:
5 ((StyleableProperty<Background>)getSkinnable()
6   .backgroundProperty()).applyStyle(null, new Background(
7     list.get(0), list.get(1), list.get(2)));
```

Listing 4.26: Setzen des Background Vorher/Nachher

In der ursprünglichen Version wurde der Hintergrund „hart“ durch den Aufruf der `setBackground()`-Funktion gesetzt. Hierbei „erkennt“ JavaFX, dass der Hintergrund durch Java-Logik verändert wurde. Im Workaround wird hingegen die `backgroundProperty` des Buttons durch die Funktion `applyStyle()` verändert, was der Vorgehensweise von JavaFX beim Anwenden des CSS auf Elemente entspricht. JavaFX „erkennt“ hier also keinen Unterschied zu seinem normalen Vorgehen, daher werden auf diese Art veränderte Elemente auch weiterhin durch das CSS verändert.
Was zum korrekten Verhalten des Buttons noch fehlt, ist das Zurücksetzen in den Normalzustand, sobald der Fokus nicht mehr auf dem aktuellen Element liegt. In der ursprünglichen Version wurde lediglich ein `Transition.stop()` aufgerufen, wodurch lediglich das Pulsieren angehalten wurde. Alle Aufrufe dieser Art wurden für den Workaround durch den Aufruf von `resetAnimation()` ersetzt, welcher Listing 4.27 zu entnehmen ist.

```
1 private void resetAnimation(){
2   focusedButtonTransition.stop();
3   getSkinnable().impl_reapplyCSS();
4 }
```

Listing 4.27: Zurücksetzen der Animation

Es lässt sich gut erkennen, dass auch innerhalb der Methode als Erstes das Pulsieren angehalten wird. Zusätzlich erfolgt der Aufruf von `impl_reapplyCSS()`, welcher veranlasst, dass der zum Status des Elements passende Satz an CSS-Anweisungen neu auf das Element angewendet wird. Hierdurch wird erreicht, dass der Button auch optisch wieder den ursprünglichen Zustand annimmt. Der Aufruf von `impl_reapplyCSS()` ist jedoch nur ein vorläufiger Workaround, da er als deprecated markiert ist. Hierzu muss in Zusammenarbeit mit den Entwicklern noch eine praktikablere Lösung gefunden werden.
Es bleibt abzuwarten, wie sich das noch junge JavaFX weiterentwickelt, auch und vor allem durch die zahlreichen Bugs und Feature Requests, die nach und nach durch die Community bei Oracle eingereicht werden. Auch die Entwicklung von AeroFX hängt, wie in diesem Kapitel vorgestellt, von einigen dieser Bugreports ab. Da hier ein Handeln seitens Oracle vonnöten ist, kann an dieser Stelle nichts weiter hinzugefügt werden. Daher folgt nun ein Überblick über die Art und Weise, wie das Projekt nach außen hin aufgebaut ist.

Kapitel 5

Validierung

In diesem Kapitel werden die Ergebnisse aus Kapitel 3 aufgegriffen und ein Vergleich mit AeroFX durchgeführt, um es in den Kontext der Marktsituation einzuordnen. Des Weiteren zeigt eine genaue Bestandsaufnahme die Stärken und Schwächen des Projektes gegenüber seinen Mitbewerbern.

5.1 Lizenz

Durch die Veröffentlichung unter 2-Clause BSD-Lizenz ist das Projekt einerseits Open Source, andererseits aber auch gut als Bestandteil in proprietärer Software benutzbar, ohne dass langwierige Lizenzverhandlungen zwischen Urheber und Nutzer stattfinden müssen.
Im Vergleich zu anderen Projekten zeigt sich, dass die klare Lizenzlage ein Vorteil ist, der AeroFX auf eine Augenhöhe mit AquaFX (2-Clause BSD) und Oracles Modena (GPLv3) stellt. Im Gegensatz dazu sind die Projekte JMetro und JavaFX-Native-Themes die klaren Verlierer, da bei diesen Projekten die Lizenzfrage zum Zeitpunkt der Erstellung dieser Arbeit nicht klar geregelt ist.

5.2 Dokumentation

Die Dokumentation des Projektes erfolgte vollständig in Javadoc. Javadoc ist ein Dokumentationsstandard, der von vielen Projekten genutzt wird. Die Dokumentation wurde aufgrund des engen Zeitrahmens bisher nur in einer ersten Version fertiggestellt, beinhaltet jedoch bereits alle Klassen inklusive einer Beschreibung

der nach außen hin angebotenen Funktionen. Sie bildet die Grundlage, auf der andere Entwickler bei Interesse ihre Entwicklungen aufbauen können. Da AeroFX mittels Maven gebaut wird, wurde Maven so konfiguriert, dass beim Packaging ebenfalls ein `jar` gebaut wird, welches die komplette Dokumentation enthält.
In dieser Kategorie hebt sich AquaFX durch umfangreiche Javadoc ab, gefolgt von Modena, welches zumindest durch Inline-Kommentare dokumentiert wurde. Sowohl JavaFX-Native-Themes als auch AeroFX können noch als rudimentär dokumentiert gelten, da durch Commit-Kommentare in Bitbucket bis zu einem gewissen Grad nachvollziehbar ist, weshalb gewisse Workarounds oder Designentscheidungen umgesetzt wurden. Das Schlusslicht bildet JMetro, welches kaum dokumentiert ist und keinerlei Commit-Kommentare bietet, da die Softwareverteilung als Quellcode in einer Dropbox-Freigabe erfolgt, die solche Kommentare nicht erlaubt.

5.3 Aufbau

Der Aufbau wurde großteils von AquaFX übernommen, daher handelt es sich ebenfalls um einen Enterprise-tauglichen Ansatz, der durch den Einsatz eines Fassaden-Entwurfsmusters einen konsistenten und änderungssicheren Zugriff auf alle Funktionen des Skins gewährt.
Des Weiteren ergeben sich Vorteile durch den Aufbau als Maven-Projekt, da hierdurch das Dependency-Management in Java-Projekten erheblich vereinfacht wird. Mit anderen Worten kann über einen einheitlichen Konfigurationsansatz ein reproduzierbarer Softwarezustand für Projekte garantiert werden, der insbesondere in größeren Umgebungen mit mehreren Entwicklern eine Grundvoraussetzung ist.
Da AquaFX als Vorlage diente, befinden sich AquaFX und AeroFX auf Augenhöhe und entscheiden diese Kategorie für sich. Allerdings sind auch diese hier dicht gefolgt von Modena, welches auf dem zweiten Platz landet, da kein änderungssicherer Zugriff bei besagtem CSS möglich ist. Daher kann es passieren, dass sich durch Updates im CSS auch die Anwendungen verändern, die auf das CSS aufbauen. Mit diesem Problem hat auch JavaFX-Native-Themes zu kämpfen, zuzüglich zur bisher mangelhaften Umsetzung der Features, wodurch es hinter Modena einzuordnen ist. Erneut wurde JMetro auf dem letzten Platz eingeordnet, da sich Java-Klassen, welche noch zusätzlich zum CSS vorhanden sind, nicht kompilieren lassen.

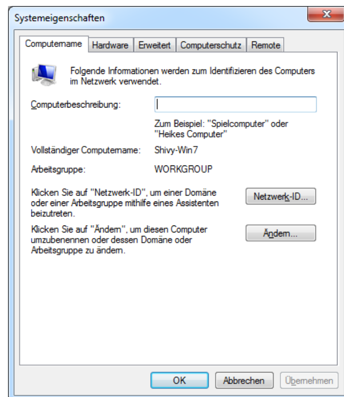
5.4 Verteilung und Updates

Auch hier zahlt sich die Wahl des AquaFX-Ansatzes aus: Der potenzielle Anwender hat die Wahl, da er entweder den aktuellsten Entwicklungsstand aus dem GitHub-Repository klonen oder eine fertige Library von Maven Central einbinden kann[1].
Durch Einsatz von Git und Maven ist eine größtmögliche Verteilung bei gleichzeitig komfortabler Versionsverwaltung möglich, die beispielsweise so einfach nicht durch Dropbox erreicht werden kann. Allerdings muss der Entwickler, der AeroFX einsetzt, trotzdem das gesamte Projekt neu kompilieren beziehungsweise die aktualisierte Library noch an alle Systeme verteilen, die seine Software einsetzen.
Wie auch beim Aufbau stehen AquaFX und AeroFX auf gleicher Höhe, diesmal jedoch angeführt von Modena, da dieses als Teil der JRE automatisiert verteilt und geupdatet wird, sobald eine neue stabile Version seitens Oracle freigegeben wird. Diesen dreien folgt JavaFX-Native-Themes, weil vor dem Neukompilieren der Software noch eventuelle Anpassungen aufgrund der fehlenden Fassade gemacht werden müssen, und JMetro, da dort die Aktualisierung vollständig händisch durchgeführt werden muss.
Alternativ ließe sich die Dropbox-Freigabe noch zu einem Dropbox-Konto hinzufügen, jedoch entsteht so die Gefahr, dass eigene Änderungen an den Klassen bei einem (grundsätzlich unangekündigten) Update durch den Entwickler verloren gehen.

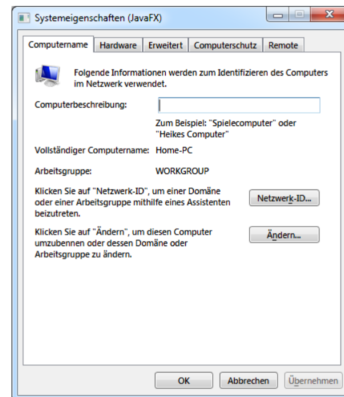
[1] vgl. http://search.maven.org/#search%7Cga%7C1%7Corg.aerofx

5.5 Windows 7 Look & Feel

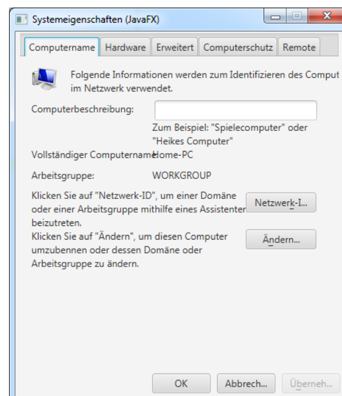
Der Windows 7 Look & Feel wurde eingangs in Kapitel 2.2 vorgestellt, zusammen mit einigen Kernkomponenten, die bei der täglichen Benutzung immer wieder Einsatz finden. Der erzeugte Look & Feel von AeroFX wurde im Detail in Kapitel 4.4 vorgestellt, weshalb hier nur eine Zusammenfassung erfolgt.



(a) Windows 7 nativ



(b) AeroFX

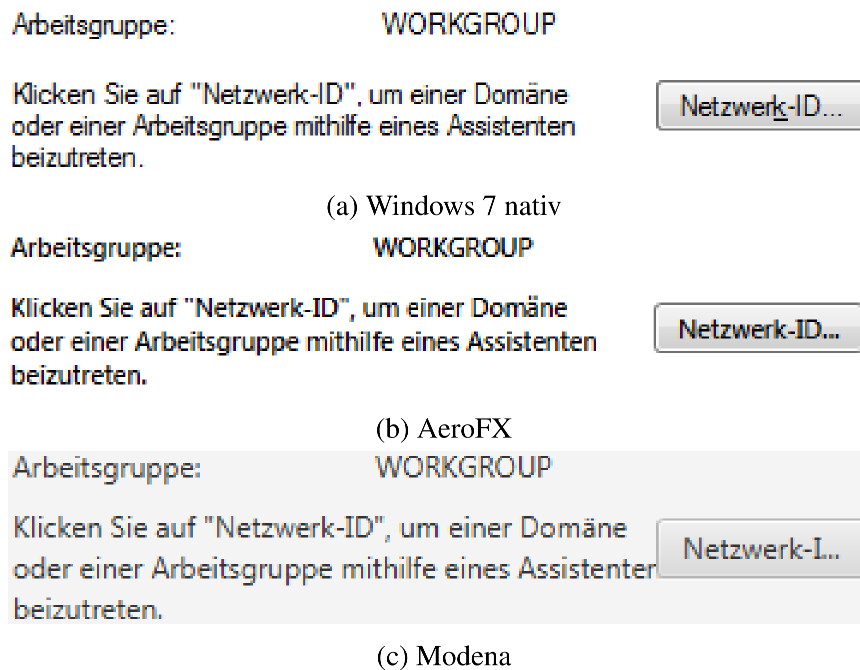


(c) Modena

Abbildung 5.1: Direkter Vergleich Windows 7 - AeroFX - Modena

Hier zeigt sich das klare Alleinstellungsmerkmal von AeroFX, nämlich die stellenweise täuschende Ähnlichkeit des Skins mit der nativen Oberfläche von Windows 7. Hierzu zeigen die Abbildungen 5.1a bis 5.1c den direkten Vergleich zwi-

schen dem Windows 7-Original, dem mit AeroFX gestylten Nachbau in JavaFX[2] und Modena.



(a) Windows 7 nativ

(b) AeroFX

(c) Modena

Abbildung 5.2: Detailvergleich Windows 7 - AeroFX - Modena

Ein vergrößerter Ausschnitt aus dem Dialog ist den Abbildungen 5.2a bis 5.2c zu entnehmen. Auch hier wird deutlich, wie ähnlich sich das Windows 7-Original und AeroFX sind, wobei gleichzeitig die Unterschiede zu Modena ersichtlich werden.

Die Stärke von AeroFX liegt in der konsistenten Nachahmung der visuellen Rückmeldungen von Windows 7. Als Beispiel kann der Nachbau des Buttons angeführt werden. Sowohl die Schriftgröße als auch sämtliche Zustände und Farbkombinationen sind im AeroFX-Button identisch im Vergleich zu seinem Windows 7-Vorbild.

Hierdurch wird die Immersion des Anwenders konsequent fortgeführt und ein harter Bruch zwischen nativen und JavaFX-basierten Anwendungen vermieden.

[2]Für einen detaillierten optischen Vergleich sei auf Anhang A verwiesen

Diesen Bruch im Oberflächendesign kann, wie Abbildung 5.2c zu entnehmen ist, Modena nicht verhindern. Bei Betrachtung besagter Abbildung zeigt sich gleichzeitig noch ein anderes gravierendes Problem von Modena, nämlich die Unterschiede im Textrendering. JavaFX nutzt die native Schriftart der jeweiligen Systemplattform, im vorliegenden Fall also Segoe UI. Obwohl es dieselbe Schriftart wie im nativen Dialog (Abbildung 5.2a) ist, sind die Unterschiede der Skalierung und Schriftgröße nicht zu vernachlässigen.
Die Schwächen von AeroFX, welche im Detail in Kapitel 4.5 vorgestellt wurden, zeigen sich bei Betrachtung von Abbildung 5.1b und 5.2b.
Erstere zeigt das Problem der Fensterdekoration, da JavaFX derzeit keine Möglichkeit bietet, die in der Titelleiste angezeigten Elemente auszublenden.
Letztere offenbart das Problem des unterschiedlichen Textrenderings, da es in der aktuellen Version von JavaFX keinerlei Möglichkeiten gibt, erweiterte Parameter im Rendering zu beeinflussen, wie beispielsweise Hinting, Smoothing oder Subpixel-Rendering.
Hinzu kommt der noch nicht vollständige Umfang der Implementierung, da zum aktuellen Zeitpunkt ein nicht zu vernachlässigender Teil von Elementen noch nicht gestylt wurde. Hierzu zählen unter Anderem Progress Bars, TreeViews, Menüleisten und vieles mehr.

AquaFX tritt in dieser Kategorie außer Konkurrenz an, da es nie als Windows-, sondern als OS X-Skin konzipiert war. JMetro fällt ebenfalls aus der Bewertung heraus, da es auf Windows 8 Look & Feel ausgelegt ist. JavaFX-Native-Themes zeigt erste Ansätze des Designs, bleibt allerdings weit hinter den Erwartungen an einen einsatzfähigen Skin zurück.
Modena zeigt die eingangs erwähnte Problematik, dass es als OS-unabhängiger Skin ausgelegt ist, dementsprechend ist die Konsistenz einer Anwendung, die Modena nutzt, gegenüber dem Windows 7 Look & Feel nicht gegeben, wodurch eingangs[3] erwähnte Probleme auftreten können, dass der Nutzer außerhalb seines gewohnten Schemas arbeiten muss.

[3] vgl. Kapitel 2.1

5.6 Tabellarische Zusammenfassung

Abbildung 5.3 greift die Tabelle aus Kapitel 3 auf und erweitert sie um die Spalte AeroFX, um die Ergebnisse nochmals übersichtlich zusammenzufassen und vergleichbar zu machen:

Projektname	Modena	JFX-Nat. Themes	JMetro	AquaFX	AeroFX
Lizenz	+	-	-	+	+
Dokumentation	o	-	-	+	+
Aufbau	+	o	o	+	+
Updateverteilung	+	o	-	+	+
Windows 7 L&F	-	o	-	-	+

Abbildung 5.3: Validierung

Aus dem neu hinzugefügten Eintrag ist ersichtlich, dass sich AeroFX in allen Wertungskategorien gegen seine Kontrahenten durchsetzt.
Wie sich bei der Validierung also gezeigt hat, muss AeroFX den Vergleich mit der aktuellen Konkurrenz nicht scheuen, obwohl noch einige Arbeiten ausstehen. Daher soll sich nun der Zusammenfassung gewidmet werden, welche ein resümierendes Bild der gesamten Arbeit zeichnet.

Kapitel 6

Zusammenfassung

Es folgt ein abschließendes Fazit mit Überblick über die Ergebnisse des Projektes inklusive eines Ausblicks auf weitere Möglichkeiten des Projekts.

6.1 Fazit

Der Einsatz von nativen Skins ist, wie in Kapitel 2.1 festgestellt, sehr empfehlenswert und wichtig, um möglichst breite Akzeptanz bei der Zielgruppe zu erzeugen.
Die derzeit am Markt befindlichen Skins decken, wie in Kapitel 3 ermittelt, jedoch den Bereich Windows 7 nur unzureichend ab. Alles, was in dieser Richtung bislang erhältlich ist, geht nicht über den Status einer Technologiedemo hinaus. Das einzige Projekt, welches als benutzbar bezeichnet werden kann, ist AquaFX, das aber keine Windows 7-Oberflächengestaltung liefert.
Aufgrund dieser Situation ist ein klarer Handlungsbedarf erkennbar gewesen, der durch die Durchführung dieser Arbeit gedeckt werden sollte.
Durch den Aufbau des Projekts als Enterprise-Anwendung mit Maven-Unterstützung und anwenderfreundlicher Lizenzierung werden Hemmungen beim Einsatz des Produktes abgebaut. In Verbindung mit der Strategie, neue Komponenten und Features erst zu veröffentlichen, wenn diese benutzbar sind, entsteht ein robuster Skin, der im täglichen Umfeld eingesetzt werden kann. Abgerundet durch die komplette, öffentliche Verfügbarkeit des Codes kann davon gesprochen werden, dass dieses Projekt einen wesentlichen Beitrag zum Stand der Technik darstellt und diesen sogar weiter voran treibt, indem mit AeroFX der erste benutzbare und teilweise funktionsfähige JavaFX-Skin für Windows 7 entwickelt wurde.

Die Entwicklung eines JavaFX-Skins von Grund auf ist zeitaufwendig und mit einigen Problemen verbunden, insbesondere da die Sprachspezifikation partiell noch nicht abgeschlossen ist und einige Elemente wie zum Beispiel die Basis-Skinklassen von Steuerelementen noch nicht öffentlich dokumentiert oder zur Nutzung freigegeben sind. Des Weiteren stößt man immer wieder auf unsaubere Implementierungen und teilweise nicht vorhandene Features, die ebenfalls den noch nicht finalen Status des Toolkits deutlich machen. Jedoch ist bereits jetzt, nach im Vergleich zu Swing sehr kurzer Zeit, gut erkennbar, wie leistungsfähig und robust JavaFX schon ist.
Die Unterstützung durch die Community ist ein Schlüssel zum Erfolg, den Oracle nicht nur erkannt hat, sondern sogar fördert. Um die Akzeptanz dieses Toolkits und dessen Einsatzhäufigkeit weiter zu steigern, sind native Themes, wie AeroFX, wichtig, da sie Hemmschwellen abbauen und den Einstieg in die Benutzung von nicht nativer Software unter den jeweiligen Zielsystemen vereinfachen.

6.2 Ausblick

Der hier vorgestellte Stand der Entwicklung soll noch nicht das Ende des Projekts markieren. Betrachtet man den aktuellen Stand, so stellt man fest, dass für den produktiven Einsatz noch weitere Elemente durch den Skin unterstützt werden müssen.
Es steht außerdem die Überlegung im Raum nach entsprechender Absprache mit der Entwicklerin die beiden Projekte AquaFX und AeroFX zu vereinen, da sie beide dasselbe Ziel verfolgen und eine vergleichbare Architektur besitzen. Ein solcher Zusammenschluss kann als Basis für weitere Arbeiten in dem Umfeld dienen, beispielsweise auch die Entwicklung von Skins für Linux-Desktopumgebungen wie GNOME oder KDE.
Aufgrund des Feedbacks aus der Community besteht eindeutig Bedarf und ein großes Interesse an nativen Skins, daher kann und sollte nicht auf die Weiterentwicklung der Arbeit verzichtet werden. Das Interesse der Community zeigt sich durch die Reaktionen auf diverse Ankündigungen und Blogeinträge. Hierzu sind zunächst die Einträge im Blog von Hendrik Ebbers zu nennen.
Die erste Ankündigung mit dem Titel „Sneak Peek: AeroFX“[1] wurde über 1200 Mal gelesen, das Statusupdate „AeroFX: It’s getting closer“[2] sogar rund 1600 Mal.

[1] http://www.guigarage.com/2014/06/sneak-peek-aerofx/
[2] http://www.guigarage.com/2014/06/aerofx-getting-closer/

Beide Ankündigungen wurden in den jeweiligen Wochen, also den Kalenderwochen 25[3] und 26[4], als „Links der Woche“ bei fxexperience mit einigen Sätzen kommentiert und empfohlen. Da die Links auf die Einträge im Blog von Hendrik Ebbers verweisen, wurden sie entsprechend auch ihm zugeordnet.
In den „Links der Woche“ für die Kalenderwoche 35[5], welche am 24. August veröffentlicht wurden, stellt sich die Situation hingegen anders dar. In besagten Empfehlungen wurde unter Nennung des Autors die Projekthomepage und das GitHub-Projekt vorgestellt und äußerst positiv bewertet. Hierdurch wird nochmals das Potential verdeutlicht, welches AeroFX birgt.

[3]http://fxexperience.com/2014/06/javafx-links-of-the-week-june-16/
[4]http://fxexperience.com/2014/06/javafx-links-of-the-week-june-23/
[5]http://fxexperience.com/2014/08/javafx-links-of-the-week-august-25/

Literatur

[AMW10] Jonathan Anderson, John McRee und Robb Wilson. *Effective UI*. 1st ed. Beijing und Cambridge: O'Reilly, 2010. ISBN: 9780596154783.

[Dea14] C. Dea. *Javafx 8: Introduction by example*. 2ND ED. Berkeley: Apress, 2014. ISBN: 978-1-4302-6460-6.

[Dja+14] Soussan Djamasbi u. a. »Designing for Success: Creating Business Value with Mobile User Experience (UX)«. In: (2014). URL: http://digitalcommons.wpi.edu/uxdmrl-pubs/43.

[Ebb14] Hendrik Ebbers. *Mastering Javafx 8 Controls: Create Custom JavaFX Controls for Cross-Platform Applications*. McGraw-Hill Osborne Media, 2014. ISBN: 9780071833776.

[Fog05] Karl Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. 1st ed. Sebastopol und CA: O'Reilly Media und O'Reilly, 2005. ISBN: 978-0596007591. URL: http://www.producingoss.com/.

[Hay] Jackie R. Hayes. »User Interface Design for Online Social Media«. Diss. URL: http://digitalcommons.calpoly.edu/grcsp/108.

[Kra12] Christian Kraft. »User Experience and Why It Matters«. In: *User Experience Innovation*. Apress, 2012, S. 1–10. ISBN: 978-1-4302-4149-2. DOI: 10.1007/978-1-4302-4150-8_1. URL: http://dx.doi.org/10.1007/978-1-4302-4150-8%5C_1.

[Kru06] Steve Krug. *Don't make me think! A common sense approach to Web usability*. 2nd ed. Berkeley und Calif: New Riders Pub., 2006. ISBN: 0321344758.

[Mic] Microsoft. *Design apps for the Windows desktop - Windows Dev Center: Learn how to design beautiful and functional desktop apps that work great with Windows*. URL: http://msdn.microsoft.com/en-US/windows/desktop/aa511258.aspx (besucht am 26. 07. 2014).

[Naw14] Ather Nawaz. »Website user experience ; A cross-cultural study of the relation between users\textasciiacute cognitive style, context of use, and information architecture of local websites«. Diss. 2014. URL: http://hdl.handle.net/10398/8871.

[Net14] Net Applications. *Desktop Operating System Market Share*. 2014. URL: http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10%5C&qpcustomd=0 (besucht am 19. 07. 2014).

[Oraa] Oracle. *[#RT-20299] Native Look and Feels - JavaFX: JavaFX Bugtracker*. URL: https://javafx-jira.kenai.com/browse/RT-20299 (besucht am 28. 07. 2014).

[Orab] Oracle. *[#RT-37956] Win: StageStyle.UTILITY does not imitate native Windows 7 UTILITY-Style - JavaFX: JavaFX Bugtracker*. URL: https://javafx-jira.kenai.com/browse/RT-37956 (besucht am 29. 07. 2014).

[Ora13a] Oracle. *Using JavaFX UI Controls: Titled Pane and Accordion: JavaFX 2 Tutorials and Documentation*. 2013. URL: http://docs.oracle.com/javafx/2/ui%5C_controls/accordion-titledpane.htm (besucht am 29. 07. 2014).

[Ora13b] Oracle. *Why, Where, and How JavaFX Makes Sense*. 2013. URL: http://www.oracle.com/technetwork/articles/java/casa-1919152.html (besucht am 10. 08. 2014).

[Ora14a] Oracle. *1 JavaFX Overview (Release 8)*. 14.07.2014. URL: http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm%5C#BABEDDGH (besucht am 28. 07. 2014).

[Ora14b] Oracle. *2 Understanding the JavaFX Architecture (Release 8)*. 2014-07-14. URL: http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-architecture.htm (besucht am 28. 07. 2014).

[Reh13] Mario Rehse. *Deutscher ITK-Markt nimmt 2014 Fahrt auf*. Berlin, 22. Okt. 2013. URL: http://www.bitkom.org/de/markt%5C_statistik/64086%5C_77663.aspx (besucht am 19. 07. 2014).

[Seo+14] Hyunmin Seo u. a. »Programmers' build errors: a case study (at google)«. In: *the 36th International Conference*. Hrsg. von Pankaj Jalote, Lionel Briand und André van der Hoek. 2014, S. 724–734. DOI: 10.1145/2568225.2568255. URL: http://research.google.com/pubs/pub42184.html.

[Tia+] Tiago Silva Da Silva u. a. »User Experience Design and Agile Development: From Theory to Practice«. Diss. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.359.7189.

[Wik14] Wikipedia. *Apache Maven*. Hrsg. von Wikipedia. 2014-07-14. URL: http://de.wikipedia.org/w/index.php?oldid=130775428 (besucht am 26. 07. 2014).

Anhang A

Ein kompletter Vergleich aller Registerkarten des Dialogs „Systemeigenschaften“ zwischen dem Windows 7-Original und JavaFX mit AeroFX-Skin.

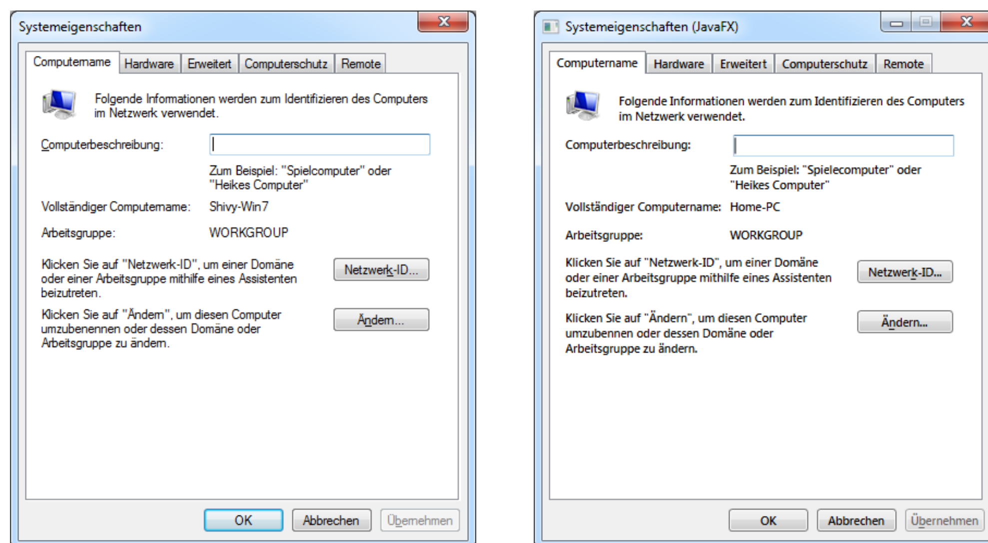


Abbildung A.1: Vergleich: Windows 7 und AeroFX - Register „Computername“

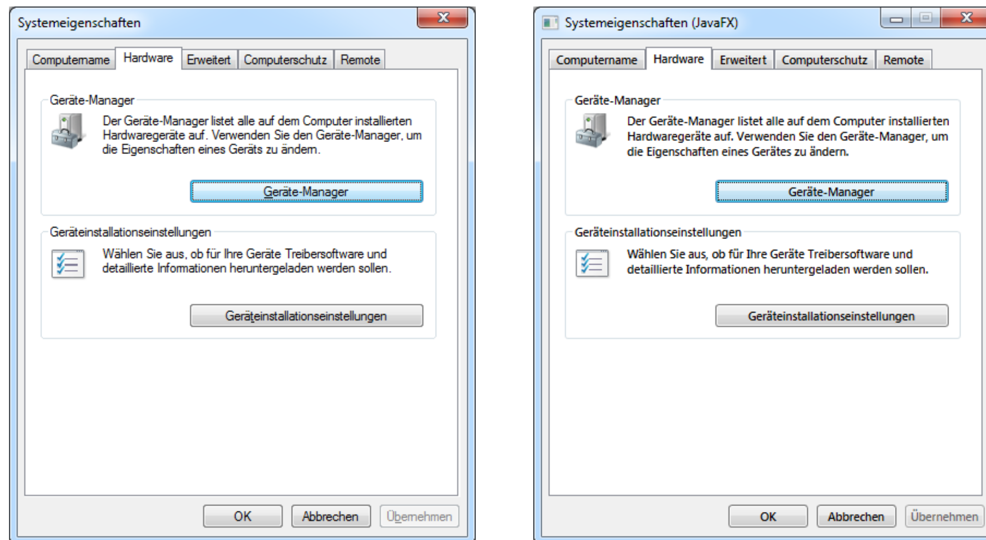


Abbildung A.2: Vergleich: Windows 7 und AeroFX - Register „Hardware“

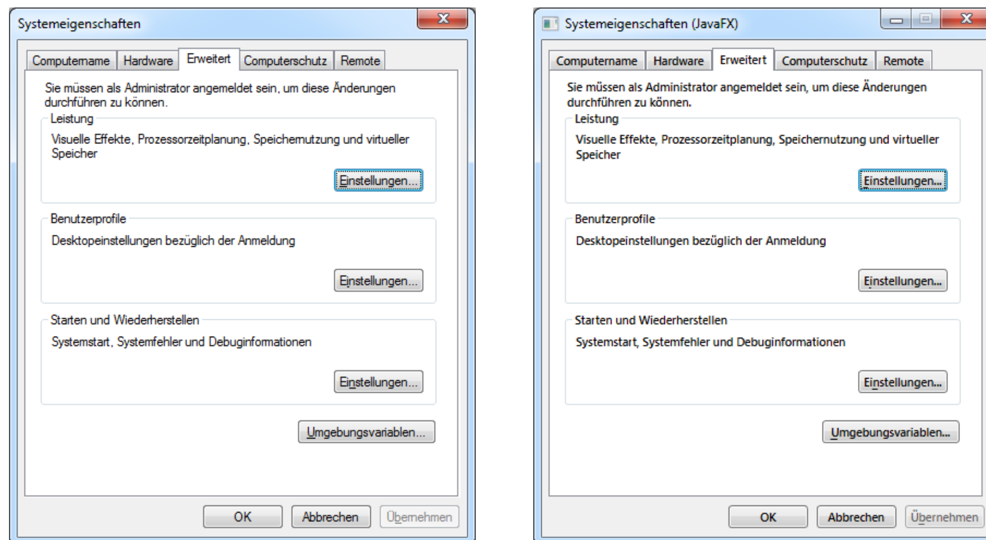


Abbildung A.3: Vergleich: Windows 7 und AeroFX - Register „Erweitert“

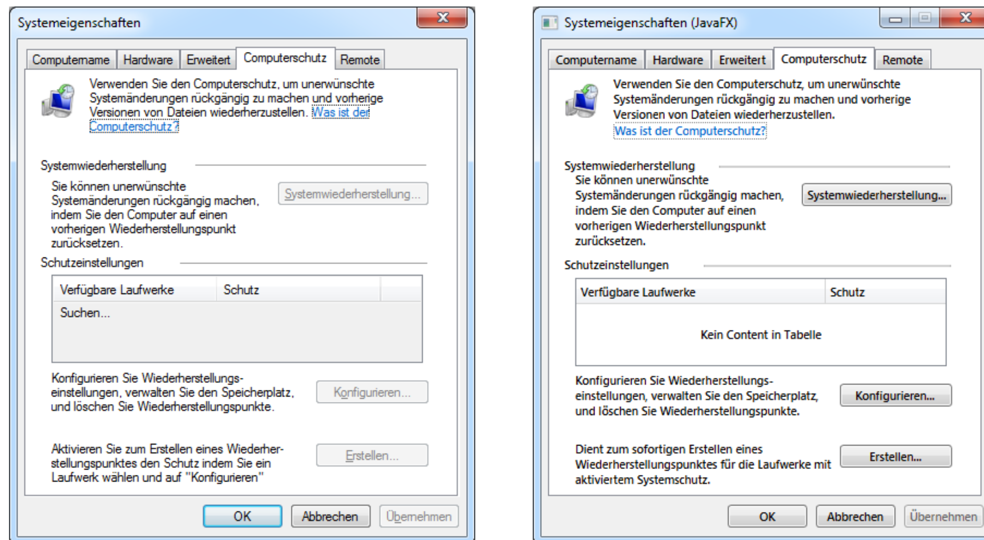


Abbildung A.4: Vergleich: Windows 7 und AeroFX - Register „Computerschutz“

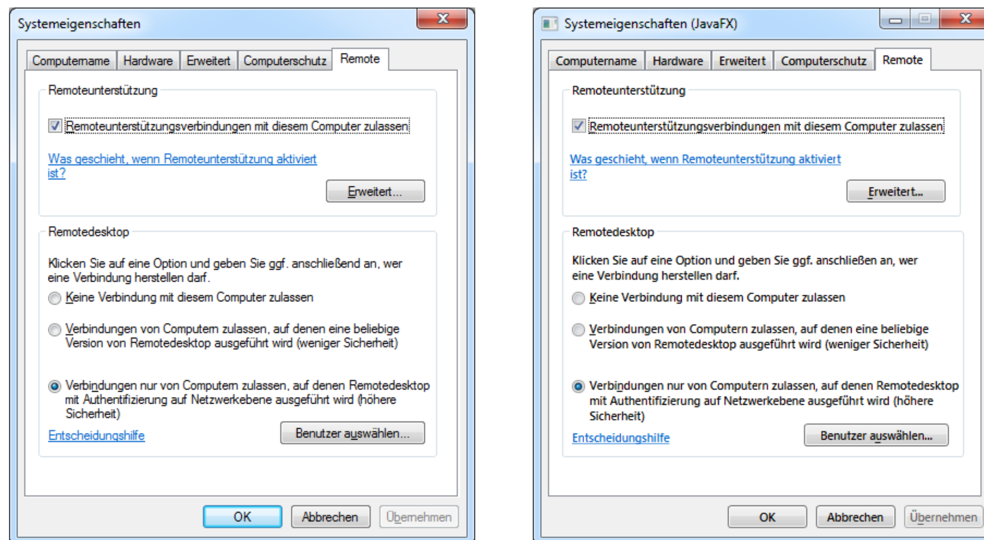


Abbildung A.5: Vergleich: Windows 7 und AeroFX - Register „Remote“

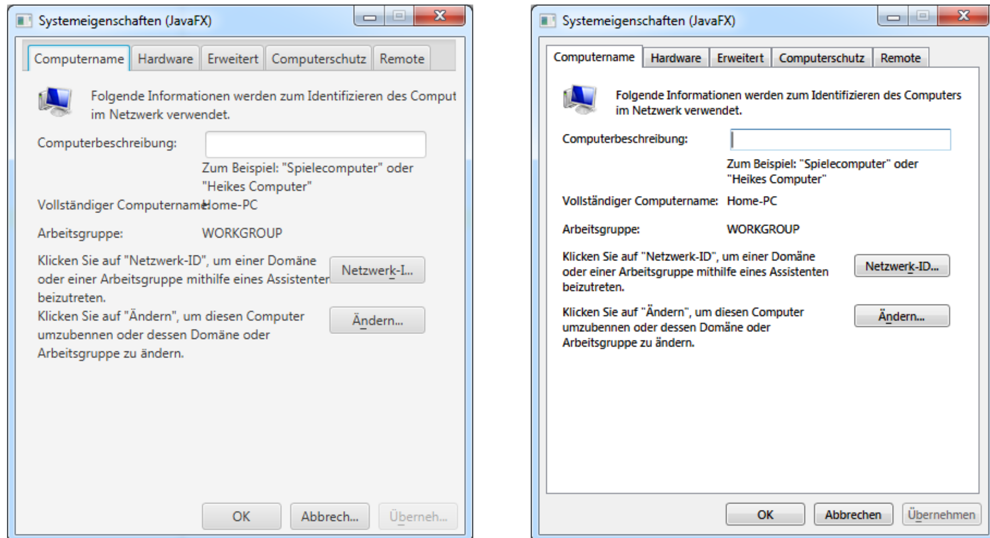


Abbildung A.6: Vergleich: Modena und AeroFX - Register „Computername“

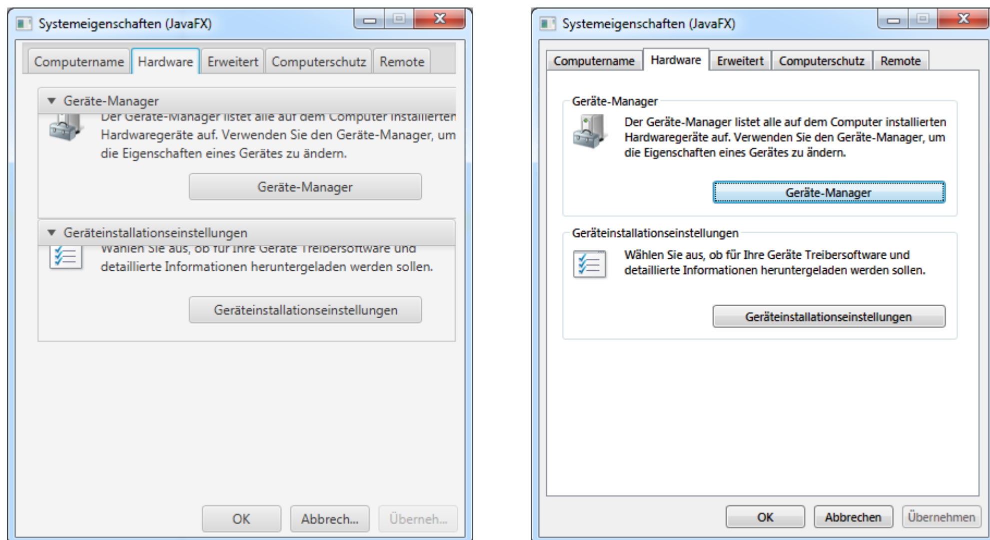


Abbildung A.7: Vergleich: Modena und AeroFX - Register „Hardware“

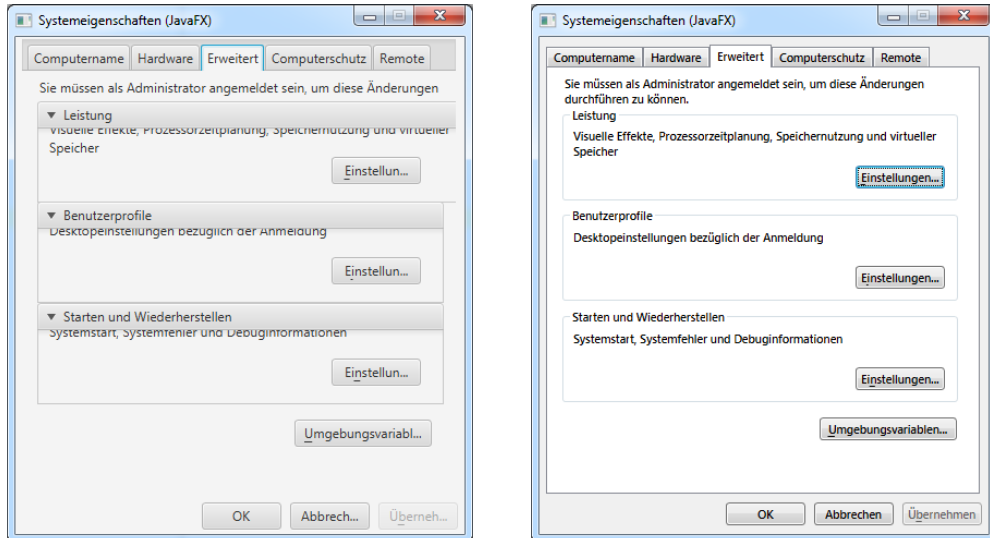


Abbildung A.8: Vergleich: Modena und AeroFX - Register „Erweitert“

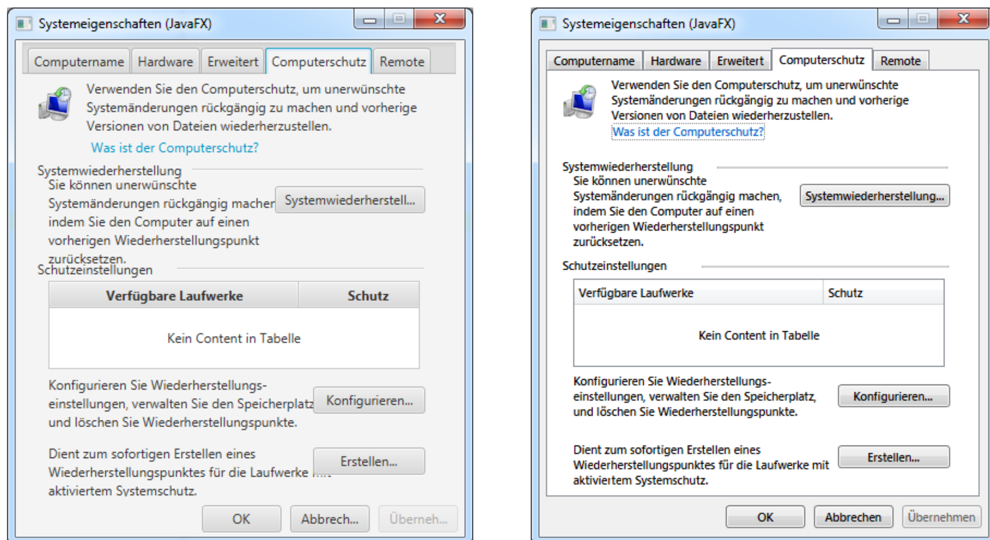


Abbildung A.9: Vergleich: Modena und AeroFX - Register „Computerschutz“

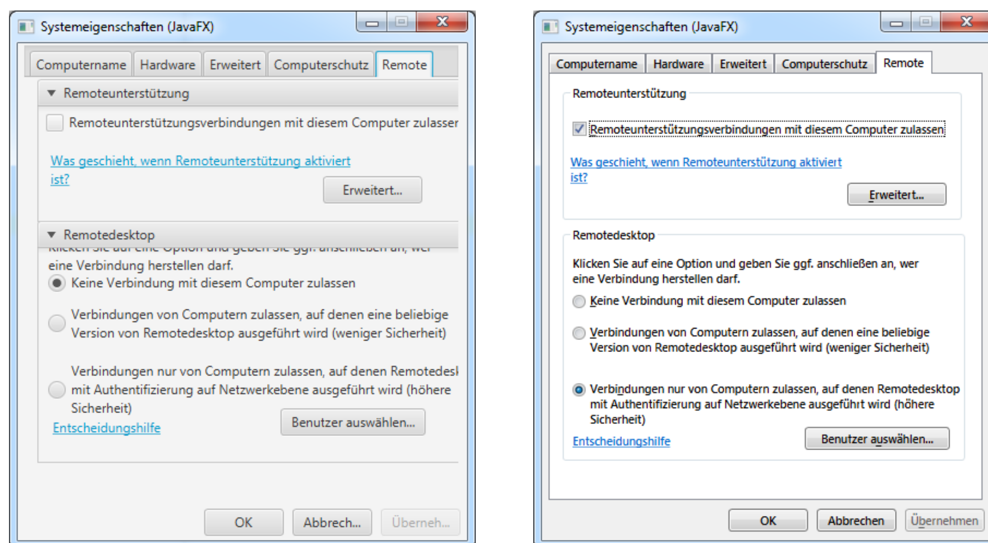


Abbildung A.10: Vergleich: Modena und AeroFX - Register „Remote“

Anhang B

Es folgen Codelistings, die aufgrund ihrer Länge keinen Platz im Hauptteil der Thesis gefunden haben.

```
1 private void setFocusedButtonAnimation(){
2   if(!getSkinnable().isDisabled()){
3
4     if(focusedButtonTransition != null &&
       focusedButtonTransition.getStatus() ==
       Animation.Status.RUNNING)
5       resetAnimation();
6
7     else{
8       final Duration duration = Duration.millis(1000);
9       focusedButtonTransition = new
       BindableTransition(duration);
10      focusedButtonTransition
11        .setCycleCount(Timeline.INDEFINITE);
12      focusedButtonTransition.setAutoReverse(true);
13
14      //starting gradient
15      final Color startColor1 = Color.rgb(242,242,242);
16      final Color startColor2 = Color.rgb(235,235,235);
17      final Color startColor3 = Color.rgb(221,221,221);
18      final Color startColor4 = Color.rgb(207,207,207);
19
20      //ending gradient
21      final Color endColor1 = Color.rgb(235,246,252);
22      final Color endColor2 = Color.rgb(229,243,251);
23      final Color endColor3 = Color.rgb(203,232,248);
24      final Color endColor4 = Color.rgb(184,221,242);
25
26
27
28
```

```
29      focusedButtonTransition.fractionProperty()
30        .addListener(new ChangeListener<Number>() {
31       @Override
32       public void changed(ObservableValue<? extends Number>
      observable, Number oldValue, Number newValue) {
33          List<BackgroundFill> list = new ArrayList<>();
34          Stop[] stops = new Stop[] {
35          new Stop(0f, Color.color(
36            (endColor1.getRed() - startColor1.getRed())
37               * newValue.doubleValue()
38               + startColor1.getRed(),
39            (endColor1.getGreen() - startColor1.getGreen())
40               * newValue.doubleValue()
41               + startColor1.getGreen(),
42            (endColor1.getBlue() - startColor1.getBlue())
43               * newValue.doubleValue()
44               + startColor1.getBlue())),
45          new Stop(0.49f, Color.color(
46            (endColor2.getRed() - startColor2.getRed())
47               * newValue.doubleValue()
48               + startColor2.getRed(),
49            (endColor2.getGreen() - startColor2.getGreen())
50               * newValue.doubleValue()
51               + startColor2.getGreen(),
52            (endColor2.getBlue() - startColor2.getBlue())
53               * newValue.doubleValue()
54               + startColor2.getBlue())),
55          new Stop(0.5f, Color.color(
56            (endColor3.getRed() - startColor3.getRed())
57               * newValue.doubleValue()
58               + startColor3.getRed(),
59            (endColor3.getGreen() - startColor3.getGreen())
60               * newValue.doubleValue()
61               + startColor3.getGreen(),
62            (endColor3.getBlue() - startColor3.getBlue())
63               * newValue.doubleValue()
64               + startColor3.getBlue())),
65          new Stop(1f, Color.color(
66            (endColor4.getRed() - startColor4.getRed())
67               * newValue.doubleValue()
68               + startColor4.getRed(),
69            (endColor4.getGreen() - startColor4.getGreen())
70               * newValue.doubleValue()
71               + startColor4.getGreen(),
72            (endColor4.getBlue() - startColor4.getBlue())
```

```
73                  * newValue.doubleValue()
74                  + startColor4.getBlue())) };
75
76
77             //Build up rectangles
78             BackgroundFill f1 = new
        BackgroundFill(Color.rgb(60, 127, 177), new
        CornerRadii(3.0), new Insets(0.0));
79             list.add(f1);
80             BackgroundFill f2 = new
        BackgroundFill(Color.rgb(72,216,251), new
        CornerRadii(2.0), new Insets(1.0));
81             list.add(f2);
82             LinearGradient gradient = new
        LinearGradient(0.0,0.0,0.0,1.0,true,
        CycleMethod.NO_CYCLE,stops);
83             BackgroundFill bgFill = new
        BackgroundFill(gradient, new CornerRadii(1.0), new
        Insets(2.0));
84             list.add(bgFill);
85             ((StyleableProperty<Background>)getSkinnable()
86               .backgroundProperty()).applyStyle(
87                  null, new Background(
88                     list.get(0), list.get(1), list.get(2)));
89          }
90        });
91     }
92  }
```

Listing B.1: AeroButtonSkin: Fokusanimation

Anhang C

Eine Tabellarische Zuordnung der Quellen zu PDF-Dateien, welche sich auf der CD befinden.
Die HTML-Quellen wurden als PDF-Dateien mit angegebenem Datum offline verfügbar gemacht und sind im Ordner ”PDF-Offline“ einsehbar.

Quelle	Name der PDF-Datei
([Mic])	Design apps for the Windows.pdf
([Net14])	ds0ihl4p.pdf
([Oraa])	Oracle - [#RT-20299] Native Look and Feels.pdf
([Orab])	Oracle - [#RT-37956] Win.pdf
([Ora13b])	Why, Where.pdf
([Ora14a])	1 JavaFX Overview Release 8 14072014.pdf
([Ora14b])	Oracle 14072014 - 2 Understanding the JavaFX Architecture.pdf
([Reh13])	Rehse 22102013 - Deutscher ITK-Markt nimmt 2014 Fahrt.pdf
([Wik14])	Wikipedia (Hg) 14072014 - Apache Maven.pdf

Anhang D - Eidesstattliche Erklärung

Gemäß § 17,(5) der BPO erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Ich habe mich keiner fremden Hilfe bedient und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, 26. August 2014 Matthias Kurt Walter Meidinger

Erklärung

Mir ist bekannt, dass nach § 156 StGB bzw. § 163 StGB eine falsche Versicherung an Eides Statt bzw. eine fahrlässige falsche Versicherung an Eides Statt mit Freiheitsstrafe bis zu drei Jahren bzw. bis zu einem Jahr oder mit Geldstrafe bestraft werden kann.

Dortmund, 26. August 2014 Matthias Kurt Walter Meidinger