



Department für Informatik
Abteilung für Medieninformatik und Multimedia-Systeme

Bachelorarbeit

Annotationsbasierte Einstiegserleichterung in
die Entwicklung von JavaFX-Anwendungen

Deniz Groenhoff

15. September 2021

1. Gutachterin: Prof. Dr. Susanne Boll-Westermann
2. Gutachter: Dr.-Ing. Dietrich Boles

Erklärung

Ich erkläre an Eides statt, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichungen, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

D. Groenhoff

Deniz Groenhoff
Matrikelnummer 5477417
Oldenburg, den 15. September 2021

Inhaltsverzeichnis

1. Einleitung	3
1.1. Motivation	3
1.2. Zielsetzung	3
1.3. Struktur	4
2. Grundlagen	7
2.1. Entwurfsmuster	7
2.1.1. Beobachter	7
2.1.2. MVC	9
2.2. JavaFX	10
2.2.1. Aufbau und Szenengraph	10
2.2.2. Properties und Bindings	11
2.2.3. Layouting: FXML vs. Quelltext	12
2.3. Java-Annotationen	13
2.3.1. Definition	14
2.3.2. Syntax	15
2.3.3. Auswertung von Annotationen zur Laufzeit	17
2.3.4. Auswertung von Annotationen zur Kompilierzeit	17
2.4. Zusammenfassung	18
3. Stand der Technik	19
3.1. Annotationen im Umfeld von JavaSE/JavaEE/JavaFX	19
3.1.1. JavaSE Umgebung und externe Bibliotheken	19
3.1.2. JavaEE/JakartaEE Umgebung	21
3.1.3. JavaFX Umgebung	22
3.2. Annotationen in anderen Programmiersprachen	22
3.2.1. Verwendung in Python	22
3.2.2. Verwendung in C# und .NET	22
3.3. Fazit	23
4. Konzeption und Entwurf	25
4.1. Identifikation von Problemen und komplexen Strukturen in der JavaFX Entwicklung	25
4.1.1. Internationalisierung und Lokalisierung	26
4.1.2. Abhängigkeitsinjektion für Controller	27
4.1.3. CSS Metadatengeneration	27
4.1.4. JavaFX Einstiegspunkt und Preloader	28

4.1.5. Controller Lebenszyklus	28
4.1.6. Konfigurationsdateien	29
4.2. Anforderungsanalyse	29
4.2.1. Funktionale Anforderungen	30
4.2.2. Nichtfunktionale Anforderungen	35
4.3. Konzept und Modellierung	36
4.3.1. Ziele des Systems	37
4.3.2. Rahmenbedingungen und Designentscheidungen	37
4.3.3. Benötigte Schnittstellen	39
4.3.4. Controller System	45
4.3.5. Annotationen	47
5. Implementierung	53
5.1. Architektur und Struktur der Software	53
5.2. Paketstrukturierung nach Funktionalität	54
5.2.1. Paket: utils	54
5.2.2. Paket: di	55
5.2.3. Paket: dagger1	55
5.2.4. Paket: guice	56
5.2.5. Paket: spring	56
5.2.6. Paket: localization	56
5.2.7. Paket: controller	56
5.2.8. Paket: exception	57
5.2.9. Paket: css	57
5.2.10. Paket: experimental	57
5.2.11. Paket: classpath	57
5.2.12. Paket: shared	59
5.2.13. Paket: event	59
5.2.14. Paket: events	59
5.2.15. Paket: application	60
5.2.16. Paket: config	60
5.3. Essentielle Quelltextausschnitte	60
5.3.1. SimpliFXMLLoader	60
5.3.2. Erweiterbare Abhängigkeitsinjektion	62
6. Prototypische Anwendung und Evaluation der Bibliothek	63
6.1. Entwicklung von Beispielsoftware	63
6.1.1. Struktur und Funktion der Beispielanwendung	64
6.1.2. Implementierung der Beispielanwendung	65
6.2. Evaluation der Bibliothek	71
6.2.1. Start der Anwendung	72
6.2.2. Einstiegspunkte	72
6.2.3. Lokalisierung	72
6.2.4. Controller	73

6.2.5. Abhängigkeitsinjektion	73
6.2.6. Konfigurationsdateien	73
7. Fazit	75
7.1. Zusammenfassung und Bewertung	75
7.2. Ausblick und mögliche Erweiterungen	77
A. Controllerbasierte JavaFX-Anwendung	79
B. Hinzufügen einer Bibliothek für die Abhängigkeitsinjektion	81
C. Schnittstellen und Beziehungen des Controller-Systems	83
D. Demo Anwendung	85
Abkürzungsverzeichnis	91
Quellcodeverzeichnis	93
Abbildungsverzeichnis	95
Tabellenverzeichnis	97
Literaturverzeichnis	99

1. Einleitung

Im Rahmen der Digitalisierung ist das Programmieren zugänglicher und massentauglicher als je zuvor und es wird immer mehr auf Fertigkeiten wie das effiziente und weitgehend fehlerfreie Programmieren, vor allem in sicherheitskritischen Umgebungen, gesetzt. Die Verwendung von Quelltextmetadaten kann verschiedenen Quelltextelementen einen hohen Grad an Struktur geben. Diese Metadaten sind in vielen Programmiersprachen aber auch außerhalb der Informatik als Annotations¹ bekannt. Diese sind ein dynamisches Programmierkonstrukt, welches immer populärer wird und in den meisten Fällen für eine Vereinfachung von komplexen Aufgaben, der Automation von repetitiven Vorgängen und der Dokumentation von Quelltextelementen sorgen kann.

1.1. Motivation

Zur Erstellung von platformunabhängigen Java-Anwendungen mit grafischer Benutzungsschnittstelle wird immer mehr auf JavaFX, den Nachfolger der Swing Bibliothek, gesetzt. Dies erlaubt es Entwicklern, mit relativ wenig Zeitaufwand zeitgemäße und moderne Applikationen zu erstellen. Selbst Entwickler, welche über keinerlei Erfahrung mit der Entwicklung von grafischen Oberflächen in der Java Programmiersprache verfügen, werden keine Schwierigkeiten haben, solche mittels JavaFX zu entwerfen und schließlich zu implementieren. Wie später in der Problemanalyse gezeigt, bietet JavaFX zwar einen einfachen Einstieg in grundlegende Programmierkonstrukte, kann aber bei der Erstellung von komplexen Anwendungen dennoch schnell zeitaufwendig und fehleranfällig werden.

1.2. Zielsetzung

In dieser Arbeit soll aufgrund der simplifizierenden Natur von Annotationen in anderen Teilgebieten der Programmierung eine Bibliothek entwickelt werden, welche verschiedene Aspekte im Entwicklungsprozess von JavaFX Anwendungen durch die Verwendung von Annotationen vereinfacht oder vollständig ersetzt. Dazu soll der Funktionsumfang erweitert werden, indem bereits vorhandene Funktionen dynamischer gestaltet oder vollständig neue Funktionen implementiert werden. Konzepte aus anderen Bibliotheken für die grafische Benutzeroberflächenentwicklung sollen dabei ebenfalls in Betracht gezogen werden. Auch wenn bei der Implementierung

¹Wenn in der Arbeit über Annotationen gesprochen wird, ist immer von Java-Annotationen auszugehen (außer anders angegeben).

hauptsächlich auf die Nutzung von Annotationen zurückgegriffen wird, sollen ebenfalls unabhängige Schnittstellen entwickelt werden, welche vom Entwickler genutzt werden können. Um mögliche komplexe Strukturen zu vereinfachen und problematische Funktionen zu identifizieren, soll eine Problemanalyse von bereits vorhandenen JavaFX Konzepten durchgeführt werden. Die gefundenen Probleme sollen dann mithilfe einer Anforderungsanalyse durch funktionale und nicht funktionale Anforderungen dargestellt werden. Für die vorher ermittelten Probleme sollen auf Annotationen basierende Konzepte entwickelt werden, um eine optimale Lösung zu finden. Wird keine Lösung durch das Einführen von neuen Annotationen gefunden, so sollen alternative Lösungsvorschläge eruiert werden. Die Konzepte bilden das Fundament der vollständigen Implementierung und Dokumentation der Bibliothek. Die Implementierung wird zum Abschluss der Arbeit durch eine Demoapplikation evaluiert.

1.3. Struktur

Die Struktur der Arbeit setzt sich aus sechs Teilen zusammen. Begonnen wird mit der Definition und Erklärung von fundamentalen Kernkonzepten, aus welchen ein optimales Verständnis der genutzten Technologien resultieren soll. Zu diesen Kernkonzepten gehören essentielle softwaretechnische Grundlagen wie Entwurfsmuster oder das Vorstellen und die anschließende nähere Beleuchtung der Annotationsfunktion in Java und anderen Programmiersprachen. Außerdem wird auf einzelne Komponenten von JavaFX eingegangen und deren Interaktionen untereinander mit Beispielen untermauert (Kapitel 2).

Danach werden aktuelle Beispiele zur Annotationsprogrammierung aus anderen Arbeiten, Bibliotheken und Programmiersprachen vorgestellt. Dabei liegt der primäre Fokus auf einer Vereinfachung des Entwicklungsprozesses in der Java Umgebung. Der Stand der Technik wird abschließend in einem Fazit im zweiten Teil der Arbeit zusammengefasst und es erfolgt ein kurzer Ausblick auf Funktionalitäten in JavaFX, welche in bisherigen Arbeiten/Bibliotheken nicht zu finden waren (Kapitel 3).

Nachdem die gegenwärtig verfügbaren Maßnahmen zur Simplifizierung durch die Verwendung von Annotationen vorgestellt wurden, wird mit dem nächsten Teil, dem Entwickeln eines Entwurfes, begonnen. Dabei erfolgt die in der Zielsetzung beschriebene Problemanalyse und außerdem werden notwendige Rahmenbedingungen an die Bibliothek und die Softwarearchitektur erhoben und verwendete externe Bibliotheken kurz vorgestellt. Außerdem werden Konzepte für essentielle Schnittstellen entworfen, die einerseits für eine Gewährleistung der Funktionalität notwendig sind und andererseits für den Bibliotheksnutzer unabhängig verwendbar gemacht werden. Des Weiteren werden alle zu implementierenden Annotationen mit ihren Funktionen und Limitationen vorgestellt (Kapitel 4).

Mit den aus dem Entwurf und dem Konzept resultierenden Problemlösungen, Rahmenbedingungen an die Architektur und Anforderungen, wird das System in Kapitel 5: Implementierung prototypisch implementiert. Dazu werden die implemen-

tierten Subsysteme näher betrachtet und teilweise mit Diagrammen untermauert. Darüber hinaus werden wichtige Teile des Entwicklungsprozesses aufgegriffen und in Form von Quelltextausschnitten erklärt.

Nachdem der Entwicklungsprozess des Systems beendet wurde, wird eine Anwendung zur Demonstration der Funktionen erstellt, welche durch das Nutzen von den hinzugefügten Annotationen als Hilfestellung für eventuelle Nutzer der Bibliothek dienen soll. Dabei wird explizit das Zusammenspiel der einzelnen Subsysteme vorgestellt. Darauf folgt ein Vergleich der implementierten Funktionen mit denen von JavaFX, mit einem besonderen Fokus auf Aspekte wie die Benutzungsfreundlichkeit oder dem Programmieraufwand (Kapitel 6).

Im letzten Kapitel erfolgt eine Zusammenfassung der erarbeiteten Ergebnisse sowie eine Bewertung dieser. Des Weiteren wird auf nicht funktionale Anforderungen wie die Erweiterbarkeit und die Wartbarkeit des Systems eingegangen (Kapitel 7).

2. Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen von essentiellen Komponenten dieser Arbeit erläutert. Dazu wird die Relevanz von Entwurfsmustern erklärt und auf zwei bedeutende Muster näher eingegangen. Diese sind sowohl erforderlich für die folgenden Kapitel als auch für das Verständnis der softwaretechnischen Prinzipien von JavaFX.

Danach wird die JavaFX-Bibliothek vorgestellt und fundamentale Konzepte wie beispielsweise die auf der Extensible Markup Language (XML) basierende Layouting-Sprache erläutert.

Abschließend wird das generelle Annotationskonzept in der Informatik mit speziellem Fokus auf die Programmiersprache Java erklärt. Dabei werden die verschiedenen Annotationstypen näher beschrieben und die Möglichkeiten der eigentlichen Auswertung dieser skizziert. Komplexe Konzepte werden dabei durch visuelle Beispiele wie Quelltextausschnitte¹ oder Unified Modeling Language (UML)-Klassendiagramme untermauert und möglicherweise vereinfacht.

2.1. Entwurfsmuster

Entwurfsmuster sind Lösungen für immer wieder auftretende Probleme bei der Softwareentwicklung. Sie stellen eine wiederverwendbare Problemlösung für architektonisch begründete Problematiken dar, welche im Endeffekt durch nur wenige Klassen und Schnittstellen effektiv und schnell gelöst werden können. Ein Entwurfsmuster setzt sich aus vier Komponenten zusammen: dem Namen des Musters, dem zu lösenden Problem, der daraus resultierenden Lösung und die auftretenden positiven sowie negativen Auswirkungen bei Nutzung des Musters [GHJV94].

Im Folgenden werden das Model-View-Controller (MVC)- sowie das Beobachter-entwurfsmuster für das Verständnis von JavaFX Prinzipien beschrieben.

2.1.1. Beobachter

Das Beobachterentwurfsmuster ist ein essentieller Bestandteil von vielen auf der reaktiven Programmierung aufbauenden Bibliotheken und APIs [SMT15] und obwohl es häufig in der Kritik steht, wird es dennoch in vielen Bereitstellungsumgebungen genutzt [MRO10].

¹Die dargestellten Quelltextausschnitte sind aufgrund der Simplizität nicht immer kompilierbar, da irrelevante Programmkonstrukte wie Importe von Klassen nicht für ein Verständnis des dargestellten Kontextes benötigt werden.

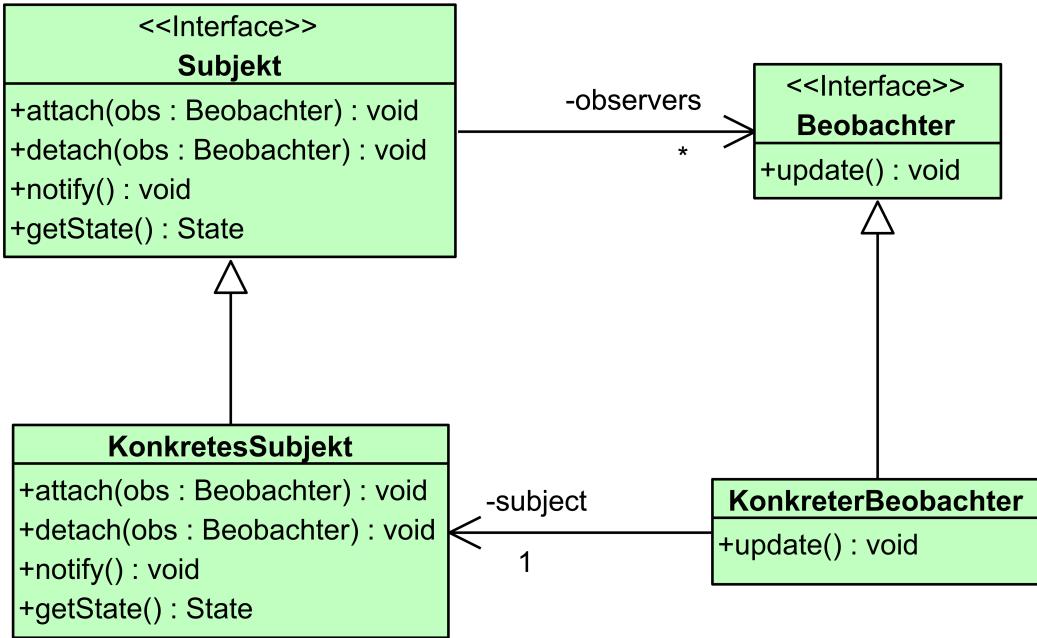


Abbildung 2.1.: UML-Diagramm – Beobachter-Entwurfsmuster

Das Entwurfsmuster benötigt für die korrekte Implementierung mindestens vier Komponenten (siehe Abbildung 2.1) [GHJV94]:

Das **Subjekt** ist das zu beobachtende Objekt, welches zu jedem Zeitpunkt alle seine Beobachter in einer internen Datenstruktur speichert. Es besitzt Methoden zum An- und Abmelden von Beobachtern und ist in der Lage alle Beobachter bei eventuellen Zustandsänderungen zu benachrichtigen.

Der **Beobachter** bietet eine Schnittstelle für Objekte, welche bei einer Zustandsänderung des Subjekts informiert werden sollen.

Die **KonkretesSubjekt** Komponente ist die konkrete Implementierung der Subjekt Schnittstelle und ist fähig, einen internen Zustand zu verwalten sowie bei einer Änderung von diesem alle registrierten Beobachter zu informieren.

Ein **KonkreterBeobachter** besitzt eine Referenz auf das zu beobachtende Subjekt und seinen internen Zustand. Bei einer Aktualisierung des Subjektzustands wird auch der interne Zustand des konkreten Beobachters aktualisiert.

2.1.2. MVC

Das MVC Entwurfsmuster² ist ein de facto Standard der objektorientierten Programmierung [Dea95], welches für eine Trennung von grafischer Oberfläche, Eingaben des Benutzers und dem eigentlichen Anwendungsmodell sorgt [Bur92]. Hält eine Anwendung diese strikte Trennung ein, so genügt sie dem softwaretechnischen Separation of Concerns (SoC) Prinzip [Gra14], woraus wiederum der Wartungsprozess vereinfacht und die Testbarkeit erhöht wird.

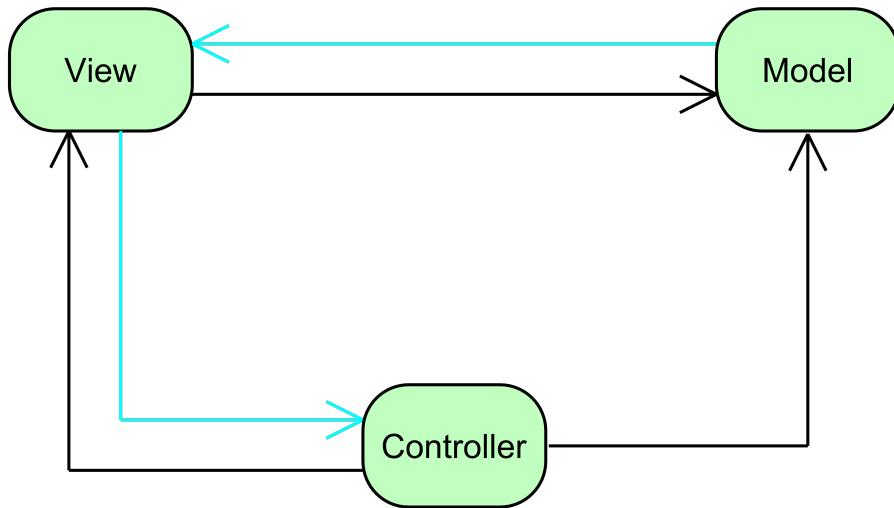


Abbildung 2.2.: Diagramm – MVC-Entwurfsmuster: Die blauen Pfeile stellen indirekte Beziehungen dar, welche meist durch die Nutzung eines Eventsystems auf Basis des Beobachtermusters implementiert werden.

Die klassische Struktur vom MVC Muster³ ist, wie in Abbildung 2.2 zu sehen, in drei Komponenten unterteilt [Dea95]:

Das **Model** beinhaltet alle Klassen, welche für die Logik und die Datenhaltung der Anwendung verantwortlich sind und ist vollständig unabhängig vom View.

Der **View** stellt die Anwendung visuell dar, ermöglicht eine Interaktion mit dieser und ist in den meisten Applikationen äquivalent zu einer grafischen Benutzeroberfläche.

Der **Controller** ist für das Verändern des Views verantwortlich. So werden beispielsweise Interaktionen mit der Benutzeroberfläche, wie ein Schaltflächenklick im Controller, verarbeitet, was wiederum den View aktualisiert.

²Je nach Interpretation, kann es sich bei dem MVC Muster auch um ein Architekturmuster handeln, bei welchem Model, View und Controller als Schichten einer Architektur gesehen werden und nicht direkt als beispielsweise Klassen implementiert werden.

³In der Literatur sind viele Interpretationen des MVC Musters beschrieben. Eine strikte Unterteilung der drei Komponenten ist jedoch immer vorhanden.

Dabei kennt der View das Model, das Model aber nicht direkt den View. Der Controller verwaltet sowohl den View als auch das Model.

2.2. JavaFX

JavaFX ist eine auf Java basierende, quelloffene Bibliothek für das Entwickeln von grafischen Benutzungsschnittstellen für Client Applikationen. Im Vergleich zum Vorgänger GUI-Toolkit Java-Swing, bietet JavaFX ein modernes, zeitgemäßes Design der allgemeinen Benutzeroberfläche sowie den dort enthaltenen Schaltflächen und Komponenten [Sha15]. Kombiniert mit den objektorientierten Konzepten von Java, ist JavaFX in der Lage auch komplexe nebenläufige Anwendungen mit vielen Abhängigkeiten darzustellen und aufgrund der Plattformunabhängigkeit auch ohne viele Restriktionen in allen bekannten Betriebssystemen einsetzbar.

Dazu ist JavaFX auch weitgehend konform mit bekannten Entwurfsmustern der Softwareentwicklung wie beispielsweise dem MVC⁴- oder dem Beobachtermuster, weshalb implementierte Anwendungen selbst bei vielen Lines of Code (LoC), eine grundsätzlich hohe Strukturiertheit auf Quelltextebene aufweisen. Durch die Verwendung von Properties und Bindings (siehe Unterabschnitt 2.2.2) ist es auch möglich das Model-View-ViewModel (MVVM) Muster zu nutzen. Das grafische Layout kann dabei nicht ausschließlich durch Java-Quelltext, sondern auch mittels der an die XML angelehnte Markup-Sprache FXML erstellt werden. Letzteres kann durch externe Tools wie dem Scene-Builder enorm vereinfacht werden [VCG⁺18].

2.2.1. Aufbau und Szenengraph

Damit eine JavaFX-Anwendung als solche identifiziert werden kann, muss die Hauptklasse von der Application-Klasse erben. Die Namensgebung der Klassen, welche für die Struktur bzw. den Aufbau einer JavaFX-Anwendung zuständig sind, basiert auf Begriffen der Theaterumgebung [AA19]:

Die **Stage** Klasse repräsentiert ein Anwendungsfenster, welches das Design des Fensterlayouts des aktuell genutzten Betriebssystems verwendet. Eine Stage ist teilweise modifizierbar; so können beispielsweise die Standardschaltflächen in der Titelleiste entfernt oder deaktiviert werden. Werden mehrere Fenster benötigt, dann können nach dem Initialisieren der Haupt-Stage durch die JavaFX-Plattform manuell weitere hinzugefügt werden.

Die **Scene** Klasse ist für das Layout und die Darstellung von vorhandenen oder selbst erstellten JavaFX-Komponenten verantwortlich. Jede Komponente, welche durch eine Scene-Instanz angezeigt und verwaltet werden soll, wird in einer hierarchisch angeordneten, objektorientierten Datenstruktur eingefügt,

⁴Die Beschreibung des MVC Musters aus Abbildung 2.2 ist nicht vollständig auf die JavaFX Situation anzuwenden, da der View meistens in eine FXML-Datei ausgelagert wird und deshalb nicht unbedingt mit dem Model interagiert.

welche in der Computergrafik als Szenengraph bekannt ist [HvDM⁺13]. Jeder Stage muss zwangsläufig eine Scene zugewiesen werden.

Die **Node** Klasse ist eine darstellbare Komponente im Szenengraphen wie beispielsweise eine Schaltfläche oder ein Containerelement. Node Instanzen im Szenengraph können Kindelemente enthalten und maximal einem Elternelement zugeordnet sein. Der Szenengraph ähnelt somit einer Baumstruktur mit einer Wurzel und einem oder mehreren Blättern. Damit eine Node-Instanz Kindelemente besitzen darf, muss diese immer von der abstrakten Parent-Klasse erben. Das Layouting und die Positionierung im lokalen Koordinatensystem wird bei vorhandenen Kindelementen immer durch das Elternelement kontrolliert. Jede darzustellende Komponente muss von der Node-Klasse erben [Jun13].

Ein minimales Beispiel für eine voll funktionsfähige JavaFX-Anwendung, welche das Zusammenspiel der oben genannten Konzepte und Klassen widerspiegelt, ist in Code 2.1 dargestellt.

```

public class TestApplication extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        ① final Pane root = new Pane();
        ① root.getChildren().add(new Button("TestButton"));
        ② final Scene scene = new Scene(root, 250, 250);
        ② primaryStage.setScene(scene);
        primaryStage.show();
    }
}

```



Code 2.1: Beispiel – Minimale JavaFX-Anwendung

Nach der Initialisierung der JavaFX Anwendung wird in ① das Wurzelement des Szenengraphen erstellt und um eine Button Instanz erweitert. Dieses wird dann in ② zu einem Scene Objekt hinzugefügt, welches wiederum als Szene der primaryStage dient. Das Beispiel wird mit dem Anzeigen der Stage beendet.

2.2.2. Properties und Bindings

JavaFX besitzt eine auf dem JavaBeans-System und dem Observer-Entwurfsmuster basierende API, welche es dem Programmierer ermöglicht, eine synchronisierende

```

Button btn = new Button("Test");
// ChangeListener für den Schaltflächentext
btn.textProperty().addListener((obs, oVal, nVal) -> {
    System.out.println(nVal);
});
// EventHandler für die Schaltflächenaktivierung
btn.setOnAction(e -> {
    System.out.println("Clicked!");
})

```

Code 2.2: Beispiel – ChangeListener & EventHandler

Beziehung zwischen zwei oder mehr Variablen zu erstellen. Im Rahmen von JavaFX werden diese Variablen auch Properties genannt. Wird eine Property in einer solchen Beziehung geändert, so wird automatisch auch die andere geändert [Hom13]. Dabei ist es auch möglich, Event Listener für eigene oder durch von JavaFX-Nodes automatisch erzeugte Properties zu registrieren. Soll beispielsweise Quelltext ausgeführt werden, wenn eine Änderung eines Wertes einer Property festgestellt wird, so kann dies mit dem Erstellen einer ChangeListener-Instanz durchgeführt werden [Gao19]. Im ersten Teil von Code 2.2 soll der Text einer Schaltfläche bei einer Änderung auf die Konsole ausgegeben werden. Dazu wird mittels Lambdaausdruck ein neuer ChangeListener mit der `textProperty-StringProperty` der Schaltfläche verknüpft.

Des Weiteren unterstützt JavaFX ein Event-System, welches anhand von verschiedenen Aktionen Events durch den Szenengraphen propagiert. Ein solches Event wird beispielsweise durch das Eintragen von Text in ein Textfeld oder das Aktivieren einer Dropdown-Liste ausgelöst. In zweiten Teil von Code 2.2 wird ein EventHandler für das Aktivieren einer Schaltfläche erstellt.

2.2.3. Layouting: FXML vs. Quelltext

Wie in der Einleitung schon angedeutet, ist es möglich das Layout der Anwendung auch per FXML zu erstellen. Eine Prävention von Boilerplate-Code kann durch das Auslagern von häufig verwendeten JavaFX-Komponenten in externe FXML-Dateien erfolgen [KDSAMR18]. Das Verwenden solcher Dateien sorgt für eine bessere Trennung von Controllern und Logik im Sinne des MVC-Entwurfsmusters [Jun13] und durch die hohe Konfigurierbarkeit sind für eine eventuelle Veröffentlichung der Applikation wichtige Konzepte wie die Internationalisierung leichter umzusetzen [Ste14]. Durch das Parsen und Aufbauen des Szenengraphen zur Laufzeit des Programms ist eine Verwendung von FXML-Dateien jedoch langsamer als benötigte Komponenten direkt im Java Quelltext zu deklarieren. Fast alle JavaFX-Nodes können ohne Weiteres in XML-Elementen verwendet und angepasst werden. Außerdem ist es möglich, direkt eine manuell erstellte Controller-Klasse mit einer FXML-Datei zu assoziieren. Das Laden einer FXML-Datei und das darauffolgende Aufbauen des Szenengraphen wird durch die `FXMLLoader`-Klasse durchgeführt.

Das Layouting-Beispiel aus Code 2.1 ist als eine funktionsgleiche FXML Variante in Code 2.3 zu erkennen. Das Laden der Datei wird durch das Instanziieren eines neuen FXMLLoader Objekts, wie in Code 2.4 dargestellt, ermöglicht. Um eine Controller-Klasse mit der FXML-Datei zu assoziieren, kann das Wurzelement dieser durch das `fx:controller` Attribut erweitert werden. Der Name des Controllers ist hierbei der voll qualifizierte Klassename. Neben externen FXML-Dateien können auch externe Cascading Style Sheets (CSS)-Dateien für das Design des Layouts verwendet werden. In Anhang A ist ein vollständig kompilierbares JavaFX-Programm, welches aus einem Controller, einer FXML-Datei sowie einer CSS-Datei aufgebaut ist, zu finden.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.Pane?>
<?import javafx.scene.control.Button?>

<Pane xmlns="http://javafx.com/javafx">
    <Button>TestButton</Button>
</Pane>
```

Code 2.3: Beispiel – FXML Layouting

```
Pane load(String fxmlPath) throws IOException {
    return new FXMLLoader(getClass().getResource(fxmlPath)).load();
}
```

Code 2.4: Beispiel – FXML Ladeprozess

2.3. Java-Annotationen

In der Sprachwissenschaft sind Annotationen eine Möglichkeit, einen vorhandenen Text mit Anmerkungen für beispielsweise Disambiguierung zu versehen, also zum Eliminieren von Mehrdeutigkeiten eines Wortes oder für das Erklären von komplexen Textabschnitten. Sie geben dem Leser Zusatzinformationen um Sachverhalte einfacher darzustellen und sorgen dadurch für ein schnelleres bzw. besseres Verständnis des Textes [Lem15]. Dabei sind solche Anmerkungen kein Hauptbestandteil von Texten, sondern dienen ausschließlich als Ergänzung.

In der Informatik sind Annotationen ebenfalls nur ein deskriptives Strukturkonzept, welches es dem Entwickler ermöglicht, verschiedenen strukturellen Elementen der Programmierung (wie Felder oder Klassen) Metadaten zuzuweisen [YBSM19]. Das Nutzen von Annotationen in Anwendungen ist aufgrund ihrer meist simpel gehaltenen Syntax auch für Programmieranfänger vorteilhaft und durch ihre Anpassungsfähigkeit und Flexibilität sind sie in vielen Bibliotheken und Programmiersprachen vertreten.

2.3.1. Definition

Annotationen wurden mit Java 5 (2014) in die Sprache eingeführt und werden seitdem immer häufiger für verschiedene Aspekte der Programmierung genutzt [RV11]. Mit ihnen kann eine Steuerung des Compilers erfolgen, eine Verarbeitung der Metadaten zur Kompilierzeit durchgeführt werden oder das Verhalten von Anwendungen zur Laufzeit modifiziert oder gelenkt werden [YBSM19]. Aufgrund der Tatsache, dass es sich nur um rein deskriptive Metadaten handelt, ist es Annotationen nicht direkt möglich mit existierendem Quelltext zu interagieren. Möglichkeiten zur Verarbeitung dieser Metadaten werden in Unterabschnitt 2.3.3 vorgestellt. Neben den von Java vordefinierten Annotationen wie zum Beispiel `@Override` für das Überschreiben von vererbten Methoden oder `@SuppressWarnings` für das Unterdrücken von Compilerwarnungen, können auch eigene Annotationen deklariert werden.

Es handelt sich bei Annotationen in Java um spezialisierte Schnittstellen bei welchen das `interface`-Schlüsselwort durch ein @-Zeichen Präfix zu `@interface` erweitert wird [GJS⁺05]. Außerdem ist es Annotationen nicht erlaubt wie bei normalen Schnittstellendefinitionen das Schlüsselwort `extends` für eine Vererbung zu verwenden, da die Superschnittstelle implizit vom Compiler auf die Annotation Klasse des `java.lang.annotation` Pakets gesetzt wird [Ora17]. Ein Beispiel einer Annotationsdefinition ist in Code 2.5 dargestellt.

```
public @interface TestAnnotation {
    // ...
}
```

Code 2.5: Beispiel einer Annotationsdefinition

In der Analogie des Kapitels 2.3 können Elemente mit strukturgebendem Charakter wie Bestandteile eines Satzes annotiert werden. Parallel dazu sind in der Java-Programmierung Klassen, Methoden, Felder etc. für die Strukturierung des Quelltextes und der Softwarearchitektur verantwortlich und somit auch mit Annotationen erweiterbar. Um Sprachelemente zu annotieren muss wie in Code 2.6 dargestellt, ein @-Präfix zum eigentlichen Klassennamen hinzugefügt werden.

```
@TestAnnotation
public class TestClass {
    // ...
}
```

Code 2.6: Beispiel einer annotierten Klasse

Aufgrund der besonders einfachen Syntax und dem vergleichsweise geringen Aufwand ist ein steigender Trend der Nutzung von Java-Annotationen in Open-Source Anwendungen zu erkennen. Werden Annotationen jedoch übermäßig verwendet kann es schnell zu Quelltext-Verschmutzung kommen, was im Kontext der An-

notationsprogrammierung auch „annotation hell“ (dt. Annotationshölle) genannt wird. Annotationen erreichen dann das Gegenteil des gewünschten Zwecks – statt den Entwicklungsprozess vereinfachend zu unterstützen, wird der Quelltext schwer nachvollziehbar und wirkt unstrukturiert und unübersichtlich.

Dennoch zeigt eine Studie aus dem Jahre 2019, welche 1094 quelloffene GitHub-Projekte auf die Verwendung von Annotationen untersucht hat, dass javabasierte Anwendungen und Bibliotheken bei aktiver Nutzung von Annotationen eine geringere Fehleranfälligkeit aufweisen [YBSM19].

2.3.2. Syntax

Annotationen können Attribute besitzen, welche zur Komplizierzeit bzw. Laufzeit ausgelesen werden können. Die Typen dieser Attribute sind nicht vollständig frei wählbar. So ist es beispielsweise nicht möglich ein Attribut vom Typ `Object` in einer Annotation zu kapseln, ohne einen Komplizierfehler auszulösen. Erlaubt sind alle primitiven bzw. atomaren Datentypen und Instanzen der `String`-, `Class`- und `Enum`-Klasse sowie eindimensionale Arrays aus den vorherigen Typen. Außerdem ist es möglich, Attributen einen voreingestellten Wert mittels des Schlüsselwortes `default` zuzuweisen [GJS⁺05]. Annotationen müssen in einer der folgenden Syntaxen benutzt werden:

Normal Annotations sind ganz normal deklarierte Annotationen, bei welchen die Attribute mittels Aufzählung in Klammern übergeben werden.

```
public @interface Entity {
    String name();
    int id();
}
```

Code 2.7: Deklaration – Normal Annotation

```
@Entity(name="test", id=2)
public class TestEntity {
    // ...
}
```

Code 2.8: Anwendung – Normal Annotation

Single-Element Annotations sind eine Kurzform der normalen Annotationen mit einem `value`-Attribut und keinen weiteren nicht-default Attributen.

```
public @interface Entity {
    String value();
    int id() default -1;
}
```

Code 2.9: Deklaration – Single-Element Annotation

```
@Entity("test")
public class TestEntity {
    // ...
}
```

Code 2.10: Anwendung – Single-Element Annotation

Marker Annotations sind ebenfalls eine Kurzform der normalen Annotationen mit keinen oder nur default Attributen.

```
public @interface Entity {
    String name() default "";
    int id() default -1;
}
```

Code 2.11: Deklaration – Marker Annotation

```
@Entity
public class TestEntity {
    // ...
}
```

Code 2.12: Anwendung – Marker Annotation

Die Sichtbarkeit von eigenen Annotationen zu verschiedenen Phasen des Codezyklus kann durch die von Java bereitgestellte Annotation `@Retention` gesteuert werden. Das übergebene Enum-Attribut klassifiziert die Annotation dann in einen von drei Typen [RV11]:

Quellcode-Annotationen sind nur beim Kompiliervorgang auslesbar und können dem Compiler Anweisungen geben oder mithilfe von Annotation-Prozessoren zum Beispiel neue Klassen automatisch generieren. Sie sind in der kompilierten Java-Anwendung nicht mehr enthalten.

Klassen-Annotationen sind nach dem Kompilierungsprozess noch in der Anwendung enthalten und können durch externe Tools wie zum Beispiel dem Code-Obfuscator ProGuard⁵ ausgelesen werden.

Laufzeit-Annotationen sind nach der Kompilierung und beim Start der Anwendung enthalten und können dann mithilfe der Reflection-API zur Laufzeit ausgewertet werden.

Des Weiteren kann gesteuert werden, welche Typen der Strukturelemente eines Quellcodes annotiert werden können. Ein Beispiel für eine zur Laufzeit beibehaltene Annotation, welche nur an Methoden angebracht werden kann ist in Code 2.13 zu erkennen.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Event {
    int id();
    int priority() default 0;
}
```

Nur an Methoden

Zur Laufzeit

Code 2.13: Beispiel einer Laufzeit Annotation

⁵ProGuard: <https://www.guardsquare.com/proguard>

2.3.3. Auswertung von Annotationen zur Laufzeit

Für eine Auswertung von Laufzeit-Annotationen, muss zwangsläufig die Reflection-API von Java genutzt werden. Wenn eine Programmiersprache eine Form von Reflection (dt. Spiegelung) aufweist, ist es möglich Attribute, Logikfluss und andere Eigenschaften während der Laufzeit zu ändern. In objektorientierten Sprachen wie Java wird diese „computational reflection“ genutzt, um die Möglichkeit einer Selbstbeobachtung der eigenen Sprachelemente zu schaffen [LTX17]. Die API ermöglicht somit beispielsweise das Auslesen von Laufzeit-Annotationen und deren deklarierte Attribute oder das dynamische Instanziieren von Klassen [FFV04]. Jedes Java-Element der Reflection API (Feld, Methode, Klasse, ...), welches annotierbar ist, wird durch das Erben von der `AnnotatedElement`-Klasse als solches klassifiziert [Sch19]. Damit alle vorhandenen Annotation ausgelesen werden können, kann die Methode `AnnotatedElement#getDeclaredAnnotations` aufgerufen werden [PN15]. Das Lesen der Attribute der in Code 2.13 vordefinierten Annotation ist in Code 2.14 zu erkennen.

```
if(Test.class.isAnnotationPresent(Event.class)) {
    Event e = Test.class.getDeclaredAnnotation(Event.class);
    int id = e.id();
    int priority = e.priority();
}
```

Code 2.14: Auslesen einer Laufzeit-Annotation

2.3.4. Auswertung von Annotationen zur Kompilierzeit

Das Auswerten von Annotationen zur Kompilierzeit kann mithilfe der Annotation-Processing API seit Java 6 durchgeführt werden. Annotationsprozessoren müssen von der `AbstractProcessor` Klasse erben und durch META-INF Metadaten mit der `ServiceLoader` Klasse registriert werden. Annotationsprozessoren müssen im Java Archiv-Format vorliegen und werden automatisch durch `javac` erkannt, wenn diese im Build-Pfad der eigentlichen Applikation präsent sind. Durch die Nutzung der von Google entwickelten `AutoService`⁶ Bibliothek müssen benötigte Metadaten nicht manuell im META-INF Ordner des Java Archivs hinterlegt werden, sondern werden automatisch erstellt, verwaltet und bei Bedarf aktualisiert. Die Struktur eines Annotationsprozessors mit der `AutoService` Bibliothek ist in Code 2.15 dargestellt und verarbeitet alle gefundenen Annotationen aufgrund des Wildcard-Zeichens in der `@SupportedAnnotationTypes` Annotation. Das Erstellen von Klassen mithilfe der von Java zur Verfügung gestellten APIs ist ein aufwendiger Prozess, da Quelltext manuell durch zum Beispiel `PrintWriter` Instanzen generiert werden muss. Außerdem ist es nicht möglich bereits vorhandene Klassen zu modifizieren. Ein Hinzufügen von Methoden und Feldern ist ausgeschlossen und der generelle Overhead beim Verwaltungs- und Erstellungsprozess ist nicht

⁶Google AutoService: <https://github.com/google/auto/tree/master/service>

nur für den Entwickler zeitaufwändig, da der Anwender ebenfalls die verwendeten Annotationsprozessoren registrieren muss. Letzteres kann durch das Verwenden von Build-Management-Tools wie Apache Maven⁷ verhindert werden und die Quelltext-generation kann durch externe Bibliotheken wie JavaPoet⁸ deutlich erleichtert werden. Um bereits vorhandene Klassen modifizieren oder erweitern, muss eine Form der Bytecode Manipulation genutzt werden. Dazu kann beispielsweise die ASM⁹ Bibliothek verwendet werden, welche es dem Entwickler ermöglicht existierende Klassen, Methoden oder Felder vollständig zu verändern [Kul07].

```

@SupportedAnnotationTypes ("*")
@AutoService(Processor.class)
public class TestProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> elems,
        RoundEnvironment env) {
        // ...
        return true;
    }
}

```

Code 2.15: Beispiel – Annotationsprozessor

2.4. Zusammenfassung

Nachdem die für das Verständnis dieser Arbeit benötigten Grundlagen wie verschiedene Entwurfsmuster, JavaFX und Java Annotationen eingeführt wurden, kann im nächsten Kapitel der aktuelle Stand der Technik vorgestellt werden. Das folgende Kapitel knüpft an das Annotationskonzept an und präsentiert verschiedene auf Annotationen basierende Implementierungen in der Java-Umgebung sowie in anderen Programmiersprachen und externen Bibliotheken.

⁷Apache Maven: <https://maven.apache.org>

⁸JavaPoet: <https://github.com/square/javapoet>

⁹ASM: <https://asm.ow2.io>

3. Stand der Technik

In diesem Kapitel werden aktuelle Konzepte und Implementierungen der Annotatonsprogrammierung zur Vereinfachung des Entwicklungsprozesses einer Anwendung dargelegt. Obwohl der primäre Fokus dabei auf die JavaFX- und die generelle Java-Umgebung gelegt wird, werden dennoch auch Bibliotheken und mögliche Strukturen aus anderen Programmiersprachen herangezogen.

3.1. Annotationen im Umfeld von JavaSE/JavaEE/JavaFX

Nach dem Einführen von Annotationen in Java vor 16 Jahren haben sich viele Bibliotheken etabliert, welche fast vollständig oder teilweise auf dieses Konzept setzen. Eine Studie aus dem Jahre 2011, welche 106 Systeme auf die Nutzung von Annotationen untersuchte, stellte fest, dass 41 dieser keine einzige aufwiesen [RV11]. Acht Jahre später wurde eine ähnliche Studie veröffentlicht, welche 1094 populäre Systeme untersucht hat und feststellte, dass jedes dieser Systeme mindestens eine Annotationen enthält [YBSM19]. Auch wenn bei beiden Studien nicht dieselben Systeme getestet worden sind, ist dennoch ein klarer Trend nach oben zu erkennen. Dazu wurden eine Vielzahl an Werken publiziert, welche mithilfe von Annotationen vorhandene Java-Konzepte vereinfachen und erweitern sollen.

Beispielsweise wurde ein System entwickelt, welches durch Semantikinformationen von annotierten JavaDoc-Elementen, das Refactoring automatisiert und den Entwickler auf das Nutzen von Entwurfsmustern und etwaige Refactoring-Operationen hinweisen soll [MP06]. Des Weiteren werden Annotationen im Kontext der automatischen Nebenläufigkeit [DPV⁺07], dem Erstellen von Parsern für Programmiersprachen [PFS09] und der Dokumentation sowie der Erzeugung von Quelltext genutzt [SNP16, MJ09]. Im Folgenden werden Beispiele gegeben, welche die Programmierung durch die Verwendung von Annotationen aktiv vereinfachen.

3.1.1. JavaSE Umgebung und externe Bibliotheken

Die am häufigsten genutzte, durch das Java Development Kit (JDK) vordefinierte Annotation (siehe Unterabschnitt 2.3.1), ist die Quelltextannotation `@Override` [RV11], welche für eine Bugprävention genutzt werden kann. Möchte der Entwickler beispielsweise eine Methode einer Superklasse überschreiben und übernimmt nicht die vorgegebene Methodendeklaration, sondern verwendet fälschlicherweise eine Methodenüberladung, so handelt es sich häufig dennoch um vollständig validen Quelltext, welcher aber unter Umständen zu einem ungewollten Verhalten führt. Auch kann das fehlerhafte Überschreiben von Methoden aus einer

Verwechslung von ähnlichen Methodennamen resultieren. Beispielsweise kann beim Erben von der Container Klasse aus dem Abstract Window Toolkit (AWT) die `paintComponents` mit der `paintComponent` Methode aus der `JComponent` Klasse verwechselt werden. Wird aber die `@Override` Annotation in solchen Fällen über die zu überschreibenden Methoden geschrieben, so wird immer ein Kompilierfehler erzeugt. Ein Beispiel, welches ersteres Szenario verbildlicht ist in Code 3.1 und Code 3.2 dargestellt.

```
interface Test {  
    default void t(int... i) {}  
}
```

Code 3.1: Beispiel – Interfacedeklaration

```
class TestClass implements Test {  
    @Override Kompilierfehler  
    public void t(int i) {}  
}
```

Code 3.2: Beispiel – Kompilierfehler

Durch das Nutzen von externen Bibliotheken können weitere Funktionalitäten durch Annotationen hinzugefügt werden. Wie in Unterabschnitt 2.3.2 ausführlich erklärt, können neue Klassen durch Annotationsprozessoren zur Kompilierzeit erstellt und wiederholende Quelltextausschnitte wie Getter und Setter dadurch automatisch generiert werden. Basierend auf diesen Möglichkeiten, wurde das Projekt Lombok¹ konstituiert, welches das Ziel verfolgt, den Entwicklungsprozess durch das Erstellen von Boilerplate-Code mithilfe einer ausschließlichen Nutzung von Annotationen zu erleichtern. Lombok ist in der Lage die obligatorischen `equals()`- und `hashCode()`-Methoden zu erstellen, was nicht nur in einer hohen Zeiteinsparung resultiert, sondern auch mögliche Bugs bei dem manuellen Implementieren dieser Methoden verhindert. In Code 3.3 ist ein POJO zu erkennen, bei welchem der Konstruktor, alle Getter und die `equals()`-, `hashCode()`- und `toString()`-Methoden automatisch generiert werden. Auch Software-Plattformen für mobile Endgeräte wie Android² setzen auf Annotationstechnologien: Damit das Minimierungs-/Optimierungs-Tool von Android keine fälschlicherweise als unbunutzt erkannten Klassen beim Build-Vorgang entfernt, kann die `@Keep`-Annotation verwendet werden. Dazukommend kann die Erweiterung `support-annotations`³ einem Android-Projekt hinzugefügt werden, um beispielsweise zu überprüfen, ob Methoden in einem bestimmten Thread ausgeführt werden, ob Einschränkungen für Methoden- oder Konstruktorparameter eingehalten werden und ob bestimmte Berechtigungen für das Ausführen von Methoden vorhanden sind.

¹Project Lombok: <https://projectlombok.org>

²Android: <https://www.android.com>

³Support-Annotations: <https://developer.android.com/studio/write/annotations>

```

@Getters
@Setter
@ToString
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@RequiredArgsConstructor
public static final class Player {

    @EqualsAndHashCode.Include
    private final UUID id;
    private final String name;
    private final Date regDate;

}

```

Code 3.3: Beispiel – Lombok POJO

3.1.2. JavaEE/JakartaEE Umgebung

Auch bei der Entwicklung von auf Java basierten Enterprise-Anwendungen werden Annotationen verwendet. Die JakartaEE Spezifikation der Version 9⁴ bietet verschiedene Bibliotheken, welche mithilfe von Annotationen verschiedene Prozesse erleichtern können. Beispielsweise existiert die Jakarta XML Binding (JAXB) Programmierschnittstelle, welche das Binden von XML Dokumenten und Java Objekten ermöglicht – Ein Java Objekt kann dann durch ein XML Dokument repräsentiert und zur Laufzeit des Programms daraus erstellt werden (und umgekehrt). Wird zu der eigentlichen JAXB Bibliothek noch die Erweiterung aus dem jakarta.xml.bind.annotation Paket genutzt, so ist es möglich die vorher genannte Bindung vollständig durch die Nutzung von Annotationen zu realisieren. Ein Beispiel für die Repräsentation eines Java Objektes durch eine XML Datei ist in Code 3.4 und Code 3.5 abgebildet.

```

<?xml version="1.0"?>
<Cat id="1">
    <color>Brown</color>
    <age>4</age>
</Cat>

```

Code 3.4: Repräsentation als XML Datei

```

@RootElement
public class Cat {

    @XmlAttribute
    public int id;

    public String color;
    public int age;
}

```

Code 3.5: Repräsentation als Java Objekt

⁴JakartaEE 9: <https://jakarta.ee/specifications/platform/9/apidocs/>

3.1.3. JavaFX Umgebung

In JavaFX direkt werden nur wenige Annotationen verwendet, welche Teil der öffentlichen API sind. Dazu gehört `@FXML`, welche für das automatische Setzen von Feldern oder für die Identifikation von Methoden für EventHandler benötigt wird [AA19]. Der Entwickler kann somit Events, welche durch JavaFX-Komponenten ausgelöst werden, per FXML-Datei mit Methoden im selben Controller verbinden. Dazu wurden Bibliotheken wie Afterburner.fx⁵ entwickelt, welche durch `@Inject`, das Inversion of Control (IoC) Programmierparadigma durch Abhängigkeitsinjektion realisiert oder das von CERN entwickelte ExtJFX⁶, welches `@RunInFxThread` nutzt, um Unitests auf dem JavaFX-Thread auszuführen.

3.2. Annotationen in anderen Programmiersprachen

Neben den Java-Annotationen, welche in Abschnitt 2.3 erklärt und in Abschnitt 3.1 vorgestellt wurden, werden Annotationen auch in vielen anderen Programmiersprachen genutzt. In den folgenden Abschnitten wird explizit auf das Annotationskonzept in Python und C# eingegangen.

3.2.1. Verwendung in Python

Python ist eine dynamisch typisierte Sprache und validiert somit den Typen einer Variablen zur Laufzeit des Programms [Tra09], kann aber durch das Verwenden von Funktionsannotationen Meta-Daten zu Parametern, Variablen und Funktionsrückgabewerten hinzufügen, um so den gewünschten Typen anzudeuten [vRLL14, WL06]. Diese Annotationen werden zwar vom Python-Interpreter ignoriert, können aber durch Softwaresysteme von Drittanbietern wie mypy zur statischen Typisierung verwendet werden. Nach einer Studie von Khan et al., welche 210, auf Python basierende GitHub-Projekte auf typebezogene Fehler untersuchte, konnten 15% der gefundenen Mängel, durch mypy verhindert werden [KCVM21]. Einige Entwicklungsumgebungen wie PyCharm⁷ sind außerdem in der Lage, Warnungen bei eventuellen Verletzungen der Typempfehlungen von Annotationen anzuzeigen [Rot17]. Das Verwenden von derartigen Annotationen kann somit durchaus die Fehleranfälligkeit von Programmcodeelementen in Python sinken – wenn auch nur implizit durch externe Bibliotheken oder Entwicklungsumgebungen.

3.2.2. Verwendung in C# und .NET

In C# wird das Hinzufügen von Meta-Informationen zu bestehenden Programmlementen durch Attribute realisiert [AJ19]. Mithilfe dieser Attribute können dann

⁵Afterburner.fx: <https://github.com/AdamBien/afterburner.fx>

⁶ExtJFX: <https://github.com/extjfx/extjfx>

⁷PyCharm: <https://www.jetbrains.com/de-de/pycharm/>

beispielsweise Klassen als serialisierbar deklariert werden oder Methoden und Funktionen für nicht verwaltete Dynamic Link Librarys (DLLs) erreichbar gemacht werden. Es ist, ähnlich wie in Java, auch möglich, eigene Attribute zu erstellen und diese zu unterschiedlichen Phasen wie zur Kompilierzeit oder Laufzeit auszuwerten. Durch die einfache Nutzung der Attribute wurde beispielsweise eine Erweiterung der grundlegenden *C#*-Sprache entwickelt, welche ein Parallelisieren von sequentiellen Programmausschnitten ermöglicht [CCC05].

3.3. Fazit

Annotationen haben in den meisten Fällen einen positiven Einfluss auf die Entwicklung von beliebigen Anwendungen. Annotationen sind keine universelle Sprachstruktur wie Schleifen und bedingte Anweisungen, werden aber immer öfter in Programmiersprachen eingesetzt. Sie sorgen, wenn nicht übermäßig verwendet, für übersichtlichere und kompaktere Klassen, was wiederum in einem besseren Verständnis des Quelltextes resultiert. Einige Bibliotheken verfolgen das Ziel dem Entwickler eine große Zeiteinsparung durch einen gewissen Grad an Automation zu ermöglichen. Dabei wird repetitiver oder fehleranfälliger Quelltext vollständig oder teilweise automatisch generiert und auf das Einhalten von bekannten Entwurfsmustern und Sprachkonventionen geachtet. Komplexe Prozesse wie das Parsen von XML Dateien, Abhängigkeitsinjektion oder objektrelationale Abbildungen in relationale Datenbanken können durch Annotationen vereinfacht werden. Dennoch existieren keine annotationsbasierten Bibliotheken für die aktuellste Java-Version (16), welche explizit einen vereinfachenden Einfluss auf den Entwicklungsprozess von auf JavaFX aufbauenden Applikationen nehmen und dabei quelloffen und frei verfügbar sind. JavaFX wird in den meisten Fällen durch Features erweitert, welche vorher in der Form noch nicht in der eigentlichen Bibliothek vorhanden sind. Dazu zählt beispielsweise ExtJFX, welche im Kern eine auf JUnit⁸ aufbauende Testumgebung für grafische Benutzungsoberflächen ist. Ein System welches auf eine Vereinfachung oder Automatisierung von komplizierten JavaFX Funktionen (Controller-Verwaltung, Properties und Bindings, Animationen, ...) durch Annotationen abzielt ist nicht vorhanden.

Eine Erweiterung des XML-Schemas für die Struktur einer FXML-Datei, um beispielsweise die Beziehung zwischen mehreren Controllern auszudrücken, ist aufgrund der restriktiven Implementierung der `FXMLLoader`-Klasse nur schwer umzusetzen. Daraus resultiert, dass die Nutzung von eigenen Annotationen welche wie `@FXML`, eine Interaktion zwischen in der FXML-Datei definierten Elementen und der eigentlichen Controller-Klasse ermöglichen, mit den Bordmitteln von JavaFX nicht möglich ist.

⁸JUnit 5: <https://junit.org/junit5/>

4. Konzeption und Entwurf

In diesem Kapitel werden mögliche Probleme bei der Entwicklung sowie bei der Nutzung von JavaFX Anwendungen identifiziert. Dabei wird ein besonderer Fokus auf das Finden von Architekturmängeln, fehlenden Funktionalitäten und verbessigungswürdigen Techniken gelegt. Um eine Fehleranfälligkeit zu reduzieren, sollen komplexe und sich häufig wiederholende Quelltextbausteine automatisch erstellt oder durch Annotationen vereinfacht werden. Die vollständige Substitution eines aufwendigen Prozesses ist dabei ebenfalls möglich. Probleme, Vereinfachungen oder Verbesserungen sollen durch das Untersuchen von vorhandenen, quelloffenen JavaFX-Projekten und Bibliotheken gefunden werden. Auch sollen Ideen und Konzepte zusammengetragen werden, welche auf JavaFX anwendbar sind, jedoch nur in anderen Bibliotheken und Frameworks aufzufinden sind.

Bei der Problemanalyse wird stets das Ziel verfolgt, das Entwickeln mit JavaFX zu vereinfachen – besonders für noch unerfahrene Entwickler. Danach wird eine Anforderungsanalyse durchgeführt, mit welcher systematisch funktionale sowie nicht-funktionale Anforderungen auf der Basis der gefundenen Probleme erstellt werden. Auf die Anforderungserhebung folgt die Konzeption des benötigten Systems und der zugrundeliegenden Architektur. Essentielle Komponenten werden mit UML Diagrammen entworfen und im Detail erläutert. Bei der Existenz verschiedener Lösungsstrategien für ein Problem, wird jede Strategie einzeln beleuchtet und nach Kriterien wie Sinnhaftigkeit und Machbarkeit entschieden, welche für das System am besten geeignet ist. Wichtige Richtlinien wie die angestrebte Softwarequalität werden ebenfalls beschrieben.

4.1. Identifikation von Problemen und komplexen Strukturen in der JavaFX Entwicklung

Im Folgenden werden generelle Probleme bei der Entwicklung von JavaFX Anwendungen identifiziert. Dazu gehören Mechanismen, welche aufgrund ihrer Komplexität nicht für Anfänger geeignet sind oder von erfahrenen Entwicklern häufig genutzt und somit möglicherweise vereinfacht werden können. Obwohl dabei Annotationen als Basis für eine Vereinfachung dienen, wird auch das Erstellen von zusätzlichen Klassen oder das Entwickeln von Erweiterungen für existierende JavaFX Konzepte als Alternative für diese Zielerreichung in Betracht gezogen. Die Lösungen der gefundenen Probleme werden in einer Anforderungsanalyse durch funktionale und nichtfunktionale Anforderungen in Abschnitt 4.2 entwickelt.

4.1.1. Internationalisierung und Lokalisierung

In der Informatik, speziell in der Softwareentwicklung, ist die Internationalisierung ein wichtiger Bestandteil eines Softwareproduktes, bei welchem die Entwickler die Software so gestalten, dass diese ohne viel Aufwand für andere internationale Märkte mit anderen Kulturen und Sprachen verfügbar gemacht werden kann [Rei05]. Dabei wird beispielsweise eine einfache Schnittstelle für das Verwenden von verschiedenen Sprachen entwickelt, welche das Übersetzen von vorhandenen Textfeldern und anderen textbasierten Elementen in grafischen Benutzeroberflächen, Konfigurationsdateien oder Konsolenausgaben ermöglicht. Die Schnittstelle wird dabei so entwickelt, dass ohne eine Änderung des Quelltextes neue Sprachen hinzugefügt werden können. Die Lokalisierung beschreibt dann unter anderem die Übersetzung von den eben genannten Elementen.

Dieses Konzept kann durch die von Java bereitgestellte `ResourceBundle` Klasse realisiert werden [DCL01]. Bei der Verwendung eines solchen `ResourceBundles` wird jedem zu übersetzenden Element ein Schlüssel zugeordnet und nach Konvention in einer `.properties` Datei gespeichert. Wenn eine neue Sprache im Laufe des Lokalisierungsprozesses hinzugefügt werden soll, so muss jeweils eine neue `.properties` Datei angelegt werden. JavaFX ermöglicht das manuelle Spezifizieren einer vordefinierten `ResourceBundle` Instanz bei dem Laden einer FXML Datei durch einen `FXMLLoader`. In der zu ladenden FXML Datei müssen hartcodierte Textelemente durch den jeweiligen Schlüssel aus der `.properties` Datei, wie in Abbildung 4.1 gezeigt, ersetzt werden.

# Properties Datei	<!-- FXML Datei -->
login.user = Benutzername	<Label text="%login.username"/>

Code 4.1: Nutzung des Schlüssels in einer FXML Datei

Das Problem bei dieser Art der Übersetzung ist, dass eine Änderung der Sprache zur Laufzeit des Programms nicht dynamisch möglich ist. Die FXML Datei bzw. der dazugehörige Controller muss nach einer Sprachänderung durch beispielsweise eine Schaltfläche oder ein Dropdown-Menü durch einen FXML-Loader neu geladen werden, damit eventuelle Änderungen übernommen werden können. Das dynamische Ändern der Sprache zur Laufzeit ist nur mit einer Modifizierung des Ladeprozesses von FXML Dateien durch eine eigene Version des `FXMLLoader`s oder durch eine vom `FXMLLoader` unabhängige Implementierung mit Properties und Bindings möglich. Die erste Variante sorgt für ein automatisches Binden der nötigen Properties und die Letztere für ein manuelles Binden wenn nötig, weshalb eine Fusion beider Möglichkeiten in eine hohe Anpassbarkeit des Systems resultiert. Außerdem ist es auf diese Weise möglich, verschiedene Bindings zu aktualisieren, falls ein bestimmtes Event auftritt, welches eine Änderung des übersetzten Textes hervorruft. Parametrisierte Schlüssel aus der `.properties` Datei können somit automatisch an die parameterverändernden Events gebunden werden.

4.1.2. Abhängigkeitsinjektion für Controller

Bei der Abhängigkeitsinjektion werden Abhängigkeiten von Objekten zur Laufzeit des Programms bestimmt und zur Verfügung gestellt. Meist konfiguriert der Entwickler mithilfe einer externen Bibliothek die Bereitstellung der Abhängigkeiten in einer Konfigurationsdatei oder Konfigurationsklasse. Bei dem Nutzen einer Art von Abhängigkeitsinjektion ist die eigentliche Implementierung der Abhängigkeiten durch die Objekte nicht bekannt. Diese kennen nur die Schnittstellen, weshalb ein Auswechseln der Schnittstellenimplementierung durch eine Änderung der jeweiligen Konfigurationsdatei/Konfigurationsklasse möglich ist. Durch diese explizite Trennung von Schnittstelle und Implementierung ist eine lose Kopplung der Komponenten gewährleistet, was wiederum zu einer hohen Flexibilität und zu einer hohen Wartbarkeit sowie Testbarkeit führt. Eine Injektion ist dabei durch Felder (Feldinjektion), Konstruktoren (Konstruktorinjektion) oder Setter (Setterinjektion) möglich. Die Funktionalität einer Injektion von Abhängigkeiten in einen Controller ist nicht direkt in JavaFX enthalten, kann aber durch das Verwenden von zum Beispiel Afterburner.fx⁵ hinzugefügt werden. Eine Unterstützung von etablierten externen Bibliotheken zur Realisierung des Abhängigkeitsinjektionsmusters wie Spring¹, Dagger² oder Guice³ ist dadurch jedoch nicht gegeben. Aufgrund der Tatsache, dass alle drei genannten Bibliotheken mit der `jakarta.inject.Inject` bzw. `javax.inject.Inject` Annotation eine grundlegende Abhängigkeitsinjektion bereitstellen, kann ebendiese Annotation für eine Injektion innerhalb von Controllern verwendet werden. Ein Beispiel eines Controllers mit verschiedenen injizierten Diensten ist in Code 4.2 zu erkennen.

```
public class TestController {

    @Inject
    private IUserService userService;

    @Inject
    private IDatabaseService dbService;

}
```

Code 4.2: Beispiel – Controller mit injizierten Diensten

4.1.3. CSS Metadatengeneration

Ein wichtiger Bestandteil von JavaFX ist die Möglichkeit, das Aussehen und den Stil von einzelnen Komponenten wie Schaltflächen und Containerelementen durch die Verwendung von CSS zu modifizieren. Werden eigene Komponenten erstellt oder bestehende Komponenten erweitert, so erlaubt JavaFX das Hinzufügen von

¹Spring: <https://spring.io>

²Dagger: <https://square.github.io/dagger/>

³Guice: <https://github.com/google/guice>

eigenen CSS Properties zu den jeweiligen Komponenten mithilfe von neuen Instanzen der `CssMetaData` Klasse. Für jede neue CSS Property muss dabei eine neue `CssMetaData` Instanz erstellt werden, weshalb das Hinzufügen von vielen dieser Instanzen ein repetitiver Prozess mit vielen Boilerplate Quelltextfragmenten ist, welcher durchaus vereinfacht werden kann.

4.1.4. JavaFX Einstiegspunkt und Preloader

Damit eine JavaFX Applikation gestartet werden kann, muss, wie in Abschnitt 2.2 beschrieben, eine Klasse existieren, welche von der `Application` Klasse erbt. Müssen bestimmte performanceintensive Aufgaben wie das Laden von Sound-, Video- oder Bilddateien vor dem Start der eigentlichen Anwendungen ausgeführt werden, so kann dies in einer Klasse, welche von der `Preloader` Klasse erben muss, realisiert werden. Der `Preloader` ist dabei eine spezialisierte Form der `Application`, welche es dem Entwickler ermöglicht, Ressourcen zu laden und dies dem Nutzer durch eventbasierte Statusaktualisierungen mitzuteilen. Damit ein `Preloader` mit einer standardmäßigen JavaFX Anwendung verbunden werden kann, muss statt `Platform#launch` die interne `PlatformImpl#launchApplication` Methode genutzt werden. Entwickler sollen das Nutzen von internen Klassen und Methoden weitgehend vermeiden, da sich Implementierungen und Funktionen dieser ständig ändern können. Das Verwalten der Aufgaben, welche vor dem Anwendungsstart durchgeführt werden müssen, soll vereinfacht werden, wobei insbesondere die vorhandenen Statusaktualisierungen erweitert werden sollen. Auch die Initialisierung einer Applikationsklasse bzw. der benötigten `Stage` läuft in den meisten Fällen gleich ab und soll bei Bedarf vom Entwickler zu einem großen Teil automatisiert werden können.

4.1.5. Controller Lebenszyklus

Der Lebenszyklus von JavaFX Controllern ist in der aktuellen Ausführung für komplexe Systeme mit vielen Controllern ungeeignet, da dieser nur aus zwei Phasen besteht. Zuerst wird die Controllerklasse instanziert und darauf folgt die Initialisierung der `@FXML` Felder und der Methodenaufruf einer vom Entwickler bereitgestellten `initialize` Methode. Damit mit fertig initialisierten `@FXML` Feldern gearbeitet werden kann, muss der dafür benötigte Quelltext immer in der `initialize` Methode definiert werden. Ein Beispiel für den Ablauf einer Controllerinstanziierung durch den `FXMLLoader` ist in Code 4.3 abgebildet. Der rudimentäre Lebenszyklus unterstützt dabei keine Methoden, welche beispielsweise bei einem Entfernen des Wurzelements eines Controllers aus dem Szenengraphen ausgeführt wird, um eventuell offene Ressourcen zu schließen und somit effektiv Ressourcenlecks zu verhindern. Ein vollständiger Lebenszyklus wie bei Activities in Android⁴, ist praktisch nicht vorhanden.

⁴<https://developer.android.com/guide/components/activities/activity-lifecycle>

```

public class TestController {

    @FXML
    private Label testLbl;

    public TestController() {
        // Phase #1 - Controllerinstanzierung
        // testLbl-Feld ist hier noch nicht initialisiert
    }

    @FXML
    private void initialize() {
        // Phase #2 - Feldinjektion fertiggestellt
        // testLbl-Feld ist initialisiert und kann verwendet werden
    }
}

```

Code 4.3: Beispiel – Instanziierungsablauf.

4.1.6. Konfigurationsdateien

Eine persistente Speicherung von Anwendungskonfigurationen wie beispielsweise Logindaten oder Informationen für Serververbindungen ist wichtig für eine optimale Benutzungsfreundlichkeit. Das Speichern von konfigurierbaren Einstellungen kann mit der Java Preferences API oder mit externen Bibliotheken wie GSON⁵ für auf JavaScript Object Notation (JSON) basierte Konfigurationsdateien oder JAXB, für auf XML basierte Dateien durchgeführt werden. Der Prozess des manuellen Erstellens einer Konfigurationsdatei und dem anschließenden Auslesen bzw. Modifizieren zur Laufzeit ist ein repetitiver Vorgang, welcher durch das Nutzen von Annotationen teilweise automatisiert werden kann.

4.2. Anforderungsanalyse

In der Anforderungsanalyse werden die gefundenen Problemlösungen und Vereinfachungen aus Abschnitt 4.1 in Form von funktionalen und nichtfunktionalen Anforderungen formuliert. Dabei werden die Anforderungen in zwei Klassen unterteilt:

Fundamentale Anforderungen sind Anforderungen, welche für eine Funktion des Systems essentiell sind, alle genannten Probleme weitgehend beheben und daher zwangsläufig implementiert werden müssen. Alle folgenden fundamentalen Anforderungen haben den [+A-##] Präfix, wobei + durch die Art der Anforderung (funktional, nichtfunktional) und ## durch die jeweilige Anforderungsnummer substituiert wird.

⁵GSON: <https://github.com/google/gson>

Optionale Anforderungen sind Anforderungen, welche keinen Einfluss auf eine ordnungsgemäße Funktionalität des Systems haben. Sie sind optional und werden möglicherweise aufgrund ihrer Komplexität nur teilweise oder gar nicht implementiert und können stattdessen für eine Erweiterung des Systems durch weitere Entwickler genutzt werden. Alle folgenden optionalen Anforderungen haben den (+A-##) Präfix, wobei + durch die Art der Anforderung (funktional, nichtfunktional) und ## durch die jeweilige Anforderungsnummer substituiert wird.

4.2.1. Funktionale Anforderungen

Im Folgenden werden alle funktionalen Anforderungen definiert. Sie beschreiben alle gewünschten Funktionen des Endproduktes.

[FA-01] Ermitteln und Konfiguration des JavaFX-Einstiegspunktes

Der Erstellen einer Applikationsklasse sowie eines optionalen Preloaders soll automatisch stattfinden, da der Erstellungsprozess von auf JavaFX basierten Anwendungen in den meisten Fällen ähnlich oder identisch ist. Das Definieren dieser Elemente soll mithilfe von Annotationen geschehen und durch das Absuchen des Klassenpfades durch die entwickelte Bibliothek gefunden werden. Alternativ soll der Entwickler die Möglichkeit haben, den Einstiegspunkt sowie den Preloader explizit zu spezifizieren, falls eine automatische Identifikation nicht erwünscht oder benötigt ist. Die vordefinierten Einstiegspunkte sollen eine Standardkonfiguration der Applikation und der zugrundeliegenden Stage Instanz bereitstellen, welche durch den Entwickler bei Bedarf auch manuell überschrieben werden kann. Dieses Überschreiben soll durch die Nutzung von Annotationen erfolgen. Die Standardkonfiguration umfasst grundsätzliche Applikationseigenschaften wie den Titel, die Fenstergröße oder Designelemente wie den StageStyle und die benötigten Titelleistenschaltflächen.

[FA-02] Eventbasierte Verwaltung des Applikationslebenszyklus

Die Verwaltung des Applikationslebenszyklus soll mithilfe von Annotationen im Einstiegspunkt geschehen. Der Applikationslebenszyklus bezeichnet hier die vererbaren Application#init, Application#start und Application#stop Methoden. Die Erstellung der Applikationsklasse soll, wie in Anforderung 1 beschrieben, durch die zu entwickelnde Bibliothek übernommen werden, weshalb die Aufrufe der eben genannten Methoden durch ein Event-System an den vom Entwickler spezifizierten Einstiegspunkt delegiert werden sollen. Beispielsweise wird statt der Überschreibung der Application#start Methode, eine Methode im Einstiegspunkt definiert, welche ein StartEvent als Parameter erhält und eine spezielle Annotation aufweist. Methoden für einen optionalen Preloader sollen analog dazu definiert werden können.

[FA-03] Dynamische Laufzeitübersetzung von JavaFX-Komponenten

Die bereits existierenden Internationalisierungsmöglichkeiten sollen durch eine dynamische Übersetzung erweitert werden. Das Ändern der aktuellen Sprache soll durch beispielsweise einen Schaltflächenklick ausgelöst werden und zur Laufzeit die grafische Benutzungsoberfläche aktualisieren. Ein Neustart der Anwendung oder ein erneutes Laden der FXML Dateien soll nicht mehr nötig sein. Der Entwickler soll dabei die zu übersetzenden JavaFX Komponenten in der jeweiligen FXML Datei deklarieren und diese sollen dann durch eine verbesserte Version der `FXMLLoader` Klasse geladen und instanziert werden. Dabei soll das dynamische Übersetzen der definierten Elemente per Element deaktivierbar sein. Die Funktionalität von externen FXML Parsern wie dem Scene-Builder soll durch das Hinzufügen von eigener FXML Syntax nicht kompromittiert werden. Übersetzbare Elemente, welche auf parameterabhängigen Übersetzungsschlüsseln basieren, sollen an JavaFX Properties gebunden werden können und bei Änderung dieser aktualisiert werden. Wird eine dynamische Übersetzung für nicht direkt durch den `FXMLLoader` übersetzbare Elemente oder Prozesse wie das Ausgeben von Nachrichten auf der Konsole oder generelle methodeninterne Textverarbeitung benötigt, so soll der Entwickler auf Utility-Klassen der Bibliothek zurückgreifen können, welche ebendiese Funktionalität zur Verfügung stellen.

[FA-04] Abhängigkeitsinjektion durch etablierte externe Bibliotheken

Werden externe Bibliotheken zur Abhängigkeitsinjektion genutzt, sollen eventuelle Abhängigkeiten und Dienste automatisch in Controller, die Applikationsinstanz und den optionalen Preloader injiziert werden. Die Nutzung etwaiger Bibliotheken und die dafür benötigten Konfigurations- bzw. Modulklassen sollen beim Start der Applikation in Form einer Annotation spezifiziert werden können. Eine Unterstützung der Bibliotheken Dagger, Spring und Guice soll dem Entwickler ermöglicht werden.

[FA-05] Automatische CSS Metadaten Generation

Das Hinzufügen von CSS Properties zu benutzerdefinierten JavaFX Komponenten durch den Entwickler soll durch das Nutzen von Annotationen erleichtert werden. Durch die automatische Generierung dieser Metadaten sollen eventuell auftretende repetitive Quelltextbausteine vermieden werden. Der verbesserte `FXMLLoader` soll hierbei nach statischen Feldern mit der CSS Metadaten Annotation suchen und die gefundenen Metadaten automatisch mit den durch die Annotation definierten Einstellungen instanziieren.

[FA-06] Annotationsbasierte Controllerdefinition

Ein Controller soll als solcher erkannt werden, wenn dieser eine spezielle Annotation aufweist. Die Annotation muss dabei die Quelle der FXML und der CSS

Datei enthalten und weitere, auf Controller bezogene Konfigurationsmöglichkeiten bereitstellen.

[FA-07] Erweiterung des Controllerlebenszyklus

Der Lebenszyklus eines Controllers soll, ähnlich wie bei Android Activities, in mehrere Phasen eingeteilt sein, welche durch die Nutzung von Annotationen durch den Entwickler abrufbar gemacht werden sollen. Methoden, welche bei einer bestimmten Phase ausgeführt werden sollen, müssen mit der zu der Phase gehörenden Annotation annotiert werden können. Der Lebenszyklus soll um folgende Phasen erweitert werden:

- In der **Setup** Phase ist die grobe Initialisierung eines Controllers beendet und die vom Entwickler spezifizierten Subcontroller werden initialisiert.
- In der **PostConstruct** Phase sind alle Abhängigkeitsinjektionen und alle Instanziierungen von JavaFX Komponenten vollständig abgeschlossen und somit durch den Entwickler im Controller verwendbar.
- In der **OnShow** Phase wird das Wurzelement des Controllers im Szenengraphen aktiv gerendert und ist somit durch den Endbenutzer in der grafischen Benutzeroberfläche zu erkennen.
- In der **OnHide** Phase wurde das Wurzelement des Controllers aus dem Szenengraphen entfernt und dementsprechend nicht gerendert.
- In der **OnDestroy** Phase werden alle zwischengespeicherten Ressourcen und Informationen des Controllers entfernt. In dieser Phase sollen eventuell offene Ressourcen geschlossen werden. Damit der Controller wiederverwendet werden kann, muss eine erneute Initialisierung durch beispielsweise einen FXMLLoader erfolgen.

[FA-08] Benachrichtigung bei Komplettierung der Einrichtung von Einstiegspunkten und Controllern

Wenn die Einrichtung des Einstiegspunktes der Applikation, des Preloaders oder der Controller beendet ist, soll die jeweilige Instanz benachrichtigt werden. Bei einer solchen Einrichtungskomplettierung, sollen speziell annotierte Methoden (@PostConstruct) aufgerufen werden. Die Reihenfolge der Aufrufe richtet sich nach der Priorität, welche in Form eines Parameters der Annotation übergeben werden kann.

[FA-09] Unterstützung von mehreren Lebenszyklusbehandlungsmethoden

Wenn mehrere Methoden mit denselben Lebenszyklusannotationen annotiert wurden, soll die Reihenfolge der Aufrufe dieser anhand eines optionalen Parameters in der Annotation bestimmt werden.

[FA-10] Controllerverschachtelung und Supercontroller

Ist das Wurzelement eines Controllers in einem, durch einen weiteren Controller aufgespannten, Teilbaum des Szenengraphen enthalten, so handelt es sich bei ersterem Controller um einen Subcontroller des Letzteren. Subcontroller sollen bei Phasenänderung des Lebenszyklus ihres Supercontrollers benachrichtigt werden. Beispielsweise soll der Subcontroller in den OnHide Zustand wechseln, wenn der Supercontroller in denselben Zustand wechselt. Diese Beziehung soll durch den Entwickler bei der Controllerdefinition deklariert werden können.

[FA-11] Gruppierung von ähnlichen Controllern

Ähnliche Controller sollen bei Definition sogenannten Controllergruppen zugeordnet werden können. Eine Controllergruppe teilt sich dasselbe Wurzelement, weshalb immer nur ein Controller einer Controllergruppe gleichzeitig aktiv sein kann. Ein Controller soll dabei nur einer Gruppe zuordenbar sein. Wechselt der aktive Controller, so soll die Controllergruppe davon in Kenntnis gesetzt werden und eventuelle Phasenwechsel von betroffenen Controllern ausgelöst werden. Verschachtelte Subcontroller sollen dabei zwischen einem Wechsel durch einen Supercontroller und einem selbst ausgelösten Wechsel differenzieren können.

[FA-12] Zyklusprävention

Damit keine unendlichen Schleifen und Prozesse aus dem falsche Nutzen des Systems resultieren, muss eine Art von Zyklusdetektion bei der Verschachtelung von Controllern erfolgen. Beispielsweise darf Controller A nicht als Supercontroller von Controller B definiert sein, wenn B den Controller A ebenfalls als Supercontroller definiert.

[FA-13] Wechsel von Controllern

Das Wechseln von Controllern soll durch das vom Controller-System bereitgestellte Eventsystem ermöglicht werden. Ein manuelles Wechseln der Controller soll somit durch die Absendung eines Events durchführbar sein. Das automatische Wechseln eines Controllers soll durch vereinfachende Strukturen wie Annotationen über Schaltflächendefinitionen erleichtert werden. Der Entwickler soll den Wechsel durch die Angabe von Animationsparametern ästhetisch individualisieren können.

[FA-14] Controller Preloading

Controller werden standardmäßig On-Demand geladen, also nur dann, wenn sie benötigt werden. Wenn der Entwickler einen Preloader definiert hat, dann soll das Erstellen und Initialisieren aller controllerrelevanten Elemente und Strukturen in dieser Preloading-Phase stattfinden können.

[FA-15] Globale Fehlerbehandlung

Beim Start der Anwendung soll optional eine Klasse zur globalen Fehlerbehandlung spezifiziert werden können. Wird eine solche Klasse nicht durch den Entwickler angegeben, so soll auf eine vom System bereitgestellte Klasse zurückgegriffen werden. Auftretende Fehler im Controllersystem, ob reflektiv bedingt oder nicht, sollen so an zentraler Stelle verwaltet und optional als Fehleranzeige auf dem Hauptcontroller angezeigt werden können. Nutzt der Entwickler einen Preloader, so sollen etwaige Ausnahmen zuerst an diesen weitergeleitet werden.

[FA-16] Geteilte Ressourcen

Controller sollen die Möglichkeit haben, bestimmte Felder mit anderen Controllern zu teilen. Geteilte Felder werden als solche identifiziert, wenn diese eine spezielle Annotation aufweisen. Das Controller-System soll dabei Referenzen von den jeweiligen Feldwerten erstellen und diese dann in Form von JavaFX Properties zur Verfügung zu stellen. Geteilte Ressourcen sollen dabei durch ihren Feldnamen oder optional durch einen Parameter der Annotation identifiziert werden. Der Annotationsparameter hat dabei immer eine höhere Priorität. Wird der Wert einer geteilten Ressource geändert, so soll sich diese Änderung in allen anderen Controllern mit ebendieser Ressource widerspiegeln.

[FA-17] Konfigurationsdateien

Felder eines Controllers sollen bei der Controllererstellung durch das Nutzen von automatisch oder manuell erstellten Konfigurationsdateien gesetzt werden, falls diese eine spezielle Annotation aufweisen. Der Konfigurationsschlüssel kann durch einen Annotationsparameter festgelegt werden. Standardwerte sollen optional spezifiziert werden können.

(FA-18) Optional – Anpassbare Events durch Annotationen

Häufig genutzte JavaFX Events wie beispielsweise das ActionEvent sollen durch einfache Annotationen mit im Controller definierten Methoden assoziiert werden. Tritt ein über diese Methode definiertes Event auf, soll die dazugehörige Methode des Controllers aufgerufen werden. Damit die Verbindung zwischen Feld und Methode gewährleistet ist, müssen beide Elemente annotiert werden und einen gemeinsamen Identifikator, welcher den Annotationen als Parameter übergeben wird, aufweisen.

(FA-19) Optional – Serialisierung von JavaFX Komponenten

Die Position und die aktuelle Konfiguration von jedem JavaFX Element, welches im Szenengraphen präsent ist, soll bei einem Ende der Anwendung gespeichert werden und bei dem nächsten Start rekonstruiert werden. Dadurch bleibt der aktuelle Zustand der grafischen Oberfläche auch bei Anwendungsterminierung erhalten.

(FA-20) Optional – Scheduling von Methoden

Controllerinterne Methoden sollen von Entwickler als Scheduler definiert werden. Eine solche Methode wird dann in einem bestimmten Zeitintervall automatisch ausgeführt und gegebenenfalls wiederholt. Das Verhalten der Methode (wiederholend, zeitverzögert) kann durch Annotationsparameter kontrolliert werden.

(FA-21) Optional – JavaFX Application-Thread Forcierung

Einige Operationen auf Elemente des JavaFX Szenengraphen müssen auf dem JavaFX Application-Thread ausgeführt werden. Dazu gehört beispielsweise das aktive Ändern von renderbaren Elementen wie Textfeldinhalten oder Schaltflächen. Controllerinterne Methoden sollen auf dem JavaFX Application-Thread ausgeführt werden, wenn diese eine bestimmte Annotation aufweisen.

(FA-22) Optional – Annotationsvalidierung zur Kompilierzeit

Reflektive Operationen sind definitionsbedingt nur per Laufzeit der Anwendung ausführbar. Das Auslesen und Verarbeiten von annotierten Elementen und Annotationsen im Allgemeinen durch reflektive Operationen kann somit nur zur Laufzeit erfolgen. Ob eine Annotation inkorrekte oder ungenügende Informationen enthält oder falsche Typen annotiert wurden, ist dem Entwickler nur durch die Auslösung von Ausnahmen zur Laufzeit mitzuteilen. Beispielsweise ist es offensichtlich inkorrekt, ein Feld, welches einen JavaFX Container repräsentiert (z.B. StackPane), mit einer Annotation zu annotieren, welche für die Übersetzung von Komponenten verantwortlich ist. Das Überprüfen der Korrektheit der Annotationsnutzung soll nicht nur durch eine Laufzeitfehlerbehandlung sondern auch durch einen Annotationsprozessor zur Kompilierzeit durchgeführt werden. Eventuelle Fehlkonfigurationen werden somit bereits beim Kompiliervorgang entdeckt und dem Entwickler mitgeteilt.

4.2.2. Nichtfunktionale Anforderungen

Im Folgenden werden alle nichtfunktionalen Anforderungen definiert. Sie beschreiben Qualitätseigenschaften an das System wie Möglichkeiten der Erweiterbarkeit und Wartbarkeit und spezifizieren Maßstäbe, welche zur Laufzeit der Anwendung eingehalten werden müssen. Dazu gehören beispielsweise die effiziente Ressourcenutzung, die Korrektheit des Systems sowie ein gewisser Grad an Zuverlässigkeit.

[NFA-23] Benutzungsfreundlichkeit

Das Erstellen von JavaFX Anwendungen soll durch das Nutzen der zu entwickelnden Bibliothek erleichtert werden. Um diese Erleichterung sicherzustellen, müssen implementierte Komponenten ausreichend dokumentiert sein und intuitiv durch den Entwickler nutzbar gemacht werden. Komplexe und überflüssige Strukturen sollen

vermieden werden. Jedes durch diese Bibliothek bereitgestellte konfigurierbare System muss Standardparameter zur Konfiguration bereitstellen und bei Bedarf eine Konfigurationsänderung durch einen Entwickler zulassen.

[NFA-24] Performance & Effizienz

Obwohl das vollständige System an vielen Stellen die Reflection API nutzen muss, soll sowohl der Entwickler als auch der Endnutzer keinen gravierenden Performanceunterschied feststellen können. Es soll sichergestellt werden, dass der durch die Reflection API geschaffene Performanceoverhead, minimal gehalten wird.

[NFA-25] Wartbarkeit

Die Bibliothek soll ohne viel Aufwand an sich verändernde Anforderungen anpassbar sein. Dazu muss das System allgemein anerkannte, softwaretechnische und objektorientierte Prinzipien erfüllen.

[NFA-26] Erweiterbarkeit

Die Architektur soll mit einer besonderen Fokussierung auf das Nutzen von Schnittstellen entwickelt werden. Annotationsbasierte Konzepte, welche nicht im Standardumfang der, von der Bibliothek bereitgestellten, Funktionalitäten enthalten sind, sollen ohne viel Aufwand durch externe Entwickler hinzugefügt werden können.

[NFA-27] Softwarequalität

Eine hohe Softwarequalität ist ein essentieller Faktor für ein System, welches auch außerhalb der eigenen Entwicklungsumgebung genutzt werden soll. Andere Entwickler müssen mit einem geringen Zeitaufwand einen Überblick über das System, mit allen wichtigen Komponenten und deren Beziehung untereinander, gewinnen können. Um eine hohe Nachvollziehbarkeit zu gewährleisten, müssen etablierte Entwurfsmuster genutzt und auf komplexen Quelltext verzichtet werden. Klassen müssen gut strukturiert und leicht verständlich implementiert werden.

[NFA-28] Lizenzierung

Die zu entwickelnde Bibliothek muss eine Konformität zu den von externen Bibliotheken genutzten Lizenzen aufweisen. Etwaige Lizenzverletzungen sind zu vermeiden.

4.3. Konzept und Modellierung

Im Folgenden wird die, für das Erfüllen aller vorher genannten Anforderungen, benötigte Architektur konstruiert. Dabei werden ähnliche Systemanforderungen in

Gruppen unterteilt und in Form von allgemeinen Zielen an das System zusammengefasst. Für jedes daraus resultierende Ziel, werden theoretische Implementierungsmöglichkeiten identifiziert, bei welchen benötigte Komponenten, die Beziehung zwischen diesen und die eventuelle Nutzung von Entwurfsmustern ermittelt werden. Dabei wird besonders auf eine hochwertige Softwarequalität und Gebrauchstauglichkeit geachtet. Grundlegende, notwendige Entscheidungen wie die Namensgebung der Bibliothek und die Nutzung von externen Bibliotheken für eine effizientere und leichtere Implementierung werden ebenfalls getroffen.

4.3.1. Ziele des Systems

Aus der Problemanalyse und der nachfolgenden Erhebung von funktionalen und nichtfunktionalen Anforderungen ergeben sich drei grundlegende Ziele des Systems:

- Erstellung von auf Annotationen basierenden Vereinfachungen für anspruchsvolle Aspekte der JavaFX-Programmierung.
- Erweiterung von bereits existierenden Funktionen wie der Lokalisierung und der Controllerverwaltung.
- Hinzufügen von Funktionen, welche aktuell nicht im Funktionsumfang von JavaFX enthalten sind wie beispielsweise die Abhängigkeitsinjektion.

Alle Ziele sollen neuen Entwicklern den Einstieg in JavaFX erleichtern aber auch erfahrenen Entwicklern die Möglichkeit geben, effizienter mit JavaFX zu programmieren. Dazu müssen diese nicht unbedingt die vom System zu Verfügung gestellten Annotationen verwenden, sondern können auch die Utility Klassen (für beispielsweise die Lokalisierung) und eventuell die vereinfachte Schnittstelle zur Java Reflection API nutzen.

4.3.2. Rahmenbedingungen und Designentscheidungen

Nachfolgend werden grundlegende Designentscheidungen der Softwarearchitektur und Rahmenbedingungen für die Entwicklung der Software als Ganzes dargelegt.

Namensgebung

Die zu entwickelnde Bibliothek benötigt einen Namen, welcher die Beziehung zu JavaFX verdeutlicht und gleichzeitig aufzeigt, dass die Vereinfachung und die Erweiterung von JavaFX Bausteinen und einzelnen Komponenten angestrebt wird und deshalb die oberste Priorität darstellt. Nachfolgend wird die Bibliothek als **SimpliFX** bezeichnet. SimpliFX ist dabei an das englische Wort *simplify* angelehnt, welches ins Deutsche übersetzt vereinfachen bzw. simplifizieren bedeutet. Damit die Verbindung zu JavaFX untermauert wird, ist das FY Suffix durch das FX Suffix substituiert worden.

Modularisierung

Gute Softwaresysteme bestehen aus vielen Komponenten, welche jeweils auf eine einzelne Funktion spezialisiert sind. Einige dieser Komponenten funktionieren unabhängig von anderen und können durch eine Dekomposition des Systems in unterschiedliche Module unterteilt werden. Eine modulare Softwarearchitektur folgt dem softwaretechnischen SoC Prinzip und resultiert in einer hohen Wartbarkeit sowie Erweiterbarkeit. Außerdem ermöglicht eine Modularisierung, dass einzelne Module aufgrund ihrer strikten Trennung einfacher ausgetauscht werden können.

Häufig genutzte reflektive Operationen der Reflection API müssen durch eine einfache Schnittstelle für SimpliFX zugänglich gemacht werden. Diese Schnittstelle muss eine ordnungsgemäße Ausnahmebehandlung aufweisen, da das Nutzen der Reflection API sehr fehleranfällig ist und muss sehr effizient implementiert werden, da etwaige Operationen langsamer sind als die Nutzung von herkömmlichen Java-Sprachfeatures. Diese Schnittstelle wird in einem von SimpliFX unabhängigen Apache Maven⁷ Modul ausgelagert. Auch die Testapplikation, welche nach der Implementierung des eigentlichen Systems entwickelt wird und für die Evaluation von diesem dient, wird in ein anderes Modul ausgelagert. Das Basissystem ist somit in einem dritten Modul zu finden und ist für die aus den Systemanforderungen resultierenden Funktionalitäten verantwortlich. Die verschiedenen Subsysteme werden dabei in Java-Pakete unterteilt.

Externe Bibliotheken

Externe Bibliotheken sind ein wichtiger Bestandteil von heutigen Anwendungen und können, aufgrund von Entwicklungsumgebungen wie IntelliJ IDEA⁶ oder Build-Management-Tools wie Apache Maven⁷ und Gradle⁷, einfacher als je zuvor einem Projekt hinzugefügt werden. Externe Bibliotheken haben dabei meist den Vorteil, dass diese oft als quelloffene Projekte für jeden Entwickler einsehbar und nutzbar sind und deshalb, aufgrund der vielen Beiträge von anderen Entwicklern, meist keine Fehler aufweisen. Auf der einen Seite ist das Nutzen von externen Bibliotheken im Vergleich zum manuellen Erstellen der gleichen Funktionen ein effektiver Weg um Zeit zu sparen und Bugs zu vermeiden. Auf der anderen Seite kann das Nutzen von vielen externen Bibliotheken, aufgrund der verschiedenen Softwarelizenzen für quelloffene Projekte, zu Problemen führen, da nicht alle Lizenzen eine Kompatibilität untereinander aufweisen. Für das Projekt werden nur Bibliotheken verwendet, welche lizenziertechnisch kompatibel sind. Alle verwendeten, externen Bibliotheken sind mit einer Kurzbeschreibung der Funktion und der jeweiligen Lizenz in Tabelle 4.1 dargestellt.

⁶IntelliJ IDEA: <https://www.jetbrains.com/de-de/idea/>

⁷Gradle: <https://gradle.org>

Bibliothek	Funktion	Version	Lizenz
JavaFX	Ermöglicht die Erstellung von Java Applikationen mit grafischen Oberflächen	16	GPLv2 ⁸
JFoenix	Layout-Erweiterungen für JavaFX	9.0.10	MIT ⁹
ASM	Erlaubt die Modifikation und das Auslesen von Java-Bytecode	9.1	BSD ¹⁰
Log4j2	Bereitstellung einer Loggingumgebung	2.14.1	Apache 2.0 ¹¹
Guice	Abhängigkeitsinjektion	5.0.1	Apache 2.0
Dagger1	Abhängigkeitsinjektion	1.2.5	Apache 2.0
Spring-Context	Abhängigkeitsinjektion	5.3.8	Apache 2.0

Tabelle 4.1.: Verwendete externe Bibliotheken

4.3.3. Benötigte Schnittstellen

Für eine optimale und effiziente Implementierung von SimpliFX, müssen einige Java Schnittstellen durch eigene vereinfacht oder Schnittstellen für nicht vorhandene Funktionen hinzugefügt werden. Fast jede der aufgestellten Anforderungen benötigt für Teilverfunktionen die Reflection API. Diese ist relativ fehleranfällig und nahezu alle reflektiven Operationen können Ausnahmen erzeugen, weshalb ein vereinfachender Zugang zu ebendieser API wünschenswert ist. Außerdem müssen Subsysteme entwickelt werden, um beispielsweise nach Anforderung 1 bestimmte annotierte Klassen aus dem Klassenpfad zur Laufzeit der Anwendung zu identifizieren. Im Folgenden werden die benötigten Subsysteme erläutert und die erforderlichen Komponenten mithilfe geeigneter Methoden modelliert. In Klassendiagrammen werden nur elementare Bestandteile der Klassen dargestellt und Funktionen wie Getter oder Setter, welche erst bei der eigentlichen Implementierung benötigt werden und für das Verständnis nicht relevant sind, werden weggelassen.

Schnittstelle: Reflection

Die Reflection Schnittstelle soll alle häufig genutzten, reflektiven Operationen, wie das Aufrufen von Methoden, das Instanziieren von Klassen, den Zugriff auf Felder

⁸ GPLv2 mit Classpath Exception: <https://openjdk.java.net/legal/gplv2+ce.html>

⁹ MIT: <https://opensource.org/licenses/MIT>

¹⁰ 3-Clause BSD: <https://asm.ow2.io/license.html>

¹¹ Apache 2.0: <https://logging.apache.org/log4j/2.x/license.html>

und das Auswerten mit Annotationen, an einem zentralen Ort ermöglichen. Reflektierbare Elemente sollen durch eigene Schnittstellen repräsentiert werden, welche für diese relevanten Operationen zur Verfügung stellen. Diese spezialisierten Klassen werden im Folgenden `ReflectionScopes` genannt, welche grundlegende Konfigurationen und Funktionen für die Modifikation des zu betrachteten Elementes bereitstellen. Alle `ReflectionScopes` sind mit ihrer jeweiligen Funktionalität in Tabelle 4.2 aufgelistet und detailliert als UML Klassendiagramme in Abbildung 4.1 dargestellt. Außerdem ist der Wechsel von `ReflectionScopes` möglich. So ist es beispielsweise möglich von einer `ClassReflection` über einen gefundenen Konstruktor zu einer `ConstructorReflection` Instanz zu wechseln. Auftretende Ausnahmen bei dem Nutzen der API, können durch die Angabe einer Ausnahmebehandlung abgefangen werden und somit durch den Entwickler kontrolliert werden.

ReflectionScope	Funktion
<code>ClassReflection</code>	Stellt Funktionen zum Finden von Methoden und Konstruktoren basierend auf modifizierbaren Filtern zur Verfügung.
<code>ConstructorReflection</code>	Ermöglicht das Erstellen von Klassen durch den repräsentierten Konstruktor.
<code>InstanceReflection</code>	Ermöglicht die Modifizierung von bereits instantiierten Objekten durch bspw. das Finden von Methoden oder Feldern.
<code>MethodReflection</code>	Erlaubt das Aufrufen von Methoden mit ihren jeweiligen Parametern.
<code>FieldReflection</code>	Stellt Funktionen zum Setzen und zum Auslesen von Feldern zur Verfügung.

Tabelle 4.2.: Alle benötigten `ReflectionScopes`

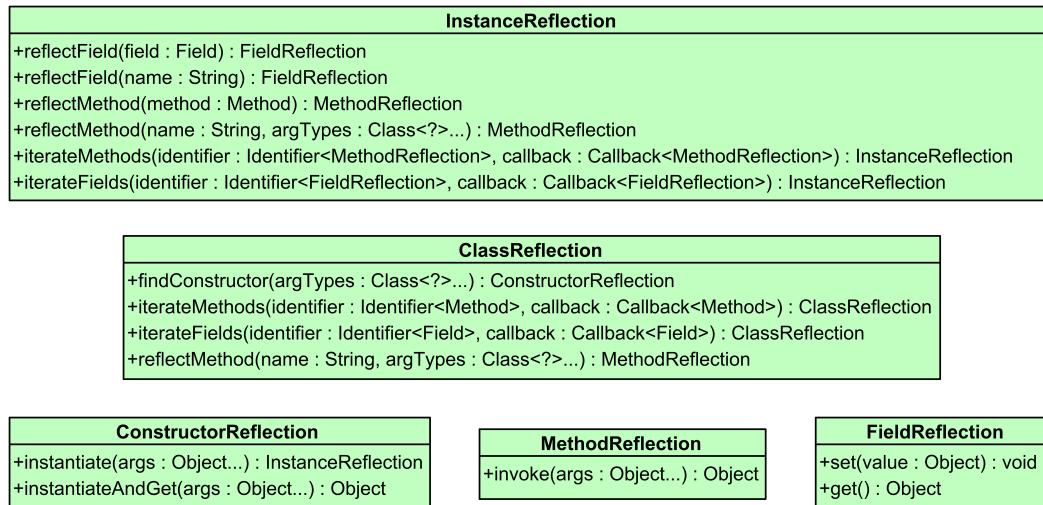


Abbildung 4.1.: Diagramm – ReflectionScopes

Aus den Klassendiagrammen ist schlusszufolgern, dass alle häufig genutzten reflektiven Operationen durch die Verwendung von Method Chaining, also der Hintereinanderausführung von Methoden, welche Operationen auf dem Objekt ausführen und dieses dabei zurückgeben, möglich sind. Ein theoretisches Anwendungsbeispiel der Reflection Schnittstelle ist in Code 4.4 dargestellt. Dabei wird eine Instanz der Test Klasse mit einem Integer Konstruktorparameter erstellt, welcher als Objektattribut in der Instanz gespeichert wird und mithilfe einer InstanceReflection modifiziert wird.

```

// Klassendefinition
public static class Test {

    private int test;

    public Test(int test) {
        this.test = test;
    }

}

// Instanziierung und Feldzugriff
InstanceReflection i = Reflection.reflect(Test.class)
    .findConstructor(int.class).instantiate(10);
FieldReflection f = i.reflectField("test").forceAccess();
f.set(42);
System.out.println(f.get()); // 42

```

Code 4.4: Beispiel – Verwendung der Reflection Schnittstelle

Schnittstelle: Klassenpfad

Damit die Automatisierungsmaßnahmen für Anforderung 1 zum Finden von speziell annotierten Elementen möglich sind, muss eine Art von Klassenpfad-Scanner implementiert werden. Dabei werden alle Klassen, welche in den durch `ClassLoader` Instanzen bereitgestellten Klassenpfaden gefunden werden, nach den jeweiligen Annotationen gefiltert (siehe Abbildung 4.2). Das Finden aller Klassen im Klassenpfad muss sowohl für normale Dateisysteme als auch für Applikationen, welche in Java Archiven gepackt worden sind, möglich sein. Um Klassen auf beispielsweise Annotationen zu filtern, darf, aufgrund von möglichen Sicherheitsrisiken und Performanceproblemen, kein `Class` Objekt mittels `Class.forName` erstellt werden. Dadurch soll verhindert werden, dass eventuelle `static`-Blöcke ausgeführt werden. Das Scannen einer Klasse, ohne den `static`-Block auszuführen, kann durch die ASM Bibliothek⁹ mithilfe des Besucherentwurfsmusters erfolgen. Außerdem muss der Scanprozess eine gewisse Konfigurierbarkeit aufweisen, da manuell `ClassLoader` Instanzen zum Scanprozess hinzugefügt werden sollen und um eine optimale Leistung zu gewährleisten, soll ein initialer Paketname oder Teilstring eines Paketes angegeben werden können, damit nicht benötigte Klassen schon im Scanprozess ignoriert werden. Ergebnisse des Scans müssen in einem Cache zwischengespeichert werden, damit nicht jeder Filtervorgang den Klassenpfad erneut scannen muss und somit unnötigerweise einen Performance-Hotspot kreiert.



Abbildung 4.2.: Diagramm – Klassenpfad-Scanprozess für Klassendateien

Schnittstelle: Lokalisierung/Internationalisierung

Für die Sprach- und `ResourceBundle`-Verwaltung (notwendig für Anforderung 3) werden Utility-Methoden benötigt, welche alle verfügbaren `ResourceBundle` Instanzen beinhalten und Funktionen für das Ändern der aktuellen Sprache bereitzustellen und JavaFX Bindings aus den jeweiligen Übersetzungsschlüsseln generieren. Dabei muss insbesondere die Unterstützung von parametrisierten Schlüsseln gewährleistet werden, da die daraus generierten Bindings bei einer Änderung der Parameterwerte aktualisiert werden müssen. Alle nötigen Basisoperationen, welche durch die `I18N` Schnittstelle verfügbar gemacht werden und durch `I18N` implementiert wurden, sind in Abbildung 4.3 dargestellt.

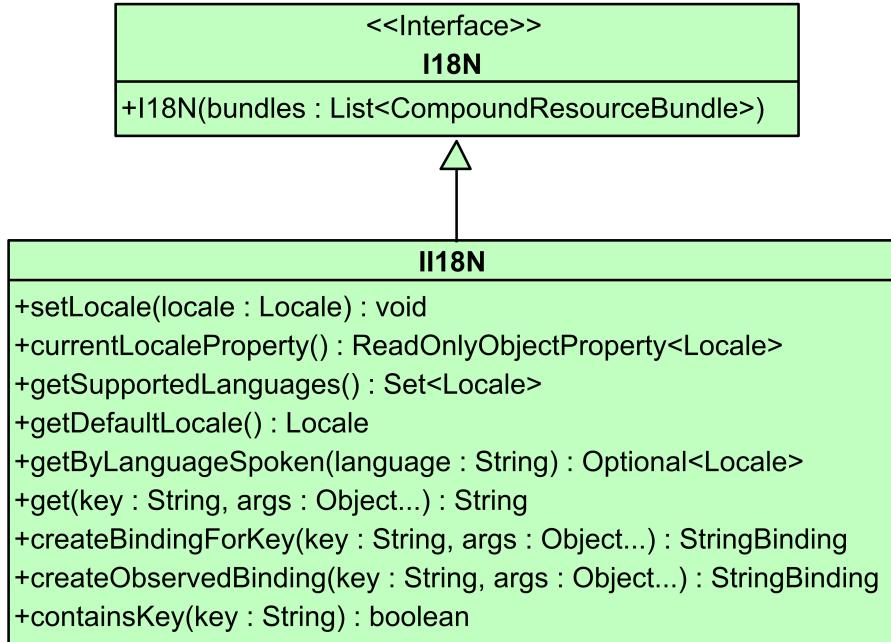


Abbildung 4.3.: Diagramm – II18N Schnittstelle

Eine `I18N` Instanz soll dabei, bei Bedarf, mehrere `ResourceBundle` verwalten, welche in Form eines `CompoundResourceBundle` Objektes per Konstruktor übergeben werden. Das Spezifizieren von `ResourceBundle` Instanzen kann durch den Nutzer von SimpliFX manuell beim Applikationsstart erfolgen oder durch das Verwenden, der in Abschnitt 4.3.3 beschriebenen Schnittstelle, automatisch geschehen. SimpliFX wird alle angegebenen/gefundenen `ResourceBundle`s in ein `CompoundResourceBundle` konvertieren und ein entsprechendes `I18N` Objekt erstellen, welches beim Erstellen aller Controller genutzt und dem Entwickler, für Übersetzungen außerhalb von Controllern, bereitgestellt wird. Ein Beispiel für die Instanziierung der `I18N` Klasse ist in Code 4.5 dargestellt.

```

Map<Locale, List< ResourceBundle >> bundles = ...;
II18N lang = new I18N(bundles.entrySet().stream()
    .map(e -> new CompoundResourceBundle(e.getKey(),
        e.getValue())))
.collect(Collectors.toList());
  
```

Code 4.5: Beispiel – Erstellen einer I18N Instanz

Schnittstelle: SimpliFXMLLoader

Damit Funktionalitäten wie die dynamische Übersetzung für Anforderung 3 und das Initialisieren sowie die Generierung von CSS Metadaten für Anforderung 5 möglich sind, muss der herkömmliche FXMLLoader durch eigene Konstrukte erweitert werden. Die Implementierung neuer Methoden und die Modifikation von

bereits existierenden Methoden wird in Unterabschnitt 5.3.1 beschrieben. Die Erweiterung der FXMLLoader Klasse wird im Folgenden als SimpliFXMLLoader bezeichnet.

Schnittstelle: SimpliFX Einstiegspunkt

SimpliFX übernimmt die Konstruktion und die anschließende Verwaltung der JavaFX-Anwendungsklasse. Essentielle Methoden, welche durch die Application Klasse zur Verfügung gestellt werden, können per Annotation an den spezifizierten Einstiegspunkt delegiert werden und sind somit nachfolgend für den Entwickler zugänglich (Anforderung 2). Der Entwickler kann, vor einem Start der Anwendung, SimpliFX konfigurieren und Eigenschaften wie das Verhalten des Klassenpfadscanners (global, paketlokal) modifizieren. Man muss zum Starten von SimpliFX eine der vordefinierten launch-Methoden verwenden (siehe Abbildung 4.4).

SimpliFX
+enableExperimentalFeatures() : void
+setClasspathScanPolicy(scanPolicy : ClasspathScanPolicy) : void
+setDefaultNotificationHandler(defaultNotificationHandler : Function<Pane, INotificationDialog>)
+setExceptionHandler(exceptionHandler : Consumer<Throwable>) : void
+launch(args : String...) : void
+launchWithPreloader(args : String...) : void
+launch(appEntryClass : Class<?>, args : String...) : void
+launch(applicationClass : Class<?>, preloaderClass : Class<?>, args : String...) : void
+launch(applicationListener : Object, args : String...) : void
+launch(applicationListener : Object, preloaderListener : Object, args : String...) : void

Abbildung 4.4.: Diagramm – Konfigurations- und Startmethoden von SimpliFX

Der Entwickler kann die Applikation- und Preloaderklasse manuell angeben oder bereits instanzierte Objekte für diese übergeben. Wird keine Applikationsklasse (bzw. Preloaderklasse) angegeben, so wird mithilfe der Klassenpfad-Schnittstelle (Abschnitt 4.3.3) eine Detektion dieser gestartet. Der Startprozess für die Lokalisierung des Einstiegspunktes ist in Abbildung 4.5 als UML-Aktivitätsdiagramm dargestellt (ähnlich dazu wird ein Preloader identifiziert). Wird ein Einstiegspunkt gefunden, so wird dieser auf Korrektheit von Annotationen überprüft. Daraufhin beginnt die Initialisierung der JavaFX Plattform, der Applikationsklasse, des Controllersystems und des Hauptcontrollers, die Konfiguration der jeweiligen Bibliothek für Abhängigkeitsinjektion (falls angegeben) und das Laden von Konfigurations- und Sprachdateien.

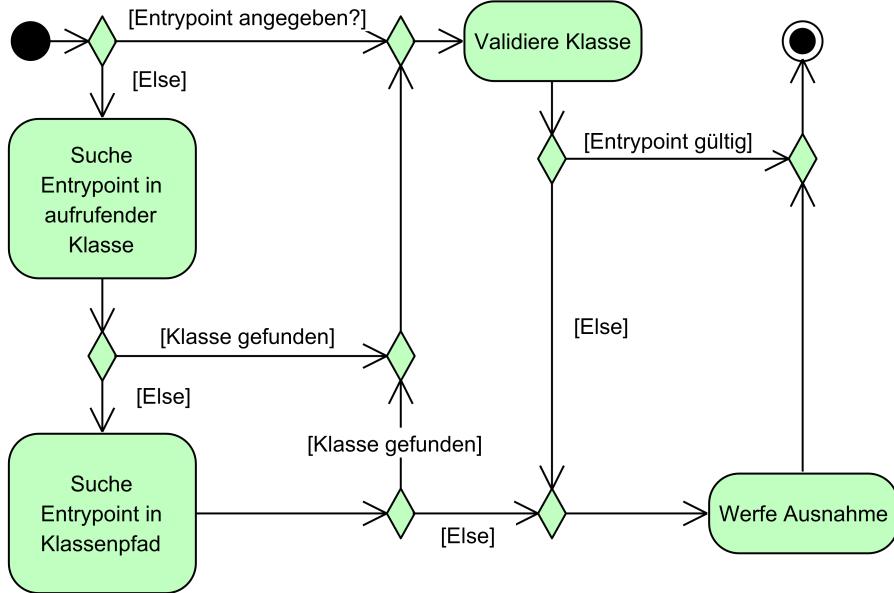


Abbildung 4.5.: Diagramm – Finden und Validierung des Einstiegspunktes

Das Setzen der `ClasspathScanPolicy` kann direkt durch eine statische Methode erfolgen. Dabei muss sichergestellt werden, dass diese explizit vor dem eigentlichen Start aufgerufen wird. Änderungen der `ClasspathScanPolicy` nach dem Applikationsstart haben keine Auswirkung auf den initialen Klassenpfadscan. Jede `launch`-Methode überprüft die angegebenen bzw. gefundenen Einstiegspunkte für die Applikation und den Preloader auf die korrekte Benutzung und Konfiguration von erforderlichen und optionalen Annotationen. Ist die Validierung der Einstiegspunkte erfolgreich, so wird die angegebene Bibliothek zur Abhängigkeitsinjektion geladen, die dazugehörigen Module initialisiert und alle Abhängigkeiten für die Applikation und den Preloader in die jeweiligen Instanzen injiziert. Ist keine Annotation zur Abhängigkeitsinjektion angegeben, so wird diese Phase übersprungen. Bevor die eigentliche JavaFX Applikation erstellt wird, werden alle angegebenen Pfade zu Lokalisierungsdateien auf Gültigkeit überprüft und mithilfe der Erstellung von `ResourceBundle` Instanzen zu jeder verfügbaren Sprache in einem `CompoundResourceBundle` kombiniert. Diese kombinierten `ResourceBundles` werden für die darauffolgende Instanziierung einer globalen `I18N` Instanz benötigt, welche optional in alle Einstiegspunkte und zukünftig erstellte Controller injiziert und für alle Ladeprozesse von FXML Dateien durch den `SimpliFXMLLoader` genutzt werden. Ist die Initialisierung abgeschlossen werden, im Rahmen der Anforderung 8, alle `PostConstruct` Methoden der Einstiegspunkte aufgerufen.

4.3.4. Controller System

Nach dem Finden und der vollständigen Konstruktion der Einstiegspunkte wird das Controller-System mit dem angegebenen Startcontroller initialisiert. Verschiedene

Controller können Controllergruppen zugeordnet werden. Jede solche Gruppe muss zu jedem Zeitpunkt mindestens einen Controller beinhalten und eindeutig durch einen String Wert identifiziert werden können. Der Erstellungsprozess eines Controllers und die Aufrufreihenfolge der Lebenszyklusbehandlungsmethoden ist in Abbildung 4.6 schematisch dargestellt. Bei der Controllervalidierung wird die korrekte Definition eines Controllers überprüft, sowie nach eventuellen Zyklen im System gesucht (Anforderung 12). Nach Anforderung 9 ist es ebenfalls möglich, mehrere Methoden für die gleiche Phase zu deklarieren. Die Aufrufreihenfolge richtet sich meist nach einem Annotationsparameter (siehe Unterabschnitt 4.3.5). Während der Konstruktion kann der Entwickler durch die Deklaration von speziell annotierten Methoden bei Phasenwechsel benachrichtigt werden. Um ursprüngliche Funktionalitäten des FXMLLoader zu bewahren, erlaubt es der SimpliFXMLLoader ebenfalls, dass ein geladener Controller die Initializable Schnittstelle implementiert und somit informiert wird, wenn der FXML Ladeprozess erfolgreich beendet wurde (alle Abhängigkeiten, welche durch eine Bibliothek zur Abhängigkeitsinjizierung bereitgestellt werden, sind zu diesem Zeitpunkt ebenfalls im Controller vorhanden). Nach dem Injizieren der von SimpliFX bereitgestellten Ressourcen in den Controller, wechselt dieser, wie in Anforderung 7 beschrieben, in die Setup-Phase. Hierbei werden eventuelle Setup-Methoden aufgerufen und eine Schnittstelle zur Interaktion mit dem Konstruktionsprozess der aktuellen Controllergruppe bzw. des Controllers bereitgestellt. Mit dieser Schnittstelle kann der Entwickler die Erstellung von Subcontrollern/Subgruppen in der Setup-Phase koordinieren (Anforderung 11). Ein Controller wechselt in die PostConstruct-Phase, wenn alle Subcontroller vollständig konstruiert oder wenn solche zur Setup-Phase nicht angegeben wurden. Methodenaufrufe für die PostConstruct-Phase werden ausschließlich auf dem JavaFX-Thread ausgeführt. Außerdem kann der Entwickler zu einem beliebigen Zeitpunkt einen Controller in einer Controllergruppe wechseln (Anforderung 13). Ein solcher Wechsel überführt den vorher angezeigten Controller in die OnHide-Phase und den neuen Controller in die OnShow-Phase. Dabei werden mögliche Subcontroller bzw. Subgruppen ebenfalls in den jeweiligen neuen Zustand des Supercontrollers versetzt. Der Entwickler kann durch einen Methodenparameter der OnHide-/OnShow-Methoden feststellen, ob der Phasenwechsel durch einen Controllerwechsel der lokalen Controllergruppe oder durch einen Supercontroller hervorgerufen wurde. Der Wechsel kann durch die Angabe von vordefinierten, keinen oder eigenen Animationen durchgeführt werden. Des Weiteren ist es möglich Controller manuell zu laden, ohne diese direkt anzeigen zu lassen. Somit müssen Controller nicht On-Demand, also bei Bedarf, durch das Controller-System erstellt werden (Anforderung 14).



Abbildung 4.6.: Diagramm – Erstellung der Hauptgruppe und des Hauptcontrollers

Daneben können Controllergruppen Benachrichtigungen anzeigen lassen. Das Layout dieser ist durch den Entwickler änderbar. Ausschließlich der Hauptcontroller

und die Hauptgruppe werden durch SimpliFX erstellt. Dabei wird für diese die Identifikation main gewählt. Dem Entwickler steht es frei, ob dieser die Hauptgruppe mit weiteren Controllern erweitert.

4.3.5. Annotationen

Im Folgenden werden alle benötigten Annotationen mit ihren jeweiligen Parametern definiert und auf die zu implementierende Funktionalität sowie mögliche Restriktionen eingegangen. Dabei wird explizit angegeben, welche Parameter optional sind und aufgrund dessen vom Entwickler nicht angegeben werden müssen.

Annotation: @ApplicationEntryPoint

Soll eine Anwendung mit SimpliFX verwendet werden, so muss eine Klasse existieren, welche als Einstiegspunkt fungieren soll und die @ApplicationEntryPoint Annotation aufweist. Dieser kann ein einziger Parameter übergeben werden, welcher notwendig ist und die Klasse des Hauptcontrollers darstellen soll (siehe Code 6.1).

Annotation: @PreloaderEntryPoint

Die @PreloaderEntryPoint Annotation besitzt ähnliche Funktionalitäten wie die @ApplicationEntryPoint Annotation, nur dass kein Parameter angegeben werden kann und nicht der Anwendungseinstiegspunkt definiert wird, sondern der Einstiegspunkt für einen Preloader. Dieser ist optional und ist für ein Starten der Applikation nicht notwendig.

Annotation: @ConfigSource

Mit der Java Properties-API kompatible Konfigurationsdateien können durch das Nutzen der @ConfigSource Annotation automatisch geladen und annotierten Properties-Feldern zugewiesen werden. Die Annotation hat einen optionalen Parameter, welcher den Namen der Konfigurationsdatei bzw. den Pfad zu dieser enthält. Der Pfad kann hierbei relativ oder absolut sein. Außerdem kann angegeben werden, ob die Konfigurationsdatei im Klassenpfad oder außerhalb davon zu finden ist. Jedes Element aus einer Konfigurationsdatei, welche auf diesem Wege geladen worden ist, kann durch die @ConfigValue Annotation verwendet werden (siehe Code 6.2).

Annotation: @ConfigValue

Konfigurierbare Elemente können mit @ConfigValue annotiert werden. Der Konfigurationsschlüssel muss in Form eines erforderlichen Parameters an die Annotation übergeben werden. Außerdem ist es möglich, einen optionalen Standardwert festzulegen, welcher dem jeweiligen annotierten Feld zugewiesen wird, wenn der angegebene Konfigurationsschlüssel in keiner Konfigurationsdatei gefunden werden kann (siehe Code 6.11).

Annotation: @PostConstruct

Hat SimpliFX den Erstellungs- bzw. Einrichtungsprozess eines Objektes vollendet, so soll die jeweilige Instanz davon in Kenntnis gesetzt werden. Dazu werden Methoden, welche die `@PostConstruct` Annotation aufweisen, automatisch aufgerufen. Optional kann ein Prioritätspараметer angegeben werden, welcher die Aufrufreihenfolge, bei der Existenz mehrerer solcher Methoden, bestimmt. Die Priorität wird dabei als Ganzzahl repräsentiert und Methoden werden in aufsteigender Reihenfolge aufgerufen (siehe Code 6.7).

Annotation: @Controller

Ein SimpliFX kompatibler Controller muss die `@Controller` Annotation aufweisen und dabei mindestens den FXML Parameter gesetzt haben. Der FXML Parameter gibt den Pfad zur FXML Datei an, welcher für die Konstruktion des Controllers benötigt wird. Analog dazu kann optional eine CSS Datei angegeben werden (siehe Code 6.5).

Annotation: @EventHandler

Methoden, welche bei einem Auftreten eines globalen Events mit den jeweiligen Eventinformationen aufgerufen werden sollen, können mit der `@EventHandler` Annotation annotiert werden. Diese akzeptiert einen optionalen Prioritätsparameter, der für die Aufrufreihenfolge gleichartiger Methoden verantwortlich ist. Annotatede Methoden werden automatisch erkannt und aufgerufen, sofern diese in einem Einstiegspunkt deklariert worden sind. Die Verwendung des Event-Systems kann ebenfalls manuell durch den Entwickler außerhalb von Controllern und Einstiegspunkten verwendet werden. Dazu muss dieser die jeweiligen Klassen zur Eventbehandlung in einem `IEventEmitter` registrieren (siehe Code 6.3).

Annotation: @ResourceBundle

Die Registrierung von `ResourceBundles` aus dem Klassenpfad erfolgt durch die `@ResourceBundle` Annotation, welche als notwendigen Parameter den Basisnamen bzw. den relativen Pfad ausgehend vom `resources` Verzeichnis akzeptiert. Nur Felder, die `I18N` oder eine Implementierung dieser Schnittstelle als Typen aufweisen, können mit `@ResourceBundle` annotiert werden (siehe Code 6.2).

Annotation: @LocalizeValue

Felder in Controllern werden automatisch übersetzt, sofern der Entwickler Pfade zu benötigten `ResourceBundle` Instanzen bereitstellt. Bei den Typen der Felder muss es sich immer um JavaFX Properties handeln, welche den jeweiligen Parametertypen darstellen können. Die Aktualisierung von Properties, welche eine Veränderung von, auf parametrisierten Übersetzungsschlüsseln basierenden, Textelementen

hervorrufen sollen, können mit der `@LocalizeValue` Annotation kontrolliert werden, sofern es sich bei ebendiesen Properties um controllerinterne Felder handelt. Statische Felder können hierfür nicht genutzt werden. Für die automatische Verbindung der Property und der jeweiligen JavaFX-Komponente, muss die FX-Id dieser als Annotationsparameter angegeben werden. Bei mehreren Übersetzungsparametern kann der Annotation ein Parameterindex übergeben werden (siehe Code 6.11).

Annotation: @CssProperty

Felder, welche einen `Collection`-Typen aufweisen, können mit der `CssProperty` Annotation annotiert werden. Dadurch wird automatisch eine neue `CssMetaData` Instanz erstellt und der Liste hinzugefügt. Der Entwickler muss den Namen der neuen CSS Property und die Konvertierungsklasse als Parameter angeben. Dazu ist es möglich, ein Feld anzugeben, welches an den konvertierten Wert der CSS Property gebunden wird und somit in der jeweiligen Klasseninstanz verfügbar ist. Die Annotation ist `Repeatable` und kann mehrmals an einem Feld angebracht werden.

Annotation: @DIAutowired

Die `@DIAutowired` Annotation ist eine Meta-Annotation, kann also nur an anderen Definitionen von Annotationen angebracht werden. Sie ist nicht für den Benutzer der Bibliothek bestimmt, sondern stellt anderen Annotationen wichtige Informationen zur Konstruktion von Komponenten zur Abhängigkeitsinjektion bereit. Es existieren drei Annotationen, welche jeweils für die Kompatibilität mit Spring, Guice und Dagger verantwortlich sind (siehe Anhang B).

Annotation: @SpringAutowired

Diese Annotation aktiviert die Unterstützung der Abhängigkeitsinjektion mit Spring und muss am Einstiegspunkt der Applikation angebracht sein. Konfigurationsklassen von Spring können als Parameter in Form eines Arrays übergeben werden.

Annotation: @GuiceAutowired

Diese Annotation aktiviert die Unterstützung der Abhängigkeitsinjektion mit Guice und muss am Einstiegspunkt der Applikation angebracht sein. Guice-Modulklassen können als Parameter in Form eines Arrays übergeben werden (siehe Code 6.1).

Annotation: @Dagger1Autowired

Diese Annotation aktiviert die Unterstützung der Abhängigkeitsinjektion mit Dagger der Version 1 und muss am Einstiegspunkt der Applikation angebracht werden. Dagger-Modulklassen können als Parameter in Form eines Arrays übergeben werden.

Annotation: @Setup

Die `@Setup` Annotation ist ausschließlich im Rahmen des Controller-Systems nutzbar und kann nur an Methoden angebracht werden. Hat SimpliFX einen Controller fertig initialisiert (Abhängigkeiten, Ressourcen etc.), so werden alle Setup-Methoden aufgerufen. In diesen Methoden kann der Entwickler neue Controller vom System initialisieren lassen und deren Wurzel-Node dann beispielsweise in anderen JavaFX-Komponenten integrieren (siehe Code 6.6).

Annotation: @OnShow und @OnHide

Methoden eines Controllers, welche mit `@OnShow` annotiert wurden, werden aufgerufen, wenn ein Controller im Szenengraphen enthalten ist und somit aktiv gerendert wird. Eine Priorisierung der Aufrufreihenfolge ist wie bei anderen Lebenszyklusbehandlungsannotationen ebenfalls möglich. Analog dazu werden `@OnHide` Methoden aufgerufen, wenn der Controller gewechselt und somit nicht mehr angezeigt wird (siehe Code 6.9).

Annotation: @OnDestroy

`@OnDestroy` Methoden werden aufgerufen, sobald ein Controller nicht länger benötigt wird und alle damit verbundenen Ressourcen gelöscht werden können.

Annotation: @EventEmitter und @EventReceiver

Die `@EventEmitter` Annotation verbindet das Auftreten von JavaFX-Events mit dem Aufruf von Methoden. Dabei werden Felder als EventEmitter deklariert und Methoden als EventReceiver. Ein notwendiger Parameter beider Annotationen ist ein gemeinsamer Identifikator, welcher die Beziehung zwischen Methode und Feld ermöglicht. Methoden müssen die jeweilige JavaFX-Event Klasse als Parameter aufweisen.

Annotation: @FXThread

Methoden, welche mit `@FXThread` annotiert wurden, sollen bei Aufruf von durch SimpliFX verwaltete Objekte, auf dem Java Application Thread ausgeführt werden. Aufgrund des dynamischen Ladens der Klasse zur Laufzeit durch einen eigenen `ClassLoader` und der damit einhergehenden Bytecode-Manipulation, handelt es sich um eine experimentelle Funktion, welche speziell vor Anwendungsstart aktiviert werden muss.

Annotation: @Shared

Objekte und Referenzen auf Objekte, welche geteilte Ressourcen darstellen und somit beispielsweise zwischen Controllern geteilt werden sollen, können mit `@Shared`

annotiert werden. Bei einer eventuellen Veränderung der Werte von geteilten Ressourcen, müssen offensichtlich alle Vorkommnisse dieser Ressource aktualisiert werden. Aufgrund dieser Eigenschaft können nur von SimpliFX bereitgestellte Klassen (z.B. eine SharedResource oder SharedReference Klasse), sowie read-only JavaFX ObjectProperty Felder eine geteilte Ressource darstellen und somit mit @Shared als solche deklariert werden. Geteilte Ressourcen sollen anhand eines String Wertes eindeutig identifizierbar sein, welcher optional per Parameter an die Annotation weitergegeben wird (siehe Code 6.8).

Annotation: @StageConfig

Die @StageConfig Annotation muss, bei Nutzung, den SimpliFX Einstiegspunkt annotieren und kann die JavaFX Hauptstage automatisch vorkonfigurieren. Mithilfe der Annotationsparameter kann der Titel, der StageStyle, das Vordergrund- und Skalierungsverhalten und das Symbol der Stage modifiziert werden. Außerdem kann der optionale autoShow Parameter angegeben werden, welcher kontrolliert, ob SimpliFX die Stage Instanz automatisch nach Erstellung mittels Stage#show anzeigt. Ein EventHandler, welcher das StartEvent der JavaFX Applikation verarbeitet, muss somit nicht definiert werden (siehe Code 6.1).

Übersicht und Zusammenfassung

In der folgenden Tabelle sind alle zu implementierenden Annotationen mit möglichen Typ- oder Elementrestriktionen, Parametern und den zugrundeliegende funktionalen Anforderungen aufgelistet.

Name	Anf.	Parameter	Restriktionen
StageConfig	1	<ul style="list-style-type: none"> • Fenstertitel • StageStyle • Immer im Vordergrund • Symbolpfad • Skalierbarkeit 	<ul style="list-style-type: none"> • Nur Einstiegpunkt ist annotierbar
Application-EntryPoint	1	<ul style="list-style-type: none"> • Hauptcontrollerklasse 	<ul style="list-style-type: none"> • Klasse muss instanzierbar sein
Preloader-EntryPoint	1	–	<ul style="list-style-type: none"> • Klasse muss instanzierbar sein
EventHandler	2	<ul style="list-style-type: none"> • Priorität 	<ul style="list-style-type: none"> • Nur Methoden
Resource-Bundle	3	<ul style="list-style-type: none"> • Basisname 	<ul style="list-style-type: none"> • Nur I18N Felder

LocalizeValue	3	<ul style="list-style-type: none"> • FX-Id • Parameterindex • Feldname der JavaFX Komponente 	<ul style="list-style-type: none"> • Nur Property-Felder
DIAnnotation	4	<ul style="list-style-type: none"> • IDIEnvironment-Factory Klasse 	<ul style="list-style-type: none"> • Nur Annotationen zur Abhängigkeitsinjektion
Spring-Injection, Guice-Injection, Dagger1-Injection	4	<ul style="list-style-type: none"> • Konfigurationsklassen 	<ul style="list-style-type: none"> • Nur Einstiegspunkt der Applikation
CssProperty	5	<ul style="list-style-type: none"> • CSS Property • Konvertierungsklasse • Lokales JavaFX-Property Feld • Java-FX Property für das Binding 	<ul style="list-style-type: none"> • Nur List-Felder
Controller	6,10	<ul style="list-style-type: none"> • FXML Pfad • CSS Pfad 	<ul style="list-style-type: none"> • Nur Klassen
Setup	7,10, 11	<ul style="list-style-type: none"> • Priorität 	<ul style="list-style-type: none"> • Nur Methoden in Controllern
PostConstruct	7,8	<ul style="list-style-type: none"> • Priorität 	<ul style="list-style-type: none"> • Nur Methoden
OnShow, OnHide, OnDestroy	7,13	<ul style="list-style-type: none"> • Priorität 	<ul style="list-style-type: none"> • Nur Methoden in Controllern
Shared	16	<ul style="list-style-type: none"> • Ressourcenidentifikator 	<ul style="list-style-type: none"> • SharedResources-, SharedReference-, ObjectProperty-Felder
ConfigSource	17	<ul style="list-style-type: none"> • Pfad oder Name der Datei • Klassenpfadquelle 	<ul style="list-style-type: none"> • Nur Properties-Felder
ConfigValue	17	<ul style="list-style-type: none"> • Konfigurationsschlüssel • Standardwert 	<ul style="list-style-type: none"> • Nur String und primitive Felder
EventEmitter	18	<ul style="list-style-type: none"> • Beziehungsidentifikator 	<ul style="list-style-type: none"> • ButtonBase-Felder
EventReceiver	18	<ul style="list-style-type: none"> • Beziehungsidentifikator 	<ul style="list-style-type: none"> • Methoden
FXThread	21	–	<ul style="list-style-type: none"> • Nur Methoden in Controllern

5. Implementierung

In diesem Kapitel werden wichtige Aspekte bei der Implementierung des Systems aufgezeigt. Auf essentielle Quelltextausschnitte, welche für eine Gewährleistung der, aus den Anforderungen resultierenden, Funktionalität verantwortlich sind, wird detailliert eingegangen. Dabei werden wichtige Konzepte, Strukturen und Designentscheidungen der entwickelten Architektur hervorgehoben und aufgetretene Probleme beim Implementierungsprozess dargelegt. Außerdem werden ausgewählte Subsysteme von SimpliFX mit geeigneten Darstellungsmitteln präsentiert und, für ein besseres Verständnis der zusammenarbeitenden Komponenten, eine beispielhafte Nutzung dieser durchgeführt. Die Verwendung und das Zusammenspiel der zuvor in Unterabschnitt 4.3.5 definierten Annotationen werden durch Quelltextbeispiele erklärt. Dazu werden mögliche mit einhergehende Restriktionen beleuchtet.

5.1. Architektur und Struktur der Software

Für die Implementierung und um eine, nach Anforderung 25 und Anforderung 26, hohe Softwarequalität sowie Erweiterbarkeit zu ermöglichen, muss das System durch eine wohlüberlegte Architektur repräsentiert werden. Um eine hohe Unabhängigkeit des Systems zu gewährleisten und eine Überladung mit unnötigen Funktion zu vermeiden, wurde auf die Nutzung von externen Bibliotheken zur Vereinfachung des Implementierungsprozesses und die Reduktion des Zeitaufwandes weitgehend verzichtet. Nur externe Bibliotheken, welche komplexe Funktionen zur Verfügung stellen und daher nicht im Rahmen dieser Arbeit implementiert werden können, sind im Projekt enthalten. Auch dürfen Bibliotheken, welche eine Inkompatibilität mit der im Projekt genutzten Software Lizenz aufweisen, nicht als Abhängigkeit genutzt werden. Für die Verwaltung der externen Bibliotheken und die Strukturierung des Systems wird, wie in Unterabschnitt 4.3.2 erläutert, das Apache Maven⁷ Build-Management-Tool verwendet. Das Projekt wird nach der finalisierten Implementierung als Maven-Artefakt öffentlich zugänglich gemacht und aufgrund der quelloffenen Natur auch per GitHub einsehbar sein. SimpliFX bietet dem Nutzer dazu auch verschiedene spezialisierte Artefakte an, welche an ein bestimmtes Framework für die Abhängigkeitsinjektion angepasst wurden. Für die interne Trennung der Belange und Funktionen von SimpliFX werden Java Pakete verwendet. Diese Paketstruktur wird im nachfolgenden Unterkapitel näher erläutert. Dabei wird auf die Funktionalität der, in den einzelnen Paketen definierten, Klassen eingegangen und mögliche wichtige Funktionen, Methoden und Klassen mit Beispielen und ausgewählten Quelltextausschnitten vorgestellt und explizit angegeben, ob eventuelle Restriktionen bei der Nutzung zu beachten sind.

5.2. Paketstrukturierung nach Funktionalität

Die verschiedenen Pakete von SimpliFX sind in Abbildung 5.1 dargestellt. Rot markierte Pakete dienen zur Abhängigkeitsinjektion und sind nicht im normalen Funktionsumfang enthalten, sondern nur mittels spezialisierter Maven-Artefakte nutzbar.

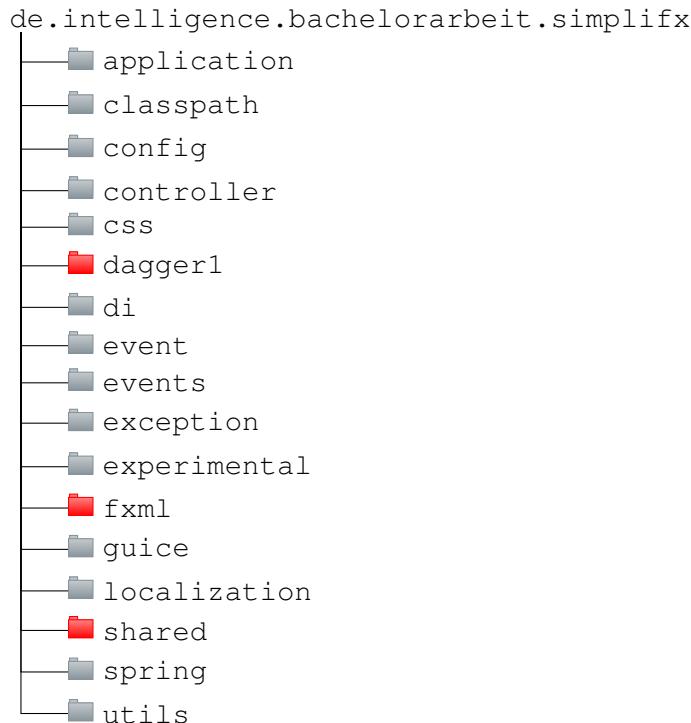


Abbildung 5.1.: Paketstruktur – SimpliFX

5.2.1. Paket: utils

Das `utils` Paket beinhaltet Klassen und Methoden, welche häufig genutzte Operationen an zentraler Stelle kombiniert. Außerdem werden Werkzeugklassen bereitgestellt, die in der Form nicht im Funktionsumfang von Java enthalten sind. Dazu gehören beispielsweise funktionale Schnittstellen mit einer Unterstützung von Ausnahmen, Klassen für das Überprüfen von Nullbarkeit oder booleschen Bedingungen und Implementierungen der `Iterator` Schnittstelle, welche durch das Nutzen von `AutoCloseable` Ressourcen schließen kann und damit eine Prävention von eventuellen Ressourcenlecks gewährleistet. Die Klasse `CloseableWrappedIterator` erlaubt das Erstellen einer `Iterator` Instanz, welche bei Nutzung der `stream` Methode alle genutzten Ressourcen bei einem Ende des Streams automatisch schließt. Außerdem wurde die Funktionalität der `Map.Entry` Klasse in die `Pair` Klasse ausgelagert, von welcher eine beispielhafte Nutzung in Code 5.1 dargestellt ist.

```
final Pair<String, Integer> pair = Pair.of("test", 0);
System.out.println(pair.getLeft()); // test
System.out.println(pair.getRight()); // 0
```

Code 5.1: Beispiel – Nutzung der Pair Klasse

5.2.2. Paket: di

Schnittstellen für die Unterstützung von Abhängigkeitsinjektion werden durch das `di` Paket bereitgestellt. Die eigentliche Implementierung dieser Schnittstellen ist in externen Artefakten zu finden, auf welche in den folgenden drei Untersektionen näher eingegangen wird. Das Paket stellt drei Klassen, `DIEnvironment` und `IDIEnvironmentFactory` sowie die `@DIAnnotation` zur Verfügung. Wenn eine weitere Bibliothek zur Laufzeitinjektion von Abhängigkeiten genutzt werden soll, welche nicht im Funktionsumfang von `SimpliFX` enthalten sind, müssen Implementierungen für beide Schnittstellen, sowie eine Annotation für die jeweilige Bibliothek bereitgestellt werden.

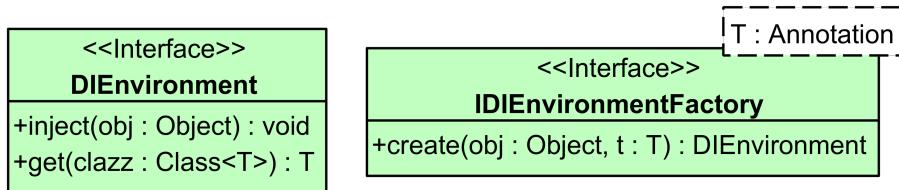


Abbildung 5.2.: Diagramm – DI Schnittstellen

`SimpliFX` kennt die Implementierung der in Abbildung 5.2 dargestellten Klassen nicht und kann daher mit Leichtigkeit um weitere Bibliotheken zur Abhängigkeitsinjektion erweitert werden. Zuerst muss der Entwickler eine Implementierung der `DIEnvironment` Klasse erstellen, welche für das Injizieren von vorhandenen Objekten und das Instanziieren von neuen Objekten verantwortlich ist. Danach muss eine `IDIEnvironmentFactory` mit dem Typen der Annotation als generischen Parameter erstellt werden, welche den alleinigen Zweck hat, neue Instanzen der vorher erstellten `DIEnvironment` Implementierung zu generieren. Letztlich muss der Entwickler eine Laufzeitannotation erstellen, welche an Typdefinitionen angebracht werden kann und dazu die Meta-Annotation `@DIAnnotation` mit der jeweiligen Implementierungsklasse der `IDIEnvironmentFactory` Schnittstelle als Parameter aufweist. Das Registrieren einer neuen Bibliothek ist in Anhang B gezeigt.

5.2.3. Paket: dagger1

Das `dagger1` Paket ist für die Integration von Dagger¹ in `SimpliFX` zuständig. Die `Dagger1Environment` Klasse definiert eine neue `ObjectGraph` Instanz aus

¹Dagger 2 realisiert Abhängigkeitsinjektion ausschließlich zur Kompilierzeit, weshalb Schnittstellen zum reflektiven Injizieren von Abhängigkeiten wie den `ObjectGraph` nicht mehr existieren und somit ein Zugang von `SimpliFX` auf den Injizierungsprozess ausgeschlossen ist.

den übergebenen Modulobjekten, welche für die Injektion verantwortlich ist. Dabei kann die Methode `ObjectGraph#inject` genutzt werden, um Abhängigkeiten in eine vorhandene Instanz zu injizieren und `ObjectGraph#get`, um eine neue Klasseninstanz zu erstellen.

5.2.4. Paket: guice

Das `guice` Paket ist für die Integration von Guice in SimpliFX zuständig. Dabei wird von der `GuiceEnvironment` Klasse ein neuer `Injector` erstellt, welcher durch die Methode `Injector#injectMembers` Abhängigkeiten injiziert und mit `Injector#getInstance` neue Instanzen erstellt.

5.2.5. Paket: spring

Das `spring` Paket ist für die Integration von Spring in SimpliFX zuständig. Die Umgebungsklasse für Spring (`SpringEnvironment`) erstellt eine neue Instanz der `AnnotationConfigApplicationContext` Klasse, welche es ermöglicht, mittels einer `AutowireCapableBeanFactory` Abhängigkeiten in bestehende Instanzen zu injizieren, sowie neue Instanzen aus Klassen zu erstellen.

5.2.6. Paket: localization

Alle Klassen und Funktionen, welche für das Umsetzen der dynamischen Lokalisierung benötigt wurden, sind im `localization` Paket enthalten. Dazu gehören beispielsweise die `I18N` Schnittstelle, eine Standard-Implementierung dieser und die `LocalizeValue` Annotation.

5.2.7. Paket: controller

Das vollständige Controller-System ist im `controller` Paket enthalten. Das System kann auch unabhängig verwendet werden. Es ist jedoch ratsam, SimpliFX für die Initialisierung zu nutzen, da vorkonfigurierte Klassen wie beispielsweise `I18N` dafür benötigt werden. Das System kann durch die Erstellung eines neuen `IControllerGroup` Objektes gestartet werden. Die Schnittstelle stellt das Fundament des Systems dar und ist in Abbildung 5.3 abgebildet. Durch die `start`-Methode kann die aktuelle Controllergruppe gestartet werden. Ist der Startcontroller zu dem Zeitpunkt noch nicht erstellt worden, so wird dieses mittels eines Aufruf der `constructController`-Methode nachgeholt. Diese Methode kann auch durch den Entwickler genutzt werden, um Controller zu generieren, bevor diese verwendet werden. Jede Gruppe muss exakt einen aktiven Controller besitzen, welcher durch `switchController` mit einer Standardanimation oder einer eigenen Animation gewechselt werden kann. Wird ein Controller bzw. eine Gruppe nicht weiter benötigt, so können mit den Aufruf der jeweiligen `destroy` Methode, alle gespeicherten Ressourcen und Instanzen gelöscht werden. Die von SimpliFX verwendete

Implementierung dieser sowie eine Darstellung der verschiedenen Beziehungen zu anderen Komponenten des Systems ist in Anhang C zu erkennen.

<<Interface>> IControllerGroup
<pre>+start() : Pane +switchController(newController : Class<?>) : void +destroy(clazz : Class<?>) : void +getContextFor(groupId : String) : ControllerGroupContext +visibilityProperty() : ObjectProperty<VisibilityState> +getActiveController() : Class<?> +getOrConstructController(clazz : Class<?>) : IController +start(primary : Stage) : void +createSubGroup(originController : Class<?>, startController : Class<?>, groupId : String, readyConsumer : Consumer<Pane>, notificationHandler : Function<Pane, INotificationDialog>) : void +switchController(newController : Class<?>, factory : IWrappedAnimation) : void +showNotification(title : StringBinding, content : StringBinding, kind : NotificationKind) : void +getStarControllerClass() : Class<?> +getGroupId() : String +getGroupContext() : ControllerGroupContext</pre>

Abbildung 5.3.: Diagramm – Einstiegspunkt des Controller-Systems

Controllergruppen werden anhand eines Strings eindeutig identifiziert und bei Konstruktion in einer globalen ControllerRegistry registriert, um doppelte Identifikatoren zu vermeiden. Dazu werden Controllerklassen vor der Konstruktion validiert. So ist es beispielsweise nicht erlaubt, eine nicht-statische, innere Klasse als Controller zu definieren, da eine direkte Instanziierung der Klasse durch SimpliFX in diesem Fall ausgeschlossen ist.

5.2.8. Paket: exception

Eigene Ausnahmeklassen sind im exception Paket aufzufinden. Dazu gehören spezielle Ausnahmen für das Controller-System oder generelle Ausnahmen, welche von einer Vielzahl der SimpliFX-Komponenten genutzt werden.

5.2.9. Paket: css

Alle CSS bezogenen Klassen sind im css Paket enthalten. Einige davon werden dabei ausschließlich durch den SimpliFXMLLoader genutzt, um automatisch CSS Metadaten für eigene JavaFX-Komponenten zu generieren.

5.2.10. Paket: experimental

Experimentelle Funktionen sind im experimental Paket zu finden. In der aktuellen Version der Bibliothek beinhaltet das Paket nur Klassen, welche für die Umsetzung der FX-Thread Forcierung (Anforderung 21) benötigt werden.

5.2.11. Paket: classpath

Wie bereits im Konzept beschrieben, wird für das Finden der Applikation- bzw. Preloader-Klasse ein System benötigt, welches in der Lage ist, den Klassenpfad zur Laufzeit des Programms nach Dateien zu durchsuchen. Dieses System findet nahezu alle Klassen und Ressourcen aus dem Klassenpfad, wird momentan aber nur für das Finden der jeweiligen Einstiegspunkte verwendet. Nur Klassenpfade aus einer

unkomprimierten Orderstruktur (beispielsweise bei der Nutzung von verschiedenen Entwicklungsumgebungen) und aus komprimierten JAR-Archiven sind möglich. Die Architektur ermöglicht aber ein Hinzufügen von weiteren Klassenpfadquellen wie zum Beispiel WAR-Archiven (siehe Abbildung 5.4).

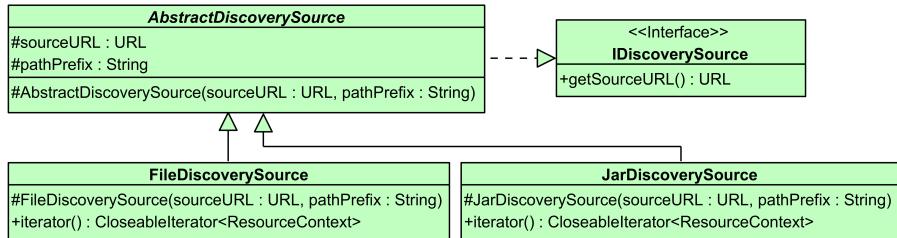


Abbildung 5.4.: Diagramm – Mögliche Quellen für Klassenpfade

Damit ein Klassenpfadscan initiiert werden kann, muss, wie in Abbildung 5.5 dargestellt, eine neue Instanz der `ClassDiscovery` Klasse erstellt werden. Eventuelle Konfigurationen wie das Hinzufügen weiterer `ClassLoader` oder dem Spezifizieren eines Basispfades, welcher den zu scannenden Klassenpfad einschränkt und somit in den meisten Fällen zu einer Reduktion der benötigten Scanzeit beiträgt, können durch die Nutzung des `DiscoveryContextBuilder` erreicht werden. Der `DiscoveryContextBuilder` bietet dabei auch standardisierte Konfigurationen an.

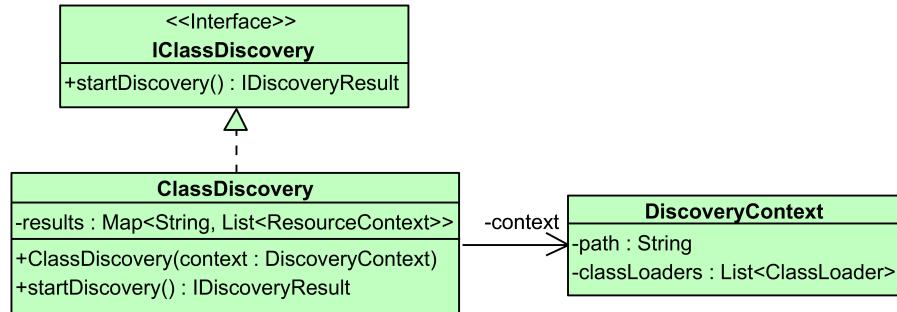


Abbildung 5.5.: Diagramm – Klassenpfad Schnittstelle und Implementierung

Eine beispielhafte Nutzung des Klassenpfadscans ist im folgenden Quelltextauschnitt dargestellt:

```

// Startet einen Scan mit Standardkonfiguration
var r = new ClassDiscovery(new DiscoveryContextBuilder()
    .setDefaultClassLoaders().build()).startDiscovery();
// Sucht alle mit @StageConfig annotierten Klassen
List<Class<?>> a = r.findClassesAnnotatedBy(StageConfig.class);

```

Code 5.2: Beispiel – Initiierung eines Klassenpfadscans.

5.2.12. Paket: shared

Die für das Konzept der geteilten Ressourcen benötigten Klassen wie zum Beispiel der SharedResources Klasse, welche alle globalen Ressourcen an zentraler Stelle sammelt und dem Entwickler bei Bedarf zur Verfügung stellt oder der SharedFieldInjector Klasse, welche für das Injizieren der Ressourcen in die Applikation sowie in alle neu erstellten Controller verantwortlich ist.

5.2.13. Paket: event

Das Event System findet seinen Ursprung im event Paket. Es setzt sich aus einer Schnittstelle, deren Implementierung und der @EventHandler Annotation zusammen. Die IEventEmitter Klasse (siehe Abbildung 5.6) ermöglicht das Registrieren sowie das Abmelden eines Objektes beim System. Bei der Registrierung wird das übergebene Objekt auf Methoden untersucht, welche mit @EventHandler annotiert wurden, validiert diese und speichert sie in einem internen Methoden Cache. Die Validierung ist nötig, da gefundene Methoden nur exakt einen Parameter als Event aufweisen dürfen.

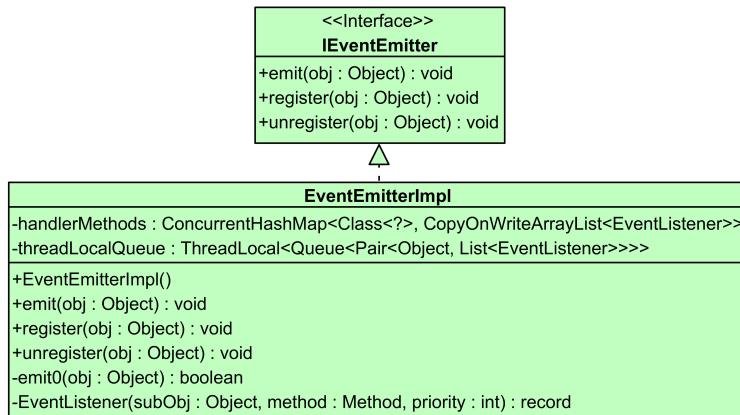


Abbildung 5.6.: Diagramm – IEventEmitter Schnittstelle und Implementierung

Mit der IEventEmitter#emit Methode, wird dem System ein neues Event übermittelt und alle Methoden, welche für dieses Event registriert wurden, nach Priorität der Annotation aufgerufen. Die daraus resultierenden Methodenaufrufe werden dabei auf dem aufrufenden Thread ausgeführt und blockieren diesen, bis alle Aufrufe abgeschlossen wurden.

5.2.14. Paket: events

Das events Paket stellt eine Vielzahl an Standard-Events für beispielsweise den Lebenszyklus der Applikation (InitEvent, StartEvent, StopEvent) oder für Statusaktualisierungen des Preloaders (StateChangeEvent, ProgressEvent) bereit. Diese können durch eine IEventEmitter genutzt werden.

5.2.15. Paket: application

Die von SimpliFX verwalteten JavaFX Applikation- und Preloader-Klassen sowie die dafür benötigten Annotation sind im `application` Paket enthalten und definieren alle Methoden, welche durch die Applikation- respektive Preloaderklasse vererbt werden (siehe Abbildung 5.7). Dabei wird bei der Erstellung der Klassen, eine für den Typ des Einstiegspunktes spezifische `IEventEmitter` Instanz übergeben, welche etwaige interne Methodenaufrufe in Form von vordefinierten Events an den vom Entwickler definierten, Einstiegspunkt delegiert.

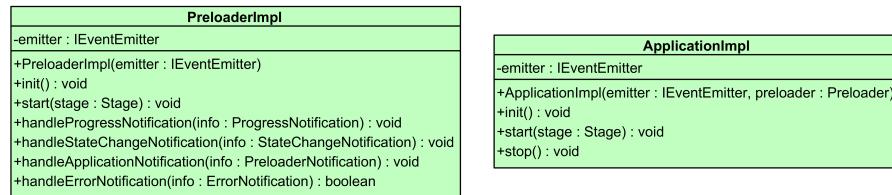


Abbildung 5.7.: Diagramm – Applikation und Preloader.

5.2.16. Paket: config

Klassen für das Verwalten von Konfigurationsdateien und deren Injektion in Controllern sind im `config` Paket aufzufinden. Dabei werden ausschließlich Dateiformate unterstützt, welche mit der Java Properties API kompatibel sind.

5.3. Essentielle Quelltextausschnitte

In den folgenden Unterkapiteln werden wichtige Quelltextausschnitte, welche für Basisfunktionen von SimpliFX verantwortlich sind, dargestellt und näher erklärt.

5.3.1. SimpliFXMLLoader

Der `SimpliFXMLLoader` unterstützt dieselbe XML-Syntax wie der `FXMLLoader`. Der einzige Unterschied liegt in der Verwendung von Übersetzungsschlüsseln aus Übersetzungsdateien. Das eigentliche Präfix zur Übersetzung („%“) wird automatisch durch den Loader in ein dynamisches Binding konvertiert. Ist die ursprüngliche Funktionalität der Übersetzung für bestimmte Elemente in der FXML Datei gewünscht, so kann das Präfix mit dem Voranstellen eines „!“ erweitert werden. Der `SimpliFXMLLoader` unterstützt keine direkte Angabe von `ResourceBundle` Instanzen in den `load` Methoden oder den Konstruktoren. Stattdessen kann eine `I18N` Instanz übergeben werden. Um eine dynamische Übersetzung zu ermöglichen, musste das Behandeln von gefundenen Übersetzungsschlüsseln teilweise neu implementiert werden. Dazu wurde unter anderem die interne Methode `Element#resolvePrefixedValue` (Zeilen 430–437) abgeändert. Ein einfaches Nutzen der `ResourceBundle#getString` Methode ist für die gewünschte Dynamik ausgeschlossen. Stattdessen wurde eine neue innere Klasse definiert, welche

das jeweils erstellte `StringBinding`, sowie den eigentlichen Übersetzungsschlüssel kapselt und Methoden zur Generierung von parametrisierten Schlüsseln zur Verfügung stellt (Abbildung 5.8). Die neue Implementierung der Ressourcenbehandlung ist in Code 5.3 dargestellt.

TranslatableBuilderProperty
-translationBinding : StringBinding
-key : String
-TranslatableBuilderProperty(translationBinding : StringBinding, key : String)
-createWithParameters(controller : Object, fxId : String, propertyName : String, key : String, ii18N : II18N) : TranslatableBuilderProperty

Abbildung 5.8.: Diagramm – Klasse zur Datenkapselung von Übersetzungsinformationen

Der Zugriff auf interne Properties von einfachen JavaFX-Komponenten wie zum Beispiel `Button` oder `Label` wird durch einen `BeanAdapter` ermöglicht, welcher automatisch durch den `FXMLLoader` erstellt wird. Mit diesem Adapter können Property-Felder auch nach Instanziierung der eigentlichen Komponentenklasse modifiziert und ausgelesen werden. Komplexe Komponenten wie `AreaChart`, werden mithilfe von `ProxyBuilder` Instanzen erstellt. Daraus resultiert, dass zum Zeitpunkt der Ressourcenbehandlung noch kein Objekt der Klasse instanziert worden ist und trivialerweise kein Zugriff auf die internen Properties möglich ist. Aus dem Grunde wird am Anfang der Behandlung überprüft, ob ein `BeanAdapter` für die aktuelle JavaFX-Komponente erstellt worden ist und anhand dessen das weitere Vorgehen bestimmt. Ist ein solcher Adapter präsent, kann ein `StringBinding` für die aktuelle Property aus der FXML Datei erstellt und bei vorhandenem Controller direkt mit eventuellen Parametern verknüpft werden. Andernfalls ist eine Erstellung eines `StringBindings` für den Schlüssel an dieser Stelle nicht möglich. Für letztere Fälle wurden vereinzelte andere Quelltextabschnitte des Loaders modifiziert, auf welche hier aber nicht näher eingegangen wird.

```

if (valueAdapter != null) {
    ObservableValue<?> val = valueAdapter.getPropertyModel("id");
    if (val instanceof StringProperty property && controller != null
        && propertyName != null) {
        return TranslatableBuilderProperty
            .createWithParameters(controller, property.get(),
                propertyName, aValue, ii18N);
    }
    return new TranslatableBuilderProperty(ii18N
        .createBindingForKey(aValue), aValue);
} else {
    return new TranslatableBuilderProperty(null, aValue);
}

```

Code 5.3: Implementierung – Ressourcenbehandlung im `SimpliFXMLLoader`

5.3.2. Erweiterbare Abhängigkeitsinjektion

Im Rahmen der Erweiterbarkeit (Anforderung 26) wurde das System zur Abhängigkeitsinjektion weitgehend abstrakt gehalten. Wie in Unterabschnitt 5.2.2 erläutert und in Anhang B beispielhaft implementiert, ist es möglich neue Bibliotheken zur Laufzeitinjizierung hinzuzufügen, ohne dass SimpliFX Komponenten abgeändert werden müssen. Dazu werden alle Annotationen des Einstiegspunktes auf die Meta-Annotation `@DIAAnnotation` überprüft. Ein exemplarischer Quelltextausschnitt, welcher dieses Konzept implementiert, ist in Code 5.4 zu erkennen. Eine ähnliche Implementierung ist in SimpliFX verwendet worden.

```
// Besitzt die Annotation die Meta-Annotation @DIAAnnotation?
if (annotation.annotationType().isAnnotationPresent(DIAAnnotation.class)) {
    // Finde IDIEnvironmentFactory Implementierung durch Annotationsparameter
    var factory = annotation.annotationType().getAnnotation(DIAAnnotation.class)
        .value();
    // Starte Reflection-Prozess mit Factory als Einstiegspunkt
    ClassReflection classRef = Reflection.reflect(factory);
    // Finde Standardkonstruktor
    Optional<ConstructorReflection> constructorRefOpt = classRef.hasConstructor();
    if (constructorRefOpt.isEmpty()) {
        // Fehler - Kein Standardkonstruktor gefunden
        break;
    }
    // Erstelle neue Instanz der Factory
    IDIEnvironmentFactory<?> factoryInstance = constructorRefOpt.get()
        .instantiateUnsafeAndGet();
    // Finde einzige Methode der Factory
    MethodReflection methodRef = Reflection.reflect(factoryInstance)
        .reflectMethod("create", Object.class,
            (Class<?>) ((ParameterizedType) factory.getGenericInterfaces()[0])
                .getActualTypeArguments()[0]);
    // Erstelle eine neue DIEnvironment Instanz
    DIEnvironment env = methodRef.invokeUnsafe(applicationListener, annotation);
    break;
}
```

Code 5.4: Implementierung – Abhängigkeitsinjektion

6. Prototypische Anwendung und Evaluation der Bibliothek

Im folgenden Kapitel werden Kernkonzepte von SimpliFX durch Quelltextbeispiele und eine Beispielanwendung erklärt. Außerdem werden die genutzten Funktionen mit reinen JavaFX Möglichkeiten verglichen. Falls ein solcher Vergleich, aufgrund von keinen äquivalenten Funktionen oder durch das Nichtvorhandensein dieser in JavaFX nicht möglich ist, werden nur die durch SimpliFX resultierenden Simplifizierungen erläutert. Vergleichbare Funktionen werden in verschiedenen Aspekten wie der Benutzungsfreundlichkeit oder dem Zeitaufwand gegenübergestellt und die Ergebnisse werden abschließend in einem Fazit zusammengefasst.

6.1. Entwicklung von Beispielsoftware

Bevor die Entwicklung der eigentlichen Software begonnen werden kann, müssen etwaige externe Bibliotheken wie JavaFX für SimpliFX bereitgestellt werden. Außerdem muss SimpliFX zur Kompilierzeit im Klassenpfad der zu entwickelnden Anwendung existieren. Da die Bibliothek im Github Maven-Repository verfügbar ist, wird der folgende Entwicklungsprozess sowie die Verwaltung von externen Bibliotheken durch die Verwendung von Maven unterstützt. In der `pom.xml` müssen für eine volle Funktionalität folgende Artefakte als Abhängigkeiten deklariert werden:

- `de.intelligence:simplifx-guice` für das Nutzen aller Basisfunktionen von SimpliFX und der Kompatibilität zu Guice für die Abhängigkeitsinjektion.
- `org.openjfx:javafx-controls:16` für Kontrollkomponenten wie zum Beispiel Schaltflächen.
- `org.openjfx:javafx-fxml:16` für das Verwenden von FXML Dateien.
- `org.openjfx:javafx-graphics:16` für das Darstellen von Komponenten im Szenengraph. Außerdem muss bei der Artefaktdeklaration die jeweilige Zielplattform als `classifier` angegeben werden. Ein Beispiel für die Windows-Plattform ist nachfolgend dargestellt.

```
<dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-graphics</artifactId>
```

```

<version>16</version>
<classifier>win</classifier>
</dependency>
```

- com.jfoenix:jfoenix:9.0.10 für erweiterte Designkomponenten.¹

Die vollständige pom.xml sowie alle in diesem Unterkapitel erstellten Klassen und Dateien sind im öffentlichen Github Maven-Repository unter dem Artefakt de.intelligence:demo-applications einsehbar.

6.1.1. Struktur und Funktion der Beispielanwendung

Die Anwendung soll weitgehend alle Funktionen von SimpliFX in Anspruch nehmen. Um die Konfigurationsschnittstelle sowie die Abhängigkeitsinjektion demonstrativ zu zeigen, wird eine Konfigurationsdatei definiert, welche Zugangsdaten und Verbindungsparameter zu einem imaginären Server enthält. Dazu wird ein Service erstellt, welcher für die Behandlung des Loginprozesses verantwortlich ist und durch ein Guice Modul zur Verfügung gestellt wird. Für die dynamische Lokalisierungsfunktion wird ein ResourceBundle mit dem Basisnamen Messages in sowohl der deutschen als auch der englischen Sprache erstellt. Wenn die Anwendung gestartet wird, soll eine grafische Schnittstelle für ein exemplarisches Einloggen angezeigt werden, welche Standardfunktionen wie das Eingeben der Zugangsdaten sowie die Änderung der Standardsprache zulassen soll. Die Funktion zur Sprachänderung kann dabei durch eine JavaFXMenuBar erfolgen. Wenn ein eventueller Login erfolgreich war, soll dem Entwickler eine Seitenleiste sowie ein Bereich, dessen Inhalt durch ebendiese kontrolliert wird, präsentiert werden. Die Seitenleiste soll vier Schaltflächen zur Navigation durch die Testapplikation sowie ein Textfeld, welches die Verbindungsdaten zum Server anzeigt, enthalten. Der Startcontroller wird im Folgenden als MainController bezeichnet. Dieser Controller hat als Wurzelement eine BorderPane und erstellt zwei Controller Untergruppen in diesem. Im oberen Bereich wird die titleBar Gruppe initialisiert, welche Anwendungseinstellungen unabhängig vom aktuell angezeigten Controller bereitstellt und im Mittelbereich die mainContent Gruppe, welche die Darstellung der eigentlichen Anwendung übernimmt und beim Start den LoginController als aktiven Controller anzeigt. Die mainContent Gruppe beinhaltet außerdem den MainMenuController, welcher wiederum eine linke Seitenleiste sowie einen Bereich für andere Inhalte neben dieser verwaltet. Eine Übersicht aller verwendeten Controller und Controllergruppen ist in Abbildung 6.1 abgebildet. Blaue Kreise sind dabei Gruppen, rot hervorgehobene Controller stellen den Startcontroller in der jeweiligen Gruppe dar und Linien, welche einen oder mehrere Controller verbinden, spezifizieren das Subcontroller Verhalten. Wenn beispielsweise der MainMenuController der aktive Controller in der mainContent Gruppe ist und zum LoginController gewechselt

¹Die neueste Version von JFoenix ist aufgrund der Jigsaw-API nicht direkt mit Java 16 kompatibel. Um die Bibliothek dennoch zu verwenden, werden eventuelle, auf das Modularitätssystem zurückführbare, Probleme durch das Nutzen der Reflection-Schnittstelle gelöst.

wird, werden auch entsprechende Lebenszyklusmethoden im aktiven Controller von sidebarContent und sidebar aufgerufen.

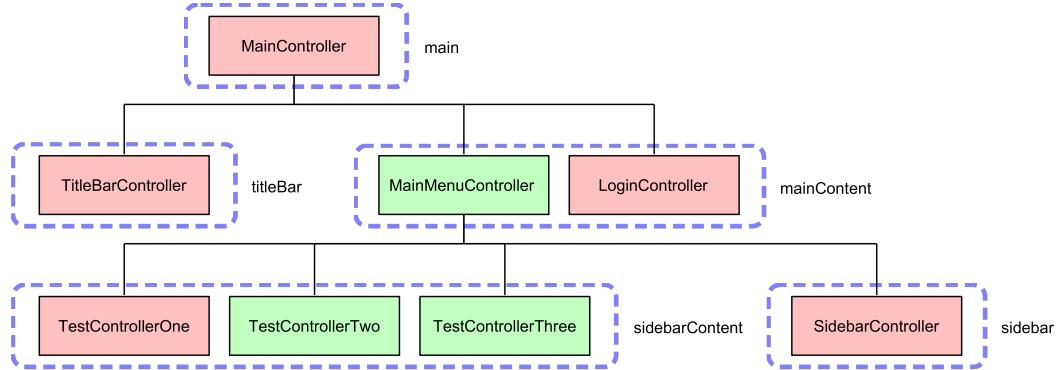


Abbildung 6.1.: Diagramm – Controller und Controllergruppen

6.1.2. Implementierung der Beispieldatenanwendung

Damit eine SimpliFX Anwendung als solche erkannt wird, muss eine Klasse definiert werden, welche als Einstiegspunkt dienen soll. Diese muss die Annotation `@ApplicationEntryPoint` aufweisen und als Parameter den Startcontroller (`MainController`) übergeben. Für die Abhängigkeitsinjektion mit Guice muss der Einstiegspunkt auch mit `@GuiceInjection` annotiert werden und die Klasse eines Guice-Moduls übergeben. Wie in der Einleitung bereits beschrieben, wird das Modul nur die Implementierung des Login Services bereitstellen (siehe Abbildung 6.2).

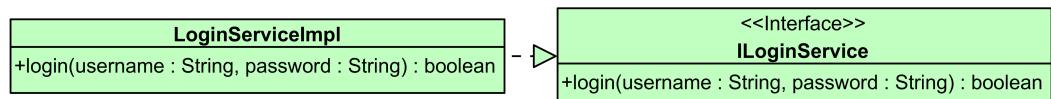


Abbildung 6.2.: Diagramm – Login Service

Dazu werden noch Konfigurationsänderungen der Stage in Form von Symbol- und Titeländerungen durchgeführt. Eine minimale Version des Einstiegspunktes ist in Code 6.1 zu sehen. Dabei wird in der ersten Zeile der Titel der Stage auf Login gesetzt und ein Symbol für die Fensterleiste angegeben. Danach wird der Hauptcontroller in der zweiten Zeile definiert, indem die jeweilige Controllerklasse als Parameter übergeben wird. Abschließend wird die Guice Kompatibilität aktiviert und ein Modul festgelegt (Zeile drei).

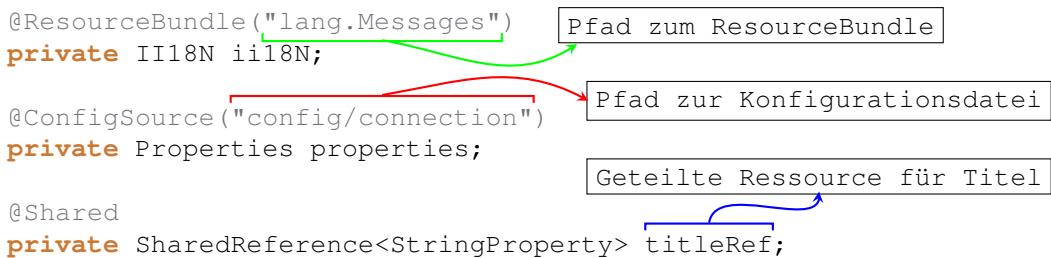
```

1  @StageConfig(title = "Login", icons = "icon.png")
2  @ApplicationEntryPoint(MainController.class)
3  @GuiceInjection(MainModule.class)
4  public final class DemoApplication { }

```

Code 6.1: Demo – Minimaler Einstiegspunkt

Für das Nutzen der dynamischen Übersetzung und der Konfigurationsschnittstelle müssen dem Einstiegspunkt zwei Felder hinzugefügt werden (Code 6.2). Außerdem sollen Subcontroller die Möglichkeit haben, den Titel der Stage abzuändern, weshalb die `titleProperty` als geteilte Ressource definiert wird und beim Start der Anwendung in einer EventHandler Methode gesetzt wird (Code 6.3).



```

@ResourceBundle("lang.Messages")
private I18N i18N;
@ConfigSource("config/connection")
private Properties properties;
private SharedReference<StringProperty> titleRef;

```

Code 6.2: Demo – Benötigte Felder

```

@EventHandler
private void onStart(StartEvent event) {
    // Titel Property setzen
    this.titleRef.set(event.getStage().titleProperty());
    // Stage zeigen
    event.getStage().show();
}

```

Code 6.3: Demo – Start EventHandler

Zum Starten der Anwendung muss der Hauptcontroller korrekt deklariert werden und eine `main` Methode existieren, welche `SimpliFX#launch` aufruft. Außerdem muss, aufgrund der Inkompatibilität zwischen JFoenix und Java 16, per Reflection-Schnittstelle eine `add-opens` Direktive vor der eigentlichen Initialisierung von SimpliFX und JavaFX hinzugefügt werden.² Der Einfachheit halber wird die `main` Methode in der `DemoApplication` Klasse definiert (Code 6.4).

²JFoenix Issue: <https://github.com/sshahine/JFoenix/issues/1052>

```
public static void main(String[] args) throws Exception {
    Reflection.addOpens("java.lang.reflect", "java.base",
        JFXTextFieldSkin.class.getModule());
    SimpliFX.launch();
}
```

Code 6.4: Demo – main Methode

Der funktionsfähige Einstiegspunkt der Anwendung ist noch einmal vollständig in Code D.1 dargestellt.

Wird zu diesem Zeitpunkt versucht, die Applikation zu starten, wird diese mit einem Fehler terminieren, da der angegebene Hauptcontroller noch nicht erstellt und konfiguriert wurde. Für eine valide Konfiguration muss die MainController Klasse die @Controller Annotation aufweisen, sowie mindestens den Pfad zu einer FXML Datei als Parameter enthalten. Außerdem wird für das Design des Layouts eine globale CSS Datei angegeben. In Code 6.5 ist der Klassenkopf des Hauptcontrollers mit allen nötigen Konfigurationen der Annotation abgebildet.³

```
@Controller(fxml = "/fxml/MainController.fxml",
            css = "css/main.css")
public final class MainController {}
```

Code 6.5: Demo – MainController Klassenkopf

Das Wurzelement ist eine BorderPane, welche in der Setup-Phase des Controllers mit der titleBar und mainContent Untergruppe initialisiert wird. Erstere Gruppe findet ihren Ursprung im oberen Teil und letztere im mittleren Bereich der BorderPane. Das Setzen der Elemente zur Aufbauphase wird in Code 6.6 gezeigt.

```
@FXML
private BorderPane root;

@Setup
private void setup(ControllerSetupContext ctx) {
    // Erstelle Untergruppe "titleBar" mit TitleBarController
    // als Startcontroller
    ctx.createSubGroup(TitleBarController.class, "titleBar",
        this.root::setTop);
    // Erstelle Untergruppe "mainContent" mit LoginController
    // als Startcontroller
    ctx.createSubGroup(LoginController.class, "mainContent",
        this.root::setCenter);
}
```

Code 6.6: Demo – MainController Setup-Phase

³Für alle weiteren Controller wird der Klassenkopf, aufgrund der fast identischen Struktur, weggelassen.

Der vollständige Quelltext des Hauptcontrollers ist in Code D.2 zu finden. Die Höhe und die Breite einer Controllergruppe richtet sich immer nach dem Startcontroller dieser. SimpliFX nutzt dazu das `prefHeight`- und `prefWidth`-Attribut des jeweiligen Wurzelementes. Ist kein solches Attribut gesetzt worden, werden Standardwerte genutzt.

Der für das Ändern der Sprache verantwortliche `TitleBarController` benötigt ein JavaFX Menu, welches durch die FXML Datei zur Verfügung gestellt wird und eine `PostConstruct` Methode, welche dieses initialisiert (Code 6.7).

```

@FXML
private Menu languageMenu;

@ResourceBundle
private I18N i18N;

@PostConstruct
private void initMenu() {
    this.i18N.setupMenu(this.languageMenu);
}

```

Code 6.7: Demo – Felder und Methoden im `TitleBarController`

Der `LoginController` übernimmt die Aufgabe eines Loginvorgangs bei einem imaginären Server. Die Zugangsdaten zu ebendiesem sind in der Konfigurationsdatei enthalten, welche im Einstiegspunkt der Anwendung mit SimpliFX verbunden wurde. Die Eingabefelder für den Benutzernamen und das Passwort werden durch den `SimpliFXMLLoader`, der `ILoginService` durch Guice und verschiedene geteilte Ressourcen durch SimpliFX in den Controller injiziert (siehe Code 6.8). Das `titleRef` Feld wird aufgrund des Feldnamens die gleiche geteilte Ressource erhalten, welche in Code 6.2 deklariert wurde.

```

@FXML
private JFXTextField usernameField;

@FXML
private JFXPasswordField passwordField;

@Shared
private SharedReference<String> usernameRef;

@Shared
private SharedReference<StringProperty> titleRef;

@Inject
private ILoginService loginService;

```

Code 6.8: Demo – Injizierte Felder des `LoginControllers`

Damit der Titel der Stage geändert wird sobald der LoginController angezeigt wird, muss eine Methode erstellt werden, welche bei der OnShow-Phase aufgerufen wird und dabei den eigentlichen Titel abändert. Des Weiteren müssen die Textfelder für das Eingeben des Benutzernamens und des Passwortes bei einem Verstecken des Controller aus Sicherheitsgründen gelöscht werden. Analog zur ersteren Methode kann dies in der OnHide Phase erfolgen (siehe Code 6.9).

```

@OnShow
private void onShow() {
    this.titleRef.get().set("Login");
}

@OnHide
private void onHide() {
    this.usernameField.clear();
    this.passwordField.clear();
}

```

Code 6.9: Demo – OnShow und OnHide Methoden

Zu diesem Zeitpunkt kann die Anwendung das erste Mal ohne einen Konfigurationsfehler gestartet werden. Der aktuelle Stand der Applikation ist in Abbildung 6.3 dargestellt. Alle bis hierhin verwendeten Controller wurden hervorgehoben.

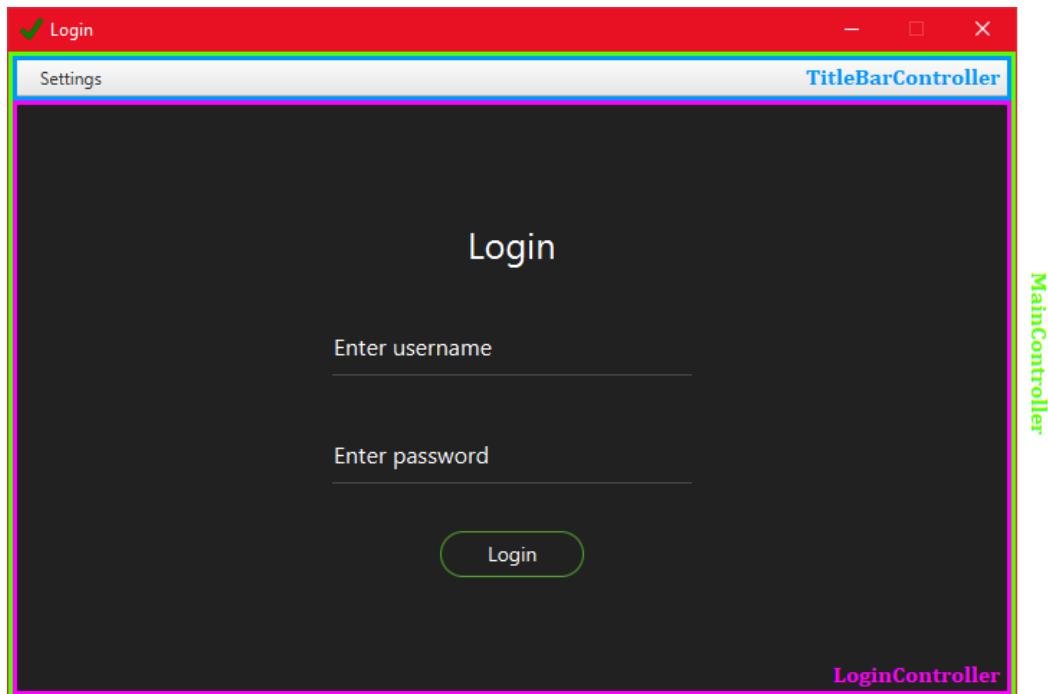


Abbildung 6.3.: Diagramm – Erster Start der Anwendung.

Nach einem Login muss der aktive Controller der mainContent Gruppe zum MainMenuController gewechselt werden. Dazu muss der LoginController zur Setup-Phase den ControllerSetupContext speichern und bei einem Klick auf die Login-Schaltfläche einen Wechsel initiieren. Die dafür nötigen Erweiterungen der Klasse sind in Code D.3 abgebildet. Der MainMenuController kapselt eine BorderPane, welche im linken Bereich die sidebar und im Mittelbereich die sidebarContent Gruppe darstellt (Code 6.10). Bei einem Wechsel soll der Stagetitel den aktuell eingeloggten Nutzer begrüßen. Die dafür benötigten Felder und Methoden sind in der vollständigen Implementierung der Klasse zu finden (Code D.4).

```

@FXML
private BorderPane root;

@FXML
private StackPane contentCenter;

@Setup
private void setup(ControllerSetupContext ctx) {
    ctx.createSubGroup(SidebarController.class, "sidebar",
        this.root::setLeft);
    ctx.createSubGroup(TestControllerOne.class, "sidebarContent",
        this.contentCenter.getChildren()::setAll);
}

```

Code 6.10: Demo – Subgruppen Einrichtung

Die sidebarContent Gruppe beinhaltet drei mögliche Controller, von welchen keiner besondere Methoden oder Felder benötigt und ein leerer Klassenrumpf teilweise für Testzwecke genügt. Aus Vollständigkeitsgründen sind diese dennoch in Code D.5 dargestellt.

Der SidebarController setzt sich aus zwei Komponenten zusammen:

- Die Serveranzeige, welche den aktuellen Server aus der Konfigurationsdatei anzeigt und per @LocalizeValue aktualisiert werden kann.
- Die Schaltflächengruppe, welche für das Wechseln des aktuellen Controllers der sidebarContent Gruppe zuständig ist und ein Ausloggen ermöglicht. Beim Ausloggen wird dabei zum LoginController gewechselt.

Das Verwenden der @LocalizeValue und der @ConfigValue Annotation wird im Folgenden beispielhaft anhand des Hostnamens demonstriert (Code 6.11).

```

@FXML
private Label connectionLbl;

@ConfigValue("host")
private String hostname;

@LocalizeValue(id = "connectionLbl", property = "text")
private StringProperty hostProperty = new SimpleStringProperty();

```

Code 6.11: Demo – @LocalizeValue und @ConfigValue

Nach der Konstruktion des Controllers muss der Stringwert der hostProperty geändert werden, um die textProperty des connectionLbl Labels mit dem aktuellen Hostnamen zu aktualisieren. Die Implementierung der Seitenleiste ist in Code D.6 und der aktuelle Zustand der Anwendung nach dem Loginvorgang in Abbildung 6.4 dargestellt.

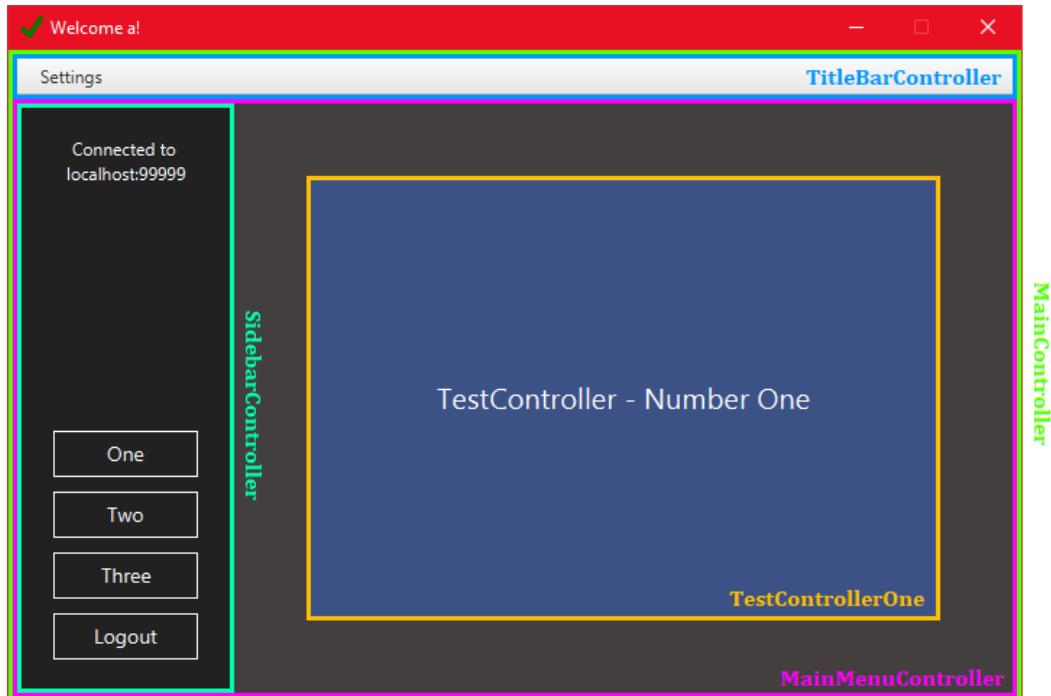


Abbildung 6.4.: Diagramm – Anwendung nach Login

6.2. Evaluation der Bibliothek

Viele der implementierten Funktionen erweitern den Funktionsumfang von JavaFX und lassen sich somit nicht direkt vergleichen. Im Folgenden werden einzelne Aspekte von SimpliFX denen von JavaFX gegenübergestellt und mögliche Vereinfachungen oder Restriktionen aufgezeigt.

6.2.1. Start der Anwendung

Der Start der Anwendung erfolgt ähnlich wie in JavaFX. Dabei wurde die interne JavaFX Launcher Implementierung vollständig durch eine eigene ersetzt, was in mehr Freiheiten bei der Applikationsinitialisierung resultiert. So ist es nun möglich, bereits initialisierte Applikationsklassen als Einstiegspunkt zuzulassen, da eine manuelle Instanziierung nicht mehr durch den Launcher vorausgesetzt ist. Dazu kann der Entwickler die Preloaderfunktion verwenden, ohne interne JavaFX Methoden aufrufen zu müssen. Des Weiteren wird der Einstiegspunkt sowohl für die Applikation als auch für den Preloader automatisch durch das Durchsuchen des Klassenpfades gefunden. Ein manuelles Angeben der Klassen ist dabei nicht mehr notwendig, sondern optional. Problematiken bei der Nutzung von JavaFX mit Apache Maven, welche auf die Jigsaw API zurückzu führen sind, werden durch SimpliFX verhindert. Die eigene Implementierung der Launcherklasse für den Anwendungsstart und das Initialisieren der JavaFX Plattform weist jedoch auch Nachteile auf. Beispielsweise wird bei in der `Application#notifyPreloader` Methode weiterhin die interne Launcher Implementierung verwendet, weshalb ein Aufruf dieser keinen Einfluss auf den von SimpliFX genutzten Preloader hat.

6.2.2. Einstiegspunkte

Die Einstiegspunkte ermöglichen eine einfache, automatische Konfiguration der Stage durch das Verwenden von Annotationen. Dazu sind die eventbasierten Methoden für das Behandeln des Applikations- bzw. Preloader-Lebenszyklus vollständig optional und können im Gegensatz zu JavaFX weggelassen werden. Eine vollständig funktionsfähige Anwendung kann mit nur zwei Klassen (Einstiegspunkt, Hauptcontroller) und drei Annotationen (`ApplicationEntryPoint`, `StageConfig`, `Controller`) realisiert werden. Ausnahmen, welche auf dem JavaFX-Thread, in Lebenszyklusmethoden oder dem Einrichten von Subsystemen auftreten, können an den Preloader weitergeleitet und durch ebendiesen behandelt werden.

6.2.3. Lokalisierung

Die Lokalisierungsmöglichkeiten in JavaFX sind strukturell durch die Implementierung des `FXMLLoader` limitiert. Der herkömmliche `FXMLLoader` erlaubt das einmalige Übersetzen in Form von Stringwerten beim Laden einer FXML Datei und verhindert somit die dynamische Änderung der Sprache ohne ein vollständiges Neuladen der jeweiligen betroffenen Dateien. Diese Limitation wurde durch den `SimpliFXMLLoader` entfernt, indem die Übersetzung nun in Form von aktualisierbaren `StringBinding` Instanzen durchgeführt wird. Auch ist die Unterstützung von parametrisierten Übersetzungsschlüsseln bei der Verwendung von JavaFX oder SimpliFX Controllern ermöglicht worden. Weiterhin können die Lokalisierungsschnittstelle und der `SimpliFXMLLoader` auch außerhalb vom Controller-System eigenständig genutzt werden.

6.2.4. Controller

Controller von SimpliFX erlauben eine Verschachtelung sowie eine Gruppierung und wurden um einen Lebenszyklus erweitert. Sie können beliebig oft mit oder ohne Verwendung von Animationen gewechselt werden und sind in der Lage, Lebenszyklusänderungen an eventuelle Untercontroller zu delegieren. Außerdem ist es möglich verschiedene Objekte als geteilte Ressourcen zu definieren, welche von allen aktiven Controllern genutzt werden können. Eine Änderung dieser wird auf alle anderen Controller reflektiert. Jede Controllergruppe ist dabei im Stande Benachrichtigungen anzuzeigen. Der Entwickler kann das Layout dieser vollständig kontrollieren oder das Standardlayout verwenden.

6.2.5. Abhängigkeitsinjektion

JavaFX besitzt keine direkte Möglichkeit für eine Abhängigkeitsinjektion durch etablierte, externe Bibliotheken. SimpliFX bringt dieses Konzept in die JavaFX Umgebung und ermöglicht somit das Verwenden solcher Bibliotheken in allen Controllern und Einstiegspunkten. Soll eine Bibliothek verwendet werden, welche nicht direkt durch SimpliFX bereitgestellt wird, kann diese mit wenig Zeitaufwand hinzugefügt werden.

6.2.6. Konfigurationsdateien

Alle Konfigurationsdateien, welche eine Kompatibilität mit der Java Properties API aufweisen, können durch SimpliFX aus dem Klassenpfad sowie aus externen Quellen geladen und den Einstiegspunkten und Controllern zur Verfügung gestellt werden. Dabei ist die Verwendung von Standardwerten möglich, wenn Konfigurationsschlüssel nicht in den angegebenen Konfigurationsdateien gefunden werden können.

7. Fazit

In dieser Arbeit wurde eine auf Java Annotationen basierende Bibliothek zur Simplifizierung und Expansion von JavaFX konzipiert und implementiert. Vorhandene Funktionen wurden in einer Problemanalyse auf Komplexität und Vollständigkeit untersucht und die daraus resultierenden Probleme dienten als Grundlage für die Anforderungsanalyse. Die identifizierten funktionalen und nicht-funktionalen Anforderungen wurden für die Entwicklung eines Konzeptes genutzt, welches wiederum das Fundament der vollständigen Systemimplementierung bildete und dabei wurde ein besonderer Fokus auf die explizite Erfüllung aller Ziele gesetzt. In der prototypischen Implementierung wurden alle erforderlichen Anforderungen mit spezieller Berücksichtigung von paradigmatischen softwaretechnischen Qualitätsrichtlinien wie der Erweiterbarkeit und der Wartbarkeit eines Systems erfüllt. Abschließend wurde der Funktionsumfang von JavaFX mit dem von SimpliFX im Rahmen der Evaluation verglichen.

7.1. Zusammenfassung und Bewertung

Gemäß der in der Einleitung beschriebenen Zielsetzung wurde eine Bibliothek entwickelt, welche das Arbeiten und die Entwicklung von Applikationen mit JavaFX in einigen Kernkonzepten vereinfacht (siehe Abschnitt 1.2). Besonders die Sprachunterstützung und die dafür benötigte dynamische Übersetzung durch einen neuen FXMLLoader sowie eine Erweiterung der Funktionen für eine Erstellung und Verwaltung von Controllern, sind ein fundamentaler Bestandteil der Bibliothek. Des Weiteren können Konfigurationsdateien und Sprachdateien mit Leichtigkeit durch Annotationsverwendung geladen und nach Initialisierung der Anwendung genutzt werden. Außerdem wurden Schnittstellen hinzugefügt, welche die Interaktion zu komplexen Systemen wie der Java Reflection API vereinfachen und effektiv die Fehleranfälligkeit dieser durch eine zentralisierte Fehlerbehandlung reduzieren. Insgesamt wurden im Rahmen der Anforderungsanalyse (siehe Abschnitt 4.2) 28 Anforderungen an die Bibliothek gestellt. Diese setzen sich aus 23 fundamentalen und fünf optionalen Anforderungen zusammen, von welchen jeweils 23 bzw. eine erfüllt worden sind. Daraus resultiert eine Erfüllquote von 100 Prozent der fundamentalen Anforderungen und eine von 20 Prozent bei den optionalen Anforderungen. Zusammenfassend wurden demnach 85 Prozent aller Anforderungen erfüllt. Wenn eine Anforderung nicht erfüllt wurde, lag dies am Umfang, welcher dann mit einem hohen Zeitaufwand einhergehen würde. In Tabelle 7.1 ist eine Gesamtübersicht dieser Anforderungen mit den jeweiligen Erfüllungsstatus zu finden. Auch wenn es sich bei der entwickelten Bibliothek keinesfalls um ein vollkommen abgeschlossenes Projekt

handelt und bei der aktiven Entwicklung von Testfällen und der generellen Erhöhung der Testabdeckung mit hoher Wahrscheinlichkeit noch Fehler im Quelltext detektiert werden, kann diese dennoch für andere Projekte verwendet werden und Neueinsteigern in die Java- bzw. JavaFX-Welt als Hilfestellung dienen.

Nr.	Typ	Anforderung	Opt.	Erfüllt
1	F	Ermitteln und Konfiguration des JavaFX-Einstiegspunktes	X	✓
2	F	Eventbasierte Verwaltung des Applikationslebenszyklus	X	✓
3	F	Dynamische Laufzeitübersetzung von JavaFX-Komponenten	X	✓
4	F	Abhängigkeitsinjektion durch etablierte externe Bibliotheken	X	✓
5	F	Automatische CSS Metadaten Generation	X	✓
6	F	Annotationsbasierte Controllerdefinition	X	✓
7	F	Erweiterung des Controllerlebenszyklus	X	✓
8	F	Benachrichtigung bei Komplettierung der Einrichtung von Einstiegspunkten und Controllern	X	✓
9	F	Unterstützung von mehreren Lebenszyklusbehandlungsmethoden	X	✓
10	F	Controllerverschachtelung und Supercontroller	X	✓
11	F	Gruppierung von ähnlichen Controllern	X	✓
12	F	Zyklusprävention	X	✓
13	F	Wechsel von Controllern	X	✓
14	F	Controller Preloading	X	✓
15	F	Globale Fehlerbehandlung	X	✓
16	F	Geteilte Ressourcen	X	✓
17	F	Konfigurationsdateien	X	✓
18	F	Anpassbare Events durch Annotationen	✓	X
19	F	Serialisierung von JavaFX Komponenten	✓	X
20	F	Scheduling von Methoden	✓	X
21	F	JavaFX Application-Thread Forcierung	✓	✓
22	F	Annotationsvalidierung zur Kompilierzeit	✓	X
23	NF	Benutzungsfreundlichkeit	X	✓
24	NF	Performance & Effizienz	X	✓
25	NF	Wartbarkeit	X	✓
26	NF	Erweiterbarkeit	X	✓
27	NF	Softwarequalität	X	✓
28	NF	Lizenzierung	X	✓

Tabelle 7.1.: Anforderungsliste

7.2. Ausblick und mögliche Erweiterungen

Im Folgenden werden mögliche Erweiterungen der implementierten Bibliothek vorgestellt. Auch wenn ein Großteil der Anforderungen durch das System erfüllt wird, existieren zu diesem Zeitpunkt noch einige optionale Anforderungen, welche nur teilweise bis gar nicht implementiert worden sind. Dazu gehört beispielsweise eine Annotationsvalidierung zur Kompilierzeit einer auf SimpliFX basierenden Applikation (Anforderung 22), was zu einer Detektion von Syntaxfehlern oder eventueller Inkorrekttheiten in Konfigurationen führt. Wird eine solche Fehlkonfiguration entdeckt, kann ein Kompilierfehler durch einen Annotationsprozessor ausgelöst und Laufzeitausnahmen somit effektiv vermieden werden.

Die Unterstützung von Konfigurationsdateien ist in den Aspekten des Dateiformats und der zugelassenen Operationen stark begrenzt. Neben dem Akzeptieren von Properties- und XML-Dateien, könnten beispielsweise noch weitere bekannte Konfigurationsformate wie JSON oder YAML Ain't Markup Language (YAML) durch SimpliFX erkannt und genutzt werden. Auch ist es momentan nicht möglich, eine schreibende Operation auf Konfigurationsdateien vorzunehmen, da zwischen Ressourcen im Klassenpfad und externen Ressourcen differenziert werden müsste und es generell kompliziert ist, Dateien zu modifizieren, welche sich innerhalb eines Java Archivs befinden.

Das System könnte ebenfalls um vom Betriebssystem des Nutzers abhängige Funktionen erweitert werden. Das Fensterdesign für eine JavaFX Stage kann durch die Nutzung des Java Native Interface (JNI) ergänzt werden. Unter Windows kann so mit beispielsweise ein Unschärfeeffekt des Fensterhintergrundes realisiert oder eine Modifikation der Titelleiste ermöglicht werden. Auch könnte eine Schnittstelle entwickelt werden, welche eine direkte Interaktion mit der Taskleiste ermöglicht und so zum Beispiel Benachrichtigungen und Statusaktualisierungen an diese übermitteln kann.

Steht eine Plattformunabhängigkeit in Vordergrund, könnte das System alternativ Titelleisten bereitstellen, welche ausschließlich durch JavaFX Komponenten konstruiert werden und auf einer nicht dekorierten oder transparenten Stage anwendbar sind. Das Fensterdesign ist somit nicht vom genutzten Betriebssystem abhängig, sondern auf jedem von JavaFX unterstützten System identisch. Basisoperationen wie das Ändern der Fenstergröße oder das Verschieben müssen dann jedoch manuell implementiert werden.

Die Ausführung von EventHandler Methoden wird derzeit auf dem aufrufenden Thread durchgeführt und blockiert diesen dadurch. Auf der einen Seite ist dieses Verhalten vorteilhaft, da nach dem Senden eines Events auf die Beendigung der Eventbehandlung gewartet wird, aber auf der anderen Seite ist in Systemen mit einem hohen Grad an Parallelität eine asynchrone Ausführung der Behandlungsmethoden gewünscht. Aufgrund dessen könnte zu der von SimpliFX bereitgestellten `IEventEmitter` Implementierung eine weitere für ausschließlich asynchrone Operationen erstellt oder die vorhandene um eine Asynchronitätsunterstützung erweitert werden.

Nicht zuletzt kann ein Ausbau der Funktionalitäten des Controllersystems durchgeführt werden. Beispielsweise können Controller in einer Controllergruppe durch die JavaFX Pagination Klasse oder ähnliche Implementierungen eine Anzeigereihenfolge zugewiesen bekommen, welche den Controllerwechsel aufgrund der prädestinierten Reihenfolge erleichtert. Auch ist eine Art Verlaufsspeicherung möglich, um solche Wechsel zu speichern und gegebenenfalls eine Zurückfunktion zu realisieren. Ein JavaFX Button kann dann mit @Previous annotiert werden und bei einem ActionEvent den vorherigen Controller anzeigen.

Auch wenn im Rahmen dieser Arbeit explizit auf eine Modifikation der FXML Syntax verzichtet wurde, um einen Einfluss auf die Kompatibilität von externen Tools wie dem SceneBuilder oder Entwicklungsumgebungen wie IntelliJ zu vermeiden, ist die Erweiterbarkeit des SimpleXMLLoader theoretisch uneingeschränkt. Die Einrichtung der Controller und Controllergruppen in der Setup-Phase könnte aus dem Javaquelltext in die FXML Datei ausgelagert werden, um eine dementsprechend automatische Konstruktion zur Laufzeit des Programms zu ermöglichen. Darüber hinaus kann die Preloader Funktion von JavaFX für den Entwickler zugänglicher gemacht werden. Wird beispielsweise ein Preloader erstellt um einen aktiven Ladeprozess in Form einer Fortschrittsanzeige darzustellen, muss diese bei einer Fortschrittsaktualisierung durch eine ProgressNotification manuell benachrichtigt werden. Der Ladeprozess und die dazugehörige Erstellung von Benachrichtigungen kann durch ein Aufgabensystem verwaltet werden. Dazu wird ein ExecutorService erstellt, welcher alle registrierten Aufgaben (Laden von Ressourcen, Überprüfen und Download von neuen Applikationsversionen, etc.) sequentiell ausführt und automatisch das nötige Inkrement des aktuellen Ladefortschrittes berechnet und an den jeweiligen Preloader übermittelt.

Auch wenn bereits eine Evaluation durch die Entwicklung einer Testapplikation erfolgt ist, kann eine weitere Zielüberprüfung durch die Durchführung einer Nutzerstudie mit beispielsweise Studentengruppen vorgenommen werden.

A. Controllerbasierte JavaFX-Anwendung

```
package de.testpackage;

import javafx.fxml.FXML;
import javafx.scene.control.Button;

public final class TestController {

    @FXML
    private Button testBtn;

    @FXML
    private void onTestBtnClick() {
        // do something
    }
}
```

Code A.1: Beispiel – Controller

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.Pane?>

<Pane xmlns="http://javafx.com/javafx" xmlns:fx="http://javafx.com/fxml"
      fx:controller="de.testpackage.TestController"
      stylesheets="test.css">
    <Button fx:id="testBtn" onAction="#onTestBtnClick">TestButton</Button>
</Pane>
```

Code A.2: FXML-Layout (test.fxml)

```
#testBtn {
    -fx-background-color: red;
}
```

Code A.3: CSS-Design (test.css)

```
package de.testpackage;

import javafx.application.Application;

public static final class TestApplication extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    private Pane loadFXML(URL fxmlPath) throws IOException {
        return new FXMLLoader(fxmlPath).load();
    }

    @Override
    public void start(Stage primaryStage) {
        URL fxmlPath = this.getClass().getResource("test.fxml");
        Pane pane = null;
        try {
            pane = this.loadFXML(fxmlPath);
        } catch(IOException ex) {
            // error handling
            return;
        }
        Scene scene = new Scene(pane, 500, 500);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Code A.4: Beispiel – Anwendungscode

B. Hinzufügen einer Bibliothek für die Abhängigkeitsinjektion

```
public class TestEnvironment implements DIEnvironment {
    public TestEnvironment(Object obj, Class<?>[] modules) {
        // Initialisiere Bibliothek
    }

    @Override
    public void inject(Object obj) {
        // Injiziere Abhängigkeiten in Objekt-Instanz
    }

    @Override
    public <T> T get(Class<T> clazz) {
        // Erstelle neue Instanz der übergebenen Klasse
    }
}
```

Code B.1: Beispiel – Erstellen einer neuen Umgebung

```
public class TestEnvironmentFactory implements
    IDIEnvironmentFactory<TestInjection> {
    @Override
    public DIEnvironment create(Object obj, TestInjection annotation) {
        // Erstellen einer neuen Instanz der Umgebung
        return new TestEnvironment(obj, annotation.value());
    }
}
```

Code B.2: Beispiel – Erstellen einer neuen Factory

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@DIAAnnotation(TestEnvironmentFactory.class)
public @interface TestInjection {
    Class<?>[] value();
}
```

Code B.3: Beispiel – Erstellen einer neuen Annotation

C. Schnittstellen und Beziehungen des Controller-Systems

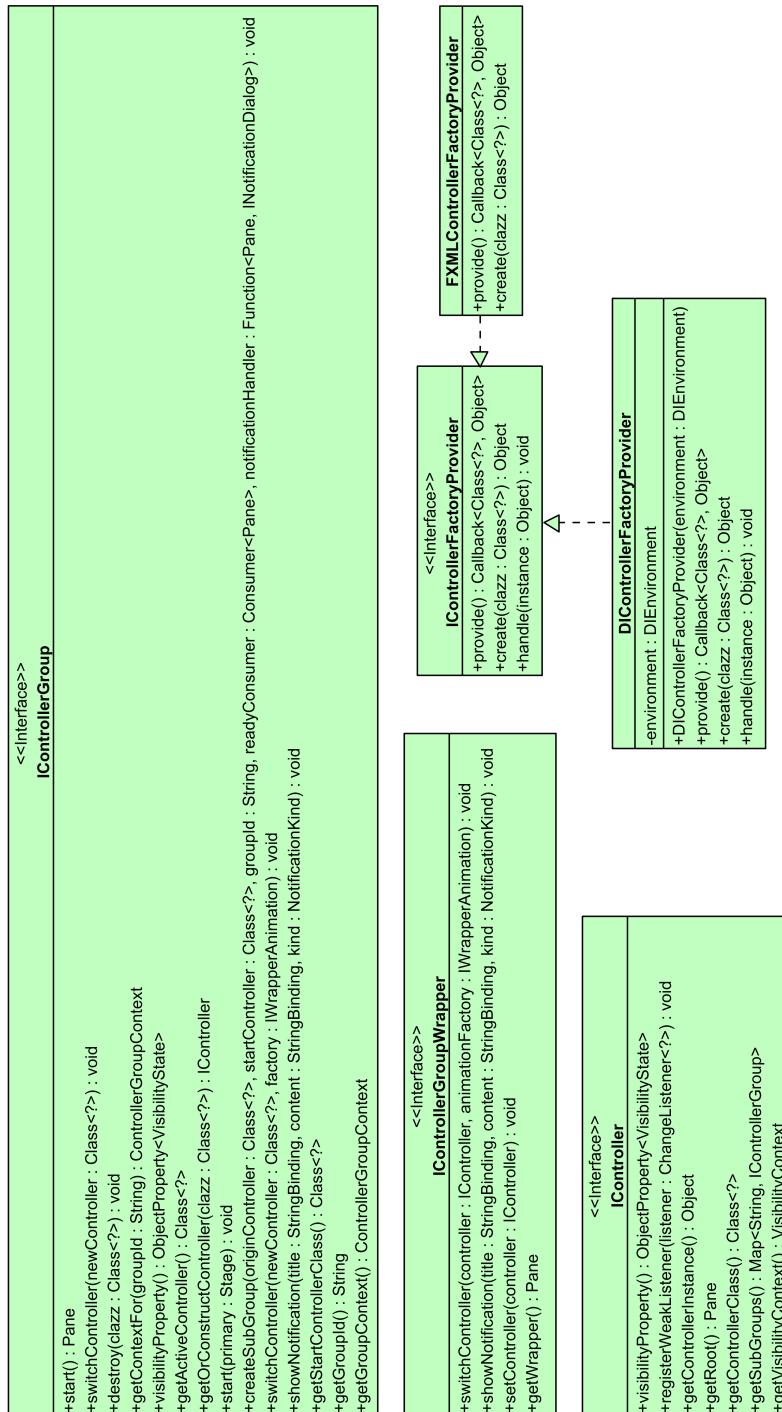


Diagramm – Controller-System: Einstiegspunkt und Beziehungen

D. Demo Anwendung

```
1  @StageConfig(title = "Login", icons = "icon.png")
2  @ApplicationEntryPoint(MainController.class)
3  @GuiceInjection(MainModule.class)
4  public final class DemoApplication {
5
6      public static void main(String[] args) throws Exception {
7          Reflection.addOpens("java.lang.reflect", "java.base",
8              JFXTextFieldSkin.class.getModule());
9          SimpliFX.launch();
10     }
11
12     @ResourceBundle("lang.Messages")
13     private II18N ii18N;
14
15     @ConfigSource("config/connection")
16     private Properties properties;
17
18     @Shared
19     private SharedReference<StringProperty> titleRef;
20
21     @EventHandler
22     private void onStart(StartEvent event) {
23         this.titleRef.set(event.getStage().titleProperty());
24         event.getStage().show();
25     }
26
27 }
```

Code D.1: Demo – Minimaler Einstiegspunkt

```
1 @Controller(fxml = "/fxml/MainController.fxml",
2             css = "css/main.css")
3 public final class MainController {
4
5     @FXML
6     private BorderPane root;
7
8     @Setup
9     private void setup(ControllerSetupContext ctx) {
10        // Erstelle Untergruppe "titleBar" mit TitleBarController
11        // als Startcontroller
12        ctx.createSubGroup(TitleBarController.class, "titleBar",
13                           this.root::setTop);
14        // Erstelle Untergruppe "mainContent" mit LoginController
15        // als Startcontroller
16        ctx.createSubGroup(LoginController.class, "mainContent",
17                           this.root::setCenter);
18    }
19
20 }
```

Code D.2: Demo – MainController

D. Demo Anwendung

```
1  @Controller(fxml = "/fxml/LoginController.fxml")
2  public final class LoginController {
3
4      @FXML
5      private JFXTextField usernameField;
6
7      @FXML
8      private JFXPasswordField passwordField;
9
10     @Shared
11     private SharedReference<String> usernameRef;
12
13     @Shared
14     private SharedReference<StringProperty> titleRef;
15
16     @Inject
17     private ILoginService loginService;
18
19     private ControllerGroupContext ctx;
20
21     @Setup
22     private void setup(ControllerSetupContext ctx) {
23         ctx.preloadController(MainMenuController.class);
24         this.ctx = ctx.getGroupContext();
25     }
26
27     @OnShow
28     private void onShow() {
29         this.titleRef.get().set("Login");
30     }
31
32     @OnHide
33     private void onHide() {
34         this.usernameField.clear();
35         this.passwordField.clear();
36     }
37
38     @FXML
39     private void onLogin() {
40         if (loginService.login(this.usernameField.getText(),
41             this.passwordField.getText())) {
42             this.usernameRef.set(this.usernameField.getText());
43             this.ctx.switchController(MainMenuController.class,
44                 new FadeAnimation(Duration.millis(250)));
45         }
46     }
47
48 }
```

Code D.3: Demo – LoginController

```
1  @Controller(fxml = "/fxml/MainMenuController.fxml")
2  public final class MainMenuController {
3
4      @FXML
5      private BorderPane root;
6
7      @FXML
8      private StackPane contentCenter;
9
10     @ ResourceBundle
11     private I18N i18N;
12
13     @ Shared
14     private SharedResources resources;
15
16     private StringBinding binding;
17
18     @ Setup
19     private void setup(ControllerSetupContext ctx) {
20         ctx.createSubGroup(SidebarController.class, "sidebar",
21             this.root::setLeft);
22         ctx.createSubGroup(TestControllerOne.class,
23             "sidebarContent",
24             this.contentCenter.getChildren()::setAll);
25     }
26
27     @ PostConstruct
28     private void afterConstruction() {
29         this.binding = this.i18N
30             .createObservedBinding("mainMenu.welcome",
31                 this.resources.getForName("username")
32                     .asProperty());
33     }
34
35     @ OnShow
36     private void onShow() {
37         final SharedReference<StringProperty> prop =
38             this.resources.getForName("title");
39         prop.get().set(this.binding.get());
40     }
41
42 }
```

Code D.4: Demo – MainMenuController

D. Demo Anwendung

```
1  @Controller(fxml = "/fxml/TestControllerOne.fxml")
2  public static final class TestControllerOne {
3
4      @Setup
5      private void onSetup(ControllerSetupContext ctx) {
6          ctx.preloadController(TestControllerTwo.class);
7          ctx.preloadController(TestControllerThree.class);
8      }
9
10 }
11
12 @Controller(fxml = "/fxml/TestControllerTwo.fxml")
13 public static final class TestControllerTwo {}
14
15 @Controller(fxml = "/fxml/TestControllerThree.fxml")
16 public static final class TestControllerThree {}
```

Code D.5: Demo – Alle TestController

```

1  @Controller(fxml = "/fxml/SidebarController.fxml")
2  public final class SidebarController {
3
4      @FXML
5      private Label connectionLbl;
6
7      @ConfigValue("host")
8      private String hostname;
9
10     @ConfigValue(value = "port")
11     private int port;
12
13     @LocalizeValue(id = "connectionLbl", property = "text")
14     private StringProperty hostProperty = new SimpleStringProperty();
15
16     @LocalizeValue(id = "connectionLbl", index = 1, property = "text")
17     private IntegerProperty portProperty = new SimpleIntegerProperty();
18
19     private ControllerGroupContext mainCtx;
20     private ControllerGroupContext sidebarContentCtx;
21
22     @Setup
23     private void onSetup(ControllerSetupContext ctx) {
24         this.mainCtx = ctx.getContextFor("mainContent");
25         this.sidebarContentCtx = ctx.getContextFor("sidebarContent");
26     }
27
28     @PostConstruct
29     private void afterConstruction() {
30         this.hostProperty.set(this.hostname);
31         this.portProperty.set(this.port);
32     }
33
34     @OnHide
35     private void onHide() {
36         sidebarContentCtx.switchController(TestControllerOne.class);
37     }
38
39     @FXML
40     private void onLogoutPressed() {
41         this.mainCtx.switchController(LoginController.class,
42             new BottomSlideAnimation(Duration.millis(250)));
43     }
44
45     // Aus Platzgründen sind onTwoPressed() und onThreePressed()
46     // nicht dargestellt worden
47     @FXML
48     private void onOnePressed() {
49         sidebarContentCtx.switchController(TestControllerOne.class,
50             new TopSlideAnimation(Duration.millis(250)));
51     }
52
53 }
```

Code D.6: Demo – SidebarController

Abkürzungsverzeichnis

AWT Abstract Window Toolkit

CSS Cascading Style Sheets

DLL Dynamic Link Library

IoC Inversion of Control

JAXB Jakarta XML Binding

JDK Java Development Kit

JNI Java Native Interface

JSON JavaScript Object Notation

LoC Lines of Code

MVC Model-View-Controller

MVVM Model-View-ViewModel

SoC Separation of Concerns

UML Unified Modeling Language

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Quellcodeverzeichnis

2.1.	Beispiel – Minimale JavaFX-Anwendung	11
2.2.	Beispiel – ChangeListener & EventHandler	12
2.3.	Beispiel – FXML Layouting	13
2.4.	Beispiel – FXML Ladeprozess	13
2.5.	Beispiel einer Annotationsdefinition	14
2.6.	Beispiel einer annotierten Klasse	14
2.7.	Deklaration – Normal Annotation	15
2.8.	Anwendung – Normal Annotation	15
2.9.	Deklaration – Single-Element Annotation	15
2.10.	Anwendung – Single-Element Annotation	15
2.11.	Deklaration – Marker Annotation	16
2.12.	Anwendung – Marker Annotation	16
2.13.	Beispiel einer Laufzeit Annotation	16
2.14.	Auslesen einer Laufzeit-Annotation	17
2.15.	Beispiel – Annotationsprozessor	18
3.1.	Beispiel – Interfacedeklaration	20
3.2.	Beispiel – Kompilierfehler	20
3.3.	Beispiel – Lombok POJO	21
3.4.	Repräsentation als XML Datei	21
3.5.	Repräsentation als Java Objekt	21
4.1.	Nutzung des Schlüssels in einer FXML Datei	26
4.2.	Beispiel – Controller mit injizierten Diensten	27
4.3.	Beispiel – Instanziierungsablauf	29
4.4.	Beispiel – Verwendung der Reflection Schnittstelle	41
4.5.	Beispiel – Erstellen einer I18N Instanz	43
5.1.	Beispiel – Nutzung der Pair Klasse	55
5.2.	Beispiel – Initiierung eines Klassenpfadscans.	58
5.3.	Implementierung – Ressourcenbehandlung im SimpliFXMLLoader	61
5.4.	Implementierung – Abhängigkeitsinjektion	62
6.1.	Demo – Minimaler Einstiegspunkt	66
6.2.	Demo – Benötigte Felder	66
6.3.	Demo – Start EventHandler	66
6.4.	Demo – main Methode	67

6.5. Demo – MainController Klassenkopf	67
6.6. Demo – MainController Setup-Phase	67
6.7. Demo – Felder und Methoden im TitleBarController	68
6.8. Demo – Injizierte Felder des LoginControllers	68
6.9. Demo – OnShow und OnHide Methoden	69
6.10. Demo – Subgruppen Einrichtung	70
6.11. Demo – @LocalizeValue und @ConfigValue	71

Abbildungsverzeichnis

2.1. UML-Diagramm – Beobachter-Entwurfsmuster	8
2.2. Diagramm – MVC-Entwurfsmuster	9
4.1. Diagramm – ReflectionScopes	41
4.2. Diagramm – Klassenpfad-Scanprozess für Klassendateien	42
4.3. Diagramm – II18N Schnittstelle	43
4.4. Diagramm – Konfigurations- und Startmethoden von SimpliFX	44
4.5. Diagramm – Finden und Validierung des Einstiegspunktes	45
4.6. Diagramm – Erstellung der Hauptgruppe und des Hauptcontrollers .	46
5.1. Paketstruktur – SimpliFX	54
5.2. Diagramm – DI Schnittstellen	55
5.3. Diagramm – Einstiegspunkt des Controller-Systems	57
5.4. Diagramm – Mögliche Quellen für Klassenpfade	58
5.5. Diagramm – Klassenpfad Schnittstelle und Implementierung	58
5.6. Diagramm – IEventEmitter Schnittstelle und Implementierung . .	59
5.7. Diagramm – Applikation und Preloader.	60
5.8. Diagramm – Klasse zur Datenkapselung von Übersetzungsinforma- tionen	61
6.1. Diagramm – Controller und Controllergruppen	65
6.2. Diagramm – Login Service	65
6.3. Diagramm – Erster Start der Anwendung.	69
6.4. Diagramm – Anwendung nach Login	71

Tabellenverzeichnis

4.1. Verwendete externe Bibliotheken	39
4.2. Alle benötigten ReflectionScopes	40
7.1. Anforderungsliste	76

Literaturverzeichnis

- [AA19] ANDERSON, GAIL und PAUL ANDERSON: *The Definitive Guide to Modern Java Clients with JavaFX*, Kapitel JavaFX Fundamentals, Seiten 33–80. Stephen Chin, Johan Vos, James Weaver, 2019.
- [AJ19] ALBAHARI, JOSEPH und ERIC JOHANNSEN: *C# 8.0 in a Nutshell: The Definitive Reference*. In a Nutshell. O'Reilly Media, 2019.
- [Bur92] BURBECK, STEVE: *Applications programming in smalltalk-80: how to use model-view-controller (mvc)*. 1992.
- [CCC05] CAZZOLA, WALTER, ANTONIO CISTERNINO und DIEGO COLOMBO: *[a]C#: C# with a customizable code annotation mechanism*. In: *Proceedings of the ACM Symposium on Applied Computing*, Band 2, Seiten 1264–1268, 2005.
- [DCL01] DEITSCH, ANDREW, DAVID CZARNECKI und MIKE LOUKIDES: *Java Internationalization*. O'Reilly & Associates, Inc., USA, 2001.
- [Dea95] DEACON, JOHN: *Model-View-Controller (MVC) Architecture*. Online, 1995.
- [DPV⁺07] DANELUTTO, MARCO, MARCELO PASIN, MARCO VANNESCHI, PATRIZIO DAZZI, DOMENICO LAFORENZA und LUIGI PRESTI: *PAL: Exploiting Java Annotations for Parallelism*, Seiten 83–96. 2007.
- [FFV04] FORMAN, IRA R., NATE FORMAN und JOHN VLASSIDES: *Java Reflection in Action*, 2004.
- [Gao19] GAO, WEIQI: *The Definitive Guide to Modern Java Clients with JavaFX*, Kapitel Properties and Bindings, Seiten 81–141. Stephen Chin, Johan Vos, James Weaver, 2019.
- [GHJV94] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLASSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software*. Seiten 1–4, 293–303. Addison-Wesley Longman Publishing Co., Inc., 1994.
- [GJS⁺05] GOSLING, JAMES, BILL JOY, GUY STEELE, GILAD BRACHA, ALEX BUCKLEY, DAVID SMITH und GAVIN BIERMAN: *The Java Language Specification, Java SE 16 Edition*, Seiten 356–386. 2005.

- [Gra14] GRANT, ANDREW: *Beginning AngularJS*, Kapitel Introduction to MVC, Seiten 47–56. Apress, Berkeley, CA, 2014.
- [Hom13] HOMMEL, SCOTT: *Using JavaFX Properties and Binding*, April 2013.
<https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm> (zuletzt abgerufen am 08.09.2021).
- [HvDM⁺13] HUGHES, JOHN F., ANDRIES VAN DAM, MORGAN McGuIRE, DAVID F. SKLAR, JAMES D. FOLEY, STEVEN FEINER und KURT AKELAY: *Computer Graphics: Principles and Practice*, Kapitel Scene Graphs, Seiten 351–353. Addison-Wesley, Upper Saddle River, NJ, 3 Auflage, 2013.
- [Jun13] JUNEAU, JOSH: *Java EE 7 Recipes: A Problem-Solution Approach*, Kapitel JavaFX in the Enterprise, Seiten 615–646. Apress, Berkeley, CA, 2013.
- [KCVM21] KHAN, FAIZAN, BOQI CHEN, DANIEL VARRO und SHANE MCINTOSH: *An Empirical Study of Type-Related Defects in Python Projects*. IEEE Transactions on Software Engineering, 2021.
- [KDSAMR18] KRUK, G., O. DA SILVA ALVES, L. MOLINARI und E. ROUX: *Best Practices for Efficient Development of JavaFX Applications*. In: *Proc. of International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS'17)*, Nummer 16, Seiten 1078–1083. JACoW, 2018.
- [Kul07] KULESHOV, EUGENE: *Using the ASM framework to implement common Java bytecode transformation patterns*. 2007.
- [Lem15] LEMNITZER, LOTHAR: *Korpuslinguistik: eine Einführung*, Seiten 57–59. 3. überarbeitete und erweiterte Auflage, 2015.
- [LTX17] LI, YUE, TIAN TAN und JINGLING XUE: *Understanding and Analyzing Java Reflection*. ACM Transactions on Software Engineering and Methodology, 28, 2017.
- [MJ09] MIROSLAV, SABO und PORUBÄN JAROSLAV: *Preserving Design Patterns using Source Code Annotations*. Journal of Computer Science and Control Systems, 2, 2009.
- [MP06] MEFFERT, KLAUS und ILKA PHILIPPOW: *Annotationen zur Anwendung beim Refactoring*, 2006.
- [MRO10] MAIER, INGO, TIARK ROMPF und MARTIN ODERSKY: *Deprecating the Observer Pattern*. 2010.

- [Ora17] ORACLE: *Java SE Specifications*, 2017.
<https://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html#jls-9.6> (zuletzt abgerufen am 08.09.2021).
- [PFS09] PORUBÄN, JAROSLAV, MICHAL FORGÁČ und MIROSLAV SABO: *Annotation based parser generator*. In: *International Multiconference on Computer Science and Information Technology*, Seiten 707–714, 2009.
- [PN15] PIGULA, PETER und MILAN NOSAL: *Unified compile-time and run-time java annotation processing*. In: *Federated Conference on Computer Science and Information Systems (FedCSIS)*, Seiten 965–975, 2015.
- [Rei05] REINEKE, DETLEF: *Einführung in die Softwarelokalisierung*. Gunter Narr Verlag, 2005.
- [Rot17] ROTHER, KRISTIAN: *Pro Python Best Practices: Debugging, Testing and Maintenance*, Kapitel Static Typing in Python, Seiten 231–244. Apress, Berkeley, CA, 2017.
- [RV11] ROCHA, HENRIQUE und MARCO TULIO VALENTE: *How Annotations are Used in Java: An Empirical Study*. International Conference on Software Engineering and Knowledge Engineering, 2011.
- [Sch19] SCHILDT, HERBERT: *Java: The complete reference*, Kapitel Enumerations, Autoboxing, and Annotations, Seiten 452–506. New York: McGraw-Hill Education, 2019.
- [Sha15] SHARAN, KISHORI: *Learn JavaFX 8: Building User Experience and Interfaces with Java 8*. Apress, USA, 1 Auflage, 2015.
- [SMT15] SALVANESCHI, GUIDO, ALESSANDRO MARGARA und GIORDANO TAMBURRELLI: *Reactive Programming: A Walkthrough*. Seiten 953–954, 2015.
- [SNP16] SULÍR, MATÚŠ, MILAN NOSÁL' und JAROSLAV PORUBÄN: *Recording concerns in source code using annotations*. Computer Languages, Systems & Structures, 46:44–65, 2016.
- [Ste14] STEYER, RALPH: *Einführung in JavaFX: Moderne GUIs für RIAs und Java-Applikationen*, Kapitel Behind the scene – der Aufbau von FXML, Seiten 123–142. 2014.
- [Tra09] TRATT, LAURENCE: *Dynamically Typed Languages*. Advances in Computers, 77:149–184, 2009.

- [VCG⁺18] VOS, JOHAN, STEPHEN CHIN, WEIQI GAO, JAMES WEAVER und DEAN IVERSON: *Pro JavaFX 9: A Definitive Guide to Building Desktop, Mobile, and Embedded Java Clients*, Kapitel Using Scene Builder to Create a User Interface, Seiten 129–191. Apress, Berkeley, CA, 2018.
- [vRLL14] ROSSUM, GUIDO VAN, JUKKA LEHTOSALO und ŁUKASZ LANGA: *Function Annotations*. PEP 484, 2014.
<https://www.python.org/dev/peps/pep-0484/> (zuletzt abgerufen am 08.09.2021).
- [WL06] WINTER, COLLIN und TONY LOWNDS: *Function Annotations*. PEP 3107, 2006.
<https://www.python.org/dev/peps/pep-3107/> (zuletzt abgerufen am 08.09.2021).
- [YBSM19] YU, ZHONGXING, CHENGGANG BAI, LIONEL SEINTURIER und MARTIN MONPERRUS: *Characterizing the Usage, Evolution and Impact of Java Annotations in Practice*. IEEE Transactions on Software Engineering, 2019.