



Department für Informatik
Abteilung für Medieninformatik und Multimedia-Systeme

Bachelorarbeit

Annotationsbasierte Einstiegserleichterung in
die Entwicklung von JavaFX-Anwendungen

Deniz Groenhoff

17. Juli 2021

1. Gutachterin: Prof. Dr. Susanne Boll
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.

Deniz Groenhoff
Matrikelnummer 5477417
Oldenburg, den 17. Juli 2021

Abstract

Hier kommt in der Regel eine ca. halbseitige Zusammenfassung von Motivation und Ergebnis der Arbeit hin. Eine zeitliche Abfolge, wann was gemacht wurde, spielt hier keine Rolle

Zusammenfassung

Hier kommt in der Regel eine ca. halbseitige Zusammenfassung von Motivation und Ergebnis der Arbeit hin. Eine zeitliche Abfolge, wann was gemacht wurde, spielt hier keine Rolle¹

¹Fussnote 1

Inhaltsverzeichnis

1. Einleitung	3
1.1. Motivation	3
1.2. Zielsetzung	3
1.3. Struktur	3
2. Grundlagen	5
2.1. Entwurfsmuster	5
2.1.1. Beobachter	5
2.1.2. MVC	7
2.2. JavaFX	8
2.2.1. Aufbau und Szenengraph	8
2.2.2. Properties und Bindings	9
2.2.3. Layouting: FXML vs. Quelltext	10
2.3. Java-Annotationen	11
2.3.1. Definition	11
2.3.2. Syntax	13
2.3.3. Auswertung von Annotationen zur Laufzeit	15
2.3.4. Auswertung von Annotationen zur Kompilierzeit	15
3. Stand der Technik	17
3.1. Annotationen im Umfeld von JavaSE/JavaEE/JavaFX	17
3.1.1. JavaSE Umgebung und externe Bibliotheken	17
3.1.2. JavaEE/JakartaEE Umgebung	19
3.1.3. JavaFX Umgebung	20
3.2. Annotationen in anderen Programmiersprachen	20
3.2.1. Verwendung in Python	20
3.2.2. Verwendung in C# und .NET	20
3.3. Fazit	21
4. Konzeption und Entwurf	23
4.1. Identifikation von Problemen und komplexen Strukturen in der JavaFX Entwicklung	23
4.1.1. Internationalisierung und Lokalisierung	24
4.1.2. Abhängigkeitsinjektion für Controller	25
4.1.3. CSS Metadatengeneration	26
4.1.4. JavaFX Einstiegspunkt und Preloader	26
4.1.5. Controller Lebenszyklus	26

4.1.6. Konfigurationsdateien	27
4.1.7. Zustandsserialisierung von grafischen Oberflächen	27
4.2. Anforderungsanalyse	28
4.2.1. Funktionale Anforderungen	28
4.2.2. Nichtfunktionale Anforderungen	34
4.3. Konzept und Modellierung	35
4.3.1. Ziele des Systems	35
4.3.2. Rahmenbedingungen und Designentscheidungen	35
4.3.3. Benötigte Schnittstellen	37
4.3.4. Controller System	43
4.3.5. Annotationen	44
5. Implementierung	51
5.1. Architektur und Struktur der Software	51
5.2. Paketstrukturierung nach Funktionalität	52
5.2.1. Paket: utils	52
5.2.2. Paket: di	53
5.2.3. Paket: dagger1	53
5.2.4. Paket: guice	54
5.2.5. Paket: spring	54
5.2.6. Paket: localization	54
5.2.7. Paket: controller	54
5.2.8. Paket: exception	55
5.2.9. Paket: css	55
5.2.10. Paket: classpath	55
5.2.11. Paket: shared	56
5.2.12. Paket: event	57
5.2.13. Paket: events	57
5.2.14. Paket: application	57
5.3. Essentielle Quelltextausschnitte	58
5.3.1. SimpliFXMLLoader	58
5.3.2. Erweiterbare Abhängigkeitsinjektion	59
6. Evaluation	61
6.1. Entwicklung von Beispielsoftware	61
6.2. Vergleich konventioneller Methoden mit entwickeltem System	61
7. Fazit	63
7.1. Zusammenfassung	63
7.2. Bewertung	63
7.3. Ausblick und mögliche Erweiterungen	63
A. Controllerbasierte JavaFX-Anwendung	65

B. Hinzufügen einer Bibliothek für die Abhängigkeitsinjektion	67
C. Implementierung und Beziehungen des Controller-Systems	69
Abkürzungsverzeichnis	71
Quellcodeverzeichnis	73
Abbildungsverzeichnis	75
Tabellenverzeichnis	77
Literaturverzeichnis	79

Fix layouting: remove
all
newpage occurrences,
fix space above and be-
low figures, fix invalid
spacing due to the ove-
ruse of the [H] figure
float

1. Einleitung

?<einleitung>?

Intro

1.1. Motivation

?<motivation>?

Motivation

1.2. Zielsetzung

?<zielsetzung>?

Zielsetzung

1.3. Struktur

?<struktur>?

Struktur der Arbeit

Bugfixing in tex code

Correction

urls in footnotes

2. Grundlagen

?<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 Layoutingsprache erläutert.

Abschließend wird das generelle Annotationenkonzept in der Informatik mit speziellen 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>? 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 resultierende 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 Beobachter Entwurfsmuster 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.

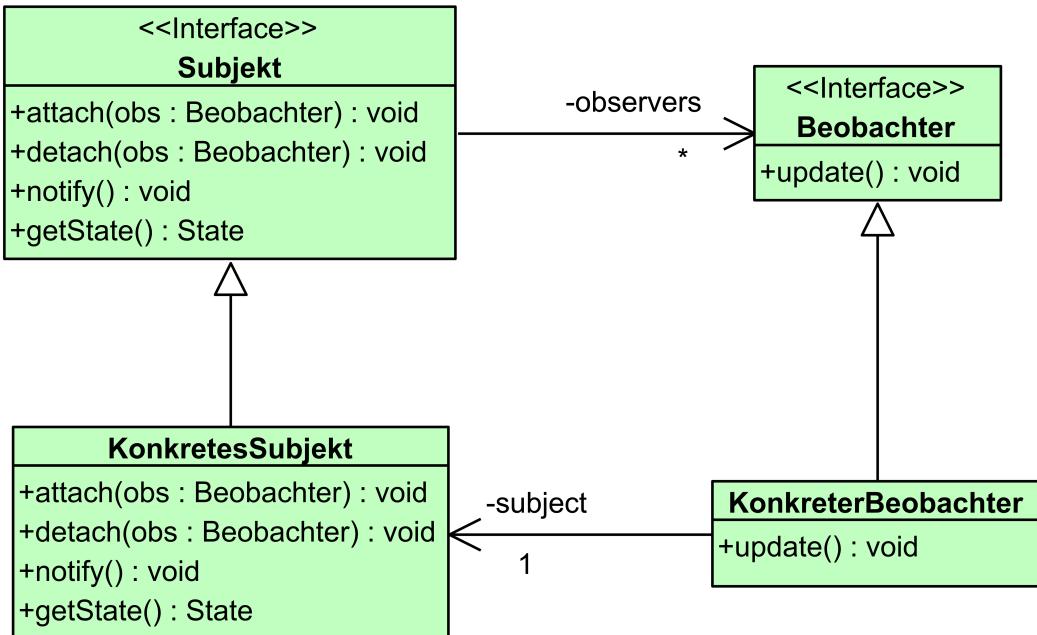


Abbildung 2.1.: UML-Diagramm – Beobachter-Entwurfsmuster

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.

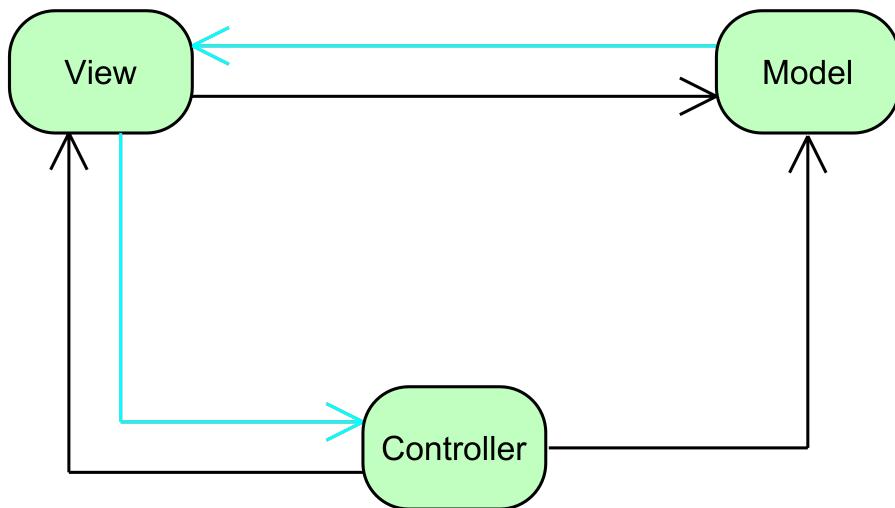


Abbildung 2.2.: Diagramm – MVC-Entwurfsmuster: Die blauen Pfeile stellen indirekte Beziehungen dar, welche meist durch die Nutzung eines Eventsystems auf Basis des Beobachter Musters implementiert werden.

Die vom MVC Muster klassische Struktur³ 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 dar, ermöglicht eine Interaktion mit diesem 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. Der

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.

View und das Model sind vollständig entkoppelt und der Controller ist die Schnittstelle zwischen diesen.

2.2. JavaFX

?<javafx>?

JavaFX ist eine auf Java basierte, quelloffene Bibliothek für das Entwickeln von grafischen Benutzerschnittstellen 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 Beobachter-Muster, weshalb implementierte Anwendung selbst bei vielen Lines of Code (LoC), eine

?<acro:loc>?

grundsätzlich hohe Strukturiertheit auf Quelltextebene aufweisen. Durch die Verwendung von Properties und Bindings (siehe Unterabschnitt 2.2.2) ist es auch mög-

lich das Model-View-ViewModel (MVVP) Muster zu nutzen. Das grafische Layout

?<acro:mvvp>?

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

?<javafx_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 Begriffe der Theaterumgebung [AA19]:

Die **Stage** Klasse repräsentiert ein Anwendungsfenster, welches das Design des Fensterlayouts des aktuell genutzten Betriebssystems nutzt. Eine Stage ist teilweise modifizierbar, so können beispielsweise die Standardschaltflächen in der Titelleiste entfernt oder deaktiviert werden. Werden mehrere Fenster benötigt, so 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 selbsterstellten 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.

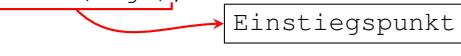
Die **Node** Klasse ist eine darstellbare Komponente im Szenengraphen wie beispielsweise eine Schaltfläche oder ein Containerelement. Node Instanzen im Szenengraph können Kindelemente enthalten und maximal einem Elternelement zugeordnet sein. Der Szenengraph ähnelt somit einer Baumstruktur mit einer 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.

```
ample_javafxapp)?    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, welche 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 Beziehung zwischen zwei oder mehr Variablen zu erstellen. Wird eine Variable

```

st:property_example)?    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.

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 Lambda Ausdruck ein neuer ChangeListener mit der 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 von solchen Dateien sorgt für eine bessere Trennung von Controllern und Logik im Sinne des z.B. 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.

```
e_fxmllayouting)? <?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.

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

Code 2.4: Beispiel – FXML Ladeprozess.

2.3. Java-Annotationen

Annotationen sind in der Sprachwissenschaft eine Möglichkeit einen vorhandenen Text mit Anmerkungen zu versehen für beispielsweise Disambiguierung, also das 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. Dabei sind solche Anmerkungen kein Hauptbestandteil von Texten sondern dienen ausschließlich als Ergänzung.

In der Informatik sind Annotationen ebenfalls nur ein deskriptives Strukturkonzept, welche 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

Move commented foot-note to first occurrence
(in intro)

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 zu 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 Sektion 2.3.3 vorgestellt. Neben den von Java vordefinierten Annotationen wie z.B. @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 [GJSB05]. 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.

```
notation_definition)?  public @interface TestAnnotation {
    // ...
}
```

Code 2.5: Beispiel einer Annotationsdefinition.

In der Analogie des Kapitels 2.3 können Elemente mit strukturgebenden Charakter wie Bestandteile eines Satzes annotiert werden. Analog 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.

```
t:annotated_example)?  @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, so kann es schnell zu Quelltext-Verschmutzung kommen, was im Kontext der Annotationsprogrammierung 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

tationen_syntax)? Annotationen können Attribute besitzen, welche bei 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 [GJSB05]. 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.

```
(lst:decl_normal)?public @interface? Entity{lst:appl_normal}? @Entity(name="test", id=2)
    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.

```
(lst:decl_single)?public @interface? Entity{lst:appl_single}? @Entity("test")
    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.

```
?<lst:decl_marker>?public @interface Entity{<lst:apply_marker>?@Entity
    String name() default "";
    int id() default -1;
}
```

Code 2.11: Deklaration – Marker Annotation.

```
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 z.B. neue Klassen automatisch generieren. Sie sind in der kompilierten Java-Anwendung nicht mehr erhalten.

Klassen-Annotationen sind nach dem Kompilierungsprozess noch in der Anwendung erhalten und können durch externe Tools wie z.B. dem Code-Obfuscator ProGuard ausgelesen werden.

Laufzeit-Annotationen sind nach der Kompilierung und beim Start der Anwendung erhalten 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.

```
_annotation_example)? @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.

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, so 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 [FFI⁺04]. Jedes Java-Element der Reflection API (Feld, Methode, Klasse, ...), welches annotierbar ist, wird durch die Vererbung der AnnotatedElement-Klasse als solches klassifiziert [Sch19]. Damit nun 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.

```
cessing_example)?    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 AutoService⁵ 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 müssen. 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 nur für den

⁵Google AutoService: <https://github.com/google/auto/tree/master/service>

Entwickler zeitaufwändig, da der Anwender ebenfalls die verwendeten Annotationsprozessoren registrieren müssen. Letzteres kann durch das Verwenden von Build-Management-Tools wie Apache Maven⁶ verhindert werden und die Quelltextgeneration kann durch externe Bibliotheken wie JavaPoet⁷ deutlich erleichtert werden. Damit bereits vorhandene Klassen modifiziert oder erweitert werden können, muss eine Form der Bytecode Manipulation genutzt werden. Dazu kann beispielsweise die ASM⁸ Bibliothek genutzt werden, welche es dem Entwickler ermöglicht existierende Klassen, Methoden oder Felder vollständig zu verändern [Kul07].

```
n_processor_example)? @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.

⁶Apache Maven: <https://maven.apache.org>

(ft:maven) ⁷JavaPoet: <https://github.com/square/javapoet>

⁸ASM: <https://asm.ow2.io>

(ft:asm)

3. Stand der Technik

In diesem Kapitel werden aktuelle Konzepte und Implementierungen der Annotationsprogrammierung zur Vereinfachung des Entwicklungsprozesses einer Anwendung dargelegt. Obwohl der primäre Fokus dabei auf der JavaFX- und der generellen 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 Entwicklung 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. Will 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 `?(acro:awt)? 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.

<pre>?<lst:decl_interface>?interface Test?<lst:compiler_error>?</pre>	<pre>class TestClass implements Test { @Override public void t(int i) {} }</pre>
---	--

Code 3.1: Beispiel – Interfacedeklaration.

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 unbenutzt 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>

```
:lombok_example)? @Getter  
    @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

eld_von_java_ee)? 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) ?<acro: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 vice versa). 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.

```
<lst:example_xml>? ?<lst:example_java>? @XMLRootElement  
    <?xml version="1.0"?> public class Cat {  
  
        <Cat id="1"> @XmlAttribute  
            <color>Brown</color> public int id;  
            <age>4</age> public String color;  
        </Cat> public int age;  
    }  
}
```

Code 3.4: Repräsentation als XML Datei.

Code 3.5: Repräsentation als Java Objekt.

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

3.1.3. JavaFX Umgebung

`_umfeld_von_java_fx)?` 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

`in_anderen_sprachen)?` 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

`erwendung_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 PyCharm 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.

maybe add code example

3.2.2. Verwendung in C# und .NET

`_in_c_sharp_dot_net)?` In C# wird das Hinzufügen von Meta-Informationen zu bestehenden Programmlementen durch Attribute realisiert [AJ19]. Mithilfe dieser Attribute können dann beispielsweise Klassen als serialisierbar deklariert werden oder Methoden und Funktionen für nicht verwaltete Dynamic Link Librarys (DLLs) erreichbar gemacht werden.

`?(acro: dll)?`

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

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

den. 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 [CCC].

3.3. Fazit

?<ms_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 JUnit⁷ aufbauende Testumgebung für grafische Benutzeroberflächen 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.

⁷JUnit 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 verbessungswürdigen 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 dem Entwickeln von Erweiterungen für existierenden 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 gelöst.

4.1.1. Internationalisierung und Lokalisierung

In der Informatik, speziell in der Softwareentwicklung, ist die Internationalisierung ein wichtiger Bestandteil eines Softwareproduktes, bei welchen 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 -->
<code>login.user = Benutzername</code>	<code><Label text="%login.username"/></code>

?{lst:fxmlkey?}

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 einem Modifizierung des Ladeprozesses von FXML Dateien durch eine eigene Version des `FXMLLoader`s oder durch eine vom `FXMLLoader` unabhängige Implementierung durch 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. Parameterisierte 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 abhängigen 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` 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.

validate

```
rollerinjection?  public class TestController {  
  
    @Inject  
    private IUserService userService;  
  
    @Inject  
    private IDatabaseService dbService;  
  
}
```

Code 4.2: Beispiel – Controller mit injizierten Diensten.

Die Implementierungen sollen durch den Entwickler in der jeweiligen Konfigurationsdatei/Konfigurationsklasse den Schnittstellen zugeordnet werden können.

¹Spring: <https://spring.io>

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

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

?{ft:guice}?

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 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 solcher 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 durch eventbasierte Statusaktualisierungen, dem Nutzer 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 Implementierungen und Funktionen dieser sich 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

roller_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 Controllerinstanzierung 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 Ressourcenelecks zu verhindern. Ein vollständiger Lebenszyklus wie bei Activities in Android⁴, ist praktisch nicht vorhanden.

```
erinstantiation)?public class TestController {
    @FXML
    private Label testLbl;

    public TestController() {
        // Phase #1 - Controllerinstanziierung
        // 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 – Instanzierungsablauf.

4.1.6. Konfigurationsdateien

Eine persistente Speicherung von Anwendungskonfigurationen wie beispielsweise Logindaten oder Informationen für Serververbindungen ist wichtig für eine optimale Benutzerfreundlichkeit. Das Speichern von konfigurierbaren Einstellungen kann mit der Java Preferences API oder mit externen Bibliotheken wie GSON, 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. Modifizierung zur Laufzeit ist ein repetitiver Vorgang, welcher durch das Nutzen von Annotationen teilweise automatisiert werden kann.

4.1.7. Zustandsserialisierung von grafischen Oberflächen

Das Speichern des aktuellen Zustands einer grafischen Oberfläche mit den jeweiligen Elementen ist aktuell mit dem Funktionsumfang von JavaFX nicht ohne Weiteres umsetzbar. Der Zustand einer Anwendung setzt sich aus den aktuellen Konfiguration von interaktiven Elementen im Szenengraphen zusammen. Beispielsweise ist die Position eines Schiebereglers oder der aktuell ausgewählte Tab einer TabPane nicht insoweit serialisierbar, dass dieser Zustand beim Applikationsende gespeichert und bei einem erneuten Applikationsstart wiederhergestellt werden kann.

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

4.2. Anforderungsanalyse

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.

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

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

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

(freq1) 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 zugrunde liegenden 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

(freq2) Die Verwaltung des Applikationslebenszyklus soll mithilfe von Annotationen im Einstiegspunkt geschehen. Der Applikationslebenszyklus bezeichnet hier die vererb-baren Application#init, Application#start und Application#stop Methoden. Die Erstellung der Applikationsklasse soll wie in Anforderung 1 beschrie-ben, durch die zu entwickelnde Bibliothek übernommen werden, weshalb die Auf-rufe 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 Einstiegs-punkt 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

(freq3) Die bereits existierenden Internationalisierungsmöglichkeiten sollen durch eine dy-namische Übersetzung erweitert werden. Das Ändern der aktuellen Sprache soll durch beispielsweise einen Schaltflächenklick ausgelöst werden und zur Laufzeit die grafische Benutzeroberfläche 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 Klas-sse geladen und instanziiert werden. Dabei soll das dynamische Übersetzen der definierten Elemente per Element deaktivierbar sein. Die Funktionalität von exter-nen FXML Parsern wie den Scene-Builder soll durch das Hinzufügen von eigener FXML Syntax dabei nicht kompromittiert werden. Übersetzbare Elemente, welche auf parameterabhängigen Übersetzungsschlüsseln basieren, sollen an JavaFX Prop-erties 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 Funktio-nalität zur Verfügung stellen.

[FA–04] Abhängigkeitsinjektion durch etablierte externe Bibliotheken

(freq4) Werden externe Bibliotheken zur Abhängigkeitsinjektion genutzt, so sollen eventu-elle 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 Unterstüt-zung der Bibliotheken Dagger, Spring und Guice soll dem Entwickler ermöglicht werden.

[FA-05] Automatische CSS Metadaten Generation

(freq⁵) Das Hinzufügen von CSS Properties zu benutzerdefinierten JavaFX Komponenten durch den Entwickler soll durch das Nutzen von Annotationen erleichtert werden. Durch die automatische Generation 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.

Manual correct mistakes after this point

[FA-06] Annotationsbasierte Controllerdefinition

(freq⁶) 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

(freq⁷) 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

`<freq8>` Wenn SimpliFX die Einrichtung des Einstiegspunktes der Applikation, des Preloaders oder der Controller beendet hat, so 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

`?<freq9>` Wenn mehrere Methoden mit denselben Lebenszyklusannotationen annotiert wurden, so soll die Reihenfolge der Aufrufe dieser, anhand eines optionalen Parameters in der Annotation bestimmt werden. Weisen mehrere Annotationen dieselbe Priorisierung auf, so soll nach der Deklarationsreihenfolge in der Klasse sortiert werden.

[FA-10] Controllerverschachtelung und Supercontroller

`<freq10>` Ist das Wurzelement eines Controllers in einem durch einen weiteren Controller aufgespannten Teilbaum des Szenengraphen enthalten, so handelt es sich bei letzterem 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

`<freq11>` 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 Controllergruppe zuordenbar sein. Wechselt der aktive Controller einer Controllergruppe, 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

`?<freq12>` 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

(freq¹³) 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

?(freq¹⁴)? 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

?(freq¹⁵)? Beim Start der Anwendung, sollen optionale Klassen zur globalen Fehlerbehandlung spezifiziert werden können. Wird eine solche Klasse nicht durch den Entwickler angegeben, so soll auf eine vom System bereitgestellte Klasse zur Fehlerbehandlung 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.

[FA-16] Geteilte Ressourcen

(freq¹⁶) 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] Anpassbare Events durch Annotationen

(freq¹⁷) 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, so 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, welche den Annotationen als Parameter übergeben wird, aufweisen.

[FA-18] Konfigurationsdateien

{freq18} 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-19) Optional – Serialisierung von JavaFX Komponenten

?(freq19)? 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

?(freq20)? Controllerinterne Methoden sollen von Entwickler als sog. 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

{freq21} 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

?(freq22)? 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

yse_nichtfunktional)? 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. Darunter gehören beispielsweise die effiziente Ressourcennutzung, die Korrektheit des Systems sowie ein gewisser Grad an Zuverlässigkeit.

[NFA-01] Benutzerfreundlichkeit

?<nreq1>?

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-02] Performance & Effizienz

?<nreq2>?

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-03] Wartbarkeit

?<nreq3>?

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

[NFA-04] Erweiterbarkeit

?<nreq4>?

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-05] Softwarequalität

?<nreq5>?

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-06] Lizenzierung

?<nreq6>?

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

nd_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 den Komponenten und die eventuelle Nutzung von Entwurfsmustern ermittelt werden. Dabei wird besonders auf eine hochwertige Softwarequalität und Usability 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 die Lokalisierung und die Controllerverwaltung.
- Hinzufügen von Funktionen, welche aktuell nicht im Funktionsumfang von JavaFX enthalten sind wie 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

nentscheidungen)? 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 ein ordnungsgemäße Ausnahmebehandlung aufweisen, da das Nutzen der Reflection API sehr fehleranfällig ist und sehr effizient implementiert werden, da etwaige Operationen langsamer sind als Nutzen von herkömmlichen Java-Sprachfeatures. Diese Schnittstelle wird in einem von SimpliFX unabhängigen Apache Maven⁶ Modul ausgelagert. Auch die Testapplikationen, welche nach der Implementierung des eigentlichen Systems entwickelt werden und für die Evaluation von diesem dienen, werden 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 IDEA⁵ oder Build-Management-Tools wie Apache Maven⁶ und Gradle⁶, 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,

⁵IntelliJ IDEA: <https://www.jetbrains.com/de-de/idea/>

⁶Gradle: <https://gradle.org>

im Vergleich zum manuellen Erstellen der selben 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 mit lizenztechnisch kompatibel sind. Alle verwendeten externen Bibliotheken sind mit einer Kurzbeschreibung der Funktion und der jeweiligen Lizenz in Tabelle 4.1 dargestellt.

Bibliothek	Funktion	Version	Lizenz
JavaFX	Ermöglicht die Erstellung von Java Applikationen mit grafischen Oberflächen	16	GPLv2 ⁷
ASM	Erlaubt die Modifikation von Java-Bytecode	9.1	BSD ⁸
...

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 Teifunktionen 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 und das Auswerten mit Annotationen, an einem zentralen Ort ermöglichen. Reflektierbare Elemente sollen durch eigene Schnittstellen repräsentiert werden, welche, für diese relevante, Operationen zur Verfügung stellen. Diese spezialisierten Klassen

⁷GPLv2 mit Classpath Exception: <https://openjdk.java.net/legal/gplv2+ce.html>

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

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

Tabelle 4.2.: Alle benötigten ReflectionScopes.

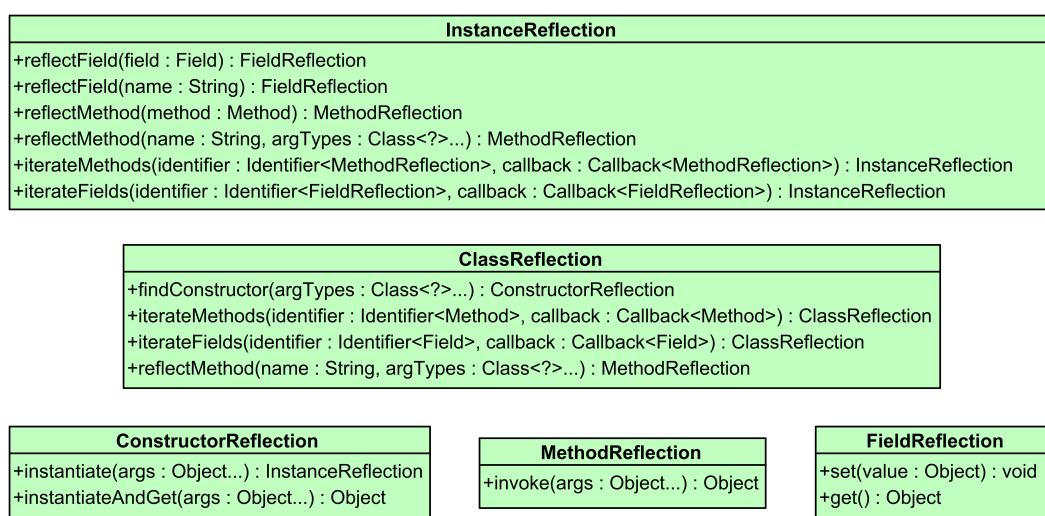


Abbildung 4.1.: Diagramm – ReflectionScopes

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 es 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

spath_interface)? 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 Bibliothek⁸, mithilfe des Besucher Entwurfsmusters 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

sonarlint

ignoriert werden. Ergebnisse des Scans müssen in einem Cache zwischengespeichert werden, damit nicht jeder Filervorgang 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 bereitstellen und JavaFX Bindings aus den jeweiligen Übersetzungsschlüsseln generieren. Dabei muss insbesondere die Unterstützung von parametrisierten Schlüsseln gewährleistet werden, das 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.

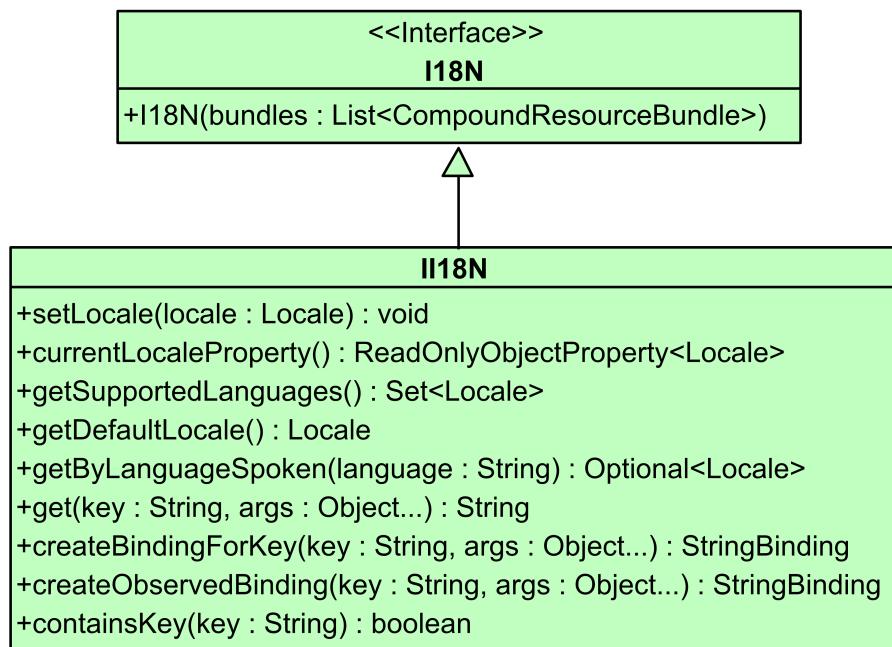


Abbildung 4.3.: Diagramm – I18N Schnittstelle.

Eine **I18N** Instanz soll dabei, bei Bedarf, mehrere ResourceBundle verwahren, 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 ResourceBundles 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.

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 Kapitel beschrieben. Die Erweiterung der FXMLLoader Klasse wird im Folgenden als SimpliFXMLLoader bezeichnet.

chapter to implementation

Schnittstelle: SimpliFX Einstiegspunkt

SimpliFX übernimmt die Konstruktion und die anschließende Verwaltung der JavaFX-Anwendungsklasse. Essentielle Methoden, welche durch die Java 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. Der Entwickler muss zum Starten von SimpliFX eine der vordefinierten launch-Methoden verwenden (siehe Abbildung 4.4).

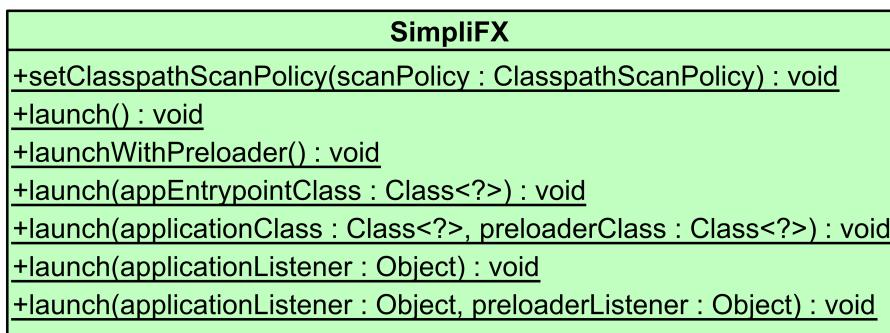


Abbildung 4.4.: Diagramm – Konfigurations- und Startmethoden von SimpliFX.

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 überprüft (Korrekte Annotationen und Parameter). Daraufhin beginnt die Initialisierung der JavaFX Plattform, der Applikationsklasse, des Controllersystems und des Hauptcontrollers, die Konfiguration von der jeweiligen Bibliothek für Abhängigkeitsinjektion (falls angegeben) und das Laden von Konfigurations- und Sprachdateien.

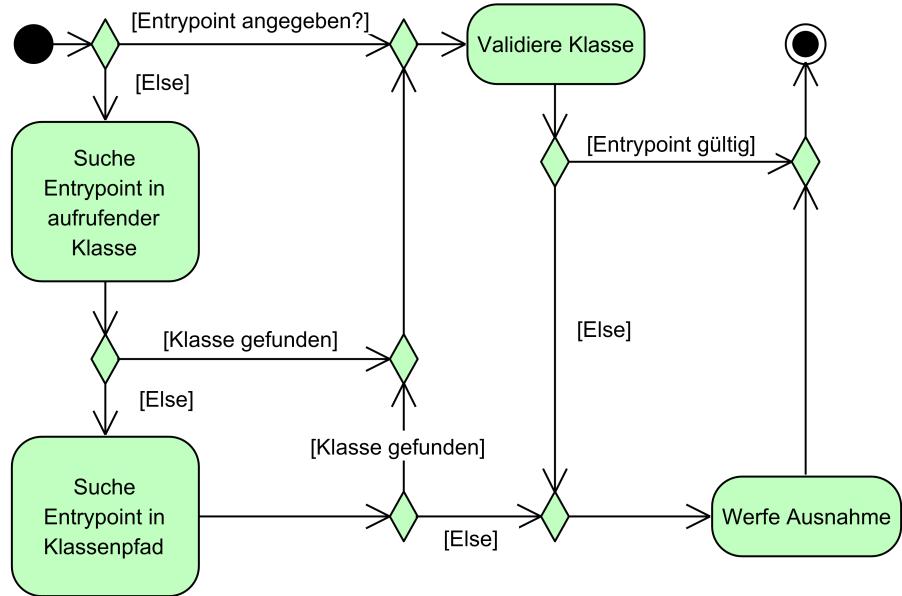


Abbildung 4.5.: Diagramm – Finden und Validierung des Einstiegspunktes.

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, 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 Instanzierung einer globalen `I18N` Instanz benötigt, welche optional in alle Einstiegspunkte und zukünftig erstellte Controller injiziert und für alle Ladeprozesse von FXML Dateien durch den `FXMLLoader`

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 zu 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 selbe 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 FXMLLoaders 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. 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 verschiedenen 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).

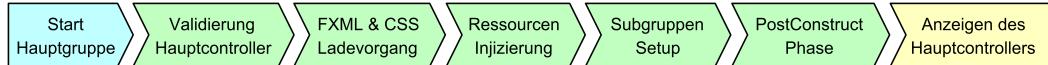


Abbildung 4.6.: Diagramm – Erstellung der Hauptgruppe und des Hauptcontrollers.

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

?<annotations>? 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 keinen default Wert besitzen und aufgrund dessen vom Entwickler manuell 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 aufweisen. Dieser kann ein einziger Parameter übergeben werden, welcher notwendig ist und die Klasse des Hauptcontrollers darstellen soll.

Annotation: @PreloaderEntryPoint

Die @PreloaderEntryPoint Annotation besitzt ähnliche Funktionalitäten wie die @ApplicationEntryPoint Annotation, nur das 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: @ConfigurationSource

Mit der Java Properties-API kompatible Konfigurationsdateien können durch das Nutzen der @ConfigurationSource Annotation automatisch geladen und dem annotierten Properties-Feld zugewiesen werden. Die Annotation hat einen optionalen Parameter, welcher den Namen der Konfigurationsdatei bzw. den Pfad zu dieser enthält. Ist kein Pfad angegeben, so wird, je nach Klassenpfadscan-Einstellung, der Klassenpfad nach einer Datei mit einem Namen, welcher dem des annotierten Fels des entspricht, durchsucht. Jedes Element aus einer Konfigurationsdatei, welche auf diesem Wege geladen worden ist, kann durch die @ConfigValue Annotation verwendet werden.

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.

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 Prioritätsparameter angegeben werden, welcher die Aufrufreihenfolge, wenn mehrere solcher Methoden existieren, bestimmt. Die Priorität wird dabei als Ganzzahl repräsentiert und Methoden werden in aufsteigender Reihenfolge dieser aufgerufen.

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. Des Weiteren ist die Angabe einer Controllergruppe möglich, um verschiedene Controller in Gruppen zu unterteilen.

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. Jede durch JavaFX ermöglichte Methode zum Abfangen von Events (`onAction`, `onClick`, ...), welche normalerweise mit `@FXML` annotiert wird, kann ebenfalls mit `@EventHandler` annotiert werden.

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.

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 Komponenten handeln, welche eine Form von übersetzbarem Text darstellen können. Dazu gehören beispielsweise die `Button` und die `Label` Klasse. 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. Bei mehreren Übersetzungsparametern kann der Annotation ein Parameterindex übergeben werden.

Maybe add functionality

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: @DIAAnnotation

Die `@DIAAnnotation` 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.

Annotation: @SpringInjection

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: @GuiceInjection

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.

Annotation: @Dagger1Injection

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, ...), 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.

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. Analog dazu werden @OnHide Methoden aufgerufen, wenn der Controller gewechselt wird und somit nicht mehr angezeigt wird.

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 als Parameter 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.

Annotation: @Shared

Objekte und Referenzen auf Objekte, welche geteilte Ressourcen darstellen sollen 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 aktualisierbare JavaFX Properties 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 werden.

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.

Übersicht und Zusammenfassung

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

the javafx menu base
class to eventemitter
restrictions

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 Einstiegspunkt 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"> • Feldname • Parameterindex 	<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 • JavaFX-Property Feld 	<ul style="list-style-type: none"> • Nur Collection-Felder
Controller	6,10	<ul style="list-style-type: none"> • FXML Pfad • CSS Pfad • Controllergruppe 	<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"> • Nur Property-Felder
EventEmitter	17	<ul style="list-style-type: none"> • Beziehungsidentifikator 	<ul style="list-style-type: none"> • ButtonBase-Felder
EventReceiver	17	<ul style="list-style-type: none"> • Beziehungsidentifikator 	<ul style="list-style-type: none"> • Methoden
ConfigSource	18	<ul style="list-style-type: none"> • Pfad oder Name der Datei 	<ul style="list-style-type: none"> • Nur Properties-Felder
ConfigValue	18	<ul style="list-style-type: none"> • Konfigurationsschlüssel • Standardwert 	<ul style="list-style-type: none"> • Nur String und primitive Felder
FXThread	21	–	<ul style="list-style-type: none"> • Nur Methoden

5. Implementierung

implementierung)? In diesem Kapitel werden wichtige Aspekte bei der Implementierung des Systems aufgezeigt. Auf essentielle Quelltextausschnitte, welche für eine Gewährleistung, die 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

?<architektur>? Für die Implementierung und um eine, nach Abschnitt 4.2.2 und Abschnitt 4.2.2, hohe Softwarequalität sowie Erweiterbarkeit, 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 Softwarelizenz 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 Maven⁶ Build-Management-Tool verwendet. Das Projekt wird nach der finalisierten Implementierung als Maven-Artefakt öffentlich zugänglich 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 wurde. 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.

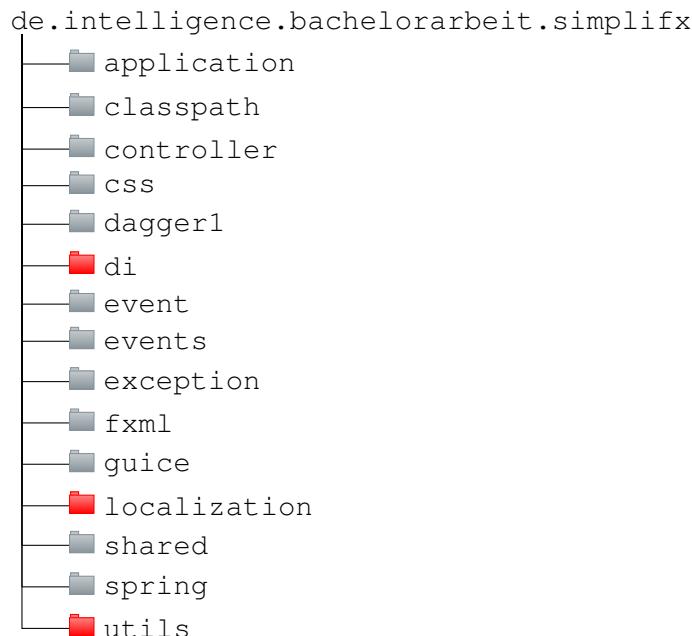


Abbildung 5.1.: Paketstruktur – SimpliFX

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 bool'schen 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.

```
?(lst:pair)? 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

?(package_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 zwei 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 Implementationen für beide Schnittstellen, sowie eine Annotation für die jeweilige Bibliothek bereitgestellt werden.

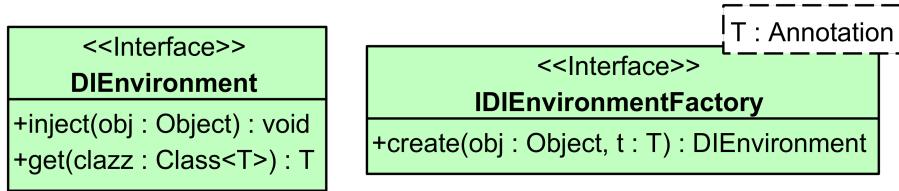


Abbildung 5.2.: Diagramm – DI Schnittstellen.

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 Dagger 1¹ 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 einem 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>+constructController(clazz : Class<?>, readyConsumer : Consumer<Pane>) : IController +start() : Pane +registerSubGroup(originController : Class<?>, startController : Class<?>, groupId : String, readyConsumer : Consumer<Pane>) : void +switchController(newController : Class<?>) : void +switchController(newController : Class<?>, factory : IWrapperAnimationFactory) : void +destroy(clazz : Class<?>) : void +destroy() : void +getContextFor(groupId : String) : ControllerGroupContext +visibilityProperty() : ObjectProperty<VisibilityState> +getActiveController() : Class<?></pre>

Abbildung 5.3.: Diagramm – Einstiegspunkt des Controller-Systems.

Controllergruppen werden anhand eines *String*s 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 *SwingFX* in diesem Fall ausgeschlossen ist.

5.2.8. Paket: exception

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

5.2.9. Paket: css

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

5.2.10. 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 z.B. WAR-Archiven (siehe Abbildung 5.4).

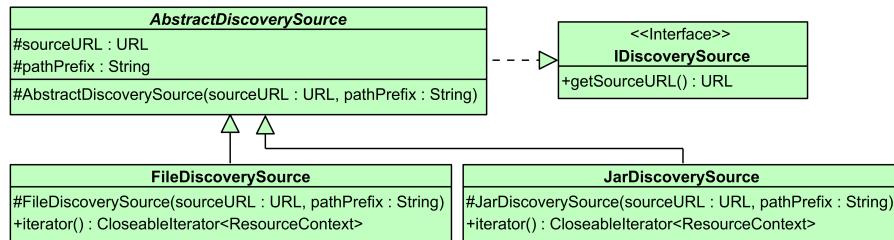


Abbildung 5.4.: Diagramm – Mögliche Quellen für Klassenpfade.

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.

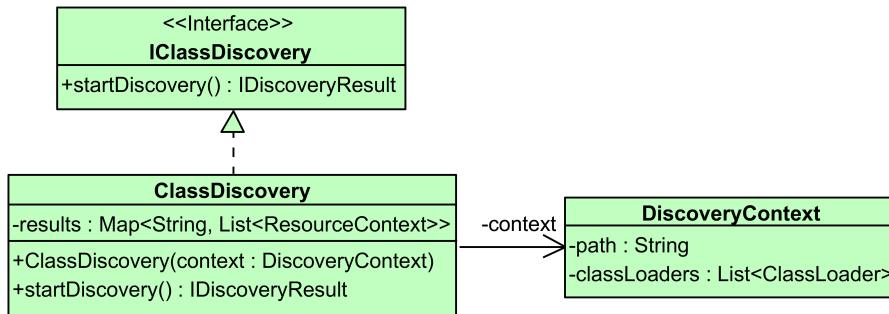


Abbildung 5.5.: Diagramm – Klassenpfad Schnittstelle und Implementierung.

Eine beispielhafte Nutzung des Klassenpfadscans ist im folgenden Quelltextauschnitt dargestellt:

```

lasspath_scan_usage)? // 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.

5.2.11. 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.

5.2.12. 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.

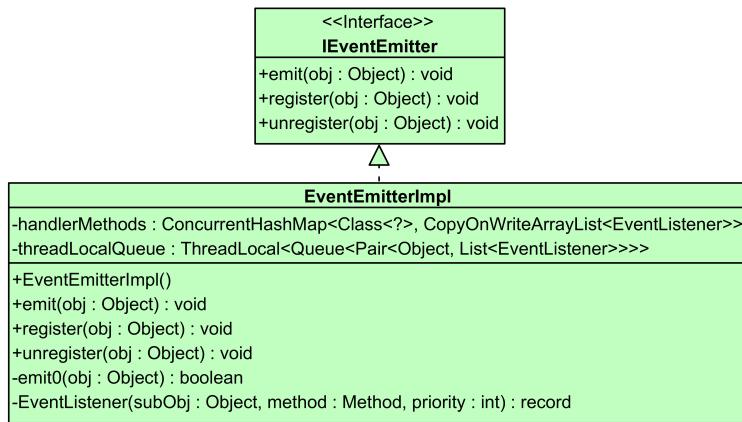


Abbildung 5.6.: Diagramm – IEventEmitter Schnittstelle und Implementierung.

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.

5.2.13. 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.

5.2.14. Paket: application

Die von SimpliFX verwalteten JavaFX Applikation- und Preloader-Klassen sowie die dafür benötigten Annotation sind im application Paket enthalten und de-

finieren 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.

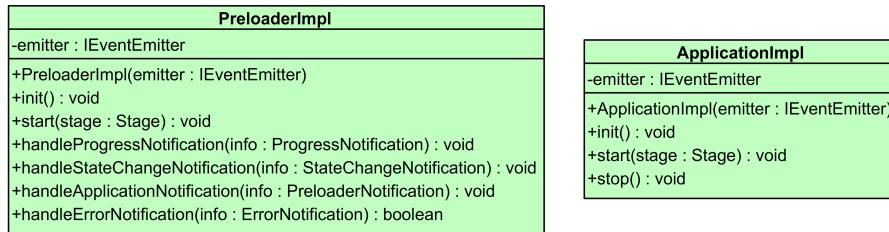


Abbildung 5.7.: Diagramm – Applikation und Preloader.

?(fig:app_package)?

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 Methode (Zeilen 430–437) `Element#resolvePrefixedValue` 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 Generation 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.

atable_property)?

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 erweitert 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.

```

source_handling)? 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 (??) 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 wer-

den müssen. Dazu werden alle Annotationen des Einstiegspunktes auf die Meta-Annotation `@DIAnnotation` überprüft. Ein exemplarischer Quelltextausschnitt, welcher dieses Konzept implementiert, ist in Code 5.4 zu erkennen. Eine ähnliche Implementierung ist in SimpliFX verwendet worden.

```
?<lst:reflective_di?// Besitzt die Annotation die Meta-Annotation @DIAnnotation?
if (annotation.annotationType().isAnnotationPresent(DIAnnotation.class)) {
    // Finde IDIEnvironmentFactory Implementierung durch Annotationsparameter
    var factory = annotation.annotationType().getAnnotation(DIAnnotation.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
    SimpliFX.appDIEnv = methodRef.invokeUnsafe(applicationListener, annotation);
    break;
}
```

Code 5.4: Implementierung – Abhängigkeitsinjektion.

6. Evaluation

?<evaluation>?

Intro

6.1. Entwicklung von Beispielsoftware

eispielsoftware)?

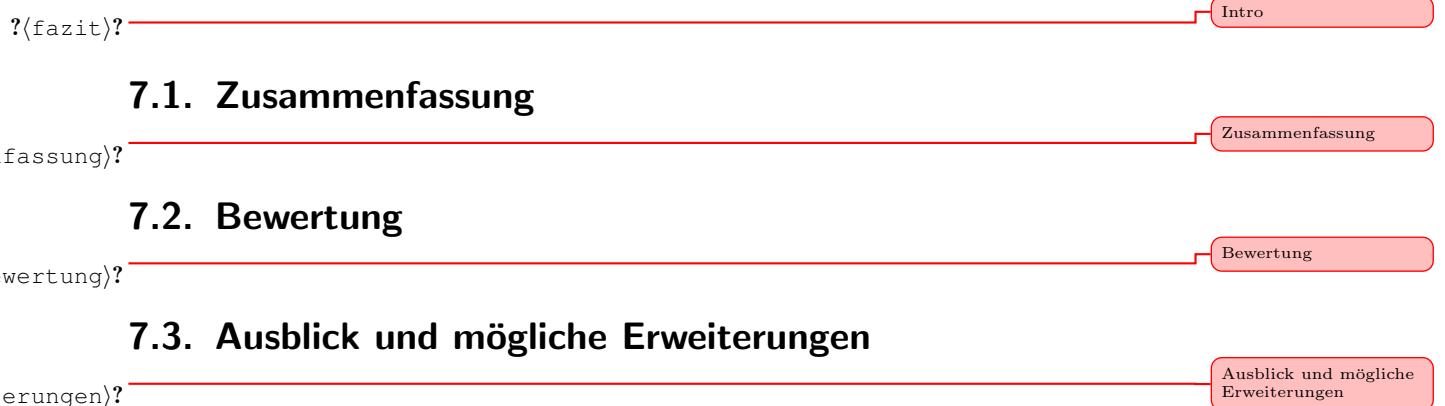
Entwicklung von Bei-
spielsoftware

6.2. Vergleich konventioneller Methoden mit entwickeltem System

h_system_javafx)?

Vergleich konven-
tionaler Methoden mit
entwickeltem System

7. Fazit



A. Controllerbasierte JavaFX-Anwendung

```
afx_application)?

---

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.

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

Code A.3: CSS-Design.

```
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

```
new_di_library)?

---

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 Umgebung.

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

Code B.3: Beispiel – Erstellen einer neuen Annotation.

C. Implementierung und Beziehungen des Controller-Systems

x:controller_system)?

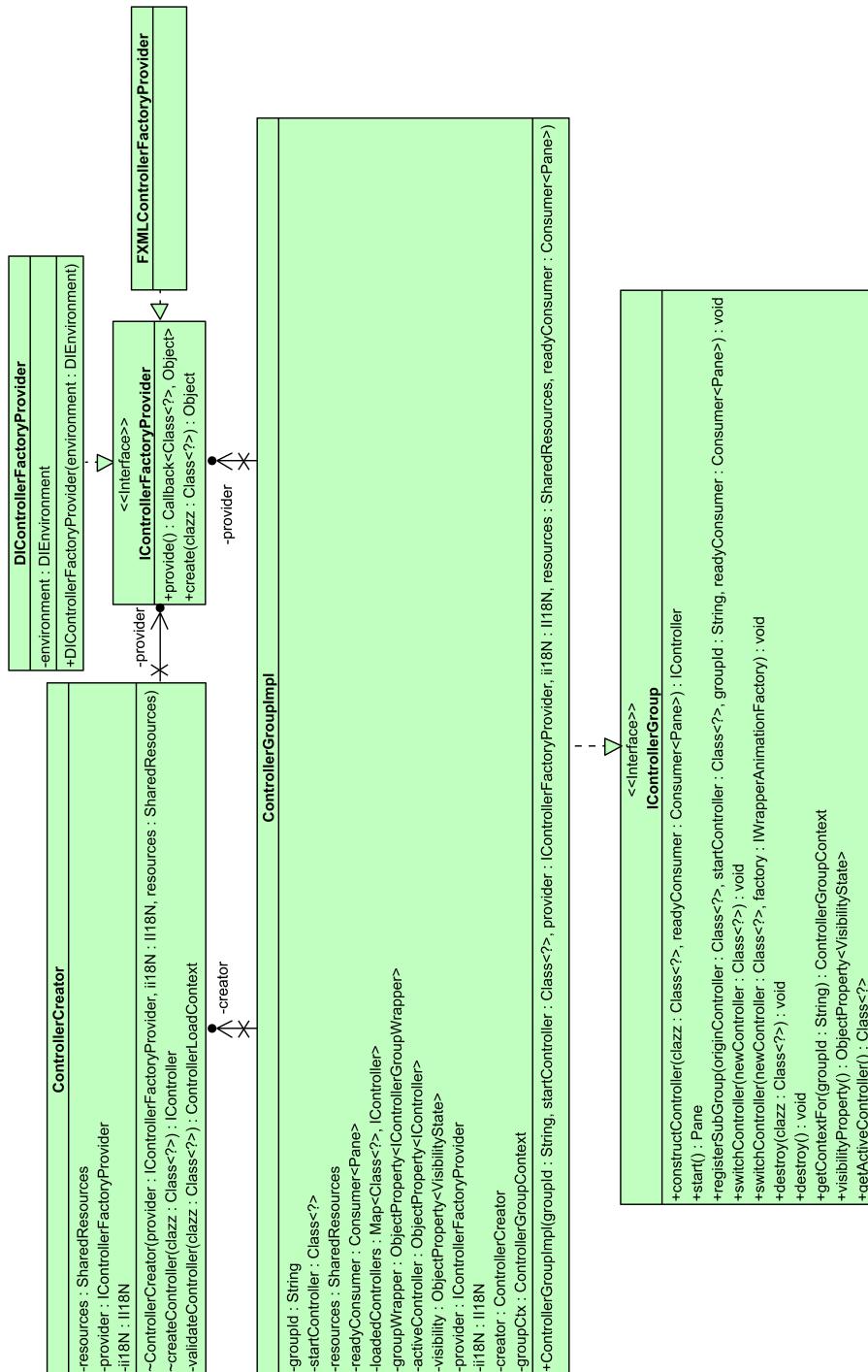


Diagramm – Controller-System: Einstiegspunkt und Beziehungen

Abkürzungsverzeichnis

ungsverzeichnis)? **AWT** Abstract Window Toolkit

CSS Cascading Style Sheets

DLL Dynamic Link Library

IoC Inversion of Control

JAXB Jakarta XML Binding

JDK Java Development Kit

JSON JavaScript Object Notation

LoC Lines of Code

MVC Model-View-Controller

MVVP Model-View-ViewModel

SoC Separation of Concerns

UML Unified Modeling Language

XML Extensible Markup Language

Quellcodeverzeichnis

2.1.	Beispiel – Minimale JavaFX-Anwendung.	9
2.2.	Beispiel – ChangeListener & EventHandler.	10
2.3.	Beispiel – FXML Layouting.	11
2.4.	Beispiel – FXML Ladeprozess.	11
2.5.	Beispiel einer Annotationsdefinition.	12
2.6.	Beispiel einer annotierten Klasse.	12
2.7.	Deklaration – Normal Annotation.	13
2.8.	Anwendung – Normal Annotation	13
2.9.	Deklaration – Single-Element Annotation.	13
2.10.	Anwendung – Single-Element Annotation	13
2.11.	Deklaration – Marker Annotation.	14
2.12.	Anwendung – Marker Annotation	14
2.13.	Beispiel einer Laufzeit Annotation.	14
2.14.	Auslesen einer Laufzeit-Annotation.	15
2.15.	Beispiel – Annotationsprozessor.	16
3.1.	Beispiel – Interfacedeklaration.	18
3.2.	Beispiel – Kompilierfehler.	18
3.3.	Beispiel – Lombok POJO.	19
3.4.	Repräsentation als XML Datei.	19
3.5.	Repräsentation als Java Objekt.	19
4.1.	Nutzung des Schlüssels in einer FXML Datei.	24
4.2.	Beispiel – Controller mit injizierten Diensten.	25
4.3.	Beispiel – Instanziierungsablauf.	27
4.4.	Beispiel – Verwendung der Reflection Schnittstelle.	39
5.1.	Beispiel – Nutzung der Pair Klasse.	53
5.2.	Beispiel – Initiierung eines Klassenpfadscans.	56
5.3.	Implementierung – Ressourcenbehandlung im SimpliFXMLLoader.	59
5.4.	Implementierung – Abhängigkeitsinjektion.	60

Abbildungsverzeichnis

2.1. UML-Diagramm – Beobachter-Entwurfsmuster	6
2.2. Diagramm – MVC-Entwurfsmuster	7
4.1. Diagramm – ReflectionScopes	38
4.2. Diagramm – Klassenpfad-Scanprozess für Klassendateien.	40
4.3. Diagramm – I18N Schnittstelle.	40
4.4. Diagramm – Konfigurations- und Startmethoden von SimpliFX. . .	41
4.5. Diagramm – Finden und Validierung des Einstiegspunktes.	42
4.6. Diagramm – Erstellung der Hauptgruppe und des Hauptcontrollers.	44
5.1. Paketstruktur – SimpliFX	52
5.2. Diagramm – DI Schnittstellen.	53
5.3. Diagramm – Einstiegspunkt des Controller-Systems.	55
5.4. Diagramm – Mögliche Quellen für Klassenpfade.	56
5.5. Diagramm – Klassenpfad Schnittstelle und Implementierung. . . .	56
5.6. Diagramm – IEventEmitter Schnittstelle und Implementierung. . .	57
5.7. Diagramm – Applikation und Preloader.	58
5.8. Diagramm – Klasse zur Datenkapselung von Übersetzungsinforma- tionen.	59

Tabellenverzeichnis

4.1. Verwendete Externe Bibliotheken.	37
4.2. Alle benötigten ReflectionScopes.	38

Literaturverzeichnis

- [Anderson2019] [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.
- [Albahari2019] [AJ19] ALBAHARI, J. und E. JOHANNSEN: *C# 8.0 in a Nutshell: The Definitive Reference*. In a Nutshell. O'Reilly Media, 2019.
- [Burbeck1992] [Bur92] BURBECK, STEVE: *Applications programming in smalltalk-80: how to use model-view-controller (mvc)*. 01 1992.
- [Cazzola2005] [CCC] CAZZOLA, WALTER, ANTONIO CISTERNINO und DIEGO COLOMBO: *[a]C*.
- [Deitsch2001] [DCL01] DEITSCH, ANDREW, DAVID CZARNECKI und MIKE LOUKIDES: *Java Internationalization*. O'Reilly amp; Associates, Inc., USA, 2001.
- [Deacon1995] [Dea95] DEACON, JOHN: *Model-View-Controller (MVC) Architecture*. Online, August 1995.
- [Danelutto2007] [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.
- [Forman2004] [FFI⁺04] FORMAN, IRA R., NATE FORMAN, DR. JOHN VLASSIDES IBM, IRA R. FORMAN und NATE FORMAN: *Java Reflection in Action*, 2004.
- [Gao2019] [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.
- [Gamma1993] [GHJV94] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLASSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software*. Seiten 1–4, 293–303, 1994.
- [Gosling2005] [GJSB05] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHA: *The Java Language Specification, Third Edition*, Seiten 268–281. 2005.
- [Grant2014] [Gra14] GRANT, ANDREW: *Introduction to MVC*, Seiten 47–56. Apress, Berkeley, CA, 2014.

- [Homme12013] [Hom13] HOMMEL, SCOTT: *Using JavaFX Properties and Binding.* <https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm>, April 2013. letzter Abruf: 26. Mai 2021.
- [Hughes2013] [HvDM⁺13] HUGHES, JOHN F., ANDRIES VAN DAM, MORGAN MC GUIRE, DAVID F. SKLAR, JAMES D. FOLEY, STEVEN FEINER und KURT AKELAY: *Scene Graphs*, Seiten 351–353. Addison-Wesley, Upper Saddle River, NJ, 3 Auflage, 2013.
- ?Jha2020? [JN20] JHA, AJAY und SARAH NADI: *Annotation practices in Android apps.* 2020.
- [Juneau2013] [Jun13] JUNEAU, JOSH: *JavaFX in the Enterprise*, Seiten 615–646. Apress, Berkeley, CA, 2013.
- [Khan2021] [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, Seiten 1–1, 2021.
- [Kruk2018] [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), Barcelona, Spain, 8-13 October 2017*, Nummer 16 in *International Conference on Accelerator and Large Experimental Control Systems*, Seiten 1078–1083, Geneva, Switzerland, Jan. 2018. JACoW.
- [Kuleshov2007] [Kul07] KULESHOV, EUGENE: *Using the ASM framework to implement common Java bytecode transformation patterns.* 01 2007.
- [Li2017] [LTX17] LI, YUE, TIAN TAN und JINGLING XUE: *Understanding and Analyzing Java Reflection.* ACM Transactions on Software Engineering and Methodology, 28, 2017.
- ?Mancini? [MHM] MANCINI, FEDERICO, DAG HOVLAND und KHALID A. MUGHAL: *Investigating the limitations of Java annotations for input validation.*
- [Miroslav2009] [MJ09] MIROSLAV, SABO und PORUBÄN JAROSLAV: *Preserving Design Patterns using Source Code Annotations.* Journal of Computer Science and Control Systems, 2, 05 2009.
- [Meffert2006] [MP06] MEFFERT, KLAUS und ILKA PHILIPPOW: *Annotationen zur Anwendung beim Refactoring.* Oktober 2006.
- [Maier2010] [MRO10] MAIER, INGO, TIARK ROMPF und MARTIN ODERSKY: *Deprecating the Observer Pattern.* 01 2010.

- [Oracle2017] [Ora17] ORACLE: *Java SE Specifications*. <https://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html> jls-9.6, 2017. letzter Abruf: 26. Mai 2021.
- [Porubaen2009] [PFS09] PORUBÄN, JAROSLAV, MICHAL FORGÁČ und MIROSLAV SABO: *An-annotation based parser generator*. In: *2009 International Multiconference on Computer Science and Information Technology*, Seiten 707–714, 2009.
- ?Premkumar2010? [PM10] PREMKUMAR, LAWRENCE und PRAVEEN MOHAN: *Introduction to JavaFX*, Seiten 9–31. Apress, Berkeley, CA, 2010.
- [Pigula2015] [PN15] PIGULA, PETER und MILAN NOSAL: *Unified compile-time and run-time java annotation processing*. In: *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Seite 965–975, 2015.
- [Reineke2005] [Rei05] REINEKE, DETLEF: *Einführung in die Softwarelokalisierung*. Gunter Narr Verlag, 03 2005.
- [Rother2017] [Rot17] ROTHER, KRISTIAN: *Static Typing in Python*, Seiten 231–244. Apress, Berkeley, CA, 2017.
- [Rocha2011] [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.
- [Schildt2019] [Sch19] SCHILDT, HERBERT: *Java: The complete reference*, Kapitel Enumerations, Autoboxing, and Annotations, Seiten 452–506. New York: McGraw-Hill Education, 2019.
- [Sharan2015] [Sha15] SHARAN, KISHORI: *Learn JavaFX 8: Building User Experience and Interfaces with Java 8*. Apress, USA, 1st Auflage, 2015.
- [Salvanescchi2015] [SMT15] SALVANESCHI, GUIDO, ALESSANDRO MARGARA und GIORDANO TAMBURRELLI: *Reactive Programming: A Walkthrough*. Seiten 953–954, 05 2015.
- [Sulir2016] [SNP16] SULÍR, MATÚŠ, MILAN NOSÁL' und JAROSLAV PORUBÄN: *Recording concerns in source code using annotations*. Computer Languages, Systems & Structures, 46:44–65, 2016.
- [steyer2014] [Ste14] STEYER, RALPH: *Behind the scene – der Aufbau von FXML*, Seiten 123–142. 06 2014.
- [Tratt2009] [Tra09] TRATT, LAURENCE: *Dynamically Typed Languages*. Advances in Computers, 77:149–184, Juli 2009.

[vos2018][VCG⁺18]

VOS, JOHAN, STEPHEN CHIN, WEIQI GAO, JAMES WEAVER und DEAN IVERSON: *Using Scene Builder to Create a User Interface*, Seiten 129–191. Apress, Berkeley, CA, 2018.

[Rossum2014][vRLL14]

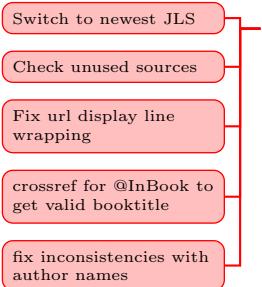
ROSSUM, GUIDO VAN, JUKKA LEHTOSALO und ŁUKASZ LANGA: *Function Annotations*. PEP 484, 2014.

[Winter2006][WL06]

WINTER, COLLIN und TONY LOWNDS: *Function Annotations*. PEP 3107, 2006.

[Yu2019][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.



Notes

Fix layouting: remove all newpage occurrences, fix space above and below figures, fix invalid spacing due to the overuse of the [H] figure float	1
Intro	3
Motivation	3
Zielsetzung	3
Struktur der Arbeit	3
Bugfixing in tex code	3
Correction	5
urls in footnotes	5
Move commented footnote to first occurrence (in intro)	11
maybe add code example	20
validate	25
Maybe add	28
Manual correct mistakes after this point	30
sonarlint	39
chapter to implementation	41
Maybe add functionality	46
the javafx menu base class to eventemitter restrictions	48
Intro	61
Entwicklung von Beispielsoftware	61
Vergleich konventioneller Methoden mit entwickeltem System	61
Intro	63
Zusammenfassung	63
Bewertung	63
Ausblick und mögliche Erweiterungen	63
Switch to newest JLS	82
Check unused sources	82
Fix url display line wrapping	82
crossref for @InBook to get valid booktitle	82
fix inconsistencies with author names	82

