



Department für Informatik

Abteilung für Medieninformatik und Multimedia-Systeme

Bachelorarbeit

Annotationsbasierte Einstiegserleichterung in
die Entwicklung von JavaFX-Anwendungen

Deniz Groenhoff

3. Juni 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 3. Juni 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

[Remove empty page](#)

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 Rolle ¹

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	6
2.2. JavaFX	7
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	12
2.3.3. Auswertung von Laufzeit-Annotationen	14
3. Stand der Technik	17
3.1. Aktuelle Verwendung von Annotationen	17
3.1.1. Annotationen im Umfeld von JavaSE/JavaEE/JavaFX	17
3.1.2. Annotationen in anderen Programmiersprachen	19
3.2. Maßnahmen zur Simplifizierung des Entwicklungsprozesses	20
3.2.1. Workflow Optimierung	20
3.2.2. Vereinfachung durch gesteigerte Übersichtlichkeit	20
3.2.3. Fazit	20
4. Konzeption und Entwurf	21
4.1. Anforderungsanalyse	21
4.1.1. Funktionale Anforderungen	21
4.1.2. Nichtfunktionale Anforderungen	21
4.2. Konzept und Modellierung	21
4.2.1. Designentscheidungen	21
4.2.2.	21

5. Implementierung	23
5.1. Architektur	23
5.1.1.	23
5.2.	23
6. Evaluation	25
6.1. Entwicklung von Beispielsoftware	25
6.2. Vergleich konventioneller Methoden mit entwickeltem System	25
7. Fazit	27
7.1. Zusammenfassung	27
7.2. Bewertung	27
7.3. Ausblick und mögliche Erweiterungen	27
A. Controllerbasierte JavaFX-Anwendung	29
Abkürzungsverzeichnis	33
Quellcodeverzeichnis	35
Abbildungsverzeichnis	37
Tabellenverzeichnis	39
Literaturverzeichnis	41

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

2. Grundlagen

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

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

`?(acro:xml)?` 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)-

`?(acro: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].

`?(acro:mvc)?` 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.

Correction

urls in footnotes

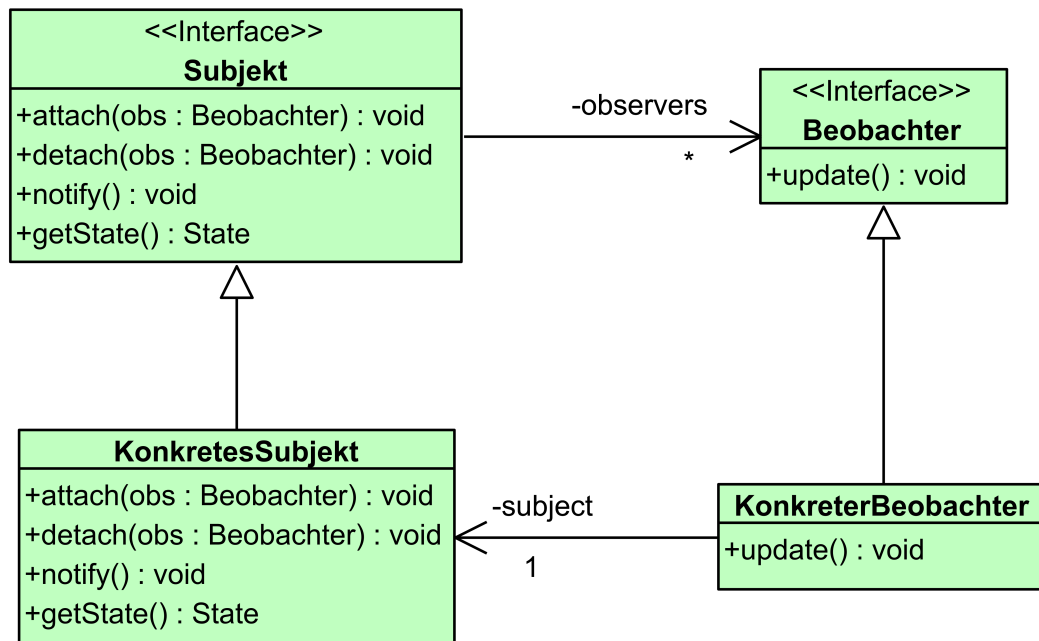


Abbildung 2.1.: UML-Diagramm – Beobachter-Entwurfsmuster

ig:observer_pattern)?

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 Entwurfsmuster 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.

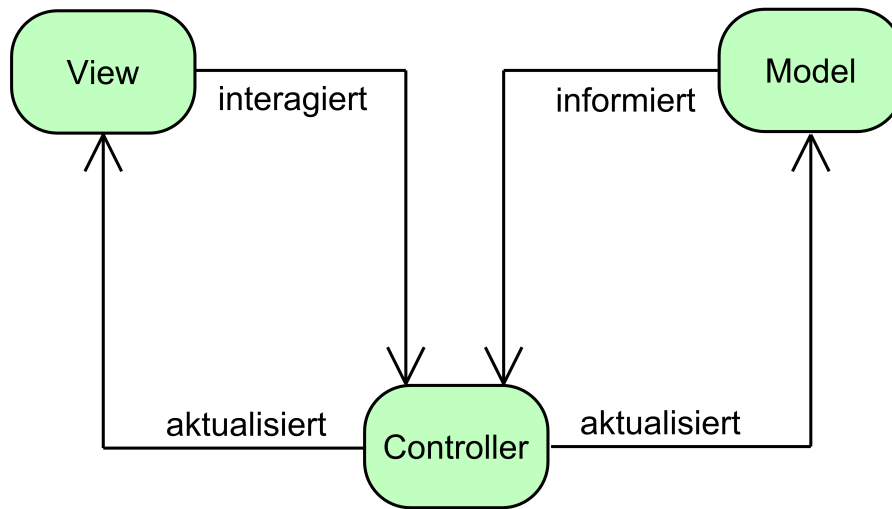


Abbildung 2.2.: Diagramm – MVC-Entwurfsmuster

Die vom MVC Muster vorgegebene Struktur 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 View und das Model sind vollständig entkoppelt und der Controller ist die Schnittstelle zwischen diesen.

2.2. 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 MVC- oder dem Beobachter-Muster, weshalb implementierte Anwendung selbst bei vielen Lines of Code (LoC), eine grundsätzlich hohe Strukturiertheit auf Quelltextebene aufweisen. Das grafische Layout kann dabei nicht ausschließlich durch Java-Quelltext sondern auch mittels der an die XML angelehnte Markup-Sprache FXML erstellt werden. Letzteres kann durch externe Tools wie dem Scene-Builder enorm vereinfacht werden [VCG⁺18].

2.2.1. Aufbau und Szenengraph

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

Die **Stage** Klasse repräsentiert ein Anwendungsfenster, welches das Design des Fensterlayouts des aktuell genutzten Betriebssystems 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 selbst erstellten JavaFX-Komponenten verantwortlich. Jede Komponente, welche durch eine Scene-Instanz angezeigt und verwaltet werden soll, wird in einer hierarchisch angeordneten, objektorientierten Datenstruktur eingefügt, 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.

maybe beautify with fancy arrows

```

ample_javafxapp)?
    public class TestApplication extends Application {

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

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

```

Code 2.1: Beispiel – Minimale JavaFX-Anwendung.

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 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].

```

roperty_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`.

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 Instanzieren eines neuen FXMLLoader Objekts, wie in Code 2.4 dargestellt, ermöglicht.

```
ample_fxmlayouting)? <?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.

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

Code 2.4: Beispiel – FXML Ladeprozess.

Um eine Controller-Klasse mit der FXML-Datei zu assoziieren, kann Wurzelelement dieser durch das `fx:controller` Attribut erweitert werden. Der Name des Controllers ist hierbei der voll qualifizierte Klassenname. 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.

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 Programmierneinsteiger vorteilhaft und durch ihre Anpassungsfähigkeit und Flexibilität sind sie in vielen Bibliotheken und Programmiersprachen vertreten.

2.3.1. Definition

Annotationen² 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 zu 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 nor-

²Wenn in der Arbeit über Annotationen gesprochen wird, ist immer von Java-Annotationen auszugehen (außer anders angegeben)

malen 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

Annotationen können Attribute besitzen, welche bei Kompilierzeit bzw. Laufzeit ausgelesen werden können. Die Typen dieser Attribute sind nicht vollständig frei wählbar – So ist es beispielsweise nicht möglich ein Attribut vom Typen `Object` in einer Annotation zu kapseln, ohne einen Kompilierfehler 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.

<pre>(lst:decl_normal)?public @interface Entity { String name(); int id(); }</pre>	<pre>(lst:appl_normal)?@Entity(name="test", id=2) public class TestEntity { // ... }</pre>
--	--

Code 2.7: Deklaration – Normal Annotation.

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.

<pre>(lst:decl_single)?public @interface Entity { String value(); int id() default -1; }</pre>	<pre>(lst:appl_single)?@Entity("test") public class TestEntity { // ... }</pre>
---	---

Code 2.9: Deklaration – Single-Element Annotation.

Code 2.10: Anwendung – Single-Element Annotation

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

<pre>(lst:decl_marker)?public @interface Entity { String name() default ""; int id() default -1; }</pre>	<pre>(lst:appl_marker)?@Entity public class TestEntity { // ... }</pre>
--	---

Code 2.11: Deklaration – Marker Annotation.

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]:

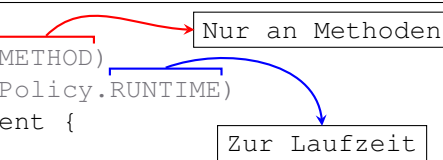
Quellcode-Annotationen sind nur beim Kompilervorgang auslesbar und können dem Compiler Anweisungen geben oder mithilfe von Annotation-Prozessoren z.B. neue Klassen automatisch generieren. Sie sind in der kompilierten Java-Anwendung nicht mehr erhalten.

Klassen-Annotationen sind nach dem Kompilierungsprozess noch in der Anwendung erhalten und können durch externe Tools wie z.B. dem Code-Obfuskator 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;
}

```

Code 2.13: Beispiel einer Laufzeit Annotation.

Add compile time annotation processing if used in this thesis

2.3.3. Auswertung von Laufzeit-Annotationen

n_laufzeitauswertung)

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 Instanzieren 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.

```
processing_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.

3. Stand der Technik

and_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.

urls in footnotes

complete intro

3.1. Aktuelle Verwendung von Annotationen

on_annotationen)?

Intro wie in github wiki

structure

3.1.1. Annotationen im Umfeld von JavaSE/JavaEE/JavaFX

umfeld_von_java)?

Nach dem Einführen von Annotationen in Java vor sechs 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 Semantikinformatoren 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:

JavaSE Umgebung

?(acro:jdk)?

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

ebendiese Methodendeklaration, sondern überlädt diese fälschlicherweise, so handelt es sich häufig dennoch um vollständig validen Quelltext, welcher aber unter Umständen zu einem ungewollten Verhalten führt. Wird aber die `@Override` Annotation in solchen Fällen über die zu überschreibenden Methoden geschrieben, so wird immer ein Kompilierfehler erzeugt. Ein Beispiel, welches dieses Szenario verbildlicht ist in Code 3.1 und Code 3.2 dargestellt.

<pre> ?{lst:decl_interface)? interface Test { default void t(int... i) {} } </pre>	<pre> ?{lst:compiler_error)? class TestClass implements Test { @Override public void t(int i) {} } </pre> <div style="position: relative; height: 40px;"> <div style="position: absolute; top: 0; right: 0; border: 1px solid black; padding: 2px 5px; font-size: 0.8em;">Kompilierfehler</div> <div style="position: absolute; top: 10px; left: 10px; color: red; font-size: 0.8em;">→</div> </div>
---	---

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.

¹<https://projectlombok.org>

²<https://www.android.com>

³<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.

JavaFX Umgebung

In JavaFX direkt werden nur wenige Annotationen verwendet, welche Teil der öffentlichen API sind. Dazu gehört `@FXML`, welche für das automatische Setzen von Feldern oder für die Identifikation von Methoden für EventHandler benötigt wird [AA19]. Der Entwickler kann somit Events, welche durch JavaFX-Komponenten ausgelöst werden, per FXML-Datei mit Methoden im selben Controller verbinden. Dazu wurden Bibliotheken wie `Afterburner.fx`⁴ entwickelt, welche durch `@Inject`, das Inversion of Control (IoC) Programmierparadigma durch Abhängigkeitsinjektion realisiert oder das von CERN entwickelte `ExtJFX`⁵, welches `@RunInFxThread` nutzt, um Unittests auf dem JavaFX-Thread auszuführen.

3.1.2. Annotationen in anderen Programmiersprachen

Neben den Java-Annotationen, welche in Abschnitt 2.3 erklärt und in Unterabschnitt 3.1.1 vorgestellt wurden, werden Annotationen auch in vielen anderen Programmiersprachen genutzt.

Verwendung in Python

Python ist eine dynamisch typisierte Sprache und validiert somit den Typen einer Variablen zur Laufzeit des Programms [Tra09], kann aber durch das Verwenden von Funktionsannotationen, Meta-Daten zu Parametern, Variablen und Funktionsrückgabewerten hinzufügen, um so den gewünschten Typen anzudeuten [vRLL14, WL06]. Diese Annotationen werden zwar vom Python-Interpreter ignoriert, können aber durch Softwaresysteme von Drittanbietern wie `mypy` zur statischen Typisierung verwendet werden. Nach einer Studie von Khan et al., welche 210 auf Python

⁴<https://github.com/AdamBien/afterburner.fx>

⁵<https://github.com/extjfx/extjfx>

basierende GitHub-Projekte auf typbezogene Fehler untersuchte, konnten 15% der gefundenen Mängel, durch mypy verhindert werden [KCV21]. 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 Programmcodetelementen in Python sinken – wenn auch nur implizit durch externe Bibliotheken oder Entwicklungsumgebungen.

maybe add code example

Verwendung in C# und .NET

In C# wird das Hinzufügen von Meta-Informationen zu bestehenden Programm-elementen 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. 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].

?{acro:dll}?

_in_c_sharp_dot_net)?

3.2. Maßnahmen zur Simplifizierung des Entwicklungsprozesses

Intro

twicklungsprozesses)?

Workflow Optimierung

sprozesses_workflow)?

Vereinfachung durch
gesteigerte Übersicht-
lichkeit

es_)?

Fazit

ungsprozesses_fazit)?

3.2.1. Workflow Optimierung

3.2.2. Vereinfachung durch gesteigerte Übersichtlichkeit

3.2.3. Fazit

4. Konzeption und Entwurf

Intro

4.1. Anforderungsanalyse

Intro (https://de.wikipedia.org/wiki/Software_anforderungsanalyse)

4.1.1. Funktionale Anforderungen

Funktionale Anforderungen als Unterpunkte

...

4.1.2. Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen als Unterpunkte

...

4.2. Konzept und Modellierung

Intro

4.2.1. Designentscheidungen

4.2.2. ...

5. Implementierung

implementierung)?

Implementierung

5.1. Architektur

?<architektur)?

Architektur

5.1.1. ...

5.2. ...

Extend

6. Evaluation

?(evaluation)?

Intro

6.1. Entwicklung von Beispielsoftware

ispielsoftware)?

Entwicklung von Beispielsoftware

6.2. Vergleich konventioneller Methoden mit entwickeltem System

h_system_javafx)?

Vergleich konventioneller Methoden mit entwickeltem System

7. Fazit

?<fazit>?

Intro

7.1. Zusammenfassung

zusammenfassung)?

Zusammenfassung

7.2. Bewertung

?<bewertung>?

Bewertung

7.3. Ausblick und mögliche Erweiterungen

e_erweiterungen)?

Ausblick und mögliche
Erweiterungen

A. Controllerbasierte JavaFX-Anwendung

afx_application)?

```
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.1: Anwendungscode.

```
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.2: Beispielcontroller.

```
<?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.3: FXML-Layout.

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

Code A.4: CSS-Design.

Abkürzungsverzeichnis

ungsverzeichnis)?

MVC Model-View-Controller

XML Extensible Markup Language

LoC Lines of Code

CSS Cascading Style Sheets

UML Unified Modeling Language

SoC Separation of Concerns

DLL Dynamic Link Library

IoC Inversion of Control

JDK Java Development Kit

order alphanumerical

maybe switch to the
acronym package for
automatic sorting

Quellcodeverzeichnis

2.1. Beispiel – Minimale JavaFX-Anwendung.	9
2.2. Beispiel – ChangeListener & EventHandler.	9
2.3. Beispiel – FXML Layouting.	10
2.4. Beispiel – FXML Ladeprozess.	10
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.	13
2.12. Anwendung – Marker Annotation	13
2.13. Beispiel einer Laufzeit Annotation.	14
2.14. Auslesen einer Laufzeit-Annotation.	15
3.1. Beispiel – Interfacedeklaration.	18
3.2. Beispiel – Kompilierfehler.	18
3.3. Beispiel – Lombok POJO.	19

Abbildungsverzeichnis

2.1. UML-Diagramm – Beobachter-Entwurfsmuster	6
2.2. Diagramm – MVC-Entwurfsmuster	7

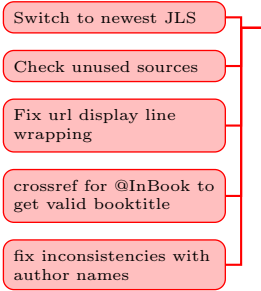
Tabellenverzeichnis

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*.
- [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 VLISSIDES 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 VLISSIDES: *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.
- [Hommel2013][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 MCGUIRE, DAVID F. SKLAR, JAMES D. FOLEY, STEVEN FEINER und KURT AKELEY: *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 (ICALPCS'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.
- [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 ODESKY: *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: *Annotation based parser generator*. In: *2009 International Multiconference on Computer Science and Information Technology*, Seiten 707–714, 2009.

- ?Premkumar2010? [PM10] PREM Kumar, LAWRENCE und PRAVEEN MOHAN: *Introduction to JavaFX*, Seiten 9–31. Apress, Berkeley, CA, 2010.
- Pigula2015 [PN15] FIGULA, 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.
- Rother2017 [Rot17] ROTHER, KRISTIAN: *Static Typing in Python*, Seiten 231–244. Apress, Berkeley, CA, 2017.
- Rocha2011 [RV11] ROCHA, HENRIQUE und MARCO TULLIO 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.
- Salvaneschi2015 [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 LUKASZ 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

<input type="checkbox"/>	Remove empty page	V
<input type="checkbox"/>	Intro	3
<input type="checkbox"/>	Motivation	3
<input type="checkbox"/>	Zielsetzung	3
<input type="checkbox"/>	Struktur der Arbeit	3
<input type="checkbox"/>	Bugfixing in tex code	3
<input type="checkbox"/>	Correction	5
<input type="checkbox"/>	urls in footnotes	5
<input type="checkbox"/>	maybe beautify with fancy arrows	8
<input type="checkbox"/>	Move footnote to first occurence	11
<input type="checkbox"/>	lst design	12
<input type="checkbox"/>	use lstnewenvironment	12
<input type="checkbox"/>	Add compile time annotation processing if used in this thesis	14
<input type="checkbox"/>	urls in footnotes	17
<input type="checkbox"/>	complete intro	17
<input type="checkbox"/>	Intro wie in github wiki	17
<input type="checkbox"/>	structure	17
<input type="checkbox"/>	maybe add code example	20
<input type="checkbox"/>	Intro	20
<input type="checkbox"/>	Workflow Optimierung	20
<input type="checkbox"/>	Vereinfachung durch gesteigerte Übersichtlichkeit	20
<input type="checkbox"/>	Fazit	20
<input type="checkbox"/>	Intro	21
<input type="checkbox"/>	Intro (https://de.wikipedia.org/wiki/Software_Requirements_Specification ?)	21
<input type="checkbox"/>	Funktionale Anforderungen als Unterpunkte	21
<input type="checkbox"/>	Nichtfunktionale Anforderungen als Unterpunkte	21
<input type="checkbox"/>	Intro	21
<input type="checkbox"/>	Implementierung	23
<input type="checkbox"/>	Architektur	23
<input type="checkbox"/>	Extend	23
<input type="checkbox"/>	Intro	25
<input type="checkbox"/>	Entwicklung von Beispielsoftware	25
<input type="checkbox"/>	Vergleich konventioneller Methoden mit entwickeltem System	25
<input type="checkbox"/>	Intro	27
<input type="checkbox"/>	Zusammenfassung	27
<input type="checkbox"/>	Bewertung	27

■ Ausblick und mögliche Erweiterungen	27
■ order alphanumerical	33
■ maybe switch to the acronym package for automatic sorting	33
■ Switch to newest JLS	44
■ Check unused sources	44
■ Fix url display line wrapping	44
■ crossref for @InBook to get valid booktitle	44
■ fix inconsistencies with author names	44