CARL VON OSSIETZKY universität OLDENBURG

Department für Informatik
Abteilung für Medieninformatik und Multimedia-Systeme

Bachelorarbeit

Annotationsbasierte Einstiegserleichterung in die Entwicklung von JavaFX-Anwendungen

Deniz Groenhoff

28. Mai 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 28. Mai 2021

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]

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

[1] Fussnote 1

Inhaltsverzeichnis

1. Einleitung

Intro

1.1. Motivation

Motivation

1.2. Zielsetzung

Zielsetzung

1.3. Struktur

Struktur der Arbeit

2. Grundlagen

Correction

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.
Abschließend wird das generelle Annotationenkonzept in der Informatik mit speziellen Fokus auf die Programmiersprache Java erklärt. Dabei werden die verschiedenen Annotationstypen näher beschrieben und jeweils mit Beispielen untermauert, sowie die Möglichkeiten der eigentlichen Auswertung von Annotationen skizziert.

2.1. Entwurfsmuster

Intro

2.1.1. Definition

Definition Entwurfsmuster

2.1.2. Notwendigkeit

Notwendigkeit & Justifikation von Entwurfsmustern

2.2. JavaFX

glossar für fxml z.B.?

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 Model-View-Controller (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 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 order 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, welche in der Computergrafik als Szenengraph bekannt ist [HvDM+13]. Jeder `Stage` muss zwangsläufig eine `Scene` zugewiesen werden.

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

Kindelemente besitzen darf, muss diese immer von der abstrakten `Parent`-Klasse erben. Das Layouting und die Positionierung im lokalen Koordinatensystem wird bei vorhandenen Kindelementen immer durch das Elternelement kontrolliert. Jede darzustellende Komponente muss von der `Node`-Klasse erben [Jun13].

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

```
public class TestApplication extends Application {

  public static void main(String[] args) {
    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

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 Beispiel aus Code 2.1 ist als FXML-Datei in Code 2.2 zu erkennen.

```
<?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 2.2: Beispiel – FXML Layouting.

2.2.4. Scene-Builder

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

2.3.1. Definition

Reference

Move footnote to first occurence

Annotationen [1] 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

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

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 2.3 dargestellt.

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

Code 2.3: 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 2.4 dargestellt, ein `@`-Präfix zum eigentlichen Klassennamen hinzugefügt werden.

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

Code 2.4: 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 2011, 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 [RV11].

2.3.2. Syntax

lst design

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.

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

Code 2.5: Deklaration – Normal Annotation.

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

Code 2.6: Anwendung – Normal Annotation

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

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

Code 2.7: Deklaration – Single-Element Annotation.

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

Code 2.8: Anwendung – Single-Element Annotation

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

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

Code 2.9: Deklaration – Marker Annotation.

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

Code 2.10: 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-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 2.11 zu erkennen.

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


Code 2.11: Beispiel einer Laufzeit Annotation.

Add compile time annotation processing if used in this thesis

2.3.3. Auswertung von Laufzeit-Annotationen

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 2.11 vordefinierten Annotation ist in Code 2.12 zu erkennen.

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

Code 2.12: Auslesen einer Laufzeit-Annotation.

2.3.4. Beispiele der Annotationsprogrammierung

Beispiele der Annotationsprogrammierung

3. Stand der Technik

Intro

3.1. Aktuelle Verwendung von Annotationen

Intro

3.1.1. JavaFX

JavaFX Beispiele

3.1.2. Android

Android Beispiele

3.1.3. JavaX

JavaX Beispiele (z.B. JAXB)

3.2. Maßnahmen zur Simplifizierung des Entwicklungsprozesses

Intro

3.2.1. Workflow Optimierung

Workflow Optimierung

3.2.2. Vereinfachung durch gesteigerte Übersichtlichkeit

Vereinfachung durch gesteigerte Übersichtlichkeit

3.2.3. Fazit

Fazit

4. Konzeption und Entwurf

Intro

4.1. Anforderungsanalyse

Intro (https://de.wikipedia.org/wiki/So
?)

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

5.1. Architektur

Architektur

5.1.1. ...

5.2. ...

Extend

6. Evaluation

Intro

6.1. Entwicklung von Beispielsoftware

Entwicklung von Beispielsoftware

6.2. Vergleich konventioneller Methoden mit entwickeltem System

Vergleich konventioneller Methoden mit entwickeltem System

7. Fazit

Intro

7.1. Zusammenfassung

Zusammenfassung

7.2. Bewertung

Bewertung

7.3. Ausblick und mögliche Erweiterungen

Ausblick und mögliche Erweiterungen

A. Appendix 1

B. Appendix 2

Abkürzungsverzeichnis

MVC Model-View-Controller

XML Extensible Markup Language

LoC Lines of Code

Quellcodeverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AA19] Anderson, Gail und Paul Anderson: *The Definitive Guide to Modern Java Clients with JavaFX*, Kapitel JavaFX Fundamentals, Seiten 33–80. Stephen Chin, Johan Vos, James Weaver, 2019.

[Dea95] Deacon, John: *Model-View-Controller (MVC) Architecture*. Online, August 1995.

[DPV+07] Danelutto, Marco, Marcelo Pasin, Marco Vanneschi, Patrizio Dazzi, Domenico Laforenza und Luigi Presti: *PAL: Exploiting Java Annotations for Parallelism*, Seiten 83–96. 2007.

[ESM05] Eichberg, Michael, Thorsten Schäfer und Mira Mezini: *Using Annotations to Check Structural Properties of Classes*. In: Cerioli, Maura (Herausgeber): *Fundamental Approaches to Software Engineering*, Seiten 237–252, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[FFI+04] Forman, Ira R., Nate Forman, Dr. John Vlissides Ibm, Ira R. Forman und Nate Forman: *Java Reflection in Action*, 2004.

[Gao19] Gao, Weiqi: *The Definitive Guide to Modern Java Clients with JavaFX*, Kapitel Properties and Bindings, Seiten 81–141. Stephen Chin, Johan Vos, James Weaver, 2019.

[GHJV94] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Seiten 1–4, 293–303, 1994.

[GJSB05] Gosling, James, Bill Joy, Guy Steele und Gilad Bracha: *The Java Language Specification, Third Edition*, Seiten 268–281. 2005.

[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.

[JN20] Jha, Ajay und Sarah Nadi: *Annotation practices in Android apps*. 2020.

[Jun13] Juneau, Josh: *JavaFX in the Enterprise*, Seiten 615–646. Apress, Berkeley, CA, 2013.

[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.

[LTX17] Li, Yue, Tian Tan und Jingling Xue: *Understanding and Analyzing Java Reflection.* ACM Transactions on Software Engineering and Methodology, 28, 2017.

[MHM] Mancini, Federico, Dag Hovland und Khalid A. Mughal: *Investigating the limitations of Java annotations for input validation.*

[MP06] Meffert, Klaus und Ilka Philippow: *Annotationen zur Anwendung beim Refactoring*, Oktober 2006.

[Ora17] Oracle: *Java SE Specifications.* https://docs.oracle.com/javase/specs/jls/se7/html/jls
9.htmljls-9.6, 2017. letzter Abruf: 26. Mai 2021.

[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.

[PM10] Premkumar, Lawrence und Praveen Mohan: *Introduction to JavaFX*, Seiten 9–31. Apress, Berkeley, CA, 2010.

[PN15] Pigula, Peter und Milan Nosal: *Unified compile-time and runtime java annotation processing.* In: *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Seite 965–975, 2015.

[RV11] Rocha, Henrique und Marco Tulio Valente: *How Annotations are Used in Java: An Empirical Study.* International Conference on Software Engineering and Knowledge Engineering, 2011.

[Sch19] Schildt, Herbert: *Java: The complete reference*, Kapitel Enumerations, Autoboxing, and Annotations, Seiten 452–506. New York: McGraw-Hill Education, 2019.

[Sha15] Sharan, Kishori: *Learn JavaFX 8: Building User Experience and Interfaces with Java 8.* Apress, USA, 1st Auflage, 2015.

[Ste14] Steyer, Ralph: *Behind the scene – der Aufbau von FXML*, Seiten 123–142. 06 2014.

[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.

[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.

Switch to newest JLS

Notes