Automatische Bewertung von JavaFX-Anwendungen

David Schuster[1] Ayla Brettle[2] Rainer Oechsle[3] Florian Grummel[4]

Abstract: As part of programming training the Department of Computer Science at Trier University of Applied Science offers courses to impart various programming languages and concepts. The course “Graphical user interfaces“ (GUI) introduces the basic techniques of programming with graphical user interfaces. Based on this the students learn how to use important design and architectural patterns. For this purpose, the class library JavaFX is used. The implementations of the lecture-accompanying exercises should also be implemented with JavaFX. To offer the students a feedback on the correctness of their submitted solutions, a system for automatic software evaluation has been in use since 2006. This article describes the general structure of the system and the extensions that were necessary for the automated testing and evaluation of JavaFX applications.

Abstract: Im Rahmen der Programmierausbildung werden im Fachbereich Informatik der Hochschule Trier Module angeboten, in denen verschiedene Programmiersprachen und Konzepte vorgestellt werden. In der Vorlesung „Grafische Benutzeroberflächen“ sollen Studierende zunächst die Grundlagen der Programmierung mit grafischen Benutzeroberflächen und darauf aufbauend u.a. wichtige Entwurfs- bzw. Architekturmuster kennenlernen. Dazu wird die Klassenbibliothek JavaFX verwendet, mit der auch zusätzlich die Implementierungen der vorlesungsbegleitenden Aufgaben umgesetzt werden sollen. Um den Studierenden ein Feedback über die Korrektheit ihrer eingereichten Lösungen zu ermöglichen, wird seit 2006 ein System zur automatischen Software-Bewertung (ASB) eingesetzt. In diesem Beitrag werden zunächst der allgemeine Aufbau des ASB-Systems beschrieben und darauf aufbauend die Erweiterungen erläutert, die für das automatisierte Testen und Bewerten von JavaFX-Anwendung innerhalb des ASB-Systems notwendig waren.

Keywords: Web-Anwendung; WebSocket; automatische Bewertung; JavaFX; JUnit; Checkstyle; TestFX; Monocle

1 Einleitung

Im Fachbereich Informatik der Hochschule Trier werden im Rahmen der Programmierausbildung Module (Vorlesungen inklusive Übungen) angeboten, in denen verschiedene Programmiersprachen vorgestellt und gelehrt werden. In dem Modul „Grafische Benutzeroberflächen“ (GBO) sollen Studierende grundlegende Konzepte zur Programmierung grafischer Benutzeroberflächen mit der *JavaFX*[5] -Klassenbibliothek von Java kennenlernen.

[1] Hochschule Trier, FB Informatik, Postfach 1826, 54208 Trier, Deutschland Da.Schuster@hochschule-trier.de
[2] Hochschule Trier, FB Informatik, Postfach 1826, 54208 Trier, Deutschland A.Brettle@hochschule-trier.de
[3] Hochschule Trier, FB Informatik, Postfach 1826, 54208 Trier, Deutschland oechsle@hochschule-trier.de
[4] Hochschule Trier, FB Informatik, Postfach 1826, 54208 Trier, Deutschland F.Grummel@hochschule-trier.de
[5] http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

In der Vorlesung werden zunächst einführende Themen wie das Anzeigen eines Fensters auf der Oberfläche und die Ereignisbehandlung mit Interaktionselementen behandelt. Neben der Vorlesung sollen die Studierenden regelmäßig, meistens wöchentlich, Programmieraufgaben lösen und einreichen. Eine mögliche Programmieraufgabe für genannte einleitende Themen könnte die Implementierung eines Zählers sein. Dieser soll durch den Benutzer des Programms über einen Plus- und einen Minus-Button erhöht und verringert werden können. Des Weiteren soll dem Benutzer der Zählerwert auf der Oberfläche angezeigt werden. Abbildung 1 zeigt eine mögliche Umsetzung dieser Aufgabenstellung.

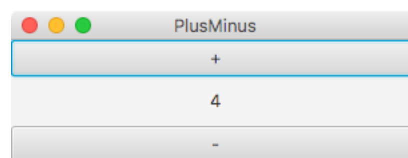

Abb. 1: Benutzeroberfläche des Zähler-Beispiels

Die Implementierung einer solchen Anwendung mit der Grafikbibliothek JavaFX ist mit wenigen Zeilen Programmiercode möglich. Da im Verlauf der Vorlesung wichtige Entwurfs- bzw. Architekturmuster wie „Observer“ oder „Model-View-Presenter“ (MVP) vorgestellt und dazu auch Programmieraufgaben gestellt werden, wächst auch das Niveau und die Komplexität der Aufgaben. Die Lösung für eine Aufgabenstellung, in der eine Anwendung nach dem MVP-Muster implementiert werden soll, kann sicherlich aus mehreren hundert Zeilen Programmcode und mehr als 10 Klassen bzw. Schnittstellen bestehen. Die im Verlauf des Semesters wachsende Größe der Aufgaben und die Anzahl der Studierenden macht ein händisches Bewerten der eingereichten Lösungen unmöglich. Um den Studierenden ein sofortiges Feedback zu ermöglichen, wird seit 2006 die Client-Server-Anwendung ASB (Automatische Software-Bewertung) im Fachbereich Informatik der Hochschule Trier eingesetzt und entwickelt. Wie auch mit anderen Bewertungssystemen bzw. Gradern wie beispielsweise Graja ([Fr15]), PABS ([If15]) oder JACK ([St17]) lässt sich mit dem ASB-System Software zunächst ausführen und im Anschluss auf vorher definierte Eigenschaften überprüfen. In diesem Kontext handelt es sich bei der Software jedoch um Programme mit grafischen Benutzeroberflächen, sodass für die Ausführung und das Testen der Benutzerschnittstelle weitere Voraussetzungen erfüllt sein müssen. Zum aktuellen Stand konnte kein System identifiziert werden, welches JavaFX-Anwendungen automatisiert innerhalb einer Client-Server-Anwendung bewertet. In diesem Beitrag wird im Wesentlichen beschrieben, wie das ASB-System erweitert wurde, um JavaFX-Anwendungen automatisiert bewerten zu können und welche Probleme bzw. Herausforderungen dadurch entstanden sind (Kapitel 3). Zunächst erfolgt in Kapitel 2 eine Vorstellung des ASB-Systems. Abgeschlossen wird der Beitrag mit einer Zusammenfassung und einem Ausblick (Kapitel 4).

2 Das ASB-System

Die webbasierte Client-Server-Anwendung zur automatischen Bewertung von Software (ASB) ist ein System, welches im Fachbereich Informatik der Hochschule Trier eingesetzt wird. Das ASB-System ermöglicht die automatisierte Bewertung von Software und somit auch eine zeitnahe Rückmeldung zu den gefundenen Mängeln des eingereichten Programms. Im Folgenden wird der grundlegende Aufbau des Systems vorgestellt. Dazu gehören die Ausführungsumgebungen, die Plugins und deren Konfigurationen. Im Rahmen dieses Beitrags kann nicht auf alle Komponenten im Detail eingegeangen werden. Nähere Informationen zum ASB-System und dessen Einsatz sind in [HOS17b] gegeben.

Das System zur automatischen Bewertung von Software wurde als ein Komponentensystem entwickelt. Vorteil einer solchen Architektur ist die Modularität. Komponenten können zur Laufzeit angelegt, bearbeitet oder gelöscht werden. Der ASB-Kern stellt die Hauptkomponente des Systems dar. Dieser realisiert neben der auf WebSockets basierten, asynchronen Client-Server-Kommunikation auch die Verwaltung der Datenbank. Des Weiteren werden am ASB-Kern Komponenten angemeldet, die für die automatisierte Ausführung und Bewertung von Software benötigt werden. Abbildung 2 zeigt den strukturellen Aufbau des ASB-Systems inklusive der verwendeten Komponenten.

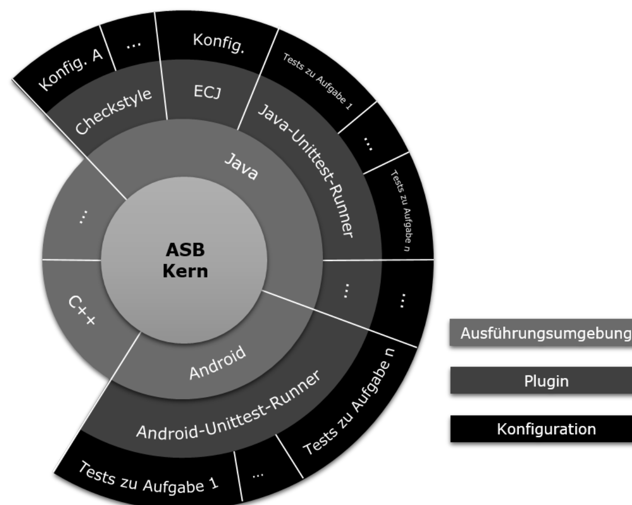

Abb. 2: Struktureller Aufbau des ASB-Systems

Um ein Programm in einer beliebigen Programmiersprache ausführen zu können, werden innerhalb des ASB-Systems sogenannte *Ausführungsumgebungen (AU)* eingesetzt und entwickelt. Für das ASB-System wurden bisher Ausführungsumgebungen für die Programmiersprachen Java, C++, Python und für die Software-Plattform Android [He15] implementiert. Die Ausführungsumgebungen ermöglichen das Kompilieren und Ausführen von Software in einer beliebigen Programmiersprache. Die eigentliche Bewertung eines

eingereichten Programms realisiert das *Plugin*. Das Plugin legt fest, welche Bewertungsmaßnahmen für das eingereichte Programm angewendet werden sollen, wie lange die Bewertung maximal dauern darf und erstellt anschließend das Feedback der Bewertung. Dazu werden nach dem Test die Ergebnisse aller Bewertungsmaßnahmen ausgewertet, in eine XML-Ergebnisdatei geschrieben und für die Darstellung im Browser aufbereitet.

Zu einem Plugin kann es unterschiedliche *Konfigurationen* geben. Sie unterscheiden sich je nach Plugin in Art und Umfang. Beispielsweise bestehen die Konfigurationen für das Checkstyle-Plugin[6] jeweils aus einer XML-Datei, in der die Codierungskonventionen definiert sind. Die Konfigurationen für den Java-Unittest-Runner enthalten die durchzuführenden Testfälle. Das heißt, für die Bewertung von Java-Programmen beinhaltet eine Konfiguration eine Anzahl von Java-Klassen mit Testmethoden. Um JavaFX-Anwendungen automatisiert bewerten zu können, wurde das bereits bestehende Plugin Java-Unittest-Runner erweitert und neue Konfigurationen implementiert. Im Folgenden wird im Detail auf die Umsetzung dieser Weiterentwicklung eingegangen.

3 Bewertung von JavaFX-Anwendungen

Bevor eine Software dynamisch getestet werden kann, muss das Programm zur Ausführung gebracht werden. Im Falle einer hochgeladenen, studentischen Lösung wie beispielsweise der in Kapitel 1 vorgestellten Einführungsaufgabe muss beim Ausführen der eingereichten JavaFX-Anwendung eine grafische Oberfläche aufgebaut werden, um anschließend die von der Aufgabe geforderten Funktionen durch Funktionstests zu prüfen. Das ASB-System läuft auf einem Ubuntu-Server[7], der über keine Desktopumgebung verfügt. Das heißt, es können keine grafischen Elemente dargestellt werden. Das Fehlen einer Desktopumgebung bei einem Serversystem ist nicht ungewöhnlich, da in den meisten Fällen eine klassische text- bzw. kommandobasierte Benutzung ausreichend ist und somit Ressourcen auf dem Server eingespart werden können. Die Testfälle für JavaFX-Anwendungen müssen demnach ohne Desktopumgebung ausgeführt werden können. Dieses Testen von Programmen mit einer Benutzerschnittstelle ohne den tatsächlichen Aufbau eines Fensters wird als „*Headless*-Test“ bezeichnet. Um die von der Aufgabenstellung geforderten Funktionen zu prüfen, müssen nach dem Start der Anwendung Interaktionen wie beispielsweise ein Klick auf einen Button oder eine Texteingabe in einer Textbox auf der Anwendung durchgeführt werden. Somit muss es für die automatisierte Bewertung von JavaFX-Anwendungen möglich sein, das Programm *headless* auszuführen, vorhandene JavaFX-Elemente zu ermitteln und auf diesen Interaktionen zu simulieren. Im Weiteren wird beschrieben, wie das ASB-System angepasst werden musste, um diese Anforderungen zu erfüllen und darauf aufbauend die Bewertungen für die Programmieraufgaben für das Modul „Grafische Benutzeroberflächen“ zu realisieren.

[6] http://checkstyle.sourceforge.net/

[7] https://www.ubuntu.com/server

3.1 Anpassungen an das ASB-System

Das Ausführen von Testfällen für Java-Programme setzt ein für das ASB-System entwickeltes Plugin namens Java-Unittest-Runner um. Dazu verwendet es u.a. die Test-Bibliothek JUnit[8]. Diese beinhaltet eine Anzahl von Methoden und Klassen, mit denen es möglich ist, Zustände oder Werte miteinander zu vergleichen um somit einen Fehler zu lokalisieren. Um Tests auf JavaFX-Anwendungen durchzuführen, wurden 2014 im Rahmen einer Bachelorarbeit verschiedene JavaFX-Test-Klassenbibliotheken u.a. anhand der oben beschriebenen Kriterien für den möglichen Einsatz im ASB-System validiert. Die Wahl fiel auf das Framework *TestFX*[9]. Die Klassenbibliothek bietet zum einen die Möglichkeit Interaktionen auf JavaFX-Elementen zu simulieren und zum anderen die Anwendung in einem sogenannten „*Headless*-Modus“ zu starten. Letzeres wird mit einer von TestFX benutzten Klassenbibliothek namens *Monocle*[10] realisiert. Diese stellt den Zugriff auf Low-Level-Betriebssystemroutinen zur Verfügung wie beispielsweise die Verwaltung der Fenster oder Timer. Dieser Zugriff wurde für unterschiedliche Fenstersysteme entwickelt. Ein Fenstersystem stellt die Grundelemente zum Bau einer grafischen Benutzerschnittstelle auf Betriebssystemebene wie zum Beispiel das Zeichnen eines Fensters oder das Behandeln von Nutzerinteraktionen zur Verfügung. Neben der Unterstützung von Standardfenstersystemen wurde auch ein *„Headless“*-Modus implementiert. Dieser kann verwendet werden, wenn das Betriebssystem über kein Fenstersystem verfügt und ein Programm, das eigentlich eine grafische Benutzeroberfläche aufbaut, gestartet werden, jedoch kein Fenster aufgebaut werden soll. Da der Server des ASB-Systems wie oben erläutert über kein Fenstersystem verfügt, wird zum automatisierten Bewerten von JavaFX-Anwendungen der „*Headless*-Modus“ von Monocle genutzt.

Um eine JavaFX-Anwendung, nachdem sie *headless* gestartet wurde, automatisiert zu testen, müssen Nutzerinteraktionen simuliert werden. Eine solche Interaktion kann ein Klick oder eine Tastatureingabe sein. Die Bibliothek TestFX stellt dafür sogenannte „*Robots*“ zur Verfügung. Diese können dann die Interaktionen auf JavaFX-Elemente wie beispielsweise einen Button oder eine Textbox ausführen. Die JavaFX-Elemente werden anhand einer zuvor definierten *ID*, Aufschrift oder anderen Merkmalen (z.B. Farbe, vorhandene Eltern- oder Kindknoten) ermittelt. Nachdem eine Aktion ausgeführt wurde und sich dadurch der Zustand bzw. ein Wert im Fenster geändert hat, können diese unter Verwendung der JUnit-Bibliothek auf Korrektheit geprüft werden. So kann beispielsweise der Inhalt eines Labels oder einer Textbox mit einer erwarteten Zeichenkette verglichen werden. Um diese Funktionalität nutzen zu können, wurde das Framework TestFX dem Klassenpfad des Java-Unittest-Runner hinzugefügt. Somit können sämtliche Funktionen zum Testen und Bewerten der Anwendung innerhalb der Konfigurationen, die Schicht im ASB-System, die die eigentlichen Tests definiert, genutzt werden. Im Folgenden wird im Detail erläutert, wie ein Testfall im ASB-System unter Verwendung der vorgestellten Komponenten implementiert ist.

[8] http://junit.org/junit4/
[9] https://github.com/TestFX/TestFX
[10] https://wiki.openjdk.java.net/display/OpenJFX/Monocle

3.2 Testfall im Detail

Unabhängig davon, ob lokal auf einem Desktop-Rechner oder auf dem Ubuntu-Server des ASB-Systems getestet wird, können die gleichen Testfälle verwendet werden. Anhand der PlusMinus-Aufgabe (Abbildung 1) aus der Einleitung wird gezeigt, wie ein Funktionstest aussehen kann und welche Arten von Tests eingesetzt werden. Für die Überprüfungen von Studierendenlösungen durch das ASB-System sind zunächst Signaturprüfungen sinnvoll, da die Lösungen von Studierenden eingereicht werden und der Inhalt somit wenig vorhersehbar ist. Durch die Signaturprüfungen wird sichergestellt, dass die geforderten Klassen in der korrekten Packagestruktur vorliegen. Bei JavaFX-Anwendungen wird zudem geprüft, ob das obligatorische Ableiten von der JavaFX-Klasse `Application` vorgenommen wurde. Die benötigte Funktionalität stellt das Plugin Java-Unittest-Runner mithilfe eines AST (Abstract Syntax Tree)-Parsers bereit. Weitere Prüfungen beispielsweise von Methodensignaturen sind ebenfalls möglich, werden hier jedoch nicht benötigt und daher nicht weiter beschrieben.

Neben der Signaturprüfung existieren u.a. Funktionstests, die das korrekte Verhalten der PlusMinus-Anwendung testen. Bei den Funktionstests handelt es sich um reine Blackbox-Tests, die ohne Kenntnisse über die innere Funktionsweise das Verhalten der eingereichten Lösung auf der Oberfläche prüfen. Das Listing 1 zeigt als Beispiel für einen solchen Test die Überprüfung des einmaligen Inkrementieren des Zählers.

```
1  @Test
2  @TestOrder(11)
3  @DependsOnCorrectnessOf("FunctionalityTest.testInitialState")
4  public void testIncrementOnce() {
5     Label counterLabel = lookup("#counterLabel").query();
6     try {
7        int initialcounter = Integer.parseInt(counterLabel.getText());
8        clickOn("#plus");
9        int expectedValue = initialcounter + 1;
10       Assert.assertThat("Ihr Label zeigt jedoch die Zeichenkette \"" +
             counterLabel.getText() + "\" statt \"" + expectedValue + "\" an.",
             counterLabel, hasText(String.valueOf(expectedValue)))
11    } catch (NumberFormatException e) {
12       Assert.fail("Ihr Label zeigt jedoch initial keinen ganzzahligen, dezimalen
             Zahlenwert, sondern \"" + counterLabel.getText() + "\".");
13     }
14 }
```

List. 1: Beispiel für eine Test-Methode

Wie bei JUnit-Tests üblich wird die als Test durchzuführende Methode mit der JUnit-Annotation `@Test` gekennzeichnet. Darunter folgen optional die Java-Unittest-Runner-eigenen Annotationen. Deren Angaben dienen zum Bestimmen der Reihenfolge der einzelnen Testfälle (Zeile 2) oder um Abhängigkeiten zu anderen Tests festzulegen (Zeile 3). In

diesem Fall hängt der Test von einem weiteren Test ab, der in der Methode `testInitialState` der Klasse `FunctionalityTest` beschrieben ist. Sollte jener Test bereits fehlschlagen, wird dieser Test gar nicht erst ausgeführt. Weitere Annotationen z.B. für Titel und Beschreibungen des Tests für die spätere Bewertungsanzeige sind ebenfalls möglich. In der Testmethode selbst wird zunächst über die von TestFX bereitgestellten Methoden `lookup()` und `query()` eine Referenz auf das Label bezogen. Sollte die eingereichte Lösung des Studierenden nicht über ein Element mit dieser ID verfügen, würde dieser Aufruf `null` zurückliefern. Bei diesem Test als Teil der Konfiguration, also als Teil der Gesamtheit aller Tests zu dieser Aufgabe, wird jedoch durch die Abhängigkeit zu separaten Existenztests sichergestellt, dass alle Funktionstests erst durchgeführt werden, wenn die Benutzeroberfläche der Studierendenlösung tatsächlich über die geforderten Elemente verfügt. Ansonsten müsste an dieser Stelle im Testfall selbst eine Überprüfung auf das Vorhandensein stattfinden. Vor allem bei umfangreicheren Aufgaben werden auf diese Weise einerseits redundante Überprüfungen in jedem Funktionstest vermieden. Andererseits kann den Studierenden auch eine gezieltere Rückmeldung gegeben werden, wenn das gesuchte Element nicht vorhanden ist.

Über das Label wird darauffolgend der Startwert des Zählers ausgelesen. Nachdem über den Aufruf der Methode `clickOn()` (Zeile 8) der Klick auf den Plus-Button durch den Klick-Robot von TestFX durchgeführt wurde, kann mittels der Klasse `Assert` aus JUnit der Labelinhalt auf den erwarteten Zählerwert überprüft werden. Abweichende Ergebnisse sowie die Auflistung der bestandenen Tests werden dem Studierenden nach Einreichen seiner Lösung auf dem ASB-System mit näherer Beschreibung auf der Weboberfläche präsentiert.

3.3 Herausforderungen

Theoretisch lassen sich mit den gezeigten Mitteln bereits viele, auch weitaus größere Anwendungen testen. Neben der Interaktion des Klickens werden alle weiteren über Maus und Tastatur bekannten Interaktionen durch TestFX bereitgestellt und lassen sich ähnlich des Klicks auf den Plus-Button in den Test einbinden. Auch kompliziertere Interaktionselemente wie beispielsweise Tabellen sind überprüfbar und das Testen über mehrere Fenster oder Dialoge hinweg ist möglich. Mit der steigenden Komplexität der Anwendungen steigt jedoch die Anzahl der möglichen Testfälle sowie die Anzahl der auszuführenden Interaktionen, sodass sich die einstmals kleinen Testfälle zu großen Szenarien ausweiten. Eine Abwägung über den Nutzen der einzelnen Tests auch im Verhältnis zum Aufwand für die Entwicklung der Tests und zur gesamten Testdauer wird zwangsläufig erforderlich.

Eine weitere Problematik soll anhand eines weiteren Beispiels erklärt werden. In einer fortgeschritteneren Aufgabe der Lehrveranstaltung muss ein Programm zum Auswählen und Bestellen von Pizza implementiert werden. Die geforderte grafische Benutzeroberfläche zeigt dazu ein Formular mit möglichen Zutaten und Größen der angebotenen Pizzas. Abhängig von den gewählten Zutaten und der Größe der Pizza soll beim Klicken auf einen „Bestellen“-Button der Preis sowie die Auflistung der konfigurierten Pizza in einer

Textbox erscheinen. Für viele Studierende liegt hierbei die Schwierigkeit im dynamischen Aufbau der Oberfläche. So sollen die Zutaten und Größen für die Pizza „von außen“ übergeben werden, das heißt, die Daten sind nicht Teil des Programmcodes des Studierenden, sondern werden z.B. erst beim Start der Anwendung in das Programm hineingegeben, sodass die Oberfläche anhand dessen aufgebaut werden muss und nicht vom Studierenden auf feste Zutaten und Größen gesetzt wird. Damit der dynamische Aufbau durch das ASB-System überprüft werden kann, müssen bei jeder Ausführung zufällige Testdaten verwendet werden, sodass der Studierende nicht mit einer vorgefertigten Preisausgabe oder im Programmcode hinterlegten Zutaten bzw. Zutatenanzahlen die Tests besteht. Die Hineingabe der Testdaten in die Anwendung über einen Konstruktor der `Application`-Subklasse ist nicht möglich, da JavaFX-Anwendungen über den Standardkonstruktor via Reflection von JavaFX gestartet und in einem separaten GUI-Thread ausgeführt werden. Die Zurverfügungstellung einer Java-Klasse, die den Datenbestand beinhaltet, hätte zwar didaktische Vorteile in Hinblick auf das MVP-Entwurfsmuster, jedoch müsste diese Klasse auf dem ASB-System durch eine Klasse mit den zufälligen Testdaten ersetzt werden. Ohne größeren Aufwand am ASB-System war das Ersetzen oder ein Verbot zum Hochladen der Konfigurationsklasse nicht zu leisten, sodass entschieden wurde, die doch eher geringe Menge an Testdaten als Kommandozeilenargumente beim Programmstart zu übergeben. Somit verbleibt für die Studierenden ein vertretbarer Aufwand beim selbstständigen Ausprobieren ihrer Anwendungen. Bei größeren Testdatenmengen erfolgt die Eingabe über einzulesende Dateien.

Strukturelle Tests zur Überprüfung, ob bestimmte Entwurfsmuster eingehalten wurden, können nicht über Funktionstests realisiert werden. Hierfür sind statische Tests nötig. Bisher werden bei Aufgaben, die das MVP-Entwurfsmuster verlangen, Signaturtests zum Auffinden von verbotenen Attributen verwendet. Enthält das Modell einer Anwendung beispielsweise eine Referenz auf die View, so widerspricht dies dem MVP-Muster. Feinere Verstöße gegen das Entwurfsmuster sind dagegen schwierig zu testen und benötigen Überprüfungen durch das Lehrpersonal.

4 Zusammenfassung und Ausblick

In diesem Beitrag wurde beschrieben, wie das ASB-System grundlegend aufgebaut ist und darauf basierend die Erweiterungen, die vorgenommen wurden, um JavaFX-Anwendungen *headless* innerhalb einer Server-Umgebung, das heißt ohne vorhandene Desktopumgebung, testen zu können. Im Anschluss wurde die Umsetzung eines Tests anhand eines Beispiels beschrieben und die Herausforderungen des automatisierten Testens von JavaFX-Anwendungen erläutert. Die Erweiterung des ASB-Plugins Java-Unittest-Runner und die Implementierung der dazugehörigen Konfigurationen zum Testen von JavaFX-Anwendungen sind seit 2015 für das Modul „Grafische Benutzeroberflächen“ produktiv im Einsatz. Die Erfahrungen, die mit den in diesem Beitrag beschriebenen Erweiterungen gemacht wurden, sind überwiegend positiv und unterscheiden sich nicht signifikant von denen, die bereits mit anderen Plugins

gesammelt wurden. [HOS17a] beschreibt einen detaillierten Erfahrungsbericht, sowohl aus der Studierenden- als auch aus der Dozentensicht. Um den Studierenden in Zukunft ein noch besseres Feedback zu ihren eingereichten Lösungen zu bieten, werden in Hinblick auf die in 3.3 vorgestellten Herausforderungen und Probleme weitere und genauere Tests implementiert.

Literaturverzeichnis

[Fr15] Fricke, Peter; Garmann, Robert; Heine, Felix; Kleiner, Carsten; Reiser, Paul; Peratoner, Immanuel De Vere; Grzanna, Sören; Wübbelt, Peter; Bott, Oliver J.: Grading mit Grappa – Ein Werkstattbericht. 2. Workshop Automatische Bewertung von Programmieraufgaben, 2015.

[He15] Heimann, Mathis; Fries, Patrick; Herres, Britta; Oechsle, Rainer; Schmal, Christian: Automatische Bewertung von Android-Apps. 2. Workshop Automatische Bewertung von Programmieraufgaben, 2015.

[HOS17a] Herres, Britta; Oechsle, Rainer; Schuster, David: Automatisierte Bewertung im Kontext der App-Entwicklung am Beispiel Android. In: Automatisierte Bewertung in der Programmierausbildung. Oliver J. Bott and Peter Fricke and Uta Priss and Michael Striewe, 2017. ISBN: 978-3-8309-3606-0.

[HOS17b] Herres, Britta; Oechsle, Rainer; Schuster, David: Der Grader ASB. In: Automatisierte Bewertung in der Programmierausbildung. Oliver J. Bott and Peter Fricke and Uta Priss and Michael Striewe, 2017. ISBN: 978-3-8309-3606-0.

[If15] Iffländer, Lukas; Dallmann, Alexander; Beck, Philip-Daniel; Ifland, Marianus: PABS - a Programming Assignment Feedback System. 2. Workshop Automatische Bewertung von Programmieraufgaben, 2015.

[St17] Striewe, Michael: Der Grader JACK. In: Automatisierte Bewertung in der Programmierausbildung. Oliver J. Bott and Peter Fricke and Uta Priss and Michael Striewe, 2017. ISBN: 978-3-8309-3606-0.