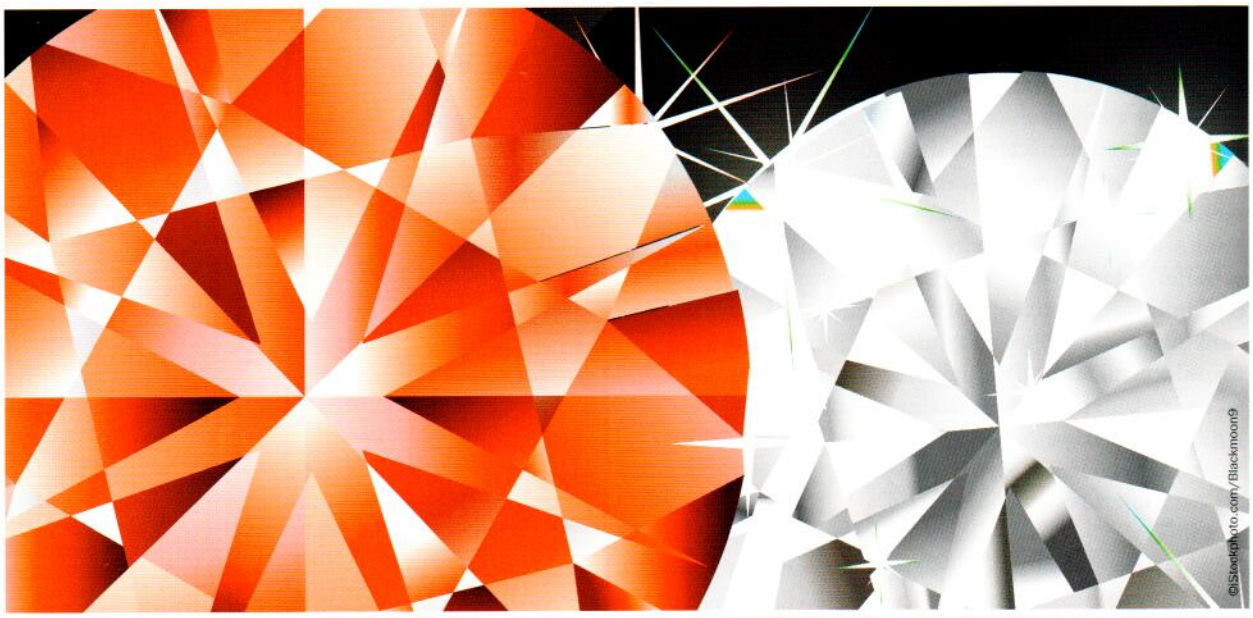

Technologietransfer leicht gemacht

Jubula goes JavaFX

Pünktlich mit Java 8 hat Oracle im Frühjahr dieses Jahres eine grundlegend überarbeitete Version der Benutzeroberflächentechnologie JavaFX veröffentlicht. Dieses auf multimediale Inhalte und Effekte optimierte UI-Toolkit gilt bereits seit einiger Zeit als Nachfolger von Swing und erlaubt es, mit Leichtigkeit moderne, reaktive und ansprechende Benutzeroberflächen zu gestalten. Für uns, das Jubula-Team [1], bedeutete das: neues Jahr – neues Toolkit! Standen für uns im letzten Jahr mobile Plattformen wie iOS im Fokus, so war es im „Luna"-Jahr die einfache und zuverlässige Anbindung zur Fernsteuerung und Testautomatisierung von JavaFX-Oberflächen. Wir geben einen detaillierten technischen Einblick in die notwendigen Grundlagen zur Anbindung von JavaFX-Applikationen in Jubula und zeigen die notwendigen Schritte des Testtechnologie- und Know-how-Transfers von bekannten Jubula-Toolkits wie Swing und SWT zu JavaFX.

von Marcel Hein und Markus Tiede

Der erste Teil richtet sich vor allem an technisch versierte Jubula-Anwender und Entwickler, die verstehen wollen, wie JavaFX „unter der Haube" funktioniert und welche technologischen Hürden genommen werden mussten, um dieses Toolkit unterstützen zu können. Denn aus der Sicht eines Testtools ist es bei der Unterstützung einer neuen UI-Technologie zwingend erforderlich, die Konzepte und technischen Details zum Aufbau des Komponentengraphen, der Eventverarbeitung und des Applikationslebenzyklus zu verstehen. Die nachfolgenden Zeilen sollen daher einen Eindruck davon vermitteln, wie das neue JavaFX-Toolkit „tickt", und dabei auch, soweit das im Rahmen dieses Artikels möglich ist, eine kleine Einführung in JavaFX geben.

Einleitend sollen daher einige technologische Grundlagen und Begriffe aus der JavaFX-Welt geklärt wer-

den. Aber keine Angst: Dies umfasst nur das Nötigste und ist stark zusammengefasst. Wer neu mit JavaFX anfängt, dem können wir – noch vor der Anschaffung eines Buchs – die exzellenten offiziellen Tutorials [1] von Oracle empfehlen. Wem die Interna von JavaFX schon bekannt sind oder wer sich mit diesen nicht auseinandersetzen möchte, kann gleich zum zweiten Abschnitt „Fallbeispiel: Migration von Tests von Swing nach JavaFX" übergehen.

JavaFX Interna: der Scene Graph und das Event Model

Spricht man vom „Scene Graph", so meint man die hierarchische Struktur, in der die Komponenten einer JavaFX-Anwendung organisiert sind. Dies ist ein inhomogener, gerichteter Baum. Es gibt dabei pro Fenster einer Anwendung einen Scene Graph. Auch Pop-ups wie Kontextmenüs, das Auswahlmenü einer Combo Box oder Choice Box oder das Menü einer MenuBar sind Fenster mit einem eigenen Scene Graph.

Eine einzelne UI-Komponente in diesem Graph wird als *Node* bezeichnet. Das „normale" Fenster, in dem Nodes zu finden sind, ist die *Stage*. Das nächste Element im Scene Graph ist die Scene. Sie referenziert den so genannten *Root Node*, unter dem alle weiteren Nodes liegen.

Im Hinblick auf eine Testautomatisierung mit Jubula war es wichtig, diesen Graph deterministisch traversieren und mit heuristischen Algorithmen bewerten zu können, um zum Testausführungszeitpunkt die anzusprechenden UI-Komponenten zuverlässig wiederfinden zu können.

Wie jede Technologie, die eine Schnittstelle zwischen Nutzer und Anwendung bildet, benötigt auch JavaFX neben dem Komponentenmodell eine Möglichkeit, auf Benutzereingaben reagieren zu können: das so genannte Event System. Das Abarbeiten eines Events besteht dabei aus vier Phasen:

1. Ziel auswählen
2. Route erstellen
3. Event Capturing
4. Event Bubbling

Das intern in JavaFX genutzte Glass Windowing Toolkit, das hier nicht näher beschrieben wird, leitet ein Event aus der nativen Event Queue des Systems an den Scene Graph des Fensters weiter, auf dem das Event aufgetreten ist. Danach wird der Knoten ermittelt, der Ziel dieses Events ist.

Das ausgewählte Ziel erzeugt nun in der nächsten Phase die so genannte *Event Dispatch Chain*. Dies ist eine Liste mit allen Nodes von der Wurzel des Scene Graph bis zum Ziel des Events. Sie enthält somit auch das Fenster, z. B. eine Stage, sowie die Scene. Der Zweck dieser Liste ist die geordnete Abarbeitung/Reaktion auf das Event in den folgenden Phasen (**Abb. 1**). In der Event-Capturing-Phase wird, beginnend bei der Wurzel des Scene Graph, von jedem Element in der

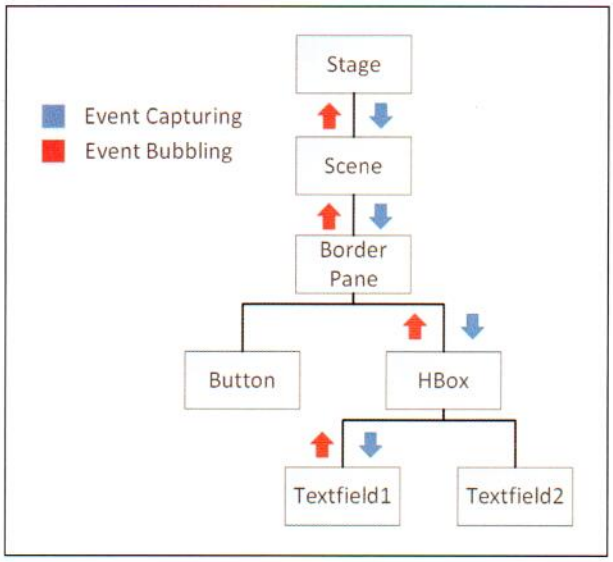

Abb. 1: Ein Beispiel für die Event Propagation in einem Scene Graph

Event Dispatch Chain das Event-Objekt an angemeldete Event-Filter weitergereicht. Sofern diese für den auftretenden Typ des Events oder für einen übergeordneten Typ angemeldet sind. An einem Element können mehrere Event-Filter angemeldet sein, die dann entsprechend ihrem Typ nacheinander aufgerufen werden. Dabei gilt die Reihenfolge: vom spezielleren zum allgemeineren Typ. Die Reihenfolge des Weiterreichens an Event-Filter, die für den gleichen Typ angemeldet sind, ist nicht spezifiziert.

Ein solcher Event-Filter kann das Event konsumieren, wodurch alle nachfolgenden Elemente der Chain dieses Event nicht bekommen würden. Wurde das Event nicht konsumiert und an alle Event-Filter eines Elements weitergereicht, ist der nächste Knoten in der Event Dispatch Chain an der Reihe.

Die letzte Phase beginnt, wenn das Event den letzten Event-Filter des letzten Elements in der Event Dispatch Chain passiert hat, also das Ziel des Events. Nun wird die Liste in umgekehrter Reihenfolge abgearbeitet. Auf diesem Weg werden angemeldete Event Handler aufgerufen. Hier gilt ebenfalls die für die Event-Filter beschriebene Reihenfolge.

Mit Blick auf Jubula war es wichtig, auf dieser Ebene des Event Models eine genaue Synchronisierung an und Differenzierung von Events zu gewährleisten, um eine stabile und zuverlässige Automatisierung von UI-Interaktionen zu ermöglichen, die auch auf verschiedenen Betriebssystemen und unter diversen Lastsituationen einwandfrei funktioniert. Denn kaum etwas wäre kritischer bei einer Automatisierung, als wenn diese nicht reproduzierbar gleichbleibende Ergebnisse liefern würde.

JavaFX Interna: der Startprozess einer JavaFX-Anwendung

Wir bleiben bei der Zahl 4, denn auch der Startvorgang einer JavaFX-Applikation besteht aus vier Phasen. Hierfür müssen zunächst jedoch zwei weitere Begriffe erläutert werden.

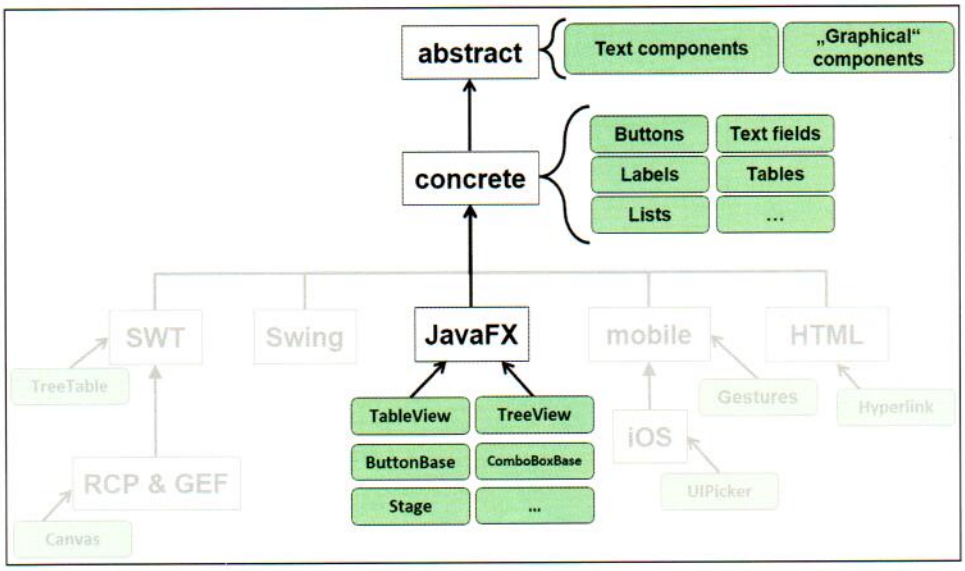

Abb. 2: Die JavaFX-Toolkithierarchie in Jubula

Preloader: Der Preloader besteht, genauso wie die Anwendung, aus einer Stage und dem damit verbundenen Scene Graph. Ein Preloader kann genutzt werden, um dem Nutzer ein visuelles Feedback zu geben, während für die Anwendung nötige Ressourcen geladen werden. Doch es ist auch möglich, bereits im Preloader Eingaben vom Nutzer zu erwarten, damit sich z. B. ein Anwender vor der Verwendung der Applikation einloggen muss. Ist kein eigener Preloader definiert, wird standardmäßig im Hintergrund ein nicht sichtbarer Preloader verwendet.

Application Class: Als Einstiegspunkt in die Anwendung dient eine Klasse, die „Application" des JavaFX-API erweitert, im Folgenden „Application Class" genannt. Warum diese Klasse den Einstiegspunkt der Anwendung darstellt, wird in Phase 4 beschrieben. Nun zu den vier Phasen des Startvorgangs:

1. Initialisierung: Es wird überprüft, ob die Java-Laufzeitumgebung in der entsprechenden Version vorhanden ist; für JavaFX 8 ist dies Java 8. Danach werden die benötigten Klassen geladen, und die *Main*-Methode der *Main*-Klasse wird ausgeführt. Bei dieser Klasse handelt es sich nicht um eine vom Entwickler erstellte Klasse, sondern um eine in der JRE enthaltene Hilfsklasse, die dafür sorgt, dass die Anwendung und der Preloader in der richtigen Reihenfolge gestartet werden.
2. Laden und Vorbereitung : In dieser Phase wird der eingangs beschriebene Preloader angezeigt.
3. Anwendungsspezifische Initialisierung: Der Preloader ist nun beendet worden, und die Anwendung startet. Nun können weitere, vom Entwickler definierte Initialisierungmaßnahmen durchgeführt werden. Dafür muss die Methode *init()* in der Application Class überschrieben werden.
4. Die Anwendung wird ausgeführt: Nun wird die Methode *start(Stage)* der Application Class aufgerufen. Die übergebene Stage wird von JavaFX erzeugt und als „Primary Stage" bezeichnet. Diese kann nun von der Anwendung dazu genutzt werden, durch Hinzufügen entsprechender Nodes das UI zu erzeugen.

Durch Jubula gestartete JavaFX-Anwendungen durchlaufen jede dieser Phasen und müssen, um als AUT („Application under Test") – also als zu testende Anwendung – verwendet werden zu können, in keiner Weise angepasst, erweitert oder modifiziert werden. Out of the box unterstützt Jubula die Testautomatisierung in all diesen Initialisierungsphasen einer Applikation.

Fallbeispiel: Migration von Tests von Swing zu JavaFX

Das JavaFX Toolkit: Jubula- und GUIdancer-Nutzer, die bereits Erfahrung mit den anderen UI-Toolkits wie Swing oder SWT haben, werden diese auch problemlos für das neue JavaFX-Toolkit nutzen können. Um dies zu zeigen, soll ein bestehendes (zugegeben imaginäres)

Der Swing-/JavaFX-AUT-Konfigurationsunterschied – die Gründe im Detail

Dieser Unterschied, vom JAR als Executable und der JRE als Executable, ist bedingt durch den Einsatz eines Java-Agents, der genutzt wird, um eine JavaFX AUT so zu starten, dass Jubula Zugriff auf die UI-Komponenten erhält. Für Swing-Anwendungen wurde mitunter einfach die Manifest-Datei ausgelesen, um die *Main*-Klasse zu ermitteln, damit diese anschließend gestartet werden kann. Dies ist in JavaFX nicht so einfach möglich. Wie einleitend beschrieben wurde, gestaltet sich der Startvorgang einer JavaFX-Anwendung etwas anders. Es wäre zwar möglich, mithilfe des Reflection-API die *start(Stage)*-Methode aufzurufen, allerdings würden dann der Preloader und mögliche Initialisierungsschritte übersprungen. Um also den normalen Startprozess beizubehalten, musste eine andere Möglichkeit gefunden werden, um JavaFX AUTs zu starten. Daher wird jetzt ein mit Java 1.5 eingeführtes Feature verwendet: der Java Agent. Über den Parameter *-javaagent*, der an die Java-Executable übergeben wird, wird ein Pfad zu einem JAR angegeben. Im Manifest dieses JARs ist eine *Pre-Main*-Klasse mit einer *Pre-Main*-Methode angegeben. Diese Methode wird, wie der Name schon sagt, vor der *Main*-Methode der eigentlichen Anwendung ausgeführt. Die Aufgabe dieser Methode ist das Anmelden eines Listeners an einer Stelle, die es uns erlaubt, auf die Scene Graphs einer AUT zuzugreifen. Dies ist nötig, damit die verschiedenen Graphen in einer hierarchischen Struktur abgebildet werden können, die das Finden von Komponenten der AUT durch eine Heuristik ermöglicht. Auf diese Hierarchie bauen die folgenden Schritte auf.

Swing-Projekt in ein JavaFX-Projekt überführt werden. Damit dies verständlich bleibt, ist die Anwendung, die als Beispiel dient, wie der Name schon sagt, sehr simpel: unser berühmt-berüchtigter *SimpleAdder*. Den SimpleAdder gibt es bisher unter anderem mit einer Swing-Oberfläche. Er ist fester Bestandteil unserer mitgelieferten Beispielapplikationen und besteht lediglich aus zwei Textfeldern, die nach der Eingabe von zwei Zahlen die Addition derselben ermöglicht und das Ergebnis zur Anzeige bringt.

Die Tests, die für solche Standardkomponenten (wie Buttons, Labels, Tabellen, Listen, Combo Components, Textfelder, Fenster, usw.) erstellt wurden, können 1:1 auch für eine Applikation in JavaFX verwendet werden. Ermöglicht wird das durch die schon seit Langem vorhandene Abstraktions- und Vererbungshierarchie der Toolkits (**Abb. 2**), in die sich das JavaFX Toolkit nahtlos einfügt. Der Transfer in eine neue Toolkit-Technologie wie JavaFX ist für alle Tests, die auf dem Toolkitlevel *concrete* modelliert wurden, ohne jede weitere Anpassung möglich.

Schließlich wurden sämtliche Konzepte von UI-Komponenten von uns schon für andere Toolkits (wie Swing, SWT, HTML, iOS, .NET, GEF) auf eben dieses *concrete*-Toolkit abstrahiert. Zum Ausführungszeitpunkt eines solchen *concrete*-Tests wird lediglich ein angepasster, JavaFX-spezifischer Treiber verwendet, der sämtliche Toolkitdetails wegkapselt. Das Ansteuern von JavaFX-spezifischen Komponenten und Aktionen ist jedoch ebenfalls, sofern nötig, problemlos möglich.

Das Starten einer JavaFX AUT

Damit ein solcher *concrete*-Test auf einer JavaFX AUT zur Ausführung gebracht werden kann, müssen jedoch zwei Schritte manuell in der Jubula-ITE (Integrated Testing Environment) durchgeführt werden: Die Applikation muss aus Jubula heraus gestartet und das toolkitspezifische Komponentenmapping angefertigt werden. Daher ist der nächste Schritt das Anlegen einer AUT-Konfiguration innerhalb eines JavaFX-Projekts (**Abb. 3**). Darüber kann dann der SimpleAdder für das Object Mapping und für die anschließende Testausführung gestartet werden. Doch im Vergleich zu Swing sieht für JavaFX die Konfiguration leicht anders aus, wie in **Abbildung 2** zu sehen ist. Als Executable wird nicht etwa das JAR des SimpleAdders angegeben, sondern die Java-Executable einer installierten JRE. Über den Parameter wird dann das auszuführende JAR der AUT selbst angeben. Abgesehen davon ändert sich aber für den Anwender beim Starten von JavaFX AUTs nichts.

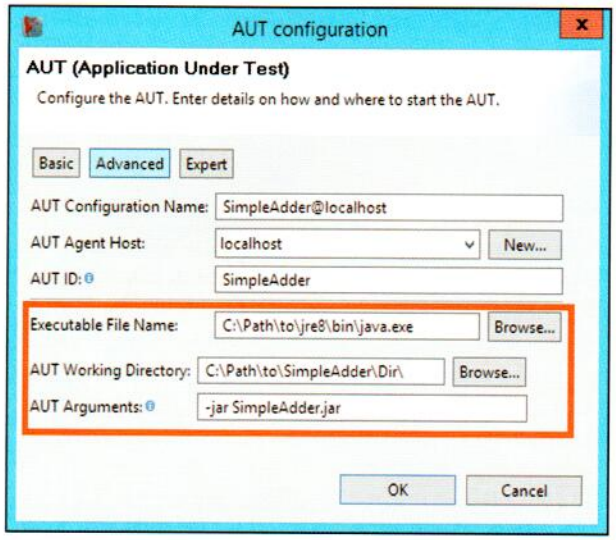

Abb. 3: AUT-Konfiguration einer JavaFX-Applikation

Das Object Mapping

Sobald die Anwendung läuft, kann mit dem „Object Mapping" begonnen werden. Es dient dazu, die realen Komponenten einer AUT mit den in der Testspezifikation verwendeten logischen Platzhaltern zu verknüpfen. Es funktioniert für JavaFX-Anwendungen genauso wie für die anderen UI-Toolkits. Doch damit dieser Vorgang so intuitiv bleibt, mussten wir uns mit der „Haut" von JavaFX-Komponenten beschäftigen und entscheiden, was „gemapped" werden kann und was nicht. Dieser Skin eines Node definiert, vereinfacht gesagt, das Aussehen einer Komponente. Das geschieht, indem der Skin weitere Nodes dem Scene Graph hinzufügt. Ein Beispiel ist der Text auf einem Button. Dieser Text ist auch selbst ein Node, der im Scene Graph gefunden wird und somit beim Object Mapping eingesammelt werden könnte. Wenn es aber möglich ist, einzelne Bestandteile einer Komponente zu „mappen", wird es unnötig kompliziert, da mitunter sehr viele Komponenten im Object Mapping ansprechbar werden, die selbst nur Bestandteil

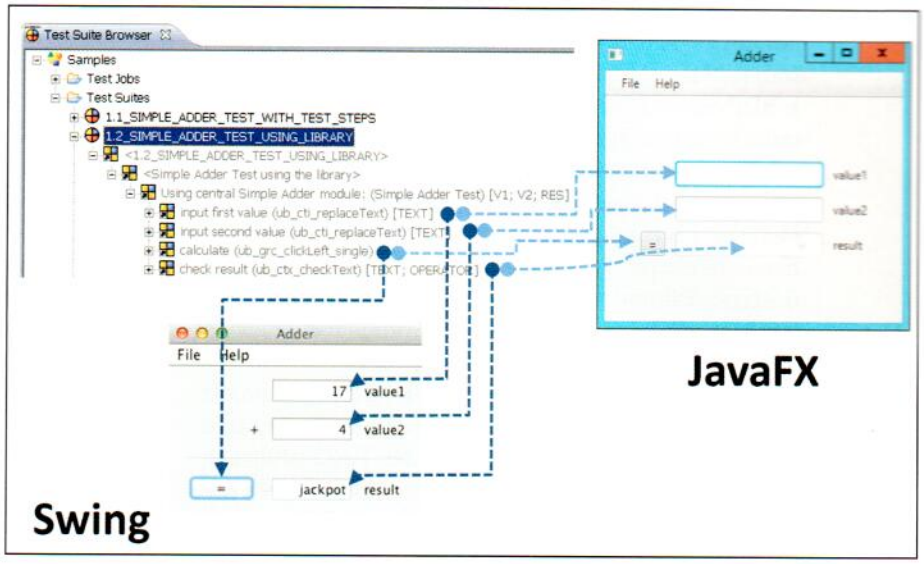

Abb. 4: Analogie des Object Mappings in Swing und JavaFX

Object Mapping funktioniert in JavaFX genauso wie für die anderen UI-Toolkits.

einer semantisch höherwertigen Komponente sind. Wir haben uns daher entschlossen, den Skin einer Komponente nicht im Object Mapping ansprechen zu können. Stattdessen lassen sich lediglich, ohne dabei jegliches technisches Know-how über die Existenz eines solchen Skins zu besitzen, die zugrunde liegenden High-Level-Komponenten einer JavaFX-Applikation einsammeln.

Die Testausführung

Ist das Object Mapping abgeschlossen, können exakt dieselben Tests, die zuvor auf dem Swing SimpleAdder ausgeführt wurden, auch die JavaFX-Variante testen. Auch die Ausführung unterscheidet sich nicht von den anderen Toolkits. Sie besteht im Wesentlichen aus zwei Teilen, dem Auffinden der mit der Testaktion verknüpften UI-Komponente durch eine Komponentenheuristik und dem anschließenden Ausführen der Aktion durch einen fernsteuernden Robot. Die Heuristik findet Komponenten anhand der beim Object Mapping ermittelten Informationen. Vereinfacht gesagt, wird dafür die Position im Scene Graph herangezogen – und der Name der Komponente. Dabei lässt sich vom Anwender diese Erkennung signifikant verbessern, wenn den Komponenten in einer JavaFX AUT IDs vergeben werden, damit diese als eindeutiger Name zur Identifikation verwendet werden können. Das ist die Aufgabe des Entwicklers der AUT, denn dieser kann Nodes mit der Methode *setID(String)* eine solche ID geben. Sofern er dies tut, was löblich ist, sollte er allerdings auch dafür sorgen, dass diese möglichst eindeutig ist.

Die in den Tests definierten Testaktionen werden dabei durch dasselbe Robot-API umgesetzt, das auch in Swing verwendet wurde. Wir verwenden den AWT-Robot, da es für den JavaFX (bzw. Glass) Robot noch kein stabiles öffentliches API gibt.

Bleiben wir noch einmal kurz bei den technischen Details. Wenn ein Test ausgeführt wird, ist es mit Blick auf ein Testtool auch wichtig herauszufinden, ob die Aktionen wirklich in der AUT stattgefunden haben. Dafür benutzten wir einen Mechanismus, der ein erwartetes Event mit den tatsächlich in der Anwendung aufgetretenen Events vergleicht. Diese Ereignisbestätigung erfolgt durch das Abfangen von Events und geschieht im JavaFX Toolkit, wie auch in den anderen Toolkits, indem das API der entsprechenden UI-Technologie verwendet wird – für den Anwender komplett transparent und automatisch im Hintergrund der Testausführung. Im Fall von JavaFX melden wir dazu einen Event Filter an den Anfang des Scene Graphs an, oder an mehrere Wurzeln, sofern es mehrere Fenster gibt. Somit ist die Wahrscheinlichkeit gering, dass das Event bereits konsumiert wurde, bevor es den Event Filter erreicht. Erst, wenn bestätigt wurde, dass das entsprechende Event aufgetreten ist, wird die Testausführung fortgesetzt. Das Aufsammeln und Verarbeiten von Testergebnissen passiert dann auf exakt dieselbe Weise wie für andere UI-Toolkits.

Fazit

Zusammenfassend lässt sich also sagen, dass sich bestehende Tests, sofern sie auf dem „concrete"-Toolkit fußen, mit nur minimalem Aufwand auf eine JavaFX-Applikation übertragen lassen. Lediglich die Startparameter und die Objektzuordnung müssen einmalig erneut vorgenommen werden.

Wer jetzt loslegen und seine ersten Schritte in seiner JavaFX-Applikation automatisieren möchte: Die Unterstützung von JavaFX in Jubula ist unser Beitrag zum diesjährigen, gerade erschienen Luna-Release. Informationen zu Art und Umfang der bereits unterstützten Komponenten sowie bekannte Probleme sind in Abhängigkeit zum Bugzilla-Eintrag unter [3] zu finden. Wir hoffen, einen detaillierten technologischen Einblick in JavaFX und Jubula geben zu können und freuen uns auf euer Feedback.

Markus Tiede arbeitet als Softwareentwickler und Testberater bei der BREDEX GmbH mit den Schwerpunkten Eclipse-RCP-Entwicklung und Entwurf und Entwicklung automatischer Tests und gehört zum GUIdancer-Entwicklungsteam. Markus ist darüber hinaus Eclipse-Committer im UI-Testautomatisierungsprojekt Jubula, Package-Maintainer für „Eclipse for Testers" und hat einen Abschluss als Diplom-Informatiker von der FH Braunschweig-Wolfenbüttel.

Marcel Hein arbeitet seit 2013 bei der BREDEX GmbH. Als Teil seiner Bachelor Arbeit entwickelte er das JavaFX-Toolkit und war damit Vorreiter im Jubula-/GUIdancer-Projekt in Sachen JavaFX. Nach dem erfolgreichen Bachelor-Abschluss arbeitet er jetzt weiterhin als studentischer Mitarbeiter bei BREDEX, während er an der TU Braunschweig seinen Master macht.

Links & Literatur

[1] http://www.eclipse.org/jubula

[2] http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

[3] http://eclip.se/421595