Java aktuell

iJUG Verbund
www.ijug.eu

Anwendungen schneller machen
Java-Performance-Analyse mit YourKit

Der neue Trend
Domain-driven Design Patterns mit Java EE 8 / Jakarta EE

Tipps vom Experten
Hilfe, ich muss JavaScript programmieren!

Ausgabe 05/2018
Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 05


Early Bird
bis zum
28. Sep.
2018
DOAG
Konferenz + Ausstellung
20. - 23. November
in Nürnberg
2018.doag.org
PROGRAMM
ONLINE
mit rund 450 Vorträgen
Eventpartner:
AOUG
SOUG
swiss oracle user group
iJUG
Verbund
ORACLE

JavaFX mit MVVM-Pattern, Usability und Gestensteuerung für Leitstände

M.Sc. Mark Gebler, B.Sc. Hannes Walz und Prof. Dr. Gudrun Görlitz, Beuth Hochschule für Technik Berlin, Fachbereich Informatik und Medien

Der Beitrag führt in JavaFX ein, zeigt das Konzept von MVVM in Verbindung mit mvvmFX am realen Beispiel aus der Logistik und gibt einen Ausblick auf gestengesteuerte und nutzerfreundliche JavaFX-Oberflächen mit Touch-Bedienung und Anbindung der Eingabegeräte Leap Motion und der Thalmic Myo.

Die Beuth Hochschule forscht in einem Kooperationsprojekt *(siehe „https://projekt.beuth-hochschule.de/kopgeo")* zu einem universell nutzbaren Gesten- und Interaktionskonzept für Leitstände mit einem situativ geführten, berührungssensitiven, aber auch berührungslosen Bediensystem. Das Ziel besteht in der Entwicklung eines neuartigen Softwaresystems (Dispositionssystems), das die heutigen Anforderungen an die Steuerung und das Monitoring von und in Leitständen erfüllt.

Die Kernaufgabe der Disposition in der Entsorgungslogistik ist die effiziente Routenplanung der verschiedenartigen Entsorgungsfahrzeuge. Der Kooperationspartner „Gesellschaft für Informationssysteme und Prozessautomatisation mbH" (GIPA), der Software für die Entsorgungslogistik entwickelt und vertreibt, befasst sich im Kooperationsprojekt mit der Entwicklung eines Frameworks zur Anbindung von gestengesteuerten User-Interfaces für komplexe verteilte Prozesse auf Basis von Microservices. Komplexe Applikationen sollen sich dabei aus einfachen Bestandteilen flexibel zusammensetzen. Es kommen Straßenkarten-Darstellungen und GPS-verortete Objekte zum Einsatz.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.StackPane?>
<StackPane xmlns="http://javafx.com/javafx"
           xmlns:fx="http://javafx.com/fxml"
           fx:controller="example2.HelloFXMLController"
           prefHeight="200.0" prefWidth="200.0">

    <Label fx:id="myLabel" text="Hello JavaFX again" />

</StackPane>
```

Listing 1

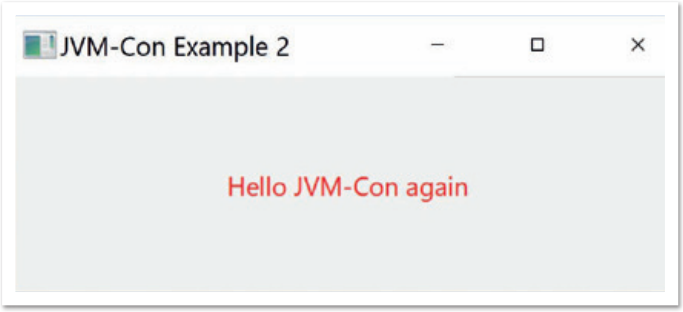

Abbildung 1: Beispiel einer Anwendungsoberfläche mit einer FXML-Datei und Controller

Im Enterprise-Bereich ist Java eine weitverbreitete Programmiersprache im Backend-Bereich. Es ist daher naheliegend, Java mit modernen Konzepten und Design-Pattern auch im Frontend einzusetzen, um unter anderem das Unternehmenspersonal in einer gewohnten Programmiersprache arbeiten zu lassen. Nicht zuletzt ist auch die plattformübergreifende Ausführung aller JavaFX-Komponenten ein fast unschlagbares Argument, wenn es um die zahlreichen Systeme und Systemkomponenten geht, die ein Fahrer im Fahrzeug bedienen muss.

JavaFX

JavaFX ist ein Framework zur Erstellung von Benutzeroberflächen in Java. Es war die Oracle-Antwort auf die im Jahr 2000 aufkommenden neuen UI-Technologien wie Silverlight von Microsoft und Flex von Adobe. Wie die Mehrzahl der aktuellen UI-Frameworks erlaubt es, Benutzeroberflächen aus einer Markup-Repräsentation aufzubauen und nur die dahinterliegende Funktionalität als Code in einer Programmiersprache zu entwickeln. Diese XML-Beschreibung ist grundsätzlich vergleichbar mit XML-Views, wie man sie aus Android kennt, oder mit Microsoft XAML. In einem einfachen Beispiel soll der grundsätzliche Aufbau einer JavaFX-Anwendung aus Application-Starter, FXML-Beschreibung und View Controller gezeigt werden *(siehe Listings 1 bis 3* und *Abbildung 1)*.

In der JavaFX-Welt entspricht ein Fenster einer Stage. Die initiale Stage bekommt der Application-Starter in seine „start"-Methode übergeben. Auf dieser Stage kann die Anwendung einen Szenengraph anlegen. Die Szenen in einem Fenster lassen sich über die „setScene"-Methode austauschen oder initial setzen. Einer Szene werden dann die UI-Elemente, die auch Nodes genannt werden, hinzugefügt. Da eine Szene nur ein Root-Element haben kann, muss es sich dabei um einen Container handeln, der die Unterelemente in einem Layout anordnet.

In unserem Beispiel laden wir die UI-Elemente – bestehend aus dem Root-Element, der „StackPane", und dem Label – aus unserer

```
public class HelloFXMLApplication extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        // UI-Graphen aus FXML-Datei laden
        Parent root = FXMLLoader.load(getClass().getResource("hello_javafx.fxml"));

        Scene scene = new Scene(root, 200, 200);

        primaryStage.setTitle("JavaFX Example 2");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Listing 2

```
public class HelloFXMLController {
    @FXML
    private Label myLabel;

    public void initialize() {
        // Die Methode wird vom FXMLLoader aufgerufen, wenn die Elemente bereit stehen (s. fx:controller)
        myLabel.setTextFill(Color.RED);
    }
}
```

Listing 3

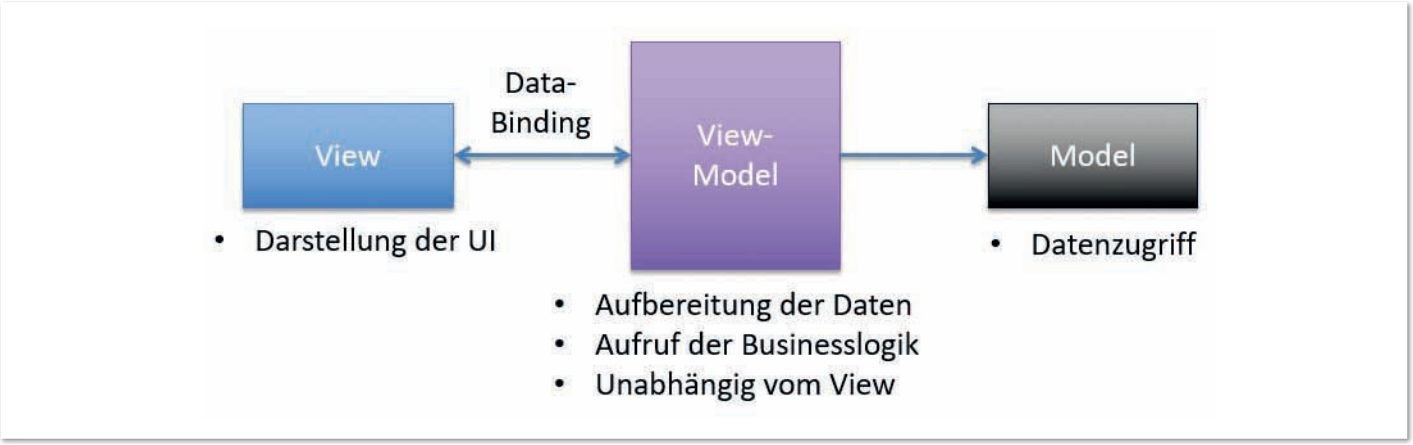

Abbildung 2: FXML-Dateien für eine UI-Beschreibung

FXML-Datei. Die Szene erstellen wir jedoch im Application-Starter und fügen ihr als Root-Element die geladene „StackPane" hinzu. Schlussendlich wird die Szene der übergebenen „primaryStage" übergeben.

Über das Attribut „fx:controller" wurde der StackPane in der FXML-Datei ein Controller zugewiesen. Controller kontrollieren einen Teil der UI und steuern die Interaktion damit. Wird einem Node über das „fx:id"-Attribut ein Identifier zugewiesen, lässt sich eine Referenz darauf über die „@FXML"-Annotation in der Controllerklasse herstellen. Das kennt man ähnlich aus der XAML-Welt.

MVVM

Bei MVVM handelt es sich um ein Entwurfsmuster. Die Buchstaben dieser Abkürzung stehen für „Model", „View" und „ViewModel"; es handelt sich dabei strenggenommen um eine Variante des MVC-Patterns *(siehe Abbildung 2)*.

Die Model-Schicht übernimmt bei MVVM die gleiche Aufgabe, die sie auch bei MVC innehat. Sie repräsentiert die Daten und ermöglicht gegebenenfalls den Datenzugriff in eine Storage-Komponente. Die View ist für die Darstellung der UI verantwortlich. Das ViewModel hingegen ist von der View unabhängig und steht zwischen View und Model. Es kann so entwickelt sein, dass es für mehrere Views verwendet werden kann – etwa für eine Mobile- und eine Desktop-View.

Das ViewModel ist für die Aufbereitung der Daten für eine oder mehrere Views zuständig und hat auch die Aufgabe, die Geschäftslogik anzusprechen. Anders als bei früheren MVC-Frameworks erfolgt die Bindung zwischen der View und der ViewModel-Ebene (die in vielen Teilen der Controller-Schicht gleicht) über sogenanntes „Data-Binding". Dabei werden Daten aus der View (etwa der Text in einem Textfeld) durch Properties in den ViewModels repräsentiert, die sich automatisch ändern, wenn sich die Daten in der View ändern, und umgekehrt. Auf dieselbe Art und Weise können auch Commands an UI-Elemente – etwa Buttons – gebunden werden, um Aktionen auszulösen.

MVVM in JavaFX und mvvmFX

Wie schon erwähnt, ist vieles, was Teil von MVVM ist, schon out of the box in JavaFX enthalten. So verfügen alle UI-Elemente über Properties, an die sich gebunden werden kann. Das Code-Beispiel in

```
Label copyLabel = new Label();
TextField textField = new TextField("Hallo JavaFX");

copyLabel.textProperty().bind(textField.textProperty());

root.getChildren().addAll(copyLabel, textField);
```

Listing 4

Listing 4 illustriert dies anhand eines Labels, das immer den Text, der in einem Texteingabefeld steht, anzeigt.

Hier werden per Code zwei neue Nodes erstellt: ein Label mit dem Namen „copyLabel", das den Text des Text-Eingabefelds („TextField") mit dem Namen „textField" darstellen soll. Durch den Aufruf der Methode „bind(...)" auf der „textProperty" des Labels und die Übergabe der „textProperty" des Textfelds werden beide Properties aneinandergebunden. Das bedeutet, dass sich die eine Property automatisch ändert, wenn sich die andere ändert. Dieses Binding ist unidirektional. Das heißt in diesem Fall, dass sich nur die „textProperty" des Labels ändert, wenn im Textfeld Text eingeben wurde, nicht aber umgekehrt. Sollen auch Änderungen von der „textProperty" an das Textfeld weitergereicht werden, ist die Methode „bindBidirectional(...)" erforderlich.

MvvmFX kann über Maven einfach in existierende oder neue Projekte eingebunden werden und hat fünf entscheidende Vorteile im Vergleich zu reiner JavaFX-Entwicklung: Es bietet eine implizite Verbindung von FXML-Markup-Dateien zur UI-Beschreibung und zugehörigen View-Klassen sowie eine explizit anzugebende Verbindung zwischen View- und ViewModel-Klassen. Hinzu kommt der Einsatz von Dependency Injection zur Entkopplung der Anwendungs-Abhängigkeiten. Außerdem ermöglicht es die Kapselung von Event-Handling in Commands und Scopes. Am einfachsten lassen sich diese Prinzipien anhand eines praktischen Beispiels erklären. Es soll ein Fenster zur Bearbeitung von Personendaten dargestellt werden, das ein Textfeld für Vor- und Nachname und einen Button zum Speichern enthält *(siehe Abbildung 3)*.

In der FXML-Datei gibt es im Vergleich zu dem, was wir bisher gesehen haben, keine Änderungen. Allein die explizite Angabe einer Action-Methode zum Click-Handling des Buttons über das „onAction"-Attribut wurde bisher nicht eingeführt, ist allerdings Bestandteil von JavaFX und nicht von mvvmFX *(siehe Listing 5)*.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.control.Button?>

<VBox xmlns="http://javafx.com/javafx"
      xmlns:fx="http://javafx.com/fxml"
      fx:controller="example5.PersonView"
      prefHeight="200.0" prefWidth="200.0">

    <TextField fx:id="firstNameField" />
    <TextField fx:id="lastNameField" />

    <Button fx:id="saveButton" text="Speichern"
onAction="#onSave"/>

</VBox>
```

Listing 5

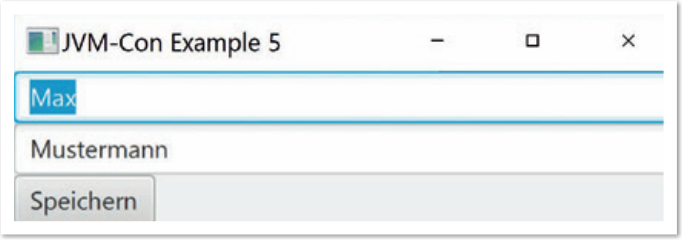

Abbildung 3: mvvmFX am Beispiel

Anders sieht es im Application-Starter aus, der hier für das Laden unserer View verantwortlich sein soll. Der FluentViewLoader aus mvvmFX lädt ein ViewTuple. Es besteht aus einer View – also dem Baum der UI-Elemente, hier mit einer VBox als Root-Element –, dem CodeBehind – hier einer Instanz der PersonView-Klasse – und einer Instanz der zur View gehörigen ViewModel-Klasse. Die Referenz auf die View wird dann, wie gewohnt, an die Szene weitergegeben und das Fenster wird aufgebaut. Man beachte, dass die FXML-Datei nirgends über den Dateinamen referenziert wird. Es gibt eine implizite Verbindung zwischen der im „res/"-Verzeichnis oder im selben Verzeichnis liegenden FXML-Datei und der View-Klasse *(siehe Listing 6)*. Klassenname und Dateiname der zugehörigen FXML-Datei müssen hier gleich sein.

Ein Blick auf die Controller-Klasse, die hier als View-Klasse fungiert, zeigt weitere Unterschiede. Am auffälligsten ist die Implementierung der Interfaces „FxmlView<T>" und „Initializable"; „T" ist dabei ein ViewModel. Auf diese Art und Weise wird eine View explizit an ein ViewModel gekoppelt. Umgekehrt kann ein ViewModel aber an mehrere Views gekoppelt sein. Das ViewModel wird dann über die Annotation „@InjectViewModel" in die View injiziert.

Das ViewModel stellt die Properties und Commands bereit, die die View benötigt. Namentlich sind das „StringProperties" für Vor- und Nachname („firstnameProperty" und „lastNameProperty"). Diese sind in diesem Fall bidirektional gebunden, um eventuelle Änderungen im Datenmodell von anderer Stelle direkt in der View darzustellen. Die „onSave"-Methode wurde aus der FXML-Datei referenziert und ruft selbst wiederum nur das vom ViewModel bereitgestellte Save-Command auf *(siehe Listing 7)*.

Das ViewModel enthält einen Verweis auf das Model, Properties zum Binden an die View und Methoden, um die Properties mit dem Model synchron zu halten. Zudem bietet es ein Command an, mit dem das Speichern des Fachobjekts ausgelöst wird *(siehe Listing 8)*. Da sich die Implementierung einer Benutzer-Schnittstelle nicht allein auf Detailfragen des eingesetzten Frameworks beschränken lässt, wird nachfolgend auf in diesem Projekt eingesetzte, neuartige Eingabemöglichkeiten eingegangen.

Natural User Interfaces

Die Interaktion über Natural User Interfaces (Touch, Leap Motion und Thalmic Myo) soll zu einer optimierten und angenehmeren Bedienung führen. Die Benutzer sollen situativ durch die Arbeitsaufgabe an den Leitständen geführt werden. Das Selektieren von Objekten auf der Bedienoberfläche wird beispielsweise in 83 Prozent der Durchführungen durch die direkte Touch-Interaktion, gegenüber der Durchführung mit der Maus, verkürzt *[1, Seiten 1 bis 4]*.

Die Touch-Bedienung an einem Natural User Interface ist spätestens durch die Einführung der Smartphones bekannt und unterliegt daher auch bestimmten Vorstellungen zur Interaktion *(siehe Abbildung 4)*. Ein Natural User Interface setzt voraus, dass es möglichst leicht ist, daran zu erinnern, wie bestimmte Funktionen bedient werden *[2, Seite 6]*.

Leap Motion ist ein kamerabasiertes Analysetool, das die Position der Hände und Finger des Nutzers im Raum analysiert *(siehe Abbildung 5, unten Mitte)*. Neben einer Handbewegung oder dem

```
public class PersonApplication extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        ViewTuple<PersonView, PersonViewModel> viewTuple = FluentViewLoader.fxmlView(PersonView.class).load();

        Scene scene = new Scene(viewTuple.getView(), 200, 200);

        primaryStage.setTitle("JavaFX Example 5");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Listing 6

```
public class PersonView implements FxmlView<PersonViewModel>, Initializable {

    @FXML
    private TextField firstNameField;

    @FXML
    private TextField lastNameField;

    @FXML
    private Button saveButton;

    @InjectViewModel
    private PersonViewModel personViewModel;

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        firstNameField.textProperty().bindBidirectional(personViewModel.firstNameProperty());
        lastNameField.textProperty().bindBidirectional(personViewModel.lastNameProperty());
    }

    @FXML
    public void onSave() {
        personViewModel.getSaveCommand().execute();
    }
}
```

Listing 7

```
public class PersonViewModel implements ViewModel {
    private SimpleStringProperty firstNameProperty;
    private SimpleStringProperty lastNameProperty;

    private Person person;

    public PersonViewModel() {
        setPerson(new Person("Max", "Mustermann"));
    }

    public SimpleStringProperty firstNameProperty() {
        return firstNameProperty;
    }


    public SimpleStringProperty lastNameProperty() {
        return lastNameProperty;
    }



    public Command getSaveCommand() {
        return new DelegateCommand(() -> new Action() {
            @Override
            protected void action() throws Exception {
                person.save();
            }
        });
    }

    // ...

    private void setPerson(Person person) {
        this.person = person;
        firstNameProperty = new SimpleStringProperty(person.getFirstName());
        lastNameProperty = new SimpleStringProperty(person.getLastName());
        setupListeners();
    }

    private void setupListeners() {
        firstNameProperty.addListener((observable, oldValue, newValue) -> updatePerson());
        lastNameProperty.addListener((observable, oldValue, newValue) -> updatePerson());
    }

    private void updatePerson() {
        person.setFirstName(firstNameProperty.get());
        person.setLastName(lastNameProperty.get());
    }
}
```

Listing 8

Abbildung 4: Varianten der Gesten-Interaktion über Touch, berührungslose Gestenbedienung mit Leap Motion und Muskelkontraktions-Analyse mit Thalmic Myo

Zusammenbewegen zweier Finger (Pinch-Gesture) bietet die Leap Motion auch die Möglichkeit, die Position der Hand vor, zwischen oder hinter dem Gerät auszuwerten, sodass ein Klicken ohne explizierte Geste möglich ist *(siehe Abbildung 5)*.

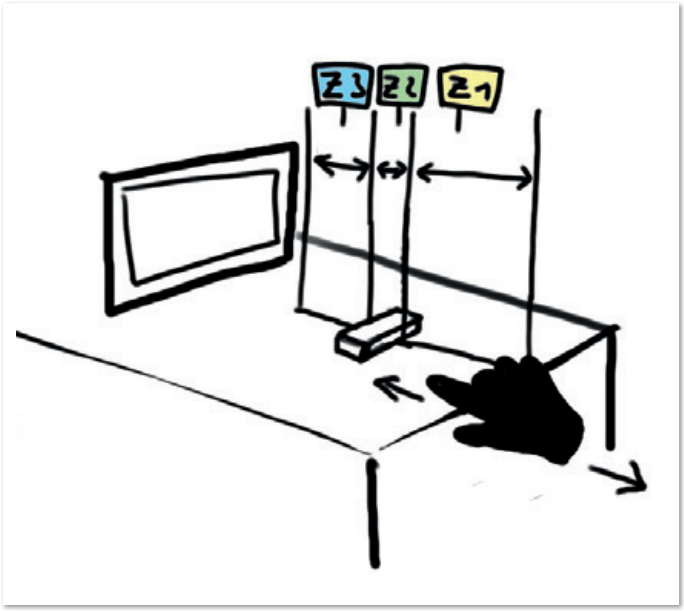

Abbildung 5: Leap Motion – Analyse der Hand (inklusive aller Finger) und die Positionen im Raum über der Leap Motion

Im Codebeispiel in *Listing 9* werden die Position der Handfläche („Palm") über der Leap Motion ausgelesen und unter anderem abgefragt, ob der Zeigefinger als einziger ausgestreckter Finger als Geste auftritt.

Das Myo-Armband hingegen wird am oberen Unterarm angelegt und analysiert die Muskelkontraktion, um eine Handbewegung zu erkennen *(siehe Abbildung 6*, links am Unterarm*)*. Entgegengesetzt zu der auf Basis von Live-Bildanalyse arbeitenden Leap Motion hat das Myo-Armband kein begrenztes Analysefeld (Kamera-Sichtfeld). Das Myo-Armband verwendet dabei das Gyroskop, das die Lagebestimmung durch Rotationskräfte bestimmt und dem „Mouse-Cursor" seine Position gibt.

Mit Paper-Prototypings zur intuitiven Leitstand-Software

Paper-Prototypings haben bei der Konzeptionierung geholfen, die Bedienfolgen zu bestimmen, bevor die MVVM-Komponenten entwickelt werden. Vor einer Auftragszuordnung zu einer Tour beispielsweise können in einem Benutzerdialog geeignete Tourenvorschläge *(siehe Abbildung 6*, Nr. 2 und 3*)* durch speziell entwickelte Services abgerufen und damit zusätzliche Informationen geboten werden. Gleichzeitig sind situativ nur die Informationen angezeigt, die an jener Stelle im Workflow notwendig sind. Bei einigen Interaktionsformen beansprucht die reine Klick-Abfolge mehr Zeit als bei anderen. Die Klicks in Benutzerdialogen sind deshalb auf ein Minimum redu-

Abbildung 6: Workflow im Benutzerdialog „Auftragszuordnungen" (Paperprototyping + Prototyp), 1 – Auftragsobjekt auswählen, 2 – Funktionswahl Tour-Zuordnung, 3 – Tourenauswahl betrachten, 4 – Position in Tour wählen

```
private void detectState(Frame frame) {
    FingerList fingerlist = frame.fingers();
    FingerList indexFingers = fingerlist.fingerType(Finger.Type.TYPE_INDEX);
    Finger indexFinger = indexFingers.get(0);

    Hand hand = frame.hands().frontmost();
    Vector palmPosition = hand.palmPosition();

    POSITIONSTATE positionState;
    int hoverStateArea = 20;
    if (palmPosition.getZ() >= hoverStateArea)
        positionState = POSITIONSTATE.NORMAL;
    else if (palmPosition.getZ() < -hoverStateArea)
        positionState = POSITIONSTATE.ACTIV;
    else
        positionState = POSITIONSTATE.HOVER;

    boolean indexFingerExtended = indexFinger.isExtended();
    int extendedCount = hand.fingers().extended().count();
    boolean handState = frame.hands().count() >= 1;

    Platform.runLater(() -> LeapMotionController.getInstance().updateLeapPointState(positionState,
indexFingerExtended, extendedCount, handState));
}
```

Listing 9

```
root.setOnMouseClicked(event -> {
    root.toFront();
});
```

Listing 10

```
private void fireEvent(EventType<? extends MouseEvent> eventType, Node node, float xValue, float yValue) {
    if(node == null) return;
    MouseEvent m = new MouseEvent(node, node, eventType, xValue, yValue, xValue, yValue,
            MouseButton.PRIMARY, 1, false, false, false, false, true, false, false, false, false, false,
            null);
    Event.fireEvent(node, m);
}
```

Listing 11

```
BeuthGestureEvent event = new BeuthGestureEvent(BeuthGestureEvent.DRAG_A_NODE, xValue, yValue);

notifyObservers(event);
setChanged();
```

Listing 12

ziert. Während der Gestendurchführung wird mit Rückmeldungen an die nutzenden Prozesse die Gestenanalyse visualisiert, was zu einer einfacheren Wahrnehmung führen soll.

JavaFX mit Gesten-Events

Das klassische Event-Handling von JavaFX für Mouse und Touch lässt sich mit Event-Handlern auf bestimmten Komponenten umsetzen *(siehe Listing 10)*. Beim Einsatz von individuellen Gesten-Erkennungstools und -geräten ist es sinnvoll, ein Event-Handling auf Basis simulierter Mouse-Events einzusetzen. Hier kann beispielsweise eine „Grab"-Geste zu einem klassischen Mouse-Event konvertiert werden, das wiederum den klassischen Event-Handler aktiviert. Alle Nodes lassen sich über ihre Position erfassen und mit der Handposition oder dem Mouse-Cursor vergleichen. Per „Event.fireEvent(node, event)" lässt sich das gewünschte Event bei diesem Node simulieren *(siehe Listing 11)*.

Falls dies nicht ausreicht und eine individuelle Geste mit bestimmten Parametern ein Ereignis auslösen soll, wird das Event-Handling mit einem Observer-Design-Pattern und eigenen Events eingesetzt. Ein entsprechendes Beispiel wären Bewegungsgesten, deren Parameter (Geschwindigkeit, Richtung) entscheidend sind und bei denen die Komponenten sich folglich auf Basis der erfassten Parameter animieren. Bei einer erkannten „Drag-Geste" können mehrere Komponenten reagieren sowie das „BeuthGestureEvent" auswerten und verarbeiten *(siehe Listing 12)*.

Fazit

Natural User Interfaces werden in Kombination mit workflowbasierten Benutzerdialogen die Bearbeitung von Aufgaben erleichtern und bieten damit mehr Raum für die inhaltliche Auseinandersetzung mit den Aufgaben. Moderne Design-Patterns wie MVVM bieten die Möglichkeit, hochwertige Komponenten für Unternehmensanwendungen zu entwickeln. Wie schon Swing und AWT ist auch JavaFX ein plattformübergreifendes UI-Framework. Seit Java 7 (JRE 7u6) ist es Teil jeder Java-SE-Installation.

Seit dem Jahr 2011 ist JavaFX über das OpenJFX-Projekt als quelloffene Software verfügbar. Im März 2018 gab Oracle im Rahmen seiner Java Client Road Map bekannt, dass JavaFX ab Java 11 nicht mehr Teil der Java Standard Edition sein wird. Oracle wird noch bis zum Jahr 2022 weiter Support anbieten und unterstützt die Weiterentwicklung im Rahmen des OpenJFX-Projekts.

Hinweis: Dieses Projekt wird im Netzwerk „Assistenz in der Logistik" unter dem Titel „Kopplung verteilter Prozesse an Gestengesteuerte Oberflächen (KoPGeO)" durch das Zentrale Innovationsprogramm Mittelstand (ZIM) gefördert.

Literatur

[1] Kin, Agrawala und DeRose (2009), Determining the benefits of direct-touch, bimanual, and multifinger input on a multitouch workstation, in A. Gooch & M. Tory (Eds.), Graphics Interface 2009, Proceedings, Kelowna, British Columbia, Canada, 25-27 May 2009 (pp. 119–124), Missisauga, Ont., Canadian Information Processing Society

[2] Norman (2010), The way I see it: Natural user interfaces are not natural, interactions, 17(3)

Mark Gebler
mgebler@beuth-hochschule.de

Mark Gebler M.Sc. hat Medieninformatik an der Beuth Hochschule für Technik Berlin studiert und entwickelte in seiner Masterarbeit eine gestengesteuerte Besucher-Informationsanwendung für das Deutsche Technikmuseum Berlin. Im Zentrum der Forschung und der Dissertation steht die Entwicklung von Koordinations- und Leitsystemen.

Hannes Walz
hwalz@beuth-hochschule.de

Hannes Walz B.Sc. hat Informatik an der Berufsakademie Berlin studiert und entwickelt seit fünf Jahren Java-Anwendungen für verschiedene Plattformen von Mobil bis Desktop. Aktuell entwickelt er auf Basis von JavaFX eine gestengesteuerte MVVM-Anwendung zur Disposition in der Abfallwirtschaft.

Prof. Dr. Gudrun Görlitz
goerlitz@beuth-hochschule.de

Prof. Dr. Gudrun Görlitz ist Professorin an der Beuth Hochschule für Technik Berlin im Fachbereich Informatik und Medien. Sie ist in der Lehre zur Programmierausbildung der Studierenden tätig, schwerpunktmäßig mit der Programmiersprache Java. Sie leitet zahlreiche anwendungsorientierte, drittmittelgeförderte F&E-Projekte, darunter das F&E-Projekt „KoPGeO – Kopplung verteilter Prozesse an gestengesteuerten Oberflächen".


ORAWORLD
Das e-Magazine für alle Oracle-Anwender!
EOUC
EMEA ORACLE USERGROUP COMMUNITY
• Spannende Geschichten aus der Oracle-Welt
• Technologische Hintergrundartikel
• Leben und Arbeiten heute und morgen
• Einblicke in andere User Groups weltweit
• Neues (und Altes) aus der Welt der Nerds
• Comics, Fun Facts und Infografiken
Jetzt Artikel einreichen oder Thema vorschlagen!
Bis 9. August 2018
Jetzt e-Magazine herunterladen
www.oraworld.org

JavaLand

19. - 21. März 2019 in Brühl bei Köln

Ab sofort Ticket & Hotel buchen!

www.javaland.eu

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu