**Protokoll Übung 1**

Gruppe: **Schwabakla**

– Robert Dziuba, Tobias Klatt & Inga Schwarze

1. **Properties implementieren**

Aufgabenteil A:

Wir schreiben eine Klasse Appointment, die folgende Attribute als StringProperties erhält: datum, titel, startzeit, endzeit, kategorie, notiz.

Die Klasse Zeitrechner ist unsere Kontrollklasse, die die Zeiten überprüft.

Neue Objekte von StringProperties werden mit SimpleStringProperty initialisiert.

Properties haben den Vorteil, dass wir im späteren Verlauf in unserem Programm mit diesen arbeiten können und das ermöglicht uns, diese Attribute zu „beobachten“ (--> Observable) und miteinander zu „verbinden“ (--> Binding). Dadurch kann der Wert des Attributes an eine andere Property gebunden werden und wird somit direkt und ständig synchronisiert (bei jeder Änderung).

Wir haben drei Konstruktoren: einen Standartkonstruktor (leer), einen Kopierkonstruktor und einen, der ein Appointment initialisiert und alle Attribute setzt.

Wir haben Getter und Setter, die wir zwingend für diesen ADT brauchen. Wir setzen also unsere Attribute damit fest.

Getter: Setzen

Setter: Holen

Da wir Properties verwenden sehen die Setter etwas anders aus als gewohnt. Es wird ein String entgegengenommen und zu einem StringProperty gemacht.

Außerdem erhalten wir, zusätzlich zu den Gettern mit String als Rückgabetyp, einen neuen Typ von Gettern, die uns StringPropertys zurückgeben.

Das ist später nützlich, da diese Objekte z.B. das Binding enthält.

To String Methode: Um den ADT auszugeben.

In der Konsole erscheint dann eine Übersicht unseres Termineintrags.

Unsere Setter sorgen dafür, dass die eingegebenen Daten überprüft werden. Wir haben dazu folgende Methoden geschrieben.

Datum: testeDatumString() überprüft, ob ein Datum vorliegt und ob dieses im richtigen Format (testeDatumFormat()) ist.

Außerdem überprüfen wir ob unser Datum im Wertebereich liegt (testeDatumWertebereich()).

Analog dazu testen wir auch die Zeit: testeZeitString(), testeZeitFormat(), testeZeitWertebereich().

Aufgabenteil B:

Bei den Settern kontrollieren wir zusätzlich, ob die Endzeit tatsächlich hinter der Startzeit liegt und andersherum (Aufgabenteil B). Die Methoden heißen testeZeitFenster().

Es wird direkt eine Exception geworfen, wenn die Startzeit nicht vor der Endzeit liegt.

1. **Editierbare ListView**

Aufgabenteil A:

Wir schreiben eine neue Klasse ObservableContactDetails, die von unserer Klasse ContactDetails erbt. Wir haben fünf Attribute vom Typ StringProperty: vorname, nachname, adresse, telefonnummer, mail. Die Klasse hat dieselben Eigenschaften wie unsere ContactDetails, bis auf die Tatsache, dass es sich nun, statt Strings, um SimpleStringProperties handelt, damit wir später mit Binding arbeiten können.

Die Klasse besitzt drei Konstruktoren: Standartkonstruktor, ruft unseren Hauptkonstruktor auf. Der normale K. ruft zusätzlich den K. von ContactDetails auf. Der Kopierk. Ruft den normalen K. auf. Die Getter geben die Werte für unsere Attribute, die Setter setzen die Attribute für die Property. Auch hier haben wir wieder die Möglichkeit das Attribut zurückzugeben, als String oder Property.

Außerdem wie üblich eine ToString-Methode und eine Main-Methode zum Ausführen, um einen Überblick über die Funktion der Klasse zu bekommen.

Aufgabenteil B:

Die Klasse AddressBookUI übernimmt für uns den View-Part. Hier werden alle wichtigen Objekte initialisiert, die wir für unsere Ansicht brauchen. Alle Logik wird in die Klasse Control ausgelagert. Mit Hilfe des Control Objektes füllen wir unsere ListView. Wie schon in unseren vorherigen Arbeiten holen wir uns aus dem AddressBook die Kontakte und übergeben sie mit Hilfe der observableArrayList der ListView. Für die Editierung unserer Zellen schreiben wir die Klasse ListCellFactory.

ListCellFactory ist eine Klasse die von ListCell erbt und uns somit die Möglichkeit bietet, in das Verhalten von editierten Feldern einzugreifen.

Speziell die Methoden updateItem(), startEdit() und canceklEdit() sind hier von Interesse. Die ersten beiden Methoden rufen unsere eigene Methode updateViewMode auf. Die letzte Methode wird immer dann ausgeführt, sollte der User das updaten abbrechen. Somit haben wir die Möglichkeit, die geänderten Werte zurückzusetzen.

UpdateViewMode hat mehrere Methoden die uns den derzeitigen Status der Editierung übergeben. Wird diese Methode aufgerufen, wird die ListView-Zelle geleert mit setText(null) und setGraphic(null). Danach wird mit isEditing() geschaut, ob die Zelle im Editier-Modus ist. Sollte sie es nicht sein. Wird nur Nachname und Vorname ausgegeben. Sollte der User aber doppelt geklickt haben, befinden wir uns im Editier-Modus und der User kann mit Hilfe eines Formulars die Daten ändern. Dabei hat der User die Möglichkeit, den Kontakt zu ändern, zu löschen oder alle Änderungen rückgängig zu machen. Nach der Editierung werden alle Änderungen im AddressBook und in der ListView gespeichert.

Aufgabenteil C:

Wir erstellen den Knopf „drucken“ in unserer UI, der an unsere Control weiter gegeben wird. Hier ruft er über ein setOnMouseClicked die toString-Methode unseres AddressBooks auf und gibt alle gespeicherten Kontakte in der Konsole aus. Wichtig: Wir haben in unserer Klasse AddressBook eine ToString-Methode selber geschrieben, die diesen Inhalt übersichtlich und korrekt darstellt. Wir nutzen dafür einen StringBuilder.

Aufgabenteil D:

In unserer Methode speicherKontakt wird unsere Methode changeDetails aufgerufen, die die Änderungen an das AddressBook weitergibt. Da die Felder „gebindet“ sind muss die ListView nicht mehr aktualisiert werden.

1. **Editierbare Tabelle**

Aufgabenteil A:

Idee: Wir trennen unsere Klassen nach dem MVC Prinzip. Das bedeutet, wir haben folgende Klassen:

View: Für die grafische Oberfläche, ruft Control auf

Control: Für die Logik des Programms, füllt die View

Die View-Klasse erzeugt eine BorderPane, TapPane und ein Control-Objekt. Control ruft dann die Methode fuelleTabelle() auf, die in die Control-Klasse ausgelagert wird (--> Logik!). Wir erzeugen außerdem eine Scene, füllen sie mit der BorderPane und legen eine feste Größe fest. Der Rest der Klasse erschließt sich von selbst (Main-Methode mit launch-Methode etc.).

Die Control-Klasse: Unsere Attribute in der Control-Klasse sind:

**private** ListView<ObservableContactDetails> listView = **new** ListView<>();

**private** TableView<ObservableContactDetails> tableView = **new** TableView<>();

**private** ObservableList<ObservableContactDetails> data = FXCollections.*observableArrayList*();

**private** AddressBook aBook= **new** AddressBook();

**private** Label errorText;

Im Konstruktor rufen die Methode fuelleAdressBook() und fuelleObservableData auf, damit dies stets zu Programmstart ausgeführt wird. Sie füllt unser Adressbuch mit Dummy-Werten und diese dann in die ObservabeleArrayList.

In der Methode fuelleTabelle() erzeugen wir nun unsere Tabelle und erkennen den Vorteil von Properties. Wichtig: die setEditable-Methode, um die Einträge dann auch wirklich editieren zu können. Mit der setItems-Methode packen wir die Kontakte in die TableView.

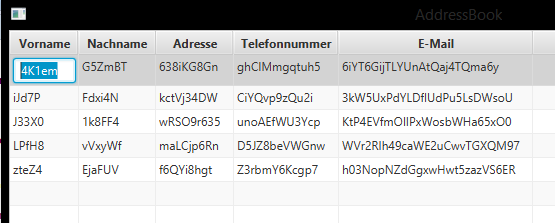
Wir müssen nun noch die Spalten der Tabelle erzeugen. Dies geschieht analog für jedes Attribut nach folgendem Schema:

TableColumn<ObservableContactDetails, String> vornameSpalte = **new** TableColumn<>("Vorname");

vornameSpalte.setCellValueFactory(e -> e.getValue().nachnameProperty());

vornameSpalte.setCellFactory(TextFieldTableCell.*forTableColumn*());

Die Spalten übergeben wir letztlich ganz einfach mit der addAll()-Methode. Wir können unsere View-Klasse nun ausführen und die Einträge erfolgreich ändern/umspeichern.



Aufgabenteil B:

Wir erstellen einen Knopf „Neu“, der mit einem einfachen Klick einen neuen Eintrag zum Editieren erzeugt. Dieser Knopf ruft durch ein ActionEvent die Methode addNewContact auf, die ein neues Objekt vom Typen ObservableContactDetails erstellt und diesen an unsere *observableArrayList* übergibt. Des weiteren wird der Kontakt an unser AdressBook durch addDetails weitergegeben. Somit ist er auch beim drucken sichtbar. :)

Aufgabenteil C:

analog wie in Aufgabe 2 c).

Aufgabenteil E:

Um die Integrierung des AddressBook zu gewährleisten, müsste auch die TabelView.setCellFactory überschrieben werden und bei jeder Editierung das AddressBook aufgerufen werden.

Uns ist aufgefallen, dass alle Änderung immer umständlich über Control zum AddressBook gehen. Eine bessere Lösung wäre hier gewesen, das AddressBook direkt als Control zu nutzen und somit umständliche Aufrufe zu vermeiden.