Е. А. АЛЬТМАН, Н. Г. АНАНЬЕВА

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ПЛАТФОРМЕ JAVAFX

ОМСК 2016

Министерство транспорта Российской Федерации
Федеральное агентство железнодорожного транспорта
Омский государственный университет путей сообщения

Е. А. Альтман, Н. Г. Ананьева

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ПЛАТФОРМЕ JAVAFX

Утверждено редакционно-издательским советом университета
в качестве учебно-методического пособия
к выполнению курсовой и лабораторных работ по дисциплине
«Объектно-ориентированное программирование»

Омск 2016

УДК 681.3
ББК 32.973
А58

Разработка приложений на платформе JavaFX: Учебно-методическое пособие к выполнению курсовой и лабораторных работ /Е. А. Альтман, Н. Г. Ананьева; Омский гос. ун-т путей сообщения. Омск, 2016. 33 с.

В методических указаниях рассматриваются основы разработки приложений на платформе JavaFX. Приведены сведения по методам разработки графического интерфейса приложения, обработки действий пользователя, созданию собственных элементов интерфейса и использованию сложных компонентов для отображения коллекций объектов. Изучении платформы сначала идет на упрощенном примере, затем студенты повторяют изученный материал, выполняя курсовую работу по индивидуальным заданиям.

Предназначено для студентов направлений подготовки «Информационные системы и технологии», «Информатика и вычислительная техника» и «Управление в технических системах», а также будет полезно для студентов других направлений подготовки.

Библиогр.: 8 назв. Табл. 0. Рис. 5.

Рецензенты: канд. техн. наук, доцент
канд. техн. наук, доцент

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Среди современных тенденций в разработке программного обеспечения можно выделить создание приложений с помощью высокоуровневых платформ. Платформа или фреймворк для разработки приложений берет на себя реализацию многих аспектов работы программы. Она позволяет, пусть и в ущерб скорости работы программы, быстро создавать сложные приложения. Программисту нужно описать только особенности приложения, а большое количество шаблонного кода предоставляется платформой.

В методических указаниях рассматривается платформа JavaFX. Она позволяет создавать приложения под основные операционные системы для персональных компьютеров, для мобильных устройств и веб приложения. JavaFX стала частью виртуальной машины языка Java, начиная с выпущенной в 2014 году 8-ой версии, что позволяет рассчитывать на ее широкое распространение.

Платформа JavaFX содержит много элементов, каждый из которых обладает большим числом свойств. Доскональное изучение всех элементов и их свойств является объемной и излишней с практической точки зрения задачей. В лабораторных работах рассматриваются только типичные элементы и свойства. Изучение остальных особенностей языка можно проводить с помощью документации по мере необходимости использования элементов.

Как показывает опыт изучения средств программирования, первая программа с использованием новой платформы оказывается неудачной с точки зрения эффективного использования возможностей платформы, но весьма полезной с точки зрения понимания принципов ее работы. После написания первой программы становится понятно, как ее нужно было делать. Поэтому изучение особенностей JavaFX будет проходить в несколько этапов.

На первом этапе студентами реализуется и модифицируется учебный пример, рассмотренный в теоретических частях лабораторных работ. На втором этапе реализуется программа по индивидуальному заданию и оформляется в виде курсовой работы.

В учебном примере рассматривается приложение для ведения учета затрат. Пользователь может вводить наименование товаров или других затрат, их категорию и стоимость. Приложение подсчитывает статистики этих затрат (например, общую сумму). В теоретической части рассматривается реализация минимального функционала для демонстрации принципов работы элементов платформы. В лабораторных работах студенты должны дополнить приложения некоторыми элементами. Полноценное приложение для ведения домашней бухгалтерии является одним из вариантов задания к курсовой работе.

Для выполнения курсовой и лабораторных работ используется бесплатно распространяемая среда разработки IntelliJ Idea (community edition) [1]. Дополнительно потребуется установить визуальный редактор Scene Builder [2]. Примеры для изучения элементов платформы JavaFX можно найти на сайте Using JavaFX UI Controls [3]. Подробное описание всех свойств элементов в документации фирмы Oracle [4].

Лабораторная работа 1

РАЗРАБОТКА ИНТЕРФЕЙСА ПРИЛОЖЕНИЯ

Ц е л ь р а б о т ы: изучить методы разработки дизайна графического интерфейса пользователя.

1.1. Теоретические сведения

1.1.1. Проект JavaFX

Платформа JavaFX разработана для упрощения создания насыщенных графических интерфейсов для приложений работающих на различных операционных системах, браузерах и мобильных устройствах.

Разработка под JavaFX поддерживается всеми основными IDE для Java. В нашем курсе мы будем использовать IDE IntelliJ Idea [1]. На рисунке 1.1 показан пример создания проекта.

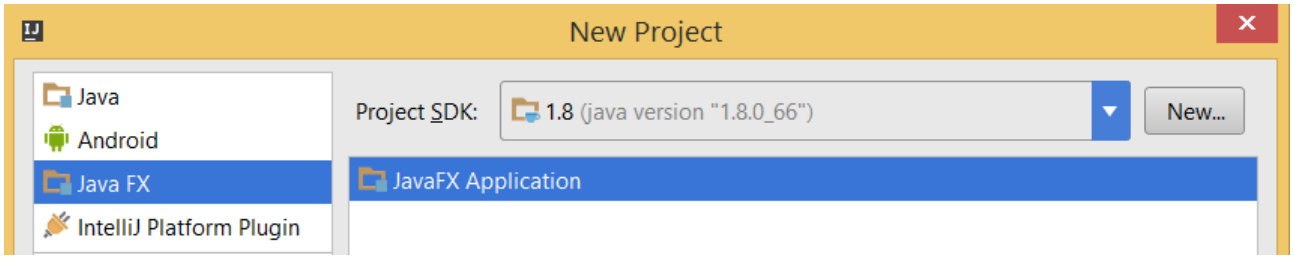

Рис. 1.1. Создание проекта JavaFX

При создании проекта среда автоматически создаст в папке «`src`» типичные файлы исходных кодов для JavaFX приложения — «`Controller.java`», «`Main.java`» и «`sample.fxml`». В файле «`sample.fxml`» в формате «`xml`» хранится разметка окна и графические элементы приложения. В файле «`Controller.java`» — код, обрабатывающий действия с графическими элементами. В файле «`Main.java`» — код, инициализирующий фреймворк JavaFX и подключающий к нему остальные файлы. Данный проект можно скомпилировать и запустить. В результате работы программы получится пустое окно.

1.1.2. Разработка графического интерфейса

Для кода, относящегося к графическому интерфейсу, создадим пакет «`view`». Перенесем в него файл «`sample.fxml`» и переименуем его как «`mainWin.fxml`».

Этот файл можно редактировать как обычный код или (что более предпочтительно) открыть его в визуальном редакторе «Scene Builder».

На правой панели редактора находятся компоненты, которые можно добавить в окно приложения. Для начального знакомства рассмотрим группы компонентов «Containers» и «Controls»

Контейнеры («Containers») или компоновщики это элементы графического интерфейса, которые не видны пользователю, а служат для размещения видимых элементов и других компоновщиков. Например, компоновщик «GridPane» позволяет размещать элементы в виде таблицы с произвольным числом строк или столбцов.

В учебном проекте будем использовать компоновщик «BorderPane». Он выделяет 5 областей для размещения элементов. В верхней («TOP») области мы разместим кнопки для вызова функционала нашего приложения. В нижней («BOTTOM») — рассчитываемые величины.

Между верхней и нижней областями в «BorderPane» находятся три области (в порядке слева на право): «LEFT» «CENTER» и «RIGHT». В «CENTER» разместим основные элементы нашего приложения, остальные области использовать не будем.

На рисунке 1.2 показано окно редактора «Scene Builder» для приложения с компоновщиком «BorderPane» и заполненными элемента в нижней и центральной областях.

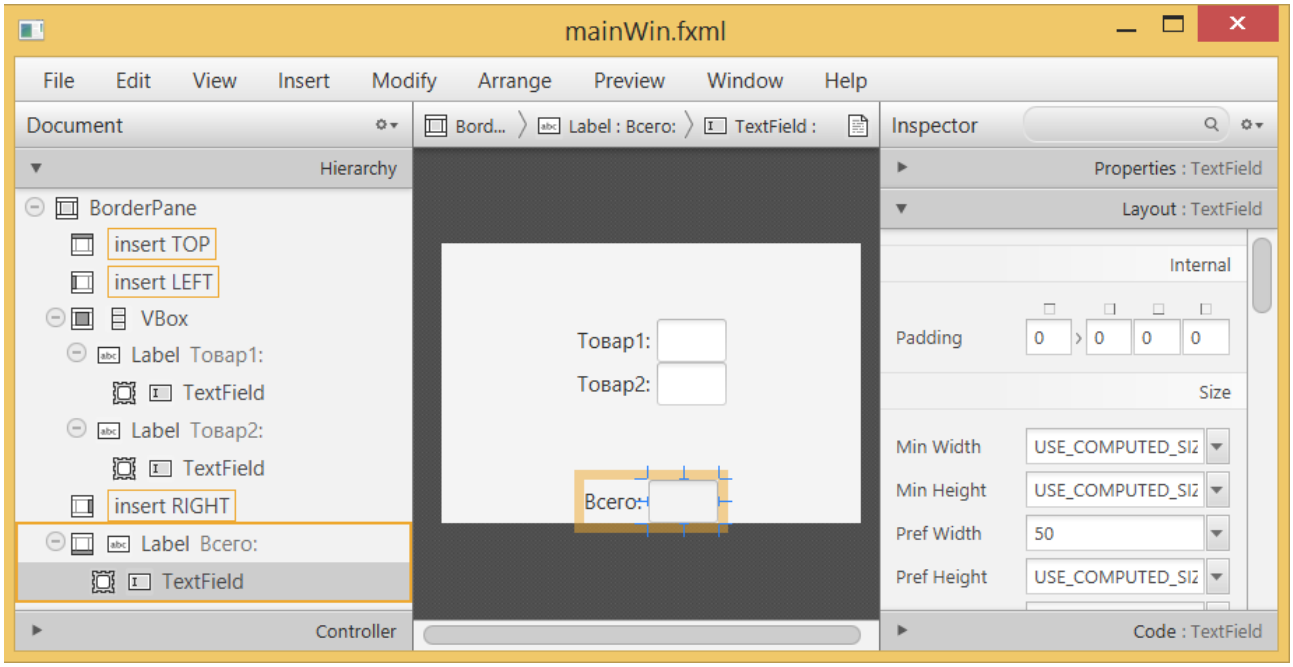

Рис. 1.2. Графический интерфейс в Scene Builder

В левой части окна открыта вкладка, содержащая иерархию элементов в документе. В нижнее поле «BorderPane» («BOTTOM») помещен элемент «Label» (метка) из группы компонентов «Controls». Этот элемент создает некоторую отметку в окне приложения, которая обычно представляет собой текст, но может содержать изображение или другие элементы графического интерфейса. В нашем примере метка содержит текст («Всего:») и элемент «TextField». Элемент «TextField» (текстовое поле) предназначен для ввода и отображения некоторых значений.

В правой части экрана находятся вкладки, предназначенные для настройки элементов. Вкладка «Properties» (свойства) предназначена для настройки свойств элементов, «Layout» (размещение) — их расположение, на вкладке «Code» (код) настраивается их связь с контроллером. Параметры во вкладках сгруппированы по нескольким разделам. В частности, во вкладке «Layout» есть разделы «Internal» (в котором можно задать внутренние отступы), «Size» (задание размеров) и другие.

В нашем примере вид метки «Всего:» отличается от вида, принятого по умолчанию. Для этого нужно установить в свойствах метки в разделе «Graphic» расположение графического элемента (текстового поля) справа, и установить предпочитаемую ширину («Pref Width») элемента «TextField», равную 50.

В центральной области временно разместим элементы для ввода цены двух товаров. Чтобы разместить несколько элементов используем компоновщик «Vbox», который размещает элементы вертикально один над другим. В этот компоновщик добавим вертикально две метки, как показано на рисунке 1.2.

1.1.3. Запуск приложения

Платформа JavaFX может преобразовать созданный в «Scene Builder» «fxml» файл в инициализированные и готовые к работе объекты. Для этого нужно инициализировать платформу и загрузить «fxml» файл. Ниже приведен исходный код класса, решающего эти задачи:

```
public class Main extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = FXMLLoader.load(
            getClass().getResource("/view/mainWin.fxml"));
        primaryStage.setTitle("Пример JavaFX приложения");
```

```
        primaryStage.setScene(new Scene(root));
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Главный класс приложения JavaFX должен наследовать класс «Application», который содержит методы по инициализации платформы. Для инициализации нужно вызвать метод «launch». После инициализации фреймворк вызывает метод «start», в котором и должна реализовываться основная логика приложения.

Метод «start» вызывается в потоке JavaFX платформы. Если приложение не использует другие потоки, то все объекты приложения лучше составлять в потоке JavaFX из метода «start». Аргументом этого метода является объект класса «Stage», который является контейнером верхнего уровня приложения. Он содержит в себе заголовок окна и стандартные кнопки работы с окном (минимизация и т. п.). С помощью метода «setTitle» установим заголовок нашего приложения.

Все компоновщики являются наследниками класса «Parent», поэтому мы загружаем «fxml» файл в переменную такого типа. Для загрузки используется метод «load» класса «FXMLLoader» платформы. Для размещения нашей разметки в окно приложения помещаем его в специальный контейнер «Scene».

Для отображения окна приложения на экране нужно вызвать метод «show». После запуска приложения на экране появиться окно, приведенное на рисунке 1.3.

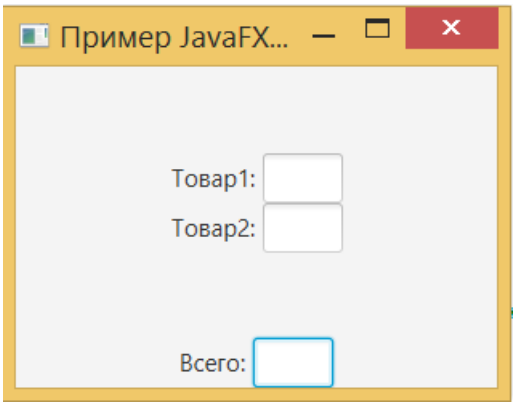

Рис. 1.3. Пример приложения

1.2. Задание к работе

1) Повторите пример приложения, приведенный в теоретической части.

2) Ознакомьтесь с имеющимися в «Scene Builder» элементами.

3) Ознакомьтесь со свойствами, которые можно установить для элементов «BorderPane» и «Label».

4) Используя компоновщик «HBox», переделайте разметку так, чтобы поле «TextField» в нижней части окна располагались рядом, а не внутри поля «Label».

5) Замените компоновщик «VBox» на «GridPane», расположите поля «TextField» и «Label» рядом.

6) Увеличьте в свойствах элемента «Всего:» размер шрифта и сделайте его полужирным.

7) Настройте расположение меток и текстовых полей для товаров так, чтобы метки занимали четверть горизонтального размера экрана.

1.3. Содержание отчета

1) Цель работы.

2) Задание по лабораторной работе.

3) Коды программ.

4) Результаты работы программы.

5) Выводы по лабораторной работе.

1.4. Контрольные вопросы

1) Назовите несколько компоновщиков, которые можно использовать в приложениях на платформе JavaFX.

2) Назовите несколько управляющих элементов, которые можно использовать в приложениях на платформе JavaFX.

3) Перечислите основные свойства, которые можно установить для элемента «Label».

4) Перечислите основные параметры размещения, которые можно установить для элемента «Label».

5) Какие классы и методы используются при запуске приложения на платформе JavaFX?

Лабораторная работа 2

ОБРАБОТКА ДЕЙСТВИЙ ПОЛЬЗОВАТЕЛЯ

Ц е л ь р а б о т ы: изучить способы обработки команд пользователя.

2.1. Теоретические сведения

2.1.1. События как объекты

Во время работы приложения происходят события, которые приводят к изменениям в его состоянии — пользователь нажал на кнопку, пришли данные по сети, некоторая функция выполнила свою задачу. В рамках объектно-ориентированного программирования такие события описываются с помощью объектов определенных классов. Такие объекты называются *событием*. Событие содержит описание изменения произошедшего в программе и объекта, который сформировал это событие (например, событие типа «нажатие на кнопку» содержит ссылку на объект типа «кнопка»).

Классы событий собраны в иерархию. На верхнем уровне находится класс «`Event`». От него наследуются классы, описывающие тип события (например, «`InputEvent`»), далее наследуется более конкретное описание события, например, по источнику («`MouseEvent`»).

Имеется несколько подходов к тому, как связать между собой объект, создающий события (например, кнопку на экране), и объект, обрабатывающий события.

Подход, принятый в ранних версиях Java, подразумевает использование объектов реализующих интерфейс «`ActionListner`» или его аналогов (*слушатели*). Этот интерфейс требовал реализации метода «`actionPerformed`», которому передается объект, описывающий событие. Объект, реализующий интерфейс «`ActionListner`» связывался с источником событий вызовом метода «`addActionListener`». Этот объект анализировал тип произошедшего события и либо обрабатывал его, либо передавал следующему слушателю.

В JavaFX обработчики обычно реализуют интерфейс «`EventHandler`», который содержит метод «`handle`» для обработки события. Объект типа «`EventHandler`» можно привязать к конкретному событию, происходящему с источником. Для этого источник содержит методы, позволяющие привязаться к нужному событию. Например, все графические элементы JavaFX содержат метод «`setOnMouseClicked`», который позволяет установить обработчик, который будет вызываться при нажатии «мыши».

2.1.2. Обработка событий на платформе JavaFX

Платформа JavaFX упрощает программирование обработки событий. Связь между источником и обработчиком можно указать прямо в «fxml» файле.

Источником событий будут элементы графического интерфейса — кнопки, поля ввода и др. Обработку событий нужно запрограммировать. Для этого создадим класс, в котором будут методы по обработке событий. В этот же класс удобно включить ссылки на элементы графического интерфейса, причем инициализировать эти ссылки можно также с помощью «fxml» кода.

Создадим файл «MainWinController.java» со следующим содержимым:

```
public class MainWinController {
    @FXML TextField  item1;
    @FXML TextField  item2;
    @FXML TextField  itogo;
    @FXML
     private void handleReset() {
        item1.setText("0");
        item2.setText("0");
        itogo.setText("0");
    }
}
```

Аннотацией «@FXML» помечаются поля и методы, которые используются загрузчиком при обработке «fxml» файла. Для того, чтобы загрузчик нашел эти поля и методы, нужно указать класс «MainWinController» как контролер для «fxml» файла. Это можно сделать в «Scene Builder» на вкладке «Controller».

В поля будут записаны ссылки на соответствующие объекты, созданные по «fxml» описанию. В «Scene Builder» для текстовых полей в метках «Товар1:», «Товар2:» и «Всего:» на вкладке «Code» в поле «fx:id» выбрать соответствующие поля класса «MainWinController».

Метод «handleReset» должен вызываться при нажатии кнопки «Сброс», которую мы добавим в наше приложение. Для того, чтобы при загрузке «fxml» файла этот метод стал обработчиком нужного события, для кнопки «Сброс» во вкладке «Code» нужно выбрать этот метод в поле «On Action». Теперь, после запуска приложения и нажатия кнопки «Сброс» во всех текстовых полях установится значение «0».

2.1.3. Реактивное программирование

Помимо действий, явно задаваемых пользователем (вроде нажатия кнопки «Сброс»), программа должна поддерживать согласованные состояния объектов. Например, в поле «Всего:» всегда должна быть сумма значений из полей товаров. В общем случае таких связей может быть много (например, несколько товаров, для которых подсчитано их количество, сумма, среднее число и т.д).

Подход к программированию, при котором устанавливаются связи между объектами и за обновлением объектов следит система, называется *реактивное программирование* (от слова реакция). Наиболее яркий пример такого подхода — программа «Excel». Формулы в ячейках «Excel» автоматически пересчитываются при изменении ячеек, которые используются в формуле.

В рамках объектно-ориентированного подхода реактивное программирование можно реализовать различными способами. Сейчас мы рассмотрим простой способ связывания объектов (более высокоуровневый подход — в следующих лабораторных работах).

Запишем «формулу» по которой будет обновляться поле «Всего:» (в классе «MainWinController»):

```
private void itogoUpdate(){
    Float sum = Float.parseFloat(item1.getText());
    sum = sum + Float.parseFloat(item2.getText());
     itogo.setText(sum.toString());
}
```

Теперь свяжем изменения элемента «Товар1:» с этой формулой:

```
public void init() {
    item1.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            itogoUpdate();
        }
    });
}
```

Метод «setOnAction» устанавливает объект типа «EventHandler <ActionEvent>» как обработчик события «onAction». Этот объект содержит метод «handle», который вызывается каждый раз при изменении объекта «item1».

Начиная с версии 8 языка Java объекты с одним методом можно заменить, так называемой, лямбда-функцией. Записывается это следующим образом. Сначала пишется аргумент метода, потом оператор «->» и в конце реализация метода.

Запись методов, вызываемых при изменении объекта через лямбда-функции, будет выглядеть следующим образом:

```
public void init() {
    item1.setOnAction(event -> itogoUpdate());
    item2.setOnAction(event -> itogoUpdate());
}
```

Наш метод обработки события не имеет аргументов, однако часто полезно получить информацию о произошедшем событии, которая содержится в переменной «event». В частности, из нее можно узнать тип и источник событий.

В нашем примере можно было бы обойтись без метода «init», указав метод «itogoUpdate» как реакцию на изменение полей в файле «fxml» с помощью «Scene Builder». На вкладке «Code» каждого элемента можно установить значение поля «On Action». В общем случае не все действия можно или удобно задавать с помощью «fxml», выбор способа остается за программистом.

Теперь нам нужно в какой-то момент вызвать метод «init». В конструкторе класса это сделать нельзя, поскольку поля класса еще не инициализированы. Этот метод нужно вызвать после загрузки «fxml» файла в методе «start» класса «Main». Заменим 1 строку этого метода на следующий код:

```
FXMLLoader loader = new FXMLLoader();
loader.setLocation(getClass()
    .getResource("view/mainWin.fxml"));
Parent root = loader.load();
MainWinController controller = loader.getController();
controller.init();
```

2.2. Задание к работе

1) Повторите пример приложения, приведенный в теоретической части.

2) Настройте вызов метода «itogoUpdate» при изменении полей товаров с помощью «fxml».

3) Добавьте поле, в котором бы отображалось среднее значение товаров.

4) Настройте поля таким образом, чтобы суммарное и среднее значение товаров пересчитывалось бы во время ввода данных.

5) Добавьте метку, в которой бы отображалось имя последнего отредактированного поля (включая поле «`Всего:`»).

2.3. Содержание отчета

1) Цель работы.
2) Задание по лабораторной работе.
3) Коды программ.
4) Результаты работы программы.
5) Выводы по лабораторной работе.

2.4. Контрольные вопросы

1) Объекты каких типов участвуют в обработке событий?

2) Какими способами можно настроить связь между источником события и его обработчиком?

3) Какая информация находится в объекте-событии?

4) К каким событиям, происходящие с текстовым полем, можно привязать обработчик событий?

Лабораторная работа 3

СОЗДАНИЕ ЭЛЕМЕНТОВ

Цель работы: изучить способы создания и настройки элементов графического интерфейса.

3.1. Теоретические сведения

3.1.1. Способы настройки элементов

Элементы JavaFX можно настраивать разными способами на различных уровнях. В простейшем случае можно установить значение некоторого свойства (например, ширину поля «`TextFiled`») или можно переопределить обработку событий, связанных с элементов (например, установить метод, который будет выполняться при завершении ввода текста в поле).

Более сложным способом настройки является создание собственного элемента. Создавать собственный элемент с нуля не имеет никакого смысла. Элементы создаются наследованием от готовых элементов на нужном уровне абстракции.

Графические элементы платформы «JavaFX» находятся в пакете «javafx.scene». Самым абстрактным элементов является элемент (класс) «Node». Он содержит самые общие свойства, без которых элемент не может быть элементом «JavaFX».

На одном из самых нижних уровней находится «TextField». Он реализует весьма конкретную вещь — текстовое поле. Можно проследить цепочку наследований, которая ведет от «Node» к «TextField»: «Node»–«Parent»–«Region»–«Control»–«TextInputControl»–«TextField».

Класс «Parent» является базовым для классов, которые могут содержать в себе другие элементы (например, метки, которые в нашем примере содержат текстовые поля). В классе «Region» добавлены методы по управлению размером объекта и его стилизации при помощи CSS. Класс «Control» добавляет методы, позволяющие пользователю взаимодействовать с графическим интерфейсом. «TextInputControl» добавляет базовый функционал для работы пользователя с текстом.

Рассмотренные ранее классы «Label» и «Button» являются наследниками класса «Labeled», наследника класса «Control». Все компоновщики являются наследниками класса «Pane», являющегося наследником класса «Region».

Создаваемый программистом элемент интерфейса может быть наследником любого класса из иерархии классов пакета «javafx.scene». Чем выше класс-родитель в иерархии, тем больше возможностей для его настройки, но и требуется больше кода при реализации элемента.

3.1.2. Пример создания элемента

В рассматриваемом примере программы поля «TextField» не совсем подходят для ввода цены товара. В них можно вводить любые символы. Определим свой элемент, в который можно вводить только числа. Для упрощения примера будем считать, что ценой может быть любое вещественное число.

Создадим в проекте пакет «Component», а в нем класс «FloatField» со следующим кодом:

```
public class FloatField extends TextField{
    public FloatField() {
```

```
        super();
        this.setText("0");
    }
}
```

Это класс пока отличается от обычного текстового поля только тем, что в конструкторе устанавливается начальное значение «0».

Теперь можно заменить текстовые поля на наш класс в «fxml» файле. Для этого добавим в заголовок следующую строку: «<?import Component.FloatField?>» и заменим все поля «TextField» на «FloatField».

Программа должна запуститься с новыми полями. Однако, при этом нельзя будет редактировать «fxml» файл в «Scene Builder», он не знает созданный нами элемент. Для того, чтобы созданные элементы можно было использовать в «Scene Builder» и других программах, их нужно оформить как отдельный модуль и скомпилировать в «jar» файл. Этот вопрос рассмотрим следующим пунктом.

Теперь нам нужно ограничить ввод пользователя вещественными числами. Обычный подход заключается в перехвате пользовательского ввода (например, события «KeyPressed») и его проверке.

В версии Java 8u40 появился класс «TextFormatter<T>», упрощающий эту задачу. Один из вариантов его использования приведен ниже (его нужно добавить в конструктор «FloatField»):

```
this.setTextFormatter(new TextFormatter<String>(
    s -> {
        if (s.getControlNewText().isEmpty())
            return s;
        try{
            Float.parseFloat(s.getControlNewText());
            return s;
        } catch (NumberFormatException e) {
            return null;
        }
    }
));
```

К текстовому полю мы добавляем класс «TextFormatter<T>», который будет перехватывать ввод пользователя. При изменении текста будет вызываться метод этого класса, определенный в лямбда-функции. Аргумен-

том s этого метода будет иметь тип «TextInputControl.Change» и он будет содержать информацию об изменении текста текстового поля.

Если в результате изменения получается пустой текст или корректное вещественное число, мы возвращаем изменение s и оно будет принято. Иначе мы возвращаем null и изменение текста будет отклонено.

3.1.3. Создание модуля

Выделение классов в отдельный модуль является полезным средством для разбиения задачи на части и повторного использования кода. К сожалению, в разных средах разработки это действие выполняется по разному, поэтому рассмотренный ниже способ будет работать только в среде «IntelliJ Idea».

Для создания модуля вызовем окно свойств проекта (сочетание клавиш «Ctrl-Shift-Alt-S») и перейдем в раздел «Modules». Создадим новый модуль, назовем его «Component». В основном модуле нашей программы на вкладке «Dependencies» добавим зависимость основного модуля от модуля «Component». Сохраним свойства и закроем их окно.

В папке «src» модуля «Component» создадим пакет «Control» и перенесем в него класс «FloatField». Наш проект остается работоспособным, основной модуль находит класс «FloatField» в зависимом модуле «Component».

Для того, чтобы модуль был доступен в IDE его нужно скомпоновать в файл типа «jar». Результаты работы средств среды разработки, которые можно запускать и использовать вне среды, называются *артифактами*. Снова запускаем окно свойств проекта и переходим на вкладку «Artifacts». Создаем артифакт типа «jar» из модуля «Component».

На вкладе «Output Layout» проверяем содержимое «jar» файла артифакта. В нем должен находиться только элемент «Component compile output» — результат компиляции компонента. Если поставить галочку «Build on make» артифакт будет собираться при каждой компиляции проекта. Это имеет смысл только во время отладки компонента. Обычно для сборки артифакта лучше давать команду из среды разработки в меню «Build-Build Artifacts».

После сборки артифакта в проекте в папке «out-artifacts» появится нужный «jar» файл. Его можно подключать как библиотеку к другим проектам или перетащить в «SceneBuilder» на вкладку «Library».

3.2. Задание к работе

1) Повторите пример приложения, приведенный в теоретической части.

2) Создайте элемент для отображения информации о товаре, который бы включал название и цену товара, поместите его в модуль «Component».

3) Переделайте приложение с использованием созданного элемента.

3.3. Содержание отчета

1) Цель работы.
2) Задание по лабораторной работе.
3) Коды программ.
4) Результаты работы программы.
5) Выводы по лабораторной работе.

3.4. Контрольные вопросы

1) Какие существуют способы настройки элементов графического интерфейса на платформе JavaFX?
2) Какие свойства элемента нужно переопределить в наследном классе?
3) Какие свойства элемента можно переопределить в наследном классе?
4) Какие свойства элемента нельзя переопределить в наследном классе?
5) Как создать «jar» файл в среде «IntelliJ Idea»?

Лабораторная работа 4

КОМПОНЕНТ TABLEVIEW

Ц е л ь р а б о т ы: ознакомиться с реактивными типами данных и POJO объектами, изучить приемы отображения их в компонент TableView.

4.1. Теоретические сведения

4.1.1. Реактивные типы данных

Данные, с которыми работает пользователь, должны быть реактивными, т.е. не только хранить значения, введенные пользователем, но и уметь сообщать об изменении значения всем объектам, нуждающимся в этой информации.

В JavaFX реактивные типы данных называются «Properties» (*свойства*). Они должны реализовывать одни из интерфейсов из иерархии, начинающийся с интерфейса «Observable»:

```
public interface Observable {
    void addListener(InvalidationListener listener);
    void removeListener(InvalidationListener listener);
}
```

Наиболее типичными интерфейсам являются интерфейсы «ObservableValue» (который добавляет метод «getValue») и «ObservableList» (добавляющий методы по работе с коллекциями).

Для основных типов данных Java в JavaFX имеются реактивные аналоги, например для String — StringProperty, Float — FloatProperty и т.д.

Для основных коллекций Java в JavaFX можно получить реактивный аналог с помощью класса «FXCollections». Например, метод «observableArrayList» вернет реактивный аналог «ArrayList».

Примеры использования реактивных данных мы рассмотрим ниже. Нужно понимать, что за реактивность приходиться платить быстродействием и расходом памяти.

4.1.2. POJO объекты

Данные, с которыми работают пользователи, обычно отображают некоторые объекты реального мира. Примером такого объекта является объект, содержащий информацию о человеке, позволяющий установить его имя, должность, зарплату и т. п.

Одним из подходов к работе с такими данными является использование POJO объектов (Plain Old Java Object — простой Java-объект в старом стиле). Такие объекты содержат поля свойств объекта реального мира и методы доступа к полям. При этом объекты не содержат поля и методы для служебных целей — выполнения транзакций, обеспечение безопасности доступа и т. п.

Для работы с POJO объектами существуют библиотеки и фреймворки. При выполнении определенных требований такой объект может без дополнительного кодирования сохранен в базе данных, файл «xml», передан по сети или представлен пользователю в структурированном виде.

В «JavFX» имеется компонент «TableView», который позволяет выводить коллекцию POJO объектов в виде таблицы. Сформируем POJO класс для хранения затрат, используя реактивные типы данных:

```
public class Expense {
    StringProperty name = new SimpleStringProperty("");
    FloatProperty  cost = new SimpleFloatProperty(0);
}
```

Для корректной работы POJO объекта большинству библиотек требуются методы чтения полей (getter — *геттер*), и записи (setter — *сеттер*). Для реактивных типов требуется дополнительный метод доступа, который возвращает соответствующее свойство.

Методы доступа требуют большое количество кода, поэтому в современных средах разработки их можно автоматически генерировать. В IDE IntelliJ Idea для этого нужно использовать сочетания клавиш Alt-Insert, выбрать пункт Getter and Setter и выбрать поля, для которых нужно генерировать методы доступа. Например, для поля name будет сгенерирован следующий код:

```
public String getName() { return name.get();  }
public StringProperty nameProperty() { return name; }
public void setName(String name) { this.name.set(name); }
```

Для тестирования нам также понадобится конструктор:

```
public Expense(String name, Float cost) {
    this.name = new SimpleStringProperty(name);
    this.cost = new SimpleFloatProperty(cost);
}
```

4.1.3. Приложение с TableView

Переделаем приложение для учета затрат. В контроллере главного окна оставим следующие поля:

```
@FXML TableView<Expense> table;
@FXML TableColumn<Expense,String> nameColumn;
@FXML TableColumn<Expense,Number> costColumn;
```

Поле «table» содержит ссылку на таблицу с затратами, «nameColumn» — на колонку с именем, «costColumn» — со стоимостью.

В «Scene Builder» вставим таблицу в центр компоновщика, переименуем колонки как «Название» и «Цена» и привяжем элементы к полям контроллера. Получим следующий «fxml» код для таблицы (показаны основные элементы):

```
<TableView fx:id="table" BorderPane.alignment="CENTER">
    <columns>
        <TableColumn fx:id="nameColumn" text="Название"/>
        <TableColumn fx:id="costColumn" text="Цена"/>
    </columns>
</TableView>
```

Для корректной работы нужно настроить компонент «TableView». Переделаем метод «init» контроллера:

```
public void init(ObservableList<Expense> expenseList){
    table.setItems(expenseList);
    nameColumn.setCellValueFactory(cellData ->
        cellData.getValue().nameProperty());
    costColumn.setCellValueFactory(cellData ->
        cellData.getValue().costProperty());
}
```

Теперь этому методу передается коллекция объектов типа «Expence». Метод «setItems» добавляет коллекцию в таблицу, элементы коллекции соответствует строкам таблицы.

Для отображения отдельных ячеек таблицы используются объекты типа «CellValueFactory». Объект этого типа имеет один метод, поэтому при его создании можно воспользоваться лямбда-функцией.

Этому методу передается объект типа «CellData», который содержит информацию как о самой ячейке (номер строки, столбца и т.д.), так и об объекте, который нужно отобразить. С помощью метода «getValue» мы получаем отображаемый объект (типа «Expence»). В нашем примере мы может без дополнительных действий вернуть соответствующей свойство объекта («nameProperty» или «costProperty»).

Теперь осталось переделать главный класс программы («Main»), чтобы он содержал коллекцию затрат, и заполнить эту коллекцию тестовыми данными. Добавим в него следующие строки:

```
ObservableList<Expense> expenseList =
    FXCollections.observableArrayList();

private void testInit(){
    expenseList.add(new Expense("Хлеб",new Float(15)));
    expenseList.add(new Expense("Вода",new Float(10)));
    expenseList.add(new Expense("Проезд",new Float(20)));
    ObservableValue<Number> x;
}
```

Не забудем включить вызов метода «`testInit`» в метод «`start`» и в вызов метода «`init`» контроллера передать коллекцию.

После запуска приложения на экране появиться окно, приведенное на рисунке 4.1.

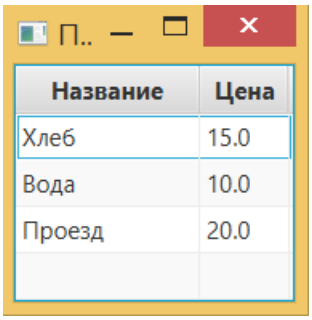

Рис. 4.1. Пример использования `TableView`

4.2. Задание к работе

1) Повторите пример приложения, приведенный в теоретической части.

2) Измените класс «`Expense`», добавив в него поле для хранения категории расходов.

3) Переделайте основное приложение для работы с новым вариантом класса «`Expense`».

4) Добавьте в приложение текстовое поле «`Итого`», в котором бы отображалось общее количество затрат.

4.3. Содержание отчета

1) Цель работы.

2) Задание по лабораторной работе.

3) Коды программ.

4) Результаты работы программы.

5) Выводы по лабораторной работе.

4.4. Контрольные вопросы

1) Что такое реактивные типы данных, и как они реализуются на платформе `JavaFX`?
2) Что такое POJO объекты?
3) Что такое геттеры и сеттеры и как они используются?
4) Как задаются столбцы компонента «`TableView`»?
5) Как задаются строки компонента «`TableView`»?
6) Как задается отображение элементов в таблице «`TableView`»?

Лабораторная работа 5

РЕДАКТИРОВАНИЕ ДАННЫХ В `TABLEVIEW`

Цель работы: изучить методы редактирования данных с помощью диалоговых окон и изменением ячеек компонента `TableView`.

5.1. Теоретические сведения

5.1.1. Создание диалоговых окон

Одним из способов изменения данных в таблице заключается в том, чтобы создать отдельное окно в котором редактируются записи для отдельной строки таблицы.

Создадим разметку для окна, приведенную на рисунке 5.1.

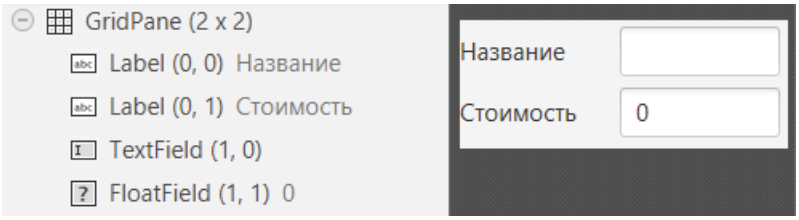

Рис. 5.1. Разметка диалогового окна

Для этого окна создадим контроллер:

```
public class EditWinController {
    ObservableList<Expense> expenseList;
    int editingIndex;
    @FXML TextField name;
    @FXML FloatField cost;
```

```
    public void init(ObservableList<Expense> expenseList,
        int editingIndex){
        this.expenseList = expenseList;
        this.editingIndex = editingIndex;
        Expense expense = expenseList.get(editingIndex);
        name.setText(expense.getName());
        cost.setText(new Float(expense.getCost()).toString());
        name.getParent().setOnMouseExited(e->exit());
    }

    public void exit(){
        Expense expense = new Expense(name.getText(),
            Float.parseFloat(cost.getText()));
        expenseList.set(editingIndex, expense);
    }
}
```

Метод «`init`» должен вызываться перед запуском окна. Он запоминает ссылку на коллекцию и индекс элемента и инициализирует значения полей. Также он устанавливает метод «`exit`» как обработчик события, возникающего при выходе из окна. Метод «`exit`» запоминает введенные пользователем данные.

В главное окно программы добавим кнопку «`Изменить`». Обработчик нажатия на эту кнопку должен запустить диалоговое окно для изменения текущей записи:

```
FXMLLoader loader = new FXMLLoader();
loader.setLocation(
getClass().getResource("editWin.fxml"));
Pane page = loader.load();
Stage dialogStage = new Stage();
dialogStage.initModality(Modality.WINDOW_MODAL);
Scene scene = new Scene(page);
dialogStage.setScene(scene);

EditWinController controller = loader.getController();
int editingIndex = table.
    getSelectionModel().getFocusedIndex();
```

```
controller.init(expenseList, editingIndex);
dialogStage.showAndWait();
```

Создание диалогового окна отличается от создания обычного указанием его модальности (т.е. того, что это окно работает «поверх» окна, в котором оно создано). Для отображения окна нужно использовать метод «`showAndWait`».

Для инициализации контроллера диалогового окна мы определяем индекс текущей строки и записываем его в переменную «`editingIndex`». После этого вызывается метод инициализации контроллера.

5.1.2. Настройка редактирования ячейки

Открытие диалогового окна является довольно тяжеловесным способом для редактирования данных. Обычно проще настроить возможность редактировать данные прямо в ячейке.

Ячейка таблицы отображается и обрабатывает события классом «`TableCell`». Это параметризованный класс с двумя параметрами. Первый параметр имеет тип, соответствующий объекту, выводимому в строке таблицы. Второй параметр имеет тип, соответствующий текущей колонке. Универсальным способом настройки работы ячейки является создание класса наследника «`TableCell`», в котором переопределяется необходимое поведение ячейки.

Создадим класс для работы с ячейкой, который бы позволял редактировать в ячейке вещественные числа:

```
public class FloatCell extends
    TableCell<Expense, Number> {
    FloatField number;
    ObservableList<Expense> expenseList;
    Expense expense;

    public FloatCell(ObservableList<Expense> expenseList)
    {   this.expenseList = expenseList;
    }
}
```

Нам понадобятся поля «`FloatField`» для хранения элемента, который появится в ячейке на время ее редактирования, реактивная коллекция выводимых данных, которую мы обновим при завершении редактирования, и объект с данными текущей строки. В конструкторе класса зададим коллекцию данных.

Для корректной работы класса нам нужно переопределить три метода: «`startEdit`», который вызывается при начале редактирования ячейки; «`cancelEdit`», который вызывается при завершении редактирования и «`updateItem`», который вызывается при необходимости обновить вид ячейки.

В начале редактирования создаем объект типа «`FloatField`», считаем текущий редактируемый объект, возьмем из объекта текущей значение стоимости в поле «`FloatField`» и установим это поле как отображаемое:

```
@Override
public void startEdit() {
    if (!isEmpty()) {
        super.startEdit();
        number=new FloatField();
        expense = getTableView()
            .getSelectionModel().getSelectedItem();
        number.setText(expense
            .costProperty().getValue().toString());
        setGraphic(number);
    }
}
```

При завершении редактирования создаем новый объект типа «`Expense`» и сохраняем его в коллекции с тем же индексом, который имел объект, отображаемый в этой строке:

```
@Override
public void cancelEdit() {
    super.cancelEdit();
    int index=expenseList.indexOf(expense);
    float newCost = Float.parseFloat(number.getText());
    Expense newExpense = new Expense(
        expense.getName(),newCost);
    expenseList.set(index, newExpense);
    setGraphic(null);
}
```

В методе, ответственном за обновление, нужно проанализировать ситуацию, в которой был вызван метод (ячейка не отображается, редактируется или просто выводится), и, в зависимости от ситуации, установить нужный отображаемый объект:

```
@Override
public void updateItem(Number item, boolean empty) {
    super.updateItem(item, empty);
    if (empty) {
        setText(null);
        setGraphic(null);
    } else
    if (isEditing())
        setGraphic(number);
    else {
        setText(getItem().toString());
        setGraphic(null);
    }
}
```

После создания класса, ответственного за редактирование ячейки, его нужно привязать для вызова в нужном столбце:

```
costColumn.setCellFactory(
    cellData -> new FloatCell(expenseList));
```

5.2. Задание к работе

1) Повторите пример приложения, приведенный в теоретической части.

2) Переделайте приложение для работы с вариантом класса «`Expense`», разработанным в предыдущей работе.

3) Реализуйте класс для ячеек в столбце с категорий затрат так, чтобы пользователь мог выбрать категорию из выпадающего списка.

5.3. Содержание отчета

1) Цель работы.
2) Задание по лабораторной работе.
3) Коды программ.
4) Результаты работы программы.
5) Выводы по лабораторной работе.

5.4. Контрольные вопросы

1) Что нужно сделать, чтобы создать диалоговое окно?
2) Как получить ссылку на элемент, который выделен в таблице?
3) Какие параметры имеет класс «`TableCell`»?
4) Какой метод вызывается при начале редактирования ячейки?
5) Какой метод вызывается при завершении редактирования ячейки?

Курсовая работа

РАЗРАБОТКА ПРИЛОЖЕНИЯ НА ПЛАТФОРМЕ JAVAFX

Цель работы: Закрепить навыки разработки приложений на платформе JavaFX.

6.1. Содержание работы

В курсовой работе разрабатывает приложение на платформе JavaFX. Приложение разрабатывается по индивидуальному заданию, примеры заданий приведены ниже. Студенты могут предложить и согласовать с преподавателем свои темы, сложность и содержание которых соответствует темам примерных заданий.

Основные этапы выполнения курсовой работы совпадают с изученными в лабораторных работах элементами платформы JavaFX. Необходимо разработать интерфейс приложения и создать «`fxml`» файлы для всех окон. Далее разрабатываются нестандартные элементы графического интерфейса. Следующим шагом разрабатываются контроллеры окон.

В дополнении к пройденному на лабораторных работах материалу самостоятельно изучается: технология «`JAXB`» для сохранения данных введенных пользователем [5], библиотека печати платформы JavaFX [6] и сборка приложений для распространения [7]. Эти библиотеки и технологии необходимо использовать в курсовой работе.

6.2. Примерные варианты заданий

«`Календарь соревнований`». Позволяет пользователю вводить и просматривать данные об участниках соревнований (команд или спортсменов), результатов соревнований, просматривать статистику участников.

«Контроль расходов». Позволяет пользователю вводить категории расходов, сами расходы (товары, услуги и т.п. вместе со стоимостью), помечать расходы (нужные, ненужные и т.д), просматривать статистику расходов.

«Домашний каталог». Позволяет вести каталог домашней коллекции (книг, дисков, и т.д). Пользователь должен иметь возможность настроить поля для свойств элементов коллекции, просматривать элементы коллекции, упорядоченные по указанному свойству.

«Журнал преподавателя». Пользователь должен иметь возможность ввести группы студентов, в группы — студентов (фамилии, имя). Для групп можно назначать занятия (задания) с весовым коэффициентом. Для студентов можно задавать процент выполнения каждого задания. Программа должна рассчитывать рейтинг студентов используя весовые коэффициенты.

«Дневник студента». Пользователь может ввести расписание занятий (повторяющиеся и одиночные). К занятием можно давать комментарии (например, задание, степень его выполнения). Информацию можно просматривать в виде расписания, списка дел (весь, срочных, невыполненных).

«Разработка тестов». Программа позволяет создать категории вопросов и в них добавлять вопросы. Должны поддерживаться основные категории вопросов: короткий ответ, однозначный или многозначный выбор и др. Должна быть предусмотрена возможность импорта вопросов в стандартные форматы (например, moodle xml).

6.3. Содержание пояснительной записки

Основная часть пояснительной записки, помимо требуемых по СТП ОмГУПС [8], должна содержать следующие разделы.

«Техническое задание» должно содержать наименование, назначение и основные функции разрабатываемого программного обеспечения (ПО); структуру и пользовательский интерфейс ПО; календарный план (примерно 1 страница).

В разделе «Средства разработки» нужно описать характеристики используемого языка программирования, используемые библиотеки и средства разработки (IDE, системы тестирования, сборки и другие используемые системы) (примерно 1 страница).

В разделе «Структура программы» для ООП программ нужно привести диаграмму классов. Если программа содержит несколько пакетов (модулей, компонентов), то нужно привести диаграмму компонентов. Для каждого

класса в тексте привести его ответственность (за какие функции или процессы класс отвечает). Описание полей и методов классов в этом разделе стоит опустить.

В разделе «Реализация программы» нужно описать работу не шаблонных методов классов. Для иллюстрации их работы можно использовать диаграммы последовательностей.

В разделе «Инструкция пользователя» нужно описать работу для пользователя, знакомого с предметной областью и стандартными интерфейсными элементами.

К работе должны быть следующие приложения (могут быть и другие). Приложение А (обязательное) — структура программы. Приложение Б (обязательное) — интерфейс программы. Приложение В (обязательное) — диаграммы последовательностей реализующие основные функции программы. Приложение Г (обязательное) — код программы, включая файлы на «fxml». Код программы должен быть оформлен в соответствии с правилами хорошего тона: разбит на небольшие части, помещающиеся на страницах целиком; инструкции внутри блоков должны быть смещены вправо; дополнительный перенос строк по сравнению с кодом в редакторе не допускается. Код может быть выведен более мелким шрифтом (8–10 пунктов).

Библиографический список

1. IntelliJ Idea the Java IDE /https://www.jetbrains.com/idea/
2. Scene Builder — Gluon /http://gluonhq.com/open-source/scene-builder/
3. Using JavaFX UI Controls /https://docs.oracle.com/javafx/2/ui_controls/jfxpub-ui_controls.htm
4. Документация по платформе JavaFX /https://docs.oracle.com/javase/8/javafx/api/overview-summary.html
5. Учебник по JavaFX 8 — Часть 5: Хранение данных в XML /http://code.makery.ch/library/javafx-8-tutorial/ru/part5/
6. Package javafx.print /https://docs.oracle.com/javase/8/javafx/api/javafx/print/package-summary.html
7. Учебник по JavaFX 8 — Часть 7: Развертывание /http://code.makery.ch/library/javafx-8-tutorial/ru/part7/
8. СТП ОмГУПС–1.2–2005 // Омск, ОмГУПС 2005

Учебное издание

АЛЬТМАН Евгений Анатольевич,
АНАНЬЕВА Надежда Геннадьевна

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ПЛАТФОРМЕ JAVAFX

Учебно-методическое пособие

Редактор Н. А. Майорова
Корректор И. А. Сенеджук

***

Подписано в печать . .2016. Формат $60 \times 84\,{}^{1}/_{16}$.
Офсетная печать. Бумага офсетная. Усл. печ. л. 2,3. Уч.-изд. л. 2,5 .
Тираж 100 экз. Заказ .

**

Редакционно-издательский отдел ОмГУПСа
Типография ОмГУПСа

*

644046, г. Омск, пр. Маркса, 35