О. В. Минакова

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ: ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ В РЕАЛИЗАЦИИ JAVAFX ПРИЛОЖЕНИЙ

Практикум

Воронеж 2020

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Воронежский государственный технический университет»

О. В. Минакова

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ: ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ В РЕАЛИЗАЦИИ JAVAFX ПРИЛОЖЕНИЙ

Практикум

Воронеж 2020

УДК 681.3(075.8)
ББК 32.973я7
М57

Рецензенты:

кафедра математических методов исследования операций
Воронежского государственного университета
(зав. кафедрой д-р техн. наук, проф. Т. В. Азарнова);
д-р техн. наук, проф. А. П. Преображенский

Минакова, О. В.

Технологии программирования: паттерны проектирования в реализации JavaFX приложений: практикум [Электронный ресурс]. – Электрон. текстовые и граф. данные (6,5 Мб) / О. В. Минакова; ФГБОУ ВО «Воронежский государственный технический университет». – Воронеж: ФГБОУ ВО «Воронежский государственный технический университет», 2020. – 1 электрон. опт. диск (CD-ROM). – Систем. требования: ПК 500 и выше; 256 Мб ОЗУ; Windows XP; SVGA с разрешением 1024x768; Adobe Acrobat; CD-ROM дисковод; мышь. – Загл. с экрана.

М57

ISBN 978-5-7731-0911-2

В практикуме представлены теоретические сведения по разработке приложений на основе паттернов Go&F и методические рекомендации для практической реализации насыщенных графических приложений с использованием технологии JavaFX.

Издание предназначено для проведения лабораторного практикума для студентов третьего курса, обучающихся по направлению 09.03.02 «Информационные системы и технологии», дисциплине «Технологии программирования», а также для проведения занятий и организации самостоятельной работы студентов, обучающихся по направлению 27.03.03 «Системный анализ и управление», дисциплине «Системная архитектура информационных систем» и направлению 09.03.03. «Прикладная информатика», дисциплине «Теория и технология программирования».

Ил. 81. Табл. 1. Библиогр. 10 назв.

УДК 681.3(075.8)
ББК 32.973я7

Издается по решению редакционно-издательского совета
Воронежского государственного технического университета

ISBN 978-5-7731-0911-2

ВВЕДЕНИЕ

Когда любой разработчик приступает к работе, он не начинает работу с чистого листа, а имеет набор стандартных приемов, которые он будет применять. Конечно, он сделает что-то и нестандартное, свое – но наверняка другими стандартными средствами, только более низкоуровневыми. Но большей частью он будет применять идеи и решения, которые были придуманы ранее, и многократно применены и проверены.

Фундаментальной основой программирования является повторное использование кода. У программистов в запасе всегда обширный набор библиотек, реализующих стандартные и часто используемые алгоритмы, и опыт правильного повторного использования кода. Но четыре ведущих специалиста IBM Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес (часто исп. GoF – «банда четырех») создали шаблоны построения самих программных приложений. Они собрали наиболее эффективные решения, систематизировали, выделили в них самое главное, дали название, классифицировали, описали достоинства и недостатки 23 стандартных приемов проектирования [1]. Эти решения теперь являются основными правилами проектирования взаимодействия программных объектов, вошли во многие языки программирования и ключевыми принципами конструирования программных модулей.

Шаблоны (паттерны, от англ. patterns) проектирования – это проверенные и готовые к использованию решения часто возникающих в повседневном программировании задач. Это не класс и не библиотека, которую можно подключить к проекту. Шаблон – это проверенное архитектурное решение, показывающее как оптимально и безопасно организовать взаимодействие модулей и библиотечного кода для конкретной задачи. Шаблон проектирования, подходящий под задачу, реализуется в каждом конкретном случае. Кроме того, он не зависит от языка программирования и легко реализуется в большинстве современных языков программирования. Будучи примененным неправильно или к неподходящей задаче, может принести немало проблем, а правильно примененный шаблон – поможет решить задачу легко и просто.

Чтобы правильно и легко использовать шаблоны (паттерны) проектирования необходимо учиться применять их при решении различных задач, поэтому целью данного практикума является формирование навыков программирования на основе шаблонов. В общем случае каждый паттерн описывается следующими составляющими:

1. Имя является уникальным идентификатором паттерна. Имена паттернов проектирования, описанных в [1], являются общепринятыми.

2. Проблема (задача) описывает ситуацию, в которой можно применять паттерн.

3. Решения задачи проектирования в виде паттерна определяет общие функции каждого элемента и отношения между ними.

4. Результаты представляют следствия применения паттерна.

Практикум состоит из практических заданий с максимально кратким описанием шаблона, которые следует применить к поставленной задаче, и детальными указаниями по их выполнению. Для более детального изучения паттернов рекомендуется обратиться к [2, 5]. Все разделы снабжены примерами построения кода.

В качестве платформы разработки предлагается использовать Java FX – мощную современную среду графического интерфейса пользователя для визуального отображения логики приложений (программных моделей) [7]. Данное учебное пособие включает основные концепции, функции и описание структуры программ на JavaFX, а также примеры, предназначенные для использования технологии разработки мультимедийных приложений Java FX, поскольку ее использование оказывает значительное влияние на архитектуру всего приложения. Практикум предполагает, что студентов есть практические знания Java.

Для создания представлений в рамках данного практикума предполагается использование прикладного программного интерфейса JavaFX - API библиотеки JavaFX. Начиная с версии Java SE 7 Update 6, JavaFX была частью реализации Oracle Java SE[1] (инструкция по созданию проекта в IntelliJ IDEA с JDK 10 и ниже (см. приложение), но начиная с JDK 11 больше не является частью JDK. Поэтому для Java 11 и выше, необходимо загрузить JavaFX SDK с открытым исходным кодом в дополнение к JDK (инструкция в приложении). JavaFX Runtime состоит из набора библиотек Java, обеспечивающих пользовательские интерфейсы современных стандартов, а также определенный рабочий код, позволяющий получить доступ к определенным аппаратным ресурсам (например, видеокарте), поэтому требует определенной настройки среды разработки.

[1] Для запуска приложений JavaFX на вашем компьютере должны быть установлены и среда исполнения Java Runtime Environment (JRE), и среда исполнения JavaFX Runtime (см. Панель управления→Программа→Программы и компоненты проверить присутствует ли JavaFX)

РАЗРАБОТКА ГРАФИЧЕСКОГО ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ НА ПЛАТФОРМЕ JAVAFX

Данный раздел содержит описание вопросов, связанных с использованием платформы JavaFX для построения представлений программных моделей.

Характеристики платформы JavaFX

JavaFX – это платформа клиентских приложений нового поколения с открытым исходным кодом для настольных, мобильных и встроенных систем, построенных на Java.

JavaFX – это набор классов и интерфейсов, который определяет современный графический интерфейс пользователя (GUI) Java. Его можно использовать для создания графических интерфейсов пользователя в многофункциональных клиентских приложениях. JavaFX предоставляет разнообразный набор элементов управления, таких как кнопки, панели прокрутки, текстовые поля, флажки, деревья и таблицы, которые можно настроить практически для любого приложения. Кроме того, для повышения визуальной привлекательности элементов управления можно использовать эффекты, преобразования и анимацию. JavaFX также упрощает создание приложения за счет упрощения управления его элементами графического интерфейса и развертывания приложения. Таким образом, JavaFX не только позволяет создавать более захватывающие и визуально привлекательные пользовательские интерфейсы, но и значительно облегчает процесс разработки пользовательских приложений.

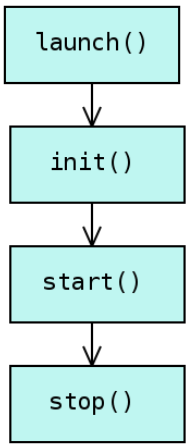
Компоненты JavaFX содержится в пакетах, которые начинаются с префикса javafx, например
`javafx.application, javafx.stage, javafx.scene`

Центральная метафора, реализованная в JavaFX, – это подмостки (Stage). Как и в случае реальной сценической пьесы, на подмостках (Stage) устанавливается сцена (Scene). Таким образом, подмостки – это контейнер для сцен, а сцена – это контейнер для элементов, составляющих сцену. В результате все приложения JavaFX имеют как минимум одни подмостки и одну сцену. Эти элементы инкапсулируются в API JavaFX классами Stage и Scene. Stage - это контейнер верхнего уровня. Все приложения JavaFX автоматически получают доступ к нему, как главному. Этот доступ предоставляется системой времени выполнения при запуске приложения JavaFX.

Scene – это контейнер для элементов графического интерфейса, составляющих сцену. Отдельные элементы сцены называются узлами, которые могут состоять из других узлов.

Структура и жизненный цикл JavaFX приложения

Приложение JavaFX создает экземпляр класса, унаследованного от класса javafx.application.Application. Объект этого класса проходит ряд этапов жизненного цикла. Все эти этапы представлены методами класса Application, которые вызываются автоматически средой JavaFX:



init(): инициализирует приложение до его запуска (нельзя использовать этот метод для создания графического интерфейса или отдельных его частей).

start(Stage stage): создается графический интерфейс.

stop(): вызывается после закрытия приложения.

В итоге весь процесс работы приложения выглядит следующим образом.

1. Запускает исполняемая среда JavaFX (если она не запущена).
2. Она вызывает конструктор класса, который расширяет класс Application, тем самым создавая экземпляр данного класса.
3. Затем среда JavaFX вызывает метод int().
4. Вызывается метод start(javafx.stage.Stage), в который среда JavaFX передает созданный объект Stage. Таким образом, приложение начинает работать.
5. Далее среда ожидает, пока либо в приложении не будет вызван программным способом метод Platform.exit(), либо пока не будет закрыто последнее окно программы.
6. После завершения работы приложения среда JavaFX вызывает метод stop().

Среда разработки приложений автоматически создает класс Main с методом main запускающим этот цикл вызовом launch() и с «шаблонным» методом start(Stage primaryStage), включающим загрузку файла описания графического интерфейса сцены из текущего доступного ресурса, установку названия сцены и запуск цикла получения сообщений от пользователя.

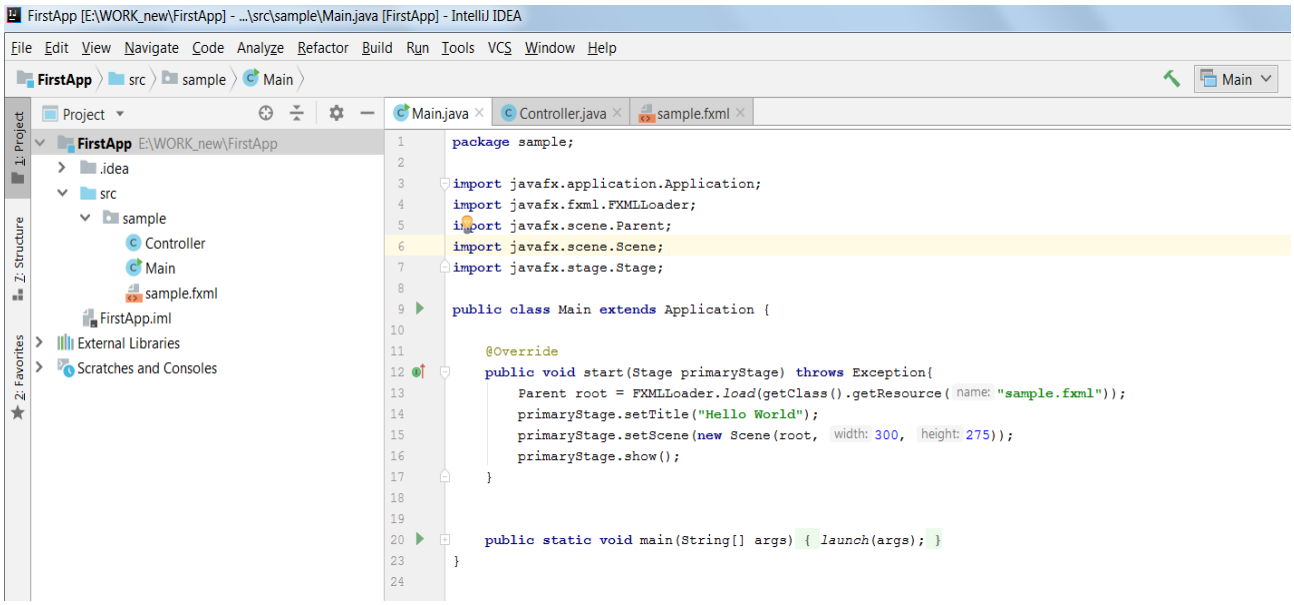
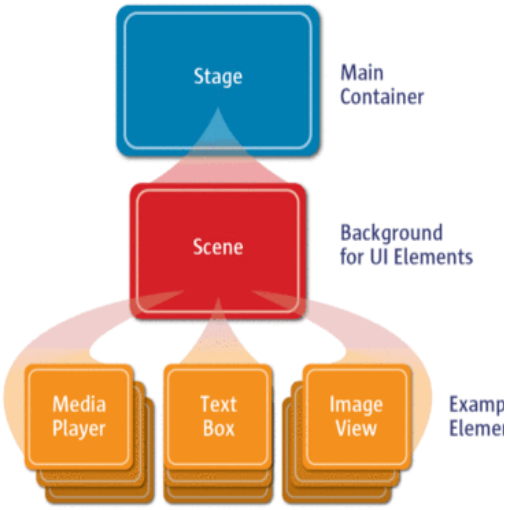


Рис. 1. Автоматически генерируемый каркас приложения JavaFX

Основная часть реализации графического интерфейса сосредоточена в методе start() и отражает структуру контейнеров и элементов, представленную на рис. 2.




```
public void start(Stage priStage){
Parent root =
FXMLLoader.load(getClass().getResourc
e(“sample.fxml"));
priStage.setTitle("Hello World");
priStage.setScene(new Scene(root,
300, 275));
priStage.show();
    }
```

Рис. 2. Иерархия компонентов GUI

Представление Stage (как театральные подмостки) является основным контейнером, который, как правило, представляет собой обрамлённое окно со стандартными кнопками: закрыть, свернуть, развернуть. Внутрь Stage добавляется сцена Scene, которая может быть заменена другой Scene. Внутрь Scene добавляются стандартные компоненты типа контейнеров компоновки AnchorPane, и элементов управления Label, Button, TextBox и другие. Они помещены в корневой узел Parent путем загрузки формы sample.fxml,

содержащей описание размещения компонентов графического интерфейса.

Приложения, созданные в рамках данного практикума должны иметь архитектуру взаимодействия логики приложения, известную как Model-View-Controller (MVC). Концепция MVC позволяет разделить логику обработки данных, их представление и обработку действий пользователя на три отдельных компонента (рис. 3).

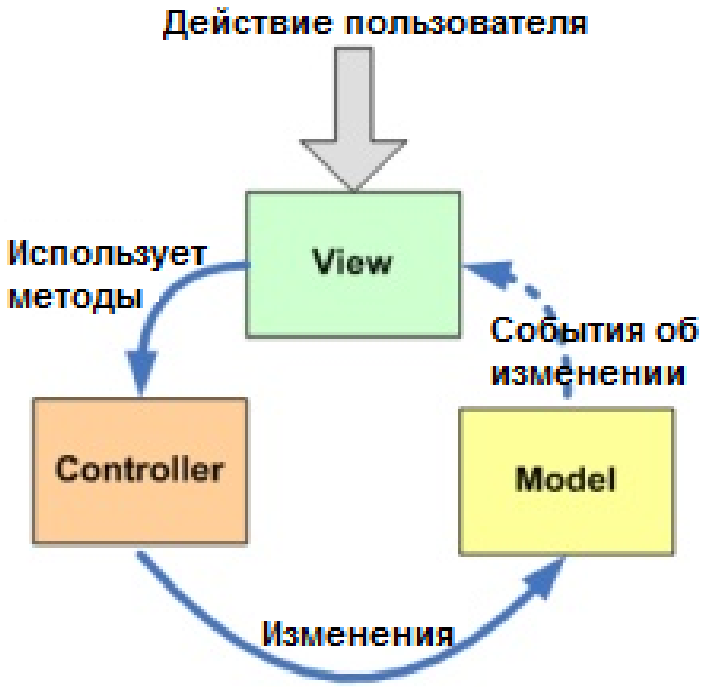


Рис. 3. Архитектурная концепция Model-View-Controller

Модель (Model) предоставляет целевую логику работы приложения: данные и методы работы с этими данными, реагирует на запросы, изменяя своё состояние. Модель легко может быть перенесена в другое окружение, так как не привязана к конкретной реализации и не содержит информации, как получение из модели знания можно визуализировать.

Представление, вид (View) отвечает за отображение информации (визуализацию). Часто его называют клиентская часть, поскольку в качестве представления выступает графический интерфейс пользователя.

Контроллер (Controller) обеспечивает связь между пользователем и программной логикой: контролирует ввод данных пользователем и передает их в модель и представление для реализации необходимой реакции.

Важно отметить, что модель не зависит ни от представления, ни от контроллера. Тем самым достигается назначение такого разделения: оно позволяет строить модель независимо от визуального представления, а также создавать несколько различных, в том числе и визуальных представлений для одной модели.

Для следования концепции MVC структура рабочих проектов должна быть следующей[2]:

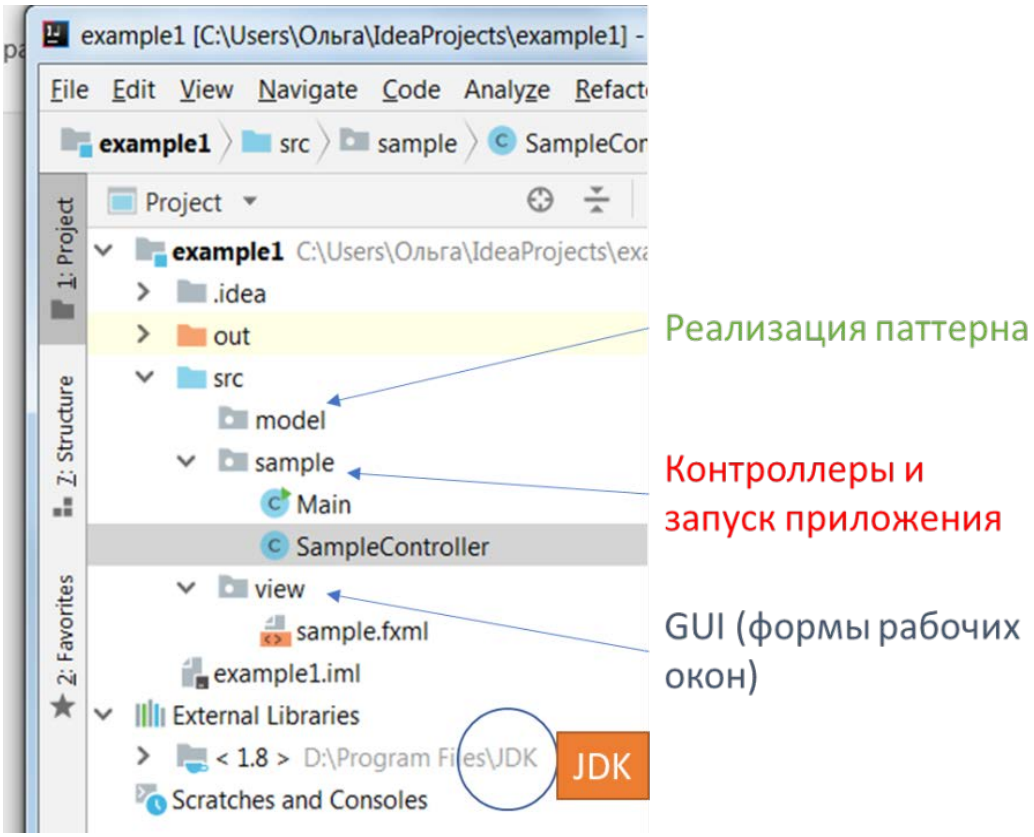


Рис. 4. Структура программного проекта

GUI-представление fx-приложения

Основу GUI в FX составляет сцена (Scene). Она организована в виде древовидного графа, содержащего объекты Node, который называется графом сцены. Граф сцены начинается с корневого узла. Узлы графа - это в основном компоненты окна, а также области макета и компоненты внутри них. Если узел Cildren содержится в другом узле Parent, то Children называется дочерним элементом Parent, а Parent является родительским элементом (рис. 5).

[2] При наличии одной формы представления допускается ее размещение в основной папке.

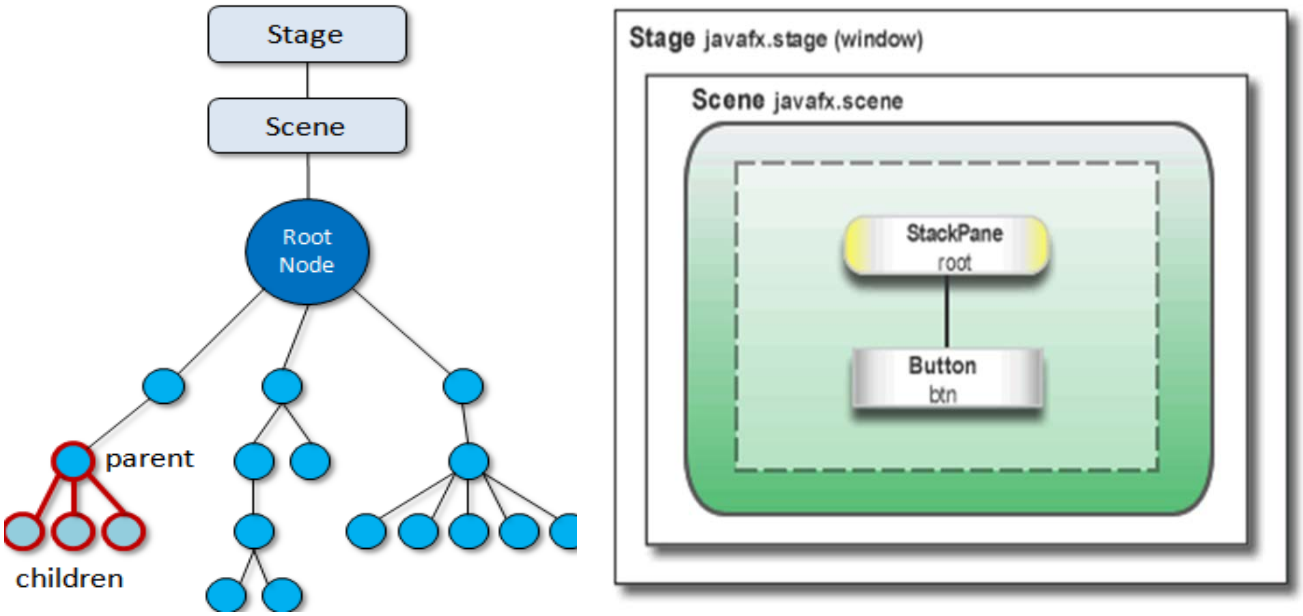
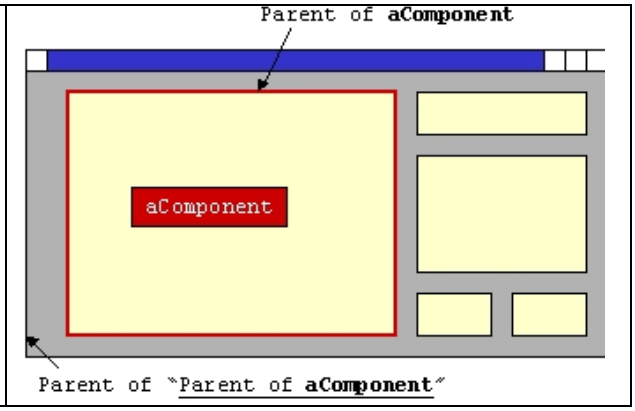


Рис. 5. Структура приложений JavaFX

Родительский узел хранят указатели на потомков, и для доступа к ним используется метод getChildren (), соответственно для доступа к родительскому узлу – getParent(), например:

```
Control aComponent;
Parent p;
Parent parentOfParent;
aComponent = ... ;
p = aComponent.getParent();
parentOfParent =
p.getParent();
```



В JavaFX оконные компоненты, например кнопки, текстовые поля, списки – это элементы управления (Control). Эти компоненты могут группироваться вместе различными способами, которые определяются теми же элементами управления. Различают простые элементы управления и контейнеры (иерархия классов элементов управления на рис. 6).

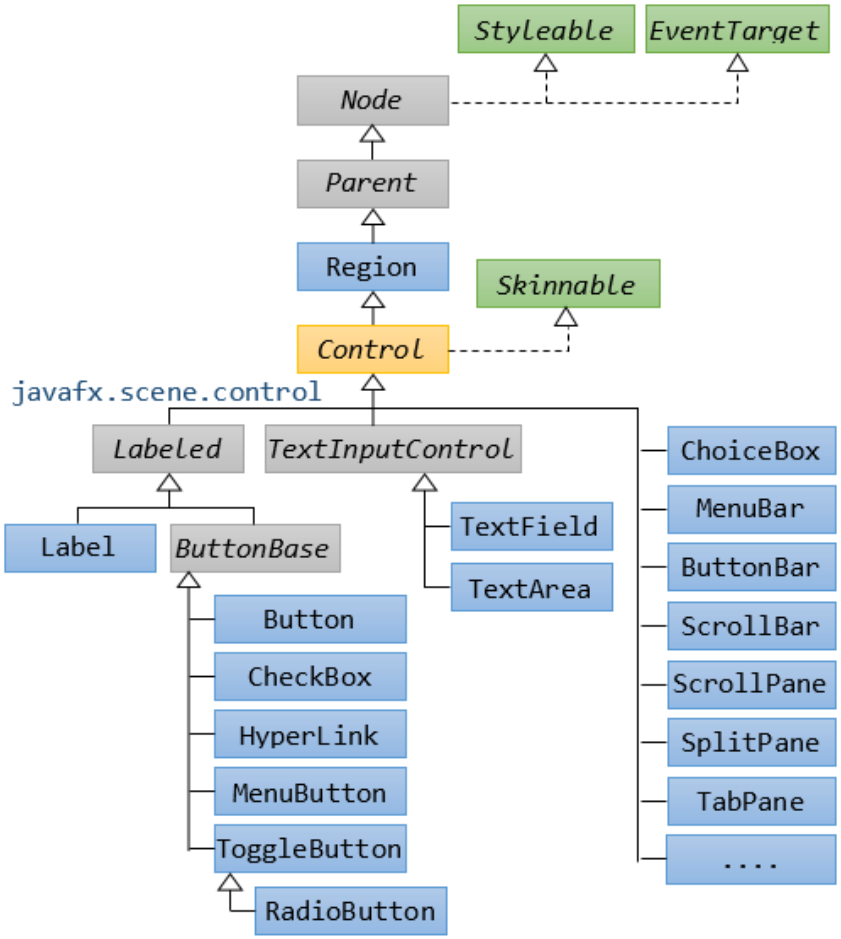


Рис. 6. Элементы графического интерфейса пользователя

Так простейший контейнер в JavaFX – это панель (Pane class), представляющая базовый класс, подклассы которого реализуют различные компоновки элементов, связанные графом сцены (рис. 7).

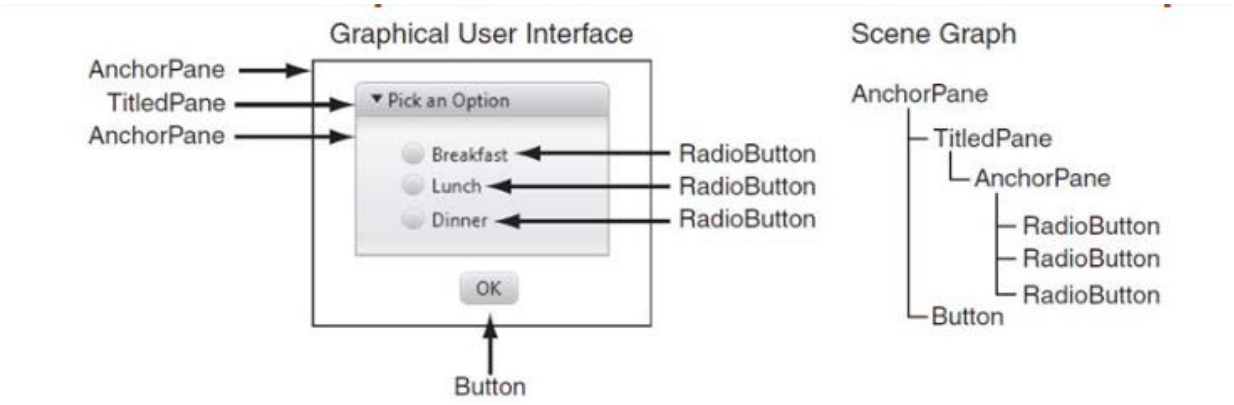


Рис. 7. Внешнее отображение GUI и его граф сцены

Панели компоновки

Панели компоновки управляют процессом размещения элементов управления в сцене, их назначение иллюстрирует рис. 8. Различают статистические и динамические макеты. В статической компоновке положение и размер узлов рассчитываются один раз, и они остаются неизменными при изменении размера окна. Динамический макет перераспределяет положение и размер некоторых или всех входящих в него узлов по мере изменения размера окна.

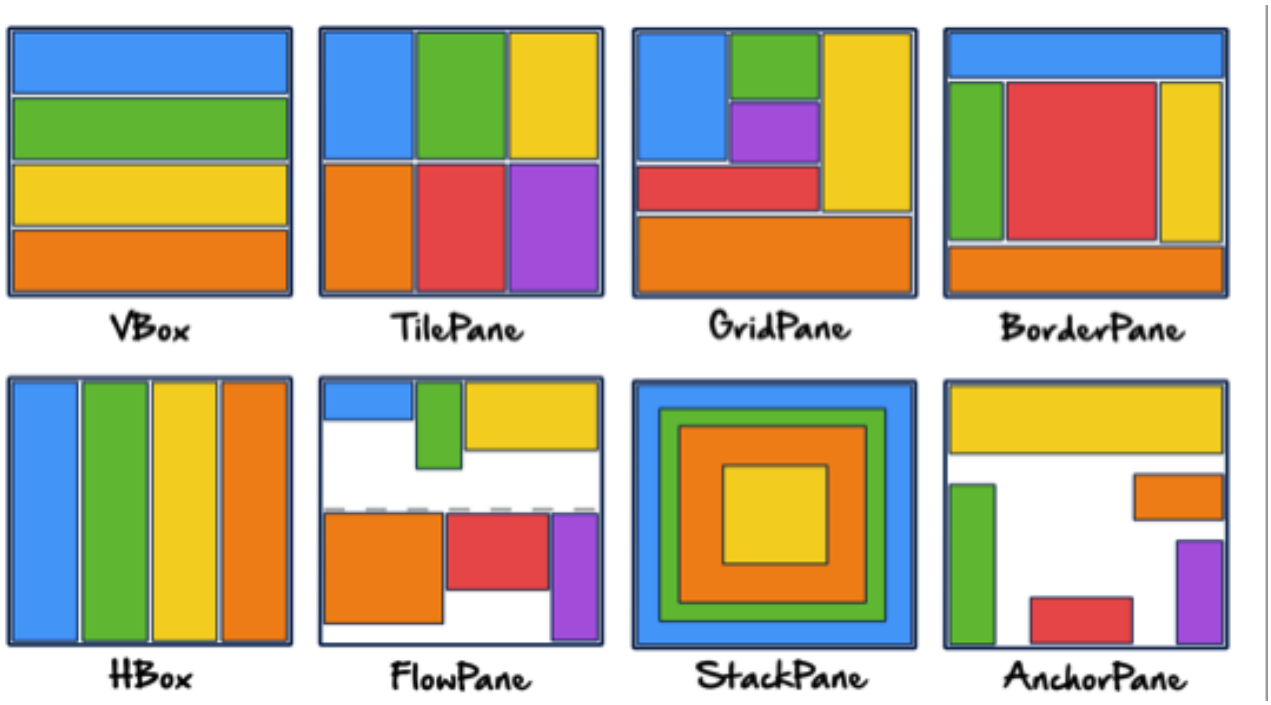


Рис. 8. Виды размещения элементов в различных панелях компоновки

BorderPane - это контейнер (container), который разделен на 5 отдельных областей (рис. 9):

Top и Bottom позволяет сократить/расстянуть по горизонтали и не менять высоту;

Left и Right – сократить/расстянуть по вертикали и не менять длину;

Center – сократить/расстянуть в оба направления.

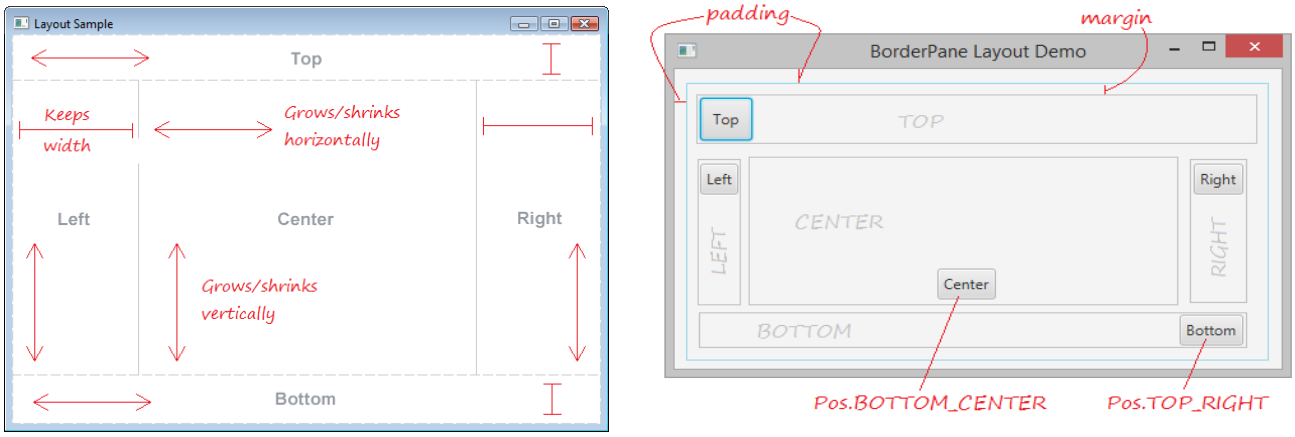


Рис. 9. Контейнер BorderPane

Если определенная область не содержит подкомпонент, другие области будут занимать ее пространство.

Каждый элемент графического интерфейса имеет свойства (рис. 9):

– padding, которое задает отступы элемента, для которого задается свойство, от краев родительского элемента со всех четырех сторон.

– margin, которое определяет отступы для дочерних элементов, которые находятся внутри.

GridPane – это контейнер, который представляет сетку-таблицу, состоящую из строк и столбцов. Дочерний компонент может находиться на ячейке или объединенной группе рядом стоящих ячеек. Ячейки в GridPane нумеруются по двум вертикалям x и y, где x – номер столбца, y – номер строки. Свойство Hgap задает расстояние между содержимым ячеек по горизонтали, Vgap – по вертикали (рис. 10).

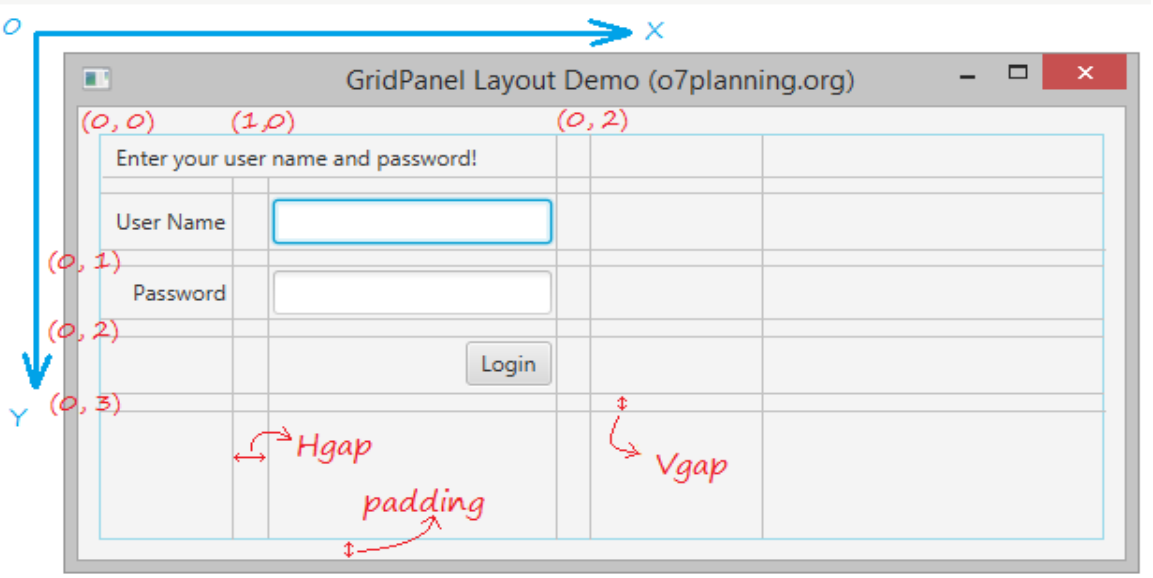


Рис. 10. Структура и расположение элементов в GridPane

Контейнер FlowPane позволяет расставить элементы управления в строку по порядку, и при заполнении строки автоматически перемещает их вниз в следующую строку (рис. 11).

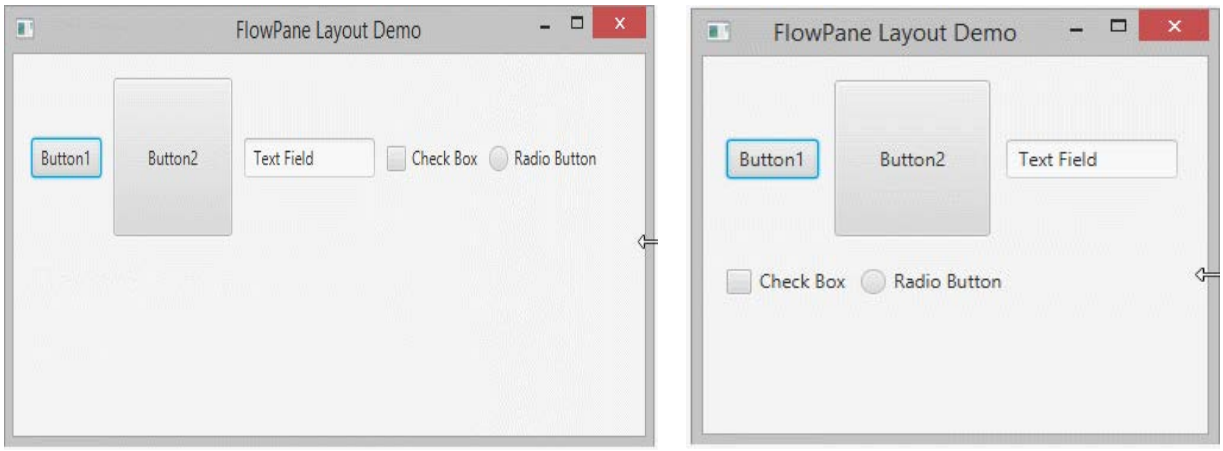


Рис. 11. Заполнение FlowPane при изменении размеров окна

TilePane расставляет дочерние компоненты в последовательном порядке в одной строке, и автоматически перемещает их в следующую строку, если в настоящей строке не имеется больше места. Но в отличие FlowPane выделяет каждому из элементов управления место одинакового размера (рис. 12).

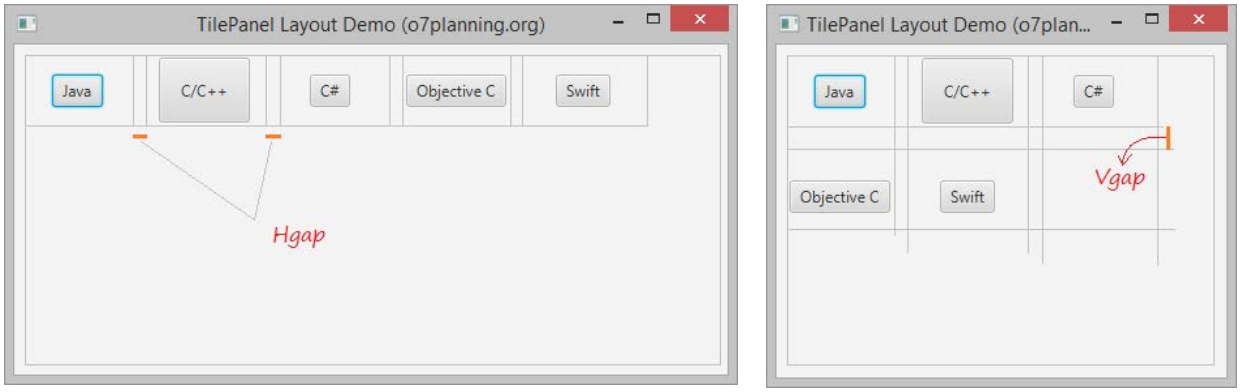


Рис. 12. Размещение элементов в панели TilePane

Контейнер HBox размещает дочерние узлы в одной строке, а контейнер VBox – в столбец (рис. 13). Расстояние между подкомпонентами задаются свойством Spacing. При изменении размера панели элементы будут изменяться свой размер вместе с панелью.

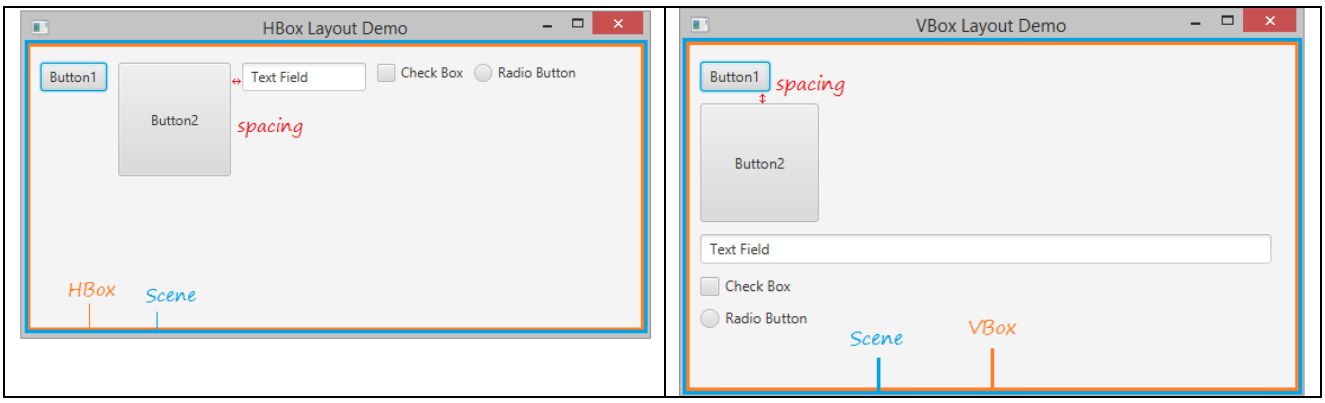


Рис. 13. Размещение элементов на панели HBox и VBox

Контейнер AnchorPane разделяет рабочее пространство на 5 областей (top, bottom, left, right), чтобы внутренний компонент мог поставить точку привязки (anchor - якорь). Для элемента, находящегося в AnchorPane, может быть поставлен якорь в один или более логических областей AnchorPane и определено фиксированное расстояние gap до точки привязки (рис. 14).

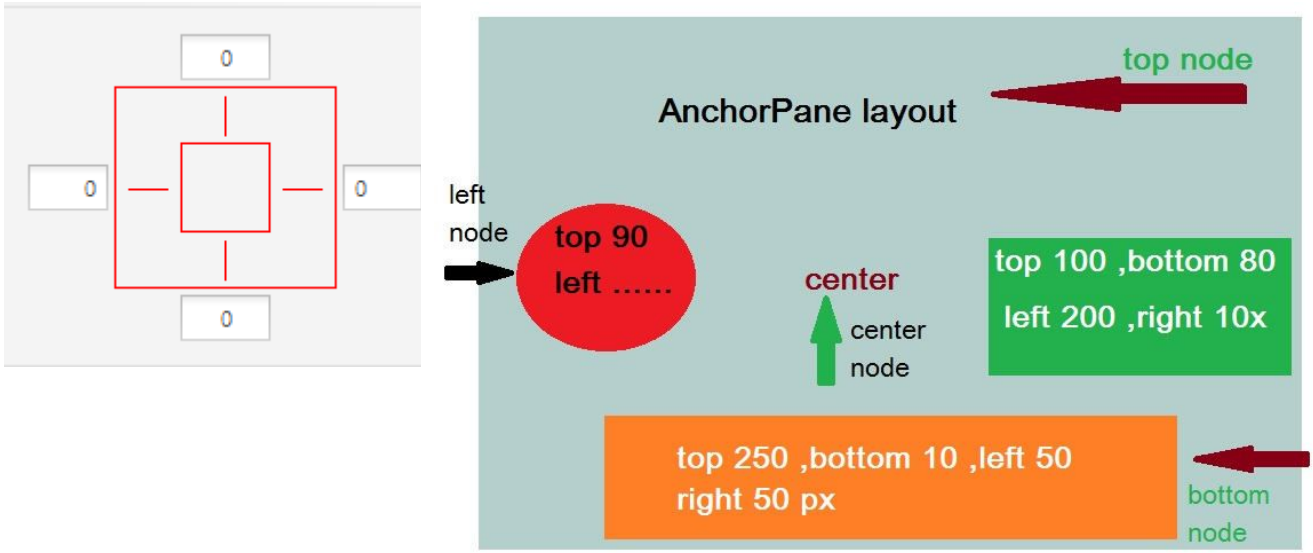


Рис. 14. Закрепление элементов на AnchorPane

Контейнер StackPane может содержать разные компоненты интерфейса, которые будут сформированы в стек, в определенный момент увидеть можно дочерний компонент расположенный вверху.

TitledPane является панелью с заголовком, и данная панель может расширяться (expaned) или сворачиваться (collapsed).

ScrollPane является компонентом прокручиваемого интерфейса (scrol-lable), он используется для отображения большого содержания в ограниченном пространстве. Он имеет горизонтальные и вертикальные полосы прокрутки (scroll bar).

Панели компоновки автоматически упорядочивают вложенные элементы определенным образом и поэтому могут изменять их размеры и положение вложенных элементов вне зависимости от начально установленных разработчиком.

Элементы управления

Label – компонент для отображения текста и/или пиктограммы. Это не интерактивный компонент, пользователь не может влиять на метку. Обычно метка используется для лучшего понимания пользователем назначения другого компонента, и размещается рядом (справа или слева) или в верхней части описываемого узла. На метку нельзя установить фокус, используя клавишу Tab. Элемент управления Label не используется для событий, а лишь для оформления. Ее удобно связывать с каким-либо полем для отражения его состояния:

```
firstNameLbl.setLabelFor(firstNameFld); // привязать Label к компоненту Field
```

TextField – это элемент управления вводом текста, наследуемый от класса TextInputControl. Позволяет пользователю вводить одну строку простого текста. Символы новой строки и табуляции в тексте удаляются. (Для ввода многострочного текста используйте TextArea). Поле ввода требует обработки событий

```
@Override
public void handle(ActionEvent e) {
 messageLbl.setText("Введите что-нибудь!"); }
```

Button – кнопка, которая выполняет команду, когда активирована. Как правило, кнопка имеет текст в качестве метки и зарегистрированный на кнопку обработчик ActionEvent. Кнопка может находиться в одном из трех режимов - normal button (обычная), default button (по умолчанию), cancel button (кнопка отмены). Для обычной кнопки ее ActionEvent запускается, когда кнопка активирована. Для кнопки по умолчанию ActionEvent запускается при нажатии клавиши Enter, и никакой другой узел в сцене не использует нажатие клавиши. Для кнопки отмены ActionEvent запускается при нажатии клавиши Esc, и никакой другой узел в сцене не использует нажатие клавиши (По умолчанию кнопка является обычной кнопкой). Эти режимы являются свойствами кнопки и устанавливаются соответствующими методами:

```
btn.setDefaultButton(true);
btn.setCancelButton(true);
```

ToggleButton – это элемент управления с двумя фиксированными и отображаемыми визуально состояниями, определяемыми свойством selected - выбран = true (выделенная кнопка) и не выбран (не выделенная). Текущее состояние отлеживается через интерфейс ObservableValue (см. CheckBox)

CheckBox – это элемент управления с тремя состояниями:

• Отмечено (True/да);
• Не отмечено (False/нет);
• Не определено (Undefined/неизвестно).

```
CheckBox iCbx = new CheckBox("Undefined");
iCbx.setAllowIndeterminate(true);
```

ComboBox – выпадающий список элементов для выбора одного из них пользователем. Элементы могут быть объектами любого типа (ComboBox - параметризованный класс).

ChoiceBox – выпадающий список элементов, которые могут быть объектами любого типа. Похож на ***ComboBox,*** но в ChoiceBox ставится отметка рядом с выбранным объектом.

Создать список элементов типа – строка (String):

```
ChoiceBox<String> listbox = new ChoiceBox<>();
```

Добавить элементы:

```
listbox.getItems().addAll("Choice1", "Choice2", "Choice3",
"Choice4");
```

ListView используется для создания списка обобщенного типа и выбора одного или нескольких элементов из списка. Каждый элемент в ListView представлен экземпляром класса ListCell, который можно настроить. Список элементов в ListView может содержать объекты любого типа, так как ListView является параметризованным классом.

Примеры отображения часто используемых элементов управления на рис. 15.

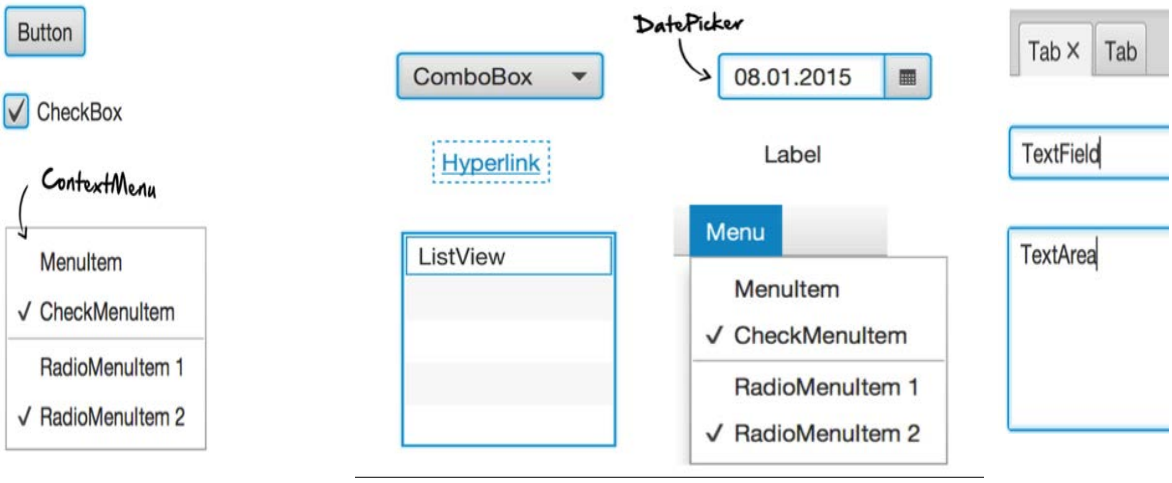


Рис. 15. Внешний вид элементов управления

Описание представления на FXML

Формат XML, специфичный для JavaFX, называется FXML. FXML Поддерживает типичный для XML принцип иерархии и вложенности элементов. В нем вы можете определить все компоненты приложения и их

свойства, а также связать их с контроллером, который отвечает за управление взаимодействиями

Для создания собственной fxml формы необходимо:

1. Создать fxml файл.

2. После создания файла в его первой строке необходимо ввести декларацию XML:

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

3. Добавить операторы импорта:

```
<?import javafx.scene.*?>
```

4. Добавить необходимые компоненты и их атрибуты(свойства), например:

```
<VBox>
    <Label text="Моя первая программа"/>
    <Button text="Решение"/>
</VBox>
```

5. Добавить обработчики события, например onAction для кнопки.

```
<Button fx:id="myButton" text="Решение" onAction="onButClick ()"/>
```

6. Используемые стили разместите в папке проекта Resources, и подключите их к компоненту или всей панели, используя вкладку Свойство (Properties) и пункт Stylesheets

Пример создания fxml-файла с помощью SceneBuilder

В качестве примера рассмотрим создание пользовательского интерфейса приложения для работы с личными делами сотрудников.

Шаг 1. Создание FXML файла

Необходимо щелкнуть правой кнопкой мыши (ПКМ) пакет «**view**» и выберате «**New** | **Other**», затем в открывшемся браузере выбрать «**FXML Document**» (рис. 16) и назвать его «**PersonOverview**», убедитесь, что корневой элемент указан AnchorPane и нажмите «Готово».

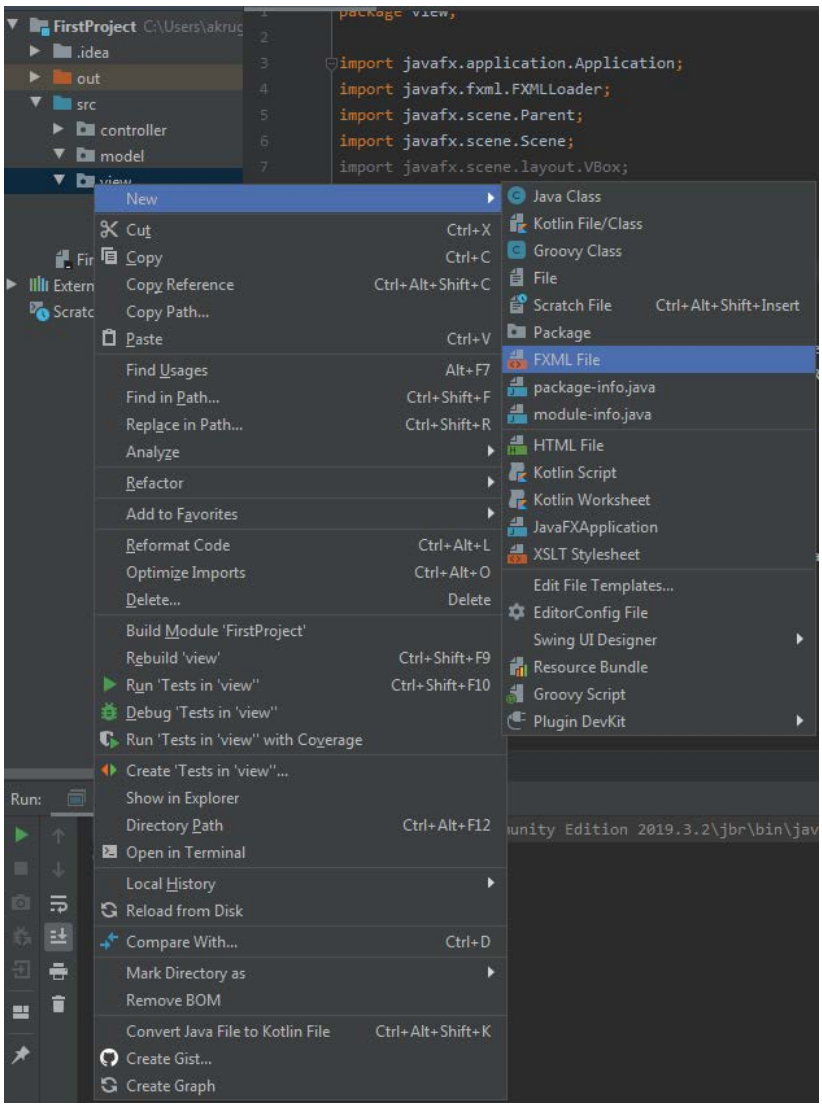


Рис. 16. Создание FXML файла

Далее необходимо открыть созданный fxml-документ в приложении Scene Builder «PersonOverview**.fxml**» и выбрать «Open with SceneBuilder**»**.

Если fxml-документ открылся автоматически в виде «**Text**», то необходим внизу нажать кнопку «Scene Builder» (рис. 17). На вкладке «Hierarchy**»** должен находиться единственный компонент «AnchorPane».

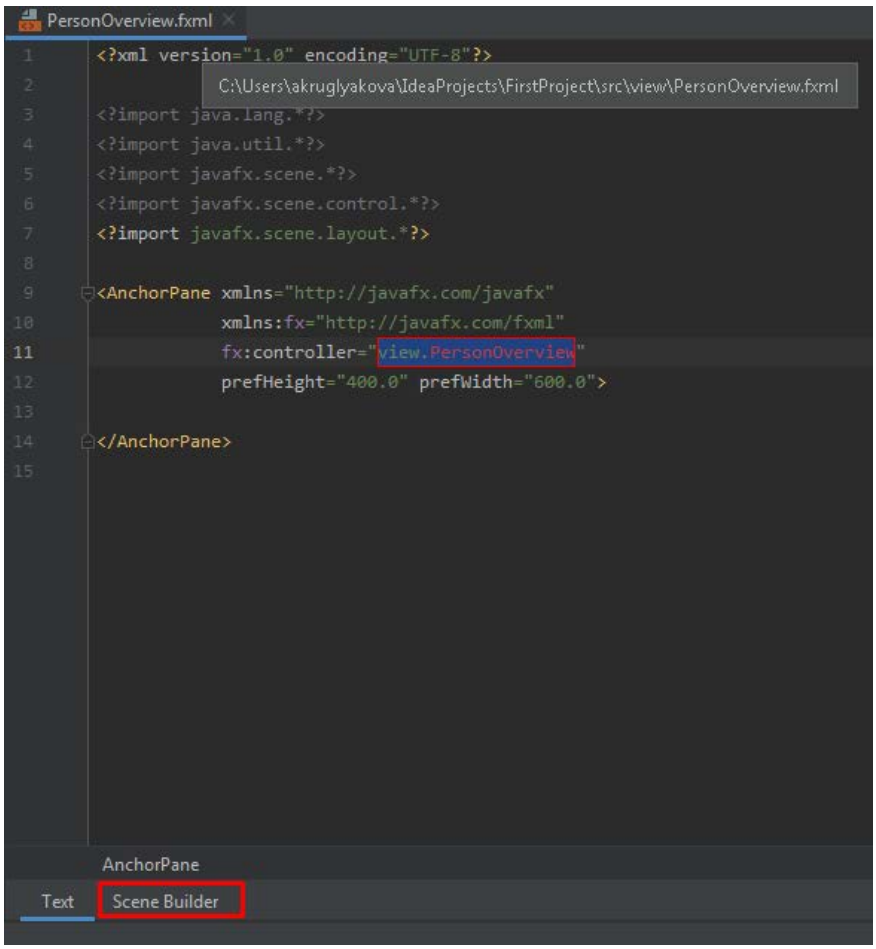


Рис. 17. Открытие fxml-документ в Scene Builder

Шаг 2. Добавление элементов пользовательского интерфейса в Scene Builder

Внутри SceneBuilder ничего нельзя увидеть в центральной области. Однако если выбрать AnchorPane в иерархии документов (слева), теперь можно увидеть перекрестие в центре окна. Это и есть AnchorPane, но с размером 0 x 0. Измените размер AnchorPane можно либо путем перетаскивания в перекрестье, пока не будет найден нужный размер, либо установите «Pref Width and Height» на вкладке «Layout» справа. Необходимо установить значение характеристикам «Pref Width» и «Pref Height» – 600 и 300 соответственно (рис. 18).

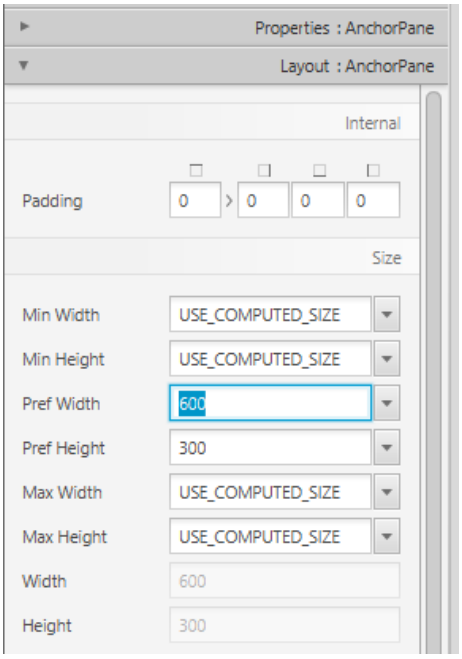


Рис. 18. Задание параметров для AnchorPane

Далее нужно компонент AnchorPane добавить новый компонент «SplitPane (horizontal)», кликнуть по нему правой кнопкой мыши и выберать «Fit to Parent» (рис. 19).

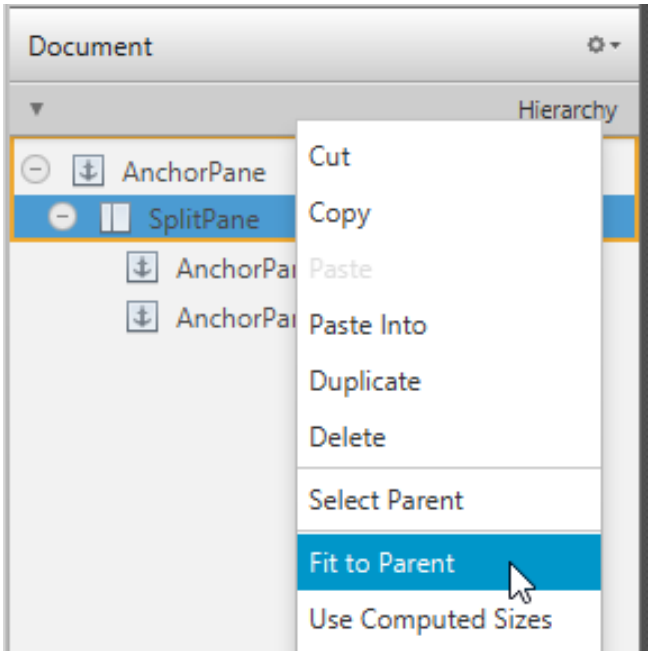


Рис. 19. Добавление компонента SplitPane (horizontal)

Далее в левую часть компонента «SplitPane» со вкладки «Controls» нужно перетащить компонент «TableView». Далее необходимо выделить его целиком (а не отдельный столбец) и проставить отступы от краёв так, как показано на

рисунке. Внутри компонента «AnchorPane» всегда можно проставить отступы от четырёх границ рамки (рис. 18).

Чтобы увидеть, правильно ли отображается созданное окно, необходимо выбрать пункт меню «Preview | Show Preview in Window». Можно попробовать поменять размер окна. Добавленная таблица должна изменяться вместе с окном, так как она прикреплена к границам окна.

В таблице нужно изменить заголовки колонок. Для этого необходимо выбрать нужную компоненту «TableColumn» и задать свойство «Text» на вкладке «Properties» «Имя», для второго столбца – «Фамилия».

Далее нужно выбрать компонент «TableView» и во вкладке «Properties» изменить значение «Column Resize Policy» на «constrained-resize». Выбор этой характеристики гарантирует, что колонки таблицы всегда будут занимать всё доступное пространство.

После в правую часть компонента «SplitPane» нужно перетащить компонент «Label» и изменить его текст на «Персональные данные» (подсказка: для скорейшего нахождения компонентов можно использовать поиск). Используя привязки к границам (вкладка «Layout»), нужно скорректировать его положение (рис. 20).

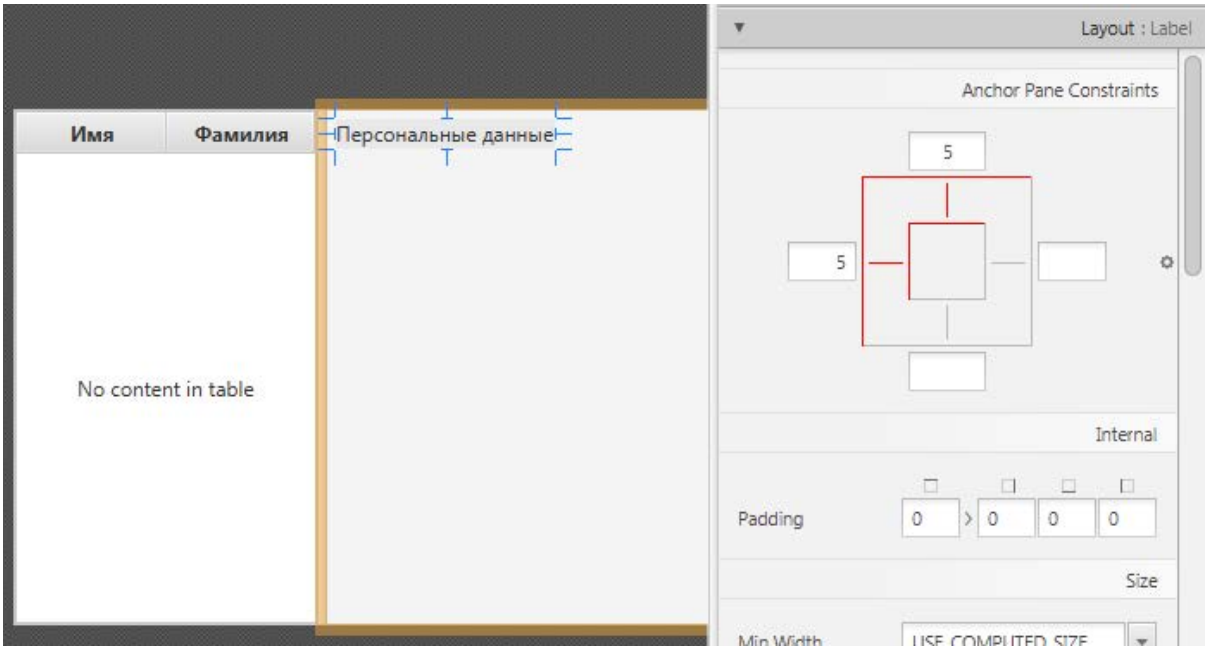


Рис. 20. Настройка привязки Label

На правую панель SplitPane нужно добавить компонент GridPane и так же настроить привязки к границам, Далее необходимо привести своё окно в соответствие с тем, что показано на рис. 21, добавляя элементы Label внутрь ячеек компонента «GridPane».

Примечание: для того, чтобы добавить новый ряд в компонент «GridPane», нужно выбрать существующий номер ряда (он окрасится жёлтым),

кликнуть мышкой на номере ряда и выбрать пункт «Add Row Above» или «Add Row Below».

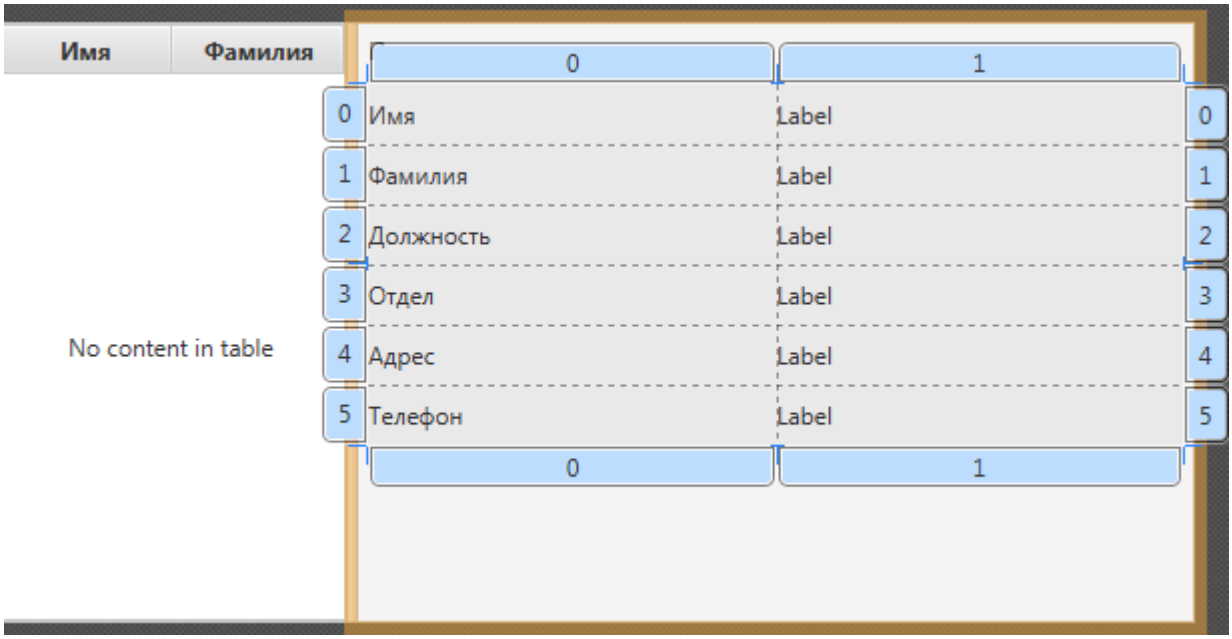


Рис. 21. Настройка панели GridPane

В первом столбце компоновки «GridPane» находятся элементы «Label», для которых в качестве свойства «Text» задан атрибут, описывающий сотрудника, а во втором столбце будет записывать самое значение атрибута. Для того, чтобы в последующем можно было задать какое-то значение для элементов «Label» во втором столбце, нужно задать «fx:id» на вкладке «Code» для каждого элемента.

Внизу необходимо добавить «ButtonBar», а в него три кнопки «Button», как на рис. 22 и установить привязки к границам (правой и нижней), чтобы «ButtonBar» всегда находилась справа.

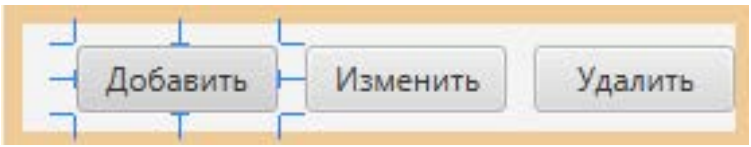


Рис. 22. Элементы Button

Если всё сделано правильно, то должно получится окно, представленное на рис. 23.

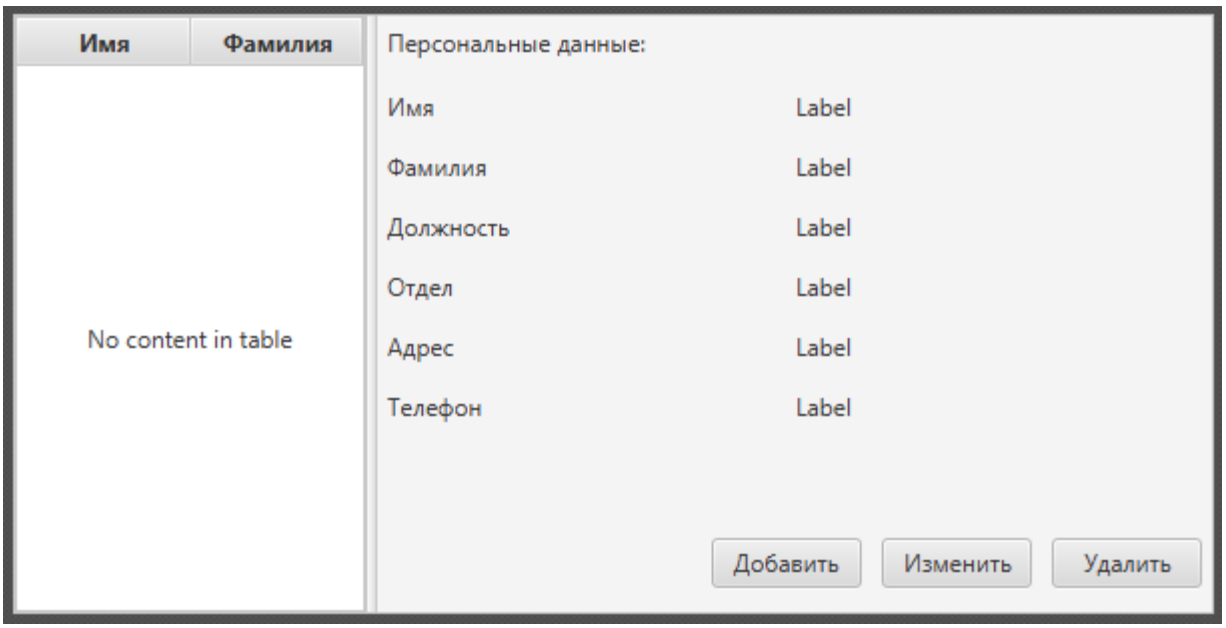


Рис. 23. Готовый документ PersonOverview.fxml

Это же можно проделать через текстовое описание в fxml-файле.

Шаг 1. Определение декларации XML.

После создания файла в его первой строке необходимо ввести декларацию XML:

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

Шаг 2. Импорт библиотек.

Прежде чем добавить отдельные компоненты в файл, необходимо убедиться, что они правильно распознаются. Для этого необходимо добавить операторы импорта. Это очень похоже на импорт в Java классах. Можно также импортировать отдельные классы или использовать знаки подстановки как обычно. Импортируем необходимые нам библиотеки:

```
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
```

Шаг 2. Привязка класса Controller.

Корневому компонентом будет выступать элемент компоновки «AnchorPane». Все остальные элементы будут добавляться на данную панель как дочерние. Для него необходимо задать высоту (`prefHeight`) и ширину (`prefWidth`).

Существует пара элементов и атрибутов FXML, которые по умолчанию недоступны. Например, атрибут позволяющий, привязать данный fxml файл к классу «Controller» (`fx:controller`). Для этого нужно добавить пространство имен (Namespace) FXML (`xmlns="http://javafx.com/javafx/10.0.2-internal"`) (`xmlns:fx="http://javafx.com/fxml/1`) к корневому компоненту, чтобы сделать их доступными.

```
<AnchorPane prefHeight="300.0" prefWidth="600.0"
```

```
xmlns="http://javafx.com/javafx/10.0.2-internal"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="controller.Controller">

</AnchorPane>
```

Шаг 3. Добавление компонентов

Далее необходимо добавить элемент «SplitPane», он будет являться родительским для всех последующих элементов. Так как элемент «SplitPane» разделяется на 2 половины, то на него необходимо добавить еще элемент «AnchorPane». Элементы, которые будут помещаться на левую часть панели «SplitPane» добавлять между первыми тегами `children`, на правую – между вторыми тегами `children`.

```
<SplitPane dividerPositions="0.29797979797979796" prefHeight="300.0" prefWidth="600.0"
AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0" AnchorPane.rightAnchor="0.0"
AnchorPane.topAnchor="0.0">
   <items>
      <AnchorPane minHeight="0.0" minWidth="0.0" prefHeight="160.0" prefWidth="100.0">
         <children>

         </children>
      </AnchorPane>
      <AnchorPane minHeight="0.0" minWidth="0.0" prefHeight="160.0" prefWidth="100.0">
         <children>

         </children>
      </AnchorPane>
   </items>
</SplitPane>
```

Следующий фрагмент кода создает панель ButtonBar с тремя кнопками, на каждую кнопку повешено событие onAction.

```
<ButtonBar layoutX="14.0" layoutY="210.0" prefHeight="40.0" prefWidth="200.0"
AnchorPane.bottomAnchor="5.0" AnchorPane.rightAnchor="5.0">
  <buttons>
    <Button mnemonicParsing="false" onAction="#addEmploy" text="Добавить" />
      <Button mnemonicParsing="false" onAction="#editEmploy" text="Изменить" />
      <Button mnemonicParsing="false" onAction="#deleteEmploy" text="Удалить" />
  </buttons>
</ButtonBar>
```

Обработка событий в JavaFX

Для взаимодействия с пользователем в JavaFX используется событийная модель, в которой участвуют источник события – некоторый элемент управления, который генерирует событие, и есть один или несколько обработчиков события, которые подписываются на событие. Событие – это действие пользователя, например щелчок по кнопке или сочетание неких условий, например срабатывание таймера. Обработчик события – это функция, программный код, который вызывается элементом, когда он генерирует событие (рис. 24).

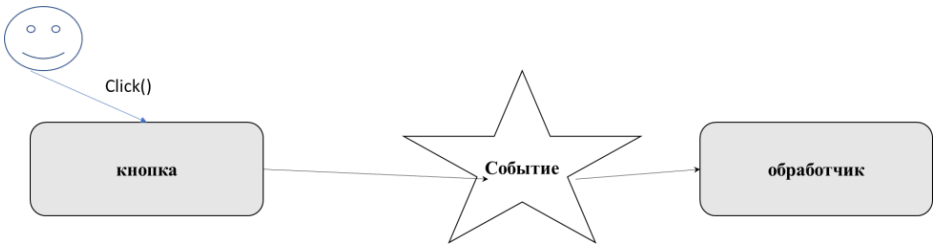


Рис. 24. Взаимодействие пользователя с программой

Базовым классом для событий JavaFX является класс Event, из пакета javafx.event. Событие наследует java.util.EventObject, что означает, что события JavaFX имеют те же базовые функции, что и классические события Java. Когда событие JavaFX генерируется, оно инкапсулируется в объект события. Событие поддерживает несколько методов, которые помогают управлять событиями, например, возможность получить источник события и тип события. В javafx.event определено несколько подклассов Event, которые представляют различные типы событий.

События обрабатываются путем реализации интерфейса EventHandler, который также находится в javafx.event. Это общий интерфейс следующей формы:

```
Interface EventHandler<T extends Event>
```

T – тип события, которое обработчик будет обрабатывать.

Он определяет один метод, называемый handle (), который получает объект Event в качестве аргумента.

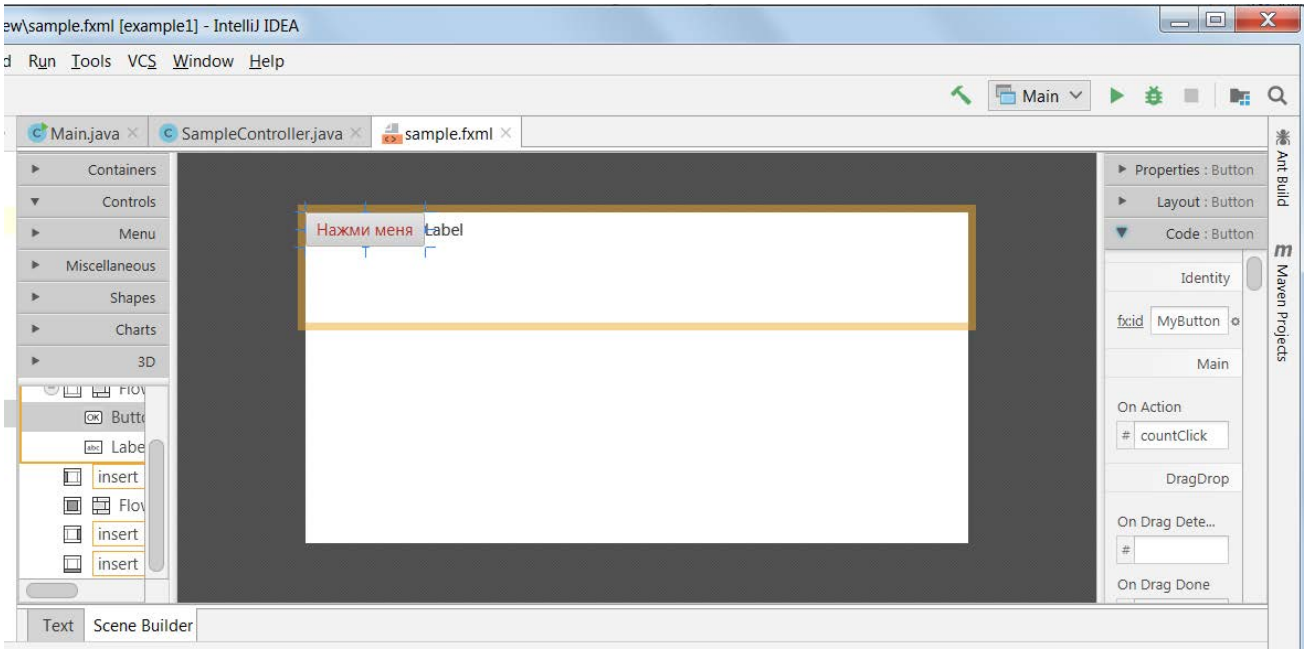
```
void handle(T eventObj)
```

где eventObj – это сгенерированное событие. Обычно обработчики событий реализуются через анонимные внутренние классы или лямбда-выражения.

Для создания связи между элементом управления и обработчиком следует:

1. Присвоить элементу, с которым вы собираетесь работать в коде, идентификатор **fx:id**.

А) В редакторе SceineBuilder выбрать элемент управления например, кнопка. Справа внизу выбрать вкладку настроек Code и в поле fx:id ввести имя элемента для его идентификации в коде программы;



Б) в fxml-файле в тегах выбранного элемента управления прописать:

```
<Button fx:id="MyButton"
```

2. Написать метод для обработки события в классе Controller.

Например:

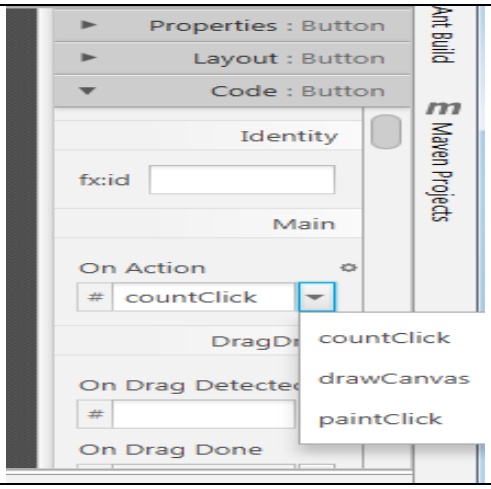
```
public void countClick(ActionEvent actionEvent) {
   Count++;
   myLabel.setText("нажато "+ Count+" раз");
}
```

3. Связать метод обработчика события с компонентом-генератором события:

А) В редакторе SceineBuilder выбрать элемент управления например, кнопка. Справа внизу выбрать вкладку настроек Code и в поле события которое следует обрабатывать выбрать метод обработчика.

Б) в fxml-файле в тегах выбранного элемента управления прописать:

```
<Button onAction="#countClick"
```



! обратите внимание, у метода onAction аргумент actionEvent, а у onMouseClick аргумент mouseEvent

Если связи формы и контроллера не произошло проверьте что в корне fxml-описания присутствует правильная ссылка на класс контроллера

<BorderPane xmlns="http://javafx.com/javafx/8.0.121" xmlns:fx="http://javafx.com/fxml/1" ***fx:controller="имяПакета.Controller"*** *>*

Пример разработки fx-приложения Процентный калькулятор

Задание

Проблема. Получая чек в ресторане, Вы хотите дать «правильные» чаевые в зависимости от Вашего настроения – большие 15%, обычные 10% или «мелочь» 3%.

Требуется написать программу, которая в зависимости от введенной итоговой суммы чека вычисляет, сколько Вы должны заплатить с учетом желаемых чаевых (чаевые зависят от настроения).

Решение

1. Построение интерфейса пользователя с помощью ScienceBuilder в соответствии с раскадровкой.

2. Создание модели расчета, например, класс Procent, содержащий:

– метод double countPr(double sum, int pr) – возвращает значение заданного процента pr от суммы sum;

– метод double countSum(double sum, int pr) – возвращает значение итоговой суммы с учетом заданного процента pr от суммы sum;

– метод int countSumTrunc(double sum, int pr) – возвращает округленное до ближайшего целого значение итоговой суммы с учетом заданного процента pr от суммы sum;

– приватные поля для хранения значений.

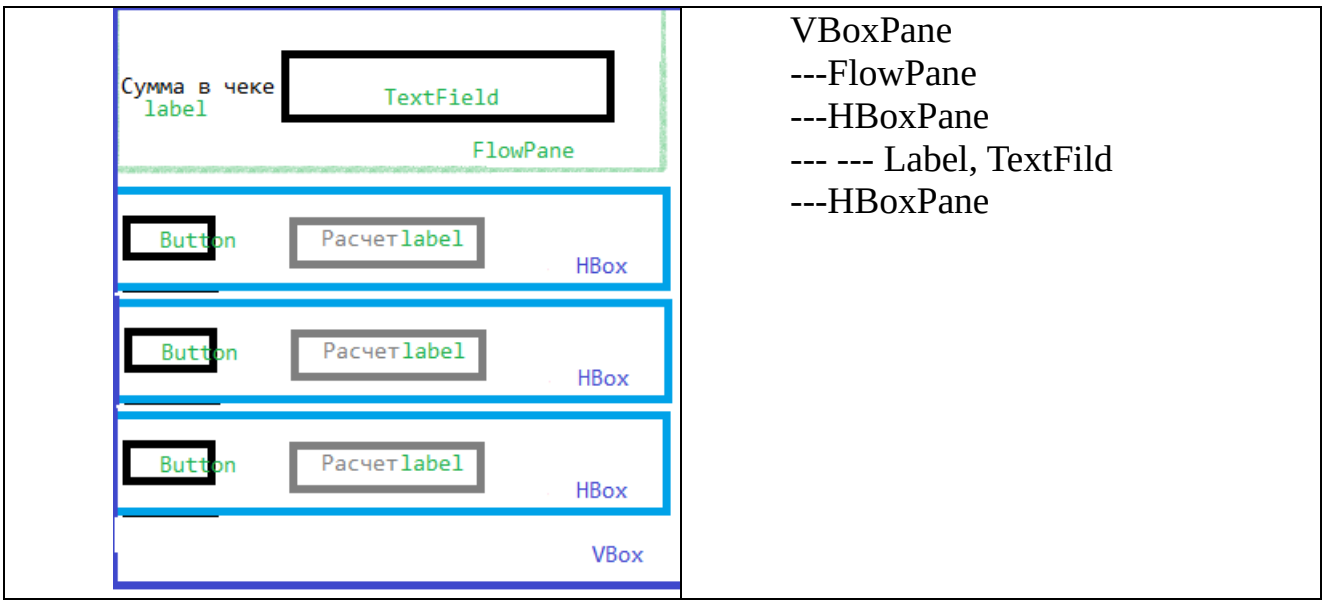


Рис. 25. Компоновка элементов графического интерфейса пользователя

3. Реализация методов обработки событий GUI через **делегирование** объекта класса созданной модели.

Делегирование представляет собой более общий подход к решению задачи расширения возможностей поведения класса. Этот подход заключается в

том, что некоторый класс вызывает методы другого класса, а не наследует их.

Делегирование решает более общие задачи, чем наследование. Любое расширение класса, которое может быть выполнено при помощи наследования, может также быть выполнено при помощи делегирования.

Реализовать делегирование несложно. Для этой цели просто используется ссылка на экземпляр класса. Диаграмма на рис. 26 показывает, что класс, выступающий в роли Delegator, использует класс, выступающий в роли Delegatee.

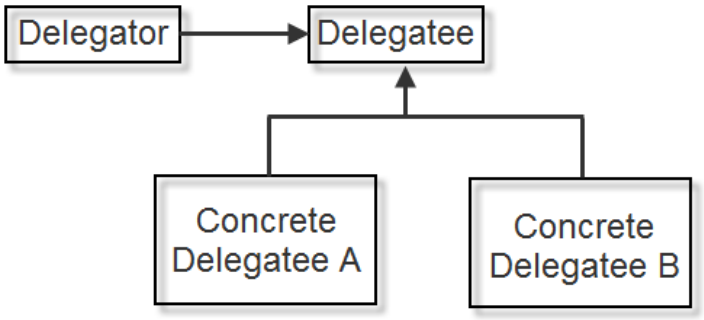


Рис. 26. Шаблон Делегирование

Пример кода:

```
public class Controller {
    //объявление глобальных переменных
    public Button big_ch;
    public Label big;
    public Button sred_ch;
    public Label sred;
    public TextField summa;
    public Label mal;
    public  double r;

   Procent procent=new Procent();//модель

    public void m_mal_ch(MouseEvent mouseEvent) {//метод для мал
чаевых
     mal.setText(("Сумма: " +
procent.countSumTrunc(Double.parseDouble(summa.getText()), 3)));

    }

    public void m_big_ch(MouseEvent mouseEvent) {//расчет больших
чаевых
mal.setText(("Сумма: " +
procent.countSumTrunc(Double.parseDouble(summa.getText()), 15)));
}
```

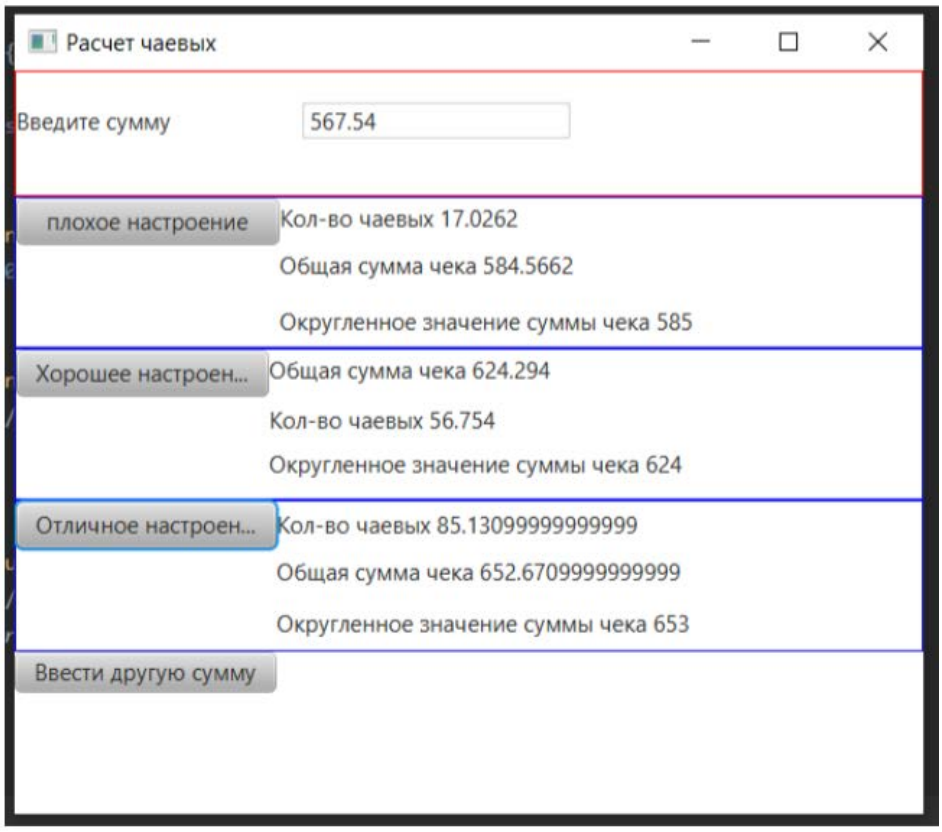


Рис. 27. Пример созданного приложения

ШАБЛОН 1. АБСТРАКТНЫЙ СУПЕРКЛАСС

Цель применения шаблона (паттерна): Гарантировать, что поведение взаимосвязанных объектов будет одинаковое.

Проблема. Разрабатывается система автоматизированного проектирования зданий и сооружений. Базовым типом каждой для каждой детали или строительной конструкции является «форма», и каждая форма имеет цвет, размер, идентификатор и другие характеристики. Все наследуемые конкретные типы фигур - блок, кирпич, стена и прочие, могут иметь дополнительные характеристики и поведение. Их характеристики могут дополняться, а поведение может отличаться, например, формула вычисления площади поверхности каждой конкретной фигуры, ее отображение - различно. Иерархия типов воплощает в себе как сходства, так и различия между формами.

Описание

Общая логика связанных классов определяется в суперклассе. Варианты поведения, зависящие от конкретного наследника, размещаются в методах с одинаковой сигнатурой и эти методы абстрактные.

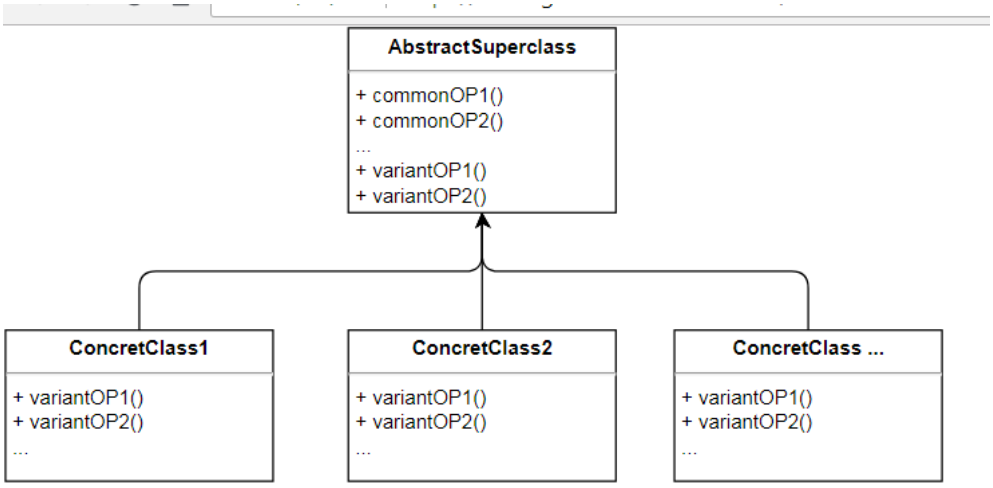


Рис. 28. Диаграмма классов шаблона Абстрактный суперкласс

Abstract Superclass – представляет собой абстрактный суперкласс, в котором инкапсулирована общая логика связанных классов. Связанные классы расширяют этот класс. Таким образом, они могут наследовать его методы. Методы с одинаковыми сигнатурами и общей логикой для всех связанных классов помещаются в суперкласс, поэтому логика этих методов может наследоваться всеми подклассами данного суперкласса. Методы с зависящей от конкретного подкласса данного суперкласса логикой, но с одинаковыми сигнатурами, объявляются в абстрактном классе как абстрактные методы, тем самым гарантируя, что каждый конкретный подкласс будет иметь методы с такими же сигнатурами.

ConcreteClassl, ConcreteClass2 и т.д. представляют собой конкретный класс, чья логика и назначение связаны с другими конкретными классами. Методы, общие для этих связанных классов, помещаются в абстрактный суперкласс.

Задание и указания по выполнению

Постановка задачи

Написать программу, которая реализует иерархию для отображения фигур и позволяет отобразить указанную на кнопке пользовательского интерфейса фигуру.

Указания по выполнению

1. Разработайте пользовательский интерфейс в соответствии с представленной раскадровкой (0).

2. Создайте абстрактный суперкласс.

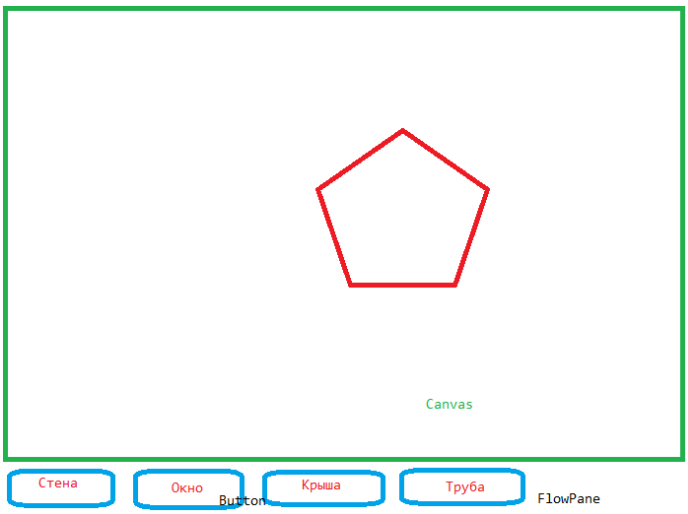


Рис. 29. Представление GUI

```
abstract class Shape  {
    //параметры фигуры - приватные поля
    Color color;
    // объявление абстрактных методов
    abstract double area();
    public abstract String toString();
    abstract void draw( GraphicContext gr);
    // конструктор
    public Shape(String color) {
        System.out.println("Shape constructor called");
        this.color = color;     }
    // реализация методов
    public String setColor(Color color) {
        this.color=color;     }
}
```

3. Реализуйте конкретные классы для каждой фигуры:

```
class Rectangle extends Shape{
    double length;
   double width;
    public Rectangle(String color,double length,double width) {
        // calling Shape constructor
        super(color);
        this.length = length;
        this.width = width;  }
 @Override
    double area() {
        return length*width;   }
  @Override
    public String toString() {
        return "Rectangle color is " + super.color +  "and area is
: " + area();   }
  }
```

4. Реализуйте методы draw(GraphicContext gr) для рисования фигур

(подробно см. следующий подраздел)

5. Реализуйте обработчик для каждой кнопки графического интерфейса, выполняющий рисование соответствующей фигуры.

FX: Рисование по точкам

Пакет javafx.scene.canvas JavaFX предоставляет API Canvas, который предлагает поверхность (холст) для рисования фигур, изображений и текста с помощью методов. API состоит только из двух классов:

• Canvas (холст);

• GraphicsContext (графический контент).

Canvas – это растровое изображение, которое используется в качестве поверхности для рисования. Экземпляр класса Canvas представляет холст, наследуемый от класса Node. Следовательно, Canvas – это узел сцены. Его можно добавить в граф сцены, и к нему можно применить эффекты и преобразования. Холст имеет графический контент (GraphicsContext), связанный с ним, который используется в методах рисования на холсте.

GraphicsContext имеет смысл пера, которое выполняет рисование, поэтому объект Canvas canvas

1) должен быть определен (прописан в fxml-форме или создан как объект
```
Canvas canvas = new Canvas(400, 200););
```

2) добавлен к панели явно (root.getChildren().add(canvas);) или неявно (автоматически fxml скриптом);

3) для рисования следует явно получить графический контент для рисования на нем:
```
GraphicsContext gc = canvas.getGraphicsContext2D();
```

4) и только получив GraphicsContext объект можно приступить к отрисовке конкретными методами, например
```
gc.strokeText("Hello Canvas", 150, 100);// написать текст
```

Рисование на GraphicsContext объекте выполняют методы:

fillXxx (), который закрашивает фигуру текущей заливкой, предварительно установленной, как `gc.setFill(Color.RED);`

strokeXxx () рисует форму с текущим контуром заранее определённым методом `gc.setStroke(Color.RED)`.

Методы рисования на канве

Фигура	Обозначение (Xxx)
ДУГА ЗАМКНУТАЯ	Arc()
ОВАЛ	Oval()
ПОЛИГОН	Polygon()
ПРЯМОУГОЛЬНИК	Rect()
скругленный ПРЯМОУГОЛЬНИК	RoundRect()

При рисовании сложных контуров, необходимо задать конкретные точки и виды соединений и в завершении вызвать метод для заливки или прорисовки контура, например:

```
gc.beginPath();// Начало контура фигуры
gc.moveTo(50, 50); // Установка начальной точки
gc.quadraticCurveTo(30, 150, 300, 200); // Задание координат
кривой контура фигуры
gc.fill();// Заполнение контура
gc.closePath();//Конец контура фигуры
```

Рисунки, которые выходят за пределы Canvas обрезаются. Canvas использует буфер. Команды рисования передают необходимые параметры в этот буфер. Как только Canvas добавлен в граф сцены, графический контекст используется только в потоке приложений JavaFX.

Пошаговый пример реализации приложения «Элементы BPMN»

Программа «Элементы BPMN» предназначена для знакомства пользователя с элементами BPMN. Она реализует следующий сценарий:

1. Пользователь нажимает на одну из кнопок (событие, задача, шлюз, управление, смс).
2. Программа рисует заданный элемент BPMN.

Шаг 1. Создание абстрактного суперкласса Shape.

```
public abstract class Shape {
    Color color;
    public abstract String toString();
    public abstract void draw(GraphicsContext gr);
    public Shape(Color color) {
        System.out.println("Shape constructor called");
        this.color = color;     }
    public void setColor(Color color) {
       this.color=color;
    }
}
```

Шаг 2. Реализация конкретных классов для каждого элемента BPMN (Поток сообщений, Управляющий поток, Шлюз, Событие, Задача)

Класс Поток Сообщений:

```
public class SMS_R extends Shape{
    double length;
    double width;

    public SMS_R(Color color, double length, double width) {
        super(color);
        this.length = length;
        this.width = width;
    }
    @Override
    double area() {
```

```
        return length*width;
    }
    @Override
    public String toString() {
        return"Rectangle color is " + super.color +
                "and area is : " + area();
    }
    @Override
    public void draw(GraphicsContext gr) {
       gr.setFill(Color.WHITE);
       gr.fillRect(10,10,500,500);
        gr.setLineDashes(2);
        gr.strokeLine(70,100,170,100);
        gr.strokeLine(170,100,168,103);
        gr.strokeLine(170,100,168,97);
    }
```

Шаг 3. Разработка механизма обратки нажатий на кнопки и описание необходимых действий в Controller

```
public void R_SMS(ActionEvent actionEvent) {
        gc = canvas.getGraphicsContext2D();
        SMS_R r_r= new SMS_R(Color.GREEN,10,20);
        r_r.draw(gc);
        label_1.setAlignment(Pos.CENTER);
        label_1.setText("Поток сообщений");
    }

    public void R_UPRAV(ActionEvent actionEvent) {
        gc = canvas.getGraphicsContext2D();
        UPRAV_R UPRAV_r = new UPRAV_R(Color.GREEN,10,20);
        UPRAV_r.draw(gc);
        label_1.setAlignment(Pos.CENTER);
        label_1.setText("Управляющий поток");
    }

    public void R_SHL(ActionEvent actionEvent) {
        gc = canvas.getGraphicsContext2D();
        SHL_R SHL_r = new SHL_R(Color.GREEN,10,20);
        SHL_r.draw(gc);
        label_1.setAlignment(Pos.CENTER);
        label_1.setText("Шлюз");
    }

    public void R_SOB(ActionEvent actionEvent) {
        gc = canvas.getGraphicsContext2D();
        SOB_R SOB_r = new SOB_R(Color.GREEN,10,20);
        SOB_r.draw(gc);
        label_1.setAlignment(Pos.CENTER);
        label_1.setText("Событие");
    }
```

Пример работы созданного приложения на рис. 30.

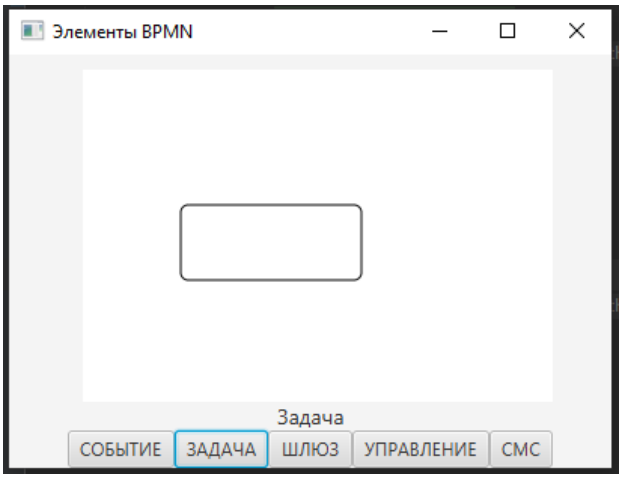


Рис. 30. Результат программы, при нажатии кнопки «ЗАДАЧА»

ШАБЛОН 2. ФАБРИЧНЫЙ МЕТОД

Цель применения: Создавать новые объекты согласно определенному интерфейсу общему для различных видов объектов.

Проблема. В CAD системе необходимо порождать новые элементы и отображать и работать с ними одинаковым образом, несмотря на их внешнее различие.

Описание паттерна

Фабричный метод (фабрика) – это порождающий паттерн проектирования. Фабричный метод генерирует на выходе объекты разных классов (типов), но которые реализуют заранее оговоренный интерфейс. Конкретный класс генерируемого объекта фабрика определяет по дополнительным условиям.

Для того чтобы система оставалась независимой от различных типов объектов, паттерн Factory Method использует механизм полиморфизма — классы всех конечных типов наследуют от одного абстрактного базового класса, предназначенного для полиморфного использования. В этом базовом классе определяется единый интерфейс, через который пользователь будет оперировать объектами конечных типов.

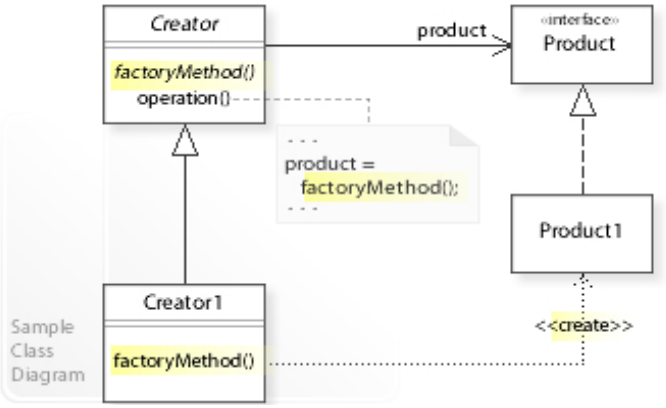


Рис. 31. Диаграмма классов паттерна Фабричный метод

Для обеспечения относительно простого добавления в систему новых типов паттерн Factory Method локализует создание объектов конкретных типов в специальном классе-фабрике. Методы этого класса, посредством которых создаются объекты конкретных классов, называются фабричными.

Задание и указания по выполнению

Постановка задачи

Требуется написать программу, которая рисует фигуру по заданному пользователем числу сторон (0-круг, 1-отрезок, 2-угол, 3-треугольник и т.д).

Указания по выполнению

1. Разработайте пользовательский интерфейс в соответствии с представленной раскадровкой (рис. 32)

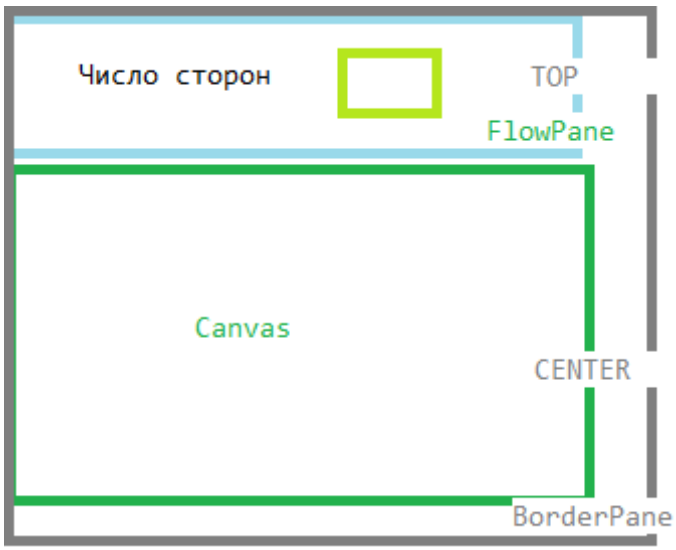


Рис. 32. Представление GUI

2. Создайте Модель (рис. 33), которая должна состоять из классов создания отдельных фигур и Фабричного метода.

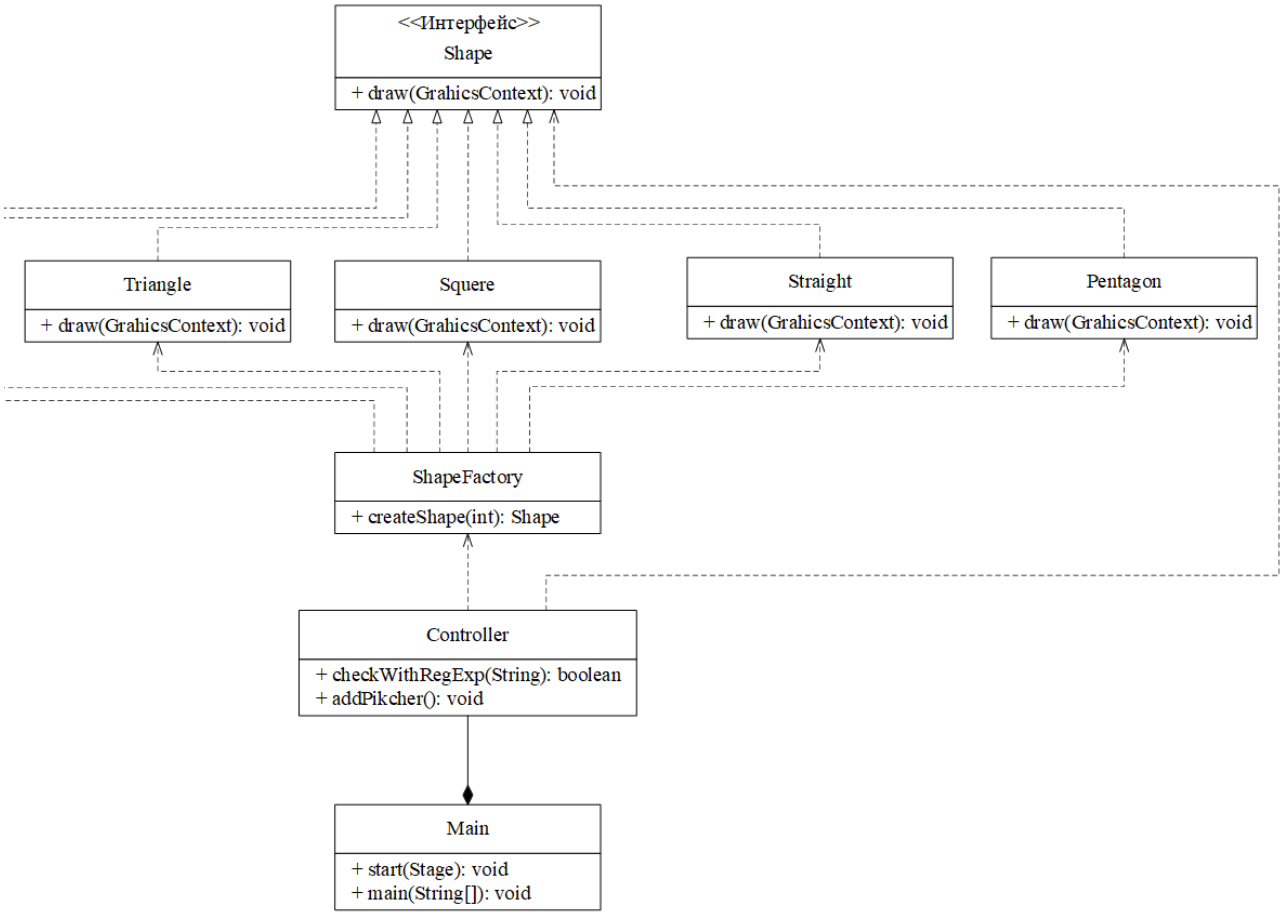


Рис. 33. Диаграмма классов прикладного решения

Каждая фигура должна быть представлена отдельным классом, унаследованным от класса Shape, содержащего абстрактные методы:

draw() – для рисования фигуры;

descriptor() – для вывода названия фигуры;

приватные поля, задающие цвет заливки и контура фигуры.

3. Реализуйте Фабричный метод для создания различных фигур:

```
public class ShapeFactory {
    public Shape createPolygon(int numberOfSides) {
        if (numberOfSides == 3) {
            return new Triangle();
        } else if (numberOfSides == 4) {
            return new Square();
        } else if (numberOfSides == 5) {
            return new Pentagon();
```

Наиболее элегантным решением для задания определения вид создаваемого объекта является использование хэш-таблицы (справка по структурам данных в следующем параграфе), которая хранит название фигуры и

ссылку на фабричный метод его создания, например

```
public class ShapeCreator {

    private static HashMap<ShapeConstants, SCreator> factoryData =
initFactoryData();

    private static HashMap<ShapeConstants, SCreator>
initFactoryData(){
        return new HashMap<ShapeConstants, SCreator>() {{
            put(CUBE, new Cube());
            put(TRIANGLE, new Triangle());
            put(SPHERE, new Sphere());
            put(RECTANGLE, new Rectangle());
        }};
    }
```

Список констант, названий фигур может расширяться, без написания дополнительного кода.

```
public enum ShapeConstants {
    CUBE,
    RECTANGLE,
    SPHERE,
    TRIANGLE
}
```

4. В классе Controller графического интерфейса пользователя реализуйте необходимые обработчики событий, использующий Фабрику:

```
ShapeFactory shapeFactory = new ShapeFactory(); //get an object
 and call its draw method.
Shape shape = shapeFactory.getShape(1+fieldsides.getText());
GraphicsContext gc = canvas.getGraphicsContext2D(); //получить
контекст(холст) для рисования
shape.draw(gc); //вызов метода рисования по холсту
```

Особенности использования структур данных

Структуры данных определяет объекты, организованные определенным образом и операции, которые можно выполнять над объектами. Доступ к объектам и все операции с ними осуществляются только через интерфейс. Интерфейс отделяет реализацию структуры данных от клиента, и является «непрозрачным» для клиента. Структура данных может быть представлена как некий «черный ящик» с интерфейсами. Контейнерные классы предназначены для хранения данных, организованных определенным образом. Для каждого типа контейнера определены методы работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере. Поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов

В библиотеке коллекций Java существует два базовых интерфейса, реализации которых и представляют совокупность всех классов коллекций:

1. Collection содержит набор объектов (элементов) и определяет основные

методы для манипуляции с данными, такие как вставка (add, addAll), удаление (remove, removeAll, clear), поиск (contains)

2. Map описывает коллекцию, состоящую из пар "ключ – значение". У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (tap). Такую коллекцию часто называют еще словарем (dictionary) или ассоциативным массивом (associative array).

Интерфейс Collection расширяют интерфейсы List, Set и Queue.

1. **List** представляет собой неупорядоченную коллекцию, в которой допустимы дублирующие значения. Иногда их называют последовательностями (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.

Методы интерфейса Collection<E>

```
public interface Collection<E> extends Iterable<E> {
    boolean contains(Object obj);
    boolean containsAll(Collection<?> c);
    boolean isEmpty();
    int size();
    Object[] toArray();
    <T> T[] toArray(T[] arr);
    // Только для модифицируемых коллекций
    boolean add(E obj);
    boolean addAll(Collection<? extends E> c);
    void clear();
    boolean remove(Object obj);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
```

Методы интерфейса List<E>

```
public interface List<E> extends Collection<E> {
    E get(int index);
    int indexOf(Object obj);
    int lastIndexOf(Object obj);
    List<E> subList(int fromIndex, int toIndex);
    // Списочный итератор
    ListIterator<E> listIterator();
    // Только для модифицируемых списков
    // Старые методы add добавляют в конец списка
    boolean add(int index, E obj);
    boolean addAll(int index, Collection<? extends E> c)
    void clear();
    E remove(int index);
    E set(int index, E obj);
}
```

2. **Set** описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (set).

3. **Queue** предназначена для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля.

Между интерфейсом и конкретной реализацией коллекции существует несколько абстрактных классов (рис. 34). Это сделано для того, что бы вынести общий функционал в абстрактный класс, таким образом реализовать повторное использование кода.

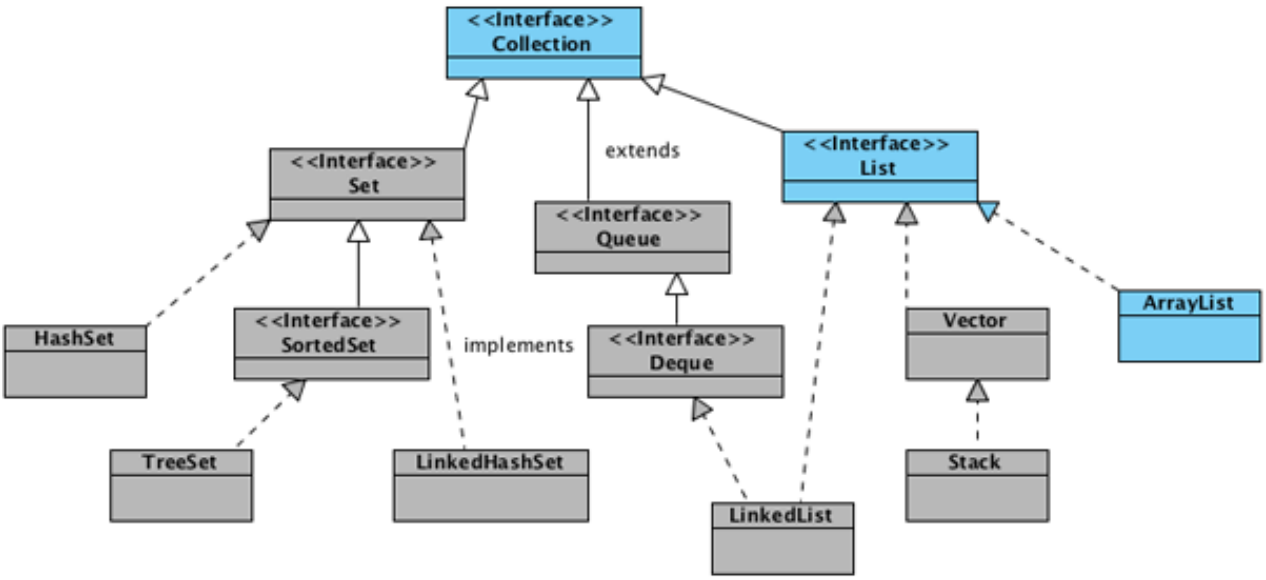


Рис. 34. Иерархия классов, предназначенных
для создания структур данных

ArrayList инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов. Так как ArrayList использует массив, то время доступа к элементу по индексу минимально (это принципиальное отличие от анналогичного по свойствам LinkedList). При удалении произвольного элемента из списка, все элементы находящиеся «правее» смещаются на одну ячейку влево, при этом реальный размер массива (его емкость, capacity) не изменяется.

LinkedList – двусвязный список. Это структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и две ссылки («связки») на следующий и предыдущий узел списка. Доступ к произвольному элементу осуществляется за линейное время (но доступ к первому и последнему элементу списка всегда осуществляется за константное время — ссылки постоянно хранятся на первый и последний, так что добавление элемента в конец списка вовсе не значит, что придется перебирать весь список в поисках последнего элемента).

HashSet – коллекция, не позволяющая хранить одинаковые объекты (как и любой Set). HashSet инкапсулирует в себе объект HashMap, к-й использует для хранения хэш-таблицу. Хеш-таблица хранит информацию, используя так называемый механизм хеширования, в котором содержимое ключа используется для определения уникального значения, называемого хеш-кодом. Этот хеш-код затем применяется в качестве индекса, с которым ассоциируются данные, доступные по этому ключу. Выгода от хеширования состоит в том, что оно обеспечивает константное время выполнения методов add(), contains(), remove() и size() , даже для больших наборов.

Важно отметить, что класс HashSet не гарантирует упорядоченности

элементов, поскольку процесс хеширования сам по себе обычно не порождает сортированных наборов. Если вам нужны сортированные наборы, то лучшим выбором может быть другой тип коллекций, такой как класс TreeSet.

LinkedHashSet поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор. То есть, когда идет перебор объекта класса LinkedHashSet с применением итератора, элементы извлекаются в том порядке, в каком они были добавлены.

TreeSet – коллекция, которая хранит свои элементы в виде упорядоченного по значениям дерева. TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов. TreeSet удобен тем, что для операций add, remove и contains потребуется гарантированное время log(n).

Пошаговый пример реализации задачи рисования фигуры, заданной числом сторон, по шаблону Фабричный метод

Шаг 1. Построение интерфейса **Shape**. Этот интерфейс является главным для фигур, создаваемых пользователем. Предоставляет метод draw, для отрисовки примитива.

```
public interface Shape {
          void draw(GraphicsContext gr);
            }
```

Шаг 2. Создание конкретных классов, реализующие один и тот же интерфейс для отображения различных фигур

Класс **Angle описывает** прямой угол.

```
public class Angle implements Shape {
    @Override
    public void draw(GraphicsContext gr) {
        gr.setStroke(Color.GRAY);
        gr.setLineWidth(10);
        gr.strokePolygon(new double[]{25, 250},
                new double[]{25,25},2
        );
        gr.strokePolygon(new double[]{30, 30},
                new double[]{25,250},2
        );
    }
}
```

Класс Circle, предназначен для рисования окружности:

```
public class Circle implements Shape {
    @Override
    public void draw(GraphicsContext gr) {
        gr.setFill(Color.GREEN);
```

```
        gr.fillOval(25, 25, 225, 225);
    }
}
```

Аналогично реализуются классы для других фигур.

Шаг 3. Создание фабрики для генерации объекта конкретного класса. У этого класса имеется метод createShape, который на основе значения параметра numberOfSides создаёт соответствующий экземпляр класса.

```
public class ShapeFactory {
    public Shape createShape(int numberOfSides){
        if(numberOfSides==5){
            return new Pentagon();
        }
        else if(numberOfSides==4){
            return new Squere();
        }
        else  if(numberOfSides==3){
         return new Triangle();
        }
        else if(numberOfSides==2){
            return new Angle();
        }
        else if (numberOfSides==1){
            return new Straight();
        }
        else if(numberOfSides==0){
            return new Circle();
        }else{
        return null;
        }
    }
}
```

Шаг 4. Реализация fxml-файла согласно заданной раскадровки, представляющего будущий графический интерфейс программы.

1. Установка элемента управления AnchorPane, который является контейнером для элементов, и позволяет позиционировать элементы внутри себя вдоль одной из сторон контейнера.

2. В контейнер были добавлены:

– метка Label, которая отображает информацию – подсказку для пользователя программы;

– текстовое поле TextField, используя которое, пользователь может ввести нужную цифру для создания конкретной фигуры;

– метка Label, которая отображает информацию – инструкцию по созданию геометрический фигур в программе;

– кнопку Button, к которой привязан обработчик onAction. Связанное с этой кнопкой действие отвечает за работу паттерна и отрисовку;

– специальный элемент Canvas, который будет использоваться для отображения созданных фигур.

Шаг 5. Реализация обработчиков событий пользовательского интерфейса. В классе **Controller** был добавлен метод для экземпляра класса Button события onAction, именующийся как *addPicker()*, вызывающий фабричный метод с параметром, заданным в текстовом поле value1.

```
public void addPikcher() {
        GraphicsContext gr = can.getGraphicsContext2D();
        if(checkWithRegExp(value1.getText())==false ||
value1.getText().equals("") ){
            Alert alert = new Alert(Alert.AlertType.INFORMATION);
            alert.setTitle("Предупреждение: ");
            alert.setHeaderText(null);
            alert.setContentText("Введено не число или число не из
диапазона от 0 до 5!");
            alert.showAndWait();
            return;
        }else {
           int numberOfSides = Integer.parseInt(value1.getText());
           ShapeFactory shapeFactory = new ShapeFactory();
           Shape shape1 = shapeFactory.createShape(numberOfSides);
            gr.clearRect(0, 0, 250, 485);
            shape1.draw(gr);
        }
    }
```

ШАБЛОН 3. АБСТРАКТНАЯ ФАБРИКА

Цель применения: Создавать изменчивые объекты без нежелательных зависимостей. Принцип инверсии зависимостей утверждает, что наиболее гибкими получаются системы, в которых зависимости в исходном коде направлены на абстракции, а не на конкретные реализации. Решением является использование интерфейсов, чтобы сделать объекты более изменчивыми и следует реализовывать больше интерфейсов, группирование их дает связанность и упрощает управление. Абстрактная фабрика позволяет создавать группы связанных объектов разных классов.

Проблема. В редакторе диаграмм необходимо реализовать возможность соединять две фигуры (точки) соединениями разных типов (штриховая линия, двухсторонняя стрелка, кривая заданного вида). Следовательно, необходимо создание общей модели стрелки для ее реализации в конкретных вариантах типа соединения и стиля линии, при различных способах задания ее начала и конца.

Описание паттерна

Паттерн Абстракная фабрика позволяет создавать семейства объектов, не инстанцируя (специфицируя) классы явно, при этом обеспечивается логическая связанность создаваемых объектов (рис. 35).

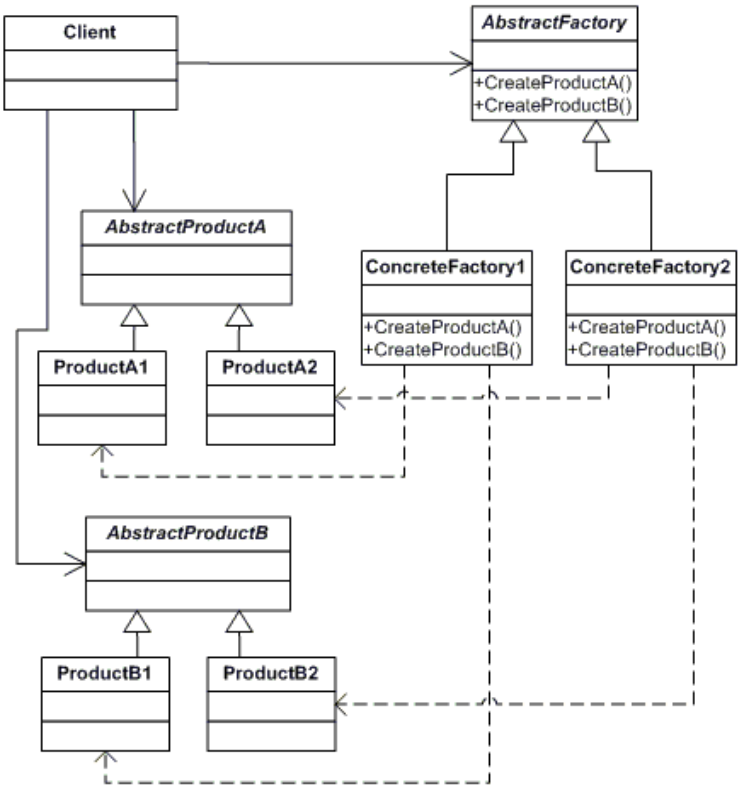


Рис. 35. Диаграмма классов паттерна Абстракная фабрика

AbstractFactory – объявляет интерфейс для создания семейства взаимосвязанных или родственных объектов.

AbstractProductA, AbstractProductB – семейство продуктов, которые будут использоваться клиентом для выполнения своих задач.

ProductA1, ProductB1 – конкретные типы продуктов.

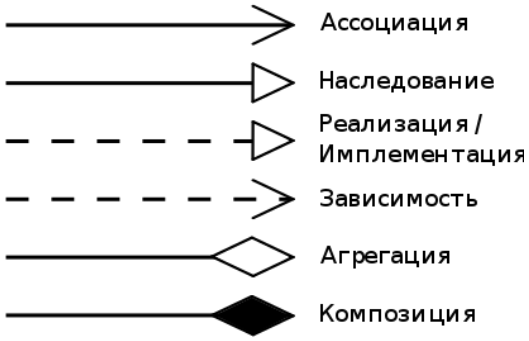
Client – клиент фабрики, который получает к AbstractFactory – объявляет интерфейс для создания семейства взаимосвязанных или родственных объектов.

Задание и указания по выполнению

Постановка задачи

Требуется написать программу, в которой на панели рисования можно соединять две указанные пользователем точки стрелкой определённого вида.

Виды стрелок:



Указания по выполнению

Следует разделить сложную задачу на две подзадачи:

А) создание продуктов вида «стрелка»;

Б) обработка действий пользователя для отображения выбранного подвида «стрелки».

Указания к выполнению подзадачи А. Создание Продукта – Стрелка, нарисованная на канве

А.1. Создайте модель абстрактной стрелки, аналогично фигуре для рисования на канве:

```
abstract public class AbstractProductArrow {
   Color color;
   double width;
   double startX;
   double startY;
   double endX;
   double endY;
   public abstract void draw(GraphicsContext gc);
}
```

А.2. Реализуйте конкретные стрелки:

```
public class ArrowDependence extends Shape {
   double arrowHeadSize;// размер кончика стрелки
   public ArrowDependence(double x, double y, double w, double h )
{
       this.color = Color.BLACK;
       ............
// определение пропорций кончика стрелки
       arrowHeadSize=0.09*Math.sqrt((startX-endX)*(startX-
endX)+(startX-endX)*(startX-endX))
       if
(arrowHeadSize<2)arrowHeadSize=0.146*start.distance(end);
       if
(arrowHeadSize<2)arrowHeadSize=0.236*start.distance(end);
   }
public void draw(GraphicsContext gc) {
      drawArrowstart( gc);//начало стрелки
      drawLine(gc);//линия
```

```
        drawArrowend(gc);//коней стрелки
    }
    public String toString() {
        return "Зависимость";   }
    public void drawArrowstart(GraphicsContext gc) {
        gc.setFill(color);
        gc.strokeOval(startX-width, startY-width, 2*width,
2*width);   }

    public void drawLine(GraphicsContext gc) {
        gc.setStroke(color);
        gc.setLineWidth(width);
        gc.setLineDashes(2.);
        gc.moveTo(startX, startY);
        gc.lineTo(endX, endY);
        gc.stroke();// соединить точки
    }
    public void drawArrowend(GraphicsContext gc) {
        gc.moveTo(endX, endY);
        double angle = Math.atan2((endY - startY), (endX - startX))
- Math.PI / 2.0;
        double sin = Math.sin(angle);
        double cos = Math.cos(angle);
        //point1
        double x1 = (- 1.0 / 2.0 * cos + Math.sqrt(3) / 2 * sin) *
arrowHeadSize + endX;
        double y1 = (- 1.0 / 2.0 * sin - Math.sqrt(3) / 2 * cos) *
arrowHeadSize + endY;
        //point2
        double x2 = (1.0 / 2.0 * cos + Math.sqrt(3) / 2 * sin) *
arrowHeadSize + endX;
        double y2 = (1.0 / 2.0 * sin - Math.sqrt(3) / 2 * cos) *
arrowHeadSize + endY;
        gc.lineTo(x1, y1);
        gc.moveTo(x2, y2);
        gc.lineTo(endX, endY);
        gc.stroke();
    }
}
```

А.3. Создайте интерфейс фабрик и реализуйте конкретные классы фабрик для различных видов стрелок.

```
public interface AbstractFactory {
    Shape CreateArrow(Canvas canvas);
}

public class FactoryGeneralization implements AbstractFactory {
    @Override
    public Shape CreateArrow() {
        startX= 1; startY= 50;
        endX=100;  endY=50;
```

```
        ArrowGeneralization arrow=new ArrowGeneralization(startX,
startY, endX, endY) ;
        return arrow;
    }
```

Указания к созданию GUI

Обязательный элемент графического интерфейса пользователя – канва для рисования, произвольный элемент управления для выбора вида стрелки и сам объект стрелка одного из конкретных классов, наследованных от Shape, т.е. для хранения текущей стрелке в Controller потребуется объект:

```
private Shape currentarrow;
```

который будет хранить всю информацию о рисуемой стрелке.

В классе Controller:

Б.1. Выполните инициализацию меню выбора стрелок, например, для public ChoiceBox <String> arrows;

в виде:

```
arrows.getItems().addAll("обобщение", "зависимость",
"реализация");
arrows.setValue("обобщение");

currentarrow=new FactoryGeneralization().CreateArrow(canvas);
```

Б.2. Добавьте обработчик выбора элемента из ChoiceBox

```
arrows.addEventHandler(MouseEvent.MOUSE_CLICKED, new
EventHandler<MouseEvent>() {
```

// реализация метода конкретной обработки

```
    public void handle(MouseEvent event) {
      String current=arrows.getValue(); // получение выбранного
текстового значения
```

// и для каждого возможного текстового значения назначается конкретная Фабрика для создания конкретной стрелки

```
        if(current.equals("обобщение")) currentarrow=new
FactoryGeneralization().CreateArrow(canvas);
        if(current.equals("реализация")) currentarrow=new
FactoryAssotioon().CreateArrow(canvas);
        if(current.equals("зависимость")) currentarrow=new
FacoryDependence().CreateArrow(canvas);   }
});
```

Б.3. Для выбранной схемы рисования стрелки пльзователем, например «нажал кнопку мыши в начальной точке стрелки и отпустил в конечной» реализуйте соответствующие обработчики, например,

```
public void onBegin(MouseEvent mouseEvent) {
    currentarrow.setStartX(mouseEvent.getX());
    currentarrow.setStartY(mouseEvent.getY());
    currentarrow.draw(canvas.getGraphicsContext2D());
}
public void onRun(DragEvent dragEvent) {
    currentarrow.setEndX(dragEvent.getX());
    currentarrow.setEndY(dragEvent.getY());
    canvas.getGraphicsContext2D().clearRect(1,1,200,200);
```

```
   currentarrow.draw(canvas.getGraphicsContext2D());
}
public void onEnd(DragEvent dragEvent) {
   currentarrow.setEndX(dragEvent.getX());
   currentarrow.setEndY(dragEvent.getY());
   currentarrow.draw(canvas.getGraphicsContext2D());
}
```

Работа с потоками средствами Java

Для того чтобы отвлечься от особенностей конкретных устройств ввода/вывода, в Java используются потоки (stream). Считается, что в программу идет входной поток (input stream) символов Unicode или просто байтов, воспринимаемый в программе методами read(). Из программы методами write() или print() выводится выходной поток (output stream) символов или байтов.

В Java предусмотрена возможность создания потоков, направляющих символы или байты не на внешнее устройство, а в массив или из массива, т. е. связывающих программу с областью оперативной памяти. Можно создать поток, связанный со строкой типа string, или создать канал (pipe) обмена информацией между подпроцессами. Еще один вид потока — поток байтов, представляющих объект Java. Его можно направить в файл или передать по сети, а потом восстановить в оперативной памяти. Эта операция называется сериализацией (serialization) объектов.

Вместо того чтобы записывать в потоки и читать оттуда отдельные байты или массивы байт, программисты обычно предпочитают пользоваться намного более удобными методами классов DataOutputStream и DataInputStream, допускающими форматированный ввод и вывод данных. Так для записи форматированных данных в поток класса DataOutputStream можно использовать методы записи символа, числа, строки и пр. Так если нужно записать в выходной поток текстовую строку, то это можно сделать с помощью методов writeBytes, writeChars или writeUTF. Первый из этих методов записывает в выходной поток только младшие байты символов, а второй - двухбайтовые символы в кодировке Unicode. Метод writeUTF предназначен для записи строки в машинно-независимой кодировке UTF-8. Все эти методы в случае возникновения ошибки создают исключение IOException, которое необходимо обработать.

Пример записи строки str в файл:

```
FileWriter fw = new FileWriter("test.txt"))
fw.write(str);
```

В классе DataInputStream определены методы readByte(), readDouble(), readFloat(), readInt(), readUTF() предназначенные для чтения форматированных данных из входного потока. Среди методов нет предназначеных для чтения данных, записанных из строк методами writeBytes и writeChars класса DataOutputStream.

Тем не менее, если входной поток состоит из отдельных строк, разделенных символами возврата каретки и перевода строки, то такие строки можно получить методом readLine. Вы также можете воспользоваться методом readFully, который заполняет прочитанными данными массив байт. Этот массив потом будет нетрудно преобразовать в строку типа String, так как в классе String предусмотрен соответствующий конструктор.

Для чтения строк, записанных методом writeUTF, необходимо пользоваться методом readUTF.

Пример чтения файла:

```
class ShowFile {
  public static void main(String args[])
  {
    int i;
    FilelnputStream fin = null;

    // Прежде всего следует убедиться, что файл был указан,
    if (args.length != 1)   {
      System.out.println("Usage: ShowFile filename");
      return;
    }

    // В следующем коде открывается файл, из которого читаются
    // символы до тех пор, пока не встретится знак EOF, а затем
    // файл закрывается в блоке finally,
    try {
      fin = new FilelnputStream(args[0]);// указать  filename
вместо args[0]
      do {
        i = fin.read() ;
        if(i != -1) System.out.print((char) i);// вывести во
внутреннею переменную String currentLine
      } while(i != -1);
    } catch(FileNotFoundException exc) {
      System.out.println("File Not Found.");
    } catch(IOException exc) {
      System.out.println("An I/O Error Occurred");
    } finally {
      // Файл закрывается в любом случае,
      try {
        if (fin != null) fin.closeO;
      } catch(IOException exc) {
        System.out.println("Error Closing File");
      }
    }
```

Пример использования Абстрактной фабрики для организации внешнего представления информации

В реализации программы «Калькулятор» энергопотребления для одного прибора может быть одно значение потребляемой мощности, а для другого несколько (рис. 36). Эти данные считываются из файлов в процессе выполнения программы, поэтому необходимо динамически формировать пользовательский интерфейс.

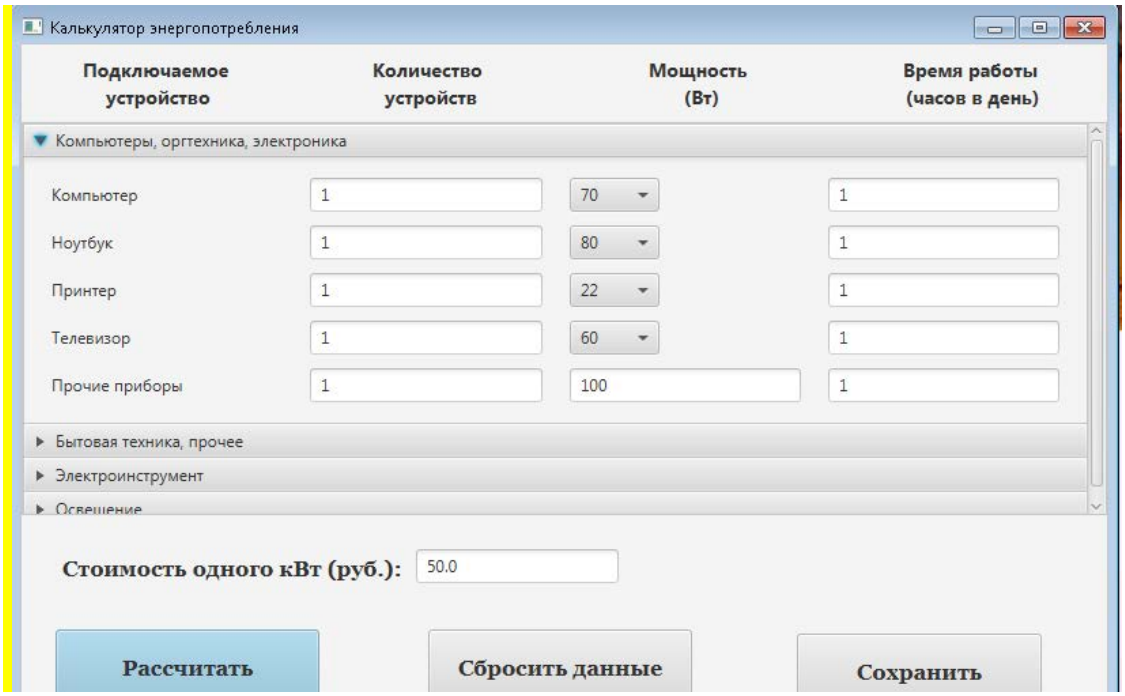


Рис. 36. Программа Калькулятор энергопотребления

Диаграмма классов приложения представлена на рис. 37.

Шаг 1. Создание общего класса, описывающегося все подключаемые устройства.

Класс «Device» представлен абстрактным классом.

```
public abstract class Device {
    private String name;
    private AtomicInteger count;
    private AtomicInteger time;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public AtomicInteger getCount() {
        return count;
    }
```

```
    public void setCount(AtomicInteger count) {
        this.count = count;
    }
    public AtomicInteger getTime() {
        return time;
    }
```

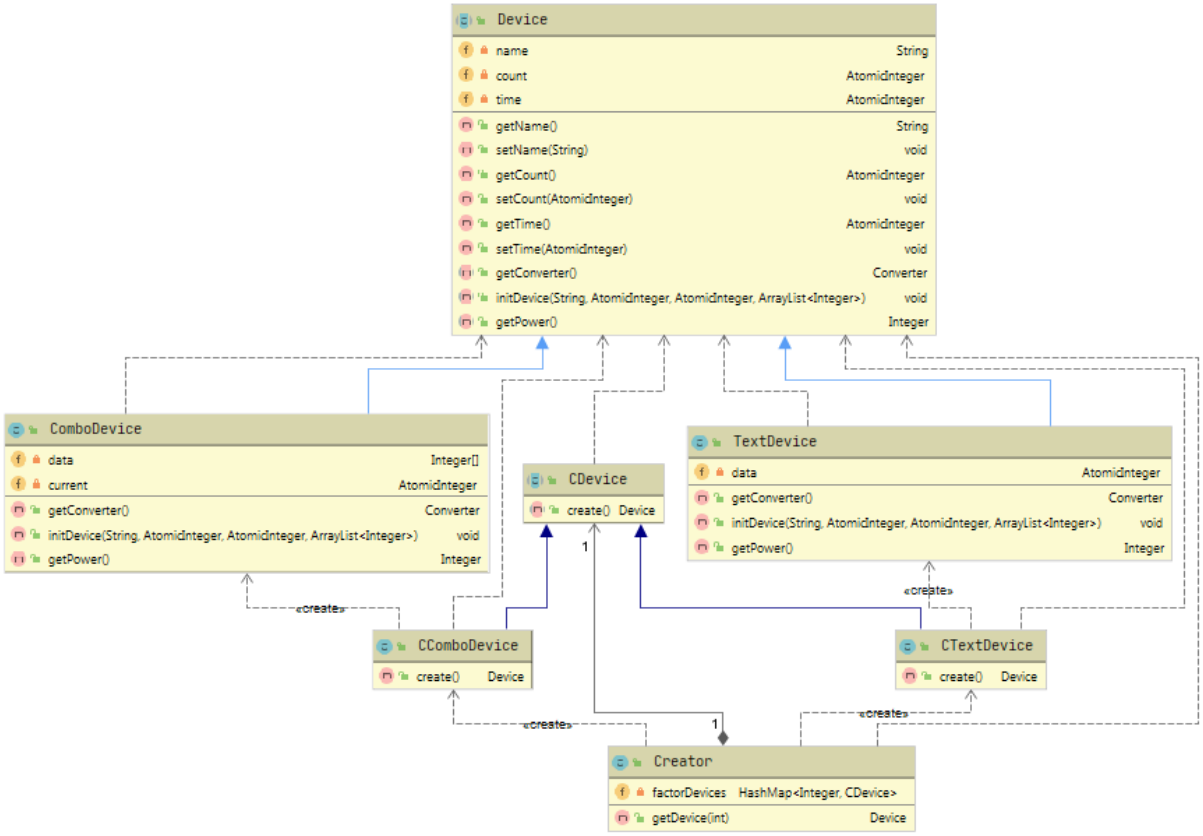


Рис. 37. Диаграмма классов динамической генерации полей
для задаиня мощности прибора Калькулятора энергопотребления

```
    public void setTime(AtomicInteger time) {
        this.time = time;
    }
    public abstract Converter getConverter();
    public abstract void initDevice(String n, AtomicInteger c,
AtomicInteger t, ArrayList<Integer> p);
    public abstract Integer getPower();
}
```

Шаг 2. Создание наследуемых классов от Device.

Класс ComboDevice определяет устройство, для которого может быть указано несколько значений мощностей (список):

```
public class ComboDevice extends Device {
    private Integer [] data;
    private AtomicInteger current;
```

```
    @Override
    public Converter getConverter() {
        return new ComboConverter(data, current);
    }
    @Override
    public void initDevice(String n, AtomicInteger c,
AtomicInteger t, ArrayList<Integer> p) {
        setName(n);
        setCount(c);
        setTime(t);
        data = p.toArray(new Integer[p.size()]);
        current = new AtomicInteger(p.get(0));
    }
    @Override
    public Integer getPower() {
        return current.get();
    }
}
```

Класс TextDevice описывает устройство, для которого может быть указано только одно значение мощности:

```
public class TextDevice extends Device {
    private AtomicInteger data;
    @Override
    public Converter getConverter() {
        return new TextConverter(data);
    }
    @Override
    public void initDevice(String n, AtomicInteger c,
AtomicInteger t, ArrayList<Integer> p) {
        setName(n);
        setCount(c);
        setTime(t);
        data = new AtomicInteger(p.get(0));
    }
    @Override
    public Integer getPower() {
        return data.get();
    }
}
```

Шаг 3. Создание абстрактной фабрики.

Абстрактная фабрика будет представлена абстрактным классом CDevice. Она содержит один метод по созданию устройства.

```
public abstract class CDevice {
    public abstract Device create();
}
```

Шаг 4. Создание наследуемых фабрик.

Наследуемые фабрики CComboDevice и CTextDevice реализуют создание контрактного типа устройства, созданного на шаге 2.

Класс «CComboDevice» создает «ComboDevice»:

```
public class CComboDevice extends CDevice {
    @Override
    public Device create() {
        return new ComboDevice();
    }
}
```
Класс « CTextDevice » создает «ComboDevice»:
```
public class CTextDevice extends CDevice {
    @Override
    public Device create() {
        return new TextDevice();
    }
}
```

Шаг 5. Создание класса Creator, который будет передавать созданное представление прибора.

```
public class Creator {    private HashMap<Integer, CDevice>
factorDevices = new HashMap<Integer, CDevice>(){{
      put(0, new CTextDevice());
      put(1, new CComboDevice());
    }};
    public Device getDevice(int i){
        try{
            return factorDevices.get(i).create();
        }catch (Exception ex){
            System.out.println(ex);
        }
        return null;
    }

}
```

ШАБЛОН 4. СТРОИТЕЛЬ

Цель применения: Создавать различные объекты по одному и тому же принципу

Проблема. Разрабатывается Редактор Смайликов. По аналогии с любым графическим редактором он должен иметь набор графических объектов, к-е могут быть нарисованы в любом месте рабочего окна выбранным цветом контура и заполнения.

Смайлик – графическое изображения для отображения эмоций. Традиционно изображается в виде жёлтого круга (объект Head) с двумя чёрными точками, представляющими глаза (объект Eyes), и чёрной дугой, символизирующей рот (объект Mouth).

Для генерации смайликов потребуется управление их построением т.е. задание последовательности рисования графических объектов (Head, Eyes и Mouth).

Хорошим решением этой задачи является использование паттерна Строитель для автогенерации различных видов смайликов (веселый, унылый, злой).

Описание

Паттерн Строитель (Builder) используется для создания новых объектов по одному и тому же процессу конструирования. Это даёт возможность использовать один и тот же алгоритм процесса строительства для получения разных видов объектов.

Диаграмма классов паттерна Строитель представлена на рис. 38.

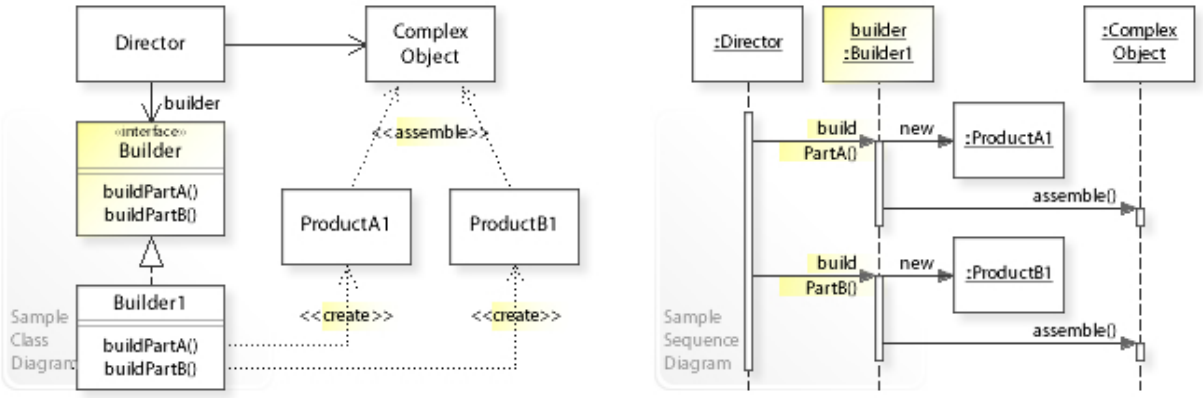


Рис. 38. Диаграмма классов и последовательностей паттерна Строитель

Результатом работы является построение нового объекта Product, который возвращает конкретный строитель ConcreteBuilder, при этом вызывает его объект класса Director, к которому обращается Client. Поэтому сначала определяют интерфейс Builder, который объявляет шаги конструирования продуктов, общие для всех видов строителей.

ConcreteBuilder реализуют шаги конструирования, каждый по-своему. Конкретные строители могут производить объекты разного представления.

Director определяет порядок вызова строительных шагов для построения той или иной конфигурации продуктов.

Client – контекст использования паттерна.

Product – создаваемый объект.

Задание и указания по выполнению

Постановка задачи

Разработать редактор смайликов, содержащий набор графических объектов (Head, Eyes и Mouth), вид и положение которых на панели рисования может быть изменен пользователем.

Для генерации смайликов потребуется управление их построением т.е. задание последовательности рисования графических объектов (Head, Eyes и Mouth).

Указания по выполнению

1. Создайте отдельные объекты *Head, Eyes, Mouth и др.* на основе

имеющихся графических объектов типа Rectangle, Circle, Line и пр. от **Shape** javaFX (детально смотрите следующий подраздел).

Например,

```
public class Head extends Circle {
       public Head(double radius, Paint fill) {
        super(radius, fill);
    }
    public Head() {
        super(50.,Color.BLUE);
    }
}
```

2. Создайте класс Smile, агрегирующий части Смайлика (включающий объекты Head, Eyes и Mouth):

```
public class Smile extends Group {
   public Smile() {
    }
   public void setHead(Head m)    {
        getChildren().add(m);
    }
   public void setMouth(Mouth m)   {
       getChildren().add(m);
   }
    public void setEyes(Eyes m)    {
        getChildren().add(m);
    }
    public String toString()
    {
       return new String ("Смайлик");
    }
```

3. Объявите интерфейс *Builder* для построения объекта Смайлика:

```
public interface Builder {
    void buildHead();
    void buildEyes();
    void buildMouth();
    Smile getSmile();
```

4. Создайте различные классы реализующие интерфейс ***Builder*** для «доброго», «злого» и пр. вариантов Смайлика:

```
 public class GoodyBuilder implements Builder{
 private Smile smile;
    public GoodyBuilder() {
        smile = new Smile();
    }

public void buildHead() {
        Head h=new Head(40, Color.GEEN);
        smile.setHead(h);
    }

 public void buildEyes() {
```

```
    }

public void buildMouth() {
        smile.setMouth(new Mouth());
    }

 public Smile getSmile() {
        return smile;
    }
```

5. Разработайте собственный пользовательский интерфейс для поставленной задачи.

6. Создайте класс ***Director*** для управления Строителями:

```
public class Director {
    public Smile construct(Builder builder) {
        builder.buildHead();
        builder.buildEyes();
        builder.buildMouth();
        return builder.getSmile();
    }
```

7. В классе Controller графического интерфейса пользователя, создайте необходимые ConcreteBuilder и объект:

Director **direct**=**new** Director();

8. В обработчиках событий реализуйте обращения к ***Director*** для отображения выбранного типа Смайлика:

```
public void clickBad(ActionEvent actionEvent) {

    Smile smile=new Smile();
    smile=direct.construct(new BadBuilder());
// добавление smile к узлу для отображения

}
```

9*. Реализуйте дополнительные функции:

– вращение фигур для придания образа «насмешливый», «грубый» и т.п.;

– вывод названия Смайлика;

– построения Смайлика нового типа.

FX: Фигуры и эффекты

Любая фигура, которую можно нарисовать в двухмерной плоскости в JavaFX называется 2D фигура (Shape) и является таким же узлом сцены как элементы управления пользовательского интерфейса. НО – это конечные узлы и к ним недопустимо добавление других shape.getChildren().add(...) .

Все классы фигур находятся в пакете javafx.scene.shape. Множество узлов для рисования различных типов фигур (линий, кругов, прямоугольников и т. д.) представлено соотвествующими классами (0).

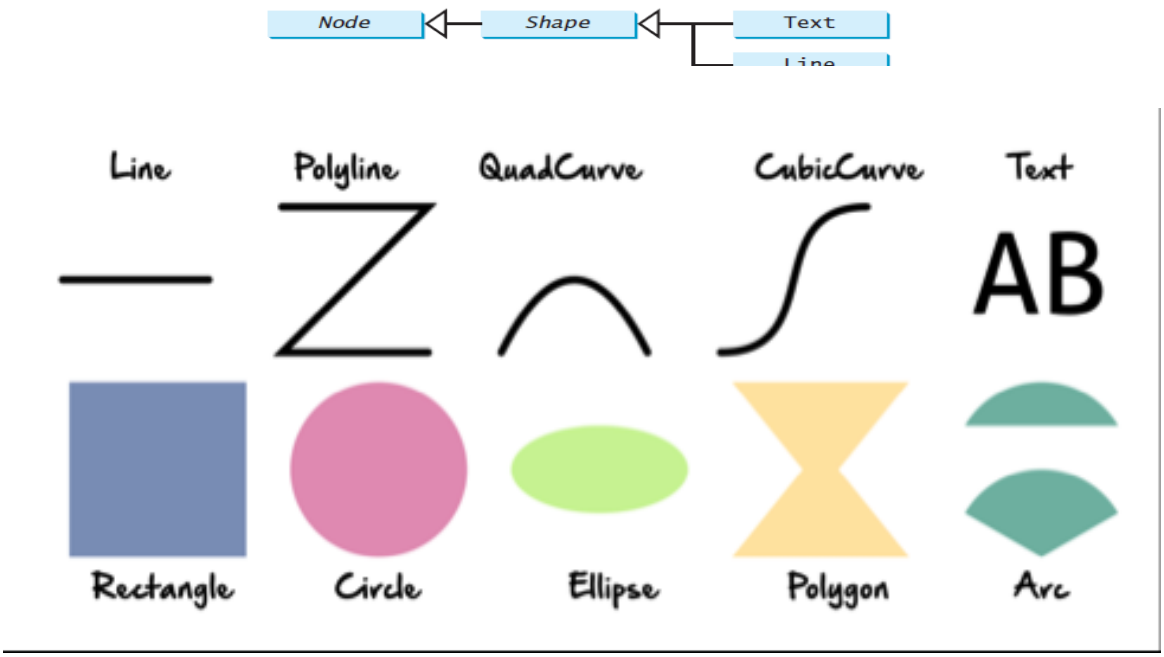


Рис. 39. Классы и внешний вид двухмерных фигур JavaFX

У фигуры есть размер и положение, которые определяются их свойствами. Например, свойства width и height определяют размер прямоугольника; свойство radius определяет размер круга, свойства x и y определяют положение верхнего левого угла прямоугольника, свойства centerX и centerY определяют центр круга и т. д. Родительские узлы не изменяют размер фигур во время перестроения макета, что отличает фигуры от элементов управления. Размер фигуры изменяется только при изменении ее свойств, связанных с размером.

Пример задания прямоугольника:

```
Rectangle rectangle = new Rectangle();
        rectangle.setX(20);// 20 пикселей от левого верхнего угла родительского узла
        rectangle.setY(50);
        rectangle.setWidth(200);
        rectangle.setHeight(150);
        rectangle.setFill(Color.BISQUE);
```

У фигур есть заливка и контур. Свойство fill указывает цвет для заливки внутренней части формы. По умолчанию заливка – Color.BLACK. Свойство stroke определяет цвет обводки контура, который по умолчанию имеет значение NULL, за исключением линий, полилиний и контуров, для которых в качестве обводки по умолчанию используется Color.BLACK. Свойство strokeWidth определяет ширину контура, которая по умолчанию составляет 1,0 пикселя.

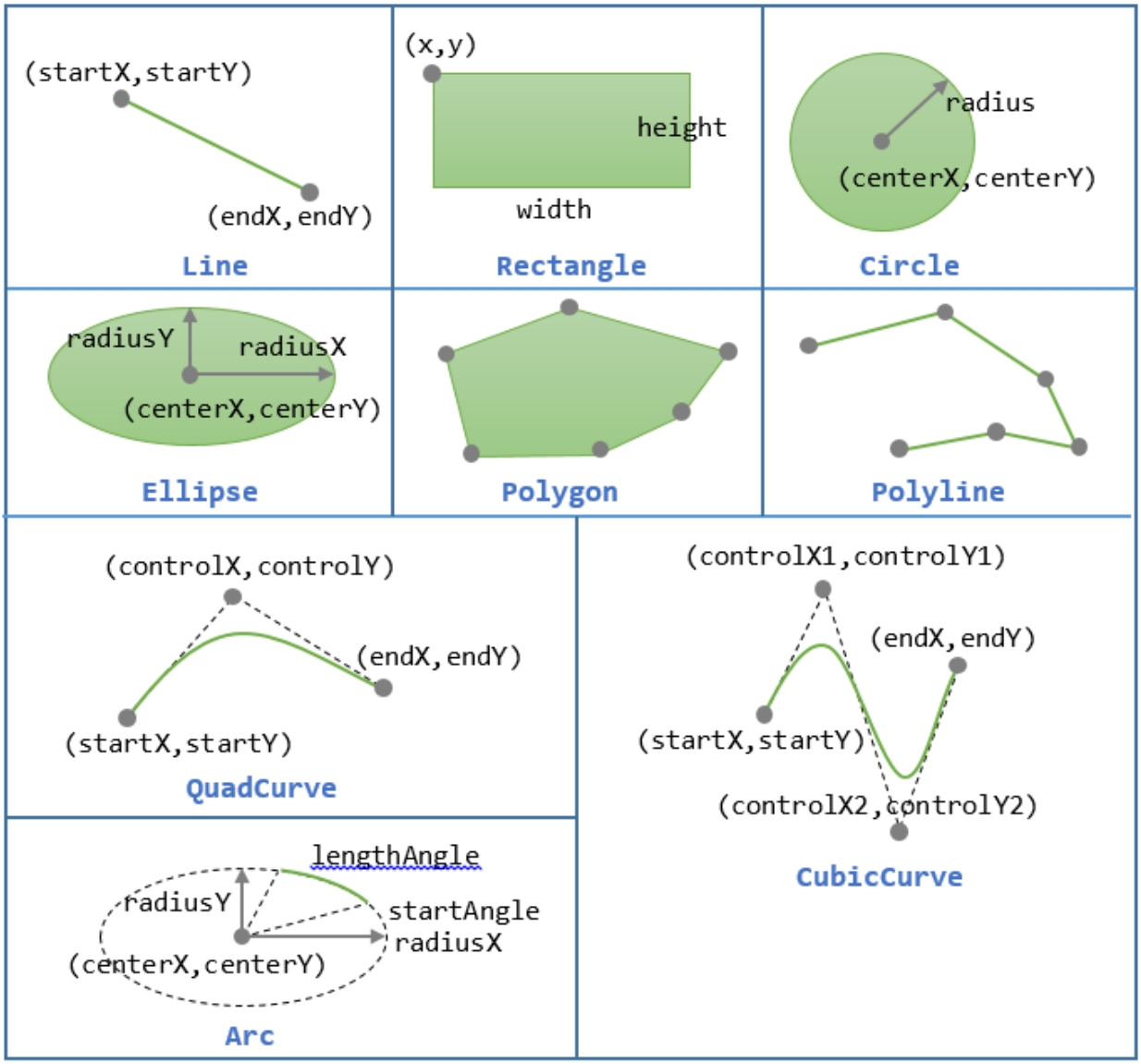


Рис. 40. Задание параметров фигур

Пример задания создания пентагон на основе Polygon:

```
Pane pane = new Pane();// создание панели
Polygon polygon = new Polygon(); // создание полигона
pane.getChildren().add(polygon); // добавление полигона к панели
polygon.setFill(Color.WHITE); // задание цвета заливки
polygon.setStroke(Color.BLACK); // задание цвета контура
ObservableList<Double> list = polygon.getPoints(); // получить
список координат для отображения
final double WIDTH = 200, HEIGHT = 200; // Установить размеры
фигуры
double centerX = WIDTH / 2, centerY = HEIGHT / 2;  //
double radius = Math.min(WIDTH, HEIGHT) * 0.4;     //
// Формирвоание списка точек фигуры
for (int i = 0; i < 6; i++)
{
```

```
    list.add(centerX + radius * Math.cos(2 * i * Math.PI / 6));
    list.add(centerY - radius * Math.sin(2 * i * Math.PI / 6));
}
```

Пример создания треугольника из имеющихся фигур:

```
Path triangle = new Path(new MoveTo(0, 0),new LineTo(0, 50),
                         new LineTo(50, 50), new ClosePath());
triangle.setFill(Color.LIGHTGRAY);//закрашивание
```

Класс Shape содержит свойство smooth, которое по умолчанию имеет значение true. Его истинное значение указывает, что для визуализации формы будет использовать сглаживание. Если установлено значение false, сглаживания – не будет, и края фигур будут нечеткими.

Пошаговый пример реализации Построителя смайликов

Шаг 1. Создание пользовательского интерфейса в Scene Builder (0).

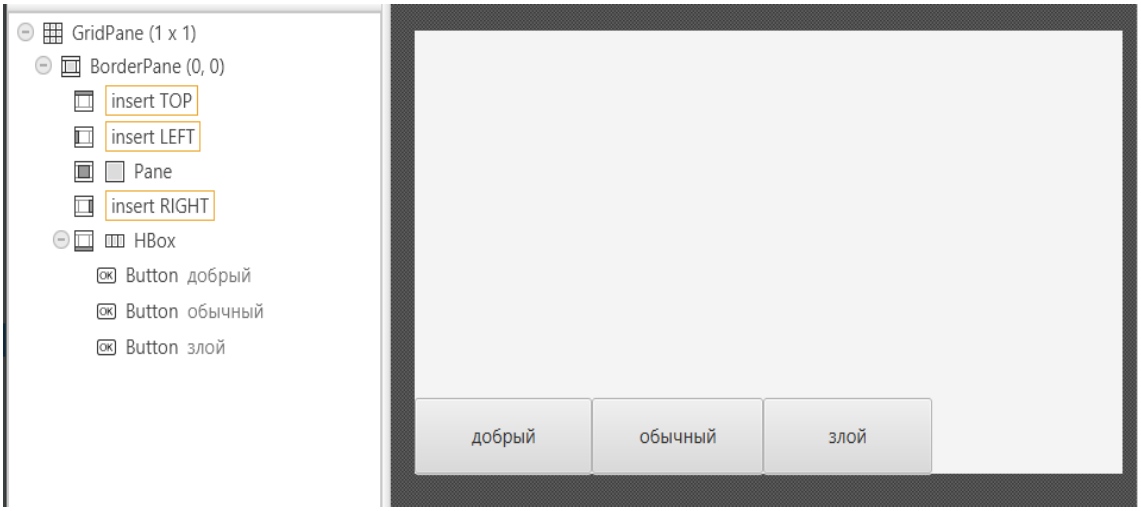


Рис. 41. Построение пользовательского интерфейса в Scene Builder

Шаг 2. Создание интерфейса **Builder** для производства смайликов, которые будет создавать пользователь. В нем описаны методы для построения головы, глаз, бровей и рта, а также метод, который вернет нам готового собранного смайла:

```
public interface Builder{
    void buildHead(Pane pane);
    void buildEyes(Pane pane);
    void buildBrows(Pane pane);
    void buildMouth(Pane pane);
    Smile getSmile();
}
```

Шаг 3. Создание классов для отдельных частей смайлика, т.е.: **Head**,

Eyes, **Brows** и **Mouth**, производных или агрегирующих графические объекты JavaFX.

```
public class Head extends Circle {

    private Circle circleHead;

    public Head(double radius, Color fill, Pane pane){
        super(radius);
        super.setFill(fill);
        super.setCenterX(pane.getWidth() / 2);
        super.setCenterY(pane.getHeight() / 2);
    }
}
```

Класс **Eyes** будет создан от класса Circle. Так же в этом классе определены методы для положения первого и второго глаза.

```
public class Eyes extends Circle {

    public Eyes(){
        super(15, Color.BLACK);
    }

    public void firstEye( Pane pane){
        super.setCenterX(pane.getWidth() / 2 - 40);
        super.setCenterY(pane.getHeight() / 2 - 25);
    }

    public void secondEye( Pane pane){
        super.setCenterX(pane.getWidth() / 2 + 40);
        super.setCenterY(pane.getHeight() / 2 - 20);
    }
}
```

Класс **Brows** будет создан от класса Polyline. В данном классе определены методы положения первой и второй брови для каждого из вида смайликов (Bad, Neutral и Goody).

```
public class Brows extends Polyline {

    public Brows(){
        super();
    }

    public void firstBadBrow(Pane pane){
        super.getPoints().addAll(new Double[]{
            pane.getWidth() / 2 - 50, pane.getHeight() / 2 - 55,
            pane.getWidth() / 2 - 30, pane.getHeight() / 2 - 52,
            pane.getWidth() / 2 - 10, pane.getHeight() / 2 - 45}
        );
    }
```

```
    public void secondBadBrow(Pane pane){
        super.getPoints().addAll(new Double[]{
          pane.getWidth() / 2 + 50, pane.getHeight() / 2 - 55,
          pane.getWidth() / 2 + 30, pane.getHeight() / 2 - 52,
          pane.getWidth() / 2 + 10, pane.getHeight() / 2 - 45 }
        );
    }
    public void firstNeutralBrow(Pane pane){
        super.getPoints().addAll(new Double[]{
            pane.getWidth() / 2 - 50, pane.getHeight() / 2 - 50,
            pane.getWidth() / 2 - 10, pane.getHeight() / 2 - 50 }
        );
    }
    public void secondNeutralBrow(Pane pane){
        super.getPoints().addAll(new Double[]{
            pane.getWidth() / 2 + 50, pane.getHeight() / 2 - 50,
            pane.getWidth() / 2 + 10, pane.getHeight() / 2 - 50 }
        );
    }
    public void firstGoodyBrow(Pane pane){
        super.getPoints().addAll(new Double[]{
            pane.getWidth() / 2 - 50, pane.getHeight() / 2 - 50,
            pane.getWidth() / 2 - 30, pane.getHeight() / 2 - 56,
            pane.getWidth() / 2 - 10, pane.getHeight() / 2 - 50 }
        );
    }

    public void secondGoodyBrow(Pane pane){
        super.getPoints().addAll(new Double[]{
            pane.getWidth() / 2 + 50, pane.getHeight() / 2 - 50,
            pane.getWidth() / 2 + 30, pane.getHeight() / 2 - 56,
            pane.getWidth() / 2 + 10, pane.getHeight() / 2 - 50 }
        );
    }
}
```

Шаг 4. Создание класса-продукта Smile, который будет создавать контур фигуры и добавлять элементы, отображающие голову, глаза, брови и рот в качестве дочерних. Для этого необходимо сделать его зависимым от класса **Group.**

```
public class Smile extends Group {
    public Smile() {
    }

    public void setHead(Head m) {
        getChildren().add(m);
    }

    public void setMouth(Mouth m) {
```

```
        getChildren().add(m);
    }

    public void setEyes(Eyes m[]) {
        for (Eyes i:m) {
            getChildren().add(i);
        }
    }

    public void setBrows(Brows m[]) {
        for (Brows i:m) {
            getChildren().add(i);
        }
    }

    public String toString() {
        return new String("Смайлик");
    }

}
```

Шаг 5. Создание конкретных классов, реализующие один и тот же интерфейс **Builder**.

Класс **BBuilder п**редоставляет реализацию злого смайлика.

```
public class BBuilder implements Builder{

    private Smile smile;
    public BBuilder() {
        smile = new Smile();
    }

    public void buildHead(Pane pane) {
        Head head = new Head(100, Color.RED, pane);
        smile.setHead(head);
    }

    public void buildEyes(Pane pane) {
        Eyes [] eyes = new Eyes[]{new Eyes(), new Eyes()};
        eyes[0].firstEye(pane);
        eyes[1].secondEye(pane);
        smile.setEyes(eyes);
    }

    public void buildBrows(Pane pane) {
        Brows [] brows = new Brows[]{new Brows(), new Brows()};
        brows[0].firstBadBrow(pane);
        brows[1].secondBadBrow(pane);
        smile.setBrows(brows);
    }
```

```
    public void buildMouth(Pane pane) {
        Mouth mouth = new Mouth();
        mouth.badMouth(pane);
        smile.setMouth(mouth);
    }

    public Smile getSmile() {
        return smile;
    }
}
```

Аналогично определяется класс NBuilder, который предоставляет реализацию нейтрального смайлика и Класс **GBuilder с** реализацией довольного смайлика.

Шаг 6. Создание класса **Director**, для реализации метода вызова конкретного строителя, который сможет в качестве аргумента принимать объект с интерфейсом **Builder.**

```
public class Director {

    public Smile build(Builder builder, Pane pane) {
        builder.buildHead(pane);
        builder.buildEyes(pane);
        builder.buildBrows(pane);
        builder.buildMouth(pane);
        return builder.getSmile();
    }
}
```

Шаг 7. Агреггирование класса ожидаемого продукта Group в управляющий класс Controller для корректной работы метода getChildren(). В контроллере устанавливаются методы, которые будут выполняться по нажатию на соответствующие кнопки при запуске программы. Так же в контроллере создаются объекты классов Pane для рисования, Director для управления и Smile, как продукта работы Builder.

```
public class Controller extends Group {

    public Pane pane;
    Director direct = new Director();
    Smile smile;

    public void clickBad(ActionEvent actionEvent) {
        smile = direct.build(new BBuilder(), pane); //вернет
значение типа Smile (в нем будет храниться уже собранный злой
смайлик).
        pane.getChildren().add(smile);// создание рисунка путем
добавления объекта к узлу -панели
```

```
    }

    public void clickNeutral(ActionEvent actionEvent) {
        smile = direct.build(new NBuilder(), pane);
        pane.getChildren().add(smile);
    }

    public void clickGoody(ActionEvent actionEvent) {
        smile = direct.build(new GBuilder(), pane);
        pane.getChildren().add(smile);
    }
}
```

Пример работы разработанного приложения на 00

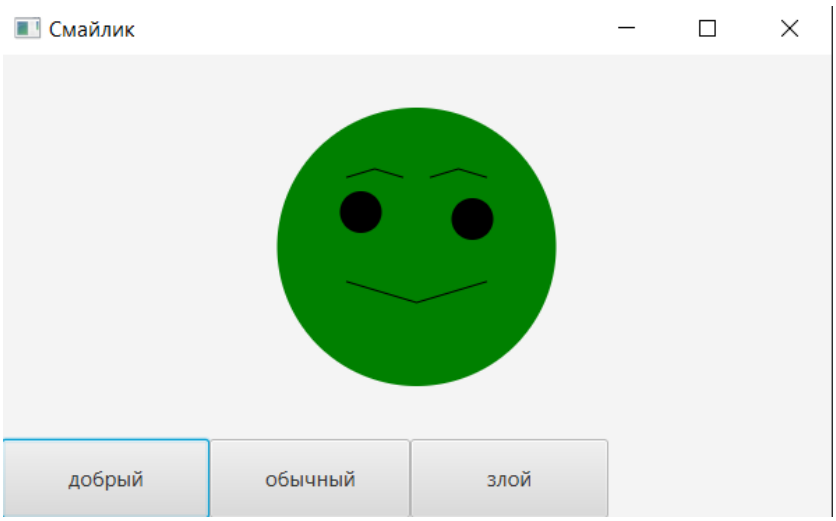


Рис. 42. Результаты работы Генератора смайликов при нажатии на кнопку «Добрый»

ШАБЛОН 5. ОДИНОЧКА (SINGLETON)

Цель применения: ограничить количество экземпляров некоторого класса

Проблема. Обеспечить возможность создавать и редактировать настройки отображения в любой части программы, при этом быть уверенным, что обеспечен общий доступ к одним и тем параметрам.

Описание паттерна Одиночка

Паттерн Singleton гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему.

Цель реализации паттерна Singleton ограничить количество экземпляров некоторого класса в пределах приложения. Практически в любом приложении возникает необходимость в глобальных переменных или объектах с огра-

ниченным числом экземпляров. Самый простой способ решить эту задачу — создать глобальный объект, который будет доступен из любой точки приложения. По своему определению Singleton гарантирует, что у некоего класса есть лишь один экземпляр. В некоторых случаях анализ предметной области строго требует, чтобы класс существовал лишь в одном экземпляре. Однако на практике паттерн Singleton обычно используется для обеспечения доступа к какому-либо ресурсу, который требуется разным частям приложения.

Диаграмма классов (0) допускает множество различных реализаций.

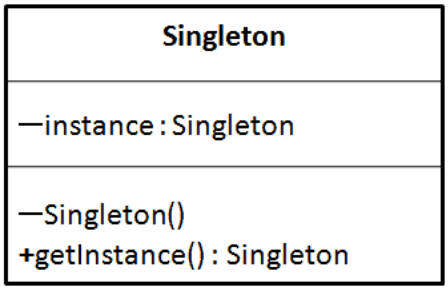


Рис. 43. Диаграмма класса Одиночка

Простая реализация:

```
public final class Singleton {
    private static Singleton _instance = null;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (_instance == null)
            _instance = new Singleton();
        return _instance;
    }
}
```

Реализация с ленивой инициализацией:

```
public final class Singleton {
    private Singleton() {}

    private static class Holder {
        private static final Singleton _instance = new
Singleton();
    }

    public static Singleton getInstance() {
        return Holder._instance;
    }
}
```

Примитивный подход:

```
public enum Singleton {
        INSTANCE;
    }
```

Задание и указания по выполнению

Постановка задачи

Написать программу, которая рисует картинку «домик», состоящую из различных графических объектов (крыша, стены, окна, дверь, и т.п.) с заранее определенными, без возможности внесения изменений параметрами отображения.

Указания по выполнению

Данная задача потребует реализации как минимум двух паттернов – порождающего для построения сложного изображения и паттерна Одиночка – для задания и сохранения неизменными параметров рисования.

Паттерн «Одиночка» инкапсулирует параметры фигуры, которая должна будет рисоваться на холсте. Класс должен содержать следующие параметры:

- толщина контура фигуры;
- цвет фигуры.

При первом обращении к классу происходит инициализация и создание скрытой ссылки на внутренний экземпляр класса. Данная ссылка хранит все параметры рисования фигуры. При последующем обращении к классу будет возвращена созданная при первом обращении ссылка. Поэтому при любой отрисовки фигуры должно происходить обращение к классу – «Одиночке», который возвращает заданные ранее параметры.

FX: Image & ImageView

Как класс File хранит информацию о файле, но может быть использован для чтения или записи в него (необходимо обращение к Scanner или PrintWriter, подключенному к объекту File). Точно так же класс Image является контейнером для хранения изображения, а фактическое отображение осуществляется специальным элементом ImageView, который является конечным узлом и может быть добавлен к сцене или панели.

Использование Image и ImageView происходит совместно и оба класса находятся в пакете javafx.sciene.image.

Класс Image используется для загрузки изображения из веб-ресурса по URL или по пути к файлу. Полученное изображение можно воспроизвести посредством элемента ImagView.

Загрузку изображений в ImageView можно выполнить непосредственно из текущей папки проекта Sourse

```
view.setImage(new Image("/монетка.jpg"));
```

Обратите внимание имя файла как url надо начинать с /.

Лучше создать папку Resource в корневой папке проекта, обязательно указатв путь к ней в настройках проекта и размещать в ней все необходимые изображения.

Пример отображения изображения тремя различными способами:

```
Pane pane = new HBox(10);      // контейнер для изображений
pane.setPadding(new Insets(5, 5, 5, 5));
Image image = new Image("image/mypicture.gif"); // загрузка
pane.getChildren().add(new ImageView(image)); // добавление
элемента отображения изображения в контейнер
ImageView imageView2 = new ImageView(image); //
imageView2.setFitHeight(100); // установка размеров
imageView2.setFitWidth(100); // вывода отображения
pane.getChildren().add(imageView2);
ImageView imageView3 = new ImageView(image);
imageView3.setRotate(90);  // поворот изображения на 90 град
pane.getChildren().add(imageView3);
```

Пошаговая реализация программы «Раскраска», использующей паттерны Одиночка и Строитель

Программа «Раскраска» позволяет создать рисунок с использованием заданных параметров отображения. В программе паттерн Одиночка используется для инкапсуляции параметров фигуры (толщины контура и цвета заливки), по которым будет отображены фигуры на полотне.

При первом обращении к классу происходит инициализация и создание скрытой ссылки на экземпляр класса внутри паттерна. Данная ссылка хранит все параметры. При последующем обращении к классу будет возвращена созданная при первом обращении ссылка. Во время отрисовки фигуры происходит обращение к классу – Одиночке, который возвращает заданные ранее параметры. На 0 изображена диаграмма классов программы «Раскраска».

Шаг 1. Опредление интерфейса *Item*, который представляет часть продукта для создания.

```
public interface Item {
    public void draw(GraphicsContext gc, Canvas canvas);

}
```

Далее выполнются три реализации этого интерфейса. В программе продуктом для создания является изображение, и у него есть три части: Трава, Дом, Дерево.

```
public class House implements Item {
    public void draw(GraphicsContext gc, Canvas canvas) {
        Parameters_Singleton singleton =
Parameters_Singleton.getInstance(Color.DARKGREY, 5);
        gc.setFill(singleton.getColor());
        gc.setLineWidth(singleton.getLineWidth());
        gc.fillRect(30, 200, 200, 250);
        gc.setFill(Color.BLUE);
        gc.fillRect(120, 250, 60, 60);
        gc.setFill(Color.BLACK);
        double[] xPoints = {20,130 , 240};
        double[] yPoints = {210, 100, 210};
```

```
        gc.fillPolygon(xPoints, yPoints, 3);

    }
}
```

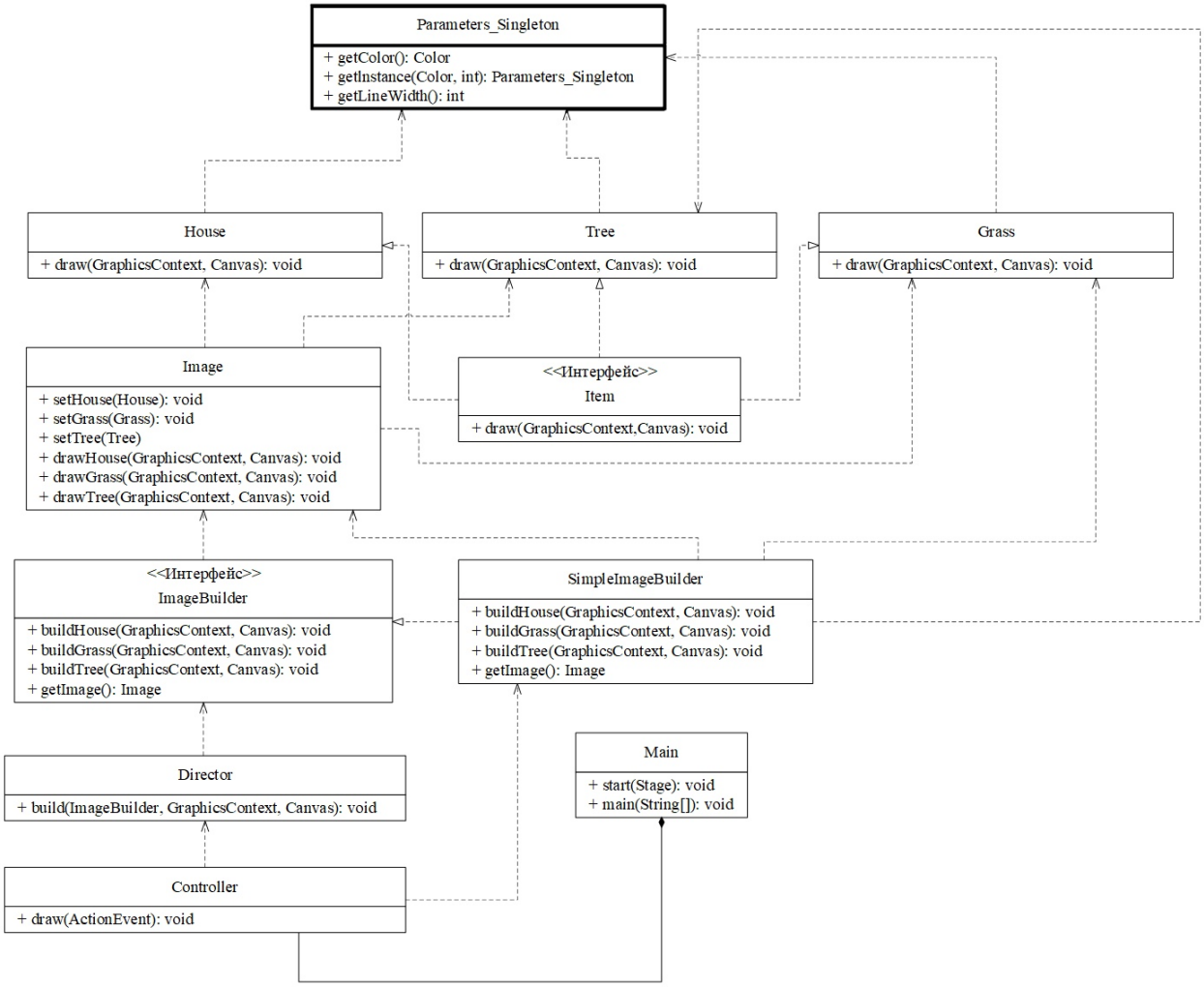


Рис. 44. Диаграмма классов приложения с выделением паттерна Одиночка

```
public class Grass implements Item {
    public static final double CANVAS_HEIGHT = 425;
    public void draw(GraphicsContext gc, Canvas canvas) {
        Parameters_Singleton singleton =
Parameters_Singleton.getInstance(Color.GREENYELLOW, 2);
        gc.setLineWidth(singleton.getLineWidth());
        gc.setFill(Color.ROYALBLUE);
        gc.fillRect(0, 0, canvas.getWidth(), canvas.getHeight());
        gc.setFill(Color.DARKGREEN);
        gc.fillRect(0, CANVAS_HEIGHT, canvas.getWidth(), 70);
    }
}
public class Tree implements Item {
    public void draw(GraphicsContext gc, Canvas canvas) {
```

```
        Parameters_Singleton singleton =
Parameters_Singleton.getInstance(Color.GREEN, 5);
        gc.setFill(singleton.getColor());
        gc.setLineWidth(singleton.getLineWidth());
        gc.fillRect(400, 350, 20, 75);
        gc.setFill(Color.LIGHTGREEN);
        double[] xPoints = {350,400, 470};
        double[] yPoints = {365, 240, 365};
        gc.fillPolygon(xPoints, yPoints, 3);
    }
}
```

Шаг 2. Создание класса – Одиночки.

Каждая из реализаций интерфейса *Item* имеет метод *draw*, в котором производится вызов паттерна Одиночки, управляющего параметрами создаваемой фигуры. Задаёт цвет и толщину линии:

```
public final class Parameters_Singleton {
    private static Parameters_Singleton _instance = null;
    private Color color;
    private int lineWidth;
    private Parameters_Singleton(Color color, int lineWidth) {
        this.color = color;
        this.lineWidth = lineWidth;
    }
    public static synchronized Parameters_Singleton
getInstance(Color color, int lineWidth){
        if (_instance == null)
            _instance = new Parameters_Singleton(color,
lineWidth);
        return _instance;
    }
    public Color getColor(){
        return color;
    }
    public int getLineWidth(){
        return lineWidth;
    }
}
```

Шаг 3. Опредление продукта, который будет конструироваться в интерфейсе, управляющим созданием продукта. Продукт включает в себя все создаваемые части и представляет собой изображение, которое будет рисоваться на полотне:

```
public class Image {
    private House house;
    private Grass grass;
    private Tree tree;
    public void setHouse(House house) {
        this.house = house;
    }
    public void setGrass(Grass grass) {
```

```
        this.grass = grass;
    }
    public void setTree(Tree tree) {
        this.tree = tree;
    }
    public void drawHouse(GraphicsContext gc, Canvas canvas){
        house.draw(gc,canvas);
    }
    public void drawGrass(GraphicsContext gc, Canvas canvas){
        grass.draw(gc,canvas);
    }
    public void drawTree(GraphicsContext gc, Canvas canvas){
        tree.draw(gc,canvas);
    }
}
```

Шаг 4. Определение интерфейса для создания конкретного продукта, включающего несколько частей:

```
public interface ImageBuilder {
    public void buildHouse(GraphicsContext gc, Canvas canvas);
    public void buildGrass(GraphicsContext gc,
javafx.scene.canvas.Canvas canvas);
    public void buildTree(GraphicsContext gc,
javafx.scene.canvas.Canvas canvas);
    public Image getImage();
}
```

Реализация этого интерфейса для создания и отрисовки конечного изображения:

```
public class SimpleImageBuilder implements ImageBuilder {
    private Image image;
    public SimpleImageBuilder(){
        this.image = new Image();
    }
    public void buildHouse(GraphicsContext gc, Canvas canvas) {
        this.image.setHouse(new House());
        this.image.drawHouse(gc,canvas);
    }

    public void buildGrass(GraphicsContext gc, Canvas canvas) {
        this.image.setGrass(new Grass());
        this.image.drawGrass(gc,canvas);
    }
    public void buildTree(GraphicsContext gc, Canvas canvas) {
        this.image.setTree(new Tree());
        this.image.drawTree(gc,canvas);
    }
    public Image getImage() {
        return this.image;
    }
}
```

Шаг 5. Определение распорядителя, который занимается созданием

экземпляра класса, который выполняет построение изображения:

```
public class Director {
    public void build(ImageBuilder builder, GraphicsContext gc,
Canvas canvas){
        builder.buildGrass(gc, canvas);
        builder.buildHouse(gc, canvas);
        builder.buildTree(gc, canvas);
    }
```

Шаг 6. Создание fxml-файла с разметкой, представляющего графический интерфейс программы.

Шаг 7. Использование паттернов Строитель и Одиночка. При нажатии на единственную кнопку в приложении активизируется обработчик события *drawPicture*, который, в свою очередь вызывает создание всего продукта (графического изображения), в методах отрисовки которого проис-ходит обращение к Одиночке. Сначала в методе происходит получение графического устройства, а затем создается экземпляр классов распорядитель «Director» и класс, который создаёт конкретный продукт «SimpleImageBuilder». Далее у распорядителя вызывается метод *build,* в который передаётся создатель продукта.

ШАБЛОН 6. ИТЕРАТОР

Цель применения: унифицированная навигация по любому составу объектов.

Проблема. Необходимо организовать просмотр изображений в виде слайд-шоу. Изображения хранятся в указанной папке, которая может иметь вложенные папки. Требуется создать возможность перебора содержимого папок с учетом вложенности и с фильтрацией по заданному правилу.

Описание

Паттерн Итератор предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления. Итератор – это интерфейс, включающий необходимые методы для просмотра всех элементов структуры данных. Наиболее распространенные методы:

hasNext () – возвращает true, если все еще в структуре есть элементы для просмотра, и false, если не осталось ни одного.

next () – возвращает следующий элемент в структуре данных.

Диаграмма классов построения приложения по паттерну Итератор представлена на 0.

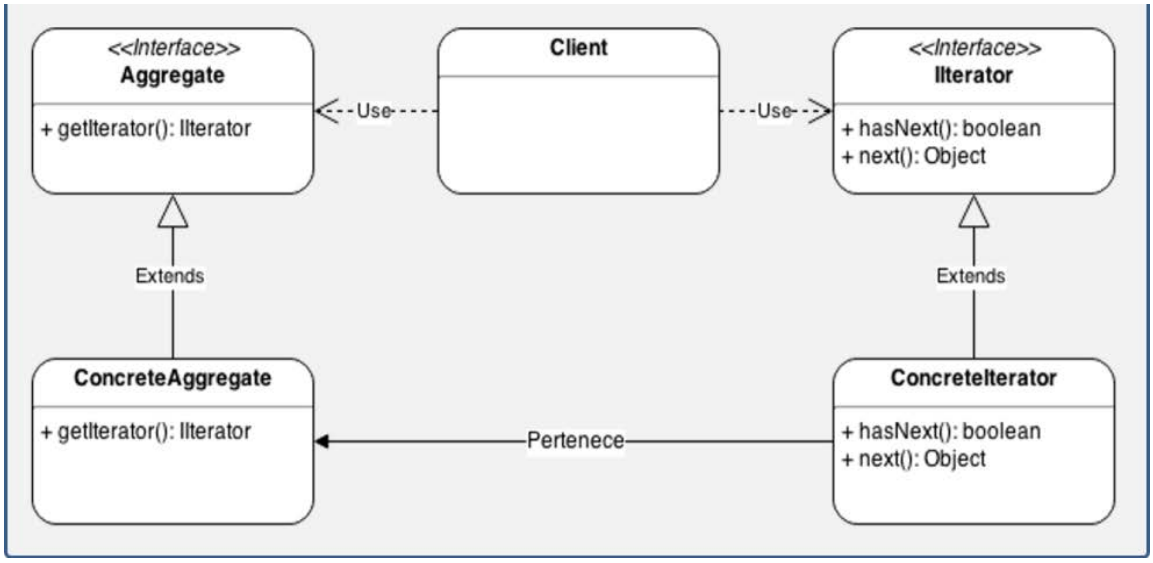


Рис. 45. Диаграмма классов шаблона Итератор

Основными компонентами являются:

Клиент (Client) – объект, использующий Итератор.

Aggregate – интерфейс, определяющий структуры классов, которые можно повторять.

ConcreteAggregate – класс, содержащий структуру данных, которую требуется перебрать.

IIterator – Интерфейс, определяющий структуру итератора, включая необходимые методы для выполнения итерации с помощью ConcreteAggregator.

ConcreteIterator – представляет конкретную реализацию итератора, которая будет отвечать за итерацию данных ConcreteAggregate.

Диаграмма последовательности модели Итератор на рис. 46.

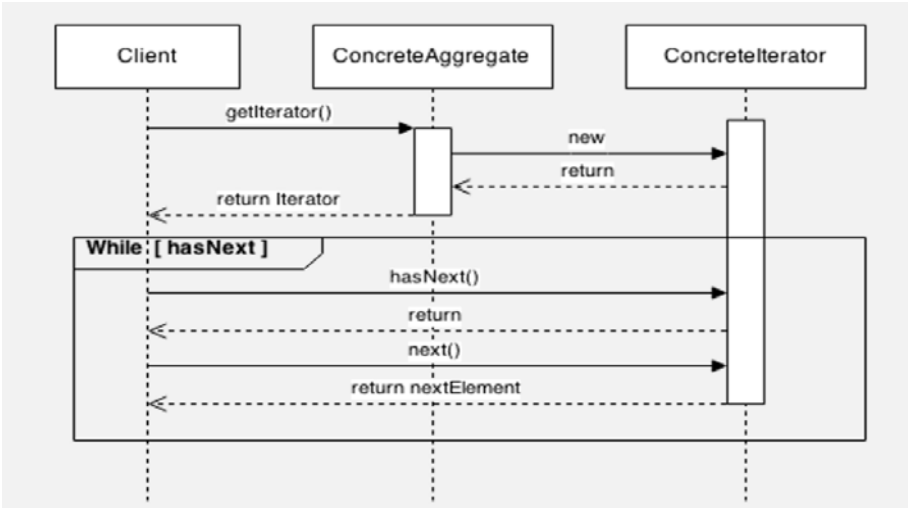


Рис. 46. Диаграмма последовательности работы приложения
по паттерну Итератор

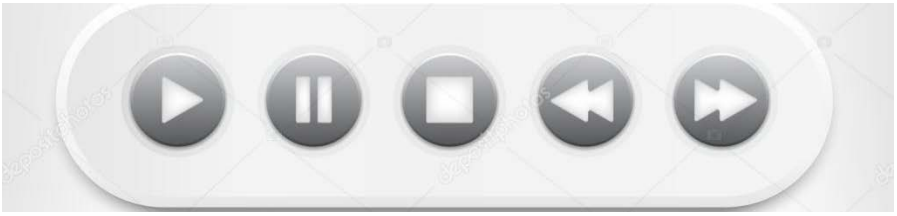
Задание и указания по выполнению

Постановка задачи

Разработайте программу для просмотра изображений, хранящихся в заданном каталоге указанным пользователем расширением. Порядок следования слайдов определен нумерацией в названии изображений.

Указания по выполнению

1. Разработайте пользовательский интерфейс по аналогии с плеером, включающим ImageView для показа слайдов, кнопки Запуска и Останова показа, выбора Каталога (Папки) и задания формата изображения (*.png, *jpg).



2. Организуйте считывание файлового каталога во внутреннюю структуру типа Дерево.

3. Используйте паттерн Итератор для навигации по структуре файлового каталога.

4. Реализуйте автоматический показ слад-шоу с использованием анимации.

Для организации слайд-шоу потребуется указать размещение каталога для хранения изображений, что следует предусмотреть в пользовательском интерфейсе.

Диаграмма классов приложения на 0.

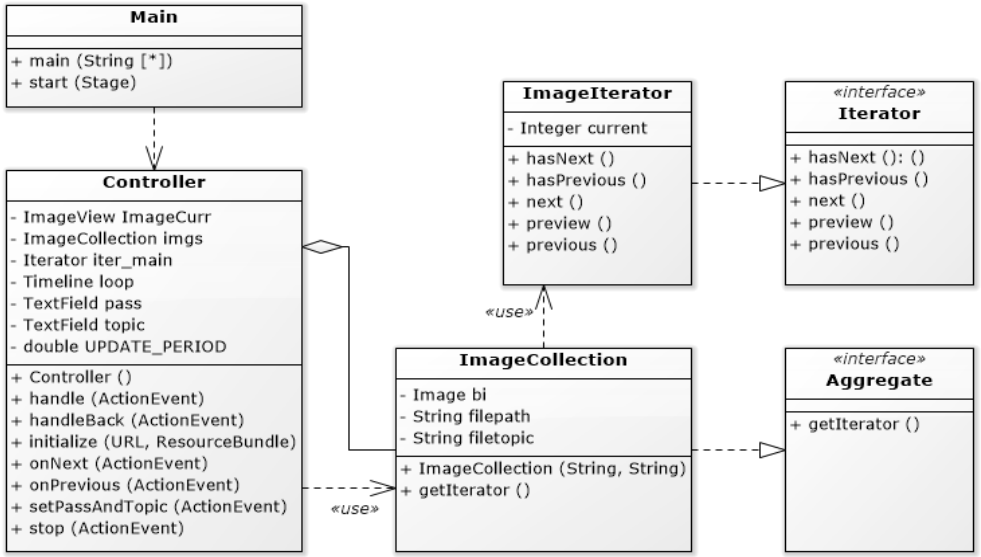


Рис. 47. Диаграмма классов приложения просмотра слайдов

Дополнительные задания

1. Реализуйте выбор и создания каталога для хранения слайдов презентации.

2. Реализуйте возможность сохранения выбранных файлов для презентации с дополнительными атрибутами – надпись, звук и пр.

3. Добавьте функции указания времени перехода от слайда к слайду, включения дополнительных визуальных эффектов к изображения, остановки и возобновление показа с произвольного места.

FX: Создание анимации

В JavaFX существуют различные подходы к анимации объектов. В основе поддержки анимации в JavaFX лежит абстрактный класс Animation, который является суперклассом для всех конкретных классов анимации. Есть два прямых подкласса анимации.

Анимация может быть связана со свойствами графических объектов и элементов управления, например размером, местоположением, цветом и т. п, что может быть задано с использованием эффектов и трансформацией. Transition является суперклассом для всех анимаций на основе переходов, таких как RotateTransition и ScaleTransition. Классы на основе переходов предоставляют предопределенные анимации, в которых JavaFX обрабатывает многие детали, что делает их очень простыми в использовании.

Второй вариант анимации использует шкалу времени. Класс Timeline позволяет определять собственную анимацию на основе ключевых кадров и ключевых значений. Объект Timeline предоставляет возможность обновлять значения свойств с течением времени. Для этого переходы состояний графической сцены объявляются начальным и конечным моментальными снимками (KeyFrame) состояния сцены в определенные моменты времени. Система может автоматически выполнять анимацию. Он может остановить, приостановить, возобновить, повернуть назад или повторить движение по запросу.

Для создания анимации по шкале необходимо:

А) установить временную шкалу Timeline timeline = new Timeline();

Б) добавить обработчик EventHandler<ActionEvent> к Timelines через javafx.animation.KeyFrame class;

В) в конструктор KeyFrame вставить javafx.util.Duration объект, который определяет время KeyFrame, и EventHandler<ActionEvent> object, с реализацией метода handle(ActionEvent e) который исполняется, когда заданное время кадра истечет, например KeyFrame frame1 = new KeyFrame(Duration.seconds(1), new MyHandler());

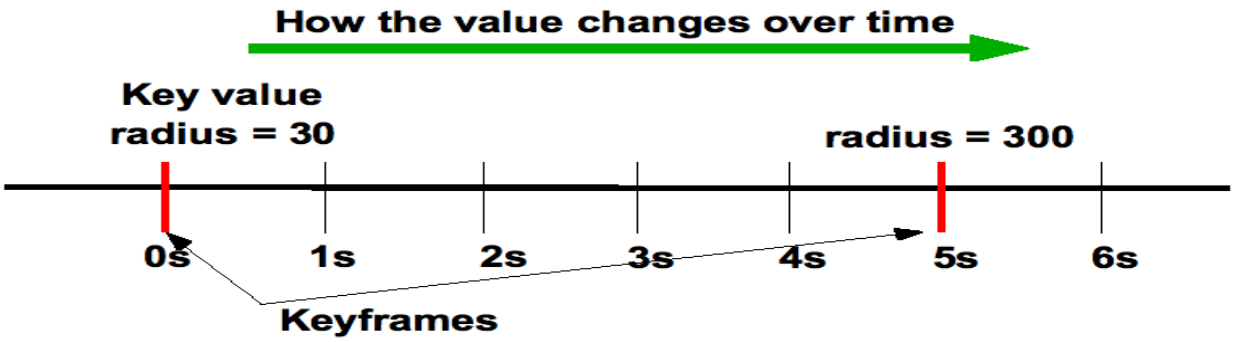


Рис. 48. Установка Timeline

В Timeline можно включать множество KeyFrames с различными длительностями и обработчиками:

```
например, timeline.getKeyFrames().addAll(frame1, frame2);
```

Для бесконечного повторения - Timeline.INDEFINITE

Для запуска анимации необходимо:

```
timeline.play();
```

Для повторного запуска с начала цикла:

```
timeline.playFromStart();
```

Для остановки:

```
timeline.stop().
```

Пример реализации заставки бесконечного движения шарика по экрану:

```
// Константы
  private static final int CANVAS_WIDTH = 640;
  private static final int CANVAS_HEIGHT = 480;
  private static final int UPDATE_PERIOD = 50; // milliseconds

          // Задание параметров шара
  private Circle ball;
  private int centerX = 280, centerY = 220;  // Center (x, y)
  private int radius = 190;
  private int xStep = 3, yStep = 5; // displacement per step in x, y

          // Установка шара в канву
    ball = new Circle(centerX, centerY, radius, Color.LIGHTSKYBLUE);
    canvas.getChildren().addAll(ball);

          // Установка временной шкалы и настройка кадров
    Timeline loop = new Timeline(new KeyFrame(Duration.millis(UPDATE_PERIOD), evt -> {
      // обновление координат (x, y)
      centerX += xStep;
      centerY += yStep;
      if (centerX + radius > CANVAS_WIDTH || centerX - radius < 0) {
        xStep = -xStep;
      }
      if (centerY + radius > CANVAS_HEIGHT || centerY - radius < 0) {
        yStep = -yStep;
```

```
        }
        ball.setCenterX(centerX);
        ball.setCenterY(centerY);
    }));

            // Запуск бесконечного числа циклов
    loop.setCycleCount(Timeline.INDEFINITE);
    loop.play();
  }
```

Пошаговая реализация

Шаг 0. Реализуем пользовательский интерфейс в ScienceBuider.

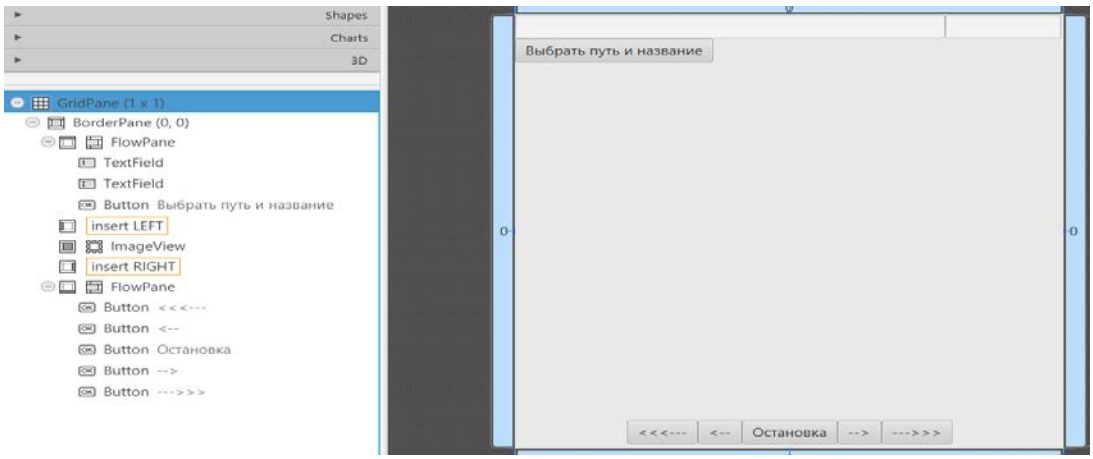


Рис. 49. Построение сцены приложения

Шаг 1. Создаем интерфейс **Iterator:**

```
public interface Iterator {
    public boolean hasNext();
    public boolean hasPrevious();
    public Object next();
    public Object previous();
    public Object preview();
}
```

Шаг 2. Создаем интерфейс **Aggregate:**

```
package model.interfaces;

public interface Aggregate {
    public Iterator getIterator();
}
```

Шаг 3. Создаем класс **ImageCollection** имплементирующий интерфейс **Aggregate:**

```
public class ImageCollection implements Aggregate {

    private String filetopic;
    private String filepath;
    private Image bi;


    public ImageCollection(String filepass, String filetopic) {
        this.filetopic = filetopic;
        this.filepath = filepass;
    }

@Override
    public Iterator getIterator() {
        return new ImageIterator();
    }
}
```

Шаг 4. В классе **ImageCollection** создаем еще один класс ImageIterator имплементирующий интерфейс **Interface:**

```
private class ImageIterator implements Iterator {

        private int current=0;
        @Override
        public boolean hasNext() {
            int k = new File(filepath).list().length;
            if (current == k - 1) {
                return false;
            } else {
                return true;
            }
        }

        public boolean hasPrevious() {
            if (current == 0){
                current = 4;
                return false;
            }
            else{
                current--;
                return true;
            }
        }

        public Object next() {
            if (this.hasNext()) {
                String filename = filetopic + (current) + ".jpg";
                bi = new Image(filename);
```

```
                current++;
                return bi;
            }
            else {
                String filename = filetopic + (current) + ".jpg";
                bi = new Image(filename);
                current = 0;
                return bi;
            }
        }

        public Object previous() {
            if (this.hasPrevious()) {
                String filename = filetopic + (current) + ".jpg";
                bi = new Image(filename);
                return bi;
            }
            else {
                String filename = filetopic + (current) + ".jpg";
                bi = new Image(filename);
                return bi;
            }
        }

        public Object preview() {
            return null;
        }
    }
```

Шаг 5. Создаем Controller для управления интерфейсом приложения:

```
public class Controller implements Initializable {
    public ImageCollection imgs = new
ImageCollection("..\image","Слайд");;
    public Iterator iter_main;
    public Timeline timeline;
    public ImageView ImageCurr;
    public Timeline loop;

    public static final double UPDATE_PERIOD = 500;

    public TextField topic;
    public TextField pass;

    public Controller() {
        this.iter_main = this.imgs.getIterator();
        this.timeline = new Timeline();
    }

    @Override
```

```
    public void initialize(URL location, ResourceBundle resources)
{
    }

    public void handle(ActionEvent t) {
        loop = new Timeline(new
KeyFrame(Duration.millis(UPDATE_PERIOD), evr-> {
            Image name = (Image) iter_main.next();
            ImageCurr.setImage(name);
        }));

        loop.setCycleCount(Timeline.INDEFINITE);
        loop.play();
    }

    public void handleBack(ActionEvent t) {
        loop = new Timeline(new
KeyFrame(Duration.millis(UPDATE_PERIOD), evr-> {
            Image name = (Image) iter_main.previous();
            ImageCurr.setImage(name);
        }));

        loop.setCycleCount(Timeline.INDEFINITE);
        loop.play();
    }

    public void onNext(ActionEvent actionEvent) {
        this.timeline.stop();
        if (this.iter_main.hasNext()) {
            Image name = (Image)this.iter_main.next();
            this.ImageCurr.setImage(name);
        }
    }

    public void onPrevious(ActionEvent actionEvent) {
        if (this.iter_main.hasPrevious()) {
            Image name = (Image)this.iter_main.previous();
            this.ImageCurr.setImage(name);
        }
    }

    public void stop(ActionEvent actionEvent) {
        loop.stop();
    }


    public void setPassAndTopic(ActionEvent actionEvent) {
        imgs = new ImageCollection(pass.getText(),
topic.getText());
    }
}
```

ШАБЛОН 7. АДАПТЕР

Цель применения: построить удобный интерфейс для доступа к имеющемуся объекту.

Проблема: Разрабатывается обучающая программа, демонстрирующая работу с различными структурами данных. Классы, определяющие структуры данных взяты из проверенного источника и не могут быть изменены. Необходимо адаптировать имеющиеся программные модули ко вновь разработанному пользовательскому интерфейсу.

Описание

Паттерн Адаптер (Adapter) предназначен для преобразования интерфейса одного класса в интерфейс другого. Реализации данного паттерна позволяет использовать вместе классы с несовместимыми интерфейсами. Диаграмма классов для реализации паттерна представлена на 0.

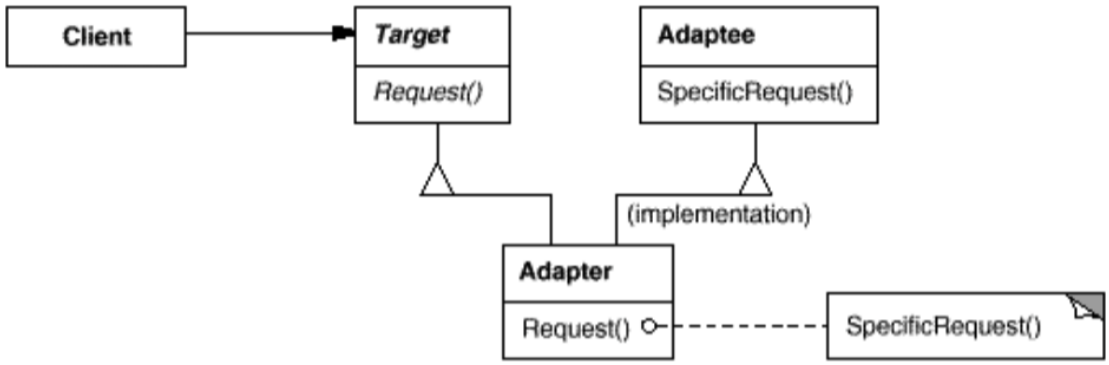


Рис. 50. Диаграмма классов шаблона Адаптер

Клиент (Client) – это класс, который содержит логику программы, которой требуется доступ к стороннему сервису.

Сервис (Adaptee) – сторонний интерфейс/класс, который Клиент не может использовать напрямую.

Адаптер (Adaptor) – это класс, который может одновременно работать и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.

Задание и указания по выполнению

Постановка задачи

Реализовать программу позволяющую работать с готовой реализацией некоторой «особой» структуры данных с набором собственных методов и приватных полей.

Нам необходимо создать простой и понятный интерфейс работы с этой структурой. При этом следует помнить, что реализацию структуры менять нельзя, следовательно, нужен класс-адаптер, обеспечивающий доступ к функциям пакета, через набор «стандартных» методов – операций со структурами данных – вставка (insert), удаление (remote), поиск элемента по заданному ключу (find).

Указания по выполнению

1. Разработайте пользовательский интерфейс (примеры рабочего окна приложения на 0).

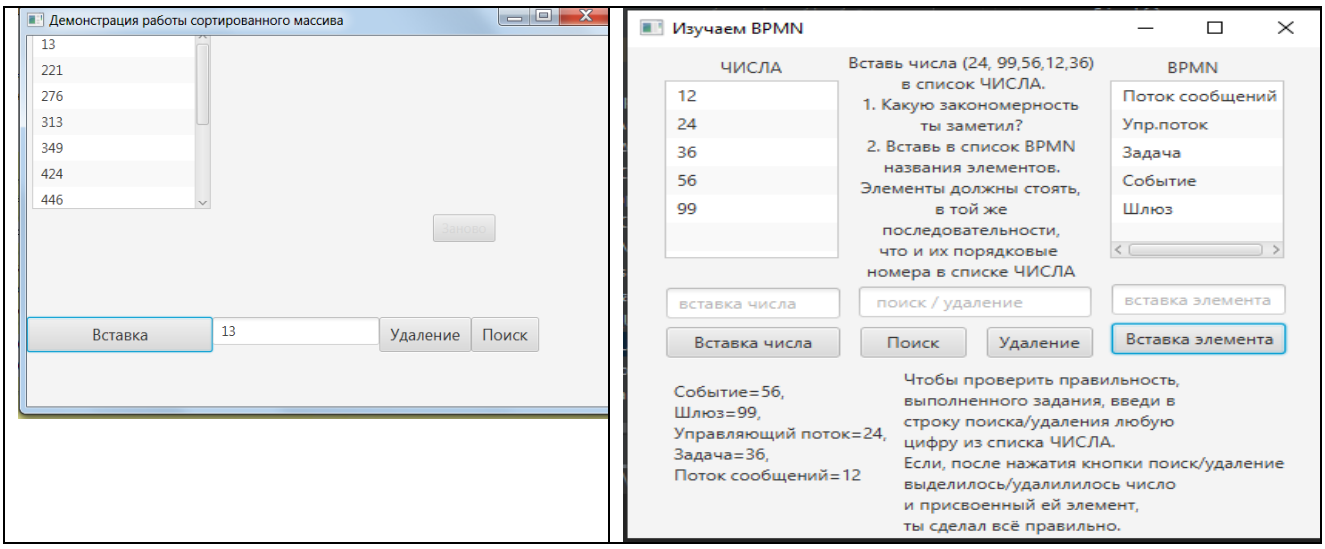


Рис. 51. Пример рабочего окна обучающих приложений, использующих готовые программные модули структур данных

В качестве клиента заданной структуры данных следует использовать элемент управления `ListView demolist`; и работать с ним для отображения текущего состояния структуры данных как:

```
demolist.getItems().clear();
demolist.getItems().addAll(demo.display());
```

где demo - объект класса Adapter, включенный в контроллер:

```
Adapter demo=new Adapter();
```

2. Для демонстрации работы со структурами данных интерфейс адаптера должен поддерживать *четыре метода*:

- вывод элементов в строку;
- вставка;

- удаление;
- поиск заданного пользователь элемента (по значению).

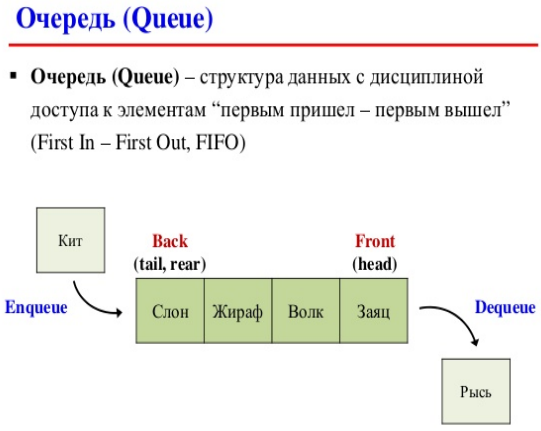
Пример реализации класса Адаптера:

```
public class Adapter {
    OrArray array; // включение сервисного класса структуры данных
    public Adapter() {

      array = new OrArray(1000);//полустатический массив - 
особенность реализации  OrArray

    }
  public ArrayList<String> display (){
    ArrayList<String> arr=new ArrayList<String>(maxSize);
    String[] subStr;
    String delimeter = " ";
    subStr = array.display().split(delimeter);
    for(int i=0; i<maxSize; i++) {

        arr.add(subStr[i]);
    }
    //(new Long(arr.get(i))).toString());
    return arr;
```

Обратите внимание, что реализация метода display() в «особой» структуре данных возвращает строку, содержащую элементы структуры данных через разделитель, которые следует распарсить в новый массив для вывода в элемент ListView.

```
public String display()              // displays array contents
{
   String str="";
    for(int j=0; j<nElems; j++)       // for each element,
        str+=a[j] + " ";  // display it
   return str;
}
```

Пошаговая реализация программы, демонстрирующей выполнение операций с очередью



Для демонстрации работы с очередью необходимо реализовать операции:
– вставка;
– извлечение.
Операции доступные для массива – поиск, к данной структуре данных не применимы, поэтому использовано копирование текущей очереди во временную структуру для извлечения ее элементов для искомого значения.

Шаг 1. Описание класса, описывающее структуру данных типа Очередь.

```
class Queue
   {
   private int maxSize;
   private long[] queArray;
   private int front;
   private int rear;
   private int nItems;
   public Queue(int s)          // constructor
      {
      maxSize = s;
      queArray = new long[maxSize];
      front = 0;
      rear = -1;
      nItems = 0;
      }
//--------------------------------------------------------------
   public void insert(long j)   // put item at rear of queue
      {
      if(rear == maxSize-1)         // deal with wraparound
         rear = -1;
      queArray[++rear] = j;         // increment rear and insert
      nItems++;                     // one more item
      }
//--------------------------------------------------------------
   public long remove()         // take item from front of queue
      {
      long temp = queArray[front++]; // get value and incr front
      if(front == maxSize)           // deal with wraparound
         front = 0;
      nItems--;                      // one less item
      return temp;
      }
//--------------------------------------------------------------
   public long peekFront()      // peek at front of queue
      {
      return queArray[front];
      }
//--------------------------------------------------------------
   public boolean isEmpty()    // true if queue is empty
      {
      return (nItems==0);
      }
//--------------------------------------------------------------
   public boolean isFull()     // true if queue is full
      {
      return (nItems==maxSize);
      }
//--------------------------------------------------------------
   public int size()           // number of items in queue
      {
      return nItems;
      }
//--------------------------------------------------------------
   }  // end class Queue
```

В соответствие с паттерном Адаптер (0) основные классы программы следующие:

Queue для работы с очередью.

Adapter, реализующий интерфейс Adaptee, содержащий методы отображения, вставки, удаления и поиска элемента в очереди.

Controller – связывает выбор пользователя (по нажатию кнопки) с конкретным обработчиком. Свойствами этого класса является объект типа Adapter, текстовые поля и элемент-список listView для отображения всей очереди.

Main осуществляет запуск fxml-окна приложения. Метод main() запускает выполнение приложения.

На 0 представлена диаграмма классов приложения.

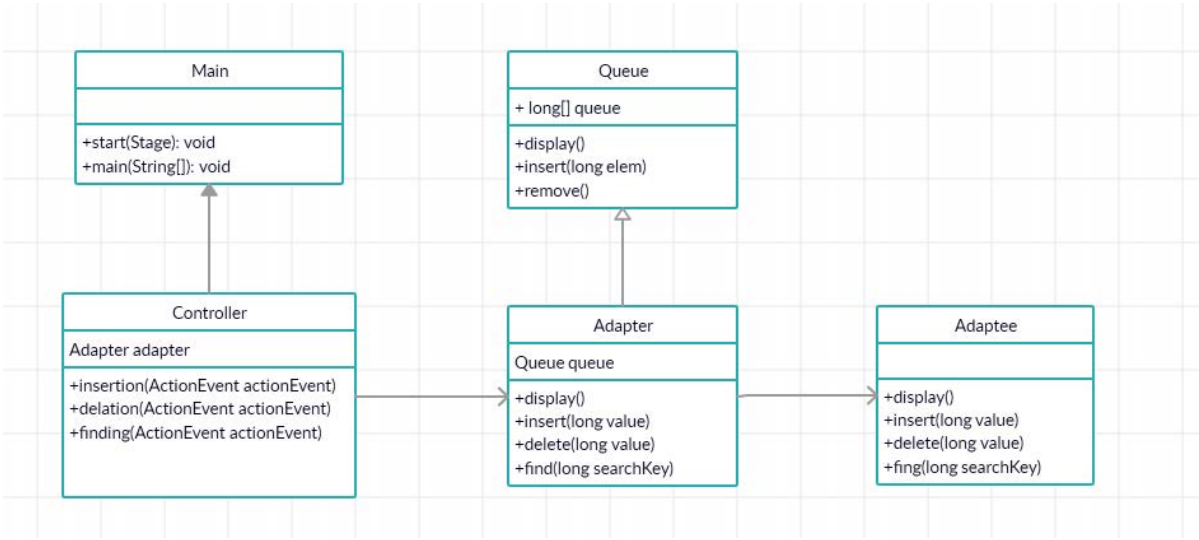


Рис. 52. Диаграмма классов приложения, демонстрирующего работу с очередью

Помимо внутреннего метода display() класса Queue, который формирует строку из элементов, разделенных через пробел, метод display() содержится и в классе Adapter. Данный метод формирует список ArrayList из элементов очереди путем разделения их по пробелу с помощью метода работы со строковым массивом split().

Далее представлено пошаговое описание реализации приложения.

Шаг 1. Определение интерфейса Adaptee:

```
public interface Adaptee {

        ArrayList<String> display();
        void insert(long value);
        boolean delete(long value);
        void find(long searchKey);

}
```

Шаг 2. Создание класса Adapter с необходимыми свойствами и реализацией отсутсвующих в агрегируем классе Queue методов.

Шаг 3. Создание fxml-файла с разметкой, который представляет будущий графический интерфейс пользователя.

Размещается элемент VBox (вертикальное размещение элементов), и верхнее поле растягивается в соответствии с раскадровкой (рис. 51). В верхнее поле добавляется элемент, отображающий список, listView. В нижнюю часть панели добавляется HBox (горизонтальное размещение элементов) и создаются кнопки Вставка, Удаление, Поиск и один элемент TextField для ввода пользователем элемента для поиска/удаления.

Шаг 4. Реализация в классе Controller обработки событий, вызванных действиями пользователя.

Создаются объекты textField и listView, объект класса Adapter и внутреннее свойство value с модификатором доступа private:

```
public TextField textField1;
public ListView arrView;
Adapter demo = new Adapter();
private long value;
```

На каждую кнопку создаются обработчики нажатия.

Кнопка вставить:

```
public void insertion(ActionEvent actionEvent) {
        value = Long.parseLong(textField1.getText());
        demo.insert(value);
        //--------
        arrView.getItems().clear();
        arrView.getItems().addAll(demo.display());
        //------------
        System.out.println("Добавлено!");
    }
```

Сразу же после изменения очереди обновляется список элементов listView. Значение value введено пользователем и содержится в поле textField.

Реализация обработчика кнопки «Удалить»:

```
public void delation(ActionEvent actionEvent) {
        value = Long.parseLong(textField1.getText());
        if (demo.delete(value)){System.out.println("Удалено!");}
        else {System.out.println("Удаление невозможно.");}
        //--------
        arrView.getItems().clear();
        arrView.getItems().addAll(demo.display());
        //------------
    }
```

Реализация обработчика кнопки «Поиск»:

```
public void finding(ActionEvent actionEvent) {
        value = Long.parseLong(textField1.getText());
        demo.find(value);
        if (demo.getFind()==demo.queue.getSize()){
            System.out.println("Элемент не найден!");
        }
        else {
            System.out.println("Позиция найденного элемента '" + value + "': " + demo.getFind());
        }
    }
```

ШАБЛОН 8. НАБЛЮДАТЕЛЬ

Цель применения: оперативно реагировать на изменения наблюдаемого объекта.

Проблема. Имеется некий сервер локального времени, который синхронизирует другие устройства. Эти устройства должны оперативно получать новые временные отчеты и выполнять запрограммированные на них собственные реакции.

Описание

Паттерн Наблюдатель (Observer) определяет зависимость «один-ко-многим» между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.

Диаграмма классов для реализации паттерна представлена на 0.

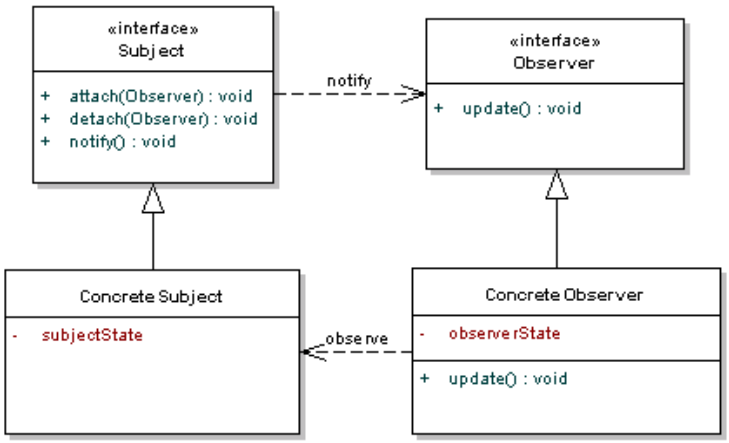


Рис. 53. Диаграмма классов паттрена Наблюдатель

Паттерн Observer определяет объект Subject, хранящий данные (модель), а всю функциональность «представлений» делегирует слабосвязанным отдельным объектам Observer.

При создании наблюдатели Observer регистрируются у объекта Subject.

Когда объект Subject изменяется, он извещает об этом всех зарегистрированных наблюдателей. После этого каждый обозреватель запрашивает у объекта Subject ту часть состояния, которая необходима для отображения данных.

Задание и указания по выполнению

Постановка задачи

Допустим, некий сервер локального времени считает секунды от момента своего запуска. Необходима синхронизация с ним трех различных визуальных компонентов:

К1: обновление тестового сообщения "прошло _ с";

К2: запуск музыкального клипа через заданное пользователем число секунд;

К3: запуск на выполнение некоторого процесса с заданным числом и периодом повторения (например, анимация).

Требуется реализовать программу в которой одни объект - выполняет некоторую задачу, а три другие отслеживают его состояния для реализации собственной функциональности в соответствии с ролями, представленными на CRC-карточке (0).

Класс	Роль	Функциональность	Дополнительные атрибуты/методы
TimeServer	Subject	Ведет отчет локального времени	*timeState* - счетчик секунд *tick()* - вызывается через определенный интервал внутренним таймером, обновляя состояние объекта и вызывает notify для извещения наблюдателей об изменении
ComponentOne	Observer	Отображает текущее локальное время	
ComponentTwo	Observer	Проигрывает музыкальный клип	start() - запуск воспроизведения stop() - остановить воспроизведение
ComponentThree	Observer	Вызывает перезапуск анимации	

Рис. 54. CRC-карточка адаптированная к ролям объектов в патерне Наблюдатель

Указания по выполнению

Шаг 1. Определите интерфейса Subject:

```
public interface Subject {
public void notifyAllObserver();
public void attach(Observer obs);
public void detach(Observer obs);
}
```

Шаг 2. Реализуйте интерфейс Subject в классе TimeServer (SubjectTimer) с созданием внутреннего отсчета времени.

Для создания собственного времени можно использовать класс Timer из пакета java.util.Timer, который запускает задачу класса TimerTask по расписанию, определяемому параметрами метода schedule (delay). Это следует сделать в конструкторе TimeServer следующим образом:

```
 public TimeServer() {
        this.timer = new Timer();
        task = new TimerTask() {
           public void run() {
                tick();
            }
        };
        timer.schedule(task, delay, period);
    }
```

Реализация метода tick() должна обязательно включать посылку оповещений наблюдателям:

```
private  void tick(){
        timeState++;
        notifyAllObservers();    }
```

Необходимо предусмотреть возможность обнуления сервера и получения значения текущего локального времени

```
public int getState() { ...
    }

public void setState(int time)   { ....
   }
```

Обязательно реализуемые методы для работы со списком наблюдателей private List<Observer> observers = new ArrayList<Observer>(); следующие:

```
public void attach(Observer observer){
    observers.add(observer);
}

public void detach(Observer observer){
    observers.remove(observer);
}

public void notifyAllObservers(){
    for (Observer observer : observers) {
        observer.update(this);
    }
}
```

3. Определите интерфейс Observer

```
public interface Observer {
       // protected Subject subject;
        public abstract void update(Subject st);
}
```

4. Создайте три класса Component, реализующих этот интерфейс и

предназначенных для различных визуализаций.

5. Создайте графический интерфейс пользователя для демонстрации работы приложения, включающий:

а) панель запуска сервера времени, включающий запуск и выключение отсчета локального времени, и состояние (активен/неактивен);

б) три панели для отображения визуальных компонентов К1, К2, К3 с панелями их настройки.

6. Реализуйте необходимые для функционарования приложения обработчики событий.

Пошаговая реализация программы синхронизации виджетов от запущенного пользователем таймера

В описываемой ниже программе реализованы четыре модели:

– таймер, ведущий отсчет времени;

– виджет «бегущая строка», отображающий текущее время таймера;

– виджет «звонок», запускающий музыкальный клип по истечению заданного времени;

– виджет «песочные часы», периодически повторяющий изображения.

Каждая из этих моделей описана отдельным классом, диаграмма классов приложения, представлена на 0.

Шаг 1. Создается абстрактный класс Observer и его реализация:

```
public abstract class Observer {
    protected Subject subject;
    public abstract void update(Subject st);
}
```

Шаг 2. Строится класс Subject. Этот объект будет обеспечивать управление временем и через Observer оповещать добавленные компоненты-виджеты об изменениях: функция start() запускает локальный сервер времени с заданными параметрами. Функция stop() останавливает локальное время, после чего отсчёт можно продолжить. Функция clean() отключает локальное время и сбрасывает параметры всех компонентов. Функции startCount() и stopCount() включают и отключают компонент отсчёта времени. Аналогичные функции реализованы для остальных виджетов.

Controller осуществляет управление паттерном observer(наблюдатель). Через интерфейс пользователь может запускать и останавливать локальное время приложения, а так же включать и выключать нужные виджеты.

```
public class Subject {
    Timer timer;
    private List<Observer> observers = new ArrayList<Observer>();
    int state;
    int d, p;
    public Subject() {
        this.state = 0;
        this.d = 0;
        this.p = 0;
    }
```

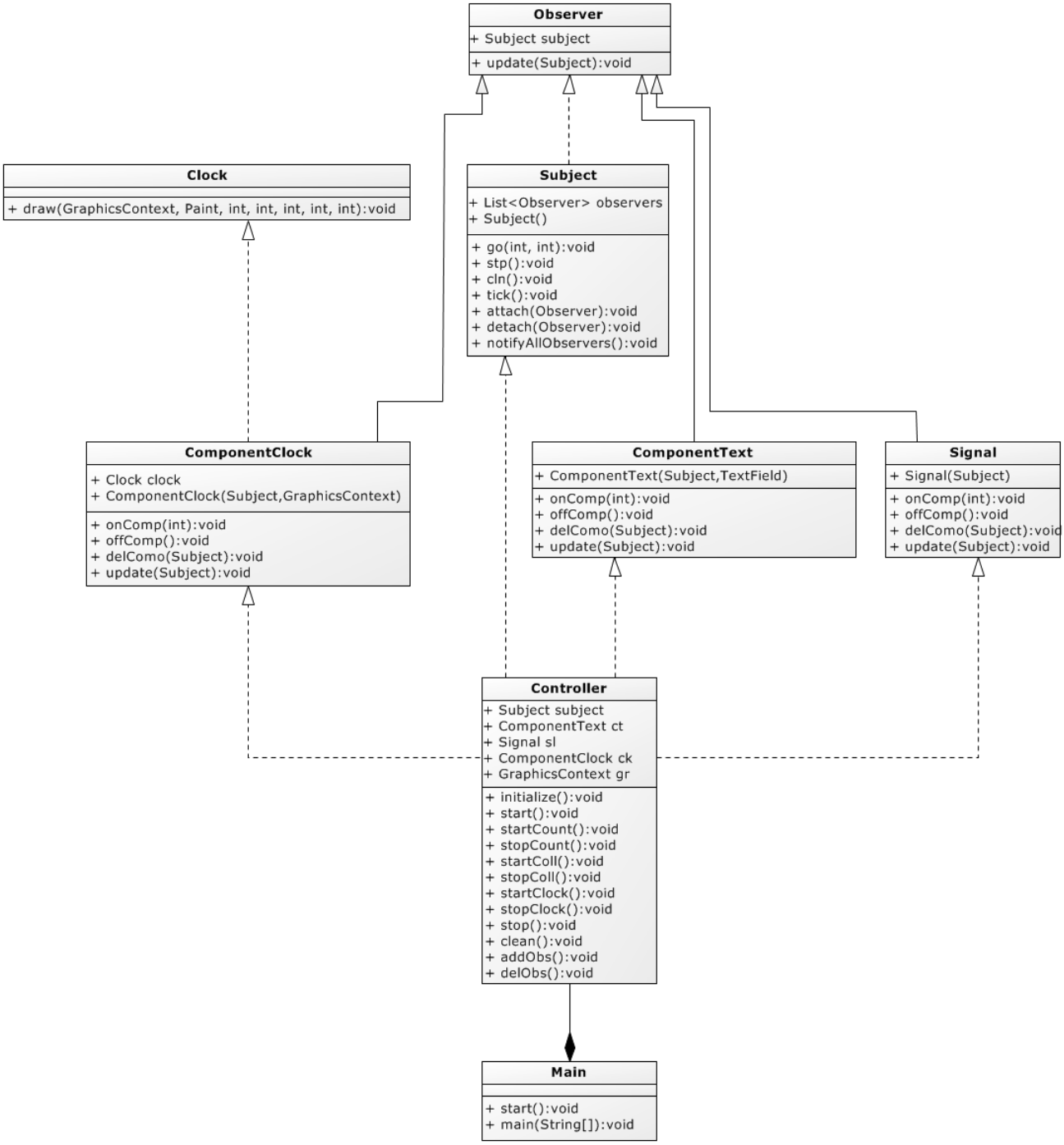


Рис. 55. Диаграмма классов приложения «Сервер времени с виджетами»

```
public void start(int d,int p){
    this.d = d;
    this.p = p;
    timer = new Timer();
    timer.schedule(new TimerTask() {
        @Override
        public void run() {
            tick();
        }
```

```
        }, d*1000, p*1000);
    }
    public void stop(){
        timer.cancel();
        System.out.println("STOP!");
    }

    public void clean(){
        state = 0;
        stop();
        System.out.println("CLEAN!");
    }
    private void tick(){
        if(state==0)
            this.state+=d;
        else this.state+=p;
        notifyAllObservers();
    }
    public int getState() {
        return this.state;
    }
    public void setState(int state) {

        this.state = state;
    }
    public void attach(Observer observer){
        observers.add(observer);
    }
    public void detach(Observer observer){
        observers.remove(observer);
    }
    public void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update(this);
        }
    }
}
```

Шаг 3. Создание компонентов-виджетов: ComponentText (поле с отсчётом времени), ComponentClock (анимированные песочные часы), Signal (звуковой сигнал).

```
public class ComponentText extends Observer {
    String txt;
    public TextField dindon;
    Boolean state;
```

```
    public ComponentText(Subject subject, TextField dindon){
        this.txt = "Прошло _ с";
        this.state = false;
        this.dindon = dindon;
        this.subject = subject;
        this.subject.attach(this);
    }
    public void onComp(){
        this.state = true;
    }
    public void offComp(){
        this.state = false;
    }
    public void delComo(Subject st){
        st.detach(this);
        dindon.setText("Прошло _ с");
    }
    @Override
    public void update(Subject st) {
        if(state) {
            txt = "Прошло " + st.getState() + " с";
            dindon.setText(txt);
        }
    }
}

public class Signal extends Observer {
    int count;
    int start;
    String file;
    Media sound;
    MediaPlayer mediaPlayer;
    Boolean state;
    public Signal(Subject subject){
        this.count = 0;
        this.state = false;
        this.start = subject.getState();
        this.file = "vivo.mp3";
        this.sound = new Media(new File(file).toURI().toString());
        this.mediaPlayer = new MediaPlayer(sound);
        this.subject = subject;
        this.subject.attach(this);
    }
    public void onComp(int count){
        this.count = count;
        this.start = subject.getState();
        this.state = true;
    }
```

```
    public void offComp(){
        this.state = false;
        mediaPlayer.stop();
    }
    public void delComo(Subject st){
        mediaPlayer.stop();
        st.detach(this);
    }
    @Override
    public void update(Subject st) {
        if (state) {
            if (st.getState() == start + count) {
                mediaPlayer.play();
                start += count;
            }
            if (st.getState() == start + 2)
                mediaPlayer.stop();
        }
    }
}
```

Обратите внимание! Для работы сигнала необходимо в корневую папку с проектом добавить звуковой файл (в примере имя файла vivo.mp3).

Реализация класса ComponentClock, который с течением времени перерисовывает песочные часы с новыми значениями соответствующих координат:

```
public class ComponentClock extends Observer {
    GraphicsContext gr;
    int x1;
    int x2;
    int y1;
    int y2;
    int y3;
    int count;
    int start;
    Paint p;
    Boolean state;
    Clock clock;
    public ComponentClock(Subject subject,GraphicsContext gr){
        this.clock = new Clock();

        this.state = false;
        this.gr = gr;
        this.x1 = 105;
        this.x2 = 275;
        this.y1 = 45;
        this.y2 = 45;
        this.y3 = 155;
        this.count = 0;
        this.start = subject.getState();
```

```
        this.p = Color.KHAKI;
        this.subject = subject;
        this.subject.attach(this);
    }
    public void onComp(int count){
        this.count = count;
        this.start = subject.getState();
        this.state = true;
    }
    public void offComp(){
        p = Color.KHAKI;
        this.state = false;
    }
    public void delComo(Subject st){
        p = Color.KHAKI;
        this.x1 = 105;
        this.x2 = 275;
        this.y1 = 45;
        this.y2 = 45;
        this.y3 = 155;
        gr.clearRect(0, 0, 380, 248);
        clock.draw(gr,p,x1,x2,y1,y2,y3);
        st.detach(this);
    }
    @Override
    public void update(Subject st) {
        if(state) {
            gr.clearRect(0, 0, 380, 248);
            if (st.getState() == start + count) {
                p = Color.CORAL;
                start += count;
            }
            if (st.getState() == start + 1) {
                p = Color.KHAKI;
            }
            clock.draw(gr, p, x1, x2, y1, y2, y3);
            if (y3 != 105) {
                x1 += 10;                x2 -= 10;
                y1 += 5;                y2 += 5;
                y3 -= 5;            }
     else {
                x1 = 105; x2 = 275;
                y1 = 45;  y2 = 45; y3 = 155;
            }
        }
    }
}
```

Для ComponentClock отдельно создаётся класс Clock рисующий на панели canvas песочные часы:

```
public class Clock {
    public void draw(GraphicsContext gr, Paint p, int x1, int x2,
int y1, int y2, int y3) {
        gr.setStroke(Color.BLACK);
        gr.setLineWidth(5);
        gr.strokePolygon(new double[]{90, 290, 190},
                new double[]{40, 40, 100}, 3
        );
        gr.strokePolygon(new double[]{90, 290, 190},
                new double[]{160, 160, 100}, 3
        );
        gr.setStroke(p);
        gr.strokeLine(190, 95, 190, 150);
        gr.setFill(p);
        gr.fillPolygon(new double[]{x1, x2, 190},
                new double[]{y1, y2, 95}, 3);
        gr.fillPolygon(new double[]{105, 275, 190},
                new double[]{155, 155, y3}, 3);
    }
}
```

Шаг 4. Реализация обработки событий по управлению таймером и виджетами в контроллере.

```
public class Controller {
    @FXML
    public TextField dindon, timer, clock, coll, repeat;
    public Label mess;
    public Canvas can;
    public Button start, stop, clean, stCount, spCount, stColl,
spColl, stClock, spClock;
    Subject subject = new Subject();
    ComponentText ct;
    Signal sl;
    ComponentClock ck;
    GraphicsContext gr;
    @FXML
    public void initialize(){
        gr = can.getGraphicsContext2D();
        addObs();
        dindon.setEditable(false);
        dindon.setText("Прошло _ с");
        Clock clock = new Clock();
        Paint p = Color.KHAKI;
        clock.draw(gr,p,190,190,95,95,105);
    }

    public void start(){
        mess.setText("Таймер активен");
subject.start(Integer.parseInt(timer.getText()),Integer.parseInt(r
epeat.getText()));
    }
```

```
    public void startCount(){
        ct.onComp();
    }
    public void stopCount(){
        ct.offComp();
    }
    public void startColl(){
        sl.onComp(Integer.parseInt(coll.getText()));
    }
    public void stopColl(){
        sl.offComp();
    }
    public void startClock(){
        ck.onComp(Integer.parseInt(clock.getText()));
    }
    public void stopClock(){
        ck.offComp();
    }
    public void stop(){
        mess.setText("Таймер остановлен");
        subject.stop();
    }
    public void clean(){
        mess.setText("Таймер неактивен");
        timer.setText("");
        repeat.setText("");
        coll.setText("");
        clock.setText("");
        delObs();
        addObs();
        subject.clean();
    }
    public void addObs(){
        ct = new ComponentText(subject,dindon);
        sl = new Signal(subject);
        ck = new ComponentClock(subject,gr);
    }
    public void delObs(){
        ct.delComo(subject);
        sl.delComo(subject);
        ck.delComo(subject);
    }
}
```

Результат создания приложения на рис. 56.

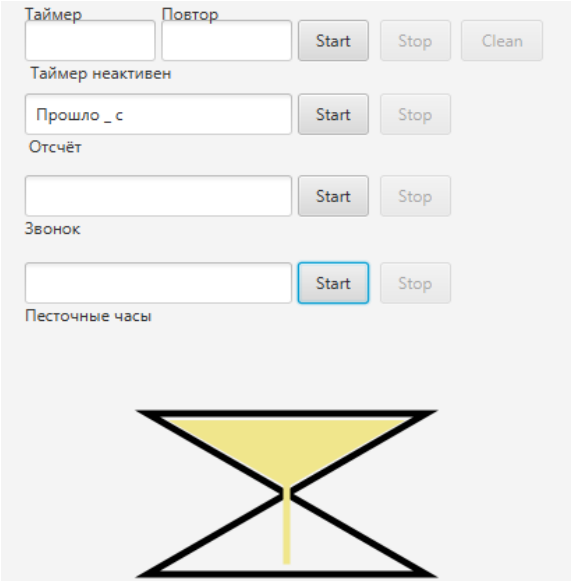


Рис. 56. Пример работы приложения «Сервер времени с виджетами»

ШАБЛОН 9. ПРОТОТИП

Цель применения: Создавать новые объекты, копируя их с ранее созданных.

Проблема. В графическом редакторе есть предопределённый набор фигур, необходимо обеспечить возможность включения фигур из этого набора в новую диаграмму.

Описание

Паттерн Prototype используется для создания новых объектов на основе прототипа. Прототип – это уже существующий в системе объект, который поддерживает операцию клонирования, то есть умеет создавать копию самого себя. Таким образом, для создания объекта некоторого класса достаточно выполнить операцию clone() соответствующего прототипа.

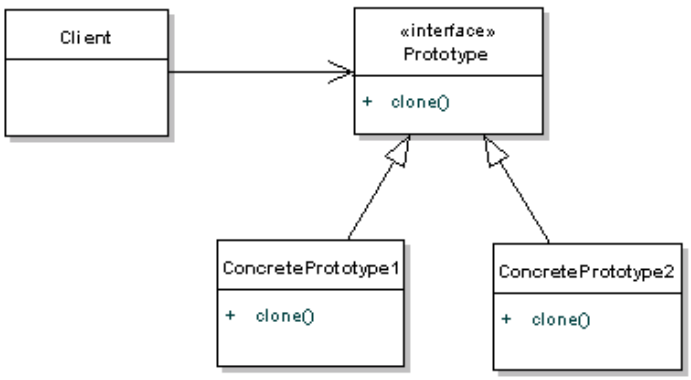


Рис. 57. Диаграмма классов паттрена Прототип

Паттерн Prototype реализует подобное поведение следующим образом.

- Все классы, объекты которых нужно создавать, должны быть подклассами одного общего абстрактного базового класса.
- Этот базовый класс должен объявлять интерфейс метода clone().

Также могут объявляться и другие общие методы, например, initialize() в случае, если после клонирования нужна инициализация вновь созданного объекта. В большинстве случаях достаточно реализовать в объекте интерфейс Cloneable.

Задание и указания по выполнению

Постановка задачи

Требуется написать программу, которая отображает выбранную пользователем фигуру (круг, треугольник, прямоугольник) из списка в произвольном месте окна.

Положение указывается щелчком мыши по панели, предназначенной для рисования.

Указания по выполнению

1. Разработайте пользовательский интерфейс, включающий ListBox и канву для рисования.

2. Каждая фигура должна быть представлена отдельным классом, унаследованным от класса Shape, содержащего абстрактные методы и реализующего интерфейс Cloneable

```
draw() для  рисования фигуры
public String getType()
public void setColor(Color )
```

и поля:

```
protected String type;
private Color colorbackgraund, colorline;
```

3. Реализуйте в Shape метод clone() - общий для всех наследников - для создания нового объекта - копии.

```
public Object clone() {
Object clone = null;
try {
clone = super.clone();
} catch (CloneNotSupportedException e) {
e.printStackTrace();   }
return clone; }
```

В JavaFX класс Controller не имеет конструктора, поэтому для инициализации полей fmxl-формы, например, заполнения списка, необходимо реализовать в нем интерфейс Initializable:
`public class Controller implements Initializable`

4. В классе Controller графического интерфейса пользователя создайте список фигур для компонента ListBox ***listboxforfigure.***

```
public void initialize(URL location, ResourceBundle resources){
        Square square =new Square();
        Rectangle rectangle=new Rectangle();
        Circle circle=new Circle();
        Triangle triangle=new Triangle();
        items = FXCollections.observableArrayList
(circle,square,rectangle,triangle);
       listboxforfigure.setItems(items);
listboxforfigure.getSelectionModel().setSelectionMode(SelectionMod
e.SINGLE);
    }
```

Допустимо реализовать `initialize()` без параметров через аннотирование:

```
@FXML
public void initialize() {
// здесь следует выполнить инициализацию Controller
}
```

При создании коллекции необходимо объявить
private ObservableList<Shape> itemsList; указав абстрактный класс элементов коллекции Shape - родитель всех фигур.

5. Реализуйте обработчик для рисования фигуры по канве **canvas** через GraphicsContext gc:

```
public void drawShape(MouseEvent mouseEvent) {


listView.getSelectionModel().setSelectionMode(SelectionMode.SINGLE
);
    int index =
listboxforfigure.getSelectionModel().getSelectedIndex();
//получение индекса выбора из списка
    Shape shape = (Shape) listboxforfigure.get(index).clone();//
создание копии фигуры
    gc.setFill(colorPicker.getValue());// установка цвета
заполнения фигуры по значению элемента управления colorPicker
    shape.draw(gc, mouseEvent.getX(), mouseEvent.getY());/
рисование копии фигуры в точке, полученной из события MouseEvent x
```

6*. Включите в программу дополнительные функции:

– выбор цвета фигуры с использованием компонента ColorPicker;

– вывод названия фигуры;

– очистка панели для рисования.

FX: Интерфейс Observable

Объекты Observable и Observer играют важную роль в реализации архитектуры Model-View-Controller в Java. Они обычно используются в системе, где один объект должен уведомлять другой о возникновении некоторых важных изменений. Observable – это класс, а Observer – это интерфейс. Они находятся в пакете java.util как часть Java Core Utility Framework.

Любой класс Java, заинтересованный в том, чтобы быть наблюдаемым, расширяет класс Observable. Расширенный класс затем может наблюдаться другими заинтересованными классами в других частях программы. Зарегистрированные классы уведомляются, как только происходит какое-либо изменение в наблюдаемом классе, в котором зарегистрирован класс наблюдателя. Это отношение между классами Observable и Observer. Классы Observer – это не что иное, как реализация интерфейса Observer. Интерфейс Observer предоставляет метод update (), который вызывается, когда наблюдатель уведомляется об изменении наблюдаемого объекта.

Класс, который хочет быть замеченным, или класс, который расширяет Observable, должен вызвать метод setChanged (), если он изменился, и должен инициировать уведомление, вызывая метод notifyObservers (). Это, в свою очередь, вызывает метод update () класса наблюдателя. Однако если объект вызывает метод notifyObserver () без вызова метода setChanged (), никаких действий не происходит. Следовательно, в наблюдаемом объекте должен вызываться метод setChange() до вызова метода notifyObservers(), чтобы впоследствии был вызван метод update () классов-наблюдателей.

Интерфейс JavaFX Observable является базовым интерфейсом для многих контейнерных классов и интерфейсов в JavaFX Collection Framework. ObserverList <E> обычно используется в элементах управления пользовательского интерфейса, таких как ListView и TableView.

Пример использования ObserverList в ListView:

```
public class ObservableListdemo extends Application {

   private final ObservableList<String> countries;
   private final ObservableList<String> capitals;

   private final ListView<String> countriesListView;
   private final ListView<String> capitalsListView;

   private final Button leftButton;
   private final Button rightButton;

   public ObservableListdemo() {
      countries = FXCollections.observableArrayList("Australia",
         "Vienna", "Canberra", "Austria", "Belgium","Santiago",
         "Chile", "Brussels", "San Jose", "Finland", "India");
      countriesListView = new ListView<>(countries);
```

```
        capitals = FXCollections.observableArrayList("Costa Rica",
            "New Delhi", "Washington DC", "USA", "UK", "London",
            "Helsinki", "Taiwan", "Taipei", "Sweden", "Stockholm");
        capitalsListView = new ListView<>(capitals);

        leftButton = new Button(" < ");
        leftButton.setOnAction(new ButtonHandler());

        rightButton = new Button(" > ");
        rightButton.setOnAction(new ButtonHandler());
    }
    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Страны и столицы");
        GridPane gridPane = new GridPane();
        Scene scene = new Scene(root, 500, 450);
        gridPane.add(countriesListView, 0, 1);
        gridPane.add(capitalsListView, 2, 1);
        VBox vbox = new VBox();
        vbox.getChildren().addAll(rightButton, leftButton);
    }

private class ButtonHandler implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        if (event.getSource().equals(leftButton)) {
            String str = capitalsListView.getSelectionModel()
                .getSelectedItem();
            if (str != null) {
                capitals.remove(str);
                countries.add(str);
            }
        } else if (event.getSource().equals(rightButton)) {
            String str = countriesListView.getSelectionModel()
                .getSelectedItem();
            if (str != null) {

countriesListView.getSelectionModel().clearSelection();
                countries.remove(str);
                capitals.add(str);
            }
        }
    }
```

Пошаговая реализация приложения для рисования фигур

Основные классы приложения (0):

1) фигуры для рисования, выполняют роль Прототипа для рисования на канве рабочего окна;

2) рабочее окно, выполняет роль холста для рисования фигур;

3) controller – класс управления, связывает выбор пользователя (выбранную фигуру) с процессом ее рисования в рабочем окне приложения.

Помимо обязательной реализации интерфейса Cloneable, каждый из классов фигур переопределяет метод toString() для того, чтобы список фигур, отображённый в главном окне, имел осмысленные названия фигур. Метод draw() – основной метод, отвечающий за отрисовку фигуры. Для однообразного обращения к объектам-фигурам используется наследование всех фигур от абстрактного класса Shape.

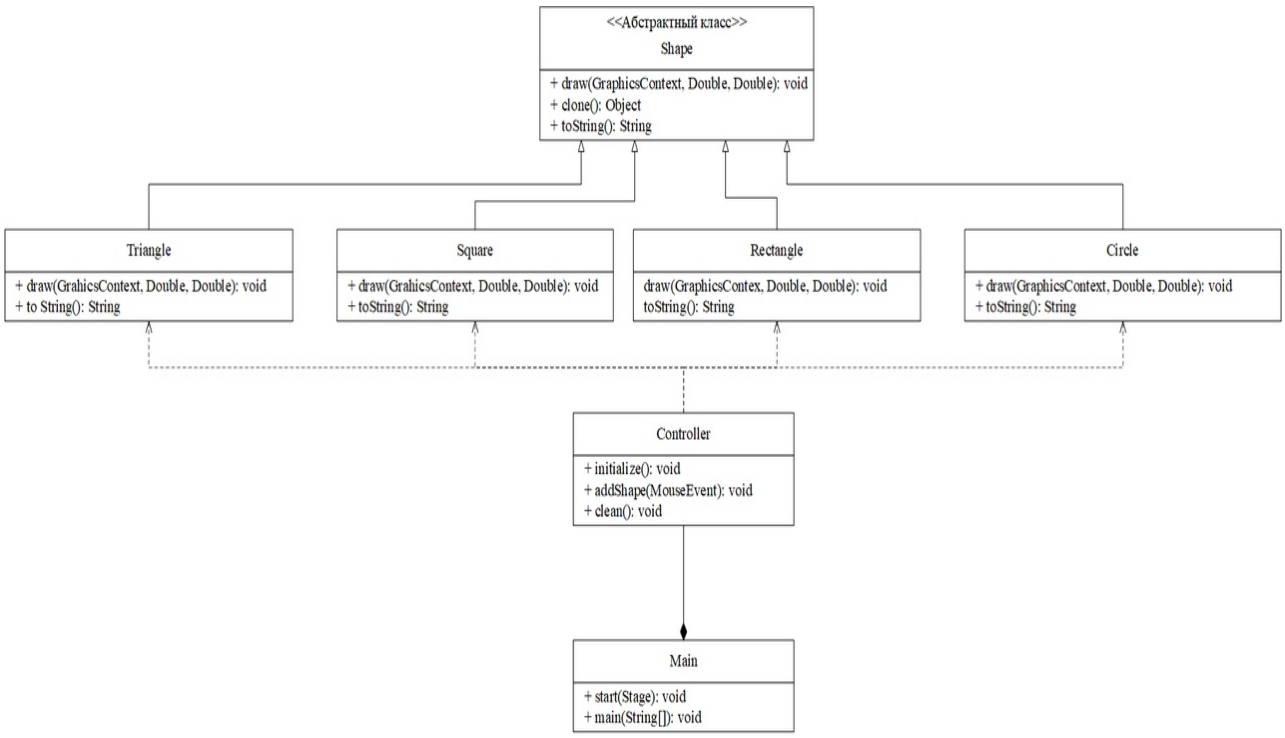


Рис. 58. Диаграмма классов приложения «Строим BPWN»

Шаг 1. Создадим абстрактный класс реализующий интерфейс Clonable. Данный класс является базовым для всех четырёх фигур. Он предоставляет метод draw, для отрисовки фигуры, и метод clone – для создания копии фигуры. Метод clone является частью паттерна «Прототип»:

```
public abstract class Shape implements Cloneable{
    protected String type;
    public abstract void draw(GraphicsContext gr, Double poinX,
Double poinY);
    public Object clone() {
        Object clone = null;
        try {
            clone = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
```

```
        return clone;
    }
    @Override
    public String toString() {
        return super.toString();
    }
}
```

Шаг 2. Реализация конкретных классов, расширяющих класс Shape.

Класс *Rectangle n*редоставляет геометрическую фигуру «прямоугольник».

```
public class Rectangle extends Shape{
   public Rectangle(){
            type="Прямоугольник";  }
        public   void draw(GraphicsContext gr, Double poinX, Double poinY) {
            gr.setFill(Color.RED);
            gr.fillPolygon(new double[]{poinX, poinX+25,poinX+25,poinX},
                    new double[]{poinY,poinY,poinY+75,poinY+75},);    }
        public String toString(){
        return "Rectangle";    }}
```

Аналогично реализуются все фигуры для заполнения списка.

Шаг 3. Создание fxml-файла с разметкой, представляющего будущий графический интерфейс пользователя. Создание происходит в несколько этапов:

Для начала в главном окне (экземпляр класса Scene) размещается элемент AnchorPane, который является контейнером для элементов, и позволяет позиционировать элементы внутри себя вдоль одной из сторон контейнера. В вышеописанный контейнер были добавлены:

– специальный элемент Canvas, который является низкоуровневым оболочкой над средствами отрисовки, которые предоставляет используемая ОС. На нём будет производиться отрисовка созданных фигур;

– компонент ListView, который отображает список всех фигур для отрисовки;

– компонент Button, который очищает панель от всех фигур.

Шаг 4. В классе Controller реализуется взаимодействие элемента управления ListView с действиями пользователя на холсте для рисования фигур.

1. Метод initialize() создаёт список фигур и помещает их в ListView:

```
public void initialize() {
    Square square =new Square();
    Rectangle rectangle=new Rectangle();
    Circle circle=new Circle();
    Triangle triangle=new Triangle();
    items = FXCollections.observableArrayList
(circle,square,rectangle,triangle);
    combo_sha.setItems(items);
}
```

2. Метод addShape является обработчиком события щелчка мышью по панели и использует вызов clone() для создания нового объекта:

```
public void addShape(MouseEvent x) {
    GraphicsContext gr = can.getGraphicsContext2D();

combo_sha.getSelectionModel().setSelectionMode(SelectionMode.SINGL
E);
    int a = combo_sha.getSelectionModel().getSelectedIndex();
    Shape new_figure = (Shape) items.get(a).clone();
    new_figure.draw(gr, x.getX(), x.getY());
}
```

3. Метод cleanCan очищает панель от всех нарисованных фигур:

```
public void cleanCan(){
    GraphicsContext gr = can.getGraphicsContext2D();
    gr.clearRect(0, 0, 548, 244);
}
```

Результат работы программы представлен на 0.

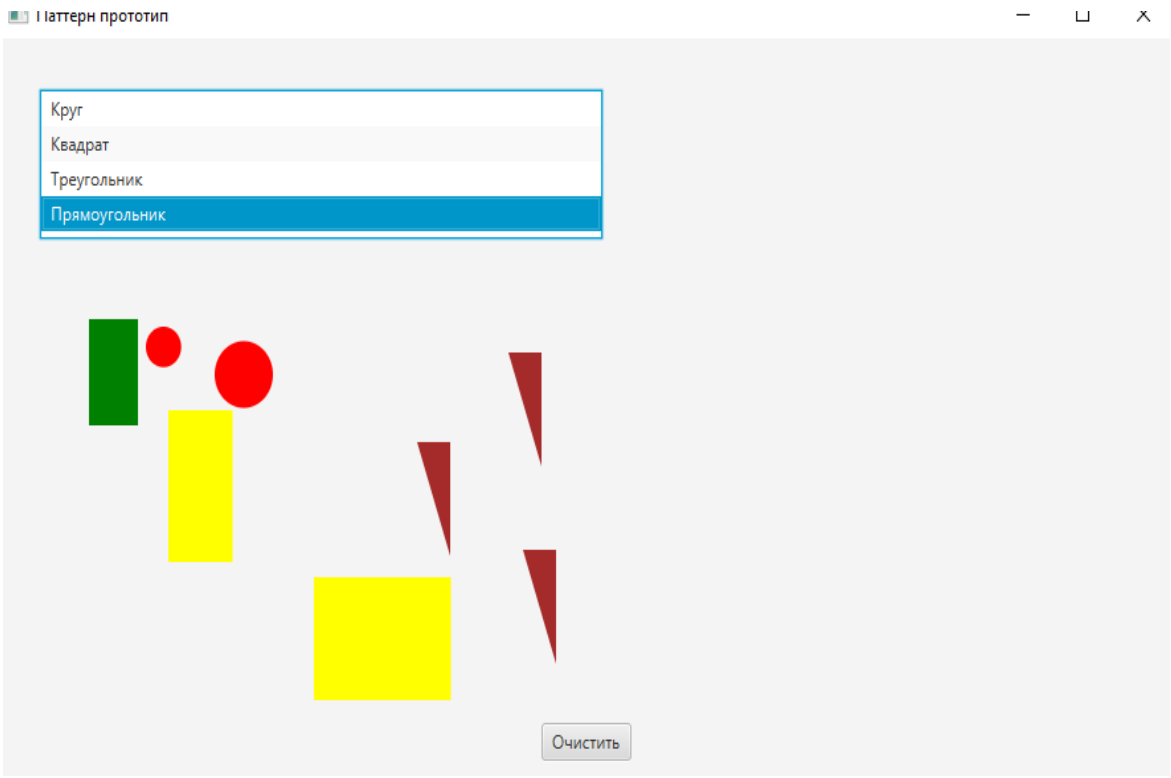


Рис. 59. Результат работы программы

ШАБЛОН 10. КОМПОНОВЩИК

Цель применения: структурирование сложных объектов и обеспечение однотипной работы как с простым, так и с сложным объектом.

Проблема. При реализации редактора диаграмм классов возникает необходимость изменить как одно поле, так и весь блок описания класса в зависимости от настроек пользователя. Паттерн Компоновщик группирует объекты в древовидные структуры и позволяет работать с ними так, если бы это был единичный объект.

Описание

Паттерн Компоновщик описывает, как можно применить рекурсивную композицию таким образом, что клиенту не придется проводить различие между простыми и составными объектами.

Структура составного объекта по Компоновщику представлена на диаграмме классов.

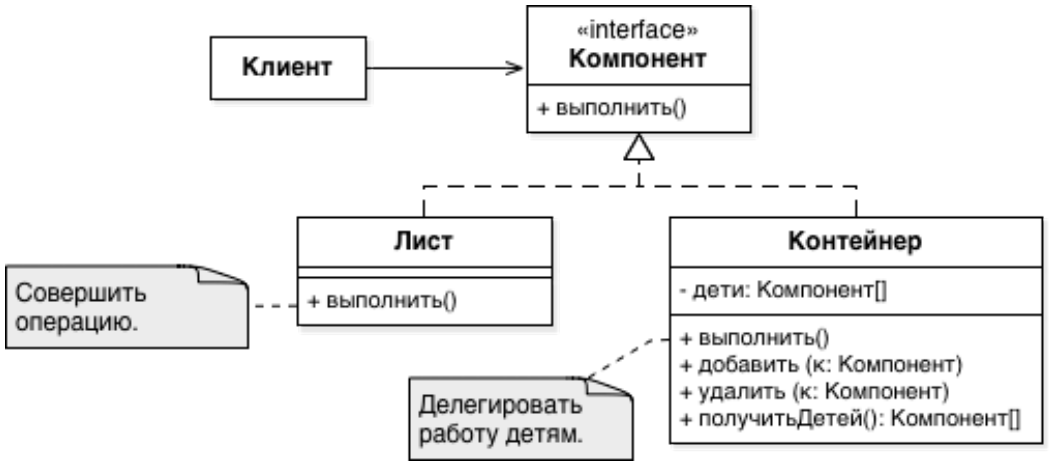


Рис. 60. Диаграмма классов паттерна Компоновщик

Компонент определяет общий интерфейс для простых и составных компонентов дерева.

Лист – это простой элемент дерева, не имеющий потомков (определяет поведение примитивных объектов в структуре).

Контейнер (Композиция/Структура) – это составной элемент дерева, содержит дочерние элементы – Листья или другие Контейнеры — но не знает какие именно, так как работает с ними только через общий интерфейс Компонента. Методы этого класса переадресуют основную работу своим дочерним компонентам, хотя и могут добавлять что-то своё к результату.

Клиент – работает с деревом через интерфейс Компонента, поэтому нет разницы что перед ним находится – простой или составной компонент дерева.

Задание и пример консольной реализации иерархической структуры хранения файлов

Условие задачи

Реализовать паттерн Компоновщик для древовидной структуры хранения Музыкальных дисков

Рис. 61. Пример каталога музыкальных треков

Шаг 1. Создание интерфейса-компонента Абстрактный файл:

```
interface AbstractFile {
    void ls();
}
```

Шаг 2. Реализация класса-листа Файл:

```
class File implements AbstractFile {
    private String name;

    public File(String name) {
        this.name = name;
    }

    public void ls() {
        System.out.println(CompositeDemo.compositeBuilder + name);
    }
}
```

Шаг 3. Реализация класса-контейнера Директорий:

```
class Directory implements AbstractFile {
    private String name;
```

```
    private ArrayList includedFiles = new ArrayList();

    public Directory(String name) {
        this.name = name;
    }

    public void add(Object obj) {
        includedFiles.add(obj);
    }

    public void ls() {
        System.out.println(CompositeDemo.compositeBuilder + name);
        CompositeDemo.compositeBuilder.append("   ");
        for (Object includedFile : includedFiles) {
            // Leverage the "lowest common denominator"
            AbstractFile obj = (AbstractFile) includedFile;
            obj.ls();
        }
        CompositeDemo.compositeBuilder.setLength
(CompositeDemo.compositeBuilder.length() - 3);
    }
}
```

Шаг 4. Создание класса-клиента для демонстрации работы паттерна Компоновщик:

```
public class CompositeDemo {
    public static StringBuffer compositeBuilder = new
StringBuffer();

    public static void main(String[] args) {
        Directory music = new Directory("MUSIC");
        Directory scorpions = new Directory("SCORPIONS");
        Directory dio = new Directory("DIO");
        File track1 = new File("Don't wary, be happy.mp3");
        File track2 = new File("track2.m3u");
        File track3 = new File("Wind of change.mp3");
        File track4 = new File("Big city night.mp3");
        File track5 = new File("Rainbow in the dark.mp3");
        music.add(track1);
        music.add(scorpions);
        music.add(track2);
        scorpions.add(track3);
        scorpions.add(track4);
        scorpions.add(dio);
        dio.add(track5);
        music.ls();
    }
}
```

Задание для самостоятельного выполения

Создайте структуру каталога вида:

MUSIC
--Don't worry, be happy.mp3
--SCORPIONS
----Wind of change.mp3
----Big city night.mp3
--ATHER
----Bon Jovi
------Slippery When Wet
--------Let It Rock.mp3
-------- Never Say Goodbye.mp3
------It's My Life.mp3
Rainbow in the dark.mp3

Обратите внимание, что чертой (-) обозначен автоматический отступ указывающие уровень вложенности и при правильной компоновке он должен отображаться в виде отступа следующего названия трека.

Указания к созданию модели диаграммы классов на основе паттерна Компоновщик

Требование к реализации функции рисования класса

Исходные данные: каждый компонент схемы представляет собой прямоугольник с текстом, который в свою очередь может быть включен как часть описания класса (поле или метод), так и как целый класс в пакет (рис. 62).

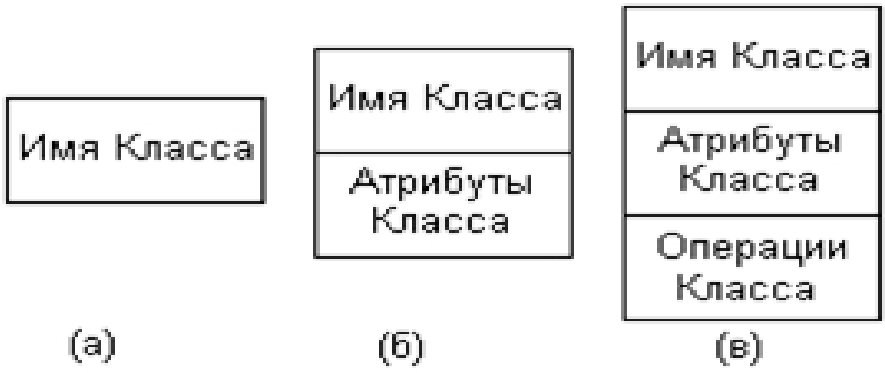


Рис. 62. Три варианта отображения класса на диаграмме

Необходимо реализовать модель Компоновщик (graphmodel) для создания графического объекта (рис. 62 – примеры отображаемых объектов) на диаграмме классов. Элементарный графический примитив диаграммы – прямоугольник с описанием заголовка или поля или метода класса, составной объект - несколько примитивов) в соответствии со схемой на 0

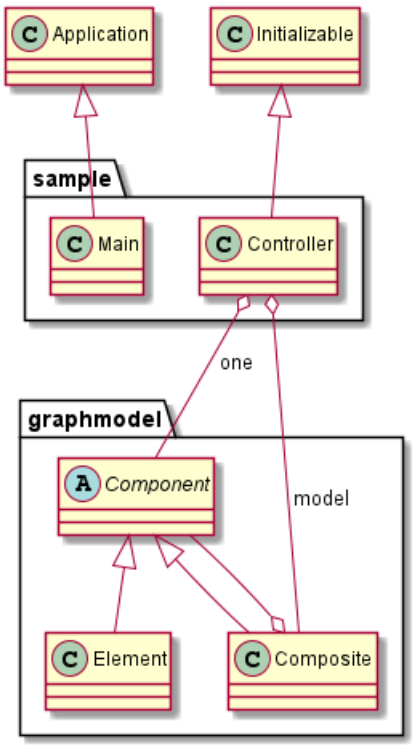


Рис. 63. Диаграмма классов программного модуля
отображения компонента диаграммы

Указания по реализации программного модуля

1. Определите обязанности классов:

Component – должен содержать общие для элементарного и составного объекта свойства, например:

```
 public abstract class Component{
    // описание поля класса
    Character prefix;
    String name;
    String type;
    // параметры рисования
    protected double x;
    protected double y;
    protected double w;
    protected double h;
    protected String style;
    // методы общие для Element и Composite, реализованные по-
разному
    public abstract void add (Component component) throws
ComponentException;
    public abstract void remove (Component component) throws
ComponentException;
    public abstract Component getChild (int index) throws
ComponentException;
    public abstract void draw(Pane pane);
    public abstract Node getPicture();
```

```
    public abstract void setX(double x);
    public abstract void setY(double y);
    public abstract void setW(double w) ;
    public abstract void setH(double h) ;
```

Класс Element – хранит данные о рисуемом объекте и отвечает за его отображение, поэтому необходимо решить каким способом будет нарисован прямоугольник (в примере для этой цели используется «готовый» компонент – Label, имеющая все необходимые свойства (координаты, стиль отображения и включает текст):

```
public class Element extends Component {
    @Override
    public void add(Component component) throws ComponentException
{
        throw new ComponentException();
    }

    public Element() {
        prefix=new Character('-');
        name=new String("varname");
        type=new String("typevar");
        x=1.;
        y=1;
        w=20;
        h=16;
        style="-fx-background-color: darkslateblue; -fx-text-fill:
black; -fx-border-insets: 3; -fx-border-width: 1; -fx-border-
style: solid;";
    }

    @Override
    public  void remove(Component component) throws
ComponentException {
        throw new ComponentException();
    }
    @Override
    public  Component getChild(int index) throws
ComponentException {
        throw new ComponentException();
    }
    @Override
    public void draw(Pane pane) {
        Label field = (Label) getPicture();// для отображения
        field.setPrefSize((pane.getWidth()*w/100.),
(pane.getHeight()*h/100.));
        field.setLayoutX(pane.getWidth()*x/100.);
        field.setLayoutY(pane.getHeight()*y/100.);
        pane.getChildren().add(field);
       }
    @Override
    public Node getPicture() {
```

```
        String text=new String(prefix.toString());
        text+=" "+name+":"+type;
        Label field = new Label(text);
        field.setStyle(style);
        return field;
    }

    public void setX(double x) {
        this.x = x;
    }
```

Composite отвечает за агрегацию элементарных объектов, поэтому его реализация другая:

```
public class Composite extends Component {

    ArrayList <Component> array=new ArrayList<>();// агрегатор
элементарных объектов
    public Composite(ArrayList<Component> array) {// может быть
реализован и с другим набором параметров
        this.array = array;
    }
    @Override
    public void add(Component component) throws ComponentException
{
        array.add(component);
    }
    @Override
    public Component getChild(int index) throws ComponentException
{
        return array.get(index);
    }
    @Override
    public void draw(Pane pane) {
        if (array.isEmpty()) return;
        for (Component comp: array) comp.draw(pane);
    }

    @Override
    public void setX(double x) {
        if (array.isEmpty()) return;
        for (Component comp: array) comp.setX(x);
       }
```

2. После создания graphmodel, реализуйте пользовательский интерфейс, позволяющий:

– выбрать один из стандартных компонентов диаграммы (см. варианты на рис. 10.3);

– отобразить его в произвольном (по щелчку мыши) месте диаграммы;

– установить стиль отображения всей диаграммы;

– переместить выбранный элемент диаграммы (как элементарный, так и

составной).

Пример реализации контроллера графического интерфейса для отображения graphmodel

Модель взаимодействия пользовательского интерфейса и graphmodel следующая:

1. Пользователь нажимает на кнопку (Один Элемент, Два Элемента, Три Элемента).

2. Пользователь получает соответствующие количеству, указанному на кнопке, названия элементов BPMN (Один Элемент – Задача; Два Элемента – Задача и Событие; Три Элемента – Задача, Событие и Шлюз).

3. Что бы очистить поле, пользователь нажимает кнопка «Очистка».

```
public class Controller implements Initializable {
//элементы пользовательского интерфейса
    public Pane paintpane;
    public VBox elementpane;
    private Map<String, Component> list_id=new HashMap<>();
    Component one=null;
    Composite model=new Composite();//объект диаграмма
    @Override
    public void initialize(URL location, ResourceBundle resources)
{//создание списка доступных элементов
        Element field=new Element();
        list_id.put(field.getPicture().getId(), field);
        elementpane.getChildren().add(field.getPicture());
        elementpane.getChildren().get(0).setStyle("-fx-background-
color: green; -fx-text-fill: white;");
        elementpane.getChildren().add(new Separator());
        Composite myclass=new Composite(new Rect(1,1,10,10));
        elementpane.getChildren().add(myclass.getPicture());
        elementpane.getChildren().add(new Separator());
        list_id.put(myclass.getPicture().getId(), myclass);
        ArrayList<Component> arr=new ArrayList<>();
         arr.add(new Element(' ', "ClassName", ""));
        arr.add(new Element(' ', "", ""));
        Composite fi1=new Composite(arr);
        elementpane.getChildren().add(fi1.getPicture());
        list_id.put(fi1.getPicture().getId(), fi1);
        elementpane.getChildren().add(new Separator());
    }
//отрисовка выбранного элемента на диаграмме
    public void onDraw(MouseEvent mouseEvent) {
        if(one==null) {System.out.println("ничего не
выбрано");return;}
        one.setX(mouseEvent.getX());///paintpane.getWidth()*100);
        one.setY(mouseEvent.getY());///paintpane.getHeight()*100);
        Rectangle d=new
Rectangle(mouseEvent.getX(),mouseEvent.getY(),10,10 );
                d.setFill(Color.RED);
        try {
```

```
            model.add(one);
         } catch (Component.ComponentException e) {

            e.printStackTrace();
         }
        one=null;
        paintpane.getChildren().clear();
        model.draw(paintpane);

        paintpane.getChildren().add(d);
    }
}
```

ШАБЛОН 11. ДЕКОРАТОР

Цель применения: динамически добавлять новые свойства к имеющемуся объекту, избегая порождения новых классов

Проблема. В системе автоматизированного проектирования необходимо предусмотерть возможность изменять изображение объекта, добавляя к нему новые элементы.

Описание

Паттерн Decorator действует как оболочка (обертка) для существующего класса, что позволяет пользователю добавлять новые функциональные возможности к существующему объекту, не изменяя его структуру.

Структура построения паттерна включает:

Component – базовый класс компонента, чье поведение будет расширяться новыми классами-декораторами;

ConcreteComponent – конкретная реализация компонента;

Decorator – базовый класс декоратора, предназначенный для расширения поведения компонента;

ConcreteDecoratorA, ConcreteDecoratorB – конкретный декоратор, который добавляет декорируемому объекту специфическое поведение.

Диаграмма классов и последовательность взаимодействия объектов представлены на рис. 64.

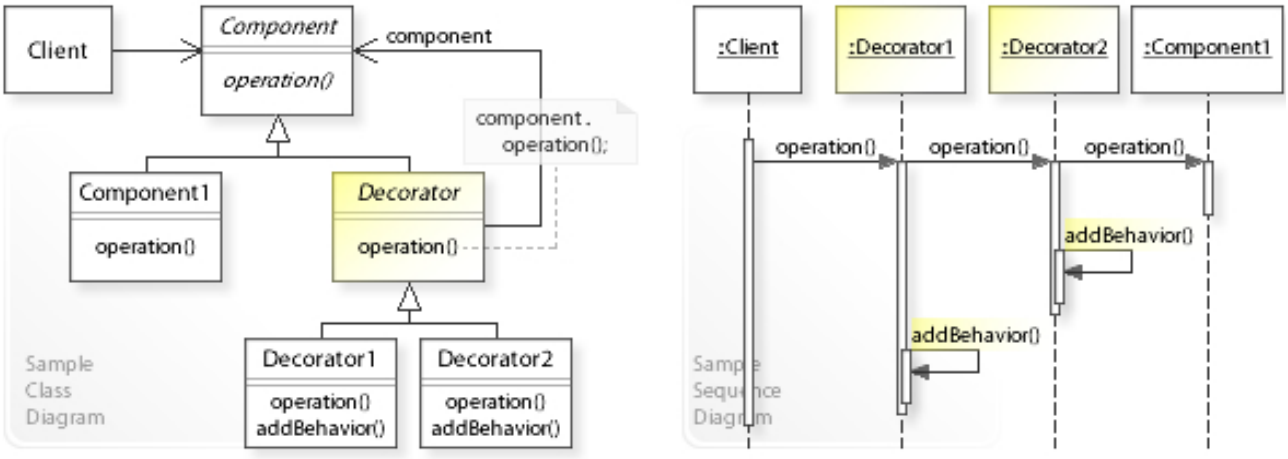


Рис. 64. Диаграмма классов и последовательностей
шаблона Декоратор

Паттерн Decorator реализует подобное поведение следующим образом.

Определяется один основной компонент и несколько дополнительных (необязательных) «оберток» и для них строится общий интерфейс.

Создается базовый класс второго уровня (Decorator) для поддержки дополнительных декорирующих классов-«оберток».

Для реализации каждой дополнительной функциональности строится класс, производный от Decorator, который реализует дополнительную функциональность и делегирует выполнение операции базовому классу Decorator.

Клиент несет ответственность за конфигурирование системы: устанавливает типы и последовательность использования основного объекта и декораторов.

Задание и указания по их выполнению

Постановка задачи

Требуется написать программу, которая наряжает (декорирует) елку в соответствии с выбором пользователя (пример внешнего вида приложения на рис. 65).

Исходные данные: изображение фигуры, например, елки, к которой с помощью элементов управления, например checkbox, можно добавить дополнительный функционал – гирлянду, игрушки, подарки под елку, звезду на макушку и прочие элементы украшения.

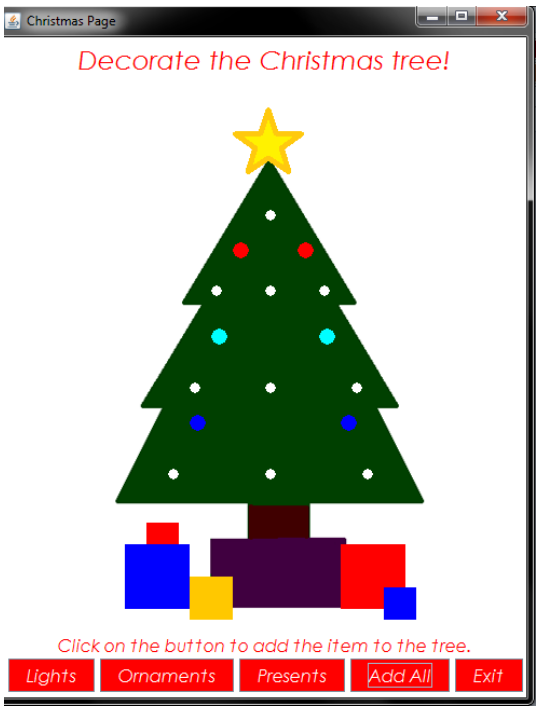


Рис. 65. Внешний вид приложения, реализующего паттерн Декоратор

Указания по выполнению

1. Придумайте и разработайте графический интерфейс пользователя.

2. Реализуйте модель Декоратор в соответствии с диаграммой классов на рис. 10.1.

Абстрактный класс-декоратор

```
public abstract class TreeDecorator implements ChristmasTree {
private ChristmasTree tree;
public TreeDecorator(ChristmasTree tree){
this.tree = tree; }
public void Operation(Pane pane);// выполняет рисование
соответствующего объекта Елки на указанной панели
}
```

Остальные классы наследуются от ***TreeDecorator*** и расширяют функционал путем реализации рисования собственного "декора" в ***Operation():***

```
public void Operation(Pane pane) {
super.draw(pane);
drawWithGirland(pane); }
```

3. В классе Controller графического интерфейса пользователя, при инициализации, создайте объект класса ChristmasTree и инициализируйте ссылкой на него объект класса TreeDecorator.

```
@Override
public void initialize(URL location, ResourceBundle resources) {
    tree=new  ChristmasTreeImpl();
    dtree=null;
}
```

Не забудьте указать для класса Controller implements Initializable!

4. Реализуйте обработчик выбора элементов декора:

```
public void addGarland(ActionEvent actionEvent) {
    if(dtree==null)
        dtree=new Garland(tree);
    else dtree=new Garland(dtree);
    dtree.Operation(paintpane);
}
```

5. Реализуйте дополнительный функционал:

1. Рисование мишуры «случайным» цветом.
2. Анимация гирлянды.
3. Заполнение елки шариками различной формы и цвета и «случайного» положения, не выходящего за пределы фигуры-елки.
4. Вывод строки состояния, отражающей виды и стоимость декора.

Пошаговая реализация приложения «Украшаем елку» на шаблоне Декоратор

Шаг 1. Создается интерфейс ChristmasTree и его реализация:

```
public interface ChristmasTree {
    void draw(Pane pane);
}
```

Шаг 2. Строится абстрактный класс TreeDecorator для этого дерева. Этот декоратор будет реализовывать интерфейс ChristmasTree и содержать тот же объект private ChristmasTree tree:

```
public abstract class TreeDecorator implements ChristmasTree {
    private ChristmasTree tree;

    public TreeDecorator(ChristmasTree tree){

        this.tree = tree;
    }

    public void draw(Pane pane) {
        tree.draw(pane);
    }
}
```

Шаг 3. Создаются элементы украшения Presents (подарки), Girland (гирлянда), Star (звезда).

```
public class Star extends TreeDecorator {
    public Star(ChristmasTree tree) {
        super(tree);
    }

    public void draw(Pane paneStar) {
        super.draw(paneStar);
        drawStar(paneStar);
    }

    ;

    private void drawStar(Pane paneStar) {

        Path star = new Path();
        star.getElements().addAll(new MoveTo(239, 49),
        new LineTo(217, 102),
        new LineTo(239, 91),
        new LineTo(262, 102),
        new ClosePath(),
        new MoveTo(207, 68),
        new LineTo(270, 68),
        new LineTo(239, 91),
        new ClosePath());

        star.setFill(Color.YELLOW);
        star.setFillRule(FillRule.EVEN_ODD);

        paneStar.getChildren().addAll(star);

    }
}
```

Girland:

```
public class Girland extends TreeDecorator {

    public Circle[] circle = new Circle[9];

    public Girland(ChristmasTree tree) {
        super(tree);
    }

    public void draw(Pane paneLight) {
        super.draw(paneLight);
        drawWithGirland(paneLight);
    }

    private void drawWithGirland(Pane paneLight) {
        FadeTransition[] ft = new FadeTransition[9];

        circle[0] = new Circle(210, 200, 10, Color.RED);
        circle[1] = new Circle(270, 200, 10, Color.BLUE);
        circle[2] = new Circle(240, 150, 10, Color.YELLOW);
        circle[3] = new Circle(180, 260, 10, Color.YELLOW);
        circle[4] = new Circle(240, 260, 10, Color.BLUE);
        circle[5] = new Circle(300, 260, 10, Color.RED);
        circle[6] = new Circle(145, 340, 10, Color.BLUE);
        circle[7] = new Circle(240, 340, 10, Color.RED);
        circle[8] = new Circle(335, 340, 10, Color.YELLOW);

        for (int i = 0; i < ft.length; i++) {
            ft[i] = new FadeTransition(Duration.millis(1000), circle[i]);
            ft[i].setFromValue(1.0);
            ft[i].setToValue(0.0);
            ft[i].setCycleCount(Timeline.INDEFINITE);
            ft[i].setAutoReverse(true);
            ft[i].play();
        }

        paneLight.getChildren().addAll(circle[0], circle[1], circle[2], circle[3], circle[4], circle[5], circle[6], circle[7], circle[8]);
    }

}
```

Для анимации гирлянды используется класс FadeTranzition. Он может использовать только готовые объекты, поэтому гирлянда рисуется не на canvas, а на pane, т.к. на pane можно добавить готовую фигуру circle. Создается массив circle с разным местоположением и цветами шаров. Далее массив FadeTransition. С помощью него и цикла for происходит анимация каждого circle.

Шаг 4. Реализация обработки событий интерфейса пользователя в контроллере:

```
public class Controller {
    @FXML
    public Pane paneLight;
    public Pane panePresent;
    public Pane paneTree;
    public Pane paneStar;

    @FXML
    public void initialize() {
        paneTree.toFront();
        ChristmasTree tree = new ChristmasTreeImpl();
        tree.draw(paneTree);
    }


    public void addLights(ActionEvent actionEvent) {
        paneLight.toFront();
        ChristmasTree tree = new Girland(new ChristmasTreeImpl());
        tree.draw(paneLight);

    }
```

Функция addLights добавляет на панель гирлянду. Добавление других украшений выполняется аналогично.

Диаграмма классов приложения на рис. 66.

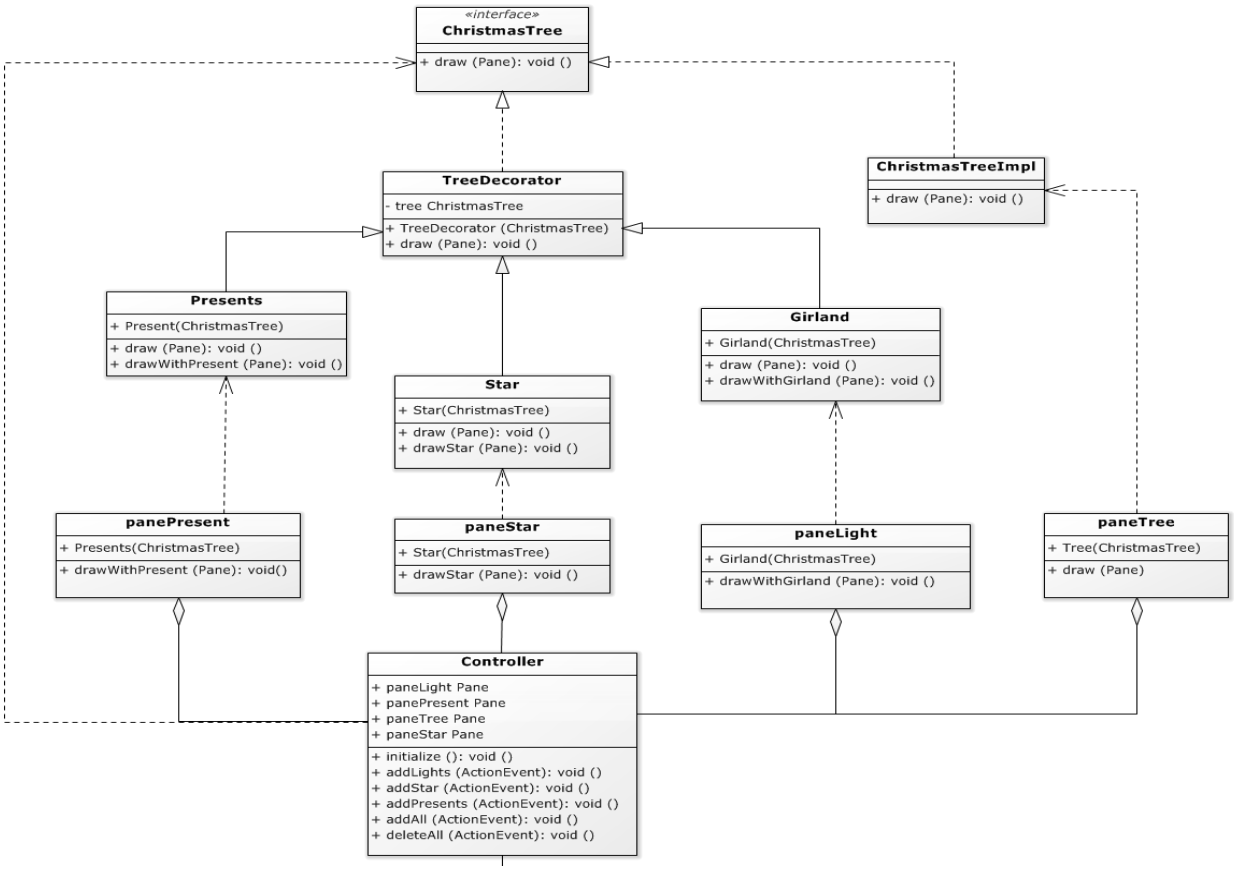


Рис. 66. Диаграмма классов приложения «Украшаем елку»

ШАБЛОН 12. СТАТЕГИЯ

Цель применения: динамический выбор алгоритма в ходе выполнения программы.

Проблема. Необходимо предусмотреть возможность изменения алгоритма сортировки/шифрования/кодирования в зависимости от текущего выбора пользователя.

Описание

Поведенческий паттерн проектирования Стратегии используется для изменения поведения класса путем «переключения» внутреннего алгоритма во время выполнения без изменения самого класса, т.е. позволяет менять алгоритм обработки по запросу пользователя.

Паттерн Strategy переносит в отдельную иерархию классов все детали, связанные с реализацией алгоритмов. Диаграмма классов представлена на рис. 67.

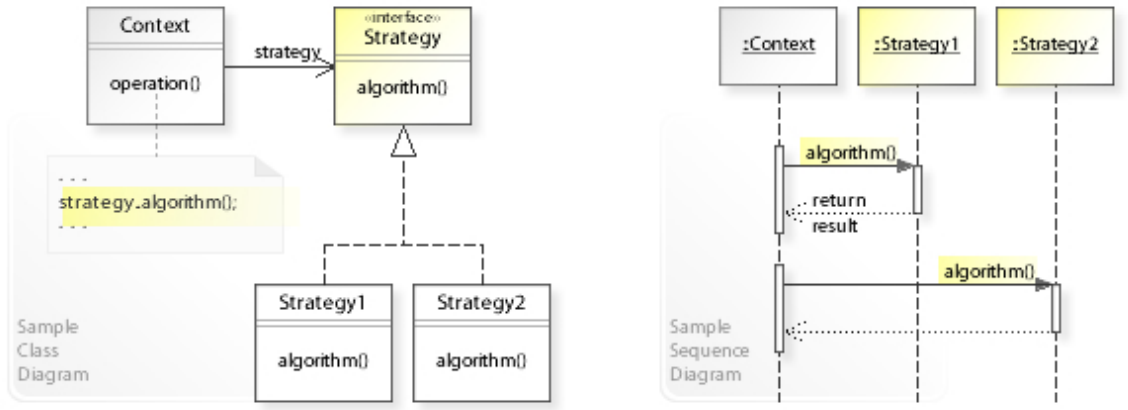


Рис. 67. Диаграмма классов паттерна Стратегия

Для целевой задачи объявляется интерфейс, общий для всех алгоритмов и используемый классом Strategy.

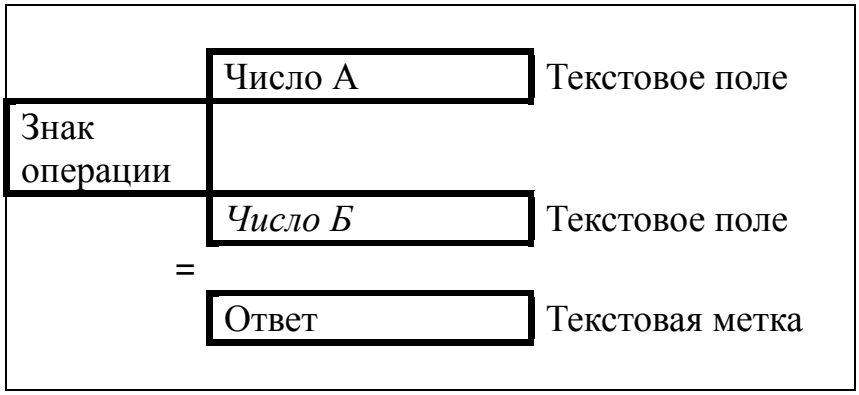
Подклассы ConcreteStrategyA, ConcreteStrategyB реализуют в соответствии с тем или иным алгоритмом.

Класс Context содержит указатель на объект абстрактного типа Strategy и предназначен для переадресации пользовательских запросов конкретному алгоритму. Для замены одного алгоритма другим достаточно перенастроить этот указатель на объект нужного типа.

Задания и указания к их выполнению

Постановка задачи

Реализовать арифметический калькулятор со следующим интерфейсом.



Ограничения: допустимы двухоперандные арифметические операции сложение, вычитание, умножение, деление и логические операции И, ИЛИ, И-НЕ, ИЛИ-НЕ, XOR. Знак операции желательно реализовать в виде выпадающего списка и каждому элементу списка предусмотреть отдельную стратегую обработки.

В качестве операндов допустимы целые и вещественные числа.

Ответ должен быть представлен в развернутом виде – «А+Б=С».

Указания к выполнению

1. Создайте графический интерфейс пользователя и определите два объекта для хранения введенных значений:

```
TextField strnum1, strnum2;
```

2. В соотвествии с диаграммой классов (рис. 67) создайте интерфейс и его реализации для арифметических и логических операций.

```
public interface Strategy {
     public int doOperation(int num1, int num2);
     public String toString();
}
public class OperationAdd implements Strategy {
     @Override
     public int doOperation(int num1, int num2) {
     return num1 + num2;
     }
     public String toString(){
          return "OR";
     }
}
```

3. Создайте контекст использования операций в классе, выполняющим интерпретацию строк из TextField, и вызов соотвествующей стратегии.

```
public class Context {
     Strategy[] concretStrategyy=new Strategy[OPERATION];
```

```
     public Context(){
     concretStrategyy[0]=new OperationAnd();
     concretStrategyy[1]=new OperationOR();
     concretStrategyy[2]=new OperationXOR();
     }

     public int executeStrategy(int op, int num1, int num2){
     Strategy strategy = concretStartegyy[op];// not safety!!,
//check the op index!!!
     return strategy.doOperation(num1, num2);
     }
}
```

4. Реализуйте обработку событий графического интерфейса.

Пошаговая реализация программы сортировки массива по выбранному пользователем алгоритму

Шаг 1. Определение интерфейса, включающий метод сортировки. Реализация данного интерфейса:

```
     public interface SortingStrategy {
         public void sort(int[] numbers);
boolean compare(int a, int b) {
if(a < b) return true;
else return false; }
     }
```

Шаг 2. Реализация классов конкретных стратегий для различных методов сортировки (BubbleSort – методом «пузырька», SelectionSort – выбором и InsertionSort – вставками):

```
public class BubbleSort implements Strategy {
    @Override
    public void sort(int[] array) {
        for (int i = array.length - 1; i >= 0; i--) {
            for (int j = 0; j < i; j++) {
                if (array[j] > array[j + 1]) {
                    int tmp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = tmp;
                }
            }
        }
    }
}
public class SelectionSort implements Strategy {
    public void sort(int[] array) {
        for (int i = 0; i < array.length - 1; i++) {
            int index = i;
```

```
            for (int j = i + 1; j < array.length; j++)
                if (array[j] < array[index])
                    index = j;
            int smallerNumber = array[index];
            array[index] = array[i];
            array[i] = smallerNumber;
        }
    }
}
```

Шаг 3. Создание класса Context, играющий роль модели или использования Стратегии в конкретной задаче:

```
 public class Context {
    private Strategy strategy; // ссылка на алгоритм сортировки
    private int[] array; // данные (модель), к-е  обрабатываются
(включая сортировку)
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
    public void sortArray(int[] array) {
        strategy.sort(array);
    }
}
```

Шаг 4. Разработка графического интерфейса пользователя:

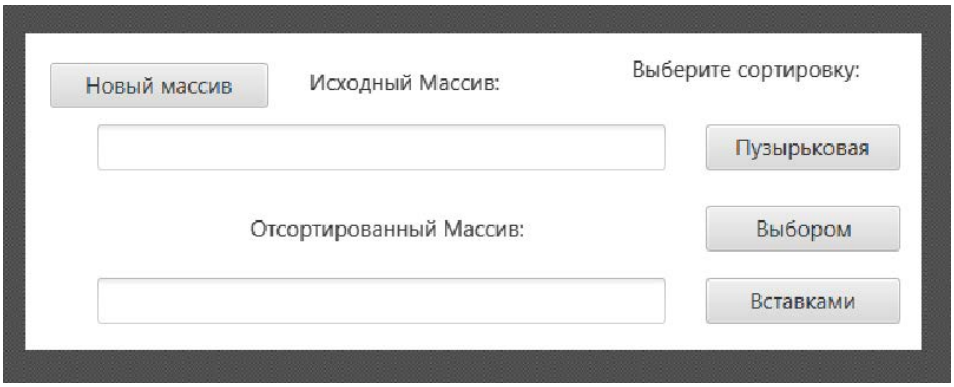


Рис. 68. Графический интерфейс приложения сортировки массива разными методами

Шаг 5. Обработка событий GUI, в частности, нажатие на кнопку:

```
    public void bubbleSort(ActionEvent actionEvent) {
        new Context(new BubbleSort()).sortArray();//сортировка
        printArray(context.getArray());  // отображение результата
}
```

ШАБЛОН 13. ЦЕПОЧКА ОБЯЗАННОСТЕЙ

Цель применения: организовать получение запроса в приложении так, что несколько объектов могут обрабатывать запрос, и обработчик не обязательно должен быть конкретным объектом и даже опеределен во время выполнения.

Проблема. Разрабатывается игровой автомат, необходимо создать случайность в розыграше, т.е. определить правило обработки в ходе самой игры.

Описание

Паттерн Цепочка обязанностей позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким связанным объектам, передавая запрос вдоль этой цепочки, пока его не обработают.

Паттерн Цепочка обязанностей реализует поведение следующим образом (диаграмма последовательностей на рис. 69):

1. Формирует цепочку объектов, способных обработать запрос и имеющих общий интерфейс.
2. Организуется передача запроса по цепочке
3. Каждый объект сам решит, будет ли обрабатывать запрос и передаст его следующему объекту в цепочке.

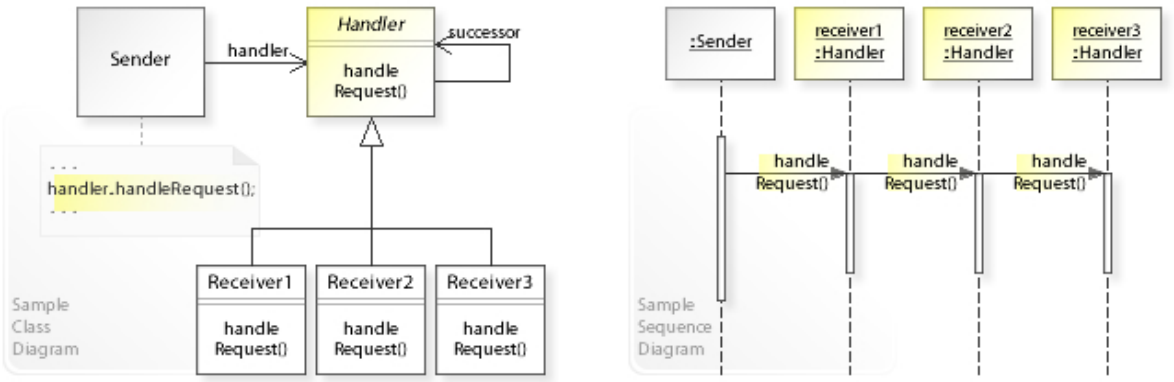


Рис. 69. Диаграмма классов и диаграмма последовательностей паттерна Цепочка обязанностей

Участники шаблона Цепочка обязанностей следующие.

Handler – интерфейс, который в первую очередь получает запрос и отправляет запрос в цепочку обработчиков. Он имеет ссылку только на первый обработчик в цепочке и ничего не знает об остальных обработчиках.

Receiver1, Receiver2, Receiver3 – это возможные обработчики запроса, связанные в некотором последовательном порядке.

Sender – инициатор запроса, который запускает поиск обработчика.

Реализация этого паттрена близка к Декоратору, и отличие состоит в том, что структурирование осуществляется во время выполнения. Цепочку обработки следует применять, если:

– необходимо отделить отправителя и получателя запроса;

– существует несколько объектов, определенных во время выполнения, могут обработать запрос;

– нельзя явно указывать обработчика запроса;

– необходимо отправить запрос к одному из нескольких объектов без указания получателя явно.

Задание и указания по его выполнению

Условие задачи

Требуется создать игровой автомат со следующим геймплеем.

Пользователю предлагается три мешочка (один из которых якобы выигрышный), пользователь выбирает мешочек, и либо в нем нет монеток (проиграл), тогда он может снова кинуть монетку для продолжения игры, либо в нем есть монетки, тогда пользователь может забрать выигрыш и закончить игру или может снова играть на выигранное.

Конечно, никто не планирует эти мешочки заполнять случайным образом, и определение удача/неудача никак не зависит от выбранной картинки на экране. Поэтому обработка одинакова для всех кнопок, а вот стратегия по отношению к пользователю может быть различна – «удержать», «обобрать», «дать надежду» и может зависеть от продолжительности игры, активности пользователя и т.п.

Указания к выполнению

1. Разработайте привлекательный пользовательский интерфейс.

2. Реализуйте шаблон Цепочка обязанностей для обработки события выбора мешочка игроком в соответствии с диаграммой классов на рис. 69.

2.1. Создайте абстрактный класс.

```
public abstract class Handler  {
    private Handler processor;

    public Handler(Handler processor){
        this.processor = processor;
    }

    public boolean process(Integer request){
        if(processor != null)
            return processor.process(request);
        else
        return true;
    }
}
```

2.2. Реализуйте конкретные обработчики для сообщения о выигрыше и проигрыше, через наследование от Handler и реализацию различных способов обработки в методе process, для анализ то, что данный класс может обработать запрос используйте сравнение с параметром-константой = типом запроса, как пример:

```
public class NegativeHandler extends Handler {

    public NegativeHandler(Handler processor) {
        super(processor);  }

    public boolean process(Integer request) {
       if(request!=LOSS) return super.process(request);// не свой
запрос передается дальше по цепочке
       else {//свой, обрабатывается здесь
            Alert alert = new Alert(Alert.AlertType.INFORMATION);
            alert.setTitle("Вы проиграли!");
            alert.setHeaderText("Монетка потеряна, но всегда можно
отыграться!");

            ButtonType replay = new ButtonType("Продолжить
играть", ButtonBar.ButtonData.YES);
            ButtonType vacation = new ButtonType("Отдохнуть",
ButtonBar.ButtonData.NO);
            alert.getButtonTypes().clear();
            alert.getButtonTypes().addAll(replay, vacation);
            Optional<ButtonType> option = alert.showAndWait();

            if (option.get().getButtonData() ==
ButtonBar.ButtonData.YES)
                return true;
            else
                return false;        }
    }
```

3. Для удобства использования создайте отдельный класс для «розыгрыша», строящий цепочку из имеющихся конкретных обработчиков, содержащий служебные константы и при необходимости генерацию решения о выигрыше/проигрыше, например, так:

```
public class ActionChain {
    Handler chain;
    public static int SUCCESS = 1;
    public static int LOSS = 3;
    Random generate;
    final int NUMHANDLER=4;
    final int NUMMAX = 7;

    public ActionChain(){
                generate =new Random();
                buildChain();
    }
```

```
    private void buildChain(){
        chain = new NegativeHandler(new PositiveHandler(null));
    }
    public boolean process() {
        int type=generate.nextInt(NUMHANDLER);//розыгрыш
        return process(type);
    }
    public boolean process(Integer a) {
        return chain.process(1+a%NUMHANDLER);//обрезка по числу
имеющихся обработчиков
    }
```

4. В классе Controller графического интерфейса пользователя предусмотрите включение объекта «розыгрыша» ActionChain action=null; и при инициализации, создайте обработчик нажатий мыши. Так в примере, к элементу ImageView добавлен новый метод обработки щелчка по изображению:

```
public void initialize(URL location, ResourceBundle resources) {
    view.setImage(new Image("монетка.jpg"));
    view.addEventHandler(MouseEvent.MOUSE_CLICKED, event -> {

        if(action==null) return;//если цепочка обработки
отсутствует
        if (action.process()) init();//продолжить играть и
проверить наличия монетки у игрока
        else action=null;//завершить игру, сделав обработку
недоступной
    });
```

5. Реализуйте оставшуюся логику игры, руководствуясь алгоритмом на рис. 70.

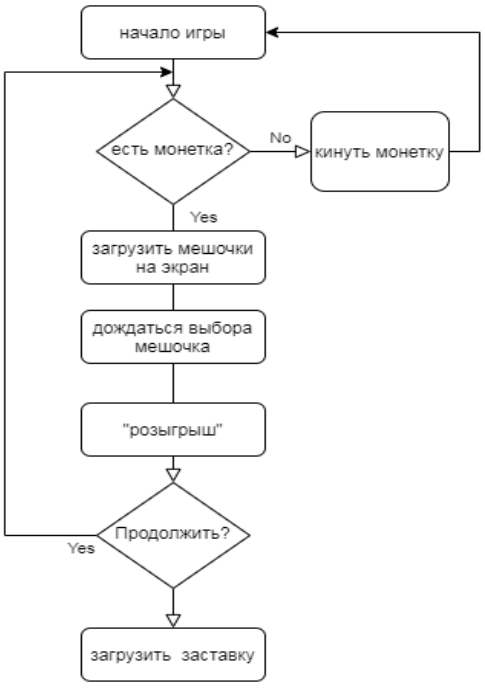


Рис. 70. Алгоритм работы игрового автомата

5.1. Создайте класс игрока, для формирования счета:

```
public class Player {
    private String name;
    private Integer count;
    private Integer number;
    public Player(String name, Integer number) {
        this.name = name;
        this.number = number;
        count=0;
    }
public boolean pay(Integer number) {
        if(number <= this.number) {
            this.number-=number;
            this.count++;
            return true;        }
      else return false;
    }
public Integer getCount() {
        return count;
    }
public Integer getNumber() {
        return number;
    }
public void addNumber(Integer num) {
        this.number+= num;
     }
```

5.2. Реализуйте обработчики кнопок Игрового автомата, например, заправка монетки:

```
public void onPay(ActionEvent actionEvent) {
    player1.addNumber(1);
}
запуск игры:
public void onStart(ActionEvent actionEvent) {
    if(!init()) return;//проверка ликвидности
    view.setImage(new Image("мешочки.png"));//загрузка автомата
    action=new ActionChain();//запуск механизма розыгрыша
    }
 проверку игрока на наличие средств:
public boolean init(){
    if(! player1.pay(1)) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setHeaderText("Средств на счете недостаточно, еще
монетку плисс!");
        alert.show();
        action=null;
        view.setImage(new Image("монетка.jpg"));
        return false;
    }
    else return true;
     }
```

Проверьте структуру программы (диаграмма классов на рис. 71). Отладьте программу.

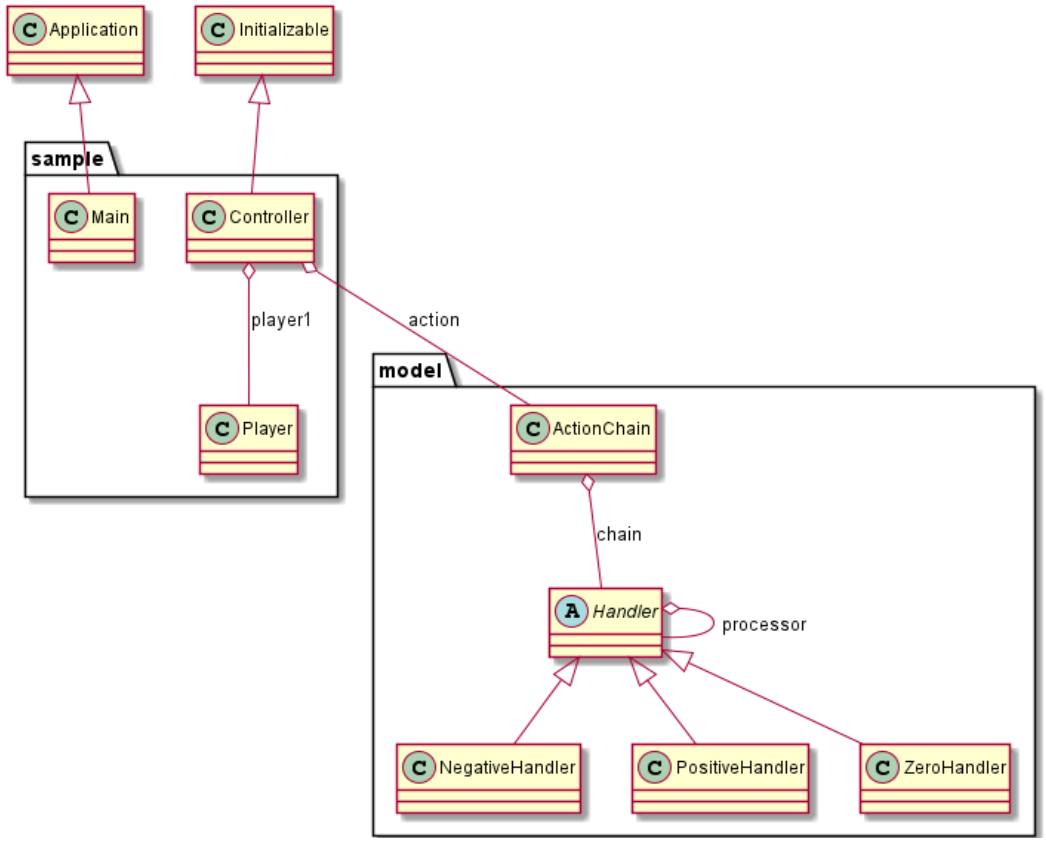


Рис. 71. Диаграмма классов реализации Игрового автомата

6. Чтобы убедиться в гибкости шаблона Цепочка обязанностей сделайте еще один вариант обработки события – шанс, например:

```
Alert alert = new Alert(Alert.AlertType.INFORMATION);
alert.setTitle("Вы проиграли!");
alert.setHeaderText("Но судьба дает Вам шанс сыграть еще раз
бесплатно!");
```

7. По мере увеличения альтернатив «розыгрыша» появится проблема – получения из цепочки размера выигрыша. Рассмотрите возможные решения проблемы – создание метода начисления выигрыша и вызов его «параллельно» process(), или запоминание обработчика или передача дополнительных параметров в вызове process() или предусмотрение возможности возвращать значение или собъект.

Обратите внимание на побочный эффект, если нет обработчика типа=0 - проходит "пустой" розыгрыш (деньги списаны, сообщений о выигрыше/ проигрыше – нет).

Пример разработки программы «Розыгрыш»

Требования: создать игровой автомат, в котором случайно генерируются элементы BPMN, а Пользователь может угадать, какой элемент высветится на экране. Чтобы сыграть нужно, кинуть монетку в автомат, в случае, если Пользователь выиграет, он получит 2 монетки.

Пошаговое описание программы

Пользователь кидает монетку в автомат из своего кошелька, нажав на кнопку Кинь монетку.

Пользователь запускает автомат, нажав на кнопку «Начать играть».

Пользователь получает задание, где нужно угадать элемент BPMN, который скрывается в сундуке.

Пользователь выбирает элемент и нажимает на кнопку «Испытать удачу».

Если Пользователь угадал, он получает 2 монетки в свой кошелёк.

Если Пользователь не угадал, он ничего не получает.

Если у Пользователя закончились монеты в кошельке, автомат даёт ему бесплатную попытку.

Основные модули приложения:

– насыщенный графический пользовательский интрфейс;

– обработчики событий игры, построенные по шаблону декоратор

– контроллер GUI, агрегирующий использование обработчиков и информации о пользователе.

Структура проекта представлена на рис. 72. Пакет Hand_R включает классы Цепочки обязанностей, пакет sample – описание GUI и его контроллер, папка Resourse находится в корневом каталоге и содержит набор изображений для насыщения GUI.

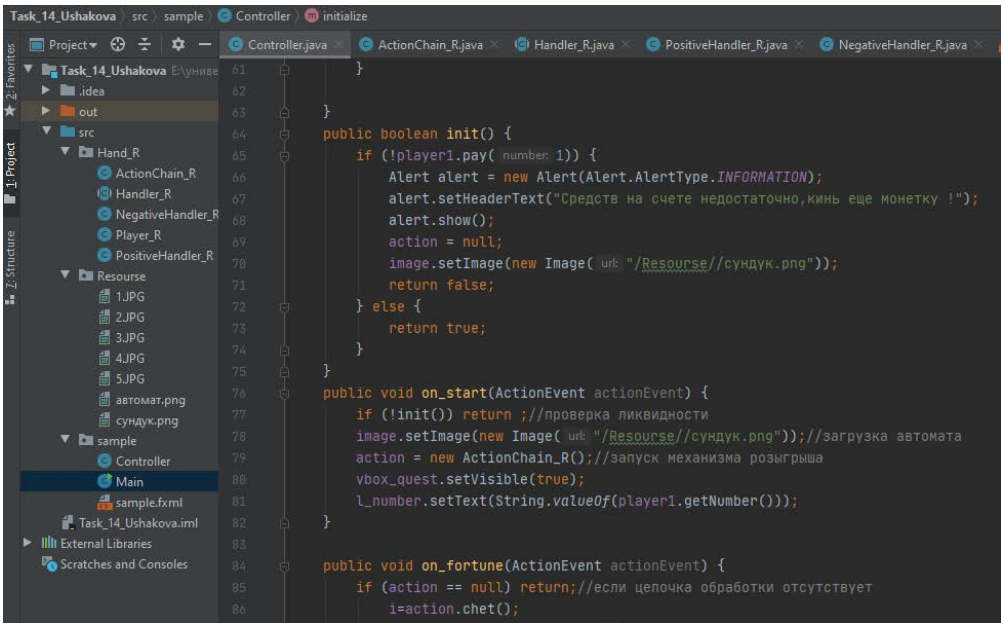


Рис. 72. Структура проекта

На рис. 73 представлена раскадровка GUI с иерархией компонентов приложения.

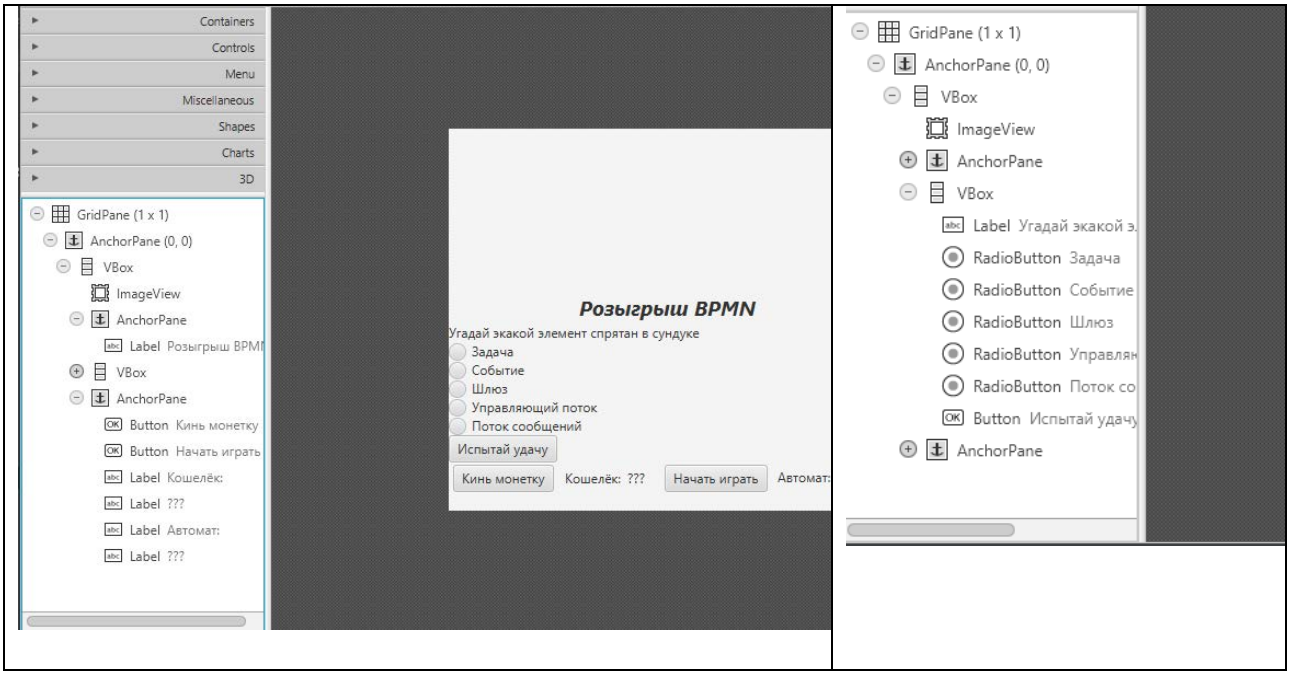


Рис. 73. Проектирование GUI

Диаграмма классов программы представлена на рис. 74.

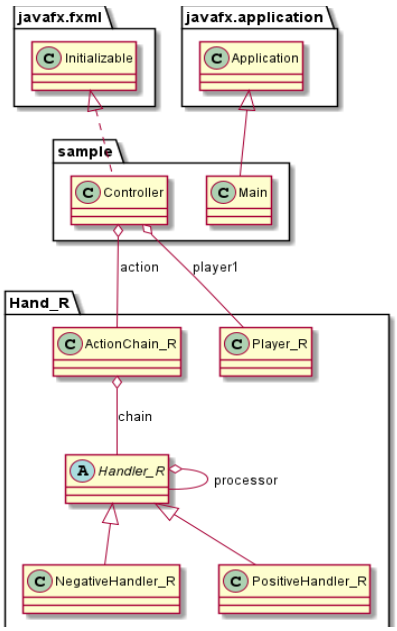


Рис. 74. Диаграмма классов приложения

Листинг программы с комментариями:

```
public abstract class Handler_R {
    private Handler_R processor;
    public Handler_R(Handler_R
processor){
        this.processor = processor;
    }
    public boolean process(Integer
request){
        if(processor != null)
            return
processor.process(request);
        else
            return true;
    }
public class PositiveHandler_R
extends Handler_R{

    public
PositiveHandler_R(Handler_R
processor) {
        super(processor);
    }
```

```
    public boolean process(Integer request) {
        if(request!=ActionChain_R.SUCCESS)
            return super.process(request);// не свой запрос
передается дальше по цепочке

        else {//свой, обрабатывается здесь

            Alert alert = new Alert(Alert.AlertType.INFORMATION);
            alert.setTitle("Вы выйграли!");
            alert.setHeaderText("Монетка у тебя, но всегда можно
получить больше!");

            ButtonType replay = new ButtonType("Продолжить
играть", ButtonBar.ButtonData.YES);
            ButtonType vacation = new ButtonType("Отдохнуть",
ButtonBar.ButtonData.NO);
            alert.getButtonTypes().clear();
            alert.getButtonTypes().addAll(replay, vacation);
            Optional<ButtonType> option = alert.showAndWait();

            if (option.get().getButtonData() ==
ButtonBar.ButtonData.YES)
                return true;
            else
                return false;        }
    }
}

public class NegativeHandler_R extends Handler_R{
    public NegativeHandler_R(Handler_R processor) {
        super(processor);
    }
    public boolean process(Integer request) {
        if(request!=ActionChain_R.LOSS)
            return super.process(request);// не свой запрос
передается дальше по цепочке
        else {//свой, обрабатывается здесь
            Alert alert = new Alert(Alert.AlertType.INFORMATION);
            alert.setTitle("Вы проиграли!");
            alert.setHeaderText("Монетка потеряна, но всегда можно
отыграться!");
            ButtonType replay = new ButtonType("Продолжить
играть", ButtonBar.ButtonData.YES);
            ButtonType vacation = new ButtonType("Отдохнуть",
ButtonBar.ButtonData.NO);
            alert.getButtonTypes().clear();
            alert.getButtonTypes().addAll(replay, vacation);
            Optional<ButtonType> option = alert.showAndWait();
            if (option.get().getButtonData() ==
ButtonBar.ButtonData.YES)
                return true;
            else
                return false;        }
```

```
    }
public class ActionChain_R {
    Handler_R chain;
    public static int SUCCESS = 1;
    public static int LOSS = 3;
    Random generate;
    int type=0;

    public ActionChain_R() {
        generate=new Random();;
        buildChainN();
        buildChainP();
    }
    public void buildChainN() {
        chain = new NegativeHandler_R
                (new PositiveHandler_R(null));
    }
    public void buildChainP() {
        chain = new PositiveHandler_R
                (new NegativeHandler_R(null));
    }    }
    public boolean processN() { // вызов NegativeHandler
        int type = 3;
        return process(type);
    }
    public int chet() {
        type = generate.nextInt(5)+1;//генерация
        return type;
    }
    public boolean process(Integer a) { // вызов цепочки
        return chain.process(a);
   }}
public class Player_R {
    private String name;
    private Integer count;
    private Integer number;

    public Player_R(String name, Integer number, Integer count) {
        this.name = name;
        this.number = number;
        this.count = count;
    }
    public boolean pay(Integer number) { //оплата
        if(number <= this.number) {
            this.number-=number;
            this.count--;
            return true;  }
        else return false;
    }
    public Integer getCount() {
        return count;
    }
    public Integer getNumber() {
        return number;
```

```
    }
    public void addNumber(Integer number) { //кинуть монетку в
автомат
        this.number+= number;
    }
    public void addCount(Integer count) { //кинуть монетку в
кошелёк
        this.count+= count;
    }
    public void delCount(Integer count) { //убрать монетку из
кошелька
        this.count-= count;    }}
public class Controller implements Initializable {
    public Player_R player1=new Player_R("Вова",0,5);
    public ActionChain_R action;
    public VBox vbox_imp;
    public ImageView image;
    public VBox vbox_quest;
    public AnchorPane pane_button;
    public RadioButton r_action;
    public RadioButton r_event;
    public RadioButton r_gateways;
    public RadioButton r_sf;
    public RadioButton r_mf;
    public Label l_count;
    public Label l_number;
    int i=0;
@Override
    public void initialize(URL location, ResourceBundle resources)
{
        vbox_quest.setVisible(false);
        image.setImage(new Image("/Resourse//автомат.png"));
        l_count.setText(String.valueOf(player1.getCount()));
    }
    public void on_pay(ActionEvent actionEvent) { //оплата игры
        if(player1.getCount()>0){ //если монетка есть в кошельке
        player1.addNumber(1); //кинуть монетку в автомат
        player1.delCount(1); //убрать монетку из кошелька
        l_count.setText(String.valueOf(player1.getCount()));
        l_number.setText(String.valueOf(player1.getNumber()));}

        else { // бесплатнаяя игра
            Alert alert = new Alert(Alert.AlertType.INFORMATION);
             alert.setTitle("Вы проиграли!");
             alert.setHeaderText("Но судьба дает Вам шанс сыграть
еще раз бесплатно!");
             alert.show();
             player1.addCount(1); //даём игроку ещё монетку
            l_count.setText(String.valueOf(player1.getCount()));
        }

    }
    public boolean init() { //проверка автомата на наличие монеток
        if (!player1.pay(1)) { //если нет монеток
```

```
            Alert alert = new Alert(Alert.AlertType.INFORMATION);
            alert.setHeaderText("Автомат пуст, кинь монетку!");
            alert.show();
            action = null;
            image.setImage(new Image("/Resourse//сундук.png"));
            return false;
        } else {
            return true; //если есть
        }
    }
    public void on_start(ActionEvent actionEvent) { // обработчик
событий кнопки Начать игру
        if (!init()) return ;//проверка ликвидности
        image.setImage(new
Image("/Resourse//сундук.png"));//загрузка автомата
        action = new ActionChain_R();//запуск механизма розыгрыша
        vbox_quest.setVisible(true); //сделать видимым викторину
        l_number.setText(String.valueOf(player1.getNumber()));
    }
    public void on_fortune(ActionEvent actionEvent) { //обработчик
событий кнопки Испытать удачу
        if (action == null) return;//если цепочка обработки
отсутствует
            i=action.chet(); //генерируем случайное изображение
элемента BPMN
            image.setImage(new Image("/Resourse//"+i+".jpg"));

            if(r_action.isSelected()==true && i==1) { //если игрок
угадал + 2 монеты
                action.process(1);
                vbox_quest.setVisible(false);
                player1.addCount(2);

l_count.setText(String.valueOf(player1.getCount()));

            }
            else if(r_event.isSelected()==true &&i==2) {
                action.process(1);
                vbox_quest.setVisible(false);
                player1.addCount(2);

l_count.setText(String.valueOf(player1.getCount()));
            }
            else if(r_gateways.isSelected()==true &&i==3) {
                action.process(1);
                vbox_quest.setVisible(false);
                player1.addCount(2);

l_count.setText(String.valueOf(player1.getCount()));
            }
            else if(r_sf.isSelected()==true &&i==4) {
                action.process(1);
                vbox_quest.setVisible(false);
                player1.addCount(2);
```

```
l_count.setText(String.valueOf(player1.getCount()));
            }
            else if(r_mf.isSelected()==true &&i==5) {
                action.process(1);
                vbox_quest.setVisible(false);
                player1.addCount(2);

l_count.setText(String.valueOf(player1.getCount()));
            }
            else{ action.processN(); //если игрок не угадал + 0
монет
                vbox_quest.setVisible(false);

l_count.setText(String.valueOf(player1.getCount()));
                action=null;//завершить игру
            }
    }
}
```

Пример работы приложения на рис. 75.

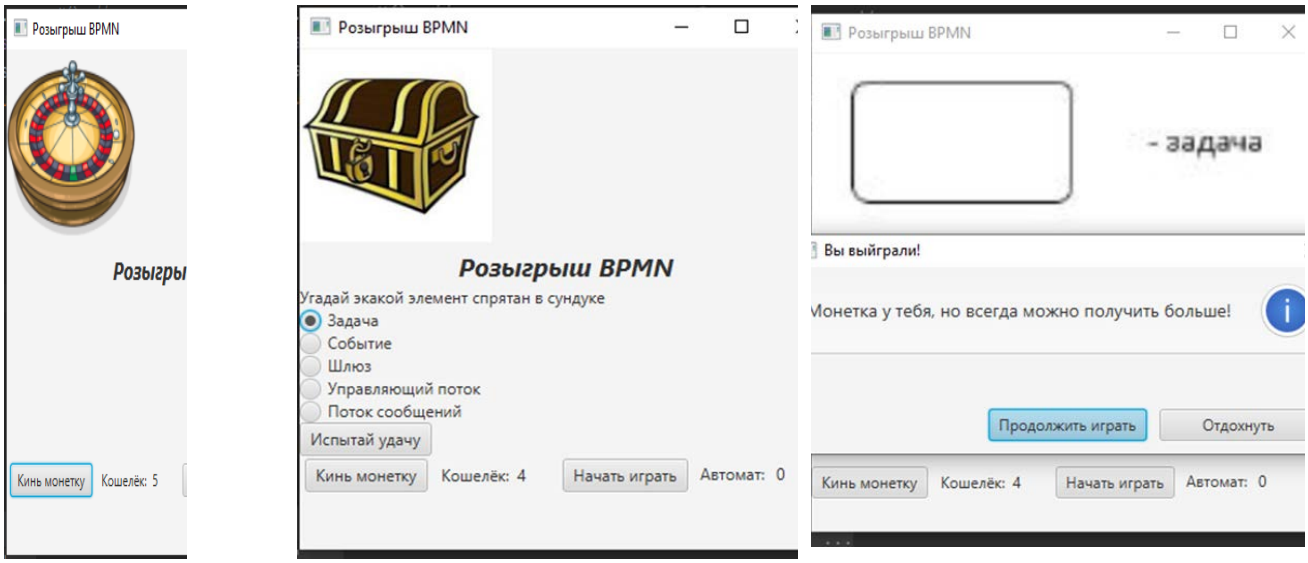


Рис. 75. Рабочие окна программы

ШАБЛОН 14. ПОСРЕДНИК

Цель применения: организовать взаимодействие объктов вида «много ко многим».

Проблема. Разрабатывается система тестирования. Необходимо реализовать различные режимы работы с тремя категориями пользователей. L1 «Студент» отвечает на вопросы теста. L2 «Преподаватель» создает, редактирует и размещает тестовые задания. L3 «Посетитель» может только просматривать содержимое тестовых заданий. Все они работают с общей

базой тестов, но по-разному. Поэтому необходимо организовать их совместное взаимодействие.

Описание

Паттерн Посредник (Mediator) определяет поведение «все ко всем», вводя посредника для развязывания множества взаимодействующих объектов. Что заменяет взаимодействие «все со всеми» взаимодействием «один со всеми».

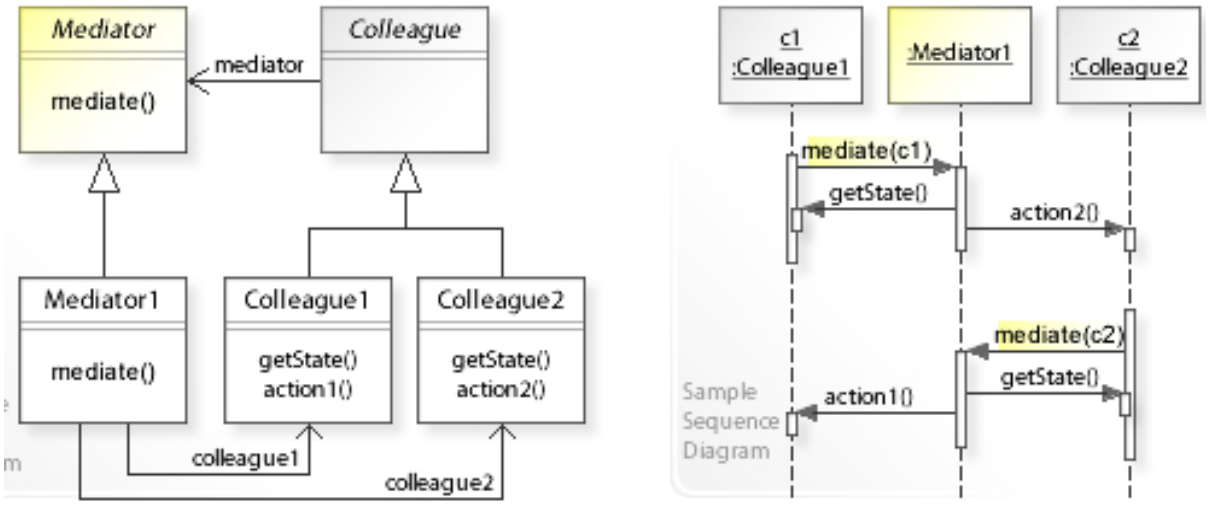


Рис. 76. Диаграмма классов и диаграмма последовательностей паттерна Mediator

Во взаимодействии на диаграмме классов (рис. 76) участвуют коллеги или взаимодействующие объекты, которые не связаны друг с другом. Каждый из них общается с посредником, который, в свою очередь, знает об остальных и управляет ими.

Colleague определяет интерфейс для обмена данными с другими Colleague через Mediator.

Для класса ConcreteColleague каждый класс Colleague знает свой объект-посредник, и каждый объект- коллега связывается со своим посредником всякий раз, когда необходимо общаться с другим ConcreteColleague.

Mediator определяет интерфейс для обмена данными с Colleague объектов. ConcreteMediator реализует кооперативное поведение путем координации ConcreteColleague объектов.

Задание и указания по их выполнению

Постановка задачи

Создать авторизационный модуль в системе тестирования для доступа к

базе заданий, работа с которой зависит от уровня доступа - только чтение (L3), редактирование (L2), генерация в формы «вопрос-ответ» (L1).

База данных содержит множество заданий, каждое из которых содержит строку вопроса, один список правильных и другой список неправильных ответов.

Под тестом понимается заданное число тестовых заданий. Тестовое задание содержит вопрос с одним правильным и тремя неправильными ответами.

Режим редактирования (пользователь Editor) предполагает вывод полей редактирования заданий.

Режим тестирования (пользователь Worker) предполагает вывод теста с возможность выбора одного ответа в каждом тестовом задании и автоматической его проверки.

Режим чтения (пользователь Viewer) предполагает вывод только для чтения – списка из 5 случайных тестовых заданий с отмеченными правильными ответами.

Посредник должен в зависимости от введенного логина запускать соответствующий режим.

Указания по выполнению

1. Создайте модель тестового задания.

Примерный класс задания, содержащего вопрос и все возможные варианты ответов:

```
public class Qweston {
    StringProperty question;
    ArrayList<StringProperty> answergood;
    ArrayList<StringProperty> badanswer;
    Integer type;//категория вопроса

    public Qweston(String qw){
        question=new SimpleStringProperty(qw);
        answergood= new ArrayList<>();
        badanswer= new ArrayList<>();
    }
    public int addTrue(String s){
        answergood.add(new SimpleStringProperty(s));
        return answergood.size();
    }
    public int addFalse(String s){
        badanswer.add(new SimpleStringProperty(s));
        return badanswer.size();
    }
    public ArrayList<StringProperty> getAnswergood() {
        return answergood;
    }
    public void setQuestion(String question) {
        this.question.set(question);
    }
```

Этот класс и будет составлять содержание сообщение, которое будут поддерживать Коллеги и сам Посредник.

2. Реализуйте шаблон Посредник с учетом использования базы задания BaseTest в соответствии с диаграммой классов на рис. 77.

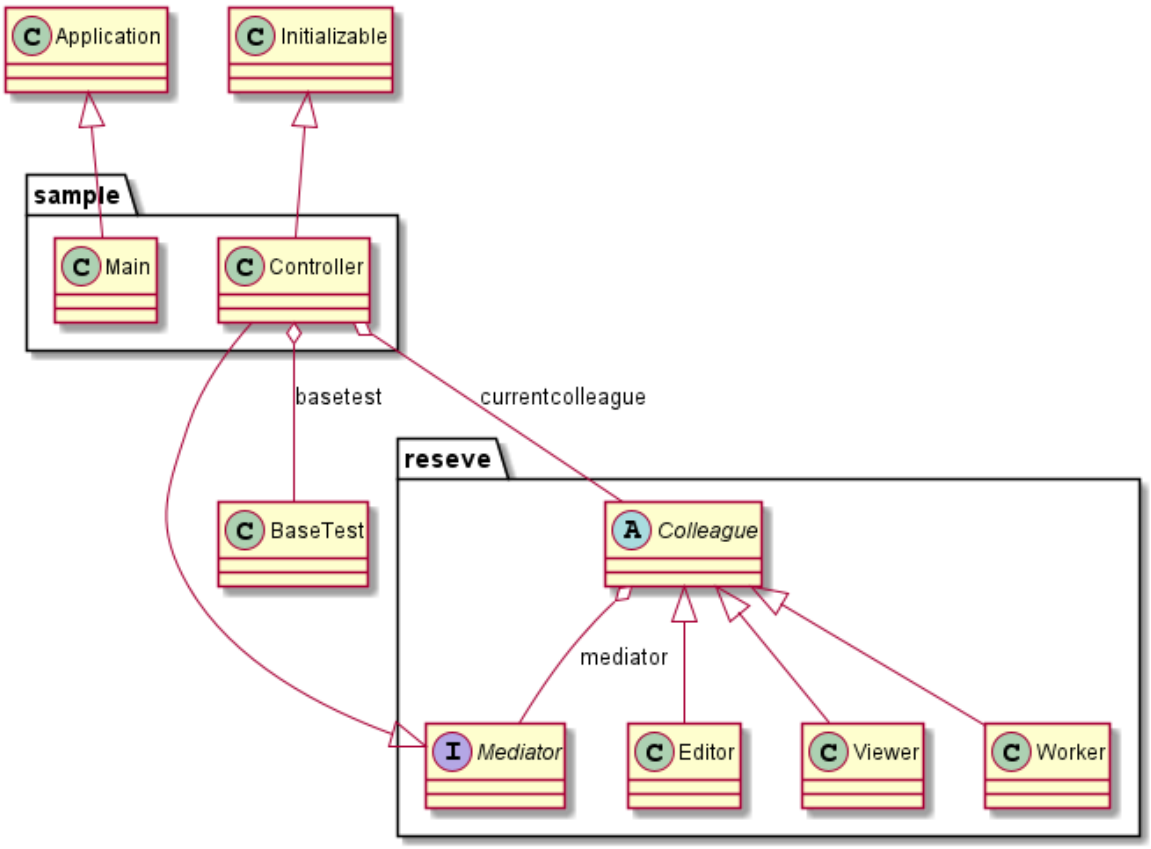


Рис. 77. Диаграмма классов приложения

2.1. Создайте абстрактный класс Colleague для организации режима работы:

```
public abstract class Colleague {
    protected Mediator mediator;
    Queston receivedMessage;

    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }
    public abstract void notifyColleague(Queston message);
    public void receive(Queston message){
        this.receivedMessage=message;
    }

    public  Queston getReceivedMessage() {
        return this.receivedMessage;
    }
```

2.2. Реализуйте конкретные режимы (Обязанность каждого режима сформировать из сообщения внешнее представление). Так класс Editor должен связать содержимое сообщения с текстовыми полями графического интерфейса пользователя:

```
public class Editor extends Colleague {
    public Editor(Mediator mediator) {
        super(mediator);
    }
    @Override
    public void notifyColleague(Queston message) {
        VBox qwpane=new VBox();
        TextField qwfield=new TextField();

qwfield.textProperty().bindBidirectional(message.questionProperty(
));
        qwpane.getChildren().add(qwfield);
        Separator separator=new Separator();
        separator.setMaxWidth(20);
        qwpane.getChildren().add(separator);
        for (int i = 0; i <message.getAnswergood().size() ; i++) {
        TextField qwfieldi=new TextField();

qwfieldi.textProperty().bindBidirectional(message.get(iqw).getAnsw
ergood().get(i));
        qwpane.getChildren().add(qwfieldi);
        }
        for (int i = 0; i <message.ggetBadanswer().size() ; i++) {
           TextField qwfieldi=new TextField();

qwfieldi.textProperty().bindBidirectional(message.getBadanswer().g
et(i));
            qwpane.getChildren().add(qwfieldi);
            }
       mediator.setView(qwpane);
    }
```

Для связи отображения заданий и их модели следует использовать механизм связывания binding (подраздел 14.3).

Пример реализации внешнего вида работы приложения в различных режимах на рис. 78.

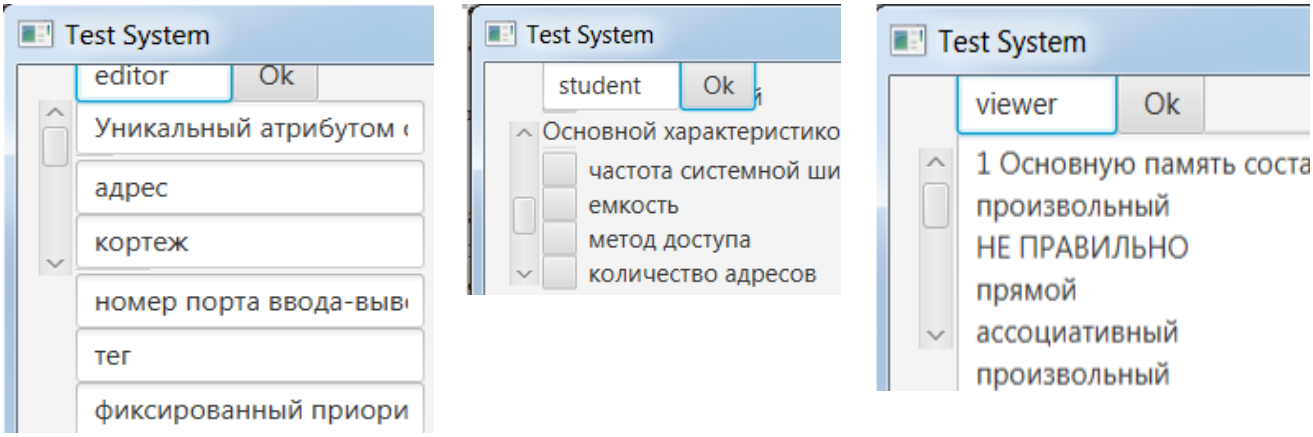


Рис. 78. Вид GUI в различных режимах работы

3. Создайте интерфейс Mediator (его обязанность установка нового режима, поэтому вместо метода рассылки оповещения Message всем режимам Colleague рекомендуется setView() для отображения теста на форме):

```
void notifyColleague(Colleague colleague, Message message);
public interface Mediator {
   void setView(Node control);
}
```

4. Создайте графический интерфейс пользователя, включающий поле для ввода логина, кнопку действия для запуска режима и панель для вывода тестовых заданий public Pane viewpane, поскольку Controller как и режимы отвечают за отображение, то можно реализовать интерфейс Mediator в самом контроллере:

```
public class Controller implements Initializable, Mediator {
    public Pane viewpane;
    public TextField login;
    private HashMap<String, Colleague> id=new HashMap<>();
    private Colleague currentcolleague;
    private Queston basetest=new Queston();

    public void onStart(ActionEvent actionEvent) {
        currentcolleague =id.get(login.getText());
        if(currentcolleague==null) currentcolleague =id.get("1");
        currentcolleague.receive(basetest.getTest());

currentcolleague.notifyColleague(currentcolleague.getReceivedMessa
ge());
    }
    @Override
    public void initialize(URL location, ResourceBundle resources)
{
        id.put("student", new Worker(this));
        id.put("lector", new Editor(this));
        id.put("1", new Viewer(this));
    }
   @Override
    public void setView(Node control) {
        viewpane.getChildren().add( control);
    }
}
```

5. Организуйте загрузку теста.

Предусмотрите инициализацию значений для класса Queston, т.е. создайте само тестовое задание. На приведенной диаграмме (рис. 78) задание подгружается из тестовой базы BaseTest.

FX: связывание (binding)

Механизм связывания объектов в JavaFX (аналогичен привязке свойств в JavaBeans, когда объекты инициируют события об изменениях в свойствах при вызове методов их установки) основан на интерфейсе Property (рис. 79) и его конкретных реализациях. Для свойства примитивного типа в качестве оболочки используется IntegerProperty, LongProperty, DoubleProperty, FloatProperty или BooleanProperty. Для этой цели имеются также классы ListProperty, MapProperty и SetProperty, а для всего остального — обобщенный класс ObjectProperty<T>. Все эти классы являются абстрактными и имеют конкретные подклассы SimpleIntegerProperty, SimpleObject Property<T> и т.д. К объекту конкретных Свойств можно присоединить приемник событий, так как он поддерживает интерфейс Observable.

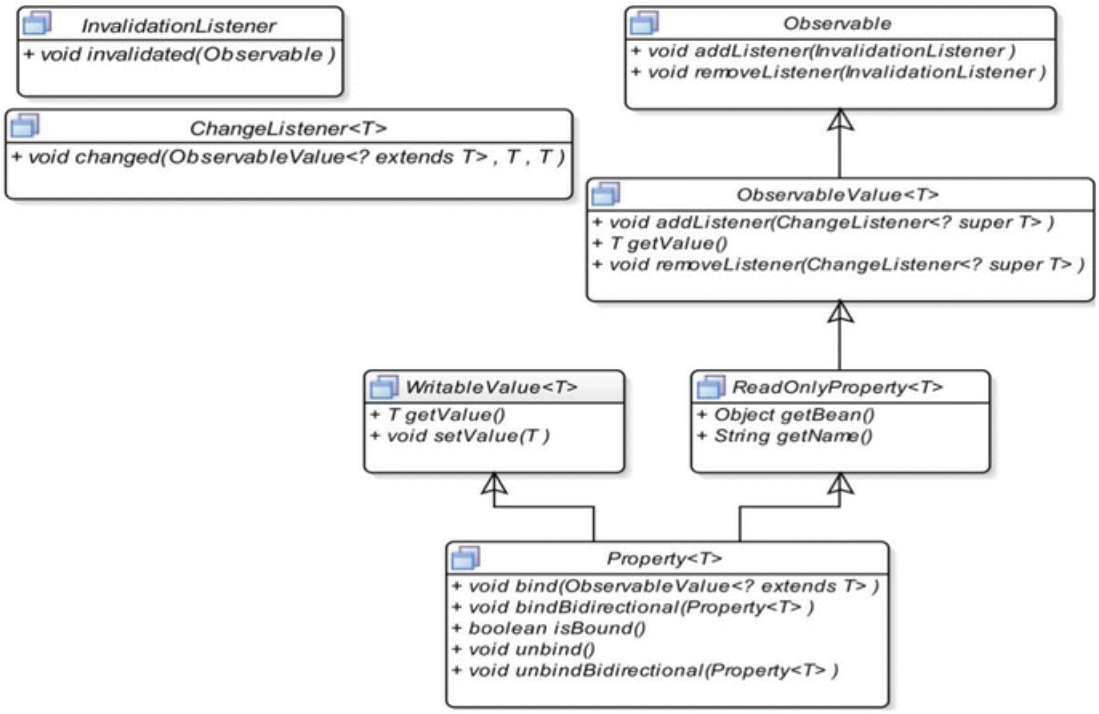


Рис. 79. Диаграмма классов для интерфейсов Свойств

Следует заметить, что у классов свойств имеются методы getValue() и setValue(), помимо методов get() и set(). В классе StringProperty метод get() равнозначен методу getValue(), а метод set() – методу setValue(). Но для примитивных типов они разные. Например, в классе IntegerProperty метод

getValue() возвращает объект типа Integer, а метод get() – значение типа int.

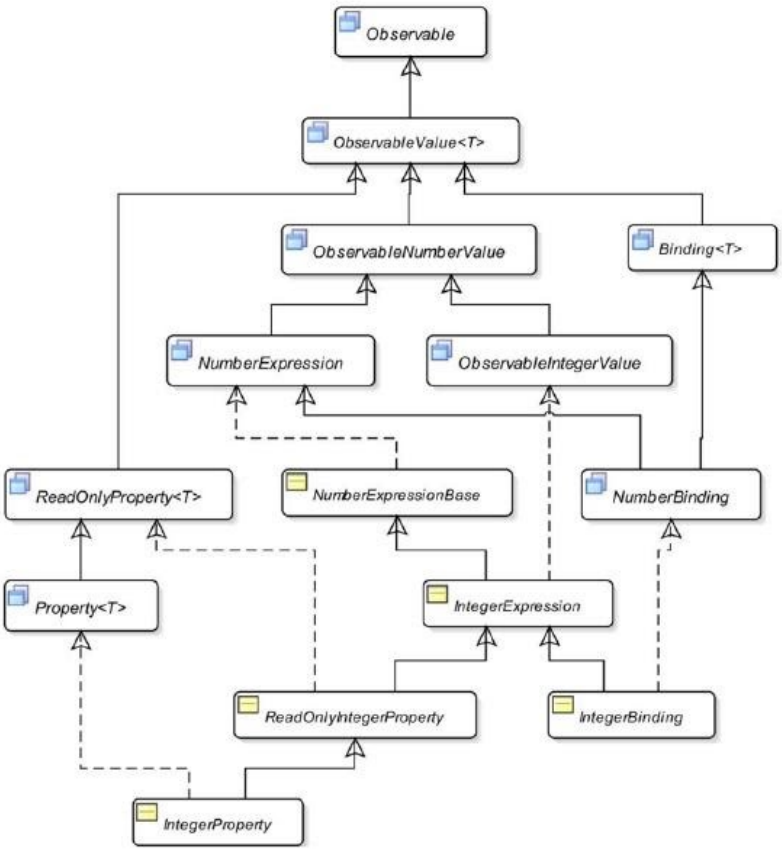


Рис. 80. Иерархия классов-наследников Свойств

Интерфейс Property включает ReadOnlyProperty and WritableValue, которые добавляют следующие методы поддержки связывания:

• void bind(ObservableValue<? extends T> observable) – добавляет однонаправленную связь между объектом Property и значением ObservableValue;

• void unbind() – разрывает связь;

• void bindBidirectional(Property<T> other) – устанавливает двунаправленную связь между двумя объектами, поддерживающими интерфейс Property;

• void unbindBidirectional(Property<T> other);

• boolean isBound() – возвращает true если к объекту установлена связь.

К свойству могут быть присоединены две разновидности приемников событий.

ChangeListener уведомляется, когда значение свойства было изменено.

InvalidationListener вызывается, когда значение свойства может быть изменено.

Для того, что выполнить над Свойствами действия и дальше работать с новыми Свойствами предусмотрен класс Bindings, имеющий методы add, subtract, multiply, divide, max, min и другие для Свойств типа

ObservableNumberValue, int, long, float,double.

Простой алгоритм связывания состоит в следующем.

1. Определяются объекты-свойства, изменение в которых будет отслеживаться, например:

```
private SimpleDoubleProperty topXProperty =
  new SimpleDoubleProperty();
private SimpleDoubleProperty topYProperty =
  new SimpleDoubleProperty();
```

2. Выделяется «следящий» объект, изменяющий свое состоятие в зависимости от значения свойтв, например:

```
Line foldLine = new Line();
foldLine.setEndX(200);
foldLine.setEndY(200);
```

3. Связывается изменяемый объект и свойства с помощью методов bind() или bindBidirectional() для двухстороннего связывания:

```
foldLine.startXProperty().bind(topXProperty);
foldLine.startYProperty().bind(topYProperty);
```

4. Изменение свойств должно происходить либо через GUI, либо программно:

```
topXProperty.set(tx);
topYProperty.set(ty);
```

Элементы управления, наследуемые от Control, уже имеют свойства и связывание осуществляется напрямую. Например:

```
public TextField ftime;
public Label fname;
fname.textProperty().bind(ftime.nameProperty());
```

Обратите внимание, для некторых элементов, например текстовых меток двухстронняя связь в принципе невозможна.

При необходимости непосредственного вычисления свойств следует использовать NumberBinding объект, например:

```
NumberBinding result=Bindings.multiply(topXProperty, topYProperty)
```

Поля передают только строковый формат, а в модели (некоторый объект model) могут быть свойства целого и вещественного типа. Следовательно, необходима взаимная конвертация из целого в строку и обратно, в методе bindBidirectional предусмотрен параметр, который задает способ конвертации посредством StringConverter <T>, где объект T должен быть определен, для свойств Double и Integer можно использовать родительский класс Number:

```
StringConverter<Number> converter = new NumberStringConverter();
ftime.textProperty().bindBidirectional(model.timeProperty(),
converter);
```

Пошаговая реализация модуля авторизации в системе тестирования

Требования: программа в зависимости от уровня авторизации – студент/преподаватель/посетитель выводит окно с разными стилями оформления.

Диаграмма классов приложения представлена на рис. 81.

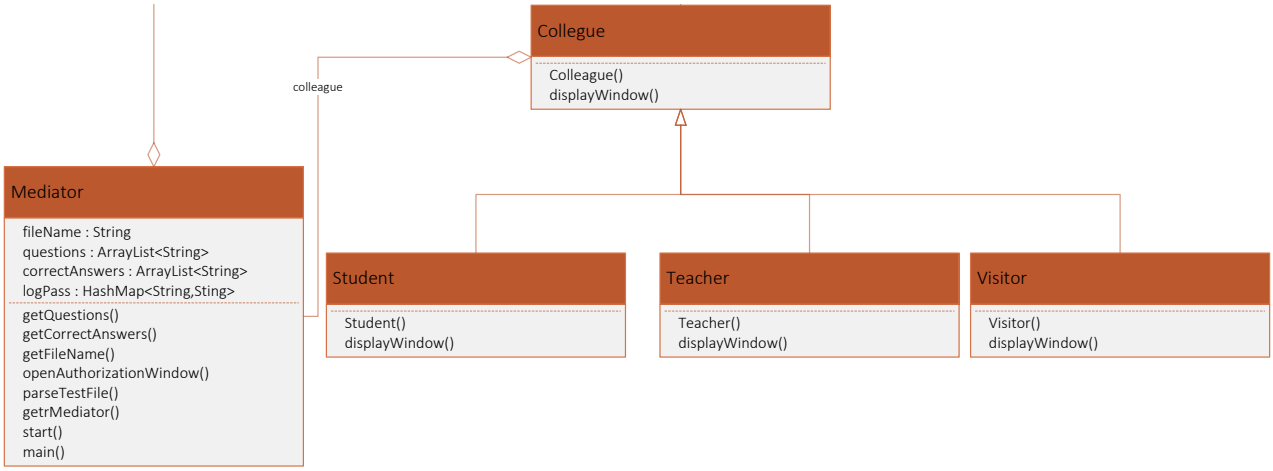


Рис. 81. Диаграмма классов модуля авторизации в системе тестирования

Шаг 1. Проектирование fxml-формы с двумя тестовыми полями login и password для авторизации.

Разработка трех различных пользовательских интерфейсов для режимов работы студента (student.fxml), преподавателя (teacher.fxml) и посетителя (visitor.fxml).

Шаг 2. Создание абстрактный класс Colleague, который инкапсулирует метод displayWindow и обеспечивает взаимодействие множества объектов для для работы с файлом по заданному уровню доступа.

```
abstract class Colleague {
    Mediator mediator;
    Colleague(Mediator mediator) {
        this.mediator = mediator;
    }
    public abstract void displayWindow();
}
```

Реализация трех различных классов ConcreteColleague, в рамках данной задачи – это Visitor, Student и Teacher. Каждый из этих классов наследуется от Colleague и переопределяет метод displayWindow, чтобы реализовать работу в одном из трех режимов - только просмотр, возможность ответа на вопросы и редактирования файла.

В displayWindow() каждого класса реализуется загрузка соотвествующего специально реализованного GUI:

```
FXMLLoader loader = new
FXMLLoader(getClass().getResource("visitor.fxml"));
Parent mainCallWindowFXML = null;
try {
    mainCallWindowFXML = loader.load();
```

```
    Stage stage = (Stage)
login.getParent().getScene().getWindow();//login компонент формы
текущего Controller, необходим для доступа к родительскому Stage
    stage.setScene(new Scene(mainCallWindowFXML, 800, 600));
    stage.show(); //показ вновь загруженной сцены
} catch (IOException e) {
    e.printStackTrace();
```

Шаг 3. В контроллере формы ввода логина и пароля реализуется функционал интерфейса Mediator, который позволяет создать объект конкретного Colleague в зависимости от входных данных.

```
if (login.equals(logPass.get("Student")) &&
pass.equals(logPass.get("Student"))) {
    colleague = new Student(getMediator());
    ((Student) colleague).displayWindow();
    stage.close();
} else if (login.equals(logPass.get("Visitor")) &&
pass.equals(logPass.get("Visitor"))) {
    colleague = new Visitor(getMediator());
    ((Visitor) colleague).displayWindow();
    stage.close();
} else if (login.equals(logPass.get("Teacher")) &&
pass.equals(logPass.get("Teacher"))) {
    colleague = new Teacher(getMediator());
    ((Teacher) colleague).displayWindow();
    stage.close();
```

ЗАКЛЮЧЕНИЕ

Дисциплина «Технология программирования» предназначена для изучения методов и средств программирования, в частности шаблонов проектирования структурного уровня. Студенты, которые приступают к изучению данной дисциплины уже владеют навыками создания простых программ с помощью языков программирования Java, поэтому в данном практикуме поставлена цель получения практических навыков использования шаблонов проектирования при разработке программного обеспечения.

Практикум описывает построение проектов и особенности работы с JavaFX, представлены базовые принципы ООП, необходимые для использования паттернов и задания на закрепление знаний ООП. Основную часть практикума составляют методические указания к выполнению лабораторных работ по освоению паттернов проектирования Go&F.

Выполнение работ по данному практикуму должно способствовать развитию навыков выбора проектных решений и получение опыта использования паттернов при написании программ и разработки мультимедийных приложений на платфортме JavaFX.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования = Design Patterns: Elements of Reusable Object — Oriented Software. — СПб: «Питер», 2007. — с. 366.

2. Гранд М. Шаблоны проектирования в JAVA. Каталог популярных шаблонов проектирования, проиллюстрированных при помощи UML = Patterns in Java, Volume 1. A Catalog of Reusable Design Patterns Illustrated with UML. — М.: «Новое знание», 2004. – 560 с.

3. Обзор паттернов проектирования - http://citforum.ru/SE/project/pattem/

4. Объектно-ориентированное проектирование, паттерны проектирования (Шаблоны) - http://www.javenue.info/themes/ood/

5. Погружение в ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ - https://refactoring.guru/ru/design-patterns/book

6. Kuchana, Partha Software architecture design patterns in Java / Partha Kuchana - CRC Press, 2004 – 476 p.

7. Schildt, Herbert Introdicing Java FX programming – Oracle Press, 2005 – 357 p.

8. Sharan K. Learn JavaFX 8: building user experience and interfaces with Java 8. – Apress, 2015.

9. Topley K. JavaFX developer's guide. – Pearson Education, 2010.

10. Vivien V. JavaFX 1.2 Application Development Cookbook. – Packt Publishing Ltd, 2010.

ПРИЛОЖЕНИЕ

СОЗДАНИЕ ПРОЕКТА JAVAFX В СРЕДЕ INTELLIJ IDEA

JavaFX – это программная платформа, используемая для разработки настольных приложений, которые могут работать на самых разных устройствах.

Поддержка JavaFX в IntelliJ IDEA включает в себя завершение кода, поиск, навигацию и рефакторинг в специфичных для JavaFX исходных файлах (в том числе .fxml и JavaFX.css-файлы), интеграция с JavaFX Scene Builder, возможности упаковки приложений JavaFX и многое другое.

Для разработки приложений в IntelliJ IDEA понадобится Java SDK (JDK). JDK (Java Development Kit) – это программный пакет, содержащий библиотеки, инструменты для разработки и тестирования Java-приложений (development tools) и инструменты для запуска приложений на платформе Java (Java Runtime Environment — JRE).

Начиная с JDK 11, JavaFX больше не является частью JDK. Поэтому, если используется Java 11 и более поздние версии, необходимо загрузить SDK JavaFX с открытым исходным кодом в дополнение к JDK.

1. Создание и запуск JavaFX приложения на Java 11 и старше

Шаг 1. Скачивание JavaFX SDK.

1. Необходимо загрузить пакет SDK JavaFX, подходящий для используемой операционной системы (https://gluonhq.com/products/javafx/).

2. Далее необходимо распоковать архив и поместить папку в значимое место, например: /Users/jetbrains/Desktop/javafx-sdk-12.

Шаг 2. Создание нового проекта.

При создании нового проекта JavaFX IntelliJ IDEA генерирует полностью настроенный пример приложения. Для этого необходимо выполнить следующие действия:

1. Запуск IntelliJ IDEA. Если откроется окно приветствия, необходимо нажать кнопку «**Create New Project**». В противном случае в главном меню выполняется такая последовательность «**File | New | Project…**» (рис. П.1).

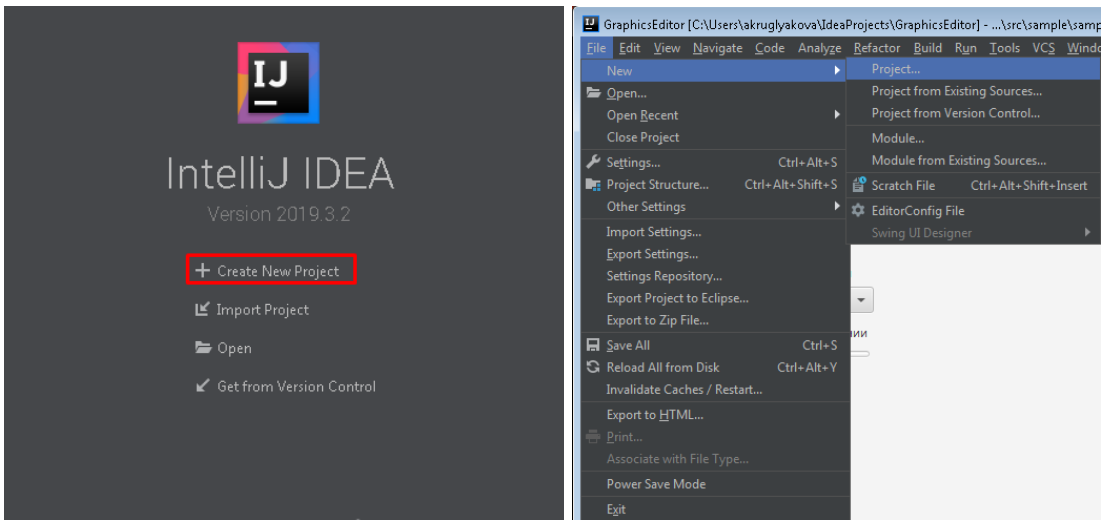


Рис. П.1. Создание проекта

2. В мастере создания проекта необходимо выбрать «**JavaFX**» из списка слева.

3. В списке Project SDK выбирается JDK, который будет использоваться в проекте. Если JDK установлен на компьютере, но не определен в IDE, нужно выбрать «**Add JDK**» и указать путь к домашнему каталогу JDK. Если на компьютере нет необходимого JDK, выбирается «**Download JDK**».

4. Далее необходимо выбрать «**JavaFX Application**», нажать на кнопку «**Next**» (рис. П.2).

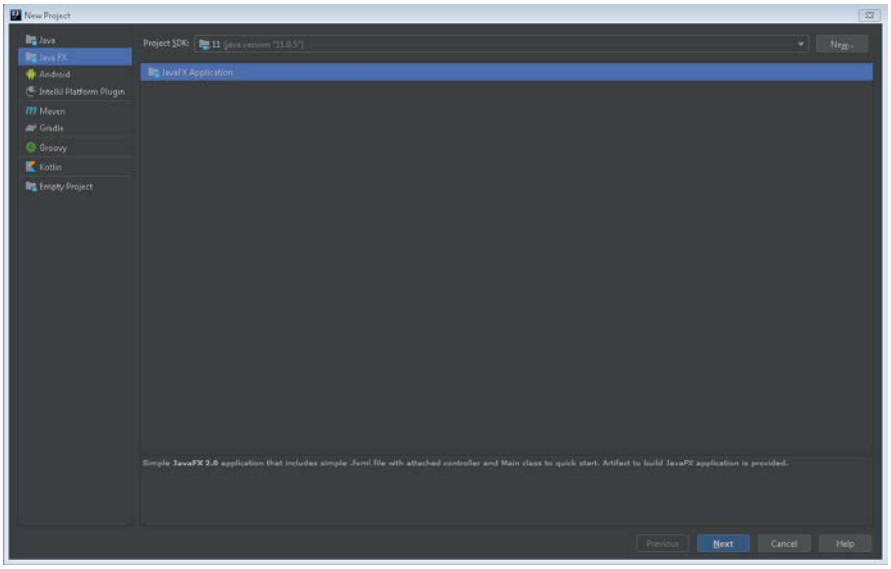


Рис. П.2. Задание параметров проекта

5. На следующем шаге мастера нужно назвать проект, например: «**FirstProject**». При необходимости можно изменить расположение проекта по умолчанию, затем необходимо нажать кнопку «**Finish**».

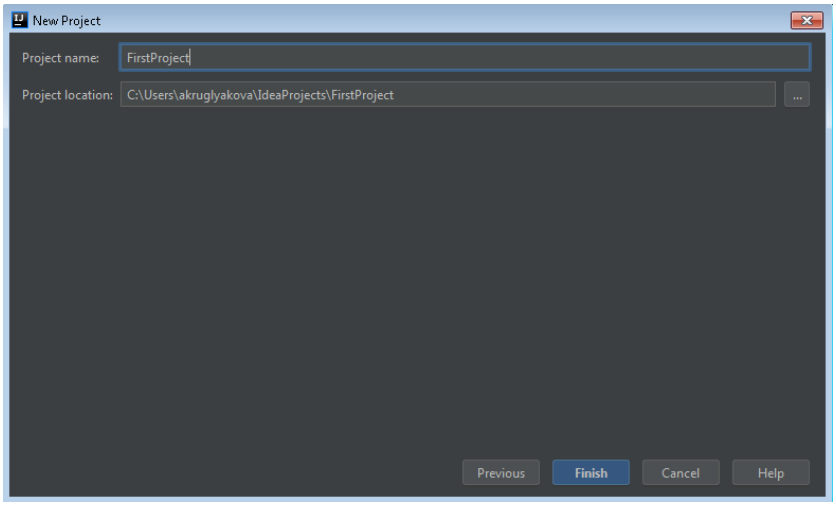


Рис. П.3. Задание имени проекта

Шаг 3. Добавление библиотеки JavaFX.

Эти действия необходимы, если используется Java 11 и более поздние версии. Если используется Java 10 и более ранние версии, перейдите к шагу 4.

1. В главном меню нужно выбрать пункт «**File** | **Project Structure**» (*Ctrl+Alt+Shift+S)* или нажать кнопку на панели инструментов (рис. П.4).

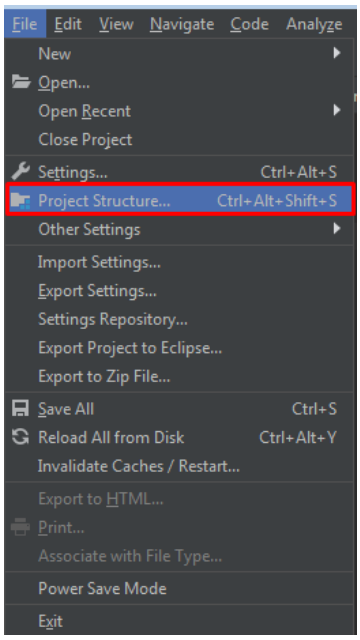


Рис. П.4. Открытие структуры проектов

2. Далее нужно открыть раздел «**Libraries**». Для этого нужно щелкнуть на кнопку + и выберать «**Java**» (рис. П.5).

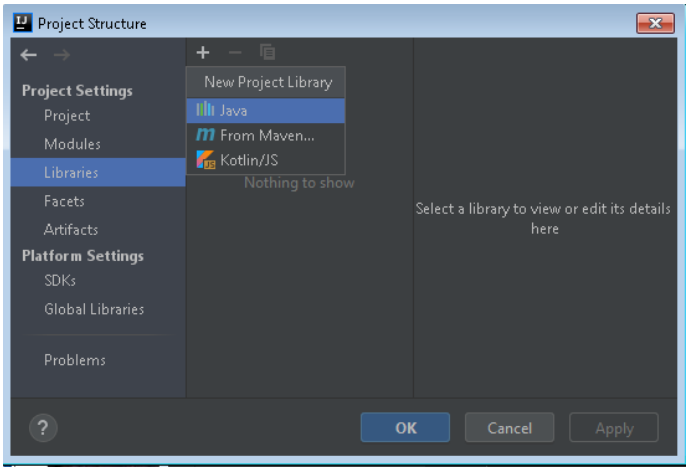


Рис. П.5. Добавление библиотеки

3. После нужно указать путь к папке «**lib**» в пакете JavaFX SDK, например: /Users/jetbrains/Desktop/javafx-sdk-12 / lib.

4. И на последнем шаге в диалоговом окне «**Choose Modules**» нужно выбрать необходимый модуль (ваш проект), в который будет загружена библиотека, и нажать кнопку «**ОК**» (рис. П.6).

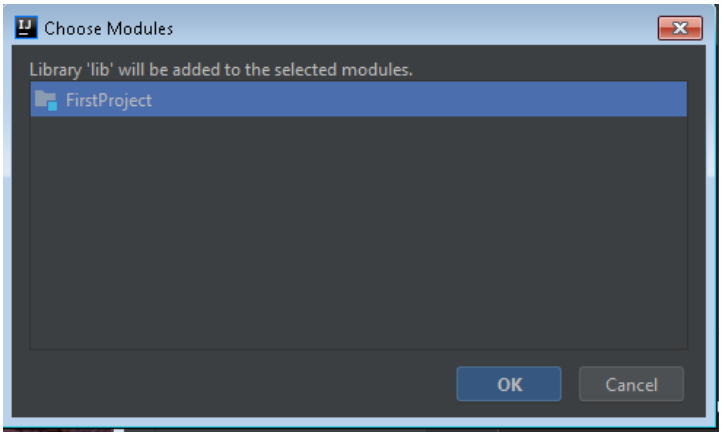


Рис. П.6. Выбор модуля

Шаг 4. Запуск приложения**.**

По умолчанию среда создаст проект со следующей структурой, представленной на рис. П.7.

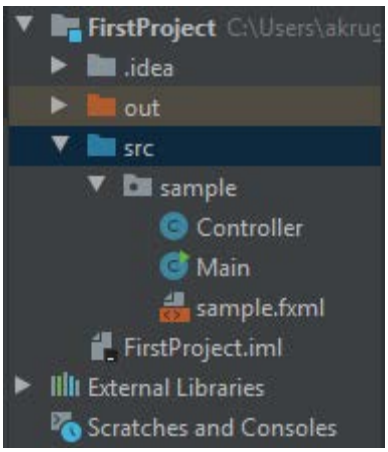


Рис. П.7. Структура проекта

Представление основного окна приложения описывает файл разметки FXML — **sample.fxml.**

Главный класс **Main** содержит метод запуска приложения main. И в этом методе, чтобы запустить приложение JavaFX, представленное классом Application, вызывается метод launch().

Пустой класс **Controller** предназначен для размещения обработчиков событий интерфейса, в рамках концепции MVC.

1. Для проверки работоспособности в главном меню нужно выбрать пункт «**Run | Run 'Main'**» (*Shift+F10*). IDE начинает компилировать ваш код. После завершения компиляции появится окно приложения.

2. Это означает, что проект настроен правильно и все работает так, как должно быть. В данный момент окно пусто, так как мы еще не добавили никаких элементов (рис. П.8).

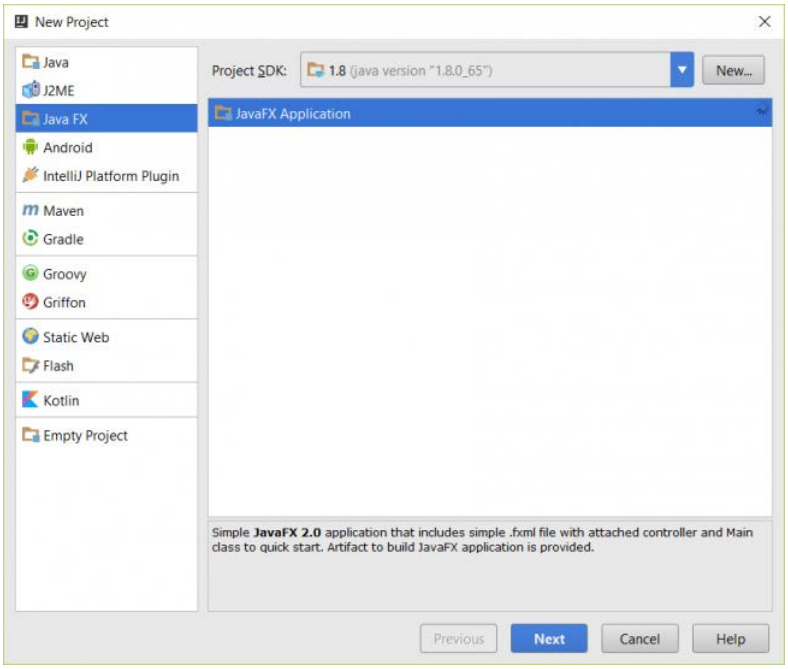
Рис. П.8. Результат запуска проекта

2. Создание и запуск JavaFX приложения на Java 8

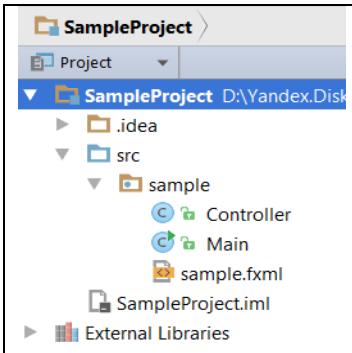
Чтобы создать JavaFX проект в IntelliJ IDEA, необходимо проделать следующее:

1. Выбрать "**Create New Project**" со стартовой страницы среды, или "**File -> New -> Project...**" из любого открытого проекта.

2. Слева выбрать вкладку "**JavaFX**", выбрать "**JavaFX Application**", нажать "**Next**", придумать название и выбрать расположение проекта.



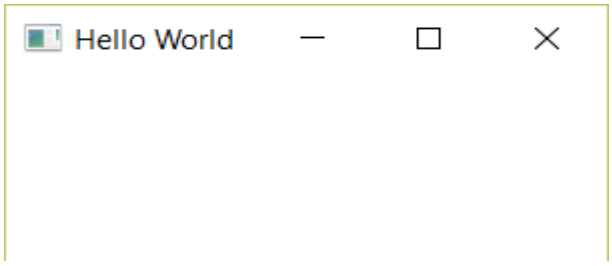
По умолчанию среда создаст проект со следующей структурой:



Представление основного окна приложения описывает файл разметки FXML — **sample.fxml**

Главный класс **Main** содержит метод запуска приложения main. И в этом методе, чтобы запустить приложение JavaFX, представленное классом Application, вызывается метод launch().

Пустой класс Controller предназначен для размещения обработчиков событий интерфейса, в рамках концепции MVC. Для запуска приложения Run -> Run Main или нажатие на зеленый треугольник получится следующий результат:



ОГЛАВЛЕНИЕ

Учебное издание

Минакова Ольга Владимировна

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ: ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ В РЕАЛИЗАЦИИ JAVAFX ПРИЛОЖЕНИЙ

Практикум

Редактор Е. А. Кусаинова

Подписано к изданию 24.12.2020.
Объем данных 6,5 Мб.

ФГБОУ ВО «Воронежский государственный технический университет»
394006 Воронеж, Московский проспект, 14