Министерство образования Республики Беларусь

Учреждение образования

«БелорусскиЙ государственный университет

информатики и радиоэлектроники»

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

Дисциплина «Основы программирования информационных систем»

|  |
| --- |
| *К защите допустить*: |
| Руководитель курсовой работымагистр технических наук, ассистент |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н. М. Бруй\_\_\_.\_\_\_. 20\_\_\_ |

**Пояснительная записка**

к курсовой работе

на тему

**Разработка приложения messenger**

БГУИР КР 1-40 05 01-10 № 139 ПЗ

|  |  |  |
| --- | --- | --- |
| Студент | (подпись студента) | Н.Мацаль |
|  |  | Курсовая работа представлена на проверку \_\_\_.\_\_\_. 20\_\_\_ |
|  |  | (подпись студента) |

Минск 2020

**РЕФЕРАТ**

БГУИР КР 1-40 05 01-10 № 139 ПЗ

Мацаль, Н.. Разработка приложения messenger / Н.. Мацаль. – Минск: БГУИР, 2020. – 43 с.

Пояснительная записка 43 с., 17 рис., 5 источников.

Ключевые слова: Приложение, messenger.

Предмет: Объектно-ориентированное программирование, А.В.Михалькевич.

Объект: шаблоны проектирования, разработка пользовательского интерфейса и бизнес-логики.

Цель: РАЗРАБОТКА ПРИЛОЖЕНИЯ MESSENGER

Методология проведения работы: разработка логики и реализации чата с регистрацией.

Результаты работы: разработан чат для пользователей на языке java.

Область применения результатов: предназначено для пользователей, которые нуждаются в общении.

Содержание

ВВЕДЕНИЕ 7

1 ОПИСАНИЕ ПРОЕКТА 8

**1.1** **Клиентская часть** 8

**1.2** **Серверная часть** 12

2 ОБОСНОВАНИЕ ВЫБОРА ТЕХНОЛОГИЙ 16

**2.1** **Объектно-ориентированное программирование** 16

**2.2 Язык программирования java** 19

3 ИНСТРУМЕНТАРИЙ 21

**3.1 IntelliJ** 21

**3.2 Система контроля версий GIT** 21

**3.3 JavaFX** 25

4 АРХИТЕКТУРНЫЙ ШАБЛОН ПРОЕКТИРОВАНИЯ MVVM 26

5 ШАБЛОНЫ ПРОЕКТИРОВАНИЯ 29

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 34

ПРИЛОЖЕНИЕ А 35

**ВВЕДЕНИЕ**

Приложение для чата между двумя пользователями не ново, но я хотел реализовать его в своей манере для того чтобы представить новый продукт и заручится опыт разработки подобных приложений.

 Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Проектирование пользовательского интерфейса.
2. Построение структуры приложения, соответствующей архитектуре MVVM. .

**1 ОПИСАНИЕ ПРОЕКТА**

* 1. **Клиентская часть**

Клиентская часть — это аналог winApi только построенные на javaFX

Дизайн собой представляет набор кнопок, окон, выпадающих списков и тд. Быстрая типизация таких вещей посредством накидывания кнопок на форму, а из этого получение готовых fxml файлов очень сильно упрощает разработку клиентской части

FXML – язык разметки созданный на основе XML, позволяет определить интерфейс приложения декларативным способом тому, как веб-страницы определяются с помощью HTML



Рисунок 1 – Наследование javafx

Панели компоновки позволяют упорядочить вложенные элементы определенным образом и могут определять их размеры. В JavaFX есть несколько встроенных панелей компоновки:

AnchorPane: позволяет прижимать вложенные элементы управления к одной из сторон контейнера или размещать их по центру

BorderPane: поволяет упорядочить вложенные элементы управления относительно одной из сторон контейнера

FlowPane: размещает все вложенные элементы либо по горизонтали (если элементы не помещаются, то они переносятся на новую строку), либо по вертикали (если элементы не помещаются, то они переносятся в новый столбик)

GridPane: образует сетку из строк и столбцов и размещает вложенные элементы ячейках полученной таблицы

HBox: размещает все элементы в виде одной строки, то есть горизонтальный стек

VBox: размещает все элементы в виде одного столбца, то есть вертикальный стек

StackPane: располагает одни элементы поверх других

TilePane: размещает все вложенные элементы в виде тайлов

Поскольку все эти классы наследуются от класса Pane, то они все они имеют один важный метод - getChildren(), который возвращает коллекцию вложенных в контейнер элементов в виде объекта интерфейса ObservableList<Node>. Интерфейс ObservableList, в свою очередь, расширяет интерфейсы Collection<E>, Iterable<E>, List<E>, Observable.

Каждый вложенный элемент представляет объект Node. Используя методы ObservableList, мы можем добавлять узлы Node в эту коллекцию, а по сути в контейнер компоновки.

Наиболее часто используемые элементы управления в JavaFX - это, без сомнения, кнопки и метки. Кнопки представлены классом javafx.scene.control.Button и позволяют по нажатию выполнять некоторые действия. А метки представлены классы javafx.scene.control.Label, и их задача - вывод текстовой информации. Класс Button унаследован от классов ButtonBase, Labeled, Region, Control, Parent и Node, а класс Label - от Labeled, Region, Control, Parent и Node. JavaFX позволяет создавать самые разнообразные кнопки и метки: с текстом, графикой, и тем и другим, с различными стилями.

Класс Button имеет три конструктора:

Button(): создает кнопку без надписи

Button​(String text): создает кнопку с определенной надписью

Button​(String text, Node graphic): создает кнопку с определенной надписью и иконкой

Класс Label имеет три похожих конструктора:

Label(): создает метку без надписи

Label(String text): создает метку с определенной надписью

Label(String text, Node graphic): создает метку с определенной надписью и иконкой

* 1. **Серверная часть**

### На сервере мы в свою очередь работаем с пулом процессов.Сейчас конкретно разберемся что это такое.

### Потоки и задачи

Все современные операционные системы поддерживают параллельное выполнение кода с помощью процессов и потоков. Процесс — это экземпляр программы, который запускается независимо от остальных. Например, когда вы запускаете программу на Java, ОС создает новый процесс, который работает параллельно другим. Внутри процессов мы можем использовать потоки, тем самым выжав из процессора максимум возможностей.

Потоки *(threads)* в Java поддерживаются начиная с JDK 1.0. Прежде чем запустить поток, ему надо предоставить участок кода, который обычно называется «задачей» *(task)*. Это делается через реализацию интерфейса Runnable, у которого есть только один метод без аргументов, возвращающий void — run(). Вот пример того, как это работает:

Runnable task = () -> {

 String threadName = Thread.currentThread().getName();

 System.out.println("Hello " + threadName);

};

task.run();

Thread thread = new Thread(task);

thread.start();

System.out.println("Done!");

Поскольку интерфейс Runnable функциональный, мы можем использовать лямбда-выражения, которые появились в Java 8. В примере мы создаем задачу, которая выводит имя текущего потока на консоль, и запускаем ее сначала в главном потоке, а затем — в отдельном.

Результат выполнения этого кода может выглядеть так:

Hello main

Hello Thread-0

Done!

или так:

Hello main

Done!

Hello Thread-0

Из-за параллельного выполнения мы не можем сказать, будет наш поток запущен до или после вывода «Done!» на экран. Эта особенность делает параллельное программирование сложной задачей в больших приложениях.

Потоки могут быть приостановлены на некоторое время. Это весьма полезно, если мы хотим сэмулировать долго выполняющуюся задачу. Например, так:

Runnable runnable = () -> {

 try {

 String name = Thread.currentThread().getName();

 System.out.println("Foo " + name);

 TimeUnit.SECONDS.sleep(1);

 System.out.println("Bar " + name);

 }

 catch (InterruptedException e) {

 e.printStackTrace();

 }

};

Thread thread = new Thread(runnable);

thread.start();

Когда вы запустите этот код, вы увидите секундную задержку между выводом первой и второй строки на экран. TimeUnit — полезный класс для работы с единицами времени, но то же самое можно сделать с помощью Thread.sleep(1000).

Работать с потоками напрямую неудобно и чревато ошибками. Поэтому в 2004 году в Java 5 добавили Concurrency API. Он находится в пакете java.util.concurrent и содержит большое количество полезных классов и методов для многопоточного программирования. С тех пор Concurrency API непрерывно развивался и развивается.

Давайте теперь подробнее рассмотрим одну из самых важных частей Concurrency API — сервис исполнителей *(executor services)*.

### Исполнители

Concurrency API вводит понятие сервиса-исполнителя *(ExecutorService)* — высокоуровневую замену работе с потоками напрямую. Исполнители выполняют задачи асинхронно и обычно используют пул потоков, так что нам не надо создавать их вручную. Все потоки из пула будут использованы повторно после выполнения задачи, а значит, мы можем создать в приложении столько задач, сколько хотим, используя один исполнитель.

Вот как будет выглядеть наш первый пример с использованием исполнителя:

ExecutorService executor = Executors.newSingleThreadExecutor();

executor.submit(() -> {

 String threadName = Thread.currentThread().getName();

 System.out.println("Hello " + threadName);

});

// => Hello pool-1-thread-1

Класс Executors предоставляет удобные методы-фабрики для создания различных сервисов исполнителей. В данном случае мы использовали исполнитель с одним потоком.

Результат выглядит так же, как в прошлый раз. Но у этого кода есть важное отличие — он никогда не остановится. Работу исполнителей надо завершать явно. Для этого в интерфейсе ExecutorService есть два метода: shutdown(), который ждет завершения запущенных задач, и shutdownNow(), который останавливает исполнитель немедленно.

Вот как я предпочитаю останавливать исполнителей:

try {

 System.out.println("attempt to shutdown executor");

 executor.shutdown();

 executor.awaitTermination(5, TimeUnit.SECONDS);

}

catch (InterruptedException e) {

 System.err.println("tasks interrupted");

}

finally {

 if (!executor.isTerminated()) {

 System.err.println("cancel non-finished tasks");

 }

 executor.shutdownNow();

 System.out.println("shutdown finished");

}

Исполнитель пытается завершить работу, ожидая завершения запущенных задач в течение определенного времени (5 секунд). По истечении этого времени он останавливается, прерывая все незавершенные задачи.

#### Callable и Future

Кроме Runnable, исполнители могут принимать другой вид задач, который называется Callable. Callable — это также функциональный интерфейс, но, в отличие от Runnable, он может возвращать значение.

Давайте напишем задачу, которая возвращает целое число после секундной паузы:

Callable task = () -> {

 try {

 TimeUnit.SECONDS.sleep(1);

 return 123;

 }

 catch (InterruptedException e) {

 throw new IllegalStateException("task interrupted", e);

 }

};

Callable-задачи также могут быть переданы исполнителям. Но как тогда получить результат, который они возвращают? Поскольку метод submit() не ждет завершения задачи, исполнитель не может вернуть результат задачи напрямую. Вместо этого исполнитель возвращает специальный объект Future, у которого мы сможем запросить результат задачи.

ExecutorService executor = Executors.newFixedThreadPool(1);

Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());

System.out.print("result: " + result);

После отправки задачи исполнителю мы сначала проверяем, завершено ли ее выполнение, с помощью метода isDone(). Поскольку задача имеет задержку в одну секунду, прежде чем вернуть число, я более чем уверен, что она еще не завершена.

Вызов метода get() блокирует поток и ждет завершения задачи, а затем возвращает результат ее выполнения. Теперь future.isDone() вернет true, и мы увидим на консоли следующее:

future done? false

future done? true

result: 123

Задачи жестко связаны с сервисом исполнителей, и, если вы его остановите, попытка получить результат задачи выбросит исключение:

executor.shutdownNow();

future.get();

Вы, возможно, заметили, что на этот раз мы создаем сервис немного по-другому: с помощью метода newFixedThreadPool(1), который вернет исполнителя с пулом в один поток. Это эквивалентно вызову метода newSingleThreadExecutor(), однако мы можем изменить количество потоков в пуле.

#### Таймауты

Любой вызов метода future.get() блокирует поток до тех пор, пока задача не будет завершена. В наихудшем случае выполнение задачи не завершится никогда, блокируя ваше приложение. Избежать этого можно, передав таймаут:

ExecutorService executor = Executors.newFixedThreadPool(1);

Future<Integer> future = executor.submit(() -> {

 try {

 TimeUnit.SECONDS.sleep(2);

 return 123;

 }

 catch (InterruptedException e) {

 throw new IllegalStateException("task interrupted", e);

 }

});

future.get(1, TimeUnit.SECONDS);

Выполнение этого кода вызовет TimeoutException:

Exception in thread "main" java.util.concurrent.TimeoutException

 at java.util.concurrent.FutureTask.get(FutureTask.java:205)

Вы уже, возможно, догадались, почему было выброшено это исключение: мы указали максимальное время ожидания выполнения задачи в одну секунду, в то время как ее выполнение занимает две.

#### InvokeAll

Исполнители могут принимать список задач на выполнение с помощью метода invokeAll(), который принимает коллекцию callable-задач и возвращает список из Future.

ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(

 () -> "task1",

 () -> "task2",

 () -> "task3");

executor.invokeAll(callables)

 .stream()

 .map(future -> {

 try {

 return future.get();

 }

 catch (Exception e) {

 throw new IllegalStateException(e);

 }

 })

 .forEach(System.out::println);

В этом примере мы использовали функциональные потоки Java 8 для обработки задач, возвращенных методом invokeAll. Мы прошлись по всем задачам и вывели их результат на консоль. Если вы не знакомы с потоками (streams) Java 8, смотрите мое руководство.

#### InvokeAny

Другой способ отдать на выполнение несколько задач — метод invokeAny(). Он работает немного по-другому: вместо возврата Future он блокирует поток до того, как завершится хоть одна задача, и возвращает ее результат.

Чтобы показать, как работает этот метод, создадим метод, эмулирующий поведение различных задач. Он будет возвращать Callable, который вернет указанную строку после необходимой задержки:

Callable callable(String result, long sleepSeconds) {

 return () -> {

 TimeUnit.SECONDS.sleep(sleepSeconds);

 return result;

 };

}

Используем этот метод, чтобы создать несколько задач с разными строками и задержками от одной до трех секунд. Отправка этих задач исполнителю через метод invokeAny() вернет результат задачи с наименьшей задержкой. В данном случае это «task2»:

ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(

 callable("task1", 2),

 callable("task2", 1),

 callable("task3", 3));

String result = executor.invokeAny(callables);

System.out.println(result);

// => task2

В примере выше использован еще один вид исполнителей, который создается с помощью метода newWorkStealingPool(). Этот метод появился в Java 8 и ведет себя не так, как другие: вместо использования фиксированного количества потоков он создает ForkJoinPool с определенным параллелизмом *(parallelism size)*, по умолчанию равным количеству ядер машины.

ForkJoinPool впервые появился в Java 7, и мы рассмотрим его подробнее в следующих частях нашего руководства. А теперь давайте посмотрим на исполнители с планировщиком *(scheduled executors)*.

### Исполнители с планировщиком

Мы уже знаем, как отдать задачу исполнителю и получить ее результат. Для того, чтобы периодически запускать задачу, мы можем использовать пул потоков с планировщиком.

ScheduledExecutorService способен запускать задачи один или несколько раз с заданным интервалом.

Этот пример показывает, как заставить исполнитель выполнить задачу через три секунды:

ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());

ScheduledFuture<?> future = executor.schedule(task, 3, TimeUnit.SECONDS);

TimeUnit.MILLISECONDS.sleep(1337);

long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);

System.out.printf("Remaining Delay: %sms", remainingDelay);

Когда мы передаем задачу планировщику, он возвращает особый тип Future — ScheduledFuture, который предоставляет метод getDelay() для получения оставшегося до запуска времени.

У исполнителя с планировщиком есть два метода для установки задач: scheduleAtFixedRate() и scheduleWithFixedDelay(). Первый устанавливает задачи с определенным интервалом, например, в одну секунду:

ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());

int initialDelay = 0;

int period = 1;

executor.scheduleAtFixedRate(task, initialDelay, period, TimeUnit.SECONDS);

Кроме того, он принимает начальную задержку, которая определяет время до первого запуска.

#  2 ОБОСНОВАНИЕ ВЫБОРА ТЕХНОЛОГИЙ

 **2.1** **Объектно-ориентированное программирование**

Объектно-ориентированное программирование (в дальнейшем ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов. В центре ООП находится понятие объекта.

Объект — это сущность, экземпляр класса, которой можно посылать сообщения и которая может на них реагировать, используя свои данные. Данные объекта скрыты от остальной программы. Сокрытие данных называется инкапсуляцией.

Наличие инкапсуляции достаточно для объектности языка программирования, но ещё не означает его объектной ориентированности — для этого требуется наличие наследования. Но даже наличие инкапсуляции и наследования не делает язык программирования в полной мере объектным с точки зрения ООП. Основные преимущества ООП проявляются только в том случае, когда в языке программирования реализован полиморфизм, то есть возможность объектов с одинаковой спецификацией иметь различную реализацию.

Абстрагирование — это способ выделить набор значимых характеристик объекта, исключая из рассмотрения не значимые. Соответственно, абстракция — это набор всех таких характеристик.

В контексте ООП абстракция — это обобщение данных и поведения для типа, находящегося выше текущего класса по иерархии. Перемещая переменные или методы из подкласса в супер класс, вы обобщаете их. Но язык добавляет также понятия абстрактных классов и абстрактных методов.

Абстрактный класс является классом, для которого нельзя создать экземпляр.

Инкапсуляция — это свойство системы, позволяющее объединить данные и методы, работающие с ними в классе, и скрыть детали реализации от пользователя. Инкапсуляция позволит скрыть детали реализации, и открыть только то что необходимо в последующем использовании. Другими словами, инкапсуляция – это механизм контроля доступа.

Цель инкапсуляции – уйти от зависимости внешнего интерфейса класса (то, что могут использовать другие классы) от реализации. Чтобы малейшее изменение в классе не влекло за собой изменение внешнего поведения класса.

Существует 4 вида модификаторов доступа: public, protected, private и default.

Public – уровень предполагает доступ к компоненту с этим модификатором из экземпляра любого класса и любого пакета.

Protected – уровень предполагает доступ к компоненту с этим модификатором из экземпляров родного класса и классов-потомков, независимо от того, в каком пакете они находятся.

Default – уровень предполагает доступ к компоненту с этим модификатором из экземпляров любых классов, находящихся в одном пакете с этим классом.

Private – уровень предполагает доступ к компоненту с этим модификатором только из этого класса.

Наследование — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом.

Наследование — это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов. Применение иерархии классов делает управляемыми большие потоки информации.

Полиморфизм — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

В более общем смысле, концепцией полиморфизма является идея “один интерфейс, множество методов“. Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор.

Говоря про ООП нужно сказать про принципы SOLID, каждое грамотное приложение соответствует этим принципам.

SOLID — это аббревиатура пяти основных принципов проектирования в объектно-ориентированном программировании — Single responsibility, Open-closed, Liskov substitution, Interface segregation и Dependency inversion. В переводе на русский: принципы единственной ответственности, открытости / закрытости, подстановки Барбары Лисков, разделения интерфейса и инверсии зависимостей)

Аббревиатура SOLID была предложена Робертом Мартином, автором нескольких книг, широко известных в сообществе разработчиков. Эти принципы позволяют строить на базе ООП масштабируемые и сопровождаемые программные продукты с понятной бизнес-логикой.

**Расшифровка:**

* Single responsibility — принцип единственной ответственности
* Open-closed — принцип открытости / закрытости
* Liskov substitution — принцип подстановки Барбары Лисков
* Interface segregation — принцип разделения интерфейса
* Dependency inversion — принцип инверсии зависимостей

**Принцип единственной обязанности / ответственности** (single responsibility principle / SRP) обозначает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности

**Принцип открытости / закрытости** (open-closed principle / OCP) декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.

**Принцип подстановки Барбары Лисков** (Liskov substitution principle / LSP) в формулировке Роберта Мартина: «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом».

 **Принцип разделения интерфейса** (interface segregation principle / ISP) в формулировке Роберта Мартина: «клиенты не должны зависеть от методов, которые они не используют». Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

**Принцип инверсии зависимостей** (dependency inversion principle / DIP) — модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.

**2.2 Язык программирования java**

Java –  строго типизированный объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle). Разработка ведётся сообществом, организованным через Java Community Process, язык и основные реализующие его технологии распространяются по лицензии GPL. Права на торговую марку принадлежат корпорации Oracle.

Программы на Java транслируются в байт-код Java, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

Часто к недостаткам концепции виртуальной машины относят снижение производительности. Ряд усовершенствований несколько увеличил скорость выполнения программ на Java:

* применение технологии трансляции байт-кода в машинный код непосредственно во время работы программы (JIT-технология) с возможностью сохранения версий класса в машинном коде,
* обширное использование платформенно-ориентированного кода (native-код) в стандартных библиотеках,
* аппаратные средства, обеспечивающие ускоренную обработку байт-кода (например, технология Jazelle, поддерживаемая некоторыми процессорами архитектуры ARM).

По данным сайта shootout.alioth.debian.org, для семи разных задач время выполнения на Java составляет в среднем в полтора-два раза больше, чем для C/C++, в некоторых случаях Java быстрее, а в отдельных случаях в 7 раз медленнее. С другой стороны, для большинства из них потребление памяти Java-машиной было в 10—30 раз больше, чем программой на C/C++. Также примечательно исследование, проведённое компанией Google, согласно которому отмечается существенно более низкая производительность и бо́льшее потребление памяти в тестовых примерах на Java в сравнении с аналогичными программами на C++

Идеи, заложенные в концепцию и различные реализации среды виртуальной машины Java, вдохновили множество энтузиастов на расширение перечня языков, которые могли бы быть использованы для создания программ, исполняемых на виртуальной машине. Эти идеи нашли также выражение в спецификации общеязыковой инфраструктуры CLI, заложенной в основу платформы .NET компанией Microsoft.

# 3 ИНСТРУМЕНТАРИЙ

 **3.1 IntelliJ**

**IntelliJ IDEA** — интегрированная среда разработки программного обеспечения для многих языков программирования, в частности Java, JavaScript, Python, разработанная компанией JetBrains.

Первая версия появилась в январе 2001 года и быстро приобрела популярность как первая среда для Java с широким набором интегрированных инструментов для рефакторинга, которые позволяли программистам быстро реорганизовывать исходные тексты программ. Дизайн среды ориентирован на продуктивность работы программистов, позволяя сконцентрироваться на функциональных задачах, в то время как IntelliJ IDEA берёт на себя выполнение рутинных операций.

Начиная с шестой версии продукта IntelliJ IDEA предоставляет интегрированный инструментарий для разработки графического пользовательского интерфейса. Среди прочих возможностей, среда хорошо совместима со многими популярными свободными инструментами разработчиков, такими как CVS, Subversion, Apache Ant, Maven и JUnit. В феврале 2007 года разработчики IntelliJ анонсировали раннюю версию плагина для поддержки программирования на языке Ruby.

Начиная с версии 9.0, среда доступна в двух редакциях: ***Community Edition*** и ***Ultimate Edition***. Community Edition является полностью свободной версией, доступной под лицензией Apache 2.0, в ней реализована полная поддержка Java SE, Kotlin, Groovy, Scala, а также интеграция с наиболее популярными системами управления версиями. В редакции Ultimate Edition, доступной под коммерческой лицензией, реализована поддержка Java EE, UML-диаграмм, подсчёт покрытия кода, а также поддержка других систем управления версиями, языков и фреймворков.

**3.2 Система контроля версий GIT**

Git — это система управления версиями. У Git две основных задачи: первая — хранить информацию о всех изменениях в коде, начиная с самой первой строчки, а вторая — обеспечение удобства командной работы над кодом. (рис. 13)

Репозиторий Git — это место, где хранится код и вся информация о его изменениях. Репозитории могут находиться на компьютере, на компьютерах коллег и на удалённом сервере.

Главные преимущества систем управлениями версиями:

* Доступ к коду. Исходники кода хранятся в удаленном репозитории (хранилище данных), куда обращаются разработчики, чтобы забрать актуальную версию файлов или внести изменения. Так выстраивается командная разработка.
* Логирование изменений в коде. Отслеживание коммитов (внесений изменений в код), помогает найти кто, что и когда менял, решить конфликты при модифицировании одних и тех же файлов, откатиться на любое предыдущее состояние.
* Ветвление разработки. Программисты параллельно ведут разработку нового функционала в отдельных ветках, не затрагивая работоспособности старого.
* Поддержка версионности продуктов. При выпуске обновлений программных продуктов, мы обозначаем релизные версии, например, с помощью тегов, чтобы зафиксировать их в этом состоянии, для дебага или ретроспективы.

Главные преимущества Git по сравнению с другими системами управления версиями:

* Распределённая разработка;
* Транзакционный подход в управлении пакетами;
* Простота управления исходным кодом;
* Простота и удобство создания патчей;
* Интеграция в VCS апстрима, в том числе и по истории;
* Прозрачная сборка как локально, так и на сервере (в последнем случае нужно лишь создать подписанный тег и задать команду на сборку по ssh);
* Быстрое бэкпортирование;
* Малый трафик при постановке на сборку в репозиторий;
* Разнообразные проверки собираемых пакетов.



Рисунок 13 – принцип работы с Git

Gitflow является методологией работы с Git. Это значит, она определяет, какие ветки нужно создать и как производить их слияние. Набор инструментов git-flow нужно установить отдельно. Пакеты команд git-flow доступны во многих операционных системах. Для системы OSX можно выполнить brew install git-flow. После установки git-flow необходимо выполнить команду git flow init. Git-flow является оберткой для Git. Команда git flow init является расширением стандартной команды git init и ничего не меняет в репозитории, кроме того, что создает ветки.

На этой схеме показан принцип того, как работает git-flow (рис. 14):



Рисунок 14 – Базовое ветвление

Вместо использования одной ветки master, в этой модели используется две ветки для записи истории проекта. В ветке master хранится официальная история релиза, а ветка develop служит в качестве интеграционной ветки для новых функций. Также, удобно тегировать все коммиты в ветке master номером версии.



Рисунок 15 – Продвинутое ветвление

Каждая новая функциональность должна разрабатываться в отдельной ветке, которую можно отправлять в центральный репозиторий для создания резервной копии/для совместной работы команды (рис. 15). Ветки функций создаются не на основе master, a на основе develop. Когда работа над новой функциональностью завершена, она вливается назад в develop. Новый код не должен отправляться напрямую в master.



Рисунок 16 – Ветвление на реальном проекте

Когда в ветку develop уже слито достаточно нового кода для релиза (или подходит установленная дата предрелиза), от ветки develop создается ветка release. Создание данной ветки означает начало следующего цикла релиза, в ходе которой новая функциональность уже не добавляется, а производится только отладка багов, создание документации и решение других задач, связанных с релизом. Когда все готово, ветка release сливается в master, и ей присваивается тег с версией. Кроме этого, она должна быть также слита обратно в ветку develop, в которой с момента создания ветки релиза могли добавляться изменения с момента создания ветки релиза.

Ветки hotfix используются для быстрого внесения исправлений в рабочую версию кода (рис. 16). Ветки hotfix очень похожи на ветки release и feature, за исключением того, что они созданы от master, а не от develop. Это единственная ветка, которая должна быть создана непосредственно от master. Как только исправление завершено, ветка hotfix должна быть объединена как с master, так и с develop (или с веткой текущего релиза), а master должен быть помечен обновленным номером версии.

Наличие специальной ветки для исправления ошибок позволяет команде решать проблемы, не прерывая остальную часть рабочего процесса и не ожидая следующего цикла подготовки к релизу. Можно говорить о ветках hotfix как об особых ветках relese, которые работают напрямую с master.

**3.3 JavaFX**

**JavaFX** — платформа на основе Java для создания приложений с насыщенным графическим интерфейсом. Может использоваться как для создания настольных приложений, запускаемых непосредственно из-под операционных систем, так и для интернет-приложений (RIA), работающих в браузерах, и для приложений на мобильных устройствах. JavaFX призвана заменить использовавшуюся ранее библиотеку Swing. Платформа JavaFX конкурирует с Microsoft Silverlight, Adobe Flash и аналогичными системами.

Начиная с версии Java 11 больше не входит в Java SE и не разрабатывается компанией Oracle (как отдельный модуль поддерживается компанией Gluon). Но Oracle будет вносить необходимые изменения до 2022 года как для части Java SE 8.

**4 АРХИТЕКТУРНЫЙ ШАБЛОН ПРОЕКТИРОВАНИЯ MVVM**

Паттерн MVVM (Model-View-ViewModel) позволяет отделить логику приложения от визуальной части (представления). Данный паттерн является архитектурным, то есть он задает общую архитектуру приложения.

Данный паттерн был представлен Джоном Госсманом в 2005 году как модификация шаблона Presentation Model и был первоначально нацелен на разработку приложений в WPF. И хотя сейчас данный паттерн вышел за пределы WPF и применяется в самых различных технологиях, в том числе при разработке под Android, iOS, тем не менее WPF является довольно показательной технологией, которая раскрывает возможности данного паттерна.

MVVM состоит из трех компонентов: модели (Model), модели представления (ViewModel) и представления (View). (рис. 17)



Рисунок 17 – Схема работы MVVM

Кроме понимания обязанностей каждого компонента, важно понимать, как они взаимодействуют друг с другом. На высоком уровне в представлении «известно о модели представления и модели представления» известно о модели, но модель не знает модель представления, а модель представления не знает об этом представлении. Таким образом, модель представления изолирует представление от модели и позволяет модели развиваться независимо от представления.

Ниже приведены преимущества использования шаблона MVVM:

1. Если реализована существующая реализация модели, которая инкапсулирует существующую бизнес-логику, она может быть сложной или рискованной для ее изменения. В этом сценарии модель представления выступает в качестве адаптера для классов модели и позволяет избежать внесения значительных изменений в код модели.
2. Разработчики могут создавать модульные тесты для модели представления и модели без использования представления. Модульные тесты для модели представления могут выполнять точно те же функциональные возможности, которые используются в представлении.
3. Пользовательский интерфейс приложения можно переконструировать, не затрагивая код, при условии, что представление полностью реализовано в XAML. Поэтому Новая версия представления должна работать с существующей моделью представления.
4. Разработчики и разработчики могут одновременно работать с компонентами в процессе разработки. Дизайнеры могут сосредоточиться на представлении, тогда как разработчики могут работать над моделью представления и компонентами модели.

Ключ к использованию MVVM эффективно полагается на понимание того, как разделить код приложения на правильные классы и как понять, как взаимодействуют классы. В следующих разделах рассматриваются обязанности каждого класса в шаблоне MVVM.

Модель описывает используемые в приложении данные. Модели могут содержать логику, непосредственно связанную этими данными, например, логику валидации свойств модели. В то же время модель не должна содержать никакой логики, связанной с отображением данных и взаимодействием с визуальными элементами управления. Нередко модель реализует интерфейсы, которые позволяют уведомлять систему об изменениях свойств модели. Благодаря этому облегчается привязка к представлению, хотя опять же прямое взаимодействие между моделью и представлением отсутствует.

Модель представления реализует свойства и команды, к которым может быть привязано представление данных, и уведомляет представление о любых изменениях состояния с помощью событий уведомления об изменениях. Свойства и команды, предоставляемые моделью представления, определяют функциональные возможности, предоставляемые пользовательским интерфейсом, но представление определяет, как эти функции должны отображаться. Модель представления также отвечает за координацию взаимодействия представления с любыми необходимыми классами модели. Как правило, существует связь «один ко многим» между моделью представления и классами модели. Модель представления может предоставить доступ к классам модели непосредственно в представлении, чтобы элементы управления в представлении могли привязывать данные к ним напрямую. В этом случае классы модели должны быть спроектированы для поддержки привязки данных и событий уведомления об изменении.

Каждая модель представления предоставляет данные из модели в форме, которую представление может легко использовать. Для этого модель представления иногда выполняет преобразование данных. Размещение этого преобразования данных в модели представления является хорошей идеей, поскольку она предоставляет свойства, к которым может быть привязано представление. Например, модель представления может сочетать значения двух свойств, чтобы упростить отображение представления.

Представление отвечает за определение структуры, макета и внешнего вида того, что видит пользователь на экране. Свойства представления совпадают со свойствами модели представления/модели. При этом модель представления не имеет ссылки на интерфейс представления. Изменение состояния модели представления автоматически изменяет представление, и наоборот. Для этого используется механизм связывания данных. Также характерная черта MVVM — двусторонняя коммуникация с представлением.

MVVM обязан содержать привязку данных. Привязка данных является ключевой функцией, которая отличает MVVM от других шаблонов разделения пользовательского интерфейса, таких как MVC и MVP.

Для привязки данных необходимо создать представление или набор элементов пользовательского интерфейса, а затем следует выбрать какой-то другой объект, на который будут указывать привязки.

* Элементы пользовательского интерфейса в представлении привязаны к свойствам, предоставляемым ViewModel.
* Порядок, на котором построены View и ViewModel, зависит от ситуации, так как мы сначала рассмотрели View.
* View и ViewModel создаются, а DataContext для View устанавливается в ViewModel.
* Привязки могут быть привязками данных OneWay или TwoWay для передачи данных назад и вперед между View и ViewModel.

**5 ШАБЛОНЫ ПРОЕКТИРОВАНИЯ**

Шаблон проектирования, или паттерн, в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования, в рамках некоторого часто возникающего контекста.

Шаблоны бывают следующих трех видов:

1. Порождающие;
2. Структурные;
3. Поведенческие.

Если говорить простыми словами, то это шаблоны, которые предназначены для создания экземпляра объекта или группы связанных объектов

Порождающие шаблоны — шаблоны проектирования, которые имеют дело с процессом создания объектов. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять наследуемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту. Эти шаблоны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

Порождающие шаблоны инкапсулируют знания о конкретных классах, которые применяются в системе, то есть скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, — это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие шаблоны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда. Можно собрать систему из «готовых» объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

Фабричный метод (Factory Method) - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам. Фабричный метод стоит применять в таких случаях:

* Когда заранее неизвестно, объекты каких типов необходимо создавать;
* Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать;
* Когда создание новых объектов необходимо делегировать из базового класса классам наследникам.

Паттерн "Абстрактная фабрика" (Abstract Factory) предоставляет интерфейс для создания семейств взаимосвязанных объектов с определенными интерфейсами без указания конкретных типов данных объектов. Данный паттерн стоит применять в таких случаях:

* Когда система не должна зависеть от способа создания и компоновки новых объектов;
* Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными.

Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа. Этот паттерн стоит использовать когда необходимо, чтобы для класса существовал только один экземпляр. Синглтон позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.

Поведенческие шаблоны — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов.

В поведенческих шаблонах уровня класса используется наследование, чтобы определить поведение для различных классов. В поведенческих шаблонах уровня объекта используется композиция. Некоторые из них описывают, как с помощью кооперации несколько равноправных объектов работают над заданием, которое они не могут выполнить по отдельности. Здесь важно то, как объекты получают информацию о существовании друг друга. Объекты-коллеги могут хранить ссылки друг на друга, но это усиливает степень связанности системы. При высокой связанности каждому объекту пришлось бы иметь информацию обо всех остальных. Некоторые из шаблонов решают эту проблему.

Паттерн Стратегия (Strategy) представляет шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм. Этот паттерн стоит использовать в таких случаях:

* Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять;
* Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий;
* Когда необходимо менять поведение объектов на стадии выполнения программы;
* Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации.

Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

Данный паттерн еще называют Publisher-Subscriber (издатель-подписчик), поскольку отношения издателя и подписчиков характеризуют действие данного паттерна: подписчики подписываются email-рассылку определенного сайта. Сайт-издатель с помощью email-рассылки уведомляет всех подписчиков о изменениях. А подписчики получают изменения и производят определенные действия: могут зайти на сайт, могут проигнорировать уведомления и т.д.

Паттерн стоит использовать в таких случаях:

* Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях;
* Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга;
* Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться.

Структурные паттерны рассматривают вопросы о компоновке системы на основе классов и объектов. При этом могут использоваться следующие механизмы:

* Наследование, когда базовый класс определяет интерфейс, а подклассы - реализацию. Структуры на основе наследования получаются статичными;
* Композиция, когда структуры строятся путем объединения объектов некоторых классов. Композиция позволяет получать структуры, которые можно изменять во время выполнения.

Фасад (Facade) представляет шаблон проектирования, который позволяет скрыть сложность системы с помощью предоставления упрощенного интерфейса для взаимодействия с ней. Фасад стоит использовать в этих случаях:

* Когда имеется сложная система, и необходимо упростить с ней работу. Фасад позволит определить одну точку взаимодействия между клиентом и системой;
* Когда надо уменьшить количество зависимостей между клиентом и сложной системой. Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо;
* Когда нужно определить подсистемы компонентов в сложной системе. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимодействие между ними и повысить их независимость друг от друга.

Декоратор (Decorator) представляет структурный шаблон проектирования, который позволяет динамически подключать к объекту дополнительную функциональность.

Для определения нового функционала в классах нередко используется наследование. Декораторы же предоставляет наследованию более гибкую альтернативу, поскольку позволяют динамически в процессе выполнения определять новые возможности у объектов. Стоит использовать когда надо динамически добавлять к объекту новые функциональные возможности. При этом данные возможности могут быть сняты с объекта

Применение наследования неприемлимо, если нам надо определить множество различных функциональностей и для каждой функциональности наследовать отдельный класс, то структура классов может очень сильно разрастись. Еще больше она может разрастись, если нам необходимо создать классы, реализующие все возможные сочетания добавляемых функциональностей.

**ЗАКЛЮЧЕНИЕ**

В результате выполнения курсовой работы были изучены материалы, связанные с разработкой Java-приложений. В результате курсовой работы было написано приложение, позволяющее переписываться. Была углублено изучена и применена на практике объектно-ориентированная технология программирования и архитектурный паттерн MVVM, который базировался на методе связывания данных «Boxing». Код программы написан на языке Java.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] ERUD.BY [Электронный ресурс]. – Режим доступа: http://erud.by/object\_orient\_program

[2] HABR [Электронный ресурс]. – Режим доступа: https://habr.com/

[3] MEDIUM [Электронный ресурс]. – Режим доступа: https://medium.com/

# ПРИЛОЖЕНИЕ А

Листинг кода

package sample;

import java.io.IOException;

import java.net.URL;

import java.util.ResourceBundle;

import javafx.fxml.FXML;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.PasswordField;

import javafx.scene.control.TextField;

import javafx.stage.Stage;

public class Controller {

 @FXML

 private ResourceBundle resources;

 @FXML

 private URL location;

 @FXML

 private Button login\_in;

 @FXML

 private Button forgot\_password;

 @FXML

 private TextField login;

 @FXML

 private PasswordField password;

 @FXML

 private Button first\_time;

 public Controller() {

 }

private static String Name;

 @FXML

 void initialize() {

 Name=login.getText();

 login\_in.setOnAction(actionEvent -> {

 if(login.getText().equals("login")&&password.getText().equals("123")){

 login\_in.getScene().getWindow().hide();

 FXMLLoader loader =new FXMLLoader();

 loader.setLocation(getClass().getResource("../Fxml/Chat.fxml"));

 try {

 loader.load();

 } catch (IOException e) {

 e.printStackTrace();

 }

 Parent root=loader.getRoot();

 Stage stage=new Stage();

 stage.setScene(new Scene(root));

 stage.setTitle

 stage.showAndWait();

 }

 });

 first\_time.setOnAction(event ->{

first\_time.getScene().getWindow().hide();

 FXMLLoader loader =new FXMLLoader();

 loader.setLocation(getClass().getResource("../Fxml/reg.fxml"));

 try {

 loader.load();

 } catch (IOException e) {

 e.printStackTrace();

 }

 Parent root=loader.getRoot();

 Stage stage=new Stage();

 stage.setScene(new Scene(root));

 stage.setTitle("mes ");

stage.showAndWait();

 });

 }

}

package sample;

import java.net.\*;

import java.io.\*;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.net.URL;

import java.util.ResourceBundle;

import javafx.fxml.FXML;

import javafx.scene.control.Button;

import javafx.scene.control.TextArea;

import javafx.scene.control.TextField;

public class Controller\_chat {

 @FXML

 private ResourceBundle resources;

 @FXML

 private URL location;

 @FXML

 Button Send\_Msg;

 @FXML

 private TextField Msg;

 @FXML

 private TextField Name;

 @FXML

 private TextArea text;

 @FXML

 void initialize() {

 new ClientSomthing("localhost", 8079);

 }

 class ClientSomthing {

 private Socket socket;

 private BufferedReader in;

 private BufferedWriter out;

 private BufferedReader inputUser;

 private String addr;

 private int port;

 private String nickname;

 Date date ;

 public ClientSomthing(String addr, int port) {

 date = new Date();

 this.addr = addr;

 this.port = port;

 try {

 this.socket = new Socket(addr, port);

 } catch (IOException e) {

 System.err.println("Socket failed");

 }

 try {

 inputUser = new BufferedReader(new InputStreamReader(System.in));

 in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

 out = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));

 this.pressNickname();

 new ReadMsg().start();

 new WriteMsg().start();

 } catch (IOException e) {

 }

 }

 private void pressNickname() {

 System.out.print("Press your nick: ");

 try {

 nickname = inputUser.readLine();

 } catch (IOException ignored) {

 }

 }

 private class ReadMsg extends Thread {

 @Override

 public void run() {

 String str;

 String name;

 try {

 while (true) {

 name=in.readLine();

 str = in.readLine();

 if(name.equals(nickname)) {

 System.out.println(str);

 text.appendText(str + "\n");

 }

 }

 } catch (IOException e) {

 }

 }

 }

 public class WriteMsg extends Thread {

 @Override

 public void run() {

 Date dateNow = new Date();

 SimpleDateFormat formatForDateNow = new SimpleDateFormat(" yyyy.MM.dd hh:mm:ss ");

 while (true) {

 Send\_Msg.setOnAction(actionEvent -> {

 try {

 out.write(Name.getText()+"\n");

 out.flush();

 out.write( nickname + ": " + Msg.getText()+" time : "+formatForDateNow.format(dateNow) + "\n");

 text.appendText(nickname + ": " + Msg.getText()+" time : "+formatForDateNow.format(dateNow) + "\n");

 out.flush();

 System.out.println(nickname + ": " + Msg.getText()+" time : "+formatForDateNow.format(dateNow) + "\n");

 } catch (IOException e) {

 }

 });

 }

 }

 }

 }

}

package sample;

import java.io.IOException;

import java.net.URL;

import java.util.ResourceBundle;

import javafx.fxml.FXML;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.PasswordField;

import javafx.scene.control.RadioButton;

import javafx.scene.control.TextField;

import javafx.stage.Stage;

public class Controller\_reg {

 @FXML

 private ResourceBundle resources;

 @FXML

 private URL location;

 @FXML

 private Button finish;

 @FXML

 private PasswordField Create\_password;

 @FXML

 private RadioButton female;

 @FXML

 private RadioButton male;

 @FXML

 private TextField Create\_login;

 @FXML

 private TextField email;

 @FXML

 private Button Back;

 @FXML

 void initialize() {

 Back.setOnAction(event ->{

 Back.getScene().getWindow().hide();

 FXMLLoader loader =new FXMLLoader();

 loader.setLocation(getClass().getResource("../Fxml/sample.fxml"));

 try {

 loader.load();

 } catch (IOException e) {

 e.printStackTrace();

 }

 Parent root=loader.getRoot();

 Stage stage=new Stage();

 stage.setScene(new Scene(root));

 stage.setTitle(" mes");

 stage.show();

 });

 }

}