

Технически университет - Варна

Факултет: ФИТА

Катедра: СИТ

Специалност: СИТ

Дисциплина:

Обектно ориентирано програмиране проект – 2 част

Проект на тема: Библиотека

Изготвен от:

Алекс Михайлов Орозов - 21621586

Богомил Георгиев Донков – 21621611

**Структура на документацията**

**Глава 1. Увод…………………………………………………**

1.1.Пълно описание на проекта (Условие)…………………………………

**Глава 2. Анализ на проблема………………………………**

2.1.Функционални изисквания……………………………………………….

2.2.Проучване и избор на технологии за реализация съвместими с Java и JavaFx……………………………………………………………………………..

2.3.Структура на проекта (Анализ и избор на архитектурен модел) …………………………………………………………………

2.4.Дефиниция на модулите на системата……………………………………

**Глава 3. Проектиране на системата……………………..**

3.1.Проектиране на отделните модули (бази данни, бизнес логика, презентационен слой)…………………………………………………………….

3.2.UML(Унифициран език за моделиране) диаграми………………………

3.3.Концептуален модел на базата от данни………………………………

**Глава 4. Реализация на системата……………….……..**

4.1.Реализация на базата от данни (PostGresSQL) – Релационна схема и описание на таблиците………………………………………………………..

4.2.Реализация на слоя за работа с базата данни (Hibernate и DAO (Data Accessable Object))…………………………………………………….

4.3.Реализация на бизнес логика (Services)………………………………….

4.4.Реализация на графичен интерфейс (JavaFX)…………………

4.5.Реализация на модул за регистриране на събития в системата (Logback и SLF4J)………………………………………………………………………

**Глава 5. Тестови резултати…………………………...**

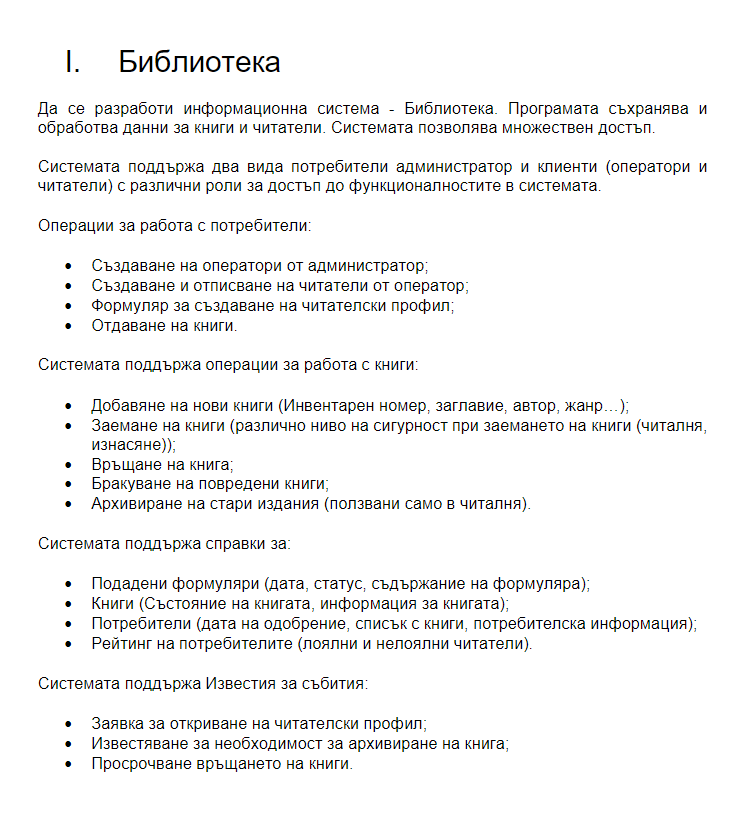
5.1.JUnit тестове…………………………………………………………….

5.2.Функционални тестове…………………………………………………..

5.3.Интеграционни тестове………………………………………………….

**Глава 1. Увод**

* 1. Пълно описание на проекта (Условие)



**Глава 2. Анализ на проблема**

2.1. Функционални изисквания

Системата трябва да поддържа като потребители администратори, оператори и читатели.

Администраторите са отговорни за регистрирането на нови книги в библиотеката, създаване и премахване на оператори, преглед на всички съществуващи потребители в системата, да архивира книги по негова преценка и да премахва книги със щети по тях.

Операторите трябва да бъдат способни да откриват и закриват читателски профили, да внасят книги в определено от тях количество, да преглеждат всички съществуващи потребители в системата, да отдават книги за различна употреба, да отговарят за връщане на книгите от даден читателски формуляр.

Читателите нямат никаква отговорност в системата. Те се явяват на гише при работник(оператора) и просто заявяват какви книги желаят да върнат или вземат. Операторът извършва цялата работа за тях.

Системата поддържа методи за известяване – изпращане на имейл на потребителка електронна поща.

2.2. Проучване и избор на технологии за реализация съвместими с Java и JavaFX

В проектът са включени множество външни библиотеки, които предоставят вече съществуващи функционалности или такива, които биха направили работата по проекта по-лесна, те са:

* Lombok – Lombok библиотеката за Java, която предоставя анотации, които автоматично генерират код за общи операции като гетъри, сетъри, методи за toString(), equals() и hashCode(), както и други.

Целта на Lombok е да опрости и улесни писането на по-кратък и по-четлив код.

Ето някои от основните анотации, предоставяни от Lombok и това, което те постигат:

import lombok.Getter;  
import lombok.Setter;  
  
@Getter @Setter  
public class MyClass {  
 private int myField;  
}

public class MyClass {  
 private int myField;  
  
 public int getMyField() {  
 return myField;  
 }  
  
 public void setMyField(int myField) {  
 this.myField = myField;  
 }  
}

-----------------------------------------------------------------------------

import lombok.ToString;  
  
@ToString  
public class MyClass {  
 private int myField;  
}

public class MyClass {  
 private int myField;  
  
 @Override  
 public String toString() {  
 return "MyClass{" +  
 "myField=" + myField +  
 '}';  
 }  
}

-----------------------------------------------------------------------------

import lombok.EqualsAndHashCode;  
  
@EqualsAndHashCode  
public class MyClass {  
 private int myField;  
}

public class MyClass {  
 private int myField;  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 MyClass myClass = (MyClass) o;  
 return myField == myClass.myField;  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(myField);  
 }  
}

-----------------------------------------------------------------------------

import lombok.NoArgsConstructor;  
  
@NoArgsConstructor  
public class MyClass {  
 private int myField;  
}

public class MyClass {  
 private int myField;  
  
 public MyClass() {  
 }  
}

-----------------------------------------------------------------------------

import lombok.RequiredArgsConstructor;  
  
@RequiredArgsConstructor  
public class MyClass {  
 private final int myField;  
}

public class MyClass {  
 private final int myField;  
  
 public MyClass(int myField) {  
 this.myField = myField;  
 }  
}

-----------------------------------------------------------------------------

import lombok.AllArgsConstructor;  
  
@AllArgsConstructor  
public class MyClass {  
 private int field1;  
 private String field2;  
}

public class MyClass {  
 private int field1;  
 private String field2;  
  
 public MyClass(int field1, String field2) {  
 this.field1 = field1;  
 this.field2 = field2;  
 }  
}

-----------------------------------------------------------------------------

import lombok.Data;  
  
@Data  
public class MyClass {  
 private int myField;  
}

import java.util.Objects;  
  
public class MyClass {  
 private int myField;  
  
 public int getMyField() {  
 return myField;  
 }  
  
 public void setMyField(int myField) {  
 this.myField = myField;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 MyClass myClass = (MyClass) o;  
 return myField == myClass.myField;  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(myField);  
 }  
  
 @Override  
 public String toString() {  
 return "MyClass{" +  
 "myField=" + myField +  
 '}';  
 }  
}

-----------------------------------------------------------------------------

import lombok.Builder;  
  
@Builder  
public class MyClass {  
 private int field1;  
 private String field2;  
}

public class MyClass {  
 private int field1;  
 private String field2;  
  
 private MyClass(Builder builder) {  
 this.field1 = builder.field1;  
 this.field2 = builder.field2;  
 }  
  
 public int getField1() {  
 return field1;  
 }  
  
 public String getField2() {  
 return field2;  
 }  
  
 public static Builder builder() {  
 return new Builder();  
 }  
  
 public static class Builder {  
 private int field1;  
 private String field2;  
  
 private Builder() {  
 }  
  
 public Builder field1(int field1) {  
 this.field1 = field1;  
 return this;  
 }  
  
 public Builder field2(String field2) {  
 this.field2 = field2;  
 return this;  
 }  
  
 public MyClass build() {  
 return new MyClass(this);  
 }  
 }  
}

-----------------------------------------------------------------------------

@Slf4j  
public class MyClass {  
 public void exampleMethod() {  
 *log*.info("Logging with Lombok!");  
 }  
}

import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
public class MyClass {  
 private static final Logger *log* = LoggerFactory.*getLogger*(MyClass.class);  
  
 public void exampleMethod() {  
 *log*.info("Logging without Lombok!");  
 }  
}

-----------------------------------------------------------------------------

* **Hibernate** – org.hibernate:hibernate-core е основната библиотека за обектно-релационно съпоставяне (ORM) в Java еко системата, която предоставя инструменти за работа с релационни бази данни чрез обекти. Hibernate улеснява извличането, обновяването и вмъкването на данни в релационни бази данни, като предоставя високо ниво на абстракция на SQL.

Ключови функционалности в Hibernate:

1. **ORM(Object-Relation Mapping):** Hibernate позволява работата с обекти в Java, които съответстват на записи в релационни бази данни. Това осигурява по-лесен и интуитивен начин за манипулиране на данни.
2. **Автоматично генериране на SQL:** Hibernate автоматично генерира SQL заявки за създаване, обновяване, извличане и изтриване на данни от базата данни, което намалява необходимостта от писане на ръчен SQL код.
3. **Кеширане:** Hibernate предоставя възможности за кеширане на данни, което подобрява производителността при честите заявки.
4. **Transaction Management:** Hibernate управлява транзакциите в рамките на приложението, което поддържа консистентността на данните.
5. **Хибриден модел на връзката между таблиците:** Hibernate поддържа както единична, така и множествена връзка между таблиците, като предоставя гъвкавост и лесна настройка на връзките.
6. **Поддръжка на наследяване:** Hibernate поддържа различни стратегии за наследяване, които позволяват лесно създаване на йерархии от класове.
7. **Поддръжка на HQL (Hibernate Query Language):** HQL предоставя алтернативен начин за извършване на заявки върху обекти, които се използват вместо SQL.
8. **Разширяемост:** Hibernate е високо разширяем и дава възможност за дефиниране на собствени стратегии и обекти.

С implementation ('org.hibernate:hibernate-core:6.2.9.Final') в проекта се добавя конкретната версия на Hibernate като зависимост. Този артефакт съдържа основния код на Hibernate, необходим за изпълнението на ORM функционалността и други ключови възможности, споменати по-горе. Този артефакт е необходим, за да може приложението ви да използва Hibernate за връзка с релационна база данни.

Примери с анотации:

* @Entity - Употреба: Анотацията се поставя над клас, за да му каже Hibernate, че този клас трябва да бъде представен като таблица в базата данни.
* @Id – Употреба: Посочва полето като основен ключ на таблицата.

import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
  
@Entity  
public class Book {  
 @Id  
 private Long id;  
 private String title;  
 private String author;  
}

* @Table – Употреба: С тази анотация може да се указва името и други свойства на таблицата, към която класът ще се съпостави.
* import jakarta.persistence.Entity;  
  import jakarta.persistence.Id;  
  import jakarta.persistence.Table;  
    
  @Entity  
  @Table(name = "library\_books")  
  public class Book {  
   @Id  
   private Long id;  
   private String title;  
   private String author;  
  }
* @GeneratedValue – Употреба: Анотацията дефинира начина, по който Hibernate генерира стойностите за основния ключ (ID).

import jakarta.persistence.\*;  
  
@Entity  
@Table(name = "library\_books")  
public class Book {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 private String title;  
 private String author;  
}

* @Column – Употреба: Позволява насрочването на свойства на колоната в базата данни.

import jakarta.persistence.\*;  
  
@Entity  
@Table(name = "library\_books")  
public class Book {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 @Column(name = "title")  
 private String title;  
 @Column(name = "book\_author")  
 private String author;  
}

Тези са само някои от основните анотации, които можете да използвате с Hibernate. Зависимо от конкретните изисквания на вашето приложение, може да бъдат използвани и други анотации.

* **Bcrypt** - "bcrypt" библиотека за Java. Това е библиотека за хеширане на пароли, която използва bcrypt алгоритъма. Bcrypt е криптографски сигурен алгоритъм за хеширане на пароли, който е проектиран да бъде забавен и използва сол (salt), което го прави устойчив на атаки срещу радуга (rainbow table attacks).

Ето кратко описание на някои от ключовите характеристики на тази библиотека:

1. **Безопастност** - Bcrypt е считан за един от най-сигурните методи за хеширане на пароли в настоящия момент. Той използва сол и включва параметър за работа (work factor), който определя колко време отнема хеширането и го прави изключително трудно за атаки.
2. **Salt(Сол)** - Bcrypt автоматично включва сол във всеки хеш, който създава. Това предотвратява успешни атаки срещу радуга, където злоумишленникът зареди предварително изчислени хешове за голям списък от обичайни пароли.
3. **Work Factor** - Параметърът за работа (work factor) е настройка, която определя колко време се изисква за създаване на хеш. Този параметър може да бъде настроен, за да се усложни атаките срещу пароли чрез използването на brute-force атаки.
4. **Проста интеграция** - Библиотеката предоставя лесен за използване API за хеширане на пароли, което я прави подходяща за интеграция в приложения.
5. **Активно развитие** - Ако версията, която използвате, е 0.10.2, това може да съобщи за активно развитие и поддръжка на библиотеката.

Пример на използване на библиотеката може да изглежда по следния начин:

import at.favre.lib.crypto.bcrypt.BCrypt;  
  
public class PasswordHashingExample {  
 public static void main(String[] args) {  
 *// Генериране на сол* String salt = BCrypt.*withDefaults*().hashToString(12, "userPassword".toCharArray());  
  
 *// Хеширане на парола със създадената сол* String hashedPassword = BCrypt.*withDefaults*().hashToString(12, "userPassword".toCharArray());  
  
 *// Проверка на парола* boolean isPasswordCorrect = BCrypt.*verifyer*().verify("userPassword".toCharArray(), hashedPassword).verified;  
  
 System.*out*.println("Salt: " + salt);  
 System.*out*.println("Hashed Password: " + hashedPassword);  
 System.*out*.println("Is Password Correct: " + isPasswordCorrect);  
 }  
}

* **Javax.mail** - Библиотеката javax.mail предоставя Java API за работа с електронна поща (email). Тя е стандартна библиотека в рамките на Java EE (Enterprise Edition) и предоставя функционалности за изпращане, получаване и обработка на електронни съобщения.

Ето някои от ключовите функционалности и ползи на библиотеката:

1. **Създаване на Електронни Съобщения (Emails):** javax.mail позволява създаването на електронни съобщения с различни свойства като тема, получатели, изпращач, прикачени файлове и текстово съдържание.
2. **Изпращане на Електронни Съобщения:** Библиотеката предоставя функционалности за изпращане на съобщения по различни протоколи като SMTP (Simple Mail Transfer Protocol).
3. **Получаване на Електронни Съобщения:** javax.mail поддържа получаването на електронни съобщения от пощенски сървъри по протоколи като POP3 (Post Office Protocol) и IMAP (Internet Message Access Protocol).
4. **Обработка на Електронни Съобщения:** Библиотеката предоставя функционалности за обработка на съдържанието на електронни съобщения, включително работа с прикачени файлове, извличане на текст и работа с хедърите на съобщението.
5. **Конфигуриране и Управление на Сървъри:** javax.mail позволява конфигуриране и управление на настройките на пощенски сървъри, където можете да зададете настройките за изпращане и получаване на съобщения.

Библиотеката javax.mail е изключително полезна за Java приложения, които имат нужда от взаимодействие с електронни съобщения, като например имейл уведомления, обработка на имейли и други сценарии, свързани с електронна поща.

Пример на употреба:

import javax.mail.\*;  
import javax.mail.internet.InternetAddress;  
import javax.mail.internet.MimeMessage;  
import java.util.Properties;  
  
public class SimpleMailSender {  
 private final String username;  
 private final String password;  
 private final Properties properties;  
  
 public SimpleMailSender(String username, String password) {  
 this.username = username;  
 this.password = password;  
  
 *// Настройка на свойствата за използване на SMTP* properties = new Properties();  
 properties.put("mail.smtp.auth", "true");  
 properties.put("mail.smtp.starttls.enable", "true");  
 properties.put("mail.smtp.host", "smtp.example.com"); *// Подменете с реалния хост* properties.put("mail.smtp.port", "587"); *// Подменете с реалния порт* }  
  
 public void sendEmail(String to, String subject, String content) throws MessagingException {  
 Session session = Session.*getInstance*(properties, new Authenticator() {  
 protected PasswordAuthentication getPasswordAuthentication() {  
 return new PasswordAuthentication(username, password);  
 }  
 });  
 Message message = new MimeMessage(session);  
 message.setFrom(new InternetAddress(username));  
 message.setRecipients(Message.RecipientType.*TO*, InternetAddress.*parse*(to));  
 message.setSubject(subject);  
 message.setText(content);  
  
 Transport.*send*(message);  
 System.*out*.println("Email sent successfully.");  
 }  
  
 public static void main(String[] args) {  
 *// Подменете със свои реални данни* SimpleMailSender mailSender = new SimpleMailSender("your\_email@example.com", "your\_password");  
  
 try {  
 mailSender.sendEmail("recipient@example.com", "Тестово съобщение", "Здравей, това е тестово съобщение!");  
 } catch (MessagingException e) {  
 e.printStackTrace();  
 }  
 }  
}

* **Google Guava API -** Guava е библиотека на Java, предоставена от Google, която предлага много полезни класове и методи, които допълват стандартната библиотека на Java. В следващите редове ще разгледаме някои от ключовите характеристики и функции на Guava:

1. **Колекции (com.google.common.collect):**

Guava предоставя много допълнителни класове за работа с колекции, които са по-удобни и мощни от тези в стандартната библиотека. Например, ImmutableList, ImmutableSet, Multimap и други.

import com.google.common.collect.ImmutableList;  
import com.google.common.collect.ImmutableSet;  
  
import java.util.List;  
import java.util.Set;  
  
public class ShowCaseClass {  
 List<String> immutableList = ImmutableList.*of*("one", "two", "three");  
 Set<Integer> immutableSet = ImmutableSet.*of*(1, 2, 3);  
  
}

1. **Работа със символни низове (com.google.common.base):**

Guava предоставя удобни методи за работа със символни низове, като например методи за проверка на null стойности и извличане на стойности.

import com.google.common.base.MoreObjects;  
  
public class ShowCaseClass {  
 String result = MoreObjects.*firstNonNull*(input, "default");  
}

1. **Функционално програмиране (com.google.common.base):**

Guava предоставя инструменти за функционално програмиране, като Function, Predicate, Supplier и други.

import java.util.function.Function;  
  
public class ShowCaseClass {  
 Function<Integer, String> squareFunction = input -> "Square: " + (input \* input);  
}

1. **Функционални операции върху колекции (Functional Operations on Collections):** Guava предоставя методи за филтриране, трансформиране и обединяване на колекции.

import com.google.common.collect.Lists;  
  
import java.util.List;  
import java.util.stream.Collectors;  
  
public class ExampleClass {  
 List<String> filteredList = Lists.*newArrayList*("apple", "banana", "orange")  
 .stream()  
 .filter(fruit -> fruit.startsWith("a"))  
 .collect(Collectors.*toList*());  
}

1. **Задачи за изпълнение (Concurrency):** Guava предоставя инструменти за управление на конкурентността.

import com.google.common.util.concurrent.ListenableFuture;  
import com.google.common.util.concurrent.ListeningExecutorService;  
import com.google.common.util.concurrent.MoreExecutors;  
  
import java.util.concurrent.Executors;  
  
public class ExampleClass {  
 ListeningExecutorService executorService = MoreExecutors.*listeningDecorator*(Executors.*newFixedThreadPool*(10));  
 ListenableFuture<String> future = executorService.submit(() -> "Hello, Guava!");  
}

1. **Разширени структури от данни (Additional Data Structures):** Guava включва структури като Multimap, Table и други, които предоставят по-богати възможности от стандартните Java колекции.

import com.google.common.collect.ArrayListMultimap;  
import com.google.common.collect.Multimap;  
  
public class ExampleClass {  
 public static void main(String[] args) {  
 Multimap<String, String> multimap = ArrayListMultimap.*create*();  
 multimap.put("Fruits", "Apple");  
 multimap.put("Fruits", "Banana");   
 }  
}

* PostGreSQL - Библиотеката org.postgresql:postgresql представлява JDBC (Java Database Connectivity) драйвер за PostgreSQL, една от популярните релационни бази от данни. JDBC драйверите се използват в Java приложенията, за да установят връзка и да взаимодействат с релационни бази данни, като например PostgreSQL.

Ето някои ключови аспекти на тази библиотека:

1. **Установяване на Връзка (Establishing Connection):** JDBC драйверите позволяват на Java приложенията да се свързват с бази от данни. Този драйвер специално е предназначен за връзка с PostgreSQL бази от данни.

import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;  
  
public class ExampleClass {  
 public static void main(String[] args) throws SQLException {  
 String jdbcUrl = "jdbc:postgresql://localhost:5432/mydatabase";  
 String username = "myuser";  
 String password = "mypassword";  
  
 Connection connection = DriverManager.*getConnection*(jdbcUrl, username, password);  
 }  
}

1. **Изпълнение на Заявки (Executing Queries):** След като е установена връзка, драйверът позволява на приложението да изпраща SQL заявки към базата данни.

import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.sql.Statement;  
  
public class ExampleClass {  
 public static void main(String[] args) throws SQLException {  
 String sql = "SELECT \* FROM mytable";  
 try (Statement statement = connection.createStatement();  
 ResultSet resultSet = statement.executeQuery(sql)) {  
 *// Обработка на резултатите* }  
 }  
}

1. **Транзакции и Управление на Базата Данни (Transactions and Database Management):** JDBC драйверът позволява управление на транзакции и изпълнение на операции за управление на базата данни.

import java.sql.SQLException;  
  
public class ExampleClass {  
 public static void main(String[] args) {  
 try {  
 connection.setAutoCommit(false);  
 *// Изпълнение на операции* connection.commit();  
 } catch (SQLException e) {  
 connection.rollback();  
 *// Обработка на грешката* } finally {  
 connection.setAutoCommit(true);  
 }  
  
 }  
}

1. **Използване на Prepared Statements:** За подобряване на ефективността и безопасността, драйверът поддържа използването на подготвени заявки.

import java.sql.PreparedStatement;  
import java.sql.SQLException;  
  
public class ExampleClass {  
 public static void main(String[] args) throws SQLException {  
 String sql = "INSERT INTO mytable (column1, column2) VALUES (?, ?)";  
 try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {  
 preparedStatement.setString(1, "value1");  
 preparedStatement.setInt(2, 42);  
 preparedStatement.executeUpdate();  
 }  
 }  
}

Тази библиотека предоставя основните средства, необходими за свързване и взаимодействие с PostgreSQL база данни в Java приложения. Използването на JDBC драйвери е стандартен начин за комуникация между Java и релационни бази данни.

* JavaFX - Библиотеката **org.controlsfx:controlsfx** е библиотека за JavaFX, която предоставя допълнителни и напълно настраиваеми контроли и компоненти за създаване на графични потребителски интерфейси (GUI). Тя е предназначена да разшири стандартните възможности на JavaFX, като предоставя допълнителни контроли и инструменти за създаване на по-богати и сложни потребителски интерфейси.

Ето някои от възможностите и компонентите, предоставени от ControlsFX:

1. **Диалогови Прозорци (Dialogs):**

Позволява лесно създаване на диалогови прозорци с различни стилове и ефекти. Предоставя готови диалогови прозорци за потвърждение, предупреждение, информация и др.

import org.controlsfx.dialog.Dialogs;  
  
public class ExampleClass {  
 public static void main(String[] args) {  
  
 Dialogs.create()  
 .title("Information Dialog")  
 .masthead("Look, an Information Dialog")  
 .message("I have a great message for you!")  
 .showInformation();  
 }  
}

1. **Календар (CalendarPicker):**

Позволява избор на дата от календар с интуитивен и удобен начин.

import org.controlsfx.control.CalendarPicker;  
  
public class ExampleClass {  
 CalendarPicker calendarPicker = new CalendarPicker();  
}

1. **Autocomplete TextField:**

Поле за текст, което предлага автоматично допълване на въведените данни.

import org.controlsfx.control.textfield.AutoCompletionTextFieldBinding;  
import org.controlsfx.control.textfield.TextFields;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class ExampleClass {  
 List<String> data = Arrays.*asList*("Java", "JavaScript", "Python", "C++");  
 TextFields.bindAutoCompletion(textField, data);  
}

1. **Task Progress View:**

Контрол за визуализация на прогреса на фонови задачи.

import org.controlsfx.control.TaskProgressView;  
  
public class ExampleClass {  
  
 TaskProgressView<Task<?>> taskProgressView = new TaskProgressView<>();  
taskProgressView.getTasks().addAll(task1, task2, task3);  
  
}

1. **Master-Detail View:**

Предоставя контрол за лесно създаване на интерфейси с master-detail структура.

import org.controlsfx.control.MasterDetailPane;  
  
  
public class ExampleClass {  
 public static void main(String[] args) {  
 MasterDetailPane masterDetailPane = new MasterDetailPane();  
 masterDetailPane.setMasterNode(masterNode);  
 masterDetailPane.setDetailNode(detailNode);  
 }  
}

1. **CheckComboBox:**

Позволява на потребителя да избира от списък със селекция на отметки.

import javafx.collections.FXCollections;  
import org.controlsfx.control.CheckComboBox;  
  
import java.util.Arrays;  
import java.util.List;  
  
  
public class ExampleClass {  
 public static void main(String[] args) {  
 List<String> items = Arrays.*asList*("Option 1", "Option 2", "Option 3");  
 CheckComboBox<String> checkComboBox = new CheckComboBox<>(FXCollections.*observableArrayList*(items));  
 }  
}

1. **InfoOverlay:**

Контрол за показване на информация върху друг контрол.

import org.controlsfx.control.InfoOverlay;  
  
import javax.swing.text.html.ImageView;  
  
public class ExampleClass {  
 public static void main(String[] args) {  
  
 InfoOverlay infoOverlay = new InfoOverlay();  
 infoOverlay.setGraphic(new ImageView(image));  
 infoOverlay.setText("This is additional information");  
 }  
}

Това са само някои от примерите за използване на компоненти от библиотеката ControlsFX. В реалния код на вашето приложение бихте добавили необходимите импорти и интегрирали тези компоненти във вашия потребителски интерфейс.

* **SLF4J (Simple Logging Facade for Java) –** SLF4J е прост интерфейс за логиране в Java, който предоставя общ API за различни системи за логиране. Той цели да улесни прехода между различни системи за логиране и да предостави гъвкавост при избора на конкретна имплементация.
* **Logback -** Logback е конкретна имплементация на SLF4J API и предоставя мощен и ефективен механизъм за логиране. Той включва две основни компонента - logback-classic и logback-core.
* **logback-classic:** Този модул предоставя статичен бекенд за логиране, което използва SLF4J API. Той е наследник на популярния log4j проект и предоставя подобен API за конфигуриране и употреба.
* **logback-core:** Този модул предоставя ядрена функционалност за Logback, като асинхронно логиране и филтриране на съобщения.

Функционалности:

* **Лесен за използване API:** Предоставя прост и лесен за използване API за логиране в Java приложения.
* **Бърз и ефективен:** Logback е известен със своята висока производителност и ефективност.
* **Гъвкавост:** Позволява конфигуриране на различни аспекти на логирането чрез конфигурационни файлове.
* **Поддръжка на различни нива на логиране:** Поддържа различни нива на логиране, като DEBUG, INFO, WARN, ERROR и др.
* **Способност за асинхронно логиране:** Предоставя възможност за асинхронно логиране, което може да подобри производителността.

Примерна употреба:

import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
public class MyClass {  
 private static final Logger *logger* = LoggerFactory.*getLogger*(MyClass.class);  
  
 public void exampleMethod() {  
 *logger*.info("Logging with SLF4J and Logback!");  
 }  
}

Този код демонстрира как да се използва SLF4J API за логиране в Java клас. Когато конфигурацията е направена правилно, съобщението "Logging with SLF4J and Logback!" ще бъде записано в лог файла чрез Logback.

* **JUnit 5:** JUnit е популярна библиотека за тестване в Java, предоставяща функционалност за създаване и изпълнение на тестове. Версията 5 на JUnit (JUnit Jupiter) предлага нови възможности и подобрения в сравнение с предишните версии.

Функционалности:

* **Декларативен синтаксис:** JUnit 5 предлага декларативен синтаксис за написване на тестове с анотации, като **@Test**, **@BeforeEach**, **@AfterEach** и други.
* **Rerunnable Tests:** Тестовете могат да бъдат маркирани като **@RepeatedTest** или **@ParameterizedTest** за повторно изпълнение с различни аргументи.
* **Extension Model:** Разширенията (Extensions) предоставят начин за добавяне на допълнителна функционалност към тестовете.
* **Поддръжка на Java 8+:** Използва функционалности като ламбда изрази и потоци, които са налични в Java 8 и по-нови версии.
* **Поддръжка на паралелно тестване:** Възможност за паралелно изпълнение на тестове, което подобрява ефективността на тестовите суити.

Примерен код:

import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*assertEquals*;  
  
public class MyTest {  
 @Test  
 void testAddition() {  
 *assertEquals*(4, 2 + 2, "2 + 2 should equal 4");  
 }  
}

* **TestFX -** TestFX е библиотека за функционално тестване на графични потребителски интерфейси (GUI) в JavaFX приложения. Това позволява на разработчиците да пишат тестове, които симулират действия на потребителя върху GUI и проверяват резултатите.

Примерен код:

import org.junit.jupiter.api.Test;  
import org.testfx.api.FxRobot;  
import org.testfx.framework.junit5.ApplicationTest;  
  
public class MyGuiTest extends ApplicationTest {  
 @Test  
 void testButtonClick(FxRobot robot) {  
 clickOn("#myButton");  
 *// Проверка за очакван резултат след натискане на бутон* }  
}

* **Mockito -** Mockito е библиотека за създаване и използване на мок обекти (mock objects) в Java тестове. Това позволява на разработчиците да изолират тестовете от външни зависимости и да създават контролирани среди за тестване.

Примерен код:

import org.junit.jupiter.api.Test;  
import static org.mockito.Mockito.\*;  
  
public class MyMockTest {  
 @Test  
 void testMockBehavior() {  
 *// Създаване на мок обект* MyService myServiceMock = *mock*(MyService.class);  
  
 *// Дефиниране на поведение на мок обекта  
 when*(myServiceMock.getResult()).thenReturn("Mocked Result");  
  
 *// Проверка на извикване на метод с мок обект* myServiceMock.getResult();  
  
 *// Проверка на очакван резултат  
 verify*(myServiceMock, *times*(1)).getResult();  
 }  
}

Тези библиотеки се използват в комбинация за покриване на различни аспекти на тестовете, като модулни, функционални и мок тестове. Те предоставят инструменти за написване и изпълнение на тестове с цел поддържане на качеството и надеждността на софтуера.

2.3. Структура на проекта (Анализ и избор на архитектурен модел)

**Описание на Архитектурния Модел - MVC**

**MVC (Model-View-Controller)** е архитектурен модел, който има за цел да раздели компонентите на софтуерната система на три основни части, които изпълняват различни роли в системата: Model (Модел), View (Изглед) и Controller (Контролер).

1. Model (Модел):

* **Описание:**
  + Моделът представлява данните и бизнес логиката на приложението. Той съдържа структури от данни, методи и бизнес логика, отговарящи за обработката на информацията.
  + Във вашия случай, моделът може да включва класове, представящи обекти като книги, четци, форми за заемане на книги и други.

2. View (Изглед):

* **Описание:**
  + Изгледът е отговорен за представянето на информацията на потребителя. Това включва графични елементи, форми, таблици и всичко, което потребителят ви вижда.
  + Използвайки JavaFX, вие бихте създали графичен интерфейс, който представя информацията от модела на потребителя.

3. Controller (Контролер):

* **Описание:**
  + Контролерът действа като посредник между модела и изгледа. Той приема вход от потребителя, обработва го и актуализира модела или изгледа според необходимостта.
  + В JavaFX, контролерът ще съдържа логика за обработка на събития (например бутони, текстови полета) и актуализация на модела и изгледа.

Обща Работа и Поток на Данни:

* **Взаимодействие:**
  + Контролерът получава вход от потребителя и извиква съответните методи в модела за обработка на данни.
  + Моделът може да уведомява контролера за промени чрез шаблона Observer или по друг начин.
  + Контролерът обновява изгледа, който отразява промените в модела.
* **Поток на Данни:**
  + **Изглед → Контролер → Модел:** Получаване на вход от потребителя, обработка и пренасяне на необходимите данни към модела.
  + **Модел → Контролер → Изглед:** Проактуализиране на изгледа чрез предоставяне на актуална информация от модела.

**Обосновка на Избора на MVC Архитектурен Модел**

Изборът на MVC (Model-View-Controller) архитектурен модел за проекта ви е основан на редица предимства и аспекти, които подобряват ефективността, поддръжката и разширяемостта на приложението.

1. **Разделяне на Отговорности:**

MVC разделя компонентите на проекта ви на три отделни, но взаимодействащи части - модел, изглед и контролер. Този принцип на разделяне на отговорности ви дава следните предимства:

* **Поддържане на Чист Код:**
  + Всяка част на приложението има своя четимост и лесно разбираем код, който е поддържаем и подходящ за промени.
* **Лесна Разработка на Екип:**
  + Разработчиците могат да работят паралелно върху различни части от системата без да се препъват по драските.

2. **Лесна Промяна на Изгледа:**

Един от ключовите аспекти на архитектурата MVC е възможността за промяна в графичния интерфейс (изгледа) без да се нарушава бизнес логиката в модела. Това включва:

* **Гъвкавост при Дизайна:**
  + Възможността за да променяте и подобрявате графичния интерфейс без да се засяга кодът, отговарящ за обработката на данни и бизнес логиката.

3. **Повторна Употреба на Компоненти:**

MVC позволява лесна повторна употреба на компоненти, като:

* **Модуларност:**
  + Моделът и контролерът могат да се използват повторно в различни части на приложението или дори в бъдещи проекти, което подобрява ефективността и бързината на разработката.

4. **По-Лесно Тестване:**

Чрез разделянето на системата на три отделни компонента, MVC архитектурата улеснява процеса на тестване:

* **Паралелни Тестове:**
  + Всеки компонент (модел, изглед, контролер) може да бъде тестван поотделно, което улеснява откриването и коригирането на проблеми в кода.

5. **Съвместимост с JavaFX, Hibernate и PostGreSQL:**

MVC се интегрира лесно с технологии като JavaFX, Hibernate и PostGreSQL:

* **JavaFX за Интерфейса:**
  + Използването на JavaFX за създаване на графичен интерфейс отговаря на модерните стандарти за UI разработка в Java.
* **Hibernate за ORM:**
  + Хибернейт се интегрира като модел в MVC, предоставяйки лесен начин за взаимодействие с базата от данни.
* **PostGreSQL за Базата Данни:**
  + Изборът на PostGreSQL осигурява стабилна и мощна релационна база от данни, съвместима с Hibernate.

6. **Контрол върху Потока на Данни:**

С MVC имате по-голям контрол върху потока на данни в приложението:

* **Централизиран Контрол:**
  + Контролерът централизира логиката за обработка на входящите данни и решава как да се актуализира моделът и изгледа.

Този анализ и обосновка на избора на MVC архитектурен модел би следвало да ви помогне да представите важността и смисъла зад решението за вашия проект.

Top of Form

2.4. Дефиниция на модулите на системата

**Модули и компоненти**

Нашата апликация е организирана в три основни модула: **backend**, **frontend**, и **database**. Всеки от тези модули е разделен на различни компоненти, които изпълняват специфични роли. По-долу е подробно обяснение на структурата и целите на всяка част.

1. **Backend:**

* **Services:**
  + **Описание:** Този модул съдържа бизнес логика, предоставена чрез различни услуги (services).
  + **Цел:** Осигурява високо ниво на абстракция и обединява логиката, необходима за взаимодействие с базата данни и обработка на данни от frontend.
* **Exceptions:**
  + **Описание:** Съдържа използвани изключения в програмата.
  + **Цел:** Добавя детайли и документира възможни грешки и изключения в системата.
* **Engines:**
  + **Описание:** Класове, отговарящи за логика, отделна от услугите.
  + **Цел:** Разделянето на различни аспекти на логиката на по-малки компоненти предоставя по-голяма гъвкавост и поддържа съгласуваност.

2. **Database:**

* **Entities:**
  + **Описание:** Модели на апликацията, които Hibernate използва за създаване на таблици в базата данни.
  + **Цел:** Предоставяне на структура и дефиниране на обекти, които отразяват бизнес обекти в базата данни.
* **Repositories:**
  + **Описание:** DAO (Data Accessable Object) класове, съдържащи операции за работа с моделите и базата данни.
  + **Цел:** Предоставя удобен интерфейс за достъп и манипулиране на данни в базата данни.
* **Enums:**
  + **Описание:** Колекция от константи, представляващи различните състояния на entity-тата.
  + **Цел:** Централизира и документира различните стойности, които могат да бъдат асоциирани със състоянията на моделите.

3. **Frontend:**

* **Controllers:**
  + **Описание:** Контролери, свързани със съответните FXML файлове, които управляват визуалния интерфейс на потребителя.
  + **Цел:** Изолира бизнес логиката от потребителския интерфейс и осигурява лесен начин за свързване на различни UI елементи.
* **Utils:**
  + **Описание:** Помощни класове и методи, които обслужват различни части от frontend.
  + **Цел:** Поддържа обща логика, която се използва от различни компоненти на frontend.

Този модулен и структуриран подход осигурява четимост, лесна поддръжка и разширяемост на вашата апликация. Всеки модул изпълнява специфични функции, като гарантира отделност на отговорностите и подобрява преизползваемостта на кода.

**Глава. 3 Проектиране на системата**

3.1. Проектиране на отделните модули (бази данни, бизнес логика, презентационен слой)