Kovalenko Dmytro

Dom života

Knižnica je chrám, v ktorom sa vždy rodí a uchováva duchovno.

Aplikácia je interaktívna online knižnica s možnosťou vyhľadávania podľa autora, žánru a názvu knihy. Používatelia môžu prečítať krátky popis knihy a pridávať si obľúbené do svojej zbierky. Administrátor má práva spravovať knihy, pridávať nové, meniť popisy a vymazávať nežiaduce položky. Taktiež má kontrolu nad celkovým počtom kníh, používateľov a možnosťou odstrániť používateľov z databázy.

V aplikácii je implementovaný systém hlasovania, kde používatelia môžu vyjadriť svoj názor na najlepšie knihy týždňa. Okrem toho je možné hlasovať aj za najlepšieho autora. Hlasovanie prebieha v etapách, kde každá etapa zúžuje výber, napríklad z 48 kníh na 24, 12, 6, 3 a nakoniec 1 najlepšiu knihu.

Tento systém hlasovania pridáva do aplikácie dynamiku a umožňuje používateľom aktívne ovplyvňovať hodnotenie kníh a autorov. Výsledky hlasovania poskytujú prehľad o popularite jednotlivých kníh a autorov v komunite.

Kritériá hodnotenia

Dedičnosť

Vo všeobecnosti mám dve vetvy dedičnosti, jednou sú knihy a druhou je User -> Client-> (BannedClient + NotBannedClient) + Admin

```java
public abstract class User implements Serializable, FavouritesObserver {  4 inheritors  Kovalenko Dmytro *
    /** List of the most recently interacted books by the user. */
    private List<Book> latestBook;  3 usages
    /** Static counter to track the number of User instances. */
    private static int COUNT_OF_PEOPLE;  7 usages
    /** Unique identifier for the user. */
    private int id;  3 usages

    static {
        try {
            COUNT_OF_PEOPLE = SerializationForCountOfPeople.loadCount();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
    // Initializer block to set the user's unique ID
    {
        COUNT_OF_PEOPLE++;
        id = COUNT_OF_PEOPLE;
    }
    /** Inner class instance containing user profile details. */
    private UserProfile profile;  10 usages
    /** List of user's favorite books. */
    private List<Book> favouriteBooks;  3 usages
    /** Flag to determine if the user has administrative privileges. */
    private boolean isAdmin;  3 usages
```

Abstraktná základná trieda reprezentujúca používateľa v systéme. Táto trieda implementuje serializovateľnú funkcionalitu a funkcionalitu pozorovateľa obľúbených kníh na správu interakcií používateľa s knihami. Každý používateľ má jedinečný identifikátor, údaje o profile a zoznamy kníh, s ktorými v poslednom čase interagoval alebo ktoré označil ako obľúbené. Poskytuje metódy na správu podrobností o používateľovi a správanie súvisiace so správou kníh

```java
public class Client extends User {  2 inheritors  Kovalenko Dmytro *
    /**
     * Default constructor that initializes a new client with default settings.
     */
    public Client(){}  Kovalenko Dmytro
    /**
     * Constructs a new client with detailed user information and admin status.
     * @param name Client's full name.
     * @param email Client's email address.
     * @param phone Client's phone number.
     * @param password Client's password for authentication.
     * @param isAdmin Indicates whether the client should have administrative privileges (typically false for clients).
     */
    public Client(String name, String email, String phone, String password, boolean isAdmin) {  8 usages  Kovalenko Dmytro
        super(name, email, phone, password, isAdmin);
    }

    /**
     * Purchases a book and prints out a confirmation.
     * @param book The book to be purchased by the client.
     */
    public void purchaseBook(Book book) { System.out.println("Book purchased: " + book.getTitle()); }
```

Reprezentuje klientskeho používateľa v systéme. Klientskí používatelia zvyčajne nemajú oprávnenia správcu. Táto trieda rozširuje triedu User a poskytuje všetky dostupné funkcie

```java
public class Admin extends User {  Kovalenko Dmytro *
    /** The singleton instance of the Admin class. */
    private static Admin instance;  3 usages
    /** The list of users managed by the admin. */
    private List<User> users;  3 usages
    /** Data access object for managing user data. */
    private UserDAO userDAO;  2 usages
    /**
     * Private constructor to prevent external instantiation and ensure singleton pattern.
     * Initializes user list and user data access object.
     */
    private Admin() {  1 usage  Kovalenko Dmytro
        userDAO = new UserDAO();
        users = userDAO.index();
        setAdmin(true);
    }
    /**
     * Provides global access to the Admin instance, creating it if it does not already exist.
     * @return The singleton instance of Admin.
     */
    public static Admin getInstance() {  Kovalenko Dmytro
        if (instance == null) {
            instance = new Admin();
        }
        return instance;
```

Reprezentuje používateľa správcu s jediným prístupom v celej aplikácii. Táto trieda rozširuje User a poskytuje ďalšie administrátorské funkcie

```java
public class BannedClient extends Client {  new *
    /**
     * Indicates whether the ban on this client is permanent.
     */
    private boolean isPermanentlyBanned;  3 usages

    /**
     * The reason why the client was banned.
     */
    private String reasonForBan;  4 usages

    /**
     * Constructs a new BannedClient with detailed user information, the reason for the ban, and its permanency status.
     * Initializes the client as non-admin by default.
     *
     * @param name The full name of the client.
     * @param email The email address of the client.
     * @param phone The phone number of the client.
     * @param password The password for the client's account.
     * @param reasonForBan The reason why the client is banned.
     * @param isPermanentlyBanned A boolean flag indicating whether the ban is permanent.
     */
    public BannedClient(String name, String email, String phone, String password, String reasonForBan, boolean isPermane
        super(name, email, phone, password,  isAdmin: false);  // Passes false for isAdmin, assuming banned clients cannot
        this.reasonForBan = reasonForBan;
        this.isPermanentlyBanned = isPermanentlyBanned;
    }
```

Predstavuje klienta, ktorému bol zakázaný prístup do knižnice. Táto trieda rozširuje triedu Client o funkcie špecifické pre prácu so zakázanými klientmi.

```java
public class NotBannedClient extends Client {  new *

    /**
     * Default constructor for creating a NotBannedClient with default settings inherited from the {@link Client} superc
     * This constructor initializes a client with no predefined information, relying on the superclass for default initi
     */
    public NotBannedClient() {  new *
        super();
    }

    /**
     * Constructs a new NotBannedClient with detailed user information, without administrative privileges.
     * This calls the constructor of the superclass {@link Client} to set up the client's basic profile.
     *
     * @param name Client's full name.
     * @param email Client's email address.
     * @param phone Client's phone number.
     * @param password Client's password for authentication.
     */
    public NotBannedClient(String name, String email, String phone, String password) {  no usages  new *
        super(name, email, phone, password,  isAdmin: false); // All NotBannedClients are initialized without admin rights
    }
```

Predstavuje klienta, ktorý nie je obmedzený z knižnice. Táto trieda rozširuje triedu Client na zapuzdrenie funkcií pre klientov s dobrou povesťou.

```java
abstract sealed public class Book implements Serializable permits FictionBook, FamilyBook, DramaBook, Autobiography, All
    private static final long serialVersionUID = 1L; // Recommended to include  no usages
    /** The title of the book. */
    private String title;  5 usages
    /** The author of the book. */
    private String author;  5 usages
    /** The publication year of the book. */
    private int year;  5 usages
    /** The genre of the book. */
    private String genre;  5 usages
    /** A brief description of the book. */
    private String description;  5 usages
    /** The URL of the image representing the book cover. */
    private String imageSrc;  3 usages
    /** The number of votes or ratings the book has received. */
    private int votes;  3 usages
    /** The unique identifier for the book. */
    private int id;  4 usages

    // Static variable to track the count of books created.
    private static int COUNT_OF_BOOKS = 1;  3 usages
    {
        id = COUNT_OF_BOOKS + 1;
    }
    /**
     * Default constructor for creating a book object without initializing fields.
     */
    public Book(){}  * Kovalenko Dmytro
```

Abstraktná zapečatená základná trieda pre knihu, ktorá poskytuje spoločnú štruktúru a funkčnosť pre všetky typy kníh

```java
public non-sealed class FictionBook extends Book {  Kovalenko Dmytro *
    /** The specific type of fiction of the book. */
    private String fictionType;  5 usages
    /**
     * Constructs a new FictionBook with specified details.
     * This constructor initializes all fields including the specific type of fiction.
     * @param id Unique identifier for the book.
     * @param title Title of the book.
     * @param author Author of the book.
     * @param year Publication year of the book.
     * @param genre Genre of the book, typically a subgenre of Fiction.
     * @param description A brief description of the book.
     * @param imageSrc Image source URL for the book cover.
     * @param votes Number of votes or ratings the book has received.
     * @param fictionType Specific type of fiction.
     */
    public FictionBook(int id,String title, String author, int year, String genre, String description, String imageSrc, in
        super(id,title, author, year, genre, description, imageSrc, votes);
        this.fictionType = fictionType;
    }
```

Reprezentuje špecifický typ knihy, konkrétne beletriu, a rozširuje všeobecnú triedu Book. Táto trieda rozširuje Book, obsahuje ďalšie atribúty špecifické pre beletristické knihy

```java
public non-sealed class FamilyBook extends Book {  Kovalenko Dmytro *
    /**
     * Indicates whether the book is considered suitable for children.
     */
    private boolean suitableForChildren;  5 usages

    /**
     * Constructs a new FamilyBook with detailed book information and suitability for children.
     * Initializes all fields with the provided values, including the suitability flag.
     *
     * @param id The unique identifier for the book.
     * @param title The title of the book.
     * @param author The author of the book.
     * @param year The publication year of the book.
     * @param genre The genre of the book.
     * @param description A brief description of the book.
     * @param imageSrc A URL or path to an image of the book's cover.
     * @param votes The number of votes or ratings the book has received.
     * @param suitableForChildren A boolean indicating whether the book is suitable for children.
     */
    public FamilyBook(int id, String title, String author, int year, String genre, String description, String imageSrc,
        super(id, title, author, year, genre, description, imageSrc, votes);
        this.suitableForChildren = suitableForChildren;
    }
```

Predstavuje knihu zaradenú do rodinného žánru, ktorá môže obsahovať ďalšie kritériá vhodnosti pre deti. Túto triedu, ktorá nie je uzavretá, možno ďalej rozšíriť, aby sa do nej mohli zaradiť špecifickejšie typy rodinných kníh. Táto trieda rozširuje Book, Pridáva vlastnosť na určenie, či je kniha vhodná pre mladších čitateľov.

```java
public non-sealed class DramaBook extends Book {  Kovalenko Dmytro *
    /**
     * The name of the main character in the drama book.
     */
    private String mainCharacter;  5 usages

    /**
     * Constructs a new DramaBook with detailed book information and the main character's name.
     * Initializes all fields with the provided values.
     *
     * @param id The unique identifier for the book.
     * @param title The title of the book.
     * @param author The author of the book.
     * @param year The publication year of the book.
     * @param genre The genre of the book, typically set to "Drama".
     * @param description A brief description of the book.
     * @param imageSrc A URL or path to an image of the book's cover.
     * @param votes The number of votes or ratings the book has received.
     * @param mainCharacter The name of the main character central to the drama narrative.
     */
    public DramaBook(int id, String title, String author, int year, String genre, String description, String imageSrc, in
        super(id, title, author, year, genre, description, imageSrc, votes);
        this.mainCharacter = mainCharacter;
    }

    /**
```

Predstavuje knihu zaradenú do žánru dráma, ktorá sa zameriava na konkrétnu hlavnú postavu. Túto neuzavretú triedu možno ďalej rozšíriť, aby sa do nej mohli zaradiť špecifickejšie typy dramatických kníh. Táto trieda rozširuje Book, Pridáva vlastnosť na uloženie mena hlavnej postavy príbehu, čím ju odlišuje v kontexte dramatických príbehov

```java
public non-sealed class Autobiography extends Book {  Kovalenko Dmytro *
    /**
     * The name of the subject whose life story is detailed in the autobiography.
     */
    private String subject;  5 usages

    /**
     * Constructs a new Autobiography with detailed book information and the subject's name.
     * Initializes all fields with the provided values, highlighting the focus on the subject's life story.
     *
     * @param id The unique identifier for the book.
     * @param title The title of the book.
     * @param author The author of the book, typically the subject or a close associate.
     * @param year The publication year of the book.
     * @param genre The genre of the book, typically set to "Autobiography".
     * @param description A brief description of the book, often outlining key events in the subject's life.
     * @param imageSrc A URL or path to an image of the book's cover.
     * @param votes The number of votes or ratings the book has received.
     * @param subject The name of the subject of the autobiography.
     */
    public Autobiography(int id, String title, String author, int year, String genre, String description, String imageSr
        super(id, title, author, year, genre, description, imageSrc, votes);
        this.subject = subject;
    }
```

Predstavuje knihu zaradenú do žánru autobiografie, ktorá sa zameriava na život konkrétneho subjektu. Túto neuzavretú triedu je možné ďalej rozšíriť tak, aby zahŕňala špecifickejšie typy autobiografických diel. Táto trieda rozširuje Book, Pridáva

vlastnosť na uloženie mena subjektu autobiografie, čím zvyšuje jej naratívne zameranie

```java
public non-sealed class AllegoricalBook extends Book {  Kovalenko Dmytro *
    /**
     * The type of allegory employed in the book, describing the symbolic narrative technique used.
     */
    private String allegoryType;  5 usages

    /**
     * Constructs a new AllegoricalBook with detailed book information and the type of allegory employed.
     * Initializes all fields with the provided values, highlighting the allegorical narrative technique.
     *
     * @param id The unique identifier for the book.
     * @param title The title of the book.
     * @param author The author of the book.
     * @param year The publication year of the book.
     * @param genre The genre of the book, typically set to "Allegorical".
     * @param description A brief description of the book, outlining its allegorical elements.
     * @param imageSrc A URL or path to an image of the book's cover.
     * @param votes The number of votes or ratings the book has received.
     * @param allegoryType A description of the specific type of allegory used in the book.
     */
    public AllegoricalBook(int id, String title, String author, int year, String genre, String description, String imageS
        super(id, title, author, year, genre, description, imageSrc, votes);
        this.allegoryType = allegoryType;
    }
```

Predstavuje knihu zaradenú do alegorického žánru, ktorá využíva alegóriu ako hlavný rozprávačský prostriedok. Túto neuzavretú triedu možno ďalej rozšíriť tak, aby zahŕňala špecifickejšie typy alegorických diel. Táto trieda rozširuje knihu, Pridáva vlastnosť na uloženie typu použitej alegórie, čím zvyšuje jej naratívnu a tematickú hĺbku

```java
public non-sealed class AdventureBook extends Book {  Kovalenko Dmytro *
    /**
     * The name of the main character in the adventure book, around whom the plot typically revolves.
     */
    private String mainCharacter;  5 usages

    /**
     * Constructs a new AdventureBook with detailed book information, including the main character's name.
     * Initializes all fields with the provided values, enhancing the thematic focus on adventure and exploration.
     *
     * @param id The unique identifier for the book.
     * @param title The title of the book.
     * @param author The author of the book.
     * @param year The publication year of the book.
     * @param genre The genre of the book, typically set to "Adventure".
     * @param description A brief description of the book, emphasizing the adventurous elements.
     * @param imageSrc A URL or path to an image of the book's cover.
     * @param votes The number of votes or ratings the book has received.
     * @param mainCharacter The name of the main character central to the adventure.
     */
    public AdventureBook(int id, String title, String author, int year, String genre, String description, String imageSr
        super(id, title, author, year, genre, description, imageSrc, votes);
        this.mainCharacter = mainCharacter;
    }
```

Predstavuje knihu zaradenú do dobrodružného žánru, ktorá sa zameriava predovšetkým na hrdinské činy a zážitky ústrednej postavy. Túto nezapečatenú triedu možno ďalej rozšíriť, aby sa do nej zmestili špecifickejšie typy dobrodružných kníh.

Táto trieda rozširuje knihu, Pridáva vlastnosť na uloženie mena hlavnej postavy, ktorá zvýrazňuje dobrodružný príbeh sústredený okolo tejto postavy

Polymorphism

```
abstract sealed public class Book implements Serializable permits FictionBook, FamilyBook, DramaBook, Autobiography, A
    public void setDescription(String description) { this.description = description; }
    /**
     * Prints detailed information about the book.
     */
    public void printInfo() {  7 usages  6 overrides  Kovalenko Dmytro
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Year: " + year);
        System.out.println("Genre: " + genre);
        System.out.println("Description: " + description);
    }
    /**
     * Returns a string representation of the book including title, author, year, genre, and description.
     * @return String representation of the book.
     */
    @Override  6 overrides  Kovalenko Dmytro
    public String toString() {
        return "Book{" +
                "title='" + title + '\'' +
                ", author='" + author + '\'' +
                ", year=" + year +
                ", genre='" + genre + '\'' +
                ", description='" + description + '\'' +
                '}';
    }
```

```
public non-sealed class FictionBook extends Book {  Kovalenko Dmytro *
    /**
     * Returns the specific type of fiction of the book.
     * @return The fiction type.
     */
    public String getFictionType() { return fictionType; }
    /**
     * Sets the specific type of fiction for the book.
     * @param fictionType New type of fiction.
     */
    public void setFictionType(String fictionType) { this.fictionType = fictionType; }
    /**
     * Prints detailed information about the fiction book, including common book info and specific fiction type.
     */
    @Override  7 usages  Kovalenko Dmytro
    public void printInfo() {
        super.printInfo(); // Call superclass method to print common info
        System.out.println("Type: Fiction");
        System.out.println("Fiction Type: " + fictionType);
    }
    /**
     * Returns a string representation of the fiction book including all common book details and the fiction type.
     * @return String representation of the fiction book.
     */
    @Override  Kovalenko Dmytro
    public String toString() { return super.toString() + " FictionType: " + fictionType; }
}
```

```java
public non-sealed class FamilyBook extends Book {  Kovalenko Dmytro *
    public void setSuitableForChildren(boolean suitableForChildren) {  no usages  Kovalenko Dmytro
        this.suitableForChildren = suitableForChildren;
    }

    /**
     * Prints detailed information about the FamilyBook, including its general information and its suitability for child
     * This method overrides the {@code printInfo} method from the {@code Book} superclass to add additional information
     * about the book's suitability for children.
     */
    @Override  7 usages  Kovalenko Dmytro
    public void printInfo() {
        super.printInfo(); // Call superclass method to print common info
        System.out.println("Type: Family");
        System.out.println("Suitable for Children: " + (suitableForChildren ? "Yes" : "No"));
    }

    /**
     * Provides a string representation of the FamilyBook, including its suitability for children.
     * This method appends the suitability information to the string representation provided by the {@code Book} supercl
     *
     * @return A string representation of the FamilyBook, including details about its suitability for children.
     */
    @Override  Kovalenko Dmytro
    public String toString() { return super.toString() + " SuitableForChildren: " + suitableForChildren; }
}
```

Prepíšem metódu toString a printInfo

```java
public static Book createBook(int id,String genre, String title, String author, int year, String description, String
    return switch (genre) {
        case "Fiction" -> new FictionBook(id, title, author, year, genre, description, imageSrc, votes, some);
        case "Drama" -> new DramaBook(id, title, author, year, genre, description, imageSrc, votes, some);
        case "Adventure" -> new AdventureBook(id, title, author, year, genre, description, imageSrc, votes, some);
        case "Allegorical" ->
                new AllegoricalBook(id, title, author, year, genre, description, imageSrc, votes, some);
        case "Autobiography" ->
                new Autobiography(id, title, author, year, genre, description, imageSrc, votes, some);
```

```java
@Override  Kovalenko Dmytro
public List<Book> index() {
    CompletableFuture<List<Book>> future = CompletableFuture.supplyAsync(() -> {
        List<Book> books = new ArrayList<>();
        String SQL = "SELECT * FROM books";
        try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
            ResultSet resultSet = preparedStatement.executeQuery();
            while (resultSet.next()) {
                Book book = createBookFromResultSet(resultSet);
                books.add(book);
            }
        } catch (SQLException e) {
            System.err.println("Database error while fetching books: " + e.getMessage());
            throw new RuntimeException("Database error occurred while fetching books", e);
        }
        return books;
```

```java
public interface Movement {  9 implementations  Kovalenko Dmytro *

    /** Moves to the top books section.
     * @param event The MouseEvent triggering the action.
     */
    void moveToTopBooks(MouseEvent event);  9 implementations  Kovalenko Dmytro
    /**
     * Moves to the main page.
     * @param event The MouseEvent triggering the action.
     */
    void moveToMainPage(MouseEvent event);  9 implementations  Kovalenko Dmytro
    /**
     * Goes to the profile page.
     * @param event The MouseEvent triggering the action.
     */
    void goToProfilePage(MouseEvent event);  9 implementations  Kovalenko Dmytro
    /**
     * Moves to the all books section of the application.
     *
     * @param event The MouseEvent that triggers this navigation.
     */
    void moveToAllBooks(MouseEvent event);  9 implementations  Kovalenko Dmytro
    /**
     * Handles the event when the user wants to see history .
     * Navigates to the history page.
     * @param event MouseEvent that triggers this navigation.
     */
    void moveToHistory(MouseEvent event);  9 implementations  new *
    /**
     * Handles the event when the user wants to vote.
     * Navigates to the voting page.
     * @param event MouseEvent that triggers this navigation.
     */
```

```java
public class FavouritesController implements Initializable, Movement {  Kovalenko Dmytro *  11  6
    @Override  Kovalenko Dmytro *
    @FXML
    public void moveToMainPage(MouseEvent event) {
        try {
            MainClientConroller.setUser(user);
            FXMLLoader loader = new FXMLLoader(getClass().getResource( name: "/com/example/loginpage/mainClient.fxml"));
            Parent root = loader.load();
            Scene profileScene = new Scene(root);
            stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
            stage.setScene(profileScene);
            stage.setFullScreen(true); // Set full screen mode after loading the new scene
            stage.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /**
     * Navigates to the user's profile page.
     * @param event MouseEvent that triggers this navigation.
     */
    @Override  Kovalenko Dmytro *
    @FXML
    public void goToProfilePage(MouseEvent event) {
        try {
            ProfileController.setUser(user);
            FXMLLoader loader = new FXMLLoader(getClass().getResource( name: "/com/example/loginpage/profile.fxml"));
            Parent root = loader.load();
            ProfileController profileController = loader.getController();
            Scene profileScene = new Scene(root);
            stage = (Stage)((Node)event.getSource()).getScene().getWindow();
            stage.setScene(profileScene);
            stage.setFullScreen(true); // Set full screen mode after loading the new scene
```

```java
@FXML  Kovalenko Dmytro *
@Override
public void moveToMainPage(MouseEvent event) {
    MainClientConroller.setUser(user);
    FXMLLoader loader = new FXMLLoader(getClass().getResource( name: "/com/example/loginpage/mainClient.fxml"));
    Parent root = null;
    try {
        root = loader.load();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    MainClientConroller mainClientConroller = loader.getController();
    Scene profileScene = new Scene(root);
    stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
    stage.setScene(profileScene);
    stage.setFullScreen(true);
    stage.show();
}

/**
 * Navigates to the user's profile page.
 * @param event MouseEvent that triggers this navigation.
 */
@Override  Kovalenko Dmytro
@FXML
public void goToProfilePage(MouseEvent event) {
    try {
        ProfileController.setUser(user);
        FXMLLoader loader = new FXMLLoader(getClass().getResource( name: "/com/example/loginpage/profile.fxml"));
        Parent root = loader.load();
```

A všetky ostatné podobné triedy, v ktorých existujú prechody

Zapuzdrenie

```java
public abstract class User implements Serializable, FavouritesObserver {  4 inheritors  Kovalenko Dmytro *
    /** List of the most recently interacted books by the user. */
    private List<Book> latestBook;  3 usages
    /** Static counter to track the number of User instances. */
    private static int COUNT_OF_PEOPLE;  7 usages
    /** Unique identifier for the user. */
    private int id;  3 usages

    static {
        try {
            COUNT_OF_PEOPLE = SerializationForCountOfPeople.loadCount();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
    // Initializer block to set the user's unique ID
    {
        COUNT_OF_PEOPLE++;
        id = COUNT_OF_PEOPLE;
    }
    /** Inner class instance containing user profile details. */
    private UserProfile profile;  10 usages
    /** List of user's favorite books. */
    private List<Book> favouriteBooks;  3 usages
    /** Flag to determine if the user has administrative privileges. */
    private boolean isAdmin;  3 usages
```

```java
// Inner class to handle user profile details
private class UserProfile {  3 usages  new *
    /** User's full name. */
    private String name;  2 usages
    /** User's email address. */
    private String email;  2 usages
    /** User's phone number. */
    private String phone;  2 usages
    /** User's account password. */
    private String password;  2 usages
    /**
     * Default constructor for user profile.
     */
    public UserProfile() {}  1 usage  new *
```

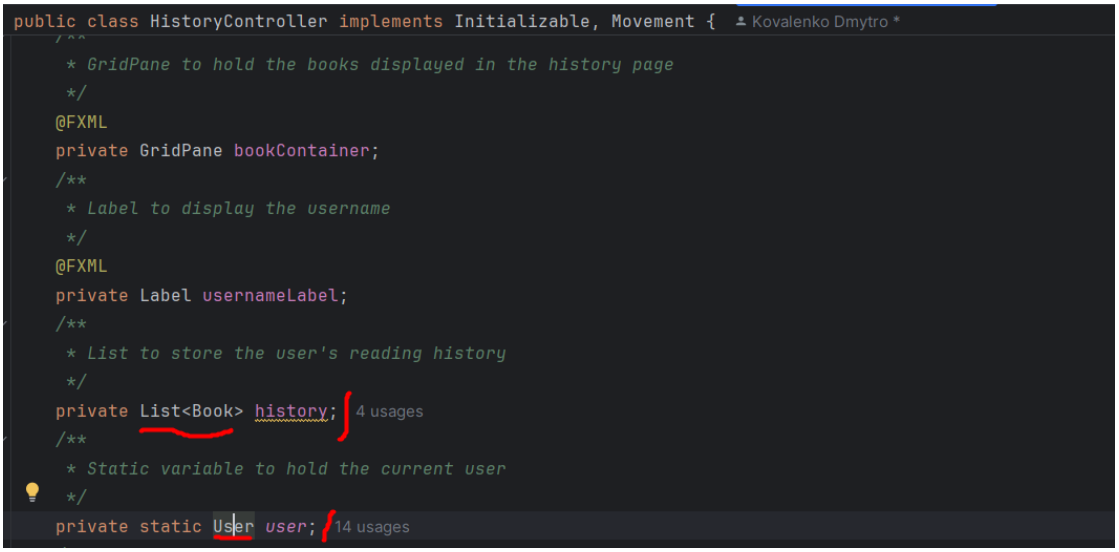
```java
abstract sealed public class Book implements Serializable permits FictionBook, FamilyBook, DramaBook, Autobiography, Atte
    private static final long serialVersionUID = 1L; // Recommended to include  no usages
    /** The title of the book. */
    private String title;  5 usages
    /** The author of the book. */
    private String author;  5 usages
    /** The publication year of the book. */
    private int year;  5 usages
    /** The genre of the book. */
    private String genre;  5 usages
    /** A brief description of the book. */
    private String description;  5 usages
    /** The URL of the image representing the book cover. */
    private String imageSrc;  3 usages
    /** The number of votes or ratings the book has received. */
    private int votes;  3 usages
    /** The unique identifier for the book. */
    private int id;  4 usages

    // Static variable to track the count of books created.
    private static int COUNT_OF_BOOKS = 1;  3 usages
    {
        id = COUNT_OF_BOOKS + 1;
    }
```

```
public class MainClientConroller implements Initializable, Movement {  Kovalenko Dmytro *
    /**
     * HBox container for displaying book cards
     */
    @FXML
    private HBox cardLayoout;
    /**
     * GridPane container for displaying books
     */
    @FXML
    private GridPane bookContainer;
    /**
     * Label displaying the username
     */
    @FXML
    private Label usernameLabel;
    /**
     *  TextField for searching books
     */
    @FXML
    private TextField filterField;
    /**
     * List to store recently added books
     */
    private List<Book> recentlyAdded;  2 usages
    /**
     * List to store recommended books
     */
    private List<Book> recommended;  2 usages
    /**
```

Všetky polia v takmer všetkých triedach som nastavil na súkromné. Je to potrebné najmä pre údaje o používateľoch.

AGREGACIA

```
public class MainClientConroller implements Initializable, Movement {  Kovalenko Dmytro *  29
    @FXML
    private Label usernameLabel;
    /**
     *  TextField for searching books
     */
    @FXML
    private TextField filterField;
    /**
     * List to store recently added books
     */
    private List<Book> recentlyAdded;  2 usages
    /**
     * List to store recommended books
     */
    private List<Book> recommended;  2 usages
    /**
     * Current user
     */
    private static User user;  15 usages
```



Premenná user obsahuje inštanciu používateľa, ktorá predstavuje používateľa, ktorý práve používa aplikáciu. Ide o kľúčový agregačný bod, pretože spája celú reláciu používateľa s konkrétnou osobou, čo ovplyvňuje spôsob prezentácie údajov, ako sú odporúčania a nedávno pridané knihy

recentlyAdded:

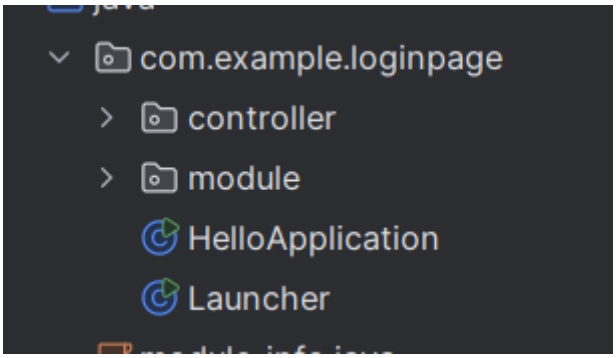
Tento zoznam obsahuje objekty typu Book, ktoré predstavujú knihy, ktoré boli nedávno pridané do knižnice. Priamo agreguje údaje o knihách tým, že udržiava zoznam novozískaných kníh

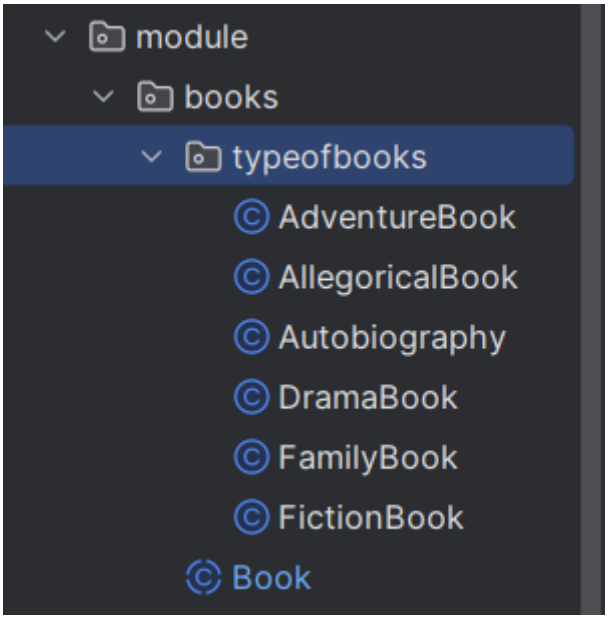
Recommended:

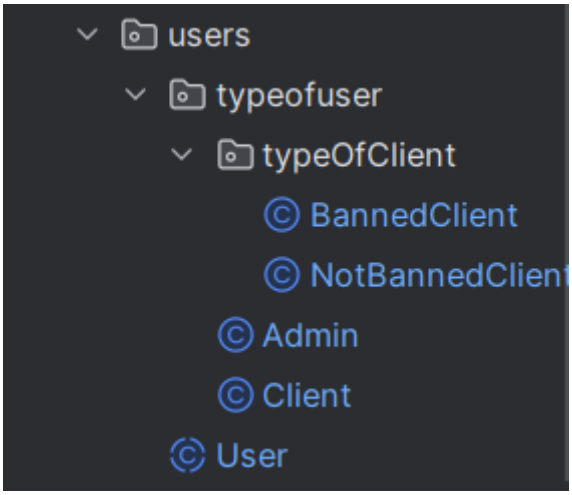
Podobne ako v prípade nedávno pridaných kníh, aj v tomto zozname sa nachádzajú knihy, ktoré sú používateľovi odporúčané, prípadne na základe jeho predchádzajúcich preferencií alebo populárnych trendov. Je to ďalšia priama agregácia objektov Book prispôsobená záujmom používateľa.

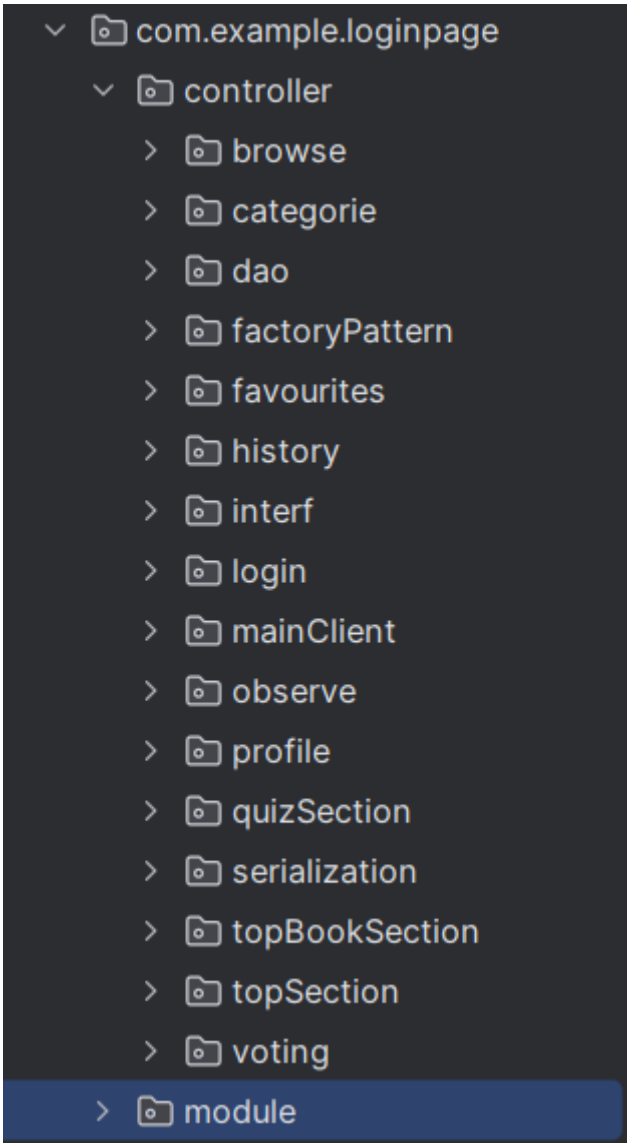
A ďalšie mám na iných miestach

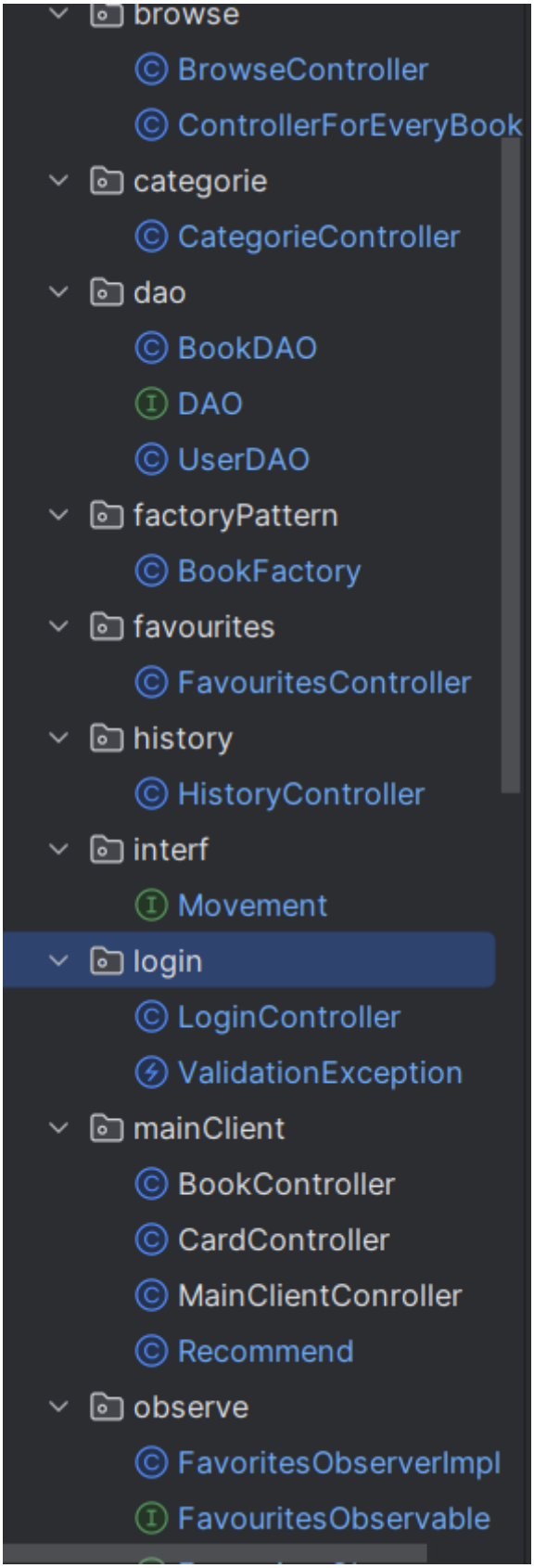
Kod v balike

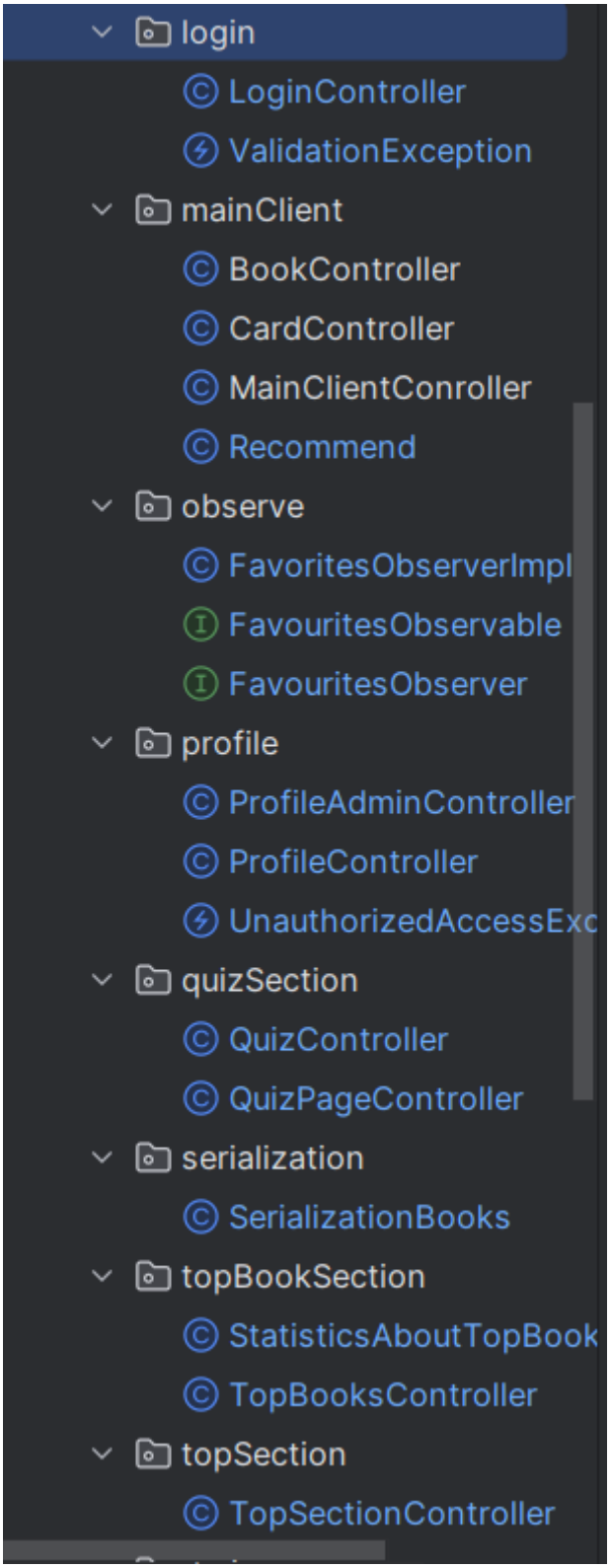



module
books
typeofbooks
AdventureBook
AllegoricalBook
Autobiography
DramaBook
FamilyBook
FictionBook
Book


users
typeofuser
typeOfClient
BannedClient
NotBannedClient
Admin
Client
User


com.example.loginpage
controller
browse
categorie
dao
factoryPattern
favourites
history
interf
login
mainClient
observe
profile
quizSection
serialization
topBookSection
topSection
voting
module


browse
BrowseController
ControllerForEveryBook
categorie
CategorieController
dao
BookDAO
DAO
UserDAO
factoryPattern
BookFactory
favourites
FavouritesController
history
HistoryController
interf
Movement
login
LoginController
ValidationException
mainClient
BookController
CardController
MainClientConroller
Recommend
observe
FavoritesObserverImpl
FavouritesObservable



Projekt som rozdelil na niekoľko častí. Prvou sú zdroje, druhou je kód. V zdrojoch sú všetky fotografie, fxml a css. V kóde som rozdelil štruktúru na controller a model.

Používanie serializácie

```java
public class SerializationBooks {  Kovalenko Dmytro *
    /** Flag for serialization */
    private static boolean flag;  3 usages
    /** Checks if the serialization flag is set. ...*/
    public static boolean isFlag() {  Kovalenko Dmytro
        return flag;
    }
    /** Default constructor for the SerializationBooks class. */
    public SerializationBooks(){}  new *

    /** Sets the serialization flag. ...*/
    public static void setFlag(boolean flag) { SerializationBooks.flag = flag; }

    /** Serializes a list of books to a specified file. ...*/
    public static void serializeLatestBooks(List<Book> latestBooks, String filename) throws IOException {  1 usage  Kovale
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filename))) {
            flag = true;
            oos.writeObject(latestBooks);
        }
    }
    /** Deserializes a list of books from a specified file. ...*/
    public static List<Book> deserializeLatestBooks(String filename) throws IOException, ClassNotFoundException {  1 usage
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(filename))) {
            return (List<Book>) ois.readObject();
        }
    }
}
```

Poskytuje obslužné metódy na serializáciu a deserializáciu zoznamov kníh do a z trvalého úložiska. Táto trieda sa zvyčajne používa na správu perzistencie údajov špecifických pre používateľa, ako sú jeho posledné interakcie s knihami.

```java
41      public class HistoryController implements Initializable, Movement {  Kovalenko Dmytro *
84          /**
85           * Initializes the controller after its root element has been completely processed.
86           * @param location  The location used to resolve relative paths for the root object, or null
87           * @param resources The resources used to localize the root object, or null if the root objec
88           */
89          @Override  Kovalenko Dmytro *
90          public void initialize(URL location, ResourceBundle resources) {
91              System.out.println(user);
92              usernameLabel.setText(user.getName());
93              try {
94                  if(SerializationBooks.isFlag()) {
95                      history = SerializationBooks.deserializeLatestBooks( filename: "latestBooks.ser"); /
96                      System.out.println(history);
```

```java
public class HelloApplication extends Application {  Kovalenko Dmytro *
    public static void exitApplication() {  1 usage  Kovalenko Dmytro *
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
        alert.setTitle("Exit Confirmation");
        alert.setHeaderText("Exit Application");
        alert.setContentText("Are you sure you want to exit?");

        alert.initModality(Modality.APPLICATION_MODAL);

        Optional<ButtonType> result = alert.showAndWait();
        if (result.isPresent() && result.get() == ButtonType.OK) {
            try {
                // Serialize the books before exiting
                if (booksToSerialize != null) {
                    SerializationBooks.serializeLatestBooks(booksToSerialize, filename: "latestBooks.ser");
                }
            } catch (IOException e) {
                e.printStackTrace();
                // Handle exception, possibly notify the user that data saving failed
                Alert errorAlert = new Alert(Alert.AlertType.ERROR);
                errorAlert.setTitle("Error");
                errorAlert.setHeaderText("Save Error");
                errorAlert.setContentText("Failed to save book data!");
                errorAlert.showAndWait();
            }
            Platform.exit();
            System.exit( status: 0);
        }
```

Deserializujem na stránke histórie a serializujem na konci aplikácie. Zvyčajne to používam na uloženie histórie konkrétneho používateľa

```java
public class SerializationForCountOfPeople {  new *

    private static final String COUNT_FILE = "count_of_people.dat";  2 usages
    /**
     * Saves the current count of people to a specified file.
     * This method writes the count to a file using {@link DataOutputStream}
     * to ensure it is stored persistently and can be retrieved in subsequent sessions.
     *
     * @param count the current count of people to be saved.
     * @throws IOException if an error occurs during the write operation.
     */
    public static void saveCount(int count) throws IOException {  new *
        System.out.println(count);
        try (DataOutputStream dos = new DataOutputStream(new FileOutputStream(COUNT_FILE))) {
            dos.writeInt(count);
        } catch (IOException e) {
            System.err.println("Could not save count: " + e.getMessage());
        }
    }
    /**
     * Loads the count of people from a specified file.
     * This method reads the count from a file using {@link DataInputStream}.
     *
     * @return the loaded count of people, or 0 if the file does not exist or an error occurs.
     * @throws IOException if an error occurs during the read operation.
     */
    public static int loadCount() throws IOException {  1 usage  new *
        File file = new File(COUNT_FILE);
        if (!file.exists()) {
            return 0;
        }
```

```java
public abstract class User implements Serializable, FavouritesObserver {  4 inheritors  Kovalenko Dmytro *
    /** List of the most recently interacted books by the user. */
    private List<Book> latestBook;  3 usages
    /** Static counter to track the number of User instances. */
    private static int COUNT_OF_PEOPLE;  7 usages
    /** Unique identifier for the user. */
    private int id;  3 usages

    static {
        try {
            COUNT_OF_PEOPLE = SerializationForCountOfPeople.loadCount();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

Na sledovanie celkového počtu zaregistrovaných osôb používam serializáciu.

Používanie lambda výrazov alebo odkazov na metódy

```java
public class BrowseController implements Initializable, Movement {  Kovalenko Dmytro *
    public void initialize(URL url, ResourceBundle resourceBundle){
        displayBooks(allBooks);

        // Setting up an event handler for the search text input field
        filterField.textProperty().addListener((observable, oldValue, newValue) -> {
            // Filter books based on the entered text
            List<Book> filteredBooks = filterBooks(newValue);

            // Display filtered books
            displayBooks(filteredBooks);
        });
    }
    // Метод для фільтрації книг за назвою
    /**
     * Filters the list of books based on the provided search text.
     * @param searchText Text to filter the books.
     * @return List of books that match the search criteria.
     */
    private List<Book> filterBooks(String searchText) {  1 usage  Kovalenko Dmytro *
        searchText = searchText.trim();
        List<Book> filteredBooks = new ArrayList<>();
        if (searchText.isEmpty()) {
            return allBooks;
        }

        String[] words = searchText.split( regex: "\\s+");

        for (Book book : allBooks) {
            boolean containsAllWords = true;
            for (String word : words) {
                if (!book.getTitle().toLowerCase().contains(word.toLowerCase())) {
                    containsAllWords = false;
                    break;
```

```
        for (Book book : allBooks) {
            boolean containsAllWords = true;
            for (String word : words) {
                // Check if the title contains each individual word
                if (!book.getTitle().toLowerCase().contains(word.toLowerCase())) {
                    containsAllWords = false;
                    break; // No need to check further if one word is not found
                }
            }
            if (containsAllWords) {
                filteredBooks.add(book);
            }
        }

        return filteredBooks;
    }
```

Použite na filtrovanie kníh

```
public boolean delete(int userId, int currentUser) {
    // Create a CompletableFuture to handle the asynchronous deletion of a user.
    CompletableFuture<Boolean> future = CompletableFuture.supplyAsync(() -> {
        try {
            // Перевірка, чи користувач не намагається видалити себе
            if (userId == currentUser) {
                System.out.println("You cannot delete yourself.");
                return false;
            }

            String SQL = "DELETE FROM users WHERE id = ?";
            try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                preparedStatement.setInt( parameterIndex: 1, userId);

                int rowsDeleted = preparedStatement.executeUpdate();
                if (rowsDeleted > 0) {
                    System.out.println("User with ID " + userId + " deleted successfully.");
                    return true;
                } else {
                    System.out.println("User with ID " + userId + " not found.");
                    return false;
                }
            }
        } catch (SQLException e) {
            System.out.println("Error deleting user: " + e.getMessage());
            return false;
        }
    });
```

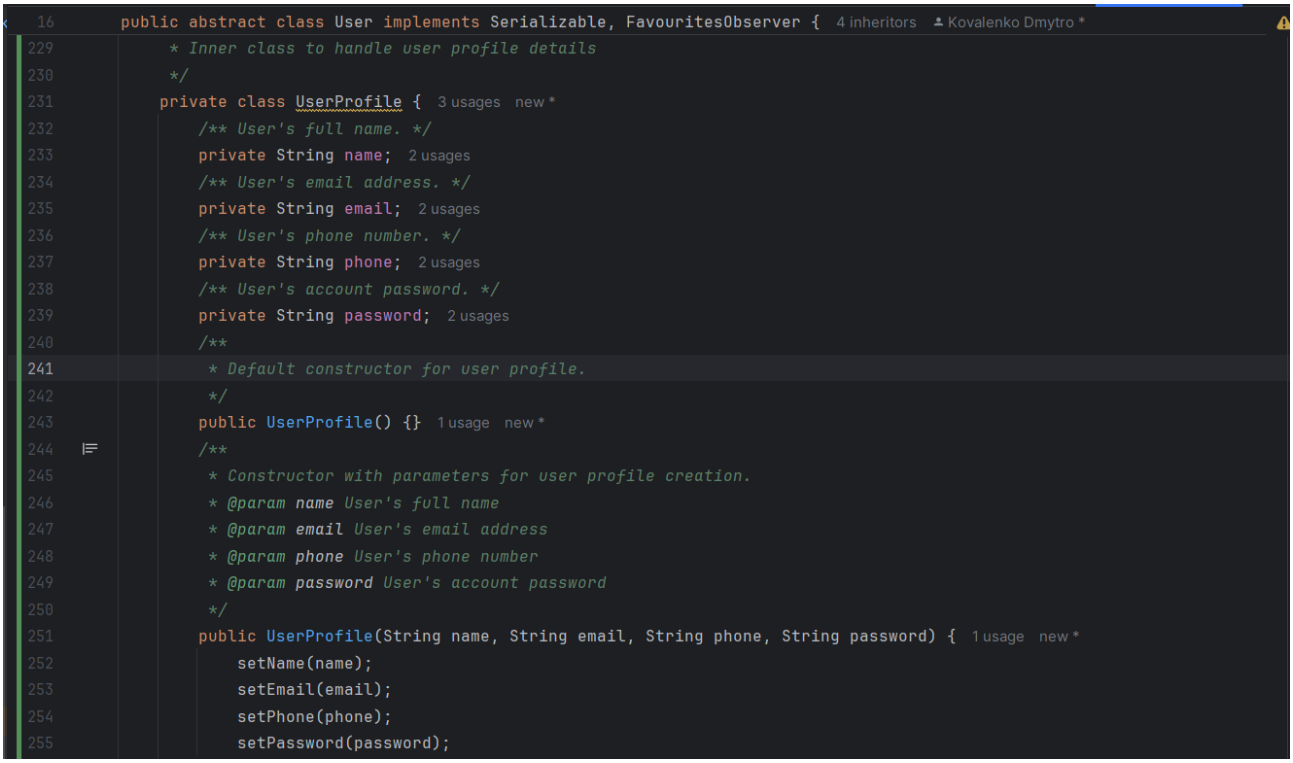
```
public void executeInBackground(Runnable task) {
    executor.submit(task);
}
```

```java
public List<User> index() {  Kovalenko Dmytro *
    List<User> users = new ArrayList<>();
    CountDownLatch latch = new CountDownLatch(1); // Create a latch to wait for background execution

    executeInBackground(() -> {
        try {
            String SQL = "SELECT * FROM users";
            try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                ResultSet resultSet = preparedStatement.executeQuery();

                while (resultSet.next()) {
                    User user = new NotBannedClient();

                    user.setId(resultSet.getInt( columnLabel: "id"));
                    user.setName(resultSet.getString( columnLabel: "name"));
                    user.setEmail(resultSet.getString( columnLabel: "email"));
                    user.setPhone(resultSet.getString( columnLabel: "phone"));
                    user.setPassword(resultSet.getString( columnLabel: "password"));

                    users.add(user);
                }
            }
            latch.countDown();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    });

    try {
```

```java
public User show(String email, String password) {  Kovalenko Dmytro *
    CompletableFuture<User> future = CompletableFuture.supplyAsync(() -> {
        try {
            String SQL = "SELECT * FROM users WHERE email = ? AND password = ?";
            try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                preparedStatement.setString( parameterIndex: 1, email);
                preparedStatement.setString( parameterIndex: 2, password);

                ResultSet resultSet = preparedStatement.executeQuery();

                if (resultSet.next()) {
                    if (resultSet.getString( columnLabel: "email").equals("kovalenkodima@gmail.com") && resultSet.getS
                        User user = Admin.getInstance();
                        user.setId(resultSet.getInt( columnLabel: "id"));
                        user.setName(resultSet.getString( columnLabel: "name"));
                        user.setEmail(resultSet.getString( columnLabel: "email"));
                        user.setPhone(resultSet.getString( columnLabel: "phone"));
                        user.setPassword(resultSet.getString( columnLabel: "password"));
                        return user;
                    } else {
                        User user = new NotBannedClient();
                        user.setId(resultSet.getInt( columnLabel: "id"));
                        user.setName(resultSet.getString( columnLabel: "name"));
                        user.setEmail(resultSet.getString( columnLabel: "email"));
                        user.setPhone(resultSet.getString( columnLabel: "phone"));
                        user.setPassword(resultSet.getString( columnLabel: "password"));
                        return user;
                    }
                }
```

Vo všeobecnosti používam funkciu lambda v DAO pre CompetableFuture: Predstavuje budúci výsledok asynchrónneho výpočtu. A tak ďalej

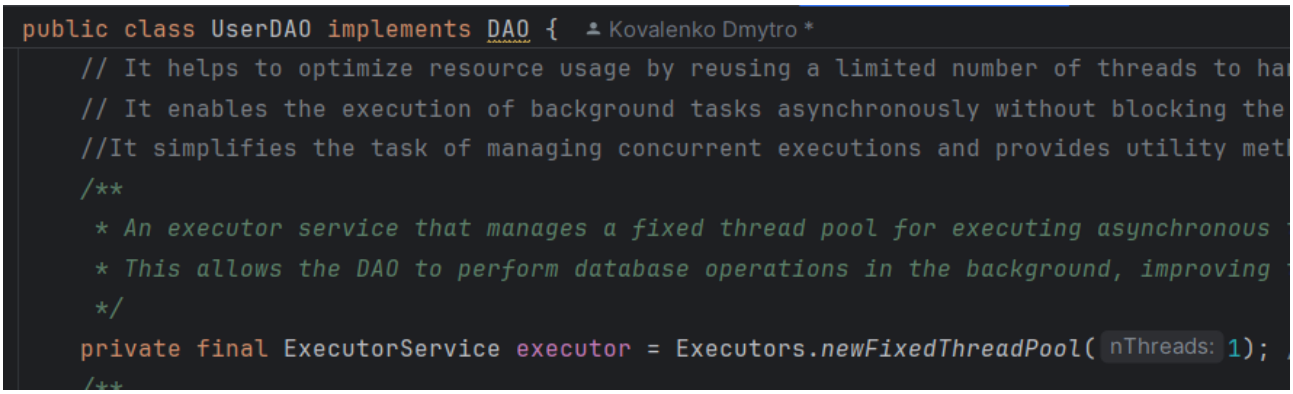
Používanie vnútorných tried a rozhraní



```
16  public abstract class User implements Serializable, FavouritesObserver {  4 inheritors  Kovalenko Dmytro *
229     * Inner class to handle user profile details
230     */
231    private class UserProfile {  3 usages  new *
232        /** User's full name. */
233        private String name;  2 usages
234        /** User's email address. */
235        private String email;  2 usages
236        /** User's phone number. */
237        private String phone;  2 usages
238        /** User's account password. */
239        private String password;  2 usages
240        /**
241         * Default constructor for user profile.
242         */
243        public UserProfile() {}  1 usage  new *
244        /**
245         * Constructor with parameters for user profile creation.
246         * @param name User's full name
247         * @param email User's email address
248         * @param phone User's phone number
249         * @param password User's account password
250         */
251        public UserProfile(String name, String email, String phone, String password) {  1 usage  new *
252            setName(name);
253            setEmail(email);
254            setPhone(phone);
255            setPassword(password);
```

Používam ho na ukladanie dôležitých informácií o používateľovi (meno, e-mail, telefón, heslo) v samostatnej triede.

Multithreading

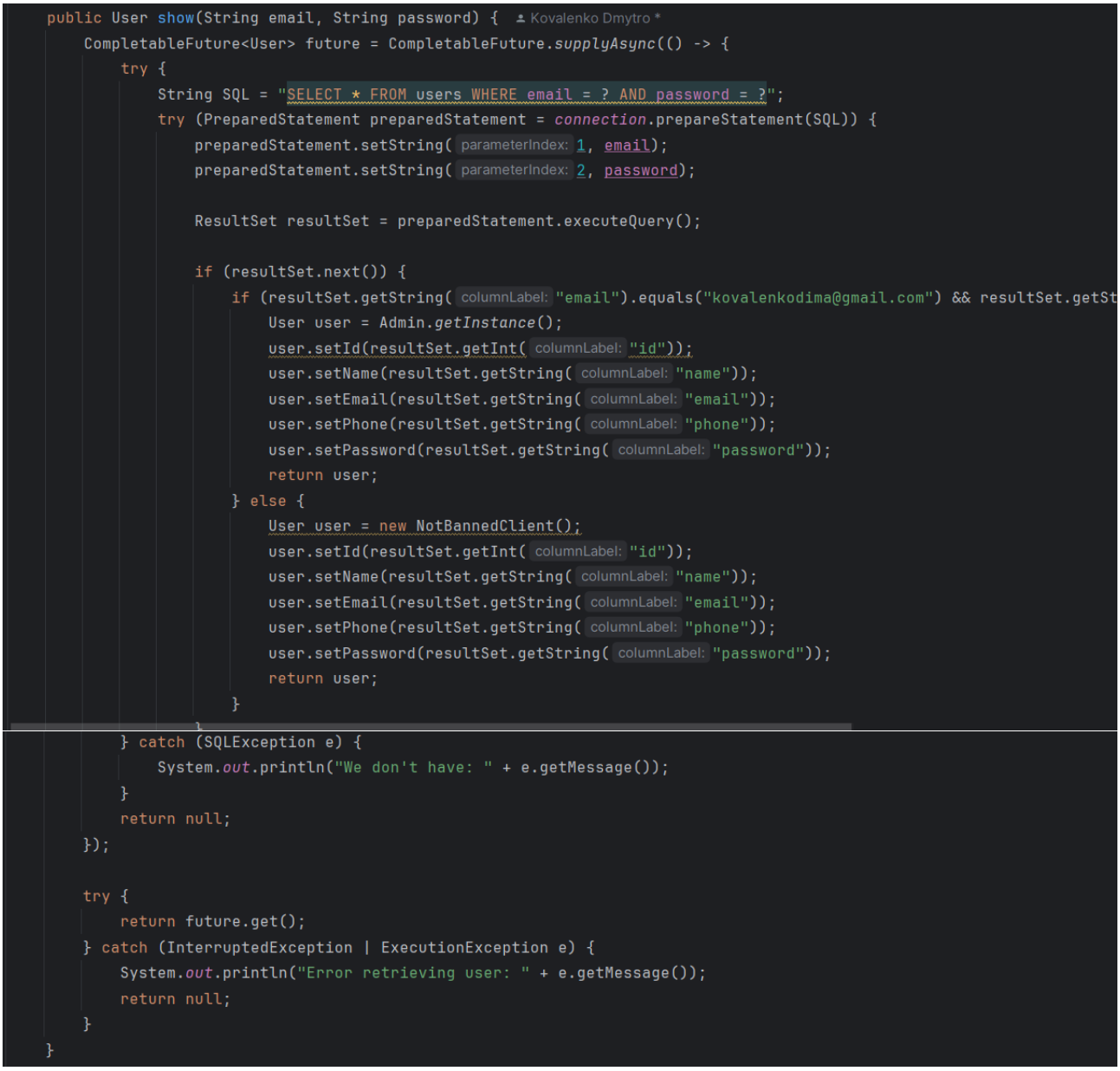


```
public class UserDAO implements DAO {  Kovalenko Dmytro *
    // It helps to optimize resource usage by reusing a limited number of threads to ha
    // It enables the execution of background tasks asynchronously without blocking the
    //It simplifies the task of managing concurrent executions and provides utility met
    /**
     * An executor service that manages a fixed thread pool for executing asynchronous
     * This allows the DAO to perform database operations in the background, improving
     */
    private final ExecutorService executor = Executors.newFixedThreadPool( nThreads: 1);
    /**
```

```java
public List<User> index() {
    List<User> users = new ArrayList<>();
    CountDownLatch latch = new CountDownLatch(1); // Create a latch to wait for background execution

    executeInBackground(() -> {
        try {
            String SQL = "SELECT * FROM users";
            try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                ResultSet resultSet = preparedStatement.executeQuery();

                while (resultSet.next()) {
                    User user = new NotBannedClient();

                    user.setId(resultSet.getInt( columnLabel: "id"));
                    user.setName(resultSet.getString( columnLabel: "name"));
                    user.setEmail(resultSet.getString( columnLabel: "email"));
                    user.setPhone(resultSet.getString( columnLabel: "phone"));
                    user.setPassword(resultSet.getString( columnLabel: "password"));

                    users.add(user);
                }
            }
            latch.countDown();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    });
```

```java
    public boolean delete(int userId, int currentUser) {
        // Create a CompletableFuture to handle the asynchronous deletion of a user.
        CompletableFuture<Boolean> future = CompletableFuture.supplyAsync(() -> {
            try {
                // Перевірка, чи користувач не намагається видалити себе
                if (userId == currentUser) {
                    System.out.println("You cannot delete yourself.");
                    return false;
                }

                String SQL = "DELETE FROM users WHERE id = ?";
                try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                    preparedStatement.setInt( parameterIndex: 1, userId);

                    int rowsDeleted = preparedStatement.executeUpdate();
                    if (rowsDeleted > 0) {
                        System.out.println("User with ID " + userId + " deleted successfully.");
                        return true;
                    } else {
                        System.out.println("User with ID " + userId + " not found.");
                        return false;
                    }
                }
            } catch (SQLException e) {
                System.out.println("Error deleting user: " + e.getMessage());
```

```java
            } catch (SQLException e) {
                System.out.println("Error deleting user: " + e.getMessage());
                return false;
            }
        });

        try {
            return future.get();
        } catch (InterruptedException | ExecutionException e) {
            System.out.println("Error deleting user: " + e.getMessage());
            return false;
        }
    }
```



```
public User show(String email, String password) {  Kovalenko Dmytro *
    CompletableFuture<User> future = CompletableFuture.supplyAsync(() -> {
        try {
            String SQL = "SELECT * FROM users WHERE email = ? AND password = ?";
            try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                preparedStatement.setString( parameterIndex: 1, email);
                preparedStatement.setString( parameterIndex: 2, password);

                ResultSet resultSet = preparedStatement.executeQuery();

                if (resultSet.next()) {
                    if (resultSet.getString( columnLabel: "email").equals("kovalenkodima@gmail.com") && resultSet.getSt
                        User user = Admin.getInstance();
                        user.setId(resultSet.getInt( columnLabel: "id"));
                        user.setName(resultSet.getString( columnLabel: "name"));
                        user.setEmail(resultSet.getString( columnLabel: "email"));
                        user.setPhone(resultSet.getString( columnLabel: "phone"));
                        user.setPassword(resultSet.getString( columnLabel: "password"));
                        return user;
                    } else {
                        User user = new NotBannedClient();
                        user.setId(resultSet.getInt( columnLabel: "id"));
                        user.setName(resultSet.getString( columnLabel: "name"));
                        user.setEmail(resultSet.getString( columnLabel: "email"));
                        user.setPhone(resultSet.getString( columnLabel: "phone"));
                        user.setPassword(resultSet.getString( columnLabel: "password"));
                        return user;
                    }
                }
        } catch (SQLException e) {
            System.out.println("We don't have: " + e.getMessage());
        }
        return null;
    });

    try {
        return future.get();
    } catch (InterruptedException | ExecutionException e) {
        System.out.println("Error retrieving user: " + e.getMessage());
        return null;
    }
}
```

Používam to aj v inej triede BookDAO. Implementoval som viacvláknové a asynchrónne operácie pomocou CompletableFuture a spravovaného fondu vlákien prostredníctvom Executors.newFixedThreadPool. To má zlepšiť výkon aplikácie vykonávaním databázových operácií na pozadí bez blokovania hlavného aplikačného vlákna

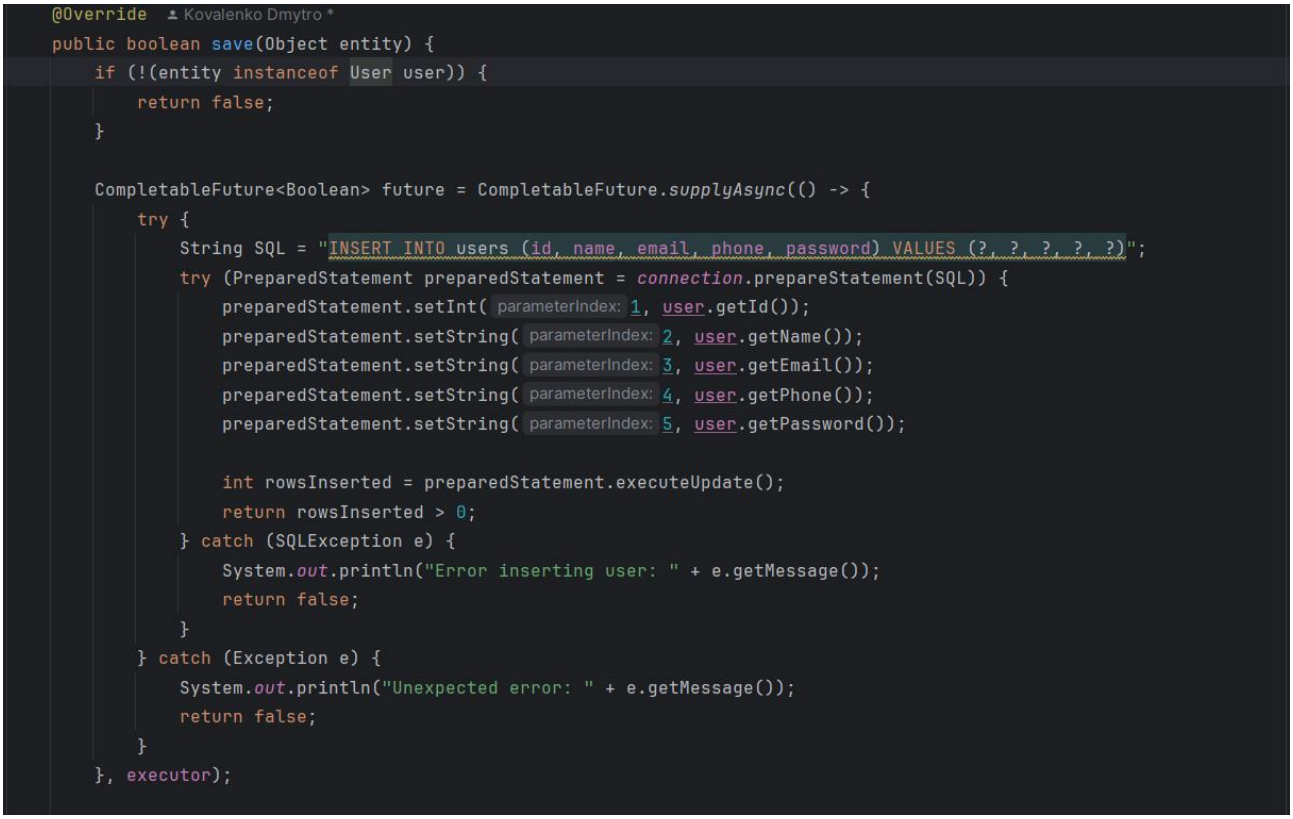
Používanie všeobecnosti vo vlastných triedach

```java
/**
 *  Interface representing Data Access Object (DAO) for generic types. DAO pattern is crucial for separating
 *  data access logic from business logic, enhancing maintainability, reusability, and scalability of the application.
 * @param <T> The type of entity handled by the DAO.
 */
public interface DAO<T> {  2 implementations  Kovalenko Dmytro *
    /**
     * Retrieves a list of entities.
     * @return A list of entities.
     */
    List<T> index();  2 implementations  Kovalenko Dmytro

    /**
     * Saves an entity.
     * @param entity The entity to be saved.
     * @return true if the entity is successfully saved, false otherwise.
     */
    boolean save(T entity);  2 implementations  Kovalenko Dmytro
    /**
     * Deletes an entity by its ID.
     *
     * @param id   The ID of the entity to be deleted.
     * @param need An additional parameter indicating whether the deletion needs confirmation.
     * @return true if the entity is successfully deleted, false otherwise.
     */
    boolean delete(int id,int need);  2 usages  2 implementations  new *
}
```

```java
    @Override  Kovalenko Dmytro
    public List<User> index() {
        List<User> users = new ArrayList<>();
        CountDownLatch latch = new CountDownLatch(1); // Create a latch to wait for background execution

        executeInBackground(() -> {
            try {
                String SQL = "SELECT * FROM users";
                try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                    ResultSet resultSet = preparedStatement.executeQuery();

                    while (resultSet.next()) {
                        User user = new NotBannedClient();

                        user.setId(resultSet.getInt( columnLabel: "id"));
                        user.setName(resultSet.getString( columnLabel: "name"));
                        user.setEmail(resultSet.getString( columnLabel: "email"));
                        user.setPhone(resultSet.getString( columnLabel: "phone"));
                        user.setPassword(resultSet.getString( columnLabel: "password"));

                        users.add(user);
                    }
                }
                latch.countDown();
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        });
```

```java
@Override  Kovalenko Dmytro *
public List<Book> index() {
    CompletableFuture<List<Book>> future = CompletableFuture.supplyAsync(() -> {
        List<Book> books = new ArrayList<>();
        String SQL = "SELECT * FROM books";
        try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
            ResultSet resultSet = preparedStatement.executeQuery();
            while (resultSet.next()) {
                Book book = createBookFromResultSet(resultSet);
                books.add(book);
            }
        } catch (SQLException e) {
            System.err.println("Database error while fetching books: " + e.getMessage());
            throw new RuntimeException("Database error occurred while fetching books", e);
        }
        return books;
    }, dbExecutor);

    try {
        return future.get();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new RuntimeException("Thread was interrupted while fetching books", e);
    } catch (ExecutionException e) {
        throw new RuntimeException("Error occurred while fetching books", e.getCause());
    }
}
```

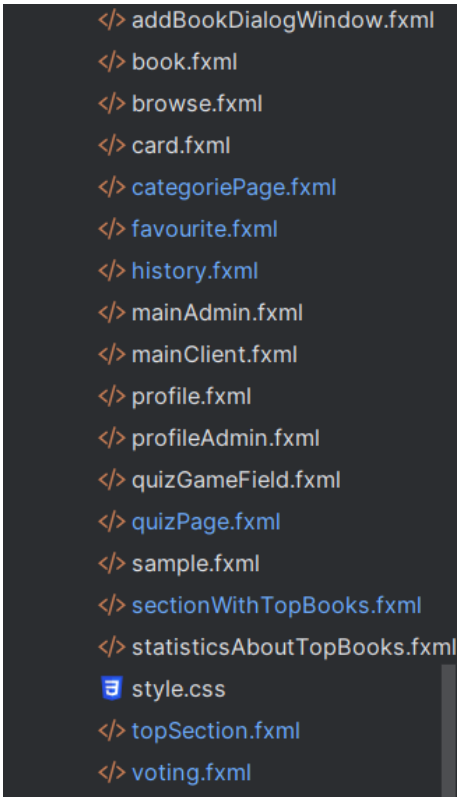
```java
@Override  Kovalenko Dmytro
public boolean save(Object entity) {
    if (!(entity instanceof Book book)) {
        return false;
    }

    String SQL = "INSERT INTO books (id, genre, title, author, year, description, votes, imagesrc) VALUES (?, ?, ?,

    try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
        preparedStatement.setInt( parameterIndex: 1, book.getId());
        preparedStatement.setString( parameterIndex: 2, book.getGenre());
        preparedStatement.setString( parameterIndex: 3, book.getTitle());
        preparedStatement.setString( parameterIndex: 4, book.getAuthor());
        preparedStatement.setInt( parameterIndex: 5, book.getYear());
        preparedStatement.setString( parameterIndex: 6, book.getDescription());
        preparedStatement.setInt( parameterIndex: 7, book.getVotes());
        preparedStatement.setString( parameterIndex: 8, book.getImageSrc());

        int affectedRows = preparedStatement.executeUpdate();
        return affectedRows > 0;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

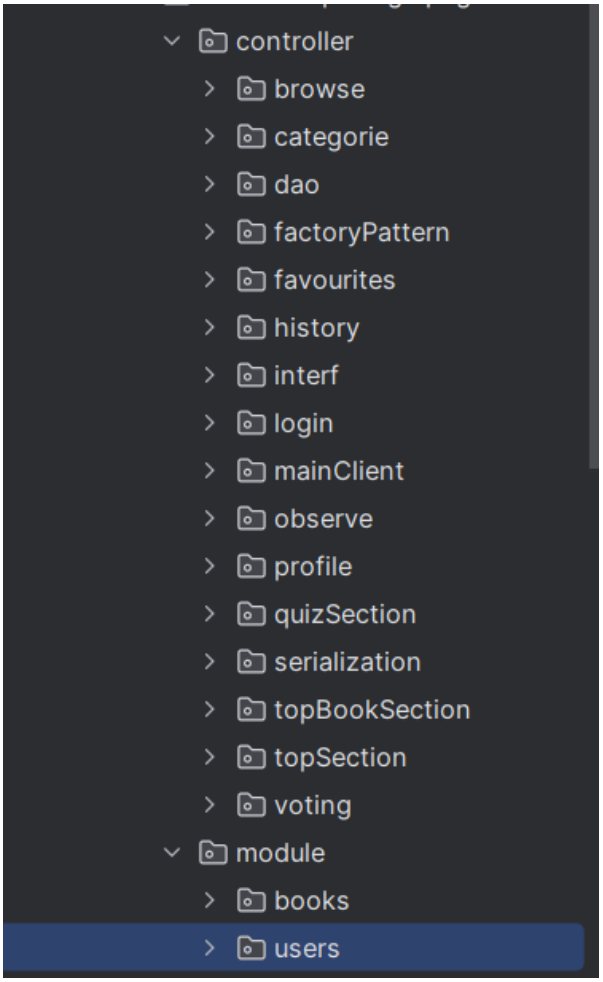


```java
@Override  Kovalenko Dmytro *
public boolean save(Object entity) {
    if (!(entity instanceof User user)) {
        return false;
    }

    CompletableFuture<Boolean> future = CompletableFuture.supplyAsync(() -> {
        try {
            String SQL = "INSERT INTO users (id, name, email, phone, password) VALUES (?, ?, ?, ?, ?)";
            try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                preparedStatement.setInt( parameterIndex: 1, user.getId());
                preparedStatement.setString( parameterIndex: 2, user.getName());
                preparedStatement.setString( parameterIndex: 3, user.getEmail());
                preparedStatement.setString( parameterIndex: 4, user.getPhone());
                preparedStatement.setString( parameterIndex: 5, user.getPassword());

                int rowsInserted = preparedStatement.executeUpdate();
                return rowsInserted > 0;
            } catch (SQLException e) {
                System.out.println("Error inserting user: " + e.getMessage());
                return false;
            }
        } catch (Exception e) {
            System.out.println("Unexpected error: " + e.getMessage());
            return false;
        }
    }, executor);
```

Všeobecné rozhranie, aby sme mohli vytvoriť spoločné metódy pre knihy a používateľa. Napríklad indexovať, uložiť, vymazať

Poskytnutie grafického používateľského rozhrania oddeleného od logiky aplikácie a s aspoň časťou obsluhy udalostí vytvorených ručne





Vo všeobecnosti sú moje stránky vytvorené v súbore fxml. Controller a Model sú rozdelené do samostatných súborov

```java
private static EventHandler<ActionEvent> addToFavouriteHandler(int userId, int bookId, Button addToFavouriteButton,
    return event -> {
        boolean isBookInFavorites = bookDAO.isFavorite(userId, bookId);
        if (isBookInFavorites) {
            if (bookDAO.deleteFromFavorites(userId, bookId)) {
                user.onFavoriteBookRemoved(book);
                addToFavouriteButton.setText("Add to Favourites");
            } else {
                Alert alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Error");
                alert.setHeaderText(null);
                alert.setContentText("Failed to remove the book from favourites.");
                alert.showAndWait();
            }
        } else {
            if (bookDAO.addToFavorites(userId, bookId)) {
                user.onFavoriteBookAdded(book);
                addToFavouriteButton.setText("Remove from Favourites");
            } else {
                Alert alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Error");
                alert.setHeaderText(null);
                alert.setContentText("Failed to add the book to favourites.");
                alert.showAndWait();
            }
        }
```

Používam ho na pridanie tejto knihy do obľúbených alebo na jej odstránenie z obľúbených.

Factory Method

```java
public class BookFactory {  Kovalenko Dmytro *
    * @param id          The unique identifier of the book.
    * @param genre       The genre of the book. Supported genres include "Fiction", "Drama", "Adventure", "Allegorical",
    *                    "Autobiography", and "Family".
    * @param title       The title of the book.
    * @param author      The author of the book.
    * @param year        The year of publication of the book.
    * @param description A brief description of the book.
    * @param imageSrc    The URL or file path of the book's cover image.
    * @param votes       The number of votes or ratings received by the book.
    * @param some        Any additional information specific to the book.
    * @return A new instance of a subclass of Book based on the specified genre.
    * @throws IllegalArgumentException If the provided genre is not supported.
    */
    public static Book createBook(int id,String genre, String title, String author, int year, String description, String imageSrc
        return switch (genre) {
            case "Fiction" -> new FictionBook(id, title, author, year, genre, description, imageSrc, votes, some);
            case "Drama" -> new DramaBook(id, title, author, year, genre, description, imageSrc, votes, some);
            case "Adventure" -> new AdventureBook(id, title, author, year, genre, description, imageSrc, votes, some);
            case "Allegorical" ->
                    new AllegoricalBook(id, title, author, year, genre, description, imageSrc, votes, some);
            case "Autobiography" ->
                    new Autobiography(id, title, author, year, genre, description, imageSrc, votes, some);
            case "Family" -> new FamilyBook(id, title, author, year, genre, description, imageSrc, votes,  suitableForChildren: true);
            default -> throw new IllegalArgumentException("Genre not supported");
        };
    }
}
```

The factory method je návrhový vzor, ktorý poskytuje rozhranie na vytváranie objektov v nadtriede, ale umožňuje podtriedam meniť typ objektov, ktoré budú vytvorené. Používam ju na vytváranie rôznych typov kníh podľa žánru

DAO

```java
public interface DAO<T> {  2 implementations  Kovalenko Dmytro *
    /**
     * Retrieves a list of entities.
     * @return A list of entities.
     */
    List<T> index();  2 implementations  Kovalenko Dmytro

    /**
     * Saves an entity.
     * @param entity The entity to be saved.
     * @return true if the entity is successfully saved, false otherwise.
     */
    boolean save(T entity);  2 implementations  Kovalenko Dmytro
    /**
     * Deletes an entity by its ID.
     *
     * @param id   The ID of the entity to be deleted.
     * @param need An additional parameter indicating whether the deletion needs confirmation.
     * @return true if the entity is successfully deleted, false otherwise.
     */
    boolean delete(int id,int need);  2 usages  2 implementations  new *
}
```

Vzor DAO (Data Access Object) je štrukturálny vzor, ktorý nám umožňuje izolovať aplikačnú/obchodnú vrstvu od vrstvy perzistencie. Používam ho na prevzatie údajov z databázy

```java
public class UserDAO implements DAO {  Kovalenko Dmytro
    /**
     * Total count of people.
     */
    private static int PEOPLE_COUNT;  no usages
    /**
     * File path for serializing user data. This variable specifies the location and filename wh
     */
    private static final String USERS_FILE = "users.ser";  2 usages
    /**
     * URL of the PostgreSQL database.
     */
```

```java
public List<User> index() {
    CountDownLatch latch = new CountDownLatch(1); // Create a latch to wait for background execution

    executeInBackground(() -> {
        try {
            String SQL = "SELECT * FROM users";
            try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                ResultSet resultSet = preparedStatement.executeQuery();

                while (resultSet.next()) {
                    User user = new NotBannedClient();

                    user.setId(resultSet.getInt( columnLabel: "id"));
                    user.setName(resultSet.getString( columnLabel: "name"));
                    user.setEmail(resultSet.getString( columnLabel: "email"));
                    user.setPhone(resultSet.getString( columnLabel: "phone"));
                    user.setPassword(resultSet.getString( columnLabel: "password"));

                    users.add(user);
                }
            }
            latch.countDown();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    });

    try {
        latch.await();
```

```java
public boolean delete(int userId, int currentUser) {
    // Create a CompletableFuture to handle the asynchronous deletion of a user.
    CompletableFuture<Boolean> future = CompletableFuture.supplyAsync(() -> {
        try {
            // Перевірка, чи користувач не намагається видалити себе
            if (userId == currentUser) {
                System.out.println("You cannot delete yourself.");
                return false;
            }

            String SQL = "DELETE FROM users WHERE id = ?";
            try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                preparedStatement.setInt( parameterIndex: 1, userId);

                int rowsDeleted = preparedStatement.executeUpdate();
                if (rowsDeleted > 0) {
                    System.out.println("User with ID " + userId + " deleted successfully.");
                    return true;
                } else {
                    System.out.println("User with ID " + userId + " not found.");
                    return false;
                }
            }
        } catch (SQLException e) {
            System.out.println("Error deleting user: " + e.getMessage());
            return false;
        }
    });
```

```java
public boolean save(User user) {  Kovalenko Dmytro
    CompletableFuture<Boolean> future = CompletableFuture.supplyAsync(() -> {
        try {
            String SQL = "INSERT INTO users (id, name, email, phone, password) VALUES (?, ?, ?, ?, ?)";
            try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                preparedStatement.setInt( parameterIndex: 1, user.getId());
                preparedStatement.setString( parameterIndex: 2, user.getName());
                preparedStatement.setString( parameterIndex: 3, user.getEmail());
                preparedStatement.setString( parameterIndex: 4, user.getPhone());
                preparedStatement.setString( parameterIndex: 5, user.getPassword());

                int rowsInserted = preparedStatement.executeUpdate();
                return rowsInserted > 0;
            } catch (SQLException e) {
                System.out.println("Error inserting user: " + e.getMessage());
                return false;
            }
        } catch (Exception e) {
            System.out.println("Unexpected error: " + e.getMessage());
            return false;
        }
    }, executor);

    try {
        return future.get();
    } catch (InterruptedException | ExecutionException e) {
```

Exception

```java
public class ValidationException extends Exception{  Kovalenko Dmytro
    /**
     * Constructs a new ValidationException with the specified detail message.
     * The message helps to provide more information about the error that occurred during validation.
     *
     * @param message The detailed message that explains the reason for the validation error.
     */
    public ValidationException(String message) {  4 usages  Kovalenko Dmytro
        super(message);
    }
}
```

```java
    private boolean checkName(String name) throws ValidationException {  1 usage  Kovalenko Dmytro
        if (!name.matches( regex: "[a-zA-Z\\s]+")) {
            throw new ValidationException("Your name should not contain numbers or special characters");
        }
        return true;
    }



    /**
     * Checks if the given email is valid.
     * @param email The email to check.
     * @return True if the email is valid, false otherwise.
     */
    private boolean checkEmail(String email) throws ValidationException {  1 usage  Kovalenko Dmytro
        if (!email.matches( regex: "\\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Z|a-z]{2,}\\b")) {
            throw new ValidationException("Invalid email format");
        }
        return true;
    }
    /**
     * Checks if the given password is valid.
     * @param password The password to check.
     * @return True if the password is valid, false otherwise.
     */
    private boolean checkPassword(String password) throws ValidationException {  1 usage  Kovalenko Dmytro
        if (password.length() < 6) {
            throw new ValidationException("Password should be at least 6 characters long");
        }
```

Na kontrolu údajov v registrácii používam ValidationException

```java
public class UnauthorizedAccessException extends Exception{  Kovalenko Dmytro
    /**
     * Constructs a new UnauthorizedAccessException with the specified detail message.
     * The message explains the reason for the error, which helps in understanding what operation was attempted
     * without proper authorization.
     *
     * @param message The detailed message that explains the reason for the unauthorized access attempt.
     */
    public UnauthorizedAccessException(String message) {  1 usage  Kovalenko Dmytro
        super(message);
    }
}
```

```java
public void goToAdminPanelMethod(MouseEvent event) throws UnauthorizedAccessException {  1 usage  Kovalenko Dmytro
    if (user.isAdmin()) {
        System.out.println(user.isAdmin());
        try {
            FXMLLoader loader = new FXMLLoader(getClass().getResource( name: "/com/example/loginpage/profileAdmin.fxm
            Parent root = loader.load();
            ProfileAdminController profileAdminController = loader.getController();
            profileAdminController.setUser(user);
            Scene profileAdminScene = new Scene(root);
            Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
            stage.setScene(profileAdminScene);
            stage.setFullScreen(true);
            stage.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    } else {
        throw new UnauthorizedAccessException("Only Admin has access to this section");
    }
}
```

Výnimka UnauthorizedAccessException sa vyhodí, keď sa bežný používateľ pokúsi vstúpiť do sekcie správcu.

Observer

```java
public interface FavouritesObserver {  6 implementations  Kovalenko Dmytro *

    /**
     * Called when a book is added to the favorites list.
     * Implementers can use this method to take action in response to a book being added,
     * such as updating a user interface, sending notifications, or logging activities.
     *
     * @param book The book that has been added to the favorites list.
     */
    void onFavoriteBookAdded(Book book);  3 usages  2 implementations  Kovalenko Dmytro

    /**
     * Called when a book is removed from the favorites list.
     * This method allows implementers to respond to the event of a book being removed,
     * such as updating displays, adjusting recommendations, or modifying data structures.
     *
     * @param book The book that has been removed from the favorites list.
     */
    void onFavoriteBookRemoved(Book book);  2 usages  2 implementations  Kovalenko Dmytro
}
```

Rozhranie na sledovanie zmien v zozname obľúbených kníh. Implementátori tohto rozhrania môžu dostávať oznámenia o pridaní alebo odstránení kníh zo zoznamu obľúbených

```java
public interface FavouritesObservable {  1 implementation  Kovalenko Dmytro
     *
     * @param observer The {@link FavouritesObserver} to be added to the list. Must not be {@code null}.
     */
    void addObserver(FavouritesObserver observer);  1 usage  1 implementation  Kovalenko Dmytro

    /**
     * Removes an observer from the list of observers. After removal, the observer will no longer receive notifications
     * about changes to the favorites list.
     *
     * @param observer The {@link FavouritesObserver} to be removed from the list. Must not be {@code null}.
     */
    void removeObserver(FavouritesObserver observer);  no usages  1 implementation  Kovalenko Dmytro

    /**
     * Notifies all registered observers that a book has been added to the favorites list. Each observer's
     * {@code onFavoriteBookAdded} method will be called.
     *
     * @param book The book that has been added to the favorites list. This book is not {@code null}.
     */
    void notifyObserversBookAdded(Book book);  1 usage  1 implementation  Kovalenko Dmytro

    /**
     * Notifies all registered observers that a book has been removed from the favorites list. Each observer's
     * {@code onFavoriteBookRemoved} method will be called.
     *
     * @param book The book that has been removed from the favorites list. This book is not {@code null}.
     */
    void notifyObserversBookRemoved(Book book);  1 usage  1 implementation  Kovalenko Dmytro
}
```

Rozhranie pre objekty, ktoré môžu sledovať zmeny v zozname obľúbených kníh. Toto rozhranie definuje metódy na pridávanie a odstraňovanie pozorovateľov, ako aj na oznamovanie pridania alebo odstránenia kníh.

```java
public class FavoritesObserverImpl implements FavouritesObserver {  Kovalenko Dmytro

    /**
     * The data access object (DAO) for books, used to interact with book data storage and manage favorite books.
     */
    private BookDAO bookDAO;  2 usages

    /**
     * Constructs a new FavoritesObserverImpl and registers itself as an observer to the BookDAO.
     * This ensures that this observer is notified of any changes to the favorite books list.
     */
    public FavoritesObserverImpl() {  Kovalenko Dmytro
        this.bookDAO = BookDAO.getInstance();
        this.bookDAO.addObserver(this);
    }

    /**
     * Handles the event of a book being added to the favorites list.
     * This method logs the addition and can be extended to include additional logic such as updating user interfaces,
     * sending notifications, or other book-related handling.
     *
     * @param book The book that has been added to the favorites list. The book is not {@code null}.
     */
    @Override  3 usages  Kovalenko Dmytro
    public void onFavoriteBookAdded(Book book) {
        System.out.println("Book added to favorites: " + book.getTitle());
        // Additional reactions to the book being added, if necessary
    }
```

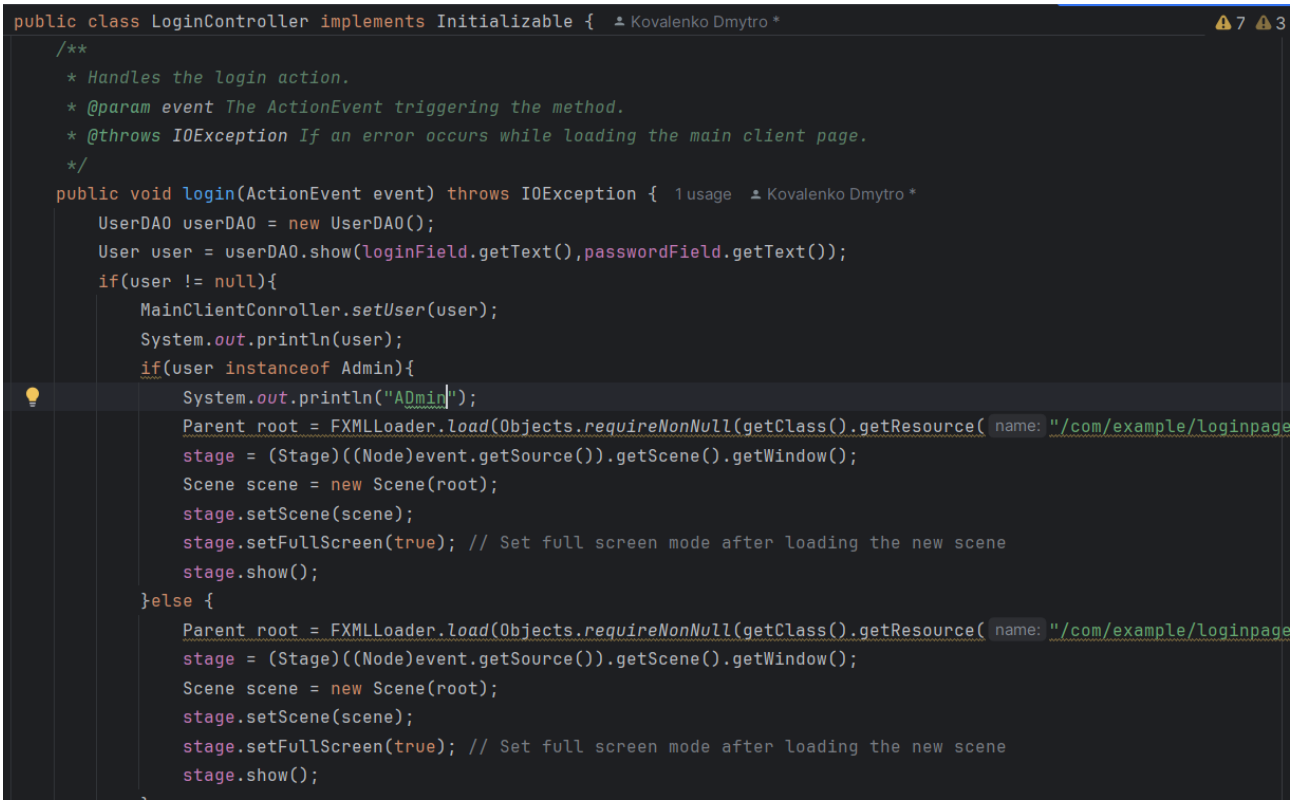
```
    */
    @Override  2 usages  Kovalenko Dmytro
    public void onFavoriteBookRemoved(Book book) {
        System.out.println("Book removed from favorites: " + book.getTitle());
        // Additional reactions to the book being removed, if necessary
    }
}
```

Implementácia rozhrania FavouritesObserver, ktoré spolupracuje s objektom prístupu k údajom (DAO). Táto trieda poskytuje špecifickú odozvu na pridávanie alebo odstraňovanie kníh zo zoznamu obľúbených, napríklad zaznamenávanie týchto udalostí.

BookDAO

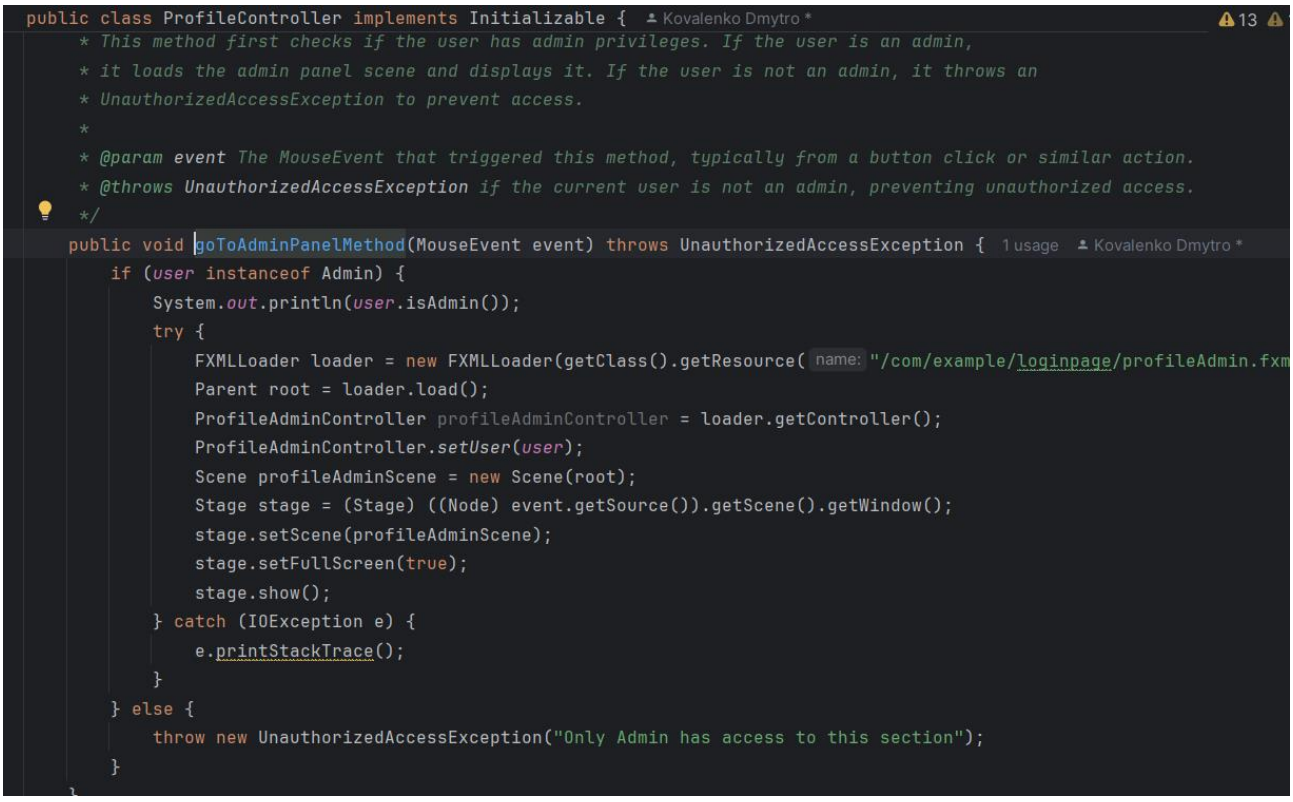
```
@Override  1 usage  Kovalenko Dmytro
public void addObserver(FavouritesObserver observer) { observers.add(observer); }
/**
 * Removes an observer from the list.
 * @param observer The observer to be removed.
 */
@Override  no usages  Kovalenko Dmytro
public void removeObserver(FavouritesObserver observer) { observers.remove(observer); }
/**
 * Notifies all observers about the addition of a new favorite book.
 * @param book The book that was added to favorites.
 */
@Override  1 usage  Kovalenko Dmytro
public void notifyObserversBookAdded(Book book) {
    if (observers.isEmpty()) {
        System.out.println("No observers registered.");
    }
    for (FavouritesObserver observer : observers) {
        System.out.println("Notifying observers about a new favorite book: " + book.getTitle());
        observer.onFavoriteBookAdded(book);
    }
}
/**
 * Notifies all observers about the removal of a favorite book.
 * @param book The book that was removed from favorites.
 */
@Override  1 usage  Kovalenko Dmytro
public void notifyObserversBookRemoved(Book book) {
    if (observers.isEmpty()) {
```

Explicitné používanie RTTI



```
public class LoginController implements Initializable {  Kovalenko Dmytro *
    /**
     * Handles the login action.
     * @param event The ActionEvent triggering the method.
     * @throws IOException If an error occurs while loading the main client page.
     */
    public void login(ActionEvent event) throws IOException {  1 usage  Kovalenko Dmytro *
        UserDAO userDAO = new UserDAO();
        User user = userDAO.show(loginField.getText(),passwordField.getText());
        if(user != null){
            MainClientConroller.setUser(user);
            System.out.println(user);
            if(user instanceof Admin){
                System.out.println("ADmin");
                Parent root = FXMLLoader.load(Objects.requireNonNull(getClass().getResource( name: "/com/example/loginpage
                stage = (Stage)((Node)event.getSource()).getScene().getWindow();
                Scene scene = new Scene(root);
                stage.setScene(scene);
                stage.setFullScreen(true); // Set full screen mode after loading the new scene
                stage.show();
            }else {
                Parent root = FXMLLoader.load(Objects.requireNonNull(getClass().getResource( name: "/com/example/loginpage
                stage = (Stage)((Node)event.getSource()).getScene().getWindow();
                Scene scene = new Scene(root);
                stage.setScene(scene);
                stage.setFullScreen(true); // Set full screen mode after loading the new scene
                stage.show();
```

Používam ho na kontrolu, či je používateľ administrátor.



```
public class ProfileController implements Initializable {  Kovalenko Dmytro *
    * This method first checks if the user has admin privileges. If the user is an admin,
    * it loads the admin panel scene and displays it. If the user is not an admin, it throws an
    * UnauthorizedAccessException to prevent access.
    *
    * @param event The MouseEvent that triggered this method, typically from a button click or similar action.
    * @throws UnauthorizedAccessException if the current user is not an admin, preventing unauthorized access.
    */
    public void goToAdminPanelMethod(MouseEvent event) throws UnauthorizedAccessException {  1 usage  Kovalenko Dmytro *
        if (user instanceof Admin) {
            System.out.println(user.isAdmin());
            try {
                FXMLLoader loader = new FXMLLoader(getClass().getResource( name: "/com/example/loginpage/profileAdmin.fxm
                Parent root = loader.load();
                ProfileAdminController profileAdminController = loader.getController();
                ProfileAdminController.setUser(user);
                Scene profileAdminScene = new Scene(root);
                Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
                stage.setScene(profileAdminScene);
                stage.setFullScreen(true);
                stage.show();
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else {
            throw new UnauthorizedAccessException("Only Admin has access to this section");
        }
```

Používam ho na kontrolu, či je používateľ administrátor, ak áno, môže prejsť do tejto sekcie pomocou panela administrátora a vykonať potrebné údaje.

```java
@Override  Kovalenko Dmytro *
public boolean save(Object entity) {
    if (!(entity instanceof User user)) {
        return false;
    }

    CompletableFuture<Boolean> future = CompletableFuture.supplyAsync(() -> {
        try {
            String SQL = "INSERT INTO users (id, name, email, phone, password) VALUES (?, ?, ?, ?, ?)";
            try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
                preparedStatement.setInt( parameterIndex: 1, user.getId());
                preparedStatement.setString( parameterIndex: 2, user.getName());
                preparedStatement.setString( parameterIndex: 3, user.getEmail());
                preparedStatement.setString( parameterIndex: 4, user.getPhone());
                preparedStatement.setString( parameterIndex: 5, user.getPassword());

                int rowsInserted = preparedStatement.executeUpdate();
                return rowsInserted > 0;
            } catch (SQLException e) {
                System.out.println("Error inserting user: " + e.getMessage());
                return false;
            }
        } catch (Exception e) {
```

Tu som použil na kontrolu, či objekt User skutočne prišiel

```java
@Override  Kovalenko Dmytro
public boolean save(Object entity) {
    if (!(entity instanceof Book book)) {
        return false;
    }

    String SQL = "INSERT INTO books (id, genre, title, author, year, description, votes, imagesrc) VALUES (?,

    try (PreparedStatement preparedStatement = connection.prepareStatement(SQL)) {
        preparedStatement.setInt( parameterIndex: 1, book.getId());
        preparedStatement.setString( parameterIndex: 2, book.getGenre());
        preparedStatement.setString( parameterIndex: 3, book.getTitle());
        preparedStatement.setString( parameterIndex: 4, book.getAuthor());
        preparedStatement.setInt( parameterIndex: 5, book.getYear());
        preparedStatement.setString( parameterIndex: 6, book.getDescription());
        preparedStatement.setInt( parameterIndex: 7, book.getVotes());
        preparedStatement.setString( parameterIndex: 8, book.getImageSrc());

        int affectedRows = preparedStatement.executeUpdate();
        return affectedRows > 0;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
```

To isté

Sealed Class

```java
abstract sealed public class Book implements Serializable permits FictionBook, FamilyBook, DramaBook, Autobiography, AllegoricalBook,AdventureBook {
    private static final long serialVersionUID = 1L; // Recommended to include  no usages
    /** The title of the book. */
    private String title;  5 usages
    /** The author of the book. */
    private String author;  5 usages
    /** The publication year of the book. */
    private int year;  5 usages
    /** The genre of the book. */
    private String genre;  5 usages
    /** A brief description of the book. */
    private String description;  5 usages
    /** The URL of the image representing the book cover. */
    private String imageSrc;  3 usages
    /** The number of votes or ratings the book has received. */
    private int votes;  3 usages
    /** The unique identifier for the book. */
    private int id;  4 usages

    // Static variable to track the count of books created.
    private static int COUNT_OF_BOOKS = 1;  3 usages
    {
        id = COUNT_OF_BOOKS + 1;
    }
    /**
     * Default constructor for creating a book object without initializing fields.
     */
    public Book(){}  Kovalenko Dmytro
```

```java
public non-sealed class AdventureBook extends Book {  Kovalenko Dmytro
    /**
     * The name of the main character in the adventure book, around whom the plot typically revo
     */
    private String mainCharacter;  5 usages

    /**
     * Constructs a new AdventureBook with detailed book information, including the main charact
     * Initializes all fields with the provided values, enhancing the thematic focus on adventur
     *
     * @param id The unique identifier for the book.
     * @param title The title of the book.
     * @param author The author of the book.
     * @param year The publication year of the book.
     * @param genre The genre of the book, typically set to "Adventure".
     * @param description A brief description of the book, emphasizing the adventurous elements.
     * @param imageSrc A URL or path to an image of the book's cover.
     * @param votes The number of votes or ratings the book has received.
     * @param mainCharacter The name of the main character central to the adventure.
     */
    public AdventureBook(int id, String title, String author, int year, String genre, String des
        super(id, title, author, year, genre, description, imageSrc, votes);
        this.mainCharacter = mainCharacter;
    }
```

Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them.

Diagram