ORDERS MANAGEMENT

DOCUMENTATIE - TEMA 3 - TP

PĂCURAR IRINA
UTCN, CTI, SERIA A, GRUPA 30223
ÎNDRUMĂTORI: PROF. DR. ING. TUDOR CIOARĂ, OVIDIU MAJA
- mai 2024 -

Cuprins

1. Obiectivul temei 4

2. Analiza problemei, modelare, scenarii, cazuri de utilizare 5

2.1.1. Analiza Problemei 5

2.1.2. Modelare 5

2.1.3. Scenarii 5

2.1.4. Cazuri de Utilizare 5

3. Proiectare 0

4. Implementare pachete 1

4.1. Org.example – Main 1

4.2. Model 2

4.2.1. Bill 2

4.2.2. Client 2

4.2.3. Order 4

4.2.4. Product 5

4.3. Bll 6

4.3.1. Validators 6

4.3.2. BillBll 9

4.3.3. ClientBll 10

4.3.4. OrderBll 11

4.3.5. ProductBll 11

4.4. Connection – ConnectionFactory 12

4.5. Dao 13

4.5.1. BillDao 13

4.5.2. ClientDao 14

4.5.3. GenericDao 14

4.5.4. OrderDao 15

4.5.5. ProductDao 16

4.6. Presentation 16

4.6.1. BillController 16

4.6.2. BillView 17

4.6.3. ClientController 17
4.6.4. ClientView 18
4.6.5. MainView 19
4.6.6. OrderController 19
4.6.7. OrderView 20
4.6.8. ProductController 21
4.6.9. ProductView 22
4.6.10. TableModelUtils 24
5. Rezultate 25
6. Concluzii și dezvoltări ulterioare 25
7. Bibliografie 26

1. Obiectivul temei

Proiectul solicită dezvoltarea unei aplicații de management al comenzilor pentru un depozit, care urmează arhitectura pe straturi și utilizează baze de date relaționale. Structura aplicației include patru tipuri principale de clase: modele de date care reprezintă structurile de date ale aplicației, clase de logică de afaceri pentru gestionarea operațiunilor aplicației, clase de prezentare pentru interfața grafică a utilizatorului și clase de acces la date pentru interacțiunea cu baza de date.

Sunt necesare: implementarea unui design orientat-obiect cu respectarea limitelor de lungime pentru clase și metode, utilizarea adecvată a JavaDoc pentru documentarea claselor, implementarea unei baze de date cu cel puțin trei tabele (Client, Produs, și Comandă), crearea unei interfețe grafice care permite gestionarea clienților și produselor, precum și plasarea comenzilor. Se solicită folosirea tehnicilor de reflecție pentru generarea dinamică a interfețelor și a query-urilor în baza de date, precum și definirea unei clase imutabile Bill în pachetul de model, utilizând înregistrările Java, pentru reprezentarea facturilor stocate într-o tabelă Log dedicată, care permite doar inserții și citiri.

2. *Analiza problemei, modelare, scenarii, cazuri de utilizare*

Pentru aplicația de gestionare a comenzilor într-un depozit, care folosește baze de date relaționale și arhitectura pe straturi, este crucial să se efectueze o analiză detaliată a problemei, modelarea sistemului, și să se definească scenariile și cazurile de utilizare relevante.

2.1.1. Analiza Problemei

Aplicația trebuie să permită gestionarea eficientă a clienților, produselor și comenzilor. Trebuie să suporte operațiuni cum ar fi adăugarea, editarea și ștergerea datelor clienților și produselor, și procesarea comenzilor. Un aspect cheie este gestionarea stocurilor, aplicația trebuie să verifice disponibilitatea produselor și să actualizeze stocul după fiecare comandă. Toate aceste date trebuie să fie stocate într-o bază de date relațională, organizată în cel puțin trei tabele principale: Client, Produs și Comandă.

2.1.2. Modelare

Modelul de date include trei entități principale:

- Client: Conține informații despre clienții depozitului.
- Produs: Deține detalii despre produsele disponibile în depozit, inclusiv stocul.
- Comandă: Asociază clienții și produsele cu cantitățile comandate.

De asemenea, va exista o tabelă Bill pentru a stoca facturi imutabile generate automat la fiecare comandă, utilizând recorduri Java.

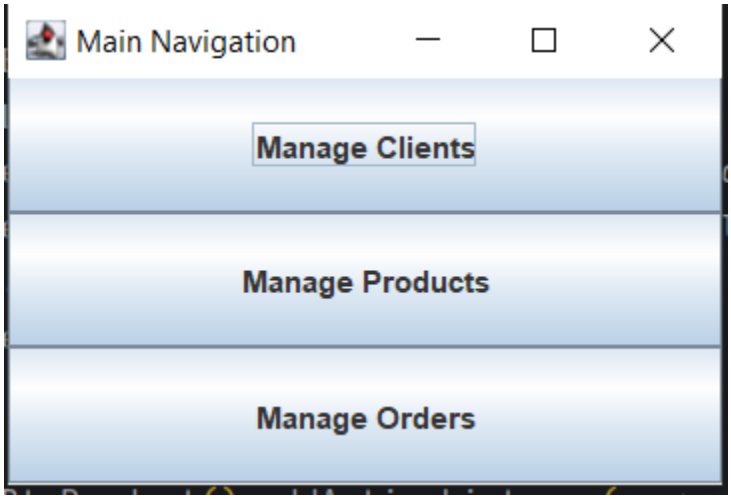
2.1.3. Scenarii

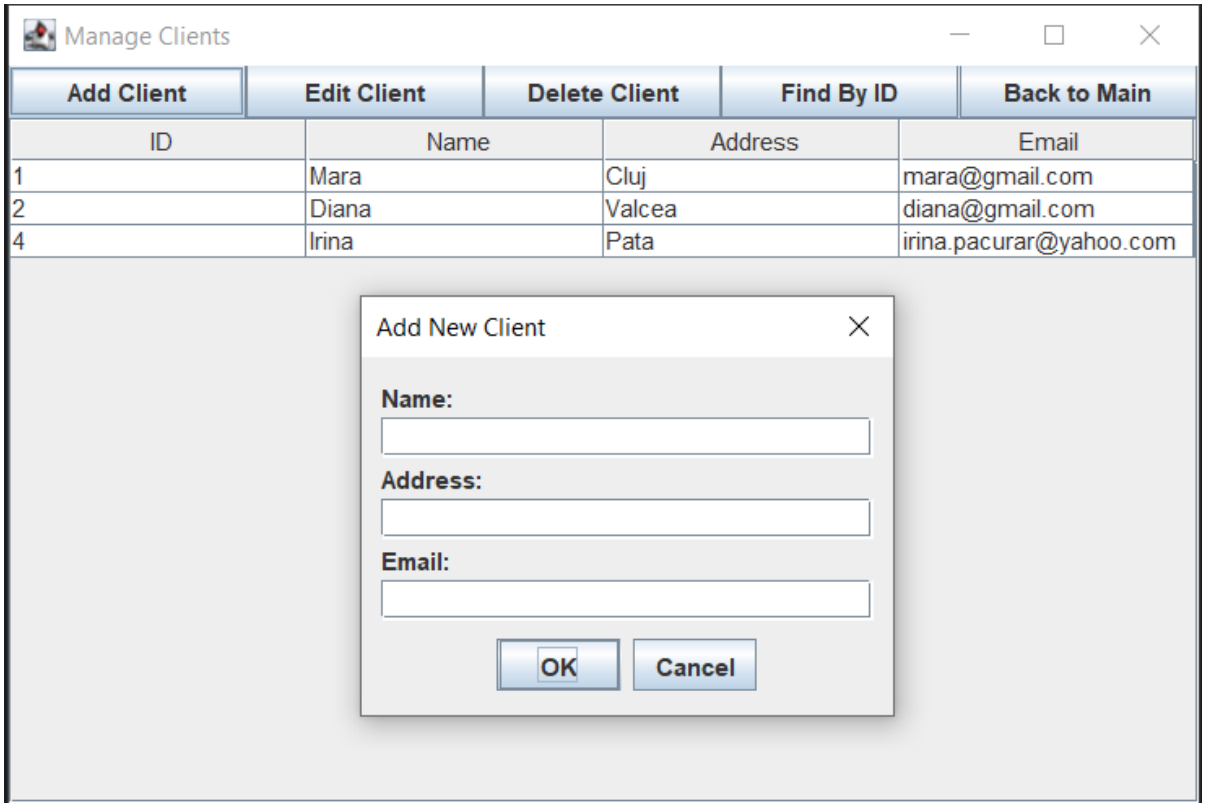
Un scenariu tipic implică un utilizator al aplicației (angajat al depozitului sau client direct) care accesează interfața grafică pentru a vizualiza, adăuga, sau modifica informații despre produse sau clienți. De exemplu, un utilizator poate dori să adauge un nou produs în stoc, să actualizeze informațiile unui client, sau să plaseze o comandă pentru un client existent.

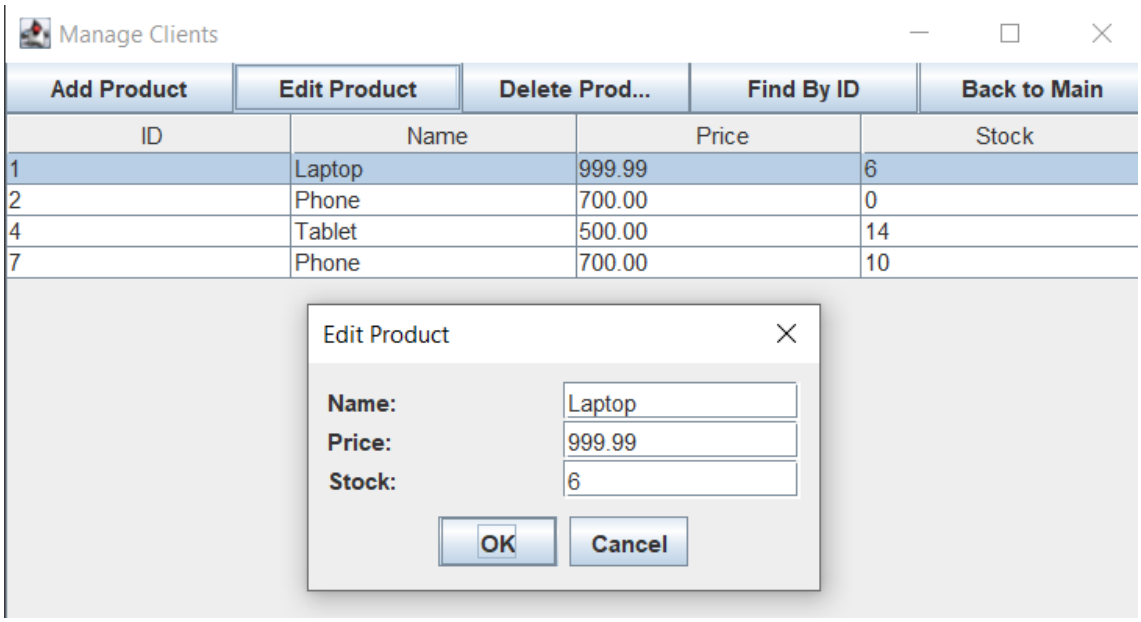
2.1.4. Cazuri de Utilizare

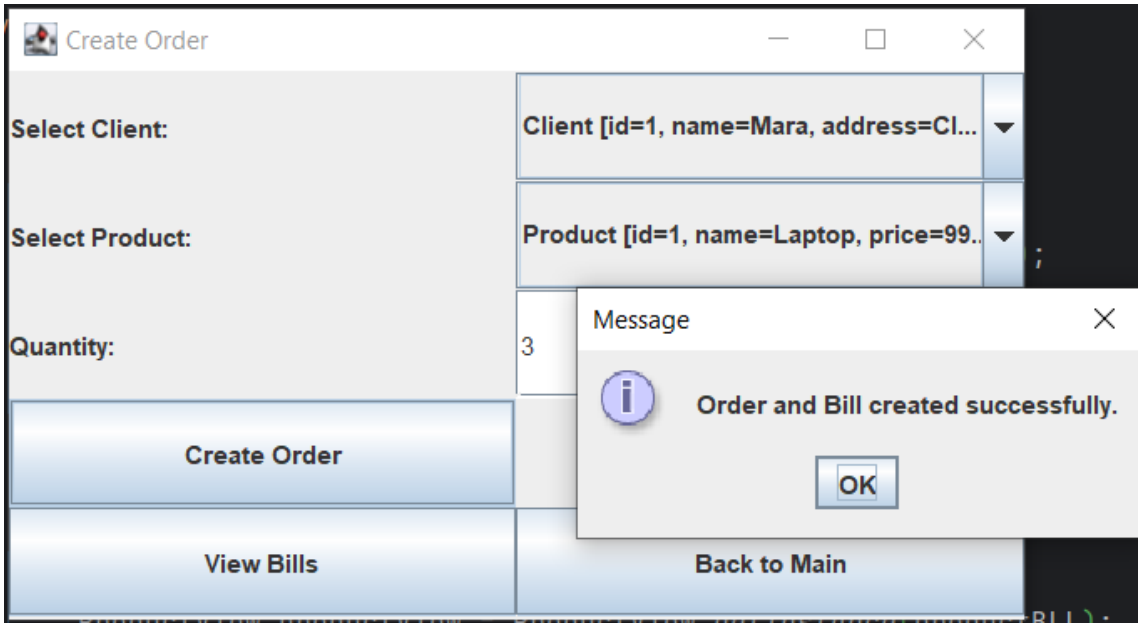
1. Adăugarea unui nou client: Utilizatorul introduce detaliile unui nou client într-un formular și le salvează în baza de date.
2. Editarea detaliilor unui client: Utilizatorul selectează un client dintr-o listă, modifică detaliile necesare și actualizează înregistrarea în baza de date.
3. Ștergerea unui client: Utilizatorul selectează un client și îl șterge din sistem.
4. Adăugarea unui nou produs: Detaliile produsului sunt introduse și salvate în baza de date.
5. Editarea unui produs: Modificările asupra unui produs existent sunt făcute prin interfața de editare.
6. Plasarea unei comenzi: Utilizatorul selectează un produs și un client, introduce cantitatea dorită și plasează comanda. Dacă stocul este insuficient, se afișează un mesaj de avertizare.
7. Generarea unei facturi: Odată ce comanda este finalizată, se generează automat o factură imutabilă, care este stocată în tabelul Bill.

Fiecare caz de utilizare implică interacțiuni directe cu baza de date prin intermediul claselor de acces la date, care folosesc tehnici de reflecție pentru a dinamiza crearea și interogarea datelor. Utilizarea unei arhitecturi pe straturi și a principiilor de programare orientată obiect asigură că aplicația este bine organizată și ușor de întreținut.







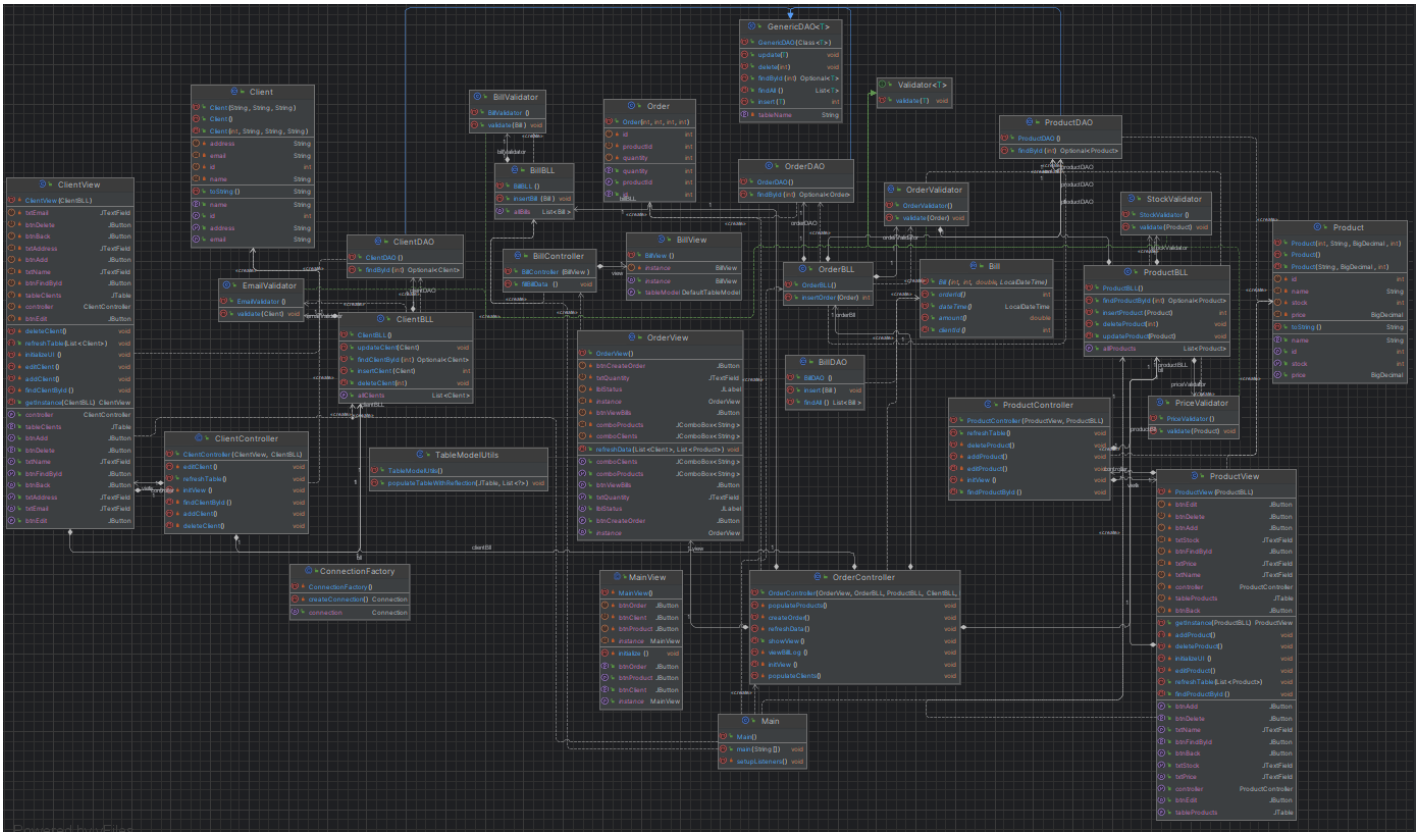


Bill Log

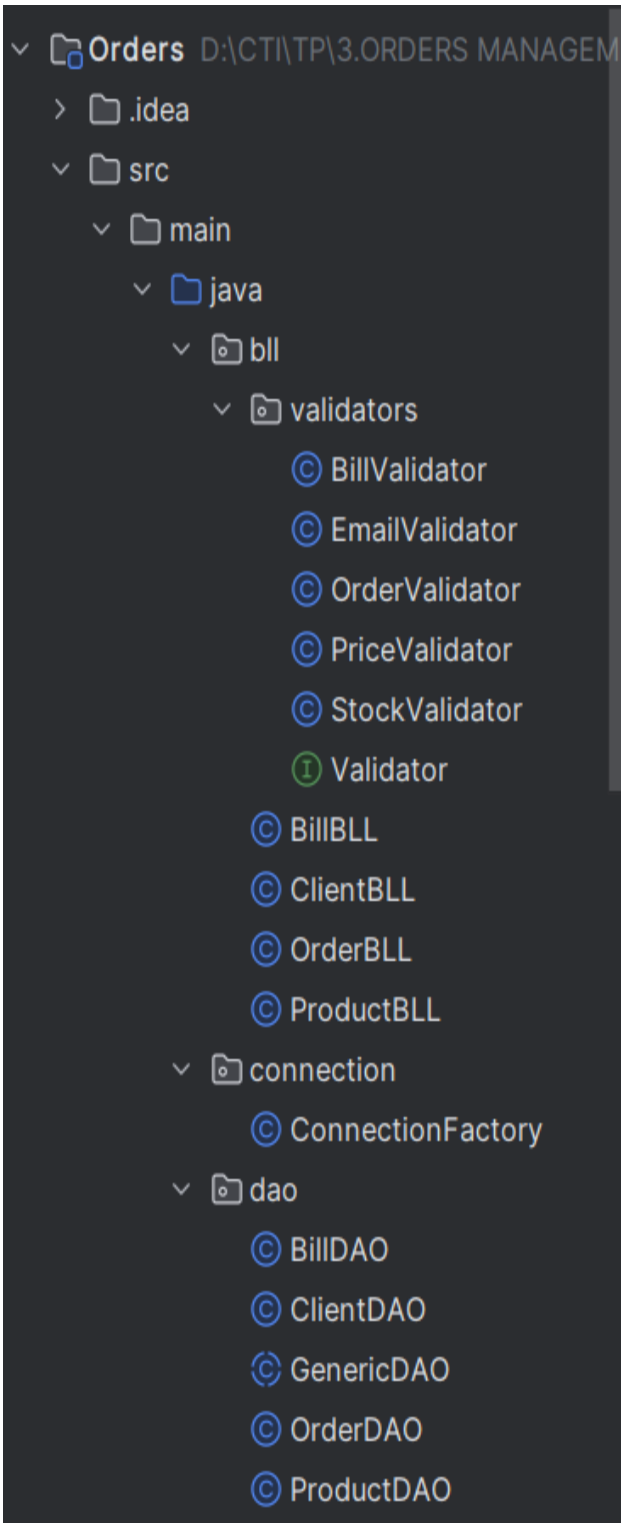
Order ID	Client ID	Amount	Date/Time
20	2	2100.0	2024-05-03T21:04:13
22	4	1000.0	2024-05-03T21:14:52
23	2	700.0	2024-05-04T00:25:47
24	2	700.0	2024-05-04T02:26:27
25	1	1999.98	2024-05-04T02:27:21
29	2	999.99	2024-05-13T01:48:43
30	1	2999.97	2024-05-13T03:54:44

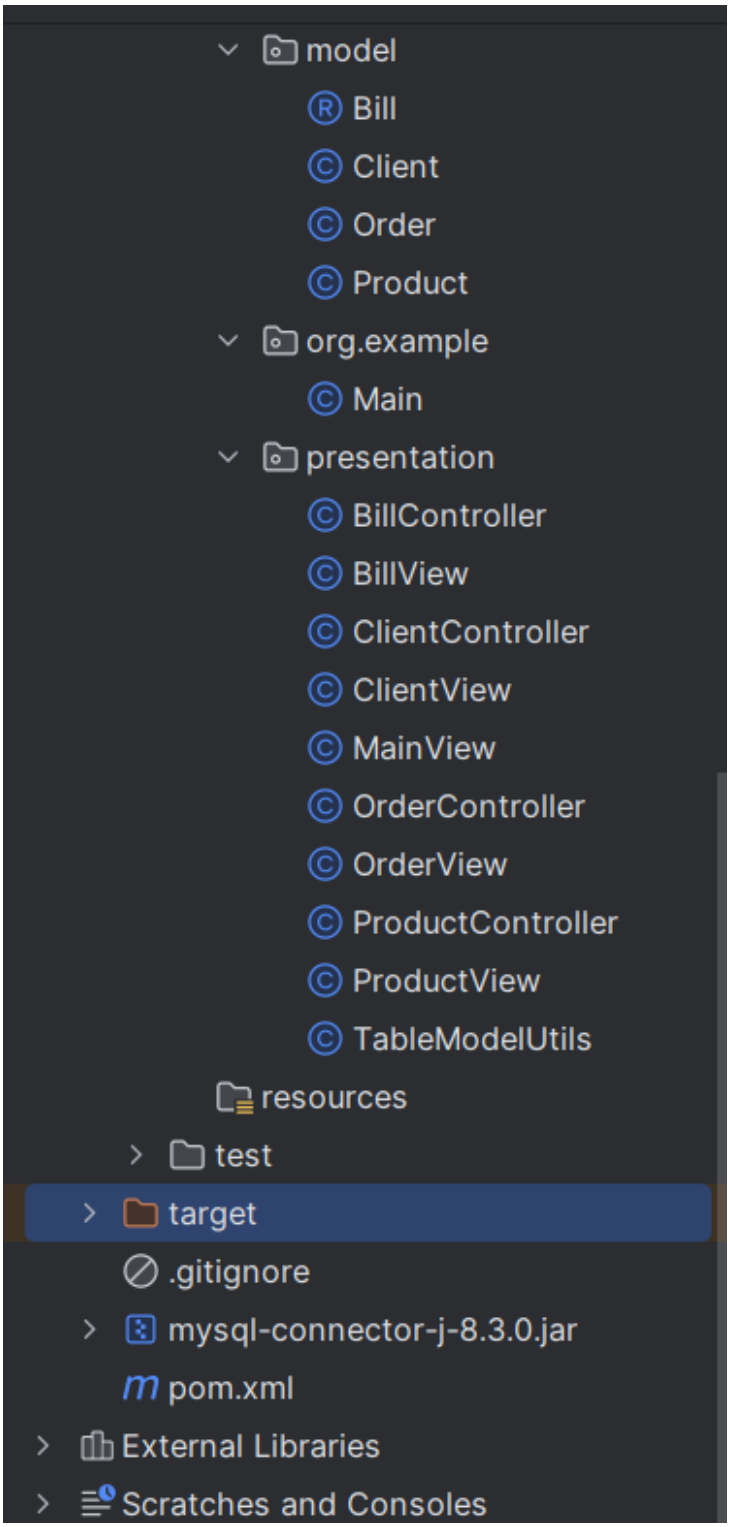
3. Proiectare

Clasele



Pachetele


Orders D:\CTI\TP\3.ORDERS MANAGEM
.idea
src
main
java
bll
validators
BillValidator
EmailValidator
OrderValidator
PriceValidator
StockValidator
Validator
BillBLL
ClientBLL
OrderBLL
ProductBLL
connection
ConnectionFactory
dao
BillDAO
ClientDAO
GenericDAO
OrderDAO
ProductDAO


model
Bill
Client
Order
Product
org.example
Main
presentation
BillController
BillView
ClientController
ClientView
MainView
OrderController
OrderView
ProductController
ProductView
TableModelUtils
resources
test
target
.gitignore
mysql-connector-j-8.3.0.jar
pom.xml
External Libraries
Scratches and Consoles

4. *Implementare pachete*

4.1. Org.example – Main

Clasa **Main** din pachetul **org.example** servește drept punct de intrare în aplicația de gestionare a comenzilor. Aceasta conține metoda **main**, care este metoda standard de lansare pentru orice aplicație Java.

```java
/**
 * The main class of the application that initializes the GUI and sets up navigation between different sections.
 * This class is responsible for launching the application and configuring the interactions between the main view
 * and other components of the application such as client, product, and order views.
 */
public class Main {
    no usages
    protected static final Logger LOGGER = Logger.getLogger(Main.class.getName());
    /**
     * Main entry point of the application. This method sets up the main GUI to be visible and initializes
     * navigation listeners to facilitate transitions between different views.
     * @param args Command line arguments passed to the program.
     */
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(() -> {
            MainView.getInstance().setVisible(true);
            setupListeners();
        });
    }
    /**
     * Sets up the action listeners for buttons in the MainView. Each button is configured to initialize
     * and display its corresponding view, managing the transitions and data refresh operations for
     * client, product, and order management.
     */
    1 usage
    private static void setupListeners() {...}
}
```

4.2. Model

4.2.1. Bill

Modelul pentru facturi.

```java
import java.time.LocalDateTime;
/**
 * Represents a billing record, encapsulating details of a transaction.
 * Records in Java are a concise way to create immutable data-only classes.
 *
 * This record holds all necessary details about a bill, including identifiers for the order and client,
 * the transaction amount, and the precise time the transaction was recorded.
 *
 * @param orderId The unique identifier of the order associated with this bill.
 * @param clientId The unique identifier of the client associated with this bill.
 * @param amount The monetary amount charged in this bill.
 * @param dateTime The timestamp of when the bill was issued.
 */
18 usages
public record Bill(int orderId, int clientId, double amount, LocalDateTime dateTime) {
}
```

4.2.2. Client

Modelul pentru clienți.

```java
/**
 * Represents a client in the system, encapsulating key details about a client such as name, address, and email.
 * This class provides constructors for creating client instances in various contexts within the application,
 * and methods for retrieving and updating client information.
 */
public class Client {
    4 usages
    private int id;
    4 usages
    private String name;
    4 usages
    private String address;
    4 usages
    private String email;
    /**
     * Constructs a new Client with specified id, name, address, and email.
     * This constructor is typically used when retrieving existing client information from a data source
     * where the client's ID is already defined.
     *
     * @param id the unique identifier for the client
     * @param name the name of the client
     * @param address the physical address of the client
     * @param email the email address of the client
     */
    public Client(int id, String name, String address, String email) {...}
    /**
     * Default constructor for creating an instance of Client without setting properties initially.
     * This constructor is typically used in situations where the client's properties will be set
     * post-creation using setter methods.
     */
```

```java
public Client() {}
/**
 * Constructs a new Client with specified name, address, and email but without an id.
 * This constructor is typically used when creating new clients whose ID will be assigned automatically by the database.
 * @param name the name of the client
 * @param address the physical address of the client
 * @param email the email address of the client
 */
public Client(String name, String address, String email) {...}
/**
 * Returns the ID of the client.
 * @return the unique identifier for the client
 */
public int getId() { return id; }
/**
 * Sets the ID of the client.
 * @param id the unique identifier to be set for the client
 */
public void setId(int id) { this.id = id; }
/**
 * Returns the name of the client.
 * @return the client's name
 */
public String getName() { return name; }
/**
 * Returns the address of the client.
 * @return the client's physical address
 */
1 usage
public String getAddress() { return address; }
```

```java
/**
 * Returns the email address of the client.
 * @return the client's email address
 */
2 usages
public String getEmail() {
    return email;
}
/**
 * Returns a string representation of the client, typically used for logging and debugging purposes.
 * @return a string description of the client
 */
@Override
public String toString() {...}
```

4.2.3. Order

Modelul pentru comenzi.

```java
/**
 * Represents an order within the system, encapsulating all relevant details such as order ID, client ID,
 * product ID, and quantity. This class is designed to handle the creation and management of order records.
 */
14 usages
public class Order {
    3 usages
    private int id;
    1 usage
    private int clientId;
    2 usages
    private int productId;
    2 usages
    private int quantity;
    /**
     * Constructs a new Order with specified details.
     * This constructor is typically used when retrieving existing order data from a data source
     * or creating a new order with known attributes.
     * @param id the unique identifier of the order
     * @param clientId the unique identifier of the client who made the order
     * @param productId the unique identifier of the product ordered
     * @param quantity the amount of the product ordered
     */
    2 usages
    public Order(int id, int clientId, int productId, int quantity) {...}
    /**
     * Returns the order ID.
     * @return the unique identifier for the order
     */
    public int getId() { return id; }
```

```java
    /**
     * Sets the order ID.
     * @param id the unique identifier to be set for the order
     */
    public void setId(int id) { this.id = id; }
    /**
     * Returns the product ID for the order.
     *
     * @return the unique identifier of the product
     */
    4 usages
    public int getProductId() {...}
    /**
     * Returns the quantity of the product ordered.
     * @return the quantity of the product
     */
    3 usages
    public int getQuantity() {...}
```

4.2.4. Product

Modelul pentru produse.

```java
/**
 * Represents a product in the system, encapsulating essential details such as product ID, name, price, and stock levels.
 * This class provides constructors for creating product instances for various use cases, including database retrieval
 * and new product creation.
 */
public class Product {
    4 usages
    private int id;
    4 usages
    private String name;
    4 usages
    private BigDecimal price;
    5 usages
    private int stock;
    /**
     * Default constructor for creating an instance of Product.
     * This constructor is typically used in situations where product properties are set after creation using setters.
     */
    public Product() {
    }
    /**
     * Constructs a new Product with specified id, name, price, and stock level.
     * This constructor is primarily used for retrieving existing products from a data source where the product ID is known.
     * @param id the unique identifier for the product
     * @param name the name of the product
     * @param price the price of the product, encapsulated in a {@link BigDecimal} for precision
     * @param stock the stock level of the product
     */
    public Product(int id, String name, BigDecimal price, int stock) {...}
```

```java
    /**
     * Constructs a new Product with specified name, price, and stock level but without an id.
     * This constructor is used for creating new products where the ID will be assigned automatically by the database.
     * @param name the name of the product
     * @param price the price of the product, encapsulated in a {@link BigDecimal} for precision
     * @param stock the stock level of the product
     */
    public Product(String name, BigDecimal price, int stock) {...}
    /**
     * Returns the ID of the product.
     * @return the unique identifier for the product
     */
    public int getId() { return id; }
    /**
     * Sets the ID of the product.
     * @param id the unique identifier to be set for the product
     */
    public void setId(int id) { this.id = id; }
    /**
     * Returns the name of the product.
     * @return the product's name
     */
    public String getName() { return name; }
    /**
     * Returns the price of the product.
     * @return the product's price, represented as a {@link BigDecimal}
     */
    3 usages
    public BigDecimal getPrice() { return price; }
```

```java
/**
 * Returns the stock level of the product.
 * @return the current stock level
 */
6 usages
public int getStock() { return stock; }
/**
 * Sets the stock level of the product.
 * @param stock the stock level to be set
 */
1 usage
public void setStock(int stock) { this.stock = stock; }
/**
 * Provides a string representation of the product, typically used for logging and display purposes.
 * @return a string description of the product
 */
@Override
public String toString() {...}
```

4.3. Bll

4.3.1. Validators

4.3.1.1. BillValidator

Validare specifică pentru facturi.

```java
import model.Bill;
/**
 * The {@code BillValidator} class is responsible for validating {@link Bill} instances
 * before they are processed in the business logic layer. This validation ensures that
 * all required attributes of a bill meet specific requirements to maintain data integrity
 * and business rules.
 */
3 usages
public class BillValidator implements Validator<Bill> {
    /**
     * Validates the specified {@code Bill} object to ensure its compliance with
     * the system's requirements.
     *
     * @param bill the {@code Bill} object to validate.
     * @throws IllegalArgumentException if any attribute of {@code Bill} does not meet the
     *         required validation rules:
     *         <ul>
     *             <li>The {@code Bill} object itself must not be null.</li>
     *             <li>The amount of the bill must be greater than zero to ensure
     *             economic validity.</li>
     *             <li>The order ID and client ID must be positive to ensure they
     *             refer to valid records in the database.</li>
     *         </ul>
     */
    9 usages
    @Override
    public void validate(Bill bill) {...}
}
```

4.3.1.2. EmailValidator

Validare pentru adresele de email.

```java
package bll.validators;

import ...
/**
 * Validates the email address of a {@code Client} object using a complex regular expression pattern.
 * This validator ensures that the email address adheres to the specific criteria defined by the pattern
 * and throws an exception if the email address is invalid.
 *
 * <p>The regular expression pattern covers a variety of email formats, including checks for
 * proper characters, symbols, domain names, and specific structures necessary for a valid email address.
 */
3 usages
public class EmailValidator implements Validator<Client> {
    /**
     * Regular expression pattern for validating email.
     * This pattern incorporates extensive checks for valid email address formats including the presence
     * of appropriate characters, acceptable symbols, and proper structure as per common email validation standards.
     */
    1 usage
    private static final String EMAIL_PATTERN = "(?:(?:\\r\\n)?[ \\t])*(?:(?:(?:[^()<>@,;:\\\\\".\\[\\] \\000-\\031]+(?:(?:
    /**
     * Validates the email address of the given client.
     *
     * @param t the client whose email needs to be validated
     * @throws IllegalArgumentException if the email does not match the pattern
     */
    9 usages
    public void validate(Client t) {...}
}
```

4.3.1.3. OrderValidator

Validare pentru comenzile plasate.

```java
import ...
/**
 * Validates an order against product availability in the database.
 * This validator checks if the product associated with the order exists and if there are sufficient quantities
 * in stock to fulfill the order. It uses {@code ProductDAO} to retrieve product details.
 */
3 usages
public class OrderValidator implements Validator<Order> {
    2 usages
    private ProductDAO productDAO;
    /**
     * Constructs an {@code OrderValidator} with a new instance of {@code ProductDAO}.
     * This setup allows access to the product database necessary for validating orders.
     */
    1 usage
    public OrderValidator() { this.productDAO = new ProductDAO(); }
    /**
     * Validates the specified order to ensure the product exists and the ordered quantity is available.
     *
     * @param order the order to be validated
     * @throws NoSuchElementException if no product with the specified ID is found in the database
     * @throws IllegalArgumentException if the ordered quantity exceeds the product's available stock
     */
    9 usages
    @Override
    public void validate(Order order) {...}
}
```

4.3.1.4. PriceValidator

Validare pentru prețurile produselor.

```java
import ...
/**
 * Validates the price of a {@code Product} to ensure it is greater than zero.
 * This validator checks that the product price is not negative or zero, throwing an exception
 * if the price does not meet the specified criteria. This is essential for ensuring that
 * all products have a valid price before being processed or saved in the database.
 */
3 usages
public class PriceValidator implements Validator<Product> {
    /**
     * Validates the price of the given product.
     * <p>
     * This method checks if the price of the product is greater than zero. It uses {@link BigDecimal}
     * for comparison to handle high precision floating-point operations and monetary values.
     *
     * @param product the product to be validated
     * @throws IllegalArgumentException if the product's price is less than or equal to zero
     */
    9 usages
    @Override
    public void validate(Product product) {...}
}
```

4.3.1.5. StockValidator

Validare pentru stocurile de produse.

```java
import model.Product;
/**
 * Validates the stock quantity of a {@code Product} to ensure it is non-negative.
 * This validator checks that the stock is not below zero, enforcing that all products have
 * a valid stock count before processing or saving in the database.
 */
3 usages
public class StockValidator implements Validator<Product> {
    /**
     * Validates the stock quantity of the given product.
     * <p>
     * This method checks if the stock count of the product is negative. It throws an
     * {@code IllegalArgumentException} if the stock is less than zero to ensure data integrity
     * and prevent logical errors in inventory management.
     *
     * @param product the product whose stock is to be validated
     * @throws IllegalArgumentException if the product's stock is negative
     */
    9 usages
    @Override
    public void validate(Product product) {...}
}
```

4.3.1.6. Validator

Interfață pentru validatoare.

```java
package bll.validators;
/**
 * Defines a validation contract for objects of type T.
 * Implementations of this interface are responsible for validating objects according to specific rules.
 * If the object does not meet the validation criteria, the implementation should indicate this
 * through an exception or by other means as per the design of the specific validator.
 *
 * @param <T> the type of objects this validator can handle
 */
5 usages  5 implementations
public interface Validator<T> {
    /**
     * Validates an object of type T.
     * <p>
     * Implementing classes should define specific validation logic to ensure that the object
     * complies with the necessary criteria. If the object fails validation, this method should
     * throw an appropriate runtime exception or handle the error according to the application's needs.
     *
     * @param t the object to be validated
     * @throws RuntimeException if the object does not meet the validation criteria
     */
    9 usages  5 implementations
    public void validate(T t);
}
```

4.3.2. BillBll

Logică de afaceri pentru gestionarea facturilor.

```java
import ...
/**
 * Handles business logic for bill operations including insertion and retrieval from a database.
 * This class ensures that all bill transactions meet business rules through validation before
 * proceeding with database operations.
 */
7 usages
public class BillBLL {
    2 usages
    private final BillValidator billValidator;
    /**
     * Constructs a {@code BillBLL} object, initializing the necessary validators.
     */
    2 usages
    public BillBLL() { this.billValidator = new BillValidator(); }
    /**
     * Inserts a bill into the database after performing necessary validations.
     * @param bill The bill to be inserted.
     * @throws IllegalArgumentException If the bill amount is not positive.
     */
    1 usage
    public void insertBill(Bill bill) {...}
    /**
     * Retrieves all bills from the database.
     * @return A list of all bills.
     */
    1 usage
    public List<Bill> getAllBills() { return BillDAO.findAll(); }
}
```

4.3.3. *ClientBll*

Logică de afaceri pentru gestionarea clienților.

```java
import ...
/**
 * Handles business logic related to client operations including search, insertion, update, and deletion.
 * Ensures all client operations are validated, particularly the client's email, before proceeding with database transactions.
 */
18 usages
public class ClientBLL {
    6 usages
    private ClientDAO clientDAO;
    2 usages
    private EmailValidator emailValidator = new EmailValidator();
    /**
     * Constructs a {@code ClientBLL} object, initializing the necessary data access object (DAO) for client operations.
     */
    5 usages
    public ClientBLL() { this.clientDAO = new ClientDAO(); }
    /**
     * Retrieves a client by their ID from the database.
     * If no client is found, the result will be an empty {@code Optional}.
     * @param id the unique identifier of the client
     * @return an {@code Optional} containing the client if found, or an empty {@code Optional} if not found
     */
    2 usages
    public Optional<Client> findClientById(int id) { return clientDAO.findById(id); }
```

```java
    /**
     * Inserts a new client into the database after validating the client's email address.
     * Throws {@code IllegalArgumentException} if the email is invalid.
     * @param client the client to be inserted into the database
     * @return the generated ID of the inserted client
     * @throws IllegalArgumentException if the email validation fails
     */
    2 usages
    public int insertClient(Client client) {...}
    /**
     * Updates an existing client's information in the database after validating the client's email.
     * Throws {@code IllegalArgumentException} if the email is invalid.
     * @param client the client whose information is to be updated
     * @throws IllegalArgumentException if the email validation fails
     */
    2 usages
    public void updateClient(Client client) {...}
    /**
     * Deletes a client from the database using the client's ID.
     * @param id the unique identifier of the client to be deleted
     */
    2 usages
    public void deleteClient(int id) { clientDAO.delete(id); }
    /**
     * Retrieves all clients from the database.
     * @return a list of all clients
     */
    8 usages
    public List<Client> getAllClients() { return clientDAO.findAll(); }
}
```

4.3.4. *OrderBll*

Logică de afaceri pentru procesarea comenzilor.

```java
import ...
/**
 * Handles business logic for order operations, including insertion, updating, and retrieval.
 * This class ensures that all order-related actions are validated and that product inventory is adjusted accordingly.
 */
5 usages
public class OrderBLL {
    2 usages
    private OrderDAO orderDAO;
    3 usages
    private ProductDAO productDAO;
    1 usage
    private OrderValidator orderValidator = new OrderValidator();
    /**
     * Constructs an {@code OrderBLL} object, initializing the necessary DAOs for order and product operations.
     */
    1 usage
    public OrderBLL() {...}
    /**
     * Inserts an order into the database after validating it and ensuring sufficient product stock.
     * The stock is updated to reflect the order quantity if the order is successfully inserted.
     *
     * @param order the order to be inserted
     * @return the generated ID of the newly inserted order
     * @throws NoSuchElementException if the product associated with the order is not found
     * @throws IllegalArgumentException if there is insufficient stock for the order
     */
    1 usage
    public int insertOrder(Order order) {...}
}
```

4.3.5. *ProductBll*

Logică de afaceri pentru gestionarea produselor.

```java
import ...
/**
 * Handles business logic for product operations, including search, insertion, update, and deletion of products.
 * This class ensures that all product-related actions are preceded by validations for price and stock,
 * ensuring that only valid products are processed and stored in the database.
 */
17 usages
public class ProductBLL {
    6 usages
    private ProductDAO productDAO;
    2 usages
    private PriceValidator priceValidator = new PriceValidator();
    2 usages
    private StockValidator stockValidator = new StockValidator();
    /**
     * Constructs a {@code ProductBLL} object, initializing the necessary data access object (DAO) for product operations.
     */
    5 usages
    public ProductBLL() { productDAO = new ProductDAO(); }
    /**
     * Retrieves a product by its ID from the database.
     * If no product is found, the result will be an empty {@code Optional}.
     *
     * @param id the unique identifier of the product
     * @return an {@code Optional} containing the product if found, or an empty {@code Optional} if not found
     */
    2 usages
    public Optional<Product> findProductById(int id) { return productDAO.findById(id); }
```

```
    /**
     * Inserts a new product into the database after validating its price and stock.
     * Validation ensures that the product has a non-negative stock and a price greater than zero.
     * @param product the product to be inserted into the database
     * @return the generated ID of the newly inserted product
     * @throws IllegalArgumentException if the product fails price or stock validation
     */
    2 usages
    public int insertProduct(Product product) {...}
    /**
     * Inserts a new product into the database after validating its price and stock.
     * Validation ensures that the product has a non-negative stock and a price greater than zero.
     * @param product the product to be inserted into the database
     * @return the generated ID of the newly inserted product
     * @throws IllegalArgumentException if the product fails price or stock validation
     */
    2 usages
    public void updateProduct(Product product) {...}
    /**
     * Deletes a product from the database using the product's ID.
     * @param id the unique identifier of the product to be deleted
     */
    2 usages
    public void deleteProduct(int id) { productDAO.delete(id); }
    /**
     * Retrieves all products currently stored in the database.
     * @return a list of all products
     */
    8 usages
    public List<Product> getAllProducts() { return productDAO.findAll(); }
}
```

4.4. Connection – ConnectionFactory

Clasa responsabilă pentru stabilirea conexiunii la baza de date.

```
import ...
/**
 * Manages the creation of database connections using the singleton design pattern.
 * This class ensures that all connections are created using the specified driver,
 * URL, user credentials, and handles any exceptions related to database connectivity.
 */
18 usages
public class ConnectionFactory {

    1 usage
    private static final Logger LOGGER = Logger.getLogger(ConnectionFactory.class.getName());
    1 usage
    private static final String DRIVER = "com.mysql.cj.jdbc.Driver";
    1 usage
    private static final String DBURL = "jdbc:mysql://localhost:3306/ordersdb";
    1 usage
    private static final String USER = "root";
    1 usage
    private static final String PASS = "";
    1 usage
    private static ConnectionFactory singleInstance = new ConnectionFactory();
    /**
     * Private constructor to prevent instantiation from outside this class.
     * Loads the database driver class.
     */
    1 usage
    private ConnectionFactory() {...}
```

```java
/**
 * Creates a new database connection using the predefined credentials.
 *
 * @return A new connection object
 * @throws RuntimeException if a connection cannot be established
 */
1 usage
private Connection createConnection() {...}
/**
 * Provides a global point of access to the singleton instance of the {@code ConnectionFactory}.
 * Returns a new connection instance each time it's called.
 * @return a connection to the database
 */
10 usages
public static Connection getConnection() { return singleInstance.createConnection(); }
```

4.5. Dao

4.5.1. BillDao

Acces la date pentru facturi.

```java
import ...
/**
 * Provides the data access object (DAO) for handling all database operations
 * related to the {@code Bill} entity. This includes operations such as inserting new bills
 * and retrieving all bills from the database.
 */
3 usages
public class BillDAO {
    1 usage
    private static final String INSERT_BILL = "INSERT INTO bill (orderId, clientId, amount, dateTime) VALUES (?, ?, ?, ?)";
    1 usage
    private static final String SELECT_ALL_BILLS = "SELECT * FROM bill";
    /**
     * Inserts a new bill into the database. This method sets up a connection to the database,
     * prepares a statement for execution, and fills it with bill data.
     *
     * @param bill the {@code Bill} object containing the data to be inserted
     */
    1 usage
    public static void insert(Bill bill) {...}
    /**
     * Retrieves all bills from the database. This method sets up a connection, prepares a statement,
     * executes it and builds a list of {@code Bill} objects from the results.
     *
     * @return a list of {@code Bill} objects representing all bills stored in the database
     */
    1 usage
    public static List<Bill> findAll() {...}
}
```

4.5.2. *ClientDao*

Acces la date pentru facturi.

```java
import ...
/**
 * Data Access Object (DAO) for managing client data in the database. This class extends {@code GenericDAO} and
 * provides specific methods for client-related database operations. It handles the CRUD operations for the {@code Client}
 * entity by interacting with a SQL database, utilizing {@code ConnectionFactory} to establish database connections.
 */
3 usages
public class ClientDAO extends GenericDAO<Client> {

    /**
     * Constructs a new {@code ClientDAO} instance for managing {@code Client} entities, utilizing a generic DAO structure.
     */
    1 usage
    public ClientDAO() {
        super(Client.class);
    }
    /**
     * Retrieves a client by their ID from the database.
     * This method returns an {@code Optional} of {@code Client}, which will be empty if no client is found with the given ID.
     * @param clientId the unique identifier of the client to find
     * @return an {@code Optional} containing the found client or an empty {@code Optional} if no client is found
     */
    4 usages
    @Override
    public Optional<Client> findById(int clientId) {...}
}
```

4.5.3. *GenericDao*

O clasă generică pentru accesul la date, utilizabilă pentru mai multe modele.

```java
/**
 * Generic Data Access Object (DAO) class that provides common database operations for any specified type of entity.
 * This class is designed to be extended by other DAO classes that specify a particular entity type to manage.
 *
 * @param <T> the type of the entity that this DAO will manage
 */
4 usages   3 inheritors
public abstract class GenericDAO<T> {
    12 usages
    private Class<T> type;
    11 usages
    protected static final Logger LOGGER = Logger.getLogger(GenericDAO.class.getName());
    /**
     * Constructs a new GenericDAO for the specified type.
     * @param type the class of the entity type this DAO will manage
     */
    3 usages
    public GenericDAO(Class<T> type) { this.type = type; }
    /**
     * Retrieves the table name based on the entity type. This method may need to be adapted based on the database schema.
     * @return the table name for the current entity type
     */
```

```java
 * Finds an entity by its ID.
 * @param id the ID of the entity to find
 * @return an {@link Optional} containing the found entity or an empty {@link Optional} if no entity is found
 */
4 usages  3 overrides
public Optional<T> findById(int id) {...}
/**
 * Retrieves all entities of type T from the database.
 * @return a list of all entities found
 */
2 usages
public List<T> findAll() {...}
/**
 * Inserts a new entity into the database.
 * @param obj the entity to insert
 * @return the generated ID of the inserted entity, or -1 if an error occurred
 */
3 usages
public int insert(T obj) {...}
/**
 * Updates an existing entity in the database.
 * @param obj the entity to update
 */
3 usages
public void update(T obj) {...}
/**
 * Deletes an entity by its ID from the database.
 * @param id the ID of the entity to delete
 */
2 usages
public void delete(int id) {...}
```

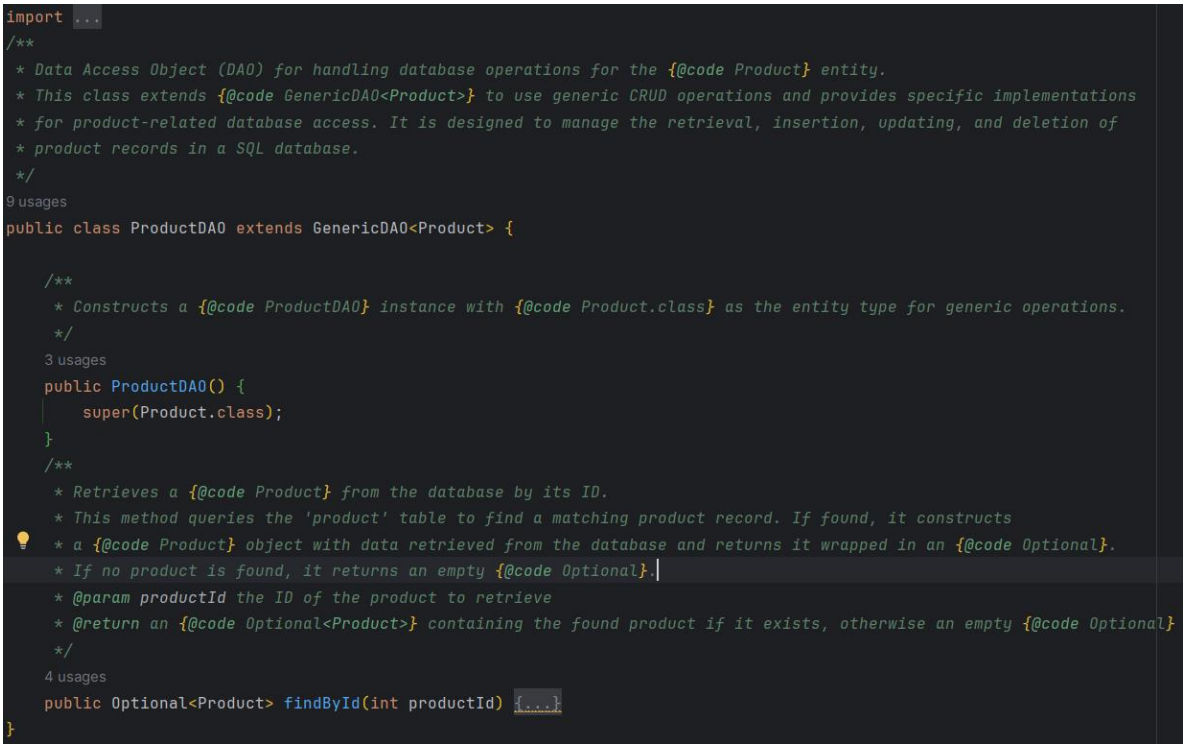
4.5.4. OrderDao

Acces la date pentru comenzile plasate.

```java
import ...
/**
 * Data Access Object (DAO) for handling database operations specific to {@code Order} entities.
 * This class provides functionalities for retrieving order details from the database. It extends
 * the {@code GenericDAO} class to utilize common CRUD operations while providing specifics for order-related queries.
 */
3 usages
public class OrderDAO extends GenericDAO<Order> {
    /**
     * Constructs an {@code OrderDAO} instance with {@code Order.class} as the entity type for generic operations.
     */
    1 usage
    public OrderDAO() { super(Order.class); }
    /**
     * Retrieves an {@code Order} from the database by its ID.
     * This method queries the 'order' table to find a matching order record. If found, it constructs
     * an {@code Order} object and returns it wrapped in an {@code Optional}. If no order is found, it returns
     * an empty {@code Optional}.
     *
     * @param orderId the ID of the order to retrieve
     * @return an {@code Optional<Order>} containing the found order if it exists, otherwise an empty {@code Optional}
     */
    4 usages
    @Override
    public Optional<Order> findById(int orderId) {...}
}
```

4.5.5. *ProductDao*

Acces la date pentru produse.



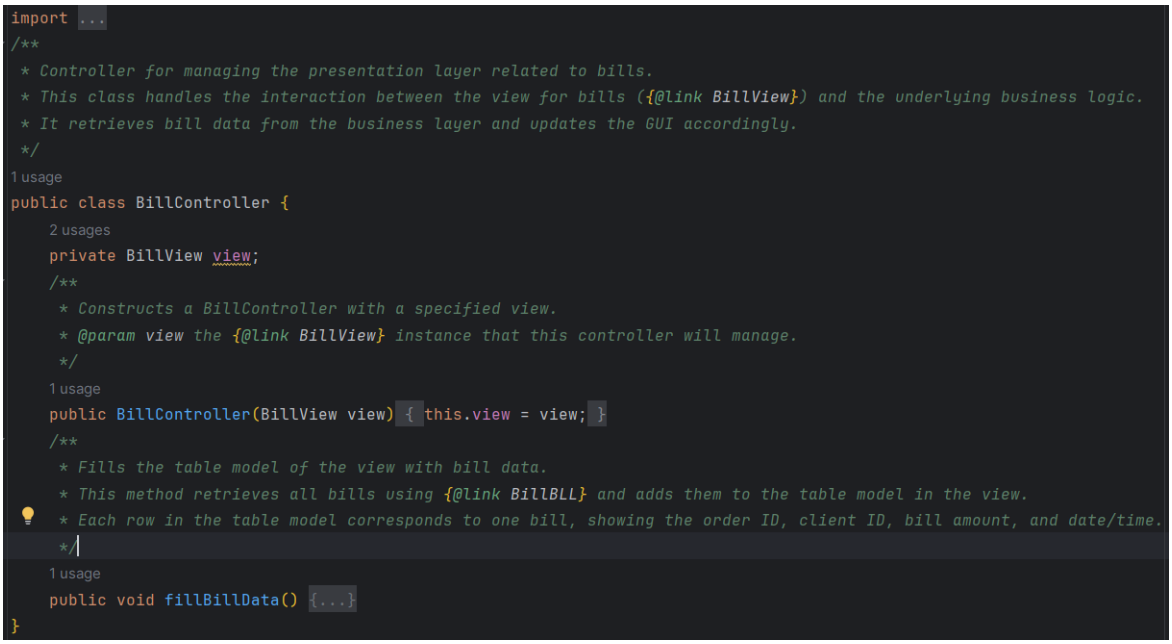
```java
import ...
/**
 * Data Access Object (DAO) for handling database operations for the {@code Product} entity.
 * This class extends {@code GenericDAO<Product>} to use generic CRUD operations and provides specific implementations
 * for product-related database access. It is designed to manage the retrieval, insertion, updating, and deletion of
 * product records in a SQL database.
 */
9 usages
public class ProductDAO extends GenericDAO<Product> {

    /**
     * Constructs a {@code ProductDAO} instance with {@code Product.class} as the entity type for generic operations.
     */
    3 usages
    public ProductDAO() {
        super(Product.class);
    }
    /**
     * Retrieves a {@code Product} from the database by its ID.
     * This method queries the 'product' table to find a matching product record. If found, it constructs
     * a {@code Product} object with data retrieved from the database and returns it wrapped in an {@code Optional}.
     * If no product is found, it returns an empty {@code Optional}.
     * @param productId the ID of the product to retrieve
     * @return an {@code Optional<Product>} containing the found product if it exists, otherwise an empty {@code Optional}
     */
    4 usages
    public Optional<Product> findById(int productId) {...}
}
```

4.6.Presentation

4.6.1. *BillController*

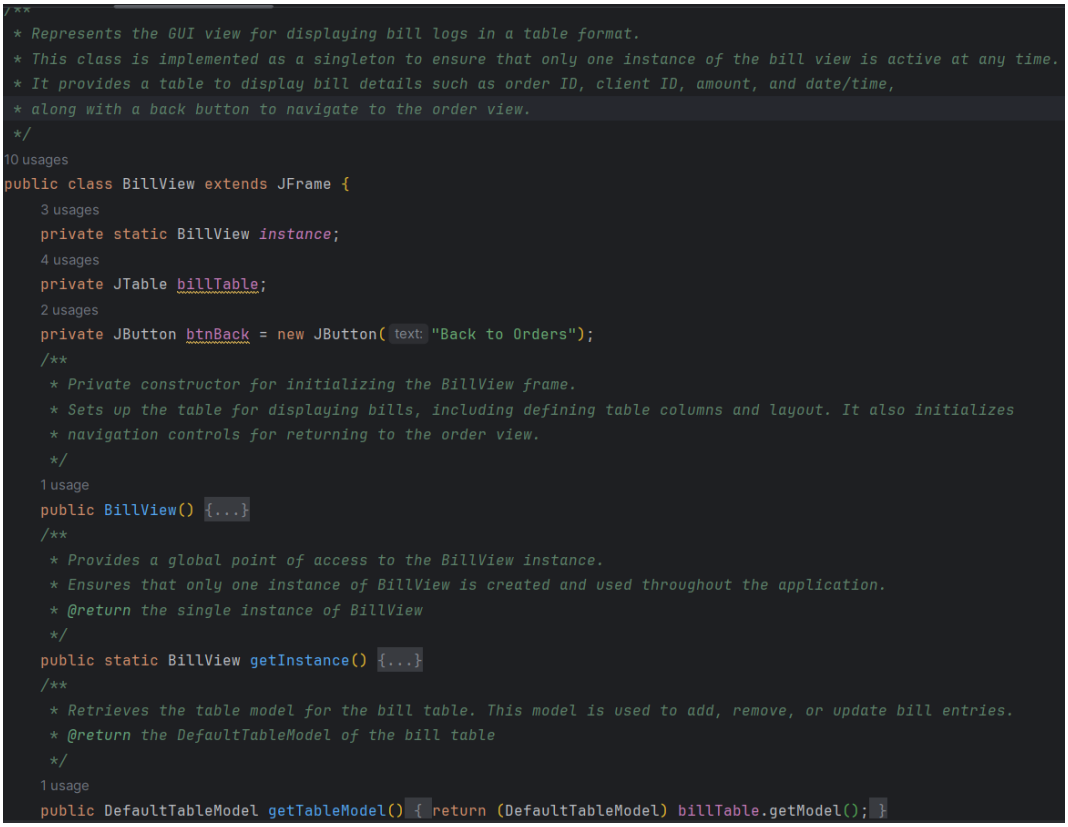
Controler pentru gestionarea UI pentru facturi.



```java
import ...
/**
 * Controller for managing the presentation layer related to bills.
 * This class handles the interaction between the view for bills ({@link BillView}) and the underlying business logic.
 * It retrieves bill data from the business layer and updates the GUI accordingly.
 */
1 usage
public class BillController {
    2 usages
    private BillView view;
    /**
     * Constructs a BillController with a specified view.
     * @param view the {@link BillView} instance that this controller will manage.
     */
    1 usage
    public BillController(BillView view) { this.view = view; }
    /**
     * Fills the table model of the view with bill data.
     * This method retrieves all bills using {@link BillBLL} and adds them to the table model in the view.
     * Each row in the table model corresponds to one bill, showing the order ID, client ID, bill amount, and date/time.
     */
    1 usage
    public void fillBillData() {...}
}
```

4.6.2. *BillView*

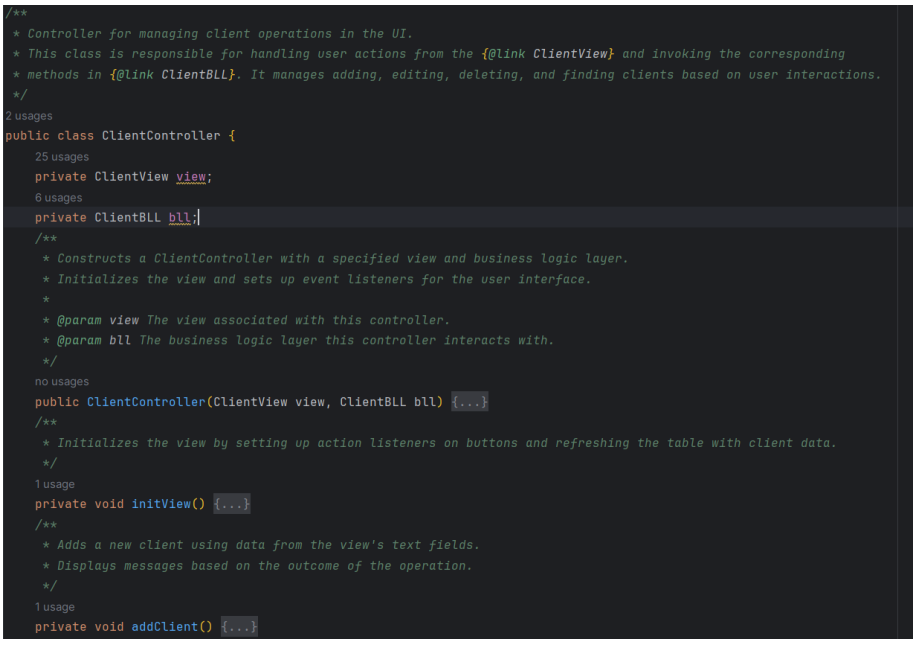
Interfață utilizator pentru facturi.



```java
/**
 * Represents the GUI view for displaying bill logs in a table format.
 * This class is implemented as a singleton to ensure that only one instance of the bill view is active at any time.
 * It provides a table to display bill details such as order ID, client ID, amount, and date/time,
 * along with a back button to navigate to the order view.
 */
10 usages
public class BillView extends JFrame {
    3 usages
    private static BillView instance;
    4 usages
    private JTable billTable;
    2 usages
    private JButton btnBack = new JButton( text: "Back to Orders");
    /**
     * Private constructor for initializing the BillView frame.
     * Sets up the table for displaying bills, including defining table columns and layout. It also initializes
     * navigation controls for returning to the order view.
     */
    1 usage
    public BillView() {...}
    /**
     * Provides a global point of access to the BillView instance.
     * Ensures that only one instance of BillView is created and used throughout the application.
     * @return the single instance of BillView
     */
    public static BillView getInstance() {...}
    /**
     * Retrieves the table model for the bill table. This model is used to add, remove, or update bill entries.
     * @return the DefaultTableModel of the bill table
     */
    1 usage
    public DefaultTableModel getTableModel() { return (DefaultTableModel) billTable.getModel(); }
```

4.6.3. *ClientController*

Controler pentru gestionarea UI pentru clienți.



```java
/**
 * Controller for managing client operations in the UI.
 * This class is responsible for handling user actions from the {@link ClientView} and invoking the corresponding
 * methods in {@link ClientBLL}. It manages adding, editing, deleting, and finding clients based on user interactions.
 */
2 usages
public class ClientController {
    25 usages
    private ClientView view;
    6 usages
    private ClientBLL bll;
    /**
     * Constructs a ClientController with a specified view and business logic layer.
     * Initializes the view and sets up event listeners for the user interface.
     *
     * @param view The view associated with this controller.
     * @param bll The business logic layer this controller interacts with.
     */
    no usages
    public ClientController(ClientView view, ClientBLL bll) {...}
    /**
     * Initializes the view by setting up action listeners on buttons and refreshing the table with client data.
     */
    1 usage
    private void initView() {...}
    /**
     * Adds a new client using data from the view's text fields.
     * Displays messages based on the outcome of the operation.
     */
    1 usage
    private void addClient() {...}
```

4.6.4. *ClientView*

Interfață utilizator pentru clienți.

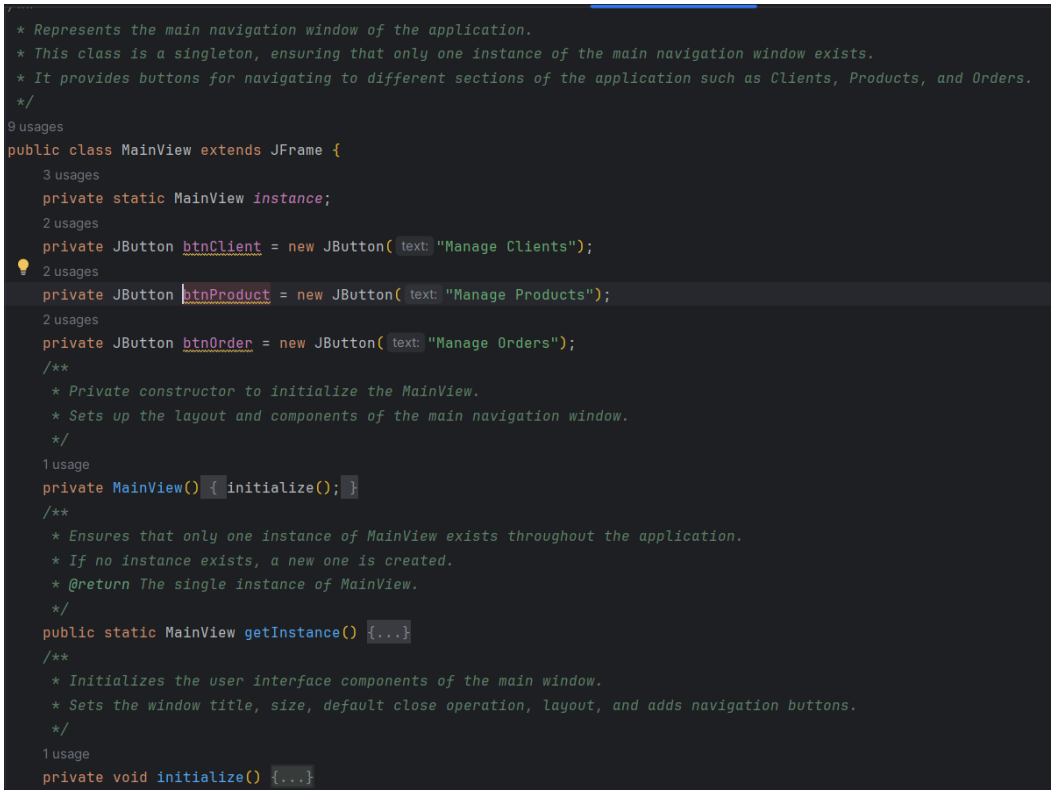
```
 * Provides a user interface for managing clients. This includes adding, editing, deleting, and searching for clie
 * It utilizes {@link ClientBLL} for business operations and updates the view accordingly.
 * This class is a singleton to ensure only one instance is used throughout the application.
 */
9 usages
public class ClientView extends JFrame {
    1 usage
    private ClientController controller;
    3 usages
    private static ClientView instance;

    3 usages
    private JButton btnAdd = new JButton( text: "Add Client");
    3 usages
    private JButton btnEdit = new JButton( text: "Edit Client");
    3 usages
    private JButton btnDelete = new JButton( text: "Delete Client");
    3 usages
    private JButton btnFindById = new JButton( text: "Find By ID");
    1 usage
    private JTextField txtName = new JTextField( columns: 20);
    1 usage
    private JTextField txtAddress = new JTextField( columns: 20);
    1 usage
    private JTextField txtEmail = new JTextField( columns: 20);
    5 usages
    private JTable tableClients = new JTable();
    3 usages
    private JButton btnBack = new JButton( text: "Back to Main");
    10 usages
    private DefaultTableModel tableModel;
```

```
/**
 * Private constructor for ClientView initializing the UI components and layout.
 * @param clientBLL The business logic layer object that provides client management functionality.
 */
1 usage
private ClientView(ClientBLL clientBLL) {
    this.clientBLL = clientBLL;
    initializeUI();
}
/**
 * Sets the controller for this view which manages user interactions.
 * @param controller The controller that interacts with this view.
 */
no usages
public void setController(ClientController controller) { this.controller = controller; }
/**
 * Initializes the user interface components and layout, including action listeners for interaction.
 */
1 usage
private void initializeUI() {...}
/**
 * Retrieves the singleton instance of ClientView with the provided business logic layer object.
 * If the instance does not exist, it creates a new one.
 * @param clientBLL The ClientBLL instance required for client operations.
 * @return The single instance of ClientView.
 */
1 usage
public static ClientView getInstance(ClientBLL clientBLL) {...}
```

4.6.5. *MainView*

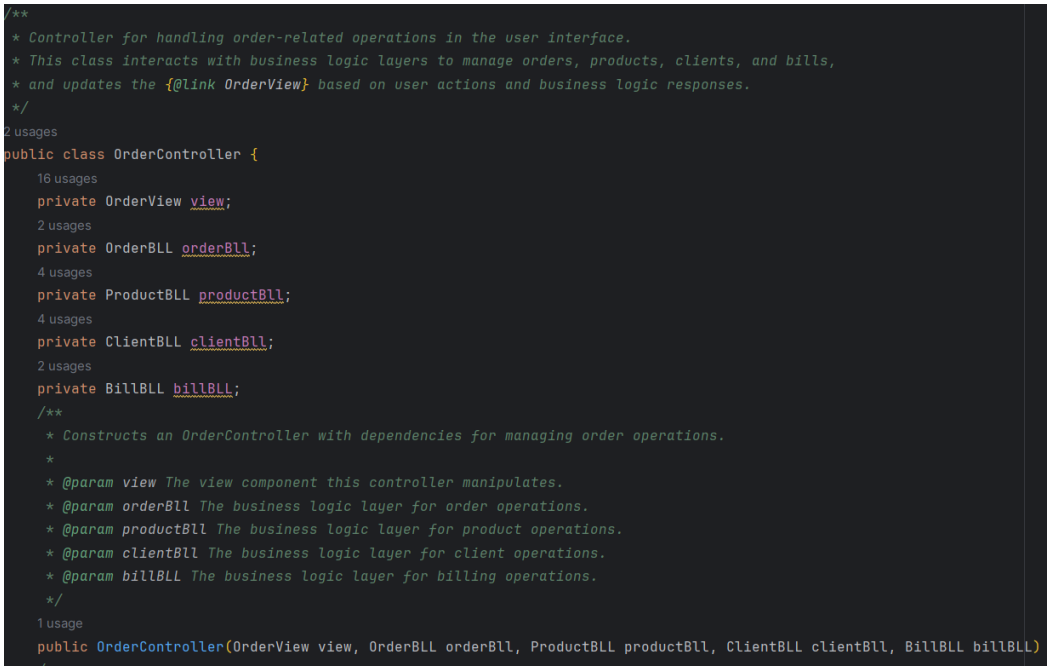
Interfața principală a aplicației.



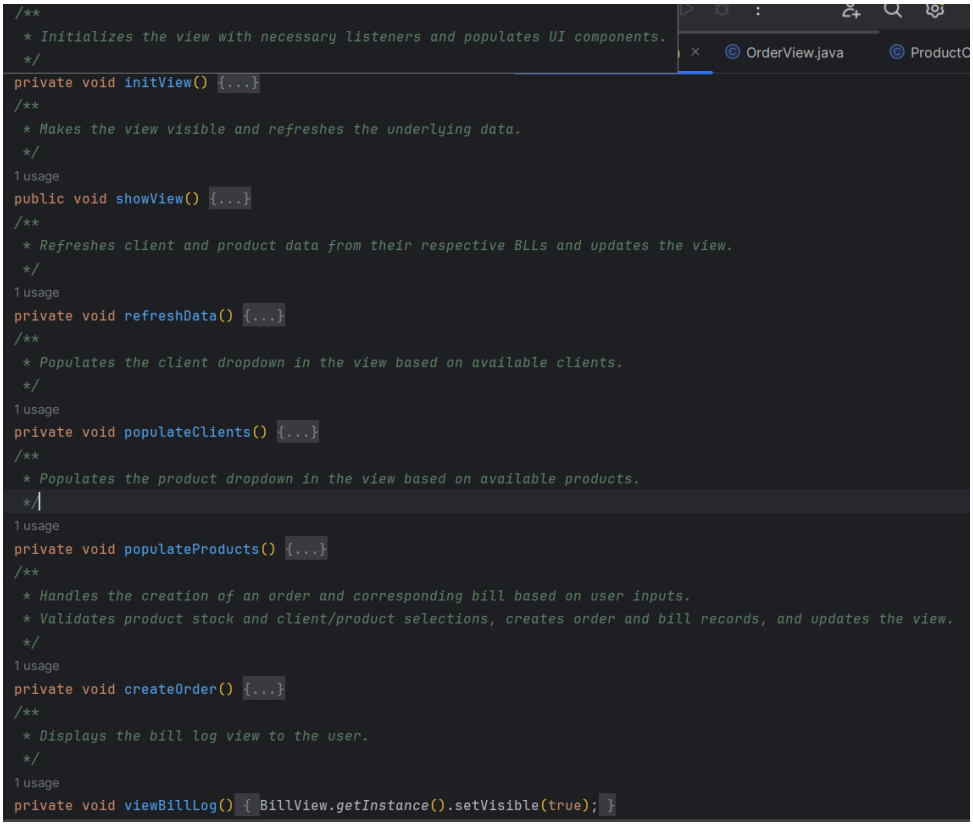
```java
/**
 * Represents the main navigation window of the application.
 * This class is a singleton, ensuring that only one instance of the main navigation window exists.
 * It provides buttons for navigating to different sections of the application such as Clients, Products, and Orders.
 */
9 usages
public class MainView extends JFrame {
    3 usages
    private static MainView instance;
    2 usages
    private JButton btnClient = new JButton( text: "Manage Clients");
    2 usages
    private JButton btnProduct = new JButton( text: "Manage Products");
    2 usages
    private JButton btnOrder = new JButton( text: "Manage Orders");
    /**
     * Private constructor to initialize the MainView.
     * Sets up the layout and components of the main navigation window.
     */
    1 usage
    private MainView() { initialize(); }
    /**
     * Ensures that only one instance of MainView exists throughout the application.
     * If no instance exists, a new one is created.
     * @return The single instance of MainView.
     */
    public static MainView getInstance() {...}
    /**
     * Initializes the user interface components of the main window.
     * Sets the window title, size, default close operation, layout, and adds navigation buttons.
     */
    1 usage
    private void initialize() {...}
```

4.6.6. *OrderController*

Controler pentru gestionarea UI pentru comenzi.

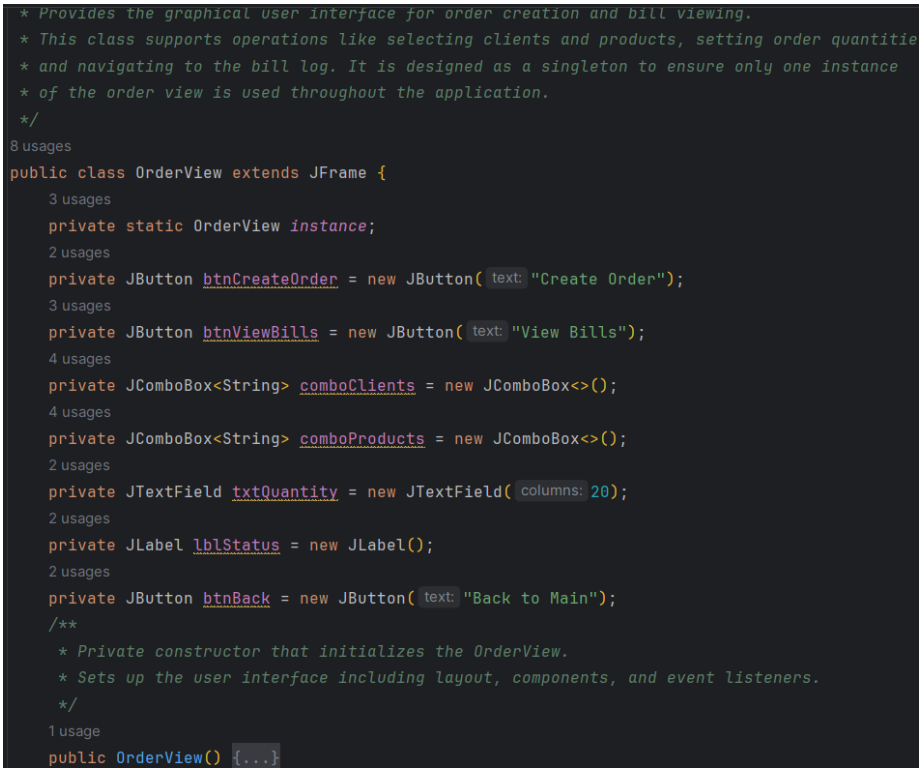


```java
/**
 * Controller for handling order-related operations in the user interface.
 * This class interacts with business logic layers to manage orders, products, clients, and bills,
 * and updates the {@link OrderView} based on user actions and business logic responses.
 */
2 usages
public class OrderController {
    16 usages
    private OrderView view;
    2 usages
    private OrderBLL orderBll;
    4 usages
    private ProductBLL productBll;
    4 usages
    private ClientBLL clientBll;
    2 usages
    private BillBLL billBLL;
    /**
     * Constructs an OrderController with dependencies for managing order operations.
     *
     * @param view The view component this controller manipulates.
     * @param orderBll The business logic layer for order operations.
     * @param productBll The business logic layer for product operations.
     * @param clientBll The business logic layer for client operations.
     * @param billBLL The business logic layer for billing operations.
     */
    1 usage
    public OrderController(OrderView view, OrderBLL orderBll, ProductBLL productBll, ClientBLL clientBll, BillBLL billBLL)
```



4.6.7. OrderView

Interfață utilizator pentru comenzi.



```java
/**
 * Provides global access to the singleton instance of OrderView.
 * If no instance exists, a new one is created.
 * @return The single instance of OrderView.
 */
public static OrderView getInstance() {...}
/**
 * Refreshes the client and product dropdown menus with the latest data.
 * @param clients List of clients to populate the client dropdown.
 * @param products List of products to populate the product dropdown.
 */
1 usage
public void refreshData(List<Client> clients, List<Product> products) {...}
// Getter methods for the UI components.
1 usage
public JButton getBtnCreateOrder() { return btnCreateOrder; }
2 usages
public JComboBox<String> getComboClients() { return comboClients; }
2 usages
public JComboBox<String> getComboProducts() { return comboProducts; }
1 usage
public JTextField getTxtQuantity() { return txtQuantity; }
1 usage
public JLabel getLblStatus() { return lblStatus; }
1 usage
public JButton getBtnViewBills() { return btnViewBills; }
```

4.6.8. ProductController

Controler pentru gestionarea UI pentru produse.

```java
/**
 * Controls the interactions between the product view and the business logic layer.
 * It manages the creation, modification, deletion, and querying of product data through the GUI.
 */
2 usages
public class ProductController {
    25 usages
    private ProductView view;
    6 usages
    private ProductBLL bll;
    /**
     * Constructs a ProductController with a given view and business logic layer handler.
     * @param view The GUI component this controller manipulates.
     * @param bll The business logic layer for handling product operations.
     */
    no usages
    public ProductController(ProductView view, ProductBLL bll) {
        this.view = view;
        this.bll = bll;
        initView();
    }
    /**
     * Initializes the view by setting up action listeners on buttons and refreshing the table.
     */
    1 usage
    private void initView() {...}
    /**
     * Adds a new product using the data entered in the view's text fields.
     */
    1 usage
    private void addProduct() {...}
```

```java
/**
 * Edits an existing product selected in the table using the updated data from the view's text fields.
 */
1 usage
private void editProduct() {...}
/**
 * Deletes an existing product selected in the table.
 */
1 usage
private void deleteProduct() {...}
/**
 * Finds a product by ID, displaying product details if found.
 */
1 usage
public void findProductById() {...}
/**
 * Refreshes the product table with current data from the business layer.
 */
4 usages
public void refreshTable() {...}
```

4.6.9. ProductView

Interfață utilizator pentru produse.

```java
* Provides a graphical user interface for managing products, including operations
* such as adding, editing, and deleting products, as well as querying for product details.
* This class is designed as a singleton to ensure a single instance manages the product views throughout the application.
*/
3 usages
public class ProductView extends JFrame {
    1 usage
    private ProductController controller;
    3 usages
    private static ProductView instance;
    3 usages
    private JButton btnAdd = new JButton( text: "Add Product");
    3 usages
    private JButton btnEdit = new JButton( text: "Edit Product");
    3 usages
    private JButton btnDelete = new JButton( text: "Delete Product");
    3 usages
    private JButton btnFindById = new JButton( text: "Find By ID");
    1 usage
    private JTextField txtName = new JTextField( columns: 20);
    1 usage
    private JTextField txtPrice = new JTextField( columns: 20);
    1 usage
    private JTextField txtStock = new JTextField( columns: 20);
    5 usages
    private JTable tableProducts = new JTable();
    3 usages
    private JButton btnBack = new JButton( text: "Back to Main");
    6 usages
    private ProductBLL productBLL;
    10 usages
```

```java
/**
 * Private constructor that initializes the ProductView, setting up UI components and event handlers.
 * @param productBLL The business logic layer that this view interacts with.
 */
1 usage
private ProductView(ProductBLL productBLL) {...}
/**
 * Sets the controller for this view which manages user interactions.
 * @param controller The controller that interacts with this view.
 */
no usages
public void setController(ProductController controller) { this.controller = controller; }
/**
 * Initializes the user interface including layout, components, and event listeners.
 */
1 usage
private void initializeUI() {...}
/**
 * Provides global access to the singleton instance of ProductView.
 * If no instance exists, a new one is created and configured.
 * @param productBLL The business logic layer required for product operations.
 * @return The single instance of ProductView.
 */
1 usage
public static ProductView getInstance(ProductBLL productBLL) {...}
/**
 * Refreshes the table displaying products using a list of products.
 * Clears the existing table content and repopulates it with updated data.
 * @param products The list of products to display in the table.
 */
```

4.6.10. TableModelUtils

Utilitare pentru modelele de tabel în interfața utilizator.

```java
/**
 * Utility class for populating a {@link JTable} using reflection to automatically
 * extract and display data from a list of objects.
 */
2 usages
public class TableModelUtils {

    /**
     * Populates a specified JTable with data extracted from a list of objects.
     * The fields of the objects in the list are used to generate the column names and
     * the values of these fields are used as the data for the table rows.
     *
     * @param table The {@link JTable} to be populated.
     * @param items The list of objects to extract data from. Each object in the list
     *              should have the same type, as the first object's fields are used
     *              to define the column names.
     */
    2 usages
    public static void populateTableWithReflection(JTable table, List<?> items) {
        Vector<String> columnNames = new Vector<>();
        Vector<Vector<Object>> data = new Vector<>();

        if (!items.isEmpty()) {
            Object firstItem = items.get(0);
            for (Field field : firstItem.getClass().getDeclaredFields()) {...}

            for (Object item : items) {...}
        }
        DefaultTableModel model = new DefaultTableModel(data, columnNames);
        table.setModel(model);
    }
```

5. Rezultate

Implementarea aplicației de gestionare a comenzilor pentru un depozit a dus la rezultate semnificative în termeni de funcționalitate și ușurință în utilizare. Aplicația oferă o interfață grafică intuitivă și eficientă, care permite utilizatorilor să gestioneze clienți, produse și comenzi într-un mod eficient. Utilizatorii pot adăuga, edita și șterge clienți cu ușurință, iar toate modificările sunt reflectate corect în baza de date. La fel, produsele pot fi adăugate, actualizate sau șterse, cu informații actualizate privind stocul, facilitând gestionarea inventarului. Procesarea comenzilor este simplă și directă, aplicația verificând automat disponibilitatea produselor și actualizând stocurile după fiecare comandă, cu mesaje de avertizare pentru stoc insuficient afișate când este necesar.

Interfața utilizatorului este clară și bine organizată, permițând o navigare ușoară între diferitele module ale aplicației, cum ar fi clienți, produse și comenzi. Documentația este completă și oferă instrucțiuni detaliate despre utilizarea aplicației, inclusiv scenarii comune și soluționarea problemelor, ceea ce îmbunătățește experiența utilizatorilor. În plus, documentația Javadoc este generată corect, oferind descrieri utile pentru clase și metode, ceea ce facilitează înțelegerea și întreținerea codului.

Implementarea tehnicilor de reflecție pentru dinamizarea creării interfețelor și a interogărilor în baza de date a redus cantitatea de cod necesară și a minimizat erorile asociate cu duplicarea codului. Prin această implementare, aplicația oferă o soluție robustă pentru gestionarea operațiunilor unui depozit, îmbunătățind eficiența proceselor de afaceri și contribuind direct la creșterea productivității și la reducerea erorilor umane sau întârzierilor în procesarea comenzilor.

6. Concluzii și dezvoltări ulterioare

Implementarea aplicației de gestionare a comenzilor pentru un depozit a validat eficacitatea utilizării unei arhitecturi pe straturi și a bazelor de date relaționale pentru o gestionare organizată și eficientă a datelor. Interfața utilizatorului, care este optimizată și intuitivă, a îmbunătățit semnificativ eficiența în operarea zilnică, crescând astfel productivitatea și minimizând posibilitățile de erori umane. Utilizarea tehnicilor de reflecție pentru generarea dinamică a interfețelor și interogărilor de bază de date a contribuit la eficientizarea dezvoltării și a redus riscul de erori asociate cu duplicarea codului.

Privind spre viitor, dezvoltarea aplicației ar putea beneficia de mai multe îmbunătățiri pentru a spori utilitatea și scalabilitatea acesteia. Integrarea cu alte sisteme de management al depozitelor ar putea oferi o gestionare mai fluidă a stocurilor și comenzi provenite din diverse canale. Un modul avansat de raportare ar permite utilizatorilor să genereze rapoarte personalizate, oferind astfel date valoroase pentru luarea deciziilor strategice.

Dezvoltarea unei aplicații mobile sau a unei interfețe web responsive ar îmbunătăți accesibilitatea sistemului, permitând gestionarea afacerii de oriunde și oricând. Implementarea

tehnologiilor de inteligență artificială pentru anticiparea necesarului de stocuri ar putea optimiza gestionarea inventarului, prevenind problemele de sub-stocare sau supra-stocare.

De asemenea, îmbunătățirea măsurilor de securitate, inclusiv autentificarea multifactor și criptarea datelor, ar proteja informațiile sensibile ale companiei și ale clienților. În cele din urmă, asigurarea conformității cu normele de sustenabilitate și reglementările actuale ar putea ajuta compania să își îmbunătățească practicile de afaceri și să rămână competitivă pe piață.

Aceste dezvoltări ar extinde funcționalitățile aplicației, consolidându-i rolul ca instrument esențial în managementul operațional al unui depozit și sprijinind adaptabilitatea și creșterea continuă a afacerii într-un mediu economic în schimbare.

7. Bibliografie

- Connect to MySql from a Java application
  - https://www.baeldung.com/java-jdbc
  - http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/
- Layered architectures
  - https://dzone.com/articles/layers-standard-enterprise
- Reflection in Java
  - http://tutorials.jenkov.com/java-reflection/index.html
- Creating PDF files in Java
  - https://www.baeldung.com/java-pdf-creation
- JAVADOC
  - https://www.baeldung.com/javadoc
- SQL dump file generation
  - https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html