

DOCUMENTATIE

TEMA 3

Nume student: Lucăcel Mălina

Grupa: 30221

CUPRINS

1. Obiectivul temei	3
2. Analiza problemei, modelare, scenarii, cazuri de utilizare.....	3
3. Proiectare	5
4. Implementare	6
5. Rezultate	17
6. Concluzii.....	19
7. Bibliografie.....	19

1. Obiectivul temei

Obiectivul proiectului constă în dezvoltarea unei aplicații de Management al Comenzilor pentru procesarea comenzilor clienților pentru un depozit. Aplicația va utiliza baze de date relaționale pentru stocarea produselor, clienților și comenzilor. Proiectul va fi proiectat conform modelului de arhitectură stratificată și va utiliza (cel puțin) următoarele clase:

- o Clasele de model - vor reprezenta modelele de date ale aplicației. Aceste clase vor defini structura și comportamentul produselor, clienților și comenzilor.
- o Clasele de logica de business - vor conține logica aplicației, inclusiv operațiile de procesare a comenzilor și interacțiunea cu baza de date. Aceste clase vor manipula obiectele de model și vor implementa regulile de afaceri specifice.
- o Clasele de prezentare - vor fi responsabile de interfața grafică a aplicației. Aceste clase vor permite utilizatorilor să vizualizeze și să interacționeze cu comenzile și să introducă date noi.
- o Clasele de acces la date - vor conține funcționalitățile necesare pentru accesul la baza de date relațională. Aceste clase vor efectua operațiile de citire și scriere în baza de date, utilizând interogări SQL sau ORM (Object-Relational Mapping).

Obiectivul proiectului este să se creeze o aplicație funcțională care să permită utilizatorilor să gestioneze comenzile clienților pentru un depozit. Aplicația va respecta principiile arhitecturii stratificate, va utiliza baze de date relaționale pentru stocarea datelor și va avea funcționalități de modelare, logica de business, prezentare și acces la date.

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

Pentru a dezvolta aplicația de Management al Comenzilor pentru un depozit, trebuie să înțelegem mai întâi problema pe care o rezolvăm și cerințele implicate. Scopul aplicației este de a facilita procesul de prelucrare a comenzilor clienților și de a asigura o gestionare eficientă a produselor și clienților într-un depozit. Principalele cerințe ale aplicației includ:

1. Înregistrarea clienților noi și stocarea datelor lor relevante, cum ar fi numele, adresa de email și vârsta.
2. Gestionarea produselor disponibile în depozit, inclusiv denumirea și cantitatea disponibilă.
3. Procesarea comenzilor clienților, înregistrarea detaliilor comenzii și actualizarea stocului de produse.
4. Generarea rapoartelor și statisticilor referitoare la stocul de produse, comenzile procesate și alte informații relevante.

Pentru a modela aplicația, vom utiliza abordarea arhitecturii stratificate, care ne permite să separăm diferitele componente și să le gestionăm în mod independent. Vom avea următoarele componente principale:

1. Modelul de date: Vom utiliza o bază de date relațională pentru a stoca informațiile despre produse, clienți și comenzile acestora. Vom defini tabelele corespunzătoare și relațiile între ele pentru a asigura o gestionare coerentă a datelor.
2. Clasele de model: Acestea vor reprezenta structura și comportamentul entităților noastre principale, cum ar fi clasa "Produs", "Client" și "Comandă". Aceste clase vor conține atributele relevante și metodele necesare pentru a manipula și accesa datele.
3. Clasele de logică de business: Acestea vor conține logica aplicației, inclusiv operațiile de procesare a comenzilor, gestionarea stocului de produse și generarea de rapoarte. Aceste clase vor interacționa cu clasele de model și cu clasele de acces la date pentru a executa operațiunile necesare.
4. Clasele de prezentare: Acestea vor gestiona interfața grafică a aplicației și interacțiunea cu utilizatorii. Vor permite utilizatorilor să vizualizeze produsele disponibile, să plaseze comenzi și să acceseze rapoartele generate.
5. Clasele de acces la date: Acestea vor furniza funcționalitățile necesare pentru a accesa și manipula datele din baza de date. Vor realiza operațiile de citire și scriere utilizând interogări SQL sau tehnici ORM (Object-Relational Mapping).

Pentru a înțelege mai bine cum va funcționa aplicația, să examinăm câteva scenarii cheie:

1. Plasarea unei comenzi:
 - Utilizatorul vizualizează lista de produse disponibile.
 - Utilizatorul selectează produsele dorite și le adaugă în "coșul de cumpărături"
 - Utilizatorul finalizează comanda (apasa add)
 - Aplicația înregistrează comanda și actualizează stocul de produse disponibile.
2. Generarea unui raport de stoc:
 - Utilizatorul selectează opțiunea de generare a rapoartelor(view)
 - Aplicația accesează baza de date și extrage informațiile despre stocul de produse disponibile.
 - Aplicația generează un raport care include denumirea produselor și cantitatea disponibilă pentru fiecare produs.

Vom identifica câteva cazuri de utilizare principale pentru aplicația noastră:

1. Înregistrarea utilizatorului:

- Aplicația validează datele introduse și le stochează în baza de date.
- Utilizatorul primește un mesaj de confirmare că înregistrarea a fost finalizată cu succes.

2. Vizualizarea produselor disponibile:

- Utilizatorul accesează secțiunea de produse din aplicație.
- Aplicația accesează baza de date și afișează lista de produse disponibile împreună cu detaliile relevante (denumire, cantitate, etc.).

3. Plasarea unei comenzi

- Utilizatorul plasează o comandă. Aceasta va fi plasată sau se va afișa un mesaj de eroare

3. Proiectare

Diagrama UML :

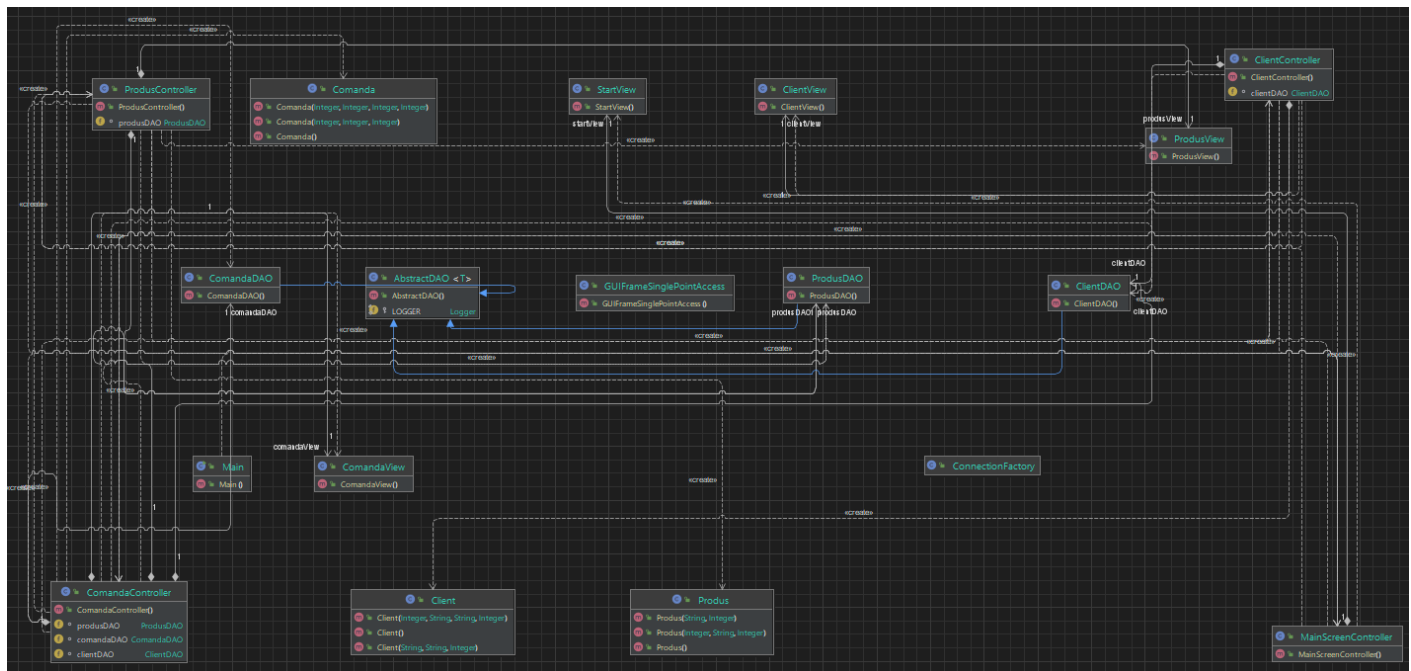
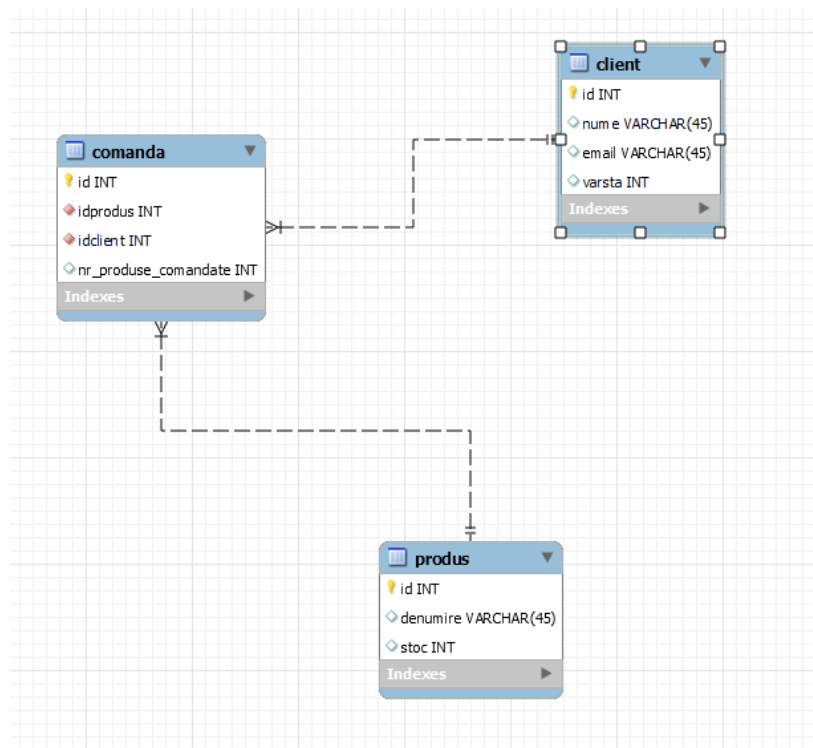


Diagrama EER :



4. Implementare

Clasa **Client** reprezintă o entitate care descrie un client în cadrul aplicației. Are următoarele atribute:

- **id** - reprezintă identificatorul unic al clientului și este de tip **Integer**.
- **nume** - reprezintă numele clientului și este de tip **String**.
- **email** - reprezintă adresa de email a clientului și este de tip **String**.
- **varsta** - reprezintă vârsta clientului și este de tip **Integer**.

Clasa **Client** oferă mai multe metode pentru a gestiona și accesa datele unui client:

- Constructorul **Client(Integer id, String nume, String email, Integer varsta)** initializează un obiect **Client** cu identificatorul, numele, adresa de email și vârsta specificate.

- Constructorul **Client(String nume, String email, Integer varsta)** initializează un obiect **Client** cu numele, adresa de email și vârsta specificate.
- Constructorul **Client()** creează un obiect **Client** gol, fără atribute inițializate.

Metodele de acces și modificare ale atributelor sunt definite prin metodele getter și setter corespunzătoare:

- **getId()** returnează identificatorul clientului.
- **setId(Integer id)** setează identificatorul clientului.
- **getNum()** returnează numele clientului.
- **setNume(String nume)** setează numele clientului.
- **getEmail()** returnează adresa de email a clientului.
- **setEmail(String email)** setează adresa de email a clientului.
- **getVarsta()** returnează vârsta clientului.
- **setVarsta(Integer varsta)** setează vârsta clientului.

Această clasă este utilizată pentru a crea și manipula obiecte **Client** în cadrul aplicației de Management al Comenzilor. Prin intermediul acestei clase, putem gestiona informațiile relevante despre clienți, cum ar fi numele, adresa de email și vârsta lor.

Clasa **Comanda** reprezintă o entitate care descrie o comandă plasată de un client pentru un anumit produs în cadrul aplicației. Are următoarele atribute:

- **id** - reprezintă identificatorul unic al comenzii și este de tip **Integer**.
- **idprodus** - reprezintă identificatorul produsului comandat și este de tip **Integer**.
- **idclient** - reprezintă identificatorul clientului care a plasat comanda și este de tip **Integer**.
- **nr_produce_comandate** - reprezintă numărul de produse comandate și este de tip **Integer**.

Clasa **Comanda** oferă mai multe metode pentru a gestiona și accesa datele unei comenzi:

- Constructorul **Comanda(Integer id, Integer idprodus, Integer idclient, Integer nr_produce_comandate)** initializează un obiect **Comanda** cu identificatorul comenzii, identificatorul produsului comandat, identificatorul clientului și numărul de produse comandate specificate.
- Constructorul **Comanda(Integer idprodus, Integer idclient, Integer nr_produce_comandate)** initializează un obiect **Comanda** cu identificatorul produsului comandat, identificatorul clientului și numărul de produse comandate specificate.

- Constructorul **Comanda()** creează un obiect **Comanda** gol, fără atribute inițializate.

Metodele de acces și modificare ale atributelor sunt definite prin metodele getter și setter corespunzătoare:

- **getNr_produce_comandate()** returnează numărul de produse comandate.
- **setNr_produce_comandate(Integer nr_produce_stoc)** setează numărul de produse comandate.
- **getId()** returnează identificatorul comenzii.
- **setId(Integer id)** setează identificatorul comenzii.
- **getIdprodus()** returnează identificatorul produsului comandat.
- **setIdprodus(Integer idprodus)** setează identificatorul produsului comandat.
- **getIdclient()** returnează identificatorul clientului care a plasat comanda.
- **setIdclient(Integer idclient)** setează identificatorul clientului care a plasat comanda.

Această clasă este utilizată pentru a crea și manipula obiecte **Comanda** în cadrul aplicației de Management al Comenzilor. Prin intermediul acestei clase, putem gestiona informațiile relevante despre comenzile plasate de clienți, inclusiv identificatorul comenzii, produsul comandat, clientul care a plasat comanda și numărul de produse comandate.

Clasa **Produs** reprezintă o entitate care descrie un produs în cadrul aplicației. Are următoarele atribute:

- **id** - reprezintă identificatorul unic al produsului și este de tip **Integer**.
- **denumire** - reprezintă denumirea produsului și este de tip **String**.
- **stoc** - reprezintă stocul disponibil pentru produs și este de tip **Integer**.

Clasa **Produs** oferă mai multe metode pentru a gestiona și accesa datele unui produs:

- Constructorul **Produs(Integer id, String denumire, Integer stoc)** initializează un obiect **Produs** cu identificatorul produsului, denumirea și stocul specificate.
- Constructorul **Produs(String denumire, Integer stoc)** initializează un obiect **Produs** cu denumirea și stocul specificate.
- Constructorul **Produs()** creează un obiect **Produs** gol, fără atribute inițializate.

Metodele de acces și modificare ale atributelor sunt definite prin metodele getter și setter corespunzătoare:

- **getId()** returnează identificatorul produsului.

- **setId(Integer id)** setează identificatorul produsului.
- **getDenumire()** returnează denumirea produsului.
- **setDenumire(String denumire)** setează denumirea produsului.
- **getStoc()** returnează stocul disponibil pentru produs.
- **setStoc(Integer stoc)** setează stocul disponibil pentru produs.

Această clasă este utilizată pentru a crea și manipula obiecte **Produs** în cadrul aplicației de Management al Comenzilor. Prin intermediul acestei clase, putem gestiona informațiile relevante despre produsele disponibile, inclusiv identificatorul produsului, denumirea și stocul disponibil.

Clasa **AbstractDAO** este o clasă generică care oferă funcționalități generale pentru interacțiunea cu baza de date în cadrul unui proiect Java. Aceasta servește drept clasă de bază pentru alte clase DAO (Data Access Object) care se ocupă de interogarea și manipularea specifică a entităților din baza de date.

Iată o explicație detaliată a fiecărei părți a clasei **AbstractDAO**:

1. Declararea pachetului și importarea claselor necesare:
 - Clasa se află în pachetul **org.example.reflection**.
 - Sunt importate clasele necesare pentru gestionarea reflecției, interacțiunea cu baza de date și alte dependențe.
2. Declararea clasei și a variabilelor membru:
 - Clasa **AbstractDAO** este declarată ca o clasă generică, cu parametrul generic **T** care reprezintă tipul entității cu care clasa interacționează.
 - Este declarată o constantă statică **LOGGER** pentru înregistrarea evenimentelor înregistrate în clasa **AbstractDAO**.
 - Este declarată o variabilă **type** de tip **Class<T>** pentru a reține tipul entității curente.
3. Constructorul clasei:
 - Constructorul clasei **AbstractDAO** este suprascriindu-se și este utilizat pentru a obține tipul entității generice **T** în timpul rulării, folosind reflecția.
 - Se utilizează **getClass().getGenericSuperclass()** pentru a obține tipul clasei generice de la care se extinde clasa **AbstractDAO**.
 - Se utilizează **getActualTypeArguments()[0]** pentru a obține primul argument de tip din lista de tipuri generice asociate superclasei.

4. Metodele private pentru generarea de interogări:

- **createSelectQuery(String field):** Metoda generează un șir de caractere pentru o interogare SELECT care selectează toate câmpurile din tabela corespunzătoare tipului entității, utilizând un anumit câmp pentru clauza WHERE.
- **createSelectSTOCQuery(String field):** Metoda generează un șir de caractere pentru o interogare SELECT care selectează doar câmpul "stoc" din tabela corespunzătoare tipului entității, utilizând un anumit câmp pentru clauza WHERE. Această metodă este specifică obiectului "Produs".
- **createDeleteQuery(String field):** Metoda generează un șir de caractere pentru o interogare DELETE care șterge înregistrările din tabela corespunzătoare tipului entității, utilizând un anumit câmp pentru clauza WHERE.
- **createInsertQuery(T t):** Metoda generează un șir de caractere pentru o interogare INSERT care inserează un obiect de tipul **T** în tabela corespunzătoare. Utilizează reflecția pentru a obține câmpurile și valorile obiectului.
- **createUpdateQuery(T t):** Metoda generează un șir de caractere pentru o interogare UPDATE care actualizează un obiect de tipul **T** în tabela corespunzătoare. Utilizează reflecția pentru a obține câmpurile și valorile obiectului.

5. Metodele publice pentru interacțiunea cu baza de date:

- **findById(int id):** Metoda returnează un obiect de tipul **T** găsit în baza de date pe baza unui ID dat.
- **findAll():** Metoda returnează o listă de obiecte de tipul **T** care reprezintă toate înregistrările din tabela corespunzătoare.
- **insert(T t):** Metoda inserează un obiect de tipul **T** în baza de date.
- **update(T t):** Metoda actualizează un obiect de tipul **T** în baza de date.
- **delete(int id):** Metoda șterge un obiect de tipul **T** din baza de date pe baza unui ID dat.

Clasa **AbstractDAO** oferă astfel metode generice care pot fi utilizate de clasele DAO concrete pentru a interacționa cu baza de date într-un mod consistent și flexibil, evitând duplicarea codului și oferind un nivel ridicat de abstractizare.

Clasele ClientDAO, ProdusDAO si ComandaDAO sunt doar o extindere a clasei AbstractDAO.

Clasa **ClientController** este responsabilă pentru gestionarea interacțiunii între interfața utilizator și codul aplicației în ceea ce privește entitatea "Client". Aceasta tratează cazurile de apăsare a fiecărui buton și implementează logica corespunzătoare.

Iată o explicație detaliată a clasei **ClientController**:

1. Importarea claselor necesare:
 - Se importă clasele necesare pentru gestionarea interacțiunii cu interfața utilizator și dependențele asociate.
2. Declararea clasei și a variabilelor membru:
 - Clasa **ClientController** este declarată în pachetul **org.example.controller**.
 - Se declară variabilele membru **clientView** de tip **ClientView** și **clientDAO** de tip **ClientDAO**.
3. Metoda **startLogic()**:
 - Metoda inițializează obiectul **clientView** cu o nouă instanță a clasei **ClientView**.
 - Se utilizează **GUIFrameSinglePointAccess.changePanel()** pentru a schimba panel-ul curent în cadrul interfeței utilizator.
 - Se definesc acțiunile care trebuie executate la apăsarea fiecărui buton din interfață utilizând **addActionListener()**.
4. Metodele private pentru tratarea acțiunilor butoanelor:
 - **getComandaButton()**: Metoda tratează cazul de apăsare al butonului "Comanda" și inițiază o nouă instanță a clasei **ComandaController** și apelează metoda **startLogic()** a acesteia.
 - **getProdusButton()**: Metoda tratează cazul de apăsare al butonului "Produs" și inițiază o nouă instanță a clasei **ProdusController** și apelează metoda **startLogic()** a acesteia.
 - **getStartButton()**: Metoda tratează cazul de apăsare al butonului "Start" și inițiază o nouă instanță a clasei **MainScreenController** și apelează metoda **startLogic()** a acesteia.
 - **getDeleteButton()**: Metoda tratează cazul de apăsare al butonului "Delete" și utilizează obiectul **clientDAO** pentru a șterge clientul cu ID-ul introdus în câmpul de text corespunzător. Apoi, se apelează metoda **getTable()** pentru actualizarea tabelului.
 - **getAddButton()**: Metoda tratează cazul de apăsare al butonului "Add" și utilizează obiectul **clientDAO** pentru a adăuga un client cu datele introduse în

câmpurile de text corespunzătoare. Apoi, se apelează metoda **getTable()** pentru actualizarea tabelului.

- **getEditButton()**: Metoda tratează cazul de apăsare al butonului "Edit" și utilizează obiectul **clientDAO** pentru a edita clientul cu datele introduse în câmpurile de text corespunzătoare. Dacă clientul nu există, se afișează un mesaj de eroare utilizând **GUIFrameSinglePointAccess.showMessageDialog()**. Apoi, se apelează metoda **getTable()** pentru actualizarea tabelului.
- **getViewButton()**: Metoda tratează cazul de apăsare al butonului "View" și apelează metoda **getTable()** pentru afișarea tabelului clientului.

5. Metoda privată **getTable()**:

- Metoda este responsabilă pentru generarea tabelului de clienti în cadrul interfeței utilizator.
- Se inițializează modelul tabelului, eliminând coloanele și rândurile existente.
- Se utilizează obiectul **clientDAO** pentru a obține antetul tabelului.
- Se adaugă coloanele în modelul tabelului.
- Se obține lista de clienți utilizând metoda **findAll()** din obiectul **clientDAO**.
- Dacă lista de clienți nu este nulă, se parcurge lista și se adaugă rândurile corespunzătoare în modelul tabelului.

Astfel, clasa **ClientController** gestionează interacțiunea utilizatorului cu interfața și comunicarea cu obiectele din nivelul de acces la date (**ClientDAO**).

Clasa **ComandaController** este responsabilă de gestionarea interacțiunii între interfața utilizator și codul aplicației în ceea ce privește entitatea "Comanda". Aceasta implementează logica corespunzătoare fiecărei acțiuni și gestionează evenimentele generate de apăsarea butoanelor din interfața utilizator.

Iată o explicație detaliată a clasei **ComandaController**:

1. Importarea claselor necesare:

- Se importă clasele necesare pentru gestionarea interacțiunii cu interfața utilizator și dependențele asociate.

2. Declararea clasei și a variabilelor membru:

- Clasa **ComandaController** este declarată în pachetul **org.example.controller**.

- Se declară variabilele membru **comandaView** de tip **ComandaView**, **comandaDAO** de tip **ComandaDAO**, **produsDAO** de tip **ProdusDAO** și **clientDAO** de tip **ClientDAO**.

3. Metoda **startLogic()**:

- Metoda inițializează obiectul **comandaView** cu o nouă instanță a clasei **ComandaView**.
- Se utilizează **GUIFrameSinglePointAccess.changePanel()** pentru a schimba panel-ul curent în cadrul interfeței utilizator.
- Se dezactivează butoanele "Delete" și "Edit" inițial.
- Se definesc acțiunile care trebuie executate la apăsarea fiecărui buton din interfață utilizând **addActionListener()**.

4. Metodele private pentru tratarea acțiunilor butoanelor:

- **getProdusButton()**: Metoda tratează cazul de apăsare al butonului "Produs" și inițiază o nouă instanță a clasei **ProdusController** și apelează metoda **startLogic()** a acesteia.
- **getStartButton()**: Metoda tratează cazul de apăsare al butonului "Start" și inițiază o nouă instanță a clasei **MainScreenController** și apelează metoda **startLogic()** a acesteia.
- **getClientButton()**: Metoda tratează cazul de apăsare al butonului "Client" și inițiază o nouă instanță a clasei **ClientController** și apelează metoda **startLogic()** a acesteia.
- **getViewButton()**: Metoda tratează cazul de apăsare al butonului "View" și apelează metoda **getTable()** pentru afișarea tabelului comenzilor.
- **getAddButton()**: Metoda tratează cazul de apăsare al butonului "Add" și implementează logica pentru adăugarea unei comenzi.
 - Se obțin datele necesare pentru crearea comenzii din câmpurile de text ale interfeței utilizator.
 - Se verifică dacă există suficiente produse în stoc și dacă clientul introdus există în baza de date utilizând obiectele **produsDAO** și **clientDAO**.
 - Dacă condițiile sunt îndeplinite, se creează o instanță a clasei **Comanda** cu datele introduse și se inserează în baza de date utilizând obiectul **comandaDAO**.
 - De asemenea, se actualizează stocul produsului folosind obiectul **produsDAO**.

5. Metoda privată **getTable()**:

- Metoda este responsabilă pentru generarea tabelului de comenzi în cadrul interfeței utilizator.
- Se inițializează modelul tabelului, eliminând coloanele și rândurile existente.
- Se utilizează obiectul **comandaDAO** pentru a obține antetul tabelului.
- Se adaugă coloanele în modelul tabelului.
- Se obține lista de comenzi utilizând metoda **findAll()** din obiectul **comandaDAO**.
- Dacă lista de comenzi nu este nulă, se parcurge lista și se adaugă rândurile corespunzătoare în modelul tabelului.

Astfel, clasa **ComandaController** gestionează interacțiunea utilizatorului cu interfața și comunicarea cu obiectele din nivelul de acces la date (**ComandaDAO**, **ProdusDAO**, **ClientDAO**).

Clasa **ProdusController** este responsabilă de gestionarea interacțiunii între interfața utilizator și codul aplicației în ceea ce privește entitatea "Produs". Aceasta implementează logica corespunzătoare fiecărei acțiuni și gestionează evenimentele generate de apăsarea butoanelor din interfața utilizator.

Iată o explicație detaliată a clasei **ProdusController**:

1. Importarea claselor necesare:

- Se importă clasele necesare pentru gestionarea interacțiunii cu interfața utilizator și dependențele asociate.

2. Declararea clasei și a variabilelor membru:

- Clasa **ProdusController** este declarată în pachetul **org.example.controller**.
- Se declară variabilele membru **produsView** de tip **ProdusView** și **produsDAO** de tip **ProdusDAO**.

3. Metoda **startLogic()**:

- Metoda inițializează obiectul **produsView** cu o nouă instanță a clasei **ProdusView**.
- Se utilizează **GUIFrameSinglePointAccess.changePanel()** pentru a schimba panel-ul curent în cadrul interfeței utilizator.
- Se definesc acțiunile care trebuie executate la apăsarea fiecărui buton din interfață utilizând **addActionListener()**.

4. Metodele private pentru tratarea acțiunilor butoanelor:

- **getTable():** Metoda este responsabilă pentru generarea tabelului de produse în cadrul interfeței utilizator.
 - Se inițializează modelul tabelului, eliminând coloanele și rândurile existente.
 - Se utilizează obiectul **produsDAO** pentru a obține antetul tabelului.
 - Se adaugă coloanele în modelul tabelului.
 - Se obține lista de produse utilizând metoda **findAll()** din obiectul **produsDAO**.
 - Dacă lista de produse nu este nulă, se parcurge lista și se adaugă rândurile corespunzătoare în modelul tabelului.

5. Metodele private pentru tratarea acțiunilor butoanelor:

- **getStartButton(), getClientButton(), getComandaButton():** Aceste metode tratează cazurile de apăsare ale butoanelor "Start", "Client" și "Comanda" și inițiază noi instanțe ale claselor **MainScreenController**, **ClientController** și **ComandaController**, respectiv, și apelează metoda **startLogic()** a acestora.
- **getViewButton():** Metoda tratează cazul de apăsare al butonului "View" și apelează metoda **getTable()** pentru afișarea tabelului de produse.
- **getEditButton():** Metoda tratează cazul de apăsare al butonului "Edit" și implementează logica pentru editarea unui produs.
 - Se obțin datele necesare pentru actualizarea produsului din câmpurile de text ale interfeței utilizator.
 - Se creează o instanță a clasei **Produs** cu datele introduse.
 - Se apelează metoda **update()** din obiectul **produsDAO** pentru a actualiza produsul în baza de date.
 - Se apelează metoda **getTable()** pentru a actualiza tabelul de produse.
- **getDeleteButton():** Metoda tratează cazul de apăsare al butonului "Delete" și implementează logica pentru ștergerea unui produs.
 - Se obține ID-ul produsului de șters din câmpul de text al interfeței utilizator.
 - Se apelează metoda **deleteObject()** din obiectul **produsDAO** pentru a șterge produsul din baza de date.
 - Se apelează metoda **getTable()** pentru a actualiza tabelul de produse.

- **getAddButton()**: Metoda tratează cazul de apăsare al butonului "Add" și implementează logica pentru adăugarea unui produs nou.
 - Se obțin datele necesare pentru adăugarea produsului din câmpurile de text ale interfeței utilizator.
 - Se creează o instanță a clasei **Produs** cu datele introduse.
 - Se apelează metoda **insert()** din obiectul **produsDAO** pentru a adăuga produsul în baza de date.
 - Se apelează metoda **getTable()** pentru a actualiza tabelul de produse.

Astfel, clasa **ProdusController** gestionează interacțiunea utilizatorului cu interfața și comunicarea cu obiectul din nivelul de acces la date (**ProdusDAO**) pentru operațiile CRUD (Create, Read, Update, Delete) asupra entității "Produs".

Clasa **MainScreenController** este responsabilă de gestionarea interacțiunii dintre interfața utilizator și codul aplicației în ceea ce privește ecranul principal al aplicației. Aceasta implementează logica corespunzătoare fiecărei acțiuni și gestionează evenimentele generate de apăsarea butoanelor din interfață.

Iată o explicație detaliată a clasei **MainScreenController**:

1. Importarea claselor necesare:
 - Se importă clasele necesare pentru gestionarea interacțiunii cu interfața utilizator și dependențele asociate.
2. Declararea clasei și a variabilei membru:
 - Clasa **MainScreenController** este declarată în pachetul **org.example.controller**.
 - Se declară variabila membru **startView** de tip **StartView**.
3. Metoda **startLogic()**:
 - Metoda inițializează obiectul **startView** cu o nouă instanță a clasei **StartView**.
 - Se utilizează **GUIFrameSinglePointAccess.changePanel()** pentru a schimba panel-ul curent în cadrul interfeței utilizator.
 - Se definesc acțiunile care trebuie executate la apăsarea fiecărui buton din interfață utilizând **addActionListener()**.

4. Metodele private pentru tratarea acțiunilor butoanelor:

- **getClientButton(), getComandaButton(), getProdusButton():** Aceste metode tratează cazurile de apăsare ale butoanelor "Client", "Comanda" și "Produs" și inițiază noi instanțe ale claselor **ClientController**, **ComandaController** și **ProdusController**, respectiv, și apelează metoda **startLogic()** a acestora.

Astfel, clasa **MainScreenController** gestionează interacțiunea utilizatorului cu interfața și inițiază logica pentru a trece la diferitele module ale aplicației prin intermediul butoanelor disponibile pe ecranul principal.

5. Rezultate

Meniul principal:



Panoul pentru realizarea operatiilor CRUD pe Client:



Client

ID Client

Nume

Email


Varsta

Add
Delete
Edit
View

id	nume	email	varsta
1	andrei	andrei@yahoo.com	25
2	mara	mara@yahoo.com	30

Start
Produs
Comanda

Panoul pentru realizarea operatiilor CRUD pe Produs:



Produs

ID Produs

Denumire produs


Stoc (numar articole disponibile)

Add
Delete
Edit
View

id	denumire	stoc
1	mar	4
3	castravete	4

Start
Client
Comanda

Panoul pentru realizarea operatiilor CRUD pe Comanda:



Comanda

ID Comanda

ID Client

ID Produs

Cantitate :

Add

Delete

Edit

View

id	idprodus	idclient	nr_produce_comandate
1	1	1	1
2	4	2	2

Start

Produs

Client

6. Concluzii

Concluzionand, acest proiect m-a ajutat sa invat sa folosesc tehnici de reflectie in Java, precum si sa genereze un fisier JavaDoc. Pe langa acestea mi-am dezvoltat si imbunatatit cunostintele in acest limbaj de programare.

La capitolul dezvoltari ulterioare pot propune: adăugarea funcționalității de validare a datelor introduse de utilizator în câmpurile de intrare, Implementarea unui sistem de autentificare și gestionare a utilizatorilor pentru a proteja aplicația și datele sensibile, adăugarea funcționalității de generare a rapoartelor și export în diferite formate (de exemplu, PDF, Excel).

7. Bibliografie

<https://www.oracle.com/technical-resources/articles/java/javareflection.html>

<https://www.dummies.com/article/technology/programming-web-design/java/how-to-use-javadoc-to-document-your-classes-153265/>

<https://dsrl.eu/courses/pt/materials/lectures/>

<https://mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>