UNIVERSITATEA TEHNICĂ "GHEORGHE ASACHI" IAŞI

FACULTATEA AUTOMATICĂ ŞI CALCULATOARE

SPECIALIZAREA CALCULATOARE ŞI TEHNOLOGIA INFORMAŢIEI

DISCIPLINA BAZE DE DATE

Gestiunea activității unui magazin de tipul duty-free

Coordonator,
Prof. Mironeanu Cătălin

Student,
Curuliuc Cosmin-Ștefan
Grupa 1306A

Iași, 2023

Titlu capitol **Pagina**

1. Descrierea proiectului

Peisajul comercial al magazinelor duty-free este într-o continuă schimbare, cu cerințe crescute și o diversitate mare de produse. Gestionarea eficientă a unui volum mare de informații și a proceselor operaționale complexe devine esențială pentru succesul acestor magazine. Proiectul propune dezvoltarea unui sistem informatic sofisticat, destinat îmbunătățirii proceselor operaționale ale unui magazin duty-free. Sistemul va adresa aspecte cheie, cum ar fi:

- **Gestionarea și Evidența Clientului**: Sistemul va asigura o evidență detaliată a clienților și a preferințelor acestora, permițând o interacțiune personalizată și eficientă.

- **Urmărirea și Administrarea Stocurilor**: Va oferi o gestionare precisă a stocurilor de produse, inclusiv categorii și prețuri, facilitând un management eficient al inventarului.

- **Procesarea Comenzilor**: De la plasarea comenzii până la livrare, sistemul va asigura un flux clar și eficient, îmbunătățind acuratețea și experiența clientului.

- **Gestionarea Angajaților**: Sistemul va oferi o structură clară pentru organizarea angajaților, inclusiv stabilirea rolurilor și ierarhiei, facilitând calculul salariilor și comisioanelor.

- **Integrarea cu Furnizorii**: Va asigura o colaborare strânsă cu furnizorii pentru aprovizionare eficientă, optimizând managementul resurselor.

2. Descrierea tehnologiilor folosite - Front-end

Aplicația este una de tip N-Tier întrucât avem un client care se conectează la un server și care are în spate o baza de date. În realizarea părții de Front-end a proiectului s-a folosit ca limbaj de programare Python 3.9, mai exact biblioteca **TKInter** din acest limbaj, întrucât acesta are o gamă largă și ușor de utilizat de obiecte vizuale precum: treeview, butoane, scroll, etc. Astfel, interfața creată este una ușor de folosit, întrucât datele din fiecare tabelă sunt la puține click- uri distanță, iar operațiile în cadrul acestora sunt intuitiv.

3. Descrierea tehnologiilor folosite - Back-end

Pentru partea de back-end a aplicației, a fost utilizat Python 3.9, mai exact biblioteca **OracleDB** din cadrul acestuia. Această ne pune la dispoziție un mod relativ ușor de a utiliza comenzi din limbajul Python împreună cu cele din SQL. Astfel, prin intermediul ei, am creat o conexiune la propria baza de date, apoi bazându-ne pe obiectul creat în urmă realizării cu succes a conexiunii, am reușit să trimitem diverse comenzi specifice limbajului SQL și afișarea rezultatului acestora în aplicația noastră.

De menționat , pentru partea de back-end cât și pentru partea de front-end, am folosit ca mediu de dezvoltare IDE PyCharm Professional Edition 2023.3, întrucât ne oferă o ușurință în instalarea diverselor biblioteci, dar și în rezolvarea diverselor erori.

Totodată , pentru crearea tabelelor cât și a bazei de date, am folosit IDE DataGrip, dar și Oracle Data Modeler.

4. Structura tabelelor

- **Clienți**: Această tabelă este esențială pentru a ști cine sunt clienții noștri și pentru a avea toate detaliile necesare în cazul în care plasează o comandă.
- **Adrese**: Importantă pentru a cunoaște adresele clienților și a altor entități relevante, această tabelă ne ajută să ne asigurăm că livrăm produsele acolo unde trebuie.
- **Angajați**: Informațiile din această tabelă ne arată câți angajați sunt implicați în procesarea comenzilor și, în caz de probleme, atât noi, cât și clienții știu la cine să apeleze.
- **Comenzi**: Utilizăm această tabelă pentru a înregistra și urmări toate detaliile legate de comenzile plasate de clienți, asigurându-ne astfel că fiecare comandă este gestionată corespunzător.
- **Produse**: Aici avem o listă a tuturor produselor disponibile în magazin, fiecare cu specificațiile sale, ajutându-ne să gestionăm eficient oferta noastră.
- **Furnizori**: Această tabelă conține informații despre furnizorii de la care achiziționăm produse, esențială pentru menținerea stocului și pentru relațiile comerciale.
- **DetaliiComenzi**: Această tabelă intermediară este folosită pentru a gestiona relațiile many-to-many, adică între Comenzi și Produse, permițându-ne să vedem exact ce produse sunt incluse în fiecare comandă.

5. Inter-relaționarea entităților

Analizând relațiile dintre aceste entități, am identificat conexiuni de tip 1:1 și 1:n, precum și relații many-to-many gestionate prin tabele intermediare.

A) Relații 1:n

- Relația dintre **Clienți** și **Comenzi** este de tip 1:n, unde un client poate plasa multiple comenzi, iar fiecare comandă este asociată cu un singur client, asigurându-ne că comenzile sunt unice pentru fiecare client. Conexiunea este realizată prin atributul **ClientID**.
- Similar, există o relație de tip 1:n între **Angajați** și **Comenzi**, în care un angajat este responsabil pentru mai multe comenzi, gestionând asistența clienților în procesul de plasare a acestora. Fiecare comandă este legată de un angajat printr-un **AngajatID**, permițând astfel segregarea responsabilităților.
- **Produsele** sunt, de asemenea, legate de **Furnizori** printr-o relație de tip 1:n, unde un furnizor poate oferi mai multe produse, dar fiecare produs provine de la un singur furnizor, asigurându-ne astfel de o aprovizionare corespunzătoare a stocurilor. Legătura se realizează prin **FurnizorID**.

B) Relații 1:1

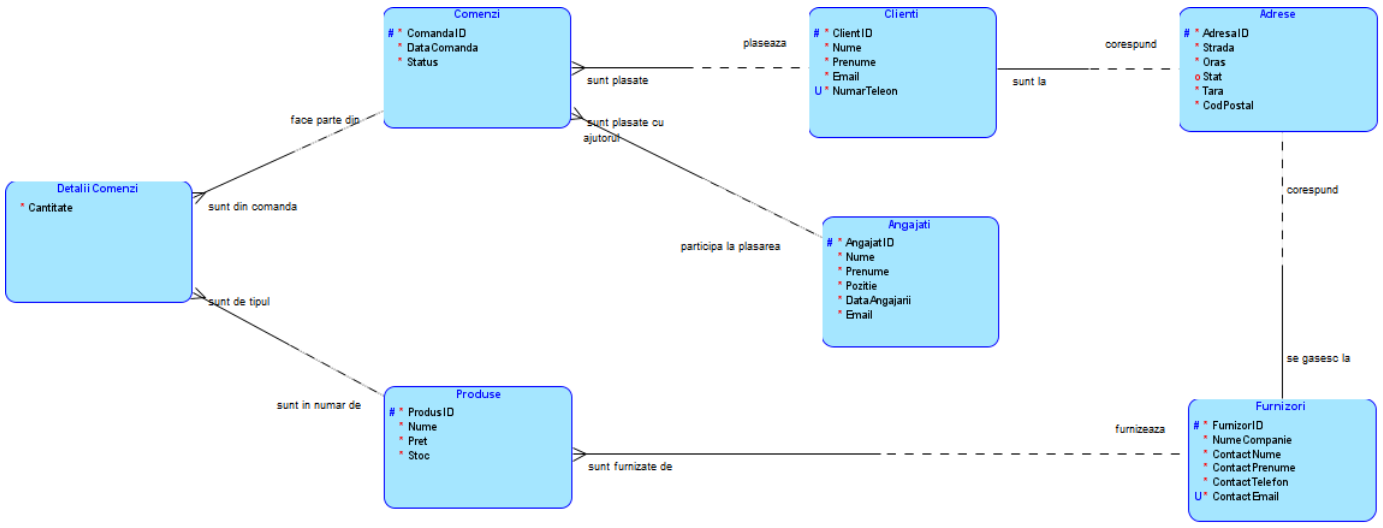
- Fiecărui **Client** și **Furnizor** este asociată cu o singură **Adresă** printr-o relație de tip 1:1, garantând că fiecare comandă este livrată la o adresă specifică, fără ambiguitate. Acest lucru este posibil prin intermediul atributului **AdresaID**.

C) Relații many-to-many

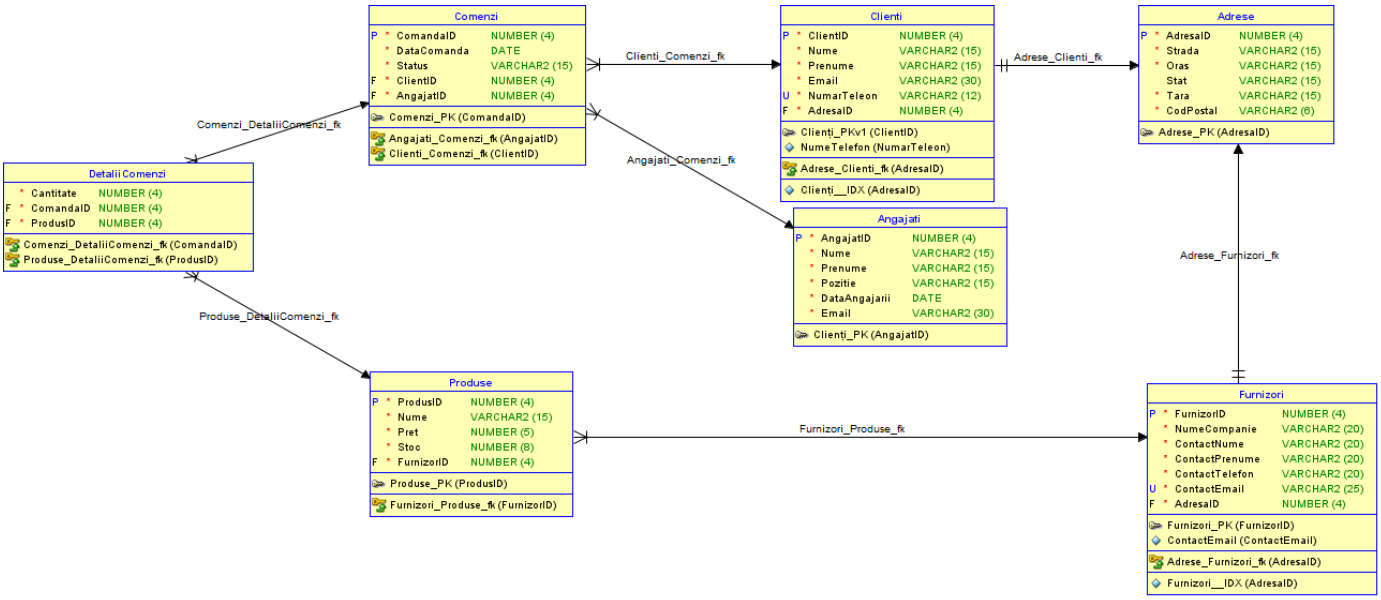
- O relație many-to-many între **Comenzi** și **Produse** este prezentă și este mediată de entitatea **DetaliiComenzi**, care permite plasarea mai multor produse în cadrul unei singure comenzi și viceversa. Acest tabel intermediar folosește **ComandaID** și **ProdusID** pentru a gestiona relațiile complexe dintre comenzi și produse, precizând cantitatea în care s-a cumpărat un produs pentru o anumită comandă.

Modelul logic

Un Produs poate fi inclus în mai multe Comenzi, și o Comanda poate conține mai multe Produse. Această relație este gestionată prin tabela DetaliiComenzi.



Modelul relațional



6. Aspecte legate de normalizare

Schema noastră de bază de date respectă principiile normalizării până la a treia formă normală, asigurând eficiență și coerență în structura datelor:

A) **Prima Formă Normală (1NF)** este îndeplinită în toate tabelele schema noastră. Acest lucru este demonstrat prin faptul că fiecare câmp în tabele conține valori unice și atomice, fără grupuri repetitive de valori. De exemplu, în tabela **Angajati**, fiecare înregistrare conține un singur număr de telefon per angajat, evitând astfel orice repetiție.

B) **A Doua Formă Normală (2NF)** este realizată deoarece fiecare tabel este deja în 1NF și toate atributele non-cheie depind în totalitate de cheia primară. În tabela **Comenzi**, **ComandaID** este cheia primară unică, iar toate celelalte atribute în tabela sunt funcțional dependente de aceasta.

C) **A Treia Formă Normală (3NF)** este aplicată deoarece toate atributele non-cheie sunt direct și non-tranzitiv dependente de cheia primară. De exemplu, în relația dintre **Comenzi** și **Adrese**, fiecare comandă este legată direct de o singură adresă prin **AdresaID**, fără a avea dependențe tranzitive între acestea.

În plus, relația many-to-many dintre **Comenzi** și **Produse** este gestionată prin tabela intermediară **DetaliiComenzi**, care asigură că fiecare produs dintr-o comandă este unic identificat, menținând astfel schema în 3NF prin eliminarea dependențelor tranzitive și îmbunătățind integritatea referențială.

Prin aceste măsuri, ne asigurăm că baza de date este structurată eficient, prevenind anomalii de inserție, actualizare sau ștergere și facilitând interogări rapide și precise.

7. Descrierea constrângerilor

Constrângerile de Tip Check:

- Verificarea corectitudinii valorilor introduse pentru nume, prenume, oraș, județ, țară, număr de telefon, cod poștal.
- Validarea formatului e-mailului, care poate fi alcătuit doar din litere, cifre, "@" și ".".
- Asigurarea că sumele introduse pentru prețuri și stoc nu sunt negative.

Constrângeri de Tip Unic:

- Unicitatea emailurilor, a numerelor de telefon și a codurilor poștale pentru a evita duplicarea în rândul clienților sau angajaților.

Constrângeri de Tip Not Null:

- Majoritatea atributelor din entități trebuie să aibă valori și nu pot fi nule.

Triggere:

- Asigurarea că data plasării unei comenzi și data angajării unui angajat nu depășesc ziua curentă.

Primary Keys Autoincrement:

- Generarea automată a cheilor primare pentru entități prin mecanisme de autoincrement.

Constrângeri Adiționale Specificate Anterior:

- Validarea Datelor de Intrare: Numele clienților și angajaților nu trebuie să conțină cifre sau caractere speciale. Adresele de email trebuie să fie validate conform unui format standard.
- Constrângeri de Tip Check pentru Produse: Verificarea că prețul produselor este pozitiv și că descrierea produselor nu conține caractere speciale.
- Constrângeri de Tip Check pentru liste: În cazul Angajaților (pentru atributul Poziție), respectiv Comenzilor (pentru atributul Status), există un set predefinit de date din care utilizatorul poate alege și care sunt sugerate cu ajutorul un Combobox.

8. Conectarea la baza de date

Așa cum aminteam anterior conexiunea la baza de date este făcută prin intermediul bibliotecii OracleDb din Python 3.9. Această permite crearea unui obiect de tip conexiune pe care îl putem salva într-o variabilă . În momentul în care operația de conectare s-a realizat cu succes, vom avea la îndemână un obiect de tip conexiune pe care îl vom putea folosi pentru a crea un cursor prin intermediul căruia vom putea execută diverse comenzi specifice SQL, cum ar fi : insert, update, delete, drop, savepoint, rollback, commit etc…

Funcțiile care realizează operațiile amintite anterior sunt:

* def **connection**(name: str, password: str, host: str, port: str, service_name: str) -> list:
Prin intemediul acestei funcții ne conectăm la baza noastră de date.

* def **close_connection**():
Cu ajutorul ei închidem conexiunea la baza noastră de date, întrucât dacă nu am face acest lucru, aglomeram baza de date cu conexiuni la care nu avem acces, iar acesta va începe a rula mai greu.

* def **select_from_table**(table_name: str) -> list:
Cu ajutorul ei facem selecția dintr -o tabela primită că parametru și returnăm o lista . Această selecție se realizează în tocmai că în limbajul SQL, diferența făcând -o modul de stocare a rezultatului.

* def **update_table**(table_name: str, column_name: str, values: str, condition: str):
Prin intermediul ei updatam o linie dintr-o tabela, acest lucru petrecându-se la fel că în SQL.

* def **add_savepoint**(savePointName, popup):
Cu ajutorul ei creăm diverse savepoint- uri pe care le adăugăm într -o lista.

* def **rollback_to_savepoint**(name, popup):
Prin intermediul listei de savepoint- urîm , utilizatorul poate selecta la care savepoint dorește a se întoarce.

* def **commit**():
Prin intermediul ei, confirm modificările făcute bazei de date.

9. Operația de tranzacție

În contextul bazelor de date, o tranzacție este o unitate de lucru care poate fi efectuată într-un singur set sau poate fi anulată într-un singur set. În general, o tranzacție este un grup de operațiuni care trebuie să fie efectuate împreună, astfel încât rezultatul lor să fie valid. Dacă una dintre operațiuni eșuează, întreaga tranzacție este anulată, astfel încât baza de date să rămână în stare consistentă.

Tranzacțiile sunt importante deoarece asigură integritatea datelor în baza de date. Dacă o tranzacție eșuează, baza de date rămâne în stare consistentă, astfel încât datele sunt în siguranță.

În cadrul proiectului a fost implementată o tranzacție pentru inserarea datelor în două tabele dinstincte, clienți , respectiv adrese. Astfel, fiecare client când dorește a se înregistra, după ce a introdus adresa, va fi trimis într-o nouă fereastră pentru a-și completa datele personale. În cazul în care, din diverse motive, clientul nu reușește în a-și completă adresa, el va fi șters din tabela de clienți, iar baza de date restabilită cu ajutorul rollback-ului la formă de dinaintea înserării acestuia. În schimb, dacă reușește a insera cu success, atât clientul cât și adresa lui vor apărea în baza de date, această modificare devenind permanente cu ajutorul comenzii de commit.

Funcțiile apelate pentru a realiza această operație sunt insert_into_clienti, respectiv insert_into_adrese. Cu ajutorul lor, ne asigurăm că operațiile pe care dorim să le facem în baza de date se realizează cu succes.

Funcțiile anterior menționate:

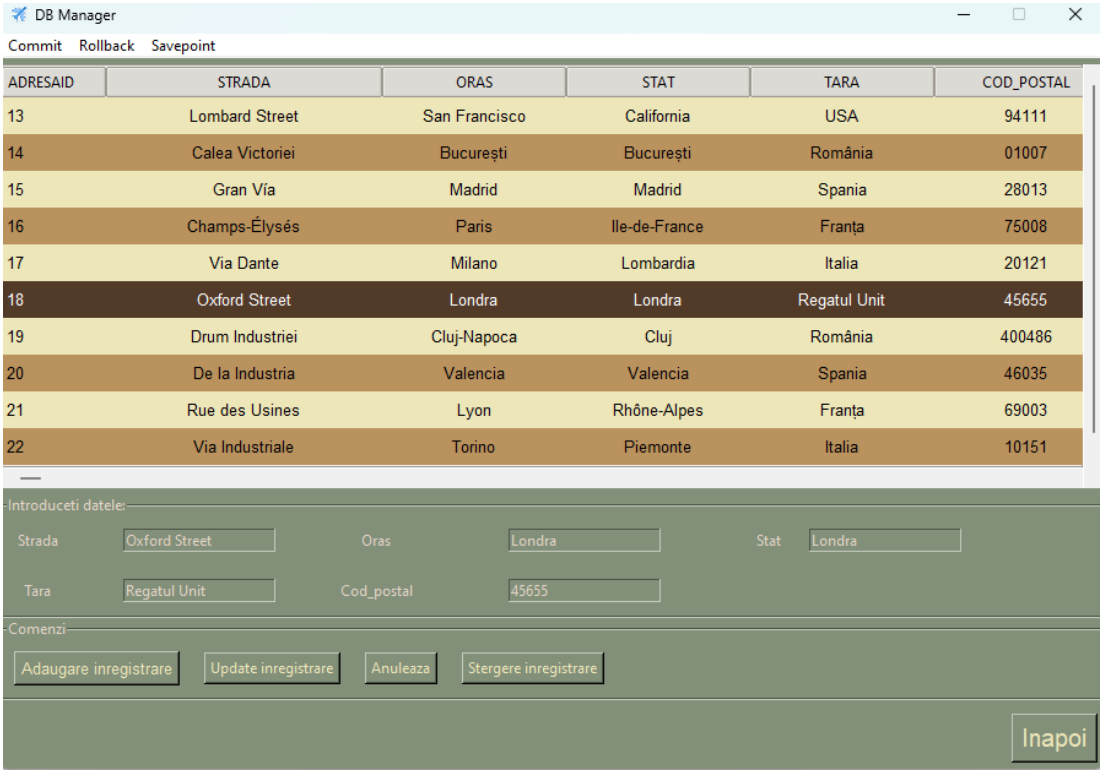
```python
def insert_into_clienti(window, tree, b, d, f, s, table, list):
    error = ''
    window.pack_forget()
    tree.forget()
    b.forget()
    d.forget()
    f.forget()
    s.forget()
    error = backend.insert_into_table_clienti(table, list)
    if error == 'WRONG':
        adrese_info()
    elif (error is None and
          len(backend.select_from_table(backend.Table_names[2])) == len(backend.select_from_table(table))):
        clienti_info()
    elif len(backend.select_from_table(backend.Table_names[2])) < len(backend.select_from_table(table)):
        adrese_info()
```

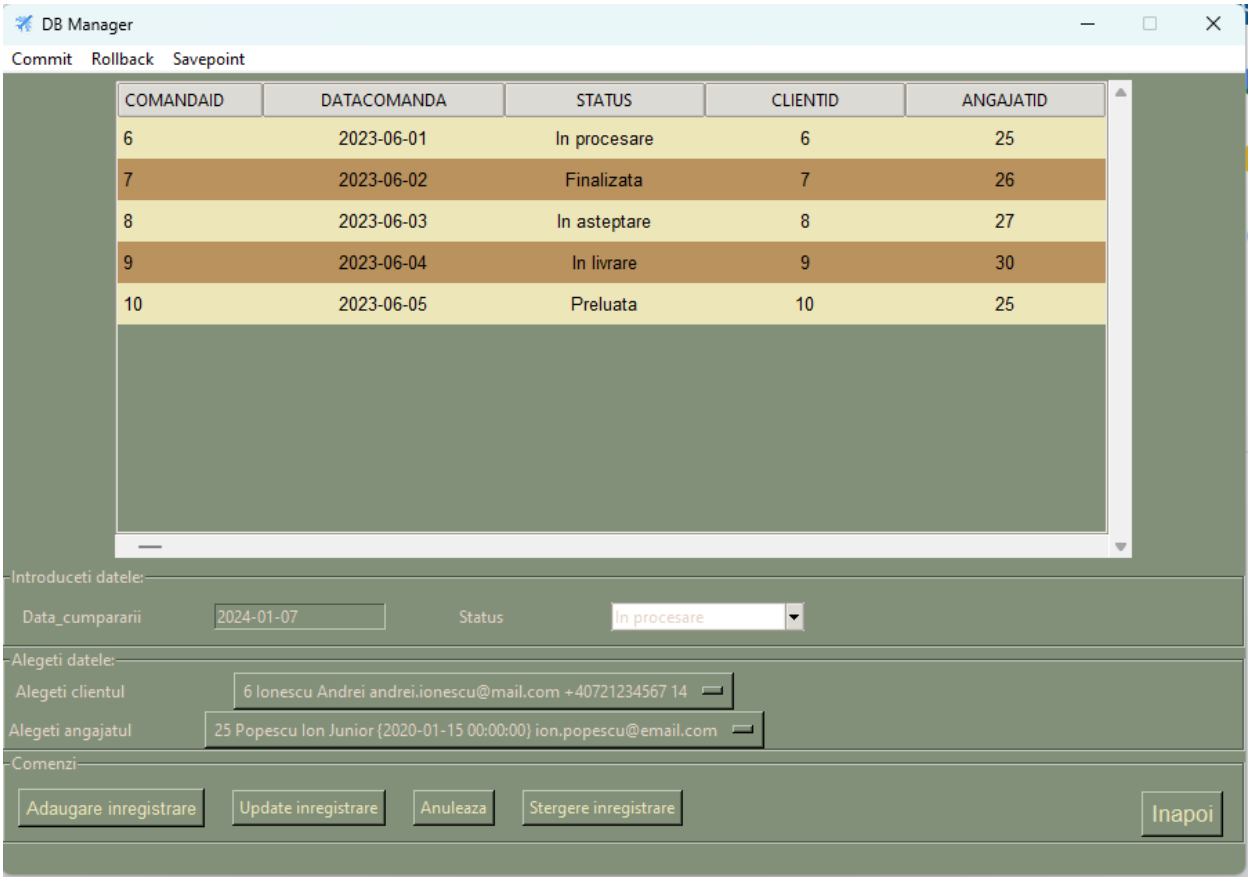
```python
def insert_into_adrese(window, tree, b, d, f, table, list):
    error = ''
    window.pack_forget()
    tree.forget()
    b.forget()
    d.forget()
    f.forget()
    error = backend.insert_into_table_adrese(table, list)
    if error == 'WRONG':
        adrese_info()
    else:
        clienti_info()
```

```python
def insert_into_table_clienti(table_name: str, values: list):
    global conn
    values_String = '('
    for x in values:
        values_String += convert_to_sql(x)

    values_String = values_String[:len(values_String) - 2] + ')'

    try:
        with conn.cursor() as cur:
            cur.execute('SAVEPOINT TEMP2')
            cur.execute('INSERT INTO ' + table_name + ' VALUES ' + values_String)
            if len(select_from_table(Table_names[0])) > len(select_from_table(table_name)):
                cur.execute('ROLLBACK TO TEMP')
            elif len(select_from_table(Table_names[0])) < len(select_from_table(table_name)):
                cur.execute('ROLLBACK TO TEMP2')
            else:
                cur.execute('COMMIT')
    except oracledb.DatabaseError as err:
        errors = 'WRONG'
        conn.rollback()
        message = 'Error while inserting into table: ' + str(err)
        tk.messagebox.showerror(title='Error', message=message)
        return errors
```

```python
def insert_into_table_adrese(table_name: str, values: list):
    global conn
    values_String = '('
    for x in values:
        values_String += convert_to_sql(x)

    values_String = values_String[:len(values_String) - 2] + ')'

    try:
        with conn.cursor() as cur:
            cur.execute('SAVEPOINT TEMP')
            cur.execute('INSERT INTO ' + table_name + ' VALUES ' + values_String)
    except oracledb.DatabaseError as err:
        errors = 'WRONG'
        conn.rollback()
        message = 'Error while inserting into table: ' + str(err)
        tk.messagebox.showerror(title='Error', message=message)
        return errors
```

10. Capturi de ecran cu interfața aplicației


DB Manager
Commit
Rollback
Savepoint
ADRESAID
STRADA
ORAS
STAT
TARA
COD_POSTAL
13
Lombard Street
San Francisco
California
USA
94111
14
Calea Victoriei
București
București
România
01007
15
Gran Vía
Madrid
Madrid
Spania
28013
16
Champs-Élysés
Paris
Ile-de-France
Franța
75008
17
Via Dante
Milano
Lombardia
Italia
20121
18
Oxford Street
Londra
Londra
Regatul Unit
45655
19
Drum Industriei
Cluj-Napoca
Cluj
România
400486
20
De la Industria
Valencia
Valencia
Spania
46035
21
Rue des Usines
Lyon
Rhône-Alpes
Franța
69003
22
Via Industriale
Torino
Piemonte
Italia
10151
Introduceti datele:
Strada
Oxford Street
Oras
Londra
Stat
Londra
Tara
Regatul Unit
Cod_postal
45655
Comenzi
Adaugare inregistrare
Update inregistrare
Anuleaza
Stergere inregistrare
Inapoi


DB Manager
Commit
Rollback
Savepoint
COMANDAID
DATACOMANDA
STATUS
CLIENTID
ANGAJATID
6
2023-06-01
In procesare
6
25
7
2023-06-02
Finalizata
7
26
8
2023-06-03
In asteptare
8
27
9
2023-06-04
In livrare
9
30
10
2023-06-05
Preluata
10
25
Introduceti datele:
Data_cumpararii
2024-01-07
Status
In procesare
Alegeti datele:
Alegeti clientul
6 Ionescu Andrei andrei.ionescu@mail.com +40721234567 14
Alegeti angajatul
25 Popescu Ion Junior {2020-01-15 00:00:00} ion.popescu@email.com
Comenzi
Adaugare inregistrare
Update inregistrare
Anuleaza
Stergere inregistrare
Inapoi

11. Capturi de ecran cu exemple de cod și instrucțiuni SQL folosite

```python
def run_query(query: str):
    global conn
    res = []
    with conn.cursor() as cur:
        try:
            cur.execute(query)
        except Exception as err:
            message = 'Error while running query: ' + str(err)
            tk.messagebox.showerror(title='Error', message=message)
            return []

        try:
            for x in cur:
                res.append(x)

        finally:

            return res
```

```python
def connection(name: str, password: str, host: str, port: int, service_name: str) -> list:
    global conn, dsn
    try:
        dsn = oracledb.makedsn(host, port, service_name=service_name)

        conn = oracledb.connect(user=name, password=password, dsn=dsn)

        with conn.cursor() as cursor:
            cursor.execute("SELECT TABLE_NAME FROM USER_TABLES ORDER BY TABLE_NAME")
            table_names = [row[0] for row in cursor]

        print("\nConnection established.\n")
        return table_names

    except Exception as err:
        print("Error while creating the connection: ", str(err))
        return []
```

```python
def select_from_table(table_name: str) -> list:
    global conn, row
    try:
        row = run_query('SELECT * FROM ' + table_name)
    except oracledb.DatabaseError as err:
        errors = 'WRONG'
        conn.rollback()
        message = 'Error while inserting into table: ' + str(err)
        tk.messagebox.showerror(title='Error', message=message)
        return errors
    return row
```

```python
def update_table(table_name: str, column_name: str, values: str, condition: str):
    try:
        conn.cursor().execute(
            'UPDATE ' + table_name + ' SET ' + column_name + ' = \'' + values + '\' WHERE \'' + condition + '\'')
    except oracledb.DatabaseError as err:
        print('Error while updating table: ', err)
```

```python
def insert_into_table(table_name: str, values: list):
    global conn
    values_String = '('
    for x in values:
        values_String += convert_to_sql(x)

    values_String = values_String[:len(values_String) - 2] + ')'

    try:
        run_query('INSERT INTO ' + table_name + ' VALUES ' + values_String)
    except oracledb.DatabaseError as err:
        errors = 'WRONG'
        conn.rollback()
        message = 'Error while inserting into table: ' + str(err)
        tk.messagebox.showerror(title='Error', message=message)
        return errors
```