DOCUMENTATIE

TEMA 3

NUME STUDENT: Simina Dan-Marius
GRUPA: 30222

CUPRINS

1. Obiectivul temei

Obiectiv principal:

- Proiectați și implementați o aplicație pentru gestionarea comenzilor clienților pentru un depozit

Sub-obiective:

- Analizați problema și identificați cerințele
- Proiectați aplicația de gestionare a comenzilor
- Implementați aplicația de gestionare a comenzilor
- Testați aplicația de gestionare a comenzilor

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

Cerințe funcționale:

- Aplicația ar trebui să permită unui angajat să adauge un client nou
- Aplicația ar trebui să permită unui angajat să adauge un produs nou

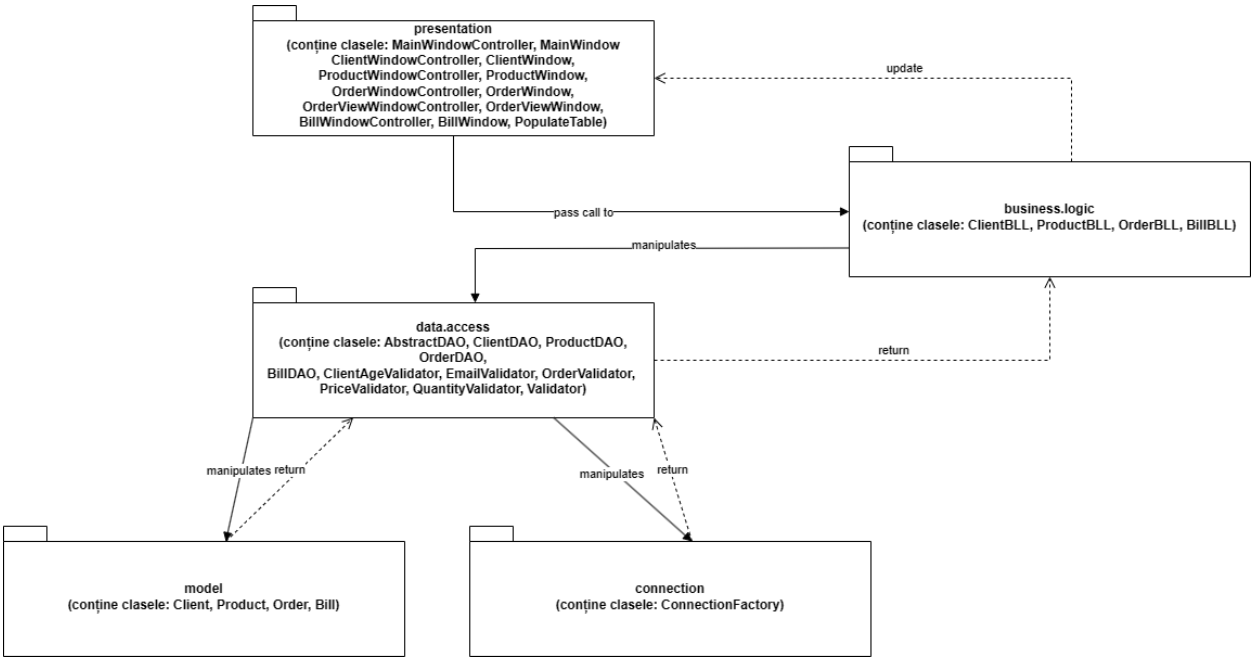
Cerințe non-funcționale:

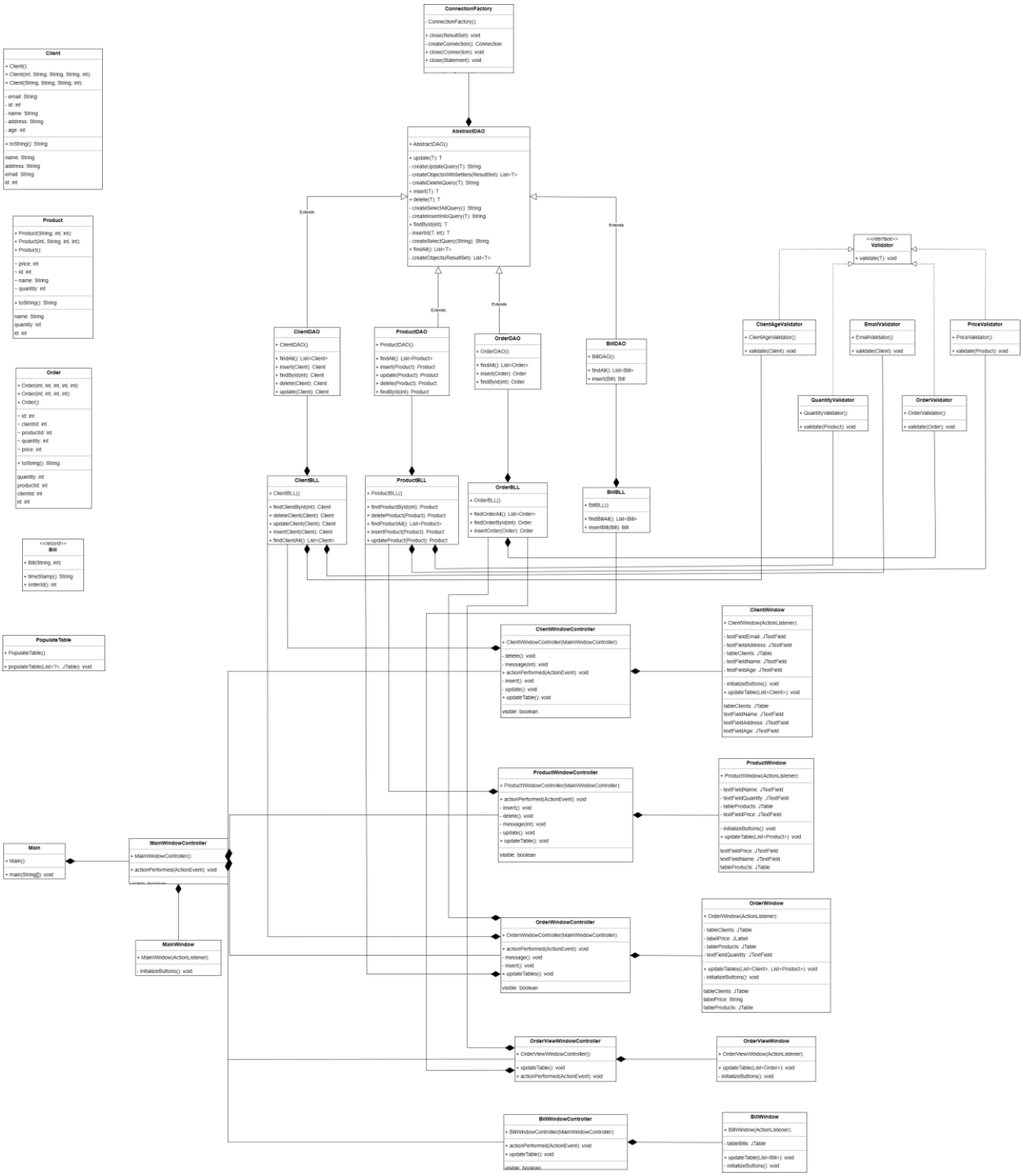
- Aplicația ar trebui să fie intuitivă și ușor de folosit de către utilizator.

Scenarii de utilizare:

- Scenariu de succes principal:
  1) Utilizatorul selectează fereastra în care vrea să meargă din meniul principal.
  2) În funcție de fereastra aleasă utilizatorului îi sunt puse la dispozitie toate acțiunile pe care are voie sa le faca: vizualizare/iserare/editare/stergere.
  3) Utilizatorul introduce datele corecte pentru acțiunea pe care dorește sa realizeze.
  4) Apasă butonul aferent acțiunii pe care dorește să o realizeze.
  5) Se face update în fereastra din aplicație și în baza de date.
- Secvență alternativă: Utilizatorul a introdus valori invalide pentru acțiunea pe care dorește sa o facă.
  - Utilizatorul primește un mesaj de eroare.
  - Nu se actualizează nimic atât în interfață dar și în baza de date.

3. Proiectare




ConnectionFactory
- ConnectionFactory()
+ close(ResultSet): void
- createConnection(): Connection
+ close(Connection): void
+ close(Statement): void
Client
+ Client()
+ Client(int, String, String, String, int)
+ Client(String, String, String, int)
- email: String
- id: int
- name: String
- address: String
- age: int
+ toString(): String
name: String
address: String
email: String
id: int
AbstractDAO
+ AbstractDAO()
+ update(T): T
- createUpdateQuery(T): String
- createObjectsWithSetters(ResultSet): List<T>
- createDeleteQuery(T): String
+ insert(T): T
+ delete(T): T
- createSelectAllQuery(): String
- createInsertIntoQuery(T): String
+ findById(int): T
- insert(T, int): T
- createSelectQuery(String): String
+ findAll(): List<T>
- createObjects(ResultSet): List<T>
Extends
Extends
Extends
Extends
Product
+ Product(String, int, int)
+ Product(int, String, int, int)
+ Product()
~ price: int
~ id: int
~ name: String
~ quantity: int
+ toString(): String
name: String
quantity: int
id: int
ClientDAO
+ ClientDAO()
+ findAll(): List<Client>
+ insert(Client): Client
+ findById(int): Client
+ delete(Client): Client
+ update(Client): Client
ProductDAO
+ ProductDAO()
+ findAll(): List<Product>
+ insert(Product): Product
+ update(Product): Product
+ delete(Product): Product
+ findById(int): Product
OrderDAO
+ OrderDAO()
+ findAll(): List<Order>
+ insert(Order): Order
+ findById(int): Order
BillDAO
+ BillDAO()
+ findAll(): List<Bill>
+ insert(Bill): Bill
<<interface>>
Validator
+ validate(T): void
ClientAgeValidator
+ ClientAgeValidator()
+ validate(Client): void
EmailValidator
+ EmailValidator()
+ validate(Client): void
PriceValidator
+ PriceValidator()
+ validate(Product): void
QuantityValidator
+ QuantityValidator()
+ validate(Product): void
OrderValidator
+ OrderValidator()
+ validate(Order): void
Order
+ Order(int, int, int, int)
+ Order(int, int, int, int)
+ Order()
~ id: int
~ clientId: int
~ productId: int
~ quantity: int
~ price: int
+ toString(): String
quantity: int
productId: int
clientId: int
id: int
ClientBLL
+ ClientBLL()
+ findClientById(int): Client
+ deleteClient(Client): Client
+ updateClient(Client): Client
+ insertClient(Client): Client
+ findClientAll(): List<Client>
ProductBLL
+ ProductBLL()
+ findProductById(int): Product
+ deleteProduct(Product): Product
+ findProductAll(): List<Product>
+ insertProduct(Product): Product
+ updateProduct(Product): Product
OrderBLL
+ OrderBLL()
+ findOrderAll(): List<Order>
+ findOrderById(int): Order
+ insertOrder(Order): Order
BillBLL
+ BillBLL()
+ findBillAll(): List<Bill>
+ insertBill(Bill): Bill
<<interface>>
Bill
+ Bill(String, int):
+ timeStamp(): String
+ orderId(): int
PopulateTable
+ PopulateTable():
+ populateTable(List<?>, JTable): void
ClientWindowController
+ ClientWindowController(MainWindowController):
- delete(): void
- message(int): void
+ actionPerformed(ActionEvent): void
- insert(): void
- update(): void
+ updateTable(): void
visible: boolean
ClientWindow
+ ClientWindow(ActionListener):
- textFieldEmail: JTextField
- textFieldAddress: JTextField
- tableClients: JTable
- textFieldName: JTextField
- textFieldAge: JTextField
- initializeButtons(): void
+ updateTable(List<Client>): void
tableClients: JTable
textFieldName: JTextField
textFieldAddress: JTextField
textFieldAge: JTextField
ProductWindowController
+ ProductWindowController(MainWindowController):
+ actionPerformed(ActionEvent): void
- insert(): void
- delete(): void
- message(int): void
- update(): void
+ updateTable(): void
visible: boolean
ProductWindow
+ ProductWindow(ActionListener):
- textFieldName: JTextField
- textFieldQuantity: JTextField
- tableProducts: JTable
- textFieldPrice: JTextField
- initializeButtons(): void
+ updateTable(List<Product>): void
textFieldPrice: JTextField
textFieldName: JTextField
tableProducts: JTable
Main
+ Main():
+ main(String[]): void
MainWindowController
+ MainWindowController():
+ actionPerformed(ActionEvent): void
MainWindow
+ MainWindow(ActionListener):
- initializeButtons(): void
OrderWindowController
+ OrderWindowController(MainWindowController):
+ actionPerformed(ActionEvent): void
- message(): void
- insert(): void
+ updateTables(): void
visible: boolean
OrderWindow
+ OrderWindow(ActionListener):
- tableClients: JTable
- labelPrice: JLabel
- tableProducts: JTable
- textFieldQuantity: JTextField
+ updateTables(List<Client>, List<Product>) void
- initializeButtons(): void
tableClients: JTable
labelPrice: String
tableProducts: JTable
OrderViewWindowController
+ OrderViewWindowController():
+ updateTable(): void
+ actionPerformed(ActionEvent): void
OrderViewWindow
+ OrderViewWindow(ActionListener):
+ updateTable(List<Order>) void
- initializeButtons(): void
BillWindowController
+ BillWindowController(MainWindowController):
+ actionPerformed(ActionEvent): void
+ updateTable(): void
visible: boolean
BillWindow
+ BillWindow(ActionListener):
- tableBills: JTable
+ updateTable(List<Bill>) void
- initializeButtons(): void

4. Implementare

Clasa **Client** modelează tipul de dată client.
Clasa **Product** modelează tipul de dată product.
Clasa **Order** modelează tipul de dată order.
Clasa **Bill** modelează tipul de dată bill.

Clasa **AbstractDAO<T>** - clasă generică care implementează operațiile realizate asupra bazei de date.

- Metoda **List<T> findAll()**: preia toate inregistrarile din baza de date.
- Metoda **T findById(int id)**: preia inregistrarea cu id-ul transmis ca parametru, din baza de date.
- Metoda **T insert(T t)**: inserează obiectul transmis ca paramentru in baza de date.
- Metoda **T update(T t)**: actualizează datele pentru obiectul transmis ca parametru.
- Metoda **T delete(T t)**: sterge obiectul transmis ca parametru, din baza de date.
- Metoda **List<T> createObjects(ResultSet resultSet)**: crează o listă de obiecte pe baza rezultatelor obținute în urma interogării, încearcă mai întâi să folosească constructorul cu toți parametrii, iar dacă nu reușește foloește constructorul fără parametrii și settere.

Clasa **ClientDAO** extinde clasa **AbstractDAO** pentru tipul de dată **Client**.
Clasa **ProductDAO** extinde clasa **AbstractDAO** pentru tipul de dată **Product**.
Clasa **OrderDAO** extinde clasa **AbstractDAO** pentru tipul de dată **Order**.
Clasa **BillDAO** extinde clasa **AbstractDAO** pentru tipul de dată **Bill**.

Interfața **Validator<T>**: conține definiția metodei pentru validarea datelor.
Clasa **ClientAgeValidator** implementează interfața **Validator<T>**, conține doar metoda **validate(Client t)** care verifică ca varsta clientului primit ca parametru să fie într-un anumit interval, în caz contrar aruncă excepție.
Clasa **EmailValidator** implementează interfața **Validator<T>**, conține doar metoda **validate(Client t)** care verifică ca adresa de e-mail a clientului primit ca parametru să respecte un anumit model, în caz contrar aruncă excepție.
Clasa **OrderValidator** implementează interfața **Validator<T>**, conține doar metoda **validate(Order order)** care verifică ca obiectul comandă primit ca parametru să conțina un client și un produs valid, iar cantitatea introdusă să fie la rândul ei validă, în caz contrar aruncă excepție.
Clasa **PriceValidator** implementează interfața **Validator<T>**, conține doar metoda **validate(Product t)** care verifică ca prețul produsului primit ca parametru să fie într-un anumit interval, în caz contrar aruncă excepție.
Clasa **QuantityValidator** implementează interfața **Validator<T>**, conține doar metoda **validate(Product t)** care verifică ca, cantitatea produsului primit ca parametru să fie într-un anumit interval, în caz contrar aruncă excepție.

Clasa **ClientBLL** mijlocește interacțiunea cu baza de date pentru obiectele de tip **Client** pentru fiecare operație asupra bazei de date, dacă este necesar aplică metodele de validare, iar daca validarea se încheie cu succes se apelează metoda aferentă din clasa **ClientDAO**.
Clasa **ProductBLL** mijlocește interacțiunea cu baza de date pentru obiectele de tip **Product** pentru fiecare operație asupra bazei de date, dacă este necesar aplică metodele de validare, iar daca validarea se încheie cu succes se apelează metoda aferentă din clasa **ProductDAO**.
Clasa **OrderBLL** mijlocește interacțiunea cu baza de date pentru obiectele de tip **Order** pentru fiecare operație asupra bazei de date, dacă este necesar aplică metodele de validare, iar daca validarea se încheie cu succes se apelează metoda aferentă din clasa **OrderDAO**.
Clasa **BillBLL** mijlocește interacțiunea cu baza de date pentru obiectele de tip **Bill** pentru fiecare operație asupra bazei de date, dacă este necesar aplică metodele de validare, iar daca validarea se încheie cu succes se apelează metoda aferentă din clasa **BillDAO**.

Clasa **MainWindowController** reprezintă controller-ul pentru clasa **MainWindow** care implementează fereastra principală de unde utilizatorul poate sa navigheze între ferestrele Client, Pruduct, Order, Bill.
Clasa **ClientWindowController** reprezintă controller-ul pentru clasa **ClientWindow** care implementează interfața grafică pentru manipularea clinetilor.
Clasa **ProductWindowController** reprezintă controller-ul pentru clasa **ProductWindow** care implementează interfața grafică pentru manipularea produselor.

Clasa **OrderWindowController** reprezintă controller-ul pentru clasa **OrderWindow** care implementează interfața grafică pentru crearea comenzilor.
Clasa **OrderViewWindowController** reprezintă controller-ul pentru clasa **OrderViewWindow** care implementează interfața grafică pentru vizualizarea comenzilor.
Clasa **BillWindowController** reprezintă controller-ul pentru clasa **BillWindow** care implementează interfața grafică pentru vizualizarea chitanțelor.

Clasa **PopulateTable** conține metoda statică **void populateTable(List<?> objectList, JTable table)** care primește o listă de obiecte și un tabel, iar apoi setează numele pentru coloanele tabelului și inserează obiectele.

5. Rezultate

Am introdus inregistrari, actualizări și ștergeri, prin intermediul ferestrei de clienti și produse, am efectuat comenzi, iar rezultatele au fost cele asteptate, totul a mers bine cand datele au fost introuse corect, iar la input-uri greșite am primit mesaj de eroare.

6. Concluzii

În urma acestei teme am învățat ce este și cum funcționează java reflection și toate avantajele pe care le aduce, ușurându-mi munca foarte mult în interacțiunea cu baza de date. Ca posibilități de dezvoltare ulterioară aplicația poate să fie dezvoltată foarte mult până la nivelul unei aplicații de gestionare adevărate, se pot adăuga mai multe acțiuni pe care utilizatorul le poate accesa, adaugare de conturi de utilizator, etc.

7. Bibliografie

ASSIGNMENT 3 – SUPPORT PRESENTATION (PART I)
ASSIGNMENT 3 – SUPPORT PRESENTATION (II)
Stack Overflow - Where Developers Learn, Share, & Build Careers
Introduction to JavaDoc | Baeldung
Java Reflection Tutorial (jenkov.com)