FUNDAMENTAL PROGRAMMING TECHNIQUES

Assignment 2

Order Management

Student: Halmai Erik

Group: 30422

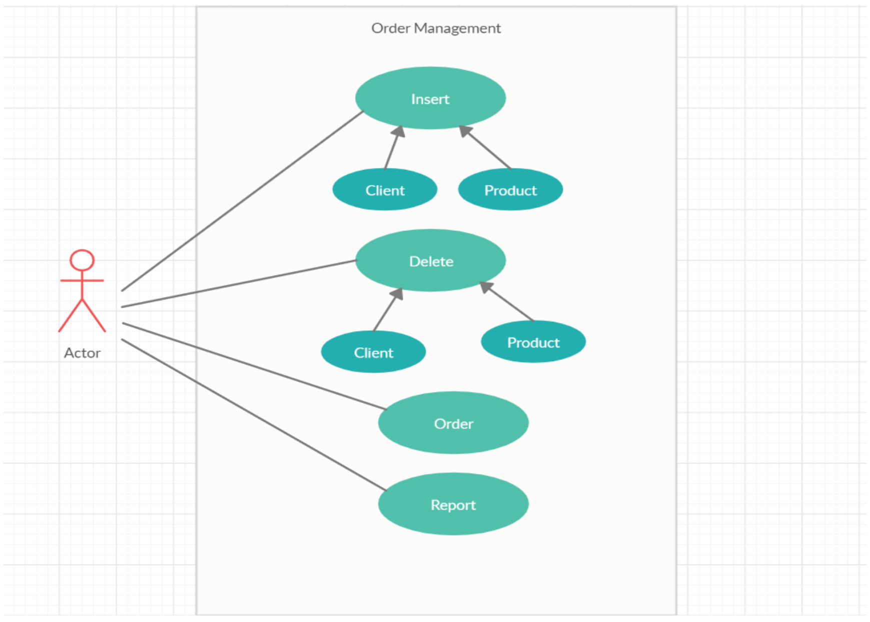
1. Objective

The main objective of the assignment is the creation of an application that processes customer orders for a warehouse. The application should use relational databases to store the products, the clients and the orders from the warehouse. Furthermore, it should be structured in packages using a layered architecture and it should use reflection techniques to create generic classes that contain methods for accessing the database with dynamically generated queries through reflection.

The main objective of the assignment can be divided into several secondary objectives, each achieved separately. These represent the steps that are taken in order to achieve the main objective of the assignment. These steps are the following:

1) **Layered architecture (Chapter 3)**: The application will contain 5 main packages: model, presentation, databaseAccessLayer, businessLogicLayer and connection.
2) **Creation of the relational database (Chapter 3)**: The database will be designed in MySQL Workbench. It will contain 4 tables: client, product, order and orderitem with appropriate links between them. The presentation layer will contain a parser to read and execute the commands.
3) **Creation of objects to store clients, orders and products (Chapter 4)**: These objects need to have the same fields as the database tables.
4) **Creation of the connection between the database and the application (Chapter 4)**: The connection will be made with reflection techniques, using queries to execute different tasks in the tables.
5) **Parsing the commands from the input file (Chapter 4):** An input text file will be given, containing commands respecting the specific structures discussed below.
6) **Correctly updating the tables according to the command (Chapter 4):** For each insert and delete commands, the specific tables need to be updated. For delete commands, MySQL deals with the foreign keys, as it is set to cascade. For order commands, the total for each client needs to be updated in the order table, as well as the product's quantity in the product table.
7) **Creation of the bills and reports (Chapter 4):** Reports need to be created on the order, product and client tables for report commands. Furthermore, for every order, a bill needs to be created specifying the order's details.

2. Problem analysis, modeling, scenarios, use cases



1. **Input command file:** The input data is read from a file given as the first parameter when running the jar file from the command prompt. This data consists of commands that need to be executed successively by the application. The commands are separated, each being on a separate line. There are 4 main commands that can be executed by the application, some can be executed on different tables of the database.
    1. **Insert:** Client or Product
    2. **Delete:** Client or Product
    3. **Order**
    4. **Report:** Client, Product or Order

    Any other command in the input file will be ignored by the application.

    All of the specific commands specified above, have their own format in which they have to be entered. The formats for each command are the following:

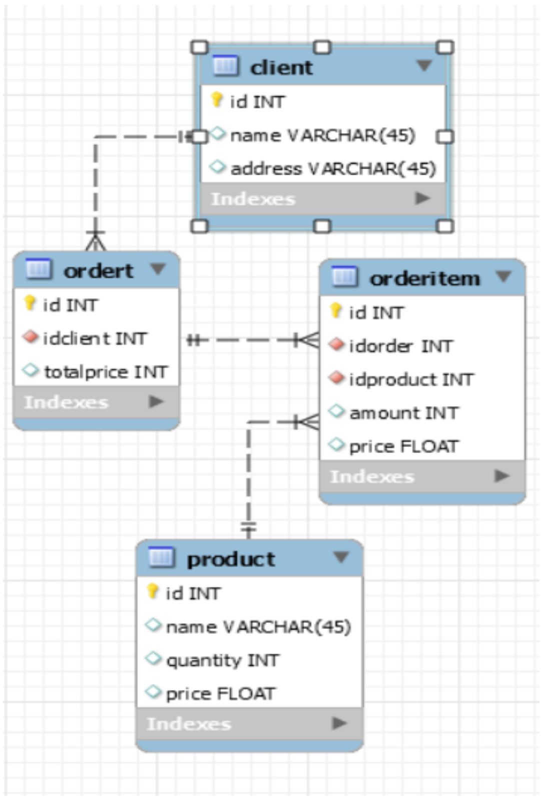
    - Inserting a client: "Insert client: Name, City"
    - Inserting a product: "Insert product: Name, Quantity, Unit price"
    - Deleting a client: "Delete client: Name, City"
    - Deleting a product: "Delete product: Name"
    - Ordering: "Order: Client name, product name, amount"
    - Reports: "Report client/order/product"

Note that the fields for client/product insertion, client deletion and ordering need to be separated by commas. Furthermore, the quantity of a product and the amount of an ordering need to be a positive integer and the unit price can be specified as a float or an integer.

2. **Output pdf files:** There are two types of output files, reports and bills, both being pdf files created by the application. The bills are the results of order commands. They contain the data given in the order command, and the total price of the order. If the desired order amount of a product is more than the quantity in the stock of the warehouse, the bill will say Insufficient stock.

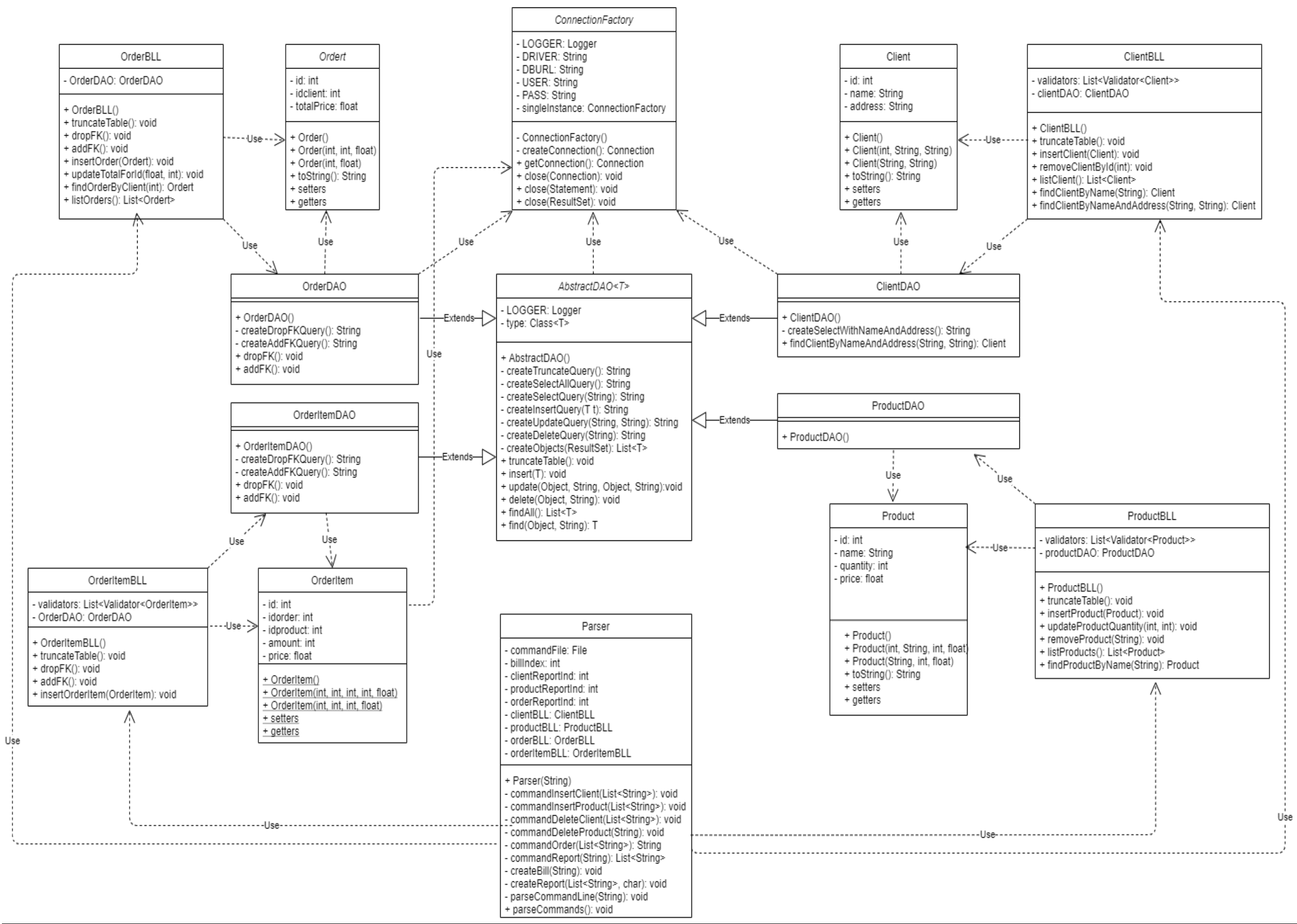
    The reports are created when the report command is inputted. Reports create a full overview of the table specified in the command, having every field and its value in separate paragraphs of the file. The possible tables for the report command are: order, client and product. In case of an empty table, the report will only say empty.

3. Design



The application uses a relational database, shopdb, having 4 tables: client, product, ordert and orderitem. The tables are connected between each other in the following way:

- Each field of the ordert table contains the id of an existing client in the client table and his or her total paid amount. This total is the client's total for all of his orders.
- The orderitem table contains an order of an item, the ordered amount and the price of the order. It is connected with the product table, having the ordered product's id and with the ordert table, having the id of the order that it is part of. It can be said that the total price for a client in the ordert table is the sum of the prices having the order's id in the orderitem table.
- If any of the parent id keys are deleted, their respective foreign keys are also deleted. Also, if a product is deleted, the orderitem fields having that product, are deleted, but the ordert's total price isn't updated. This is because I have considered that even if that product isn't sold anymore, the client still paid for it when it was in the warehouse.


ConnectionFactory
- LOGGER: Logger
- DRIVER: String
- DBURL: String
- USER: String
- PASS: String
- singleInstance: ConnectionFactory
- ConnectionFactory()
- createConnection(): Connection
+ getConnection(): Connection
+ close(Connection): void
+ close(Statement): void
+ close(ResultSet): void
OrderBLL
- OrderDAO: OrderDAO
+ OrderBLL()
+ truncateTable(): void
+ dropFK(): void
+ addFK(): void
+ insertOrder(Ordert): void
+ updateTotalForId(float, int): void
+ findOrderByClient(int): Ordert
+ listOrders(): List<Ordert>
Ordert
- id: int
- idclient: int
- totalPrice: float
+ Order()
+ Order(int, int, float)
+ Order(int, float)
+ toString(): String
+ setters
+ getters
Client
- id: int
- name: String
- address: String
+ Client()
+ Client(int, String, String)
+ Client(String, String)
+ toString(): String
+ setters
+ getters
ClientBLL
- validators: List<Validator<Client>>
- clientDAO: ClientDAO
+ ClientBLL()
+ truncateTable(): void
+ insertClient(Client): void
+ removeClientById(int): void
+ listClient(): List<Client>
+ findClientByName(String): Client
+ findClientByNameAndAddress(String, String): Client
OrderDAO
+ OrderDAO()
- createDropFKQuery(): String
- createAddFKQuery(): String
+ dropFK(): void
+ addFK(): void
AbstractDAO<T>
- LOGGER: Logger
- type: Class<T>
+ AbstractDAO()
- createTruncateQuery(): String
- createSelectAllQuery(): String
- createSelectQuery(String): String
- createInsertQuery(T t): String
- createUpdateQuery(String, String): String
- createDeleteQuery(String): String
- createObjects(ResultSet): List<T>
+ truncateTable(): void
+ insert(T): void
+ update(Object, String, Object, String):void
+ delete(Object, String): void
+ findAll(): List<T>
+ find(Object, String): T
ClientDAO
+ ClientDAO()
- createSelectWithNameAndAddress(): String
+ findClientByNameAndAddress(String, String): Client
ProductDAO
+ ProductDAO()
OrderItemDAO
+ OrderItemDAO()
- createDropFKQuery(): String
- createAddFKQuery(): String
+ dropFK(): void
+ addFK(): void
Product
- id: int
- name: String
- quantity: int
- price: float
+ Product()
+ Product(int, String, int, float)
+ Product(String, int, float)
+ toString(): String
+ setters
+ getters
ProductBLL
- validators: List<Validator<Product>>
- productDAO: ProductDAO
+ ProductBLL()
+ truncateTable(): void
+ insertProduct(Product): void
+ updateProductQuantity(int, int): void
+ removeProduct(String): void
+ listProducts(): List<Product>
+ findProductByName(String): Product
OrderItemBLL
- validators: List<Validator<OrderItem>>
- OrderDAO: OrderDAO
+ OrderItemBLL()
+ truncateTable(): void
+ dropFK(): void
+ addFK(): void
+ insertOrderItem(OrderItem): void
OrderItem
- id: int
- idorder: int
- idproduct: int
- amount: int
- price: float
+ OrderItem()
+ OrderItem(int, int, int, int, float)
+ OrderItem(int, int, int, float)
+ setters
+ getters
Parser
- commandFile: File
- billIndex: int
- clientReportInd: int
- productReportInd: int
- orderReportInd: int
- clientBLL: ClientBLL
- productBLL: ProductBLL
- orderBLL: OrderBLL
- orderItemBLL: OrderItemBLL
+ Parser(String)
- commandInsertClient(List<String>): void
- commandInsertProduct(List<String>): void
- commandDeleteClient(List<String>): void
- commandDeleteProduct(String): void
- commandOrder(List<String>): String
- commandReport(String): List<String>
- createBill(String): void
- createReport(List<String>, char): void
- parseCommandLine(String): void
+ parseCommands(): void
Use
Extends

The application uses reflection techniques with the AbstractDAO class, which helps with eliminating duplicate code. The ConnectionFactory class is used to connect to the database and execute the CRUD methods for each class type.

Packages: The application is divided in 7 packages:

- Start package: Contains a start class that has the main method.
- Presentation package: Contains the Parser class and the IllegalCommandException class, that extends RunTimeException.
- Model package: Contains the Client, OrderItem, Ordert and Product classes.
- DatabaseAccessLayer package: Contains the AbstractDAO, ClientDAO, OrderDAO, OrderItemDAO and ProductDAO classes.
- Connection package: Contains the ConnectionFactory class.
- BusinessLogicLayer package: Contains the ClientBLL, OrderBLL, OrderItemBLL and ProductBLL classes and the validators package
- Validators package: Contains the Validator interface and the classes ClientAddressValidator, ClientNameValidator, OrderAmountValidator, ProductNameValidator and ProductQuantityValidator that implement it.

4. Implementation

1) **AbstractDAO**: Represents the abstract database access object. All the other DAOs extend it.
   a) AbstractDAO(): Constructor method, that initializes the class type of the AbstractDAO.
   b) truncateTable(): Truncates the table specific to the class type.
   c) insert(T): Inserts an object of type T into the database.
   d) update(Object, String, object, String): Updates the value of the fieldToUpdate from the database table, having the fieldToFind value equal to findValue.
   e) delete(Object, String): Deletes an element from the database, having the given field equal to the given value.
   f) findAll(): Returns the selection of all the elements in the given database table.
   g) find(Object, String): Returns the element in the database having the given field equal to the given value.
2) **ConnectionFactory**: Creates the connection between the application and the database.
   a) getConnection(): Creates and gets the connection in the singleton object.
   b) close(Connection): Closes the Connection object given as parameter.
   c) close(Statement): Closes the Statement object given as parameter.
   d) close(ResultSet): Closes the ResultSet object given as parameter.
3) **ClientBLL:** Manages the client table's business logic.
   a) ClientBLL(): Constructor method. It initializes the clientDAO and the validator list by adding all the validators for the Client objects.
   b) truncateTable(): Truncates the client table from the database.
   c) insertClient(Client): Validates and inserts the client given as parameter into the database.
   d) removeClientById(): Removes the client having the id given as parameter from the database..
   e) listClients(): Lists all the clients in the client database table.
   f) findClientByName(String): Finds the client with the name given as parameter. Throws a NoSuchElementException if none found.
   g) findClientByNameAndAddress(String, String): Finds the client with the name and address given as parameters. Throws a NoSuchElementException if none found.
4) **OrderBLL:** Manages the ordert table's business logic.
   a) OrderBLL(): Constructor method. It initializes the orderDAO.
   b) truncateTable(): Truncates the ordert table from the database.
   c) dropFK(): Drops the idclient foreign key from the ordert table. The foreign key references the id from the client table.
   d) adFK(): Adds the foreign key idclient referencing the id from the client table.
   e) insertOrder(Ordert): Inserts the ordert given as parameter into the database.
   f) updateTotalForId(float, int): Updates the totalPrice to the newTotal for the order with orderId given as parameter.
   g) findOrderByClient(int): Finds the order for the client with id given as parameter.
   h) listOrder(): Returns a list with all the orders in the order database table.

5) **OrderItemBLL:** Manages the orderitem table's business logic.
   a) OrderItemBLL(): Constructor method. It initializes the orderItemDAO and the validator list by adding all the validators for the OrderItem objects.
   b) dropFK(): Drops the foreign keys idorder and idproduct from the orderitem database. The two foreign keys reference the id from the ordert table and the id from the product table.
   c) addFK(): Adds the idorder and idproduct foreign keys referencing the ids from the ordert and product tables.
   d) truncateTable(): Truncates the orderItem table from the database.
   e) insertOrderItem(OrderItem): Inserts the OrderItem given as parameter into the orderitem table.
6) **ProductBLL:** Manages the product table's business logic.
   a) ProductBLL(): Constructor method. It initializes the productDAO and the validator list by adding all the validators for the Product objects.
   b) truncateTable(): Truncates the product table from the database.
   c) insertProduct(Product): Validates and inserts the product given as parameter into the database.
   d) updateProductQuantity(int, int): Updates the parameter product's quantity to the given value.
   e) removeProduct(String): Removes the product having the name given as parameter from the database.
   f) listProducts(): Returns a list with all the products in the product database table.
   g) findProductByName(String): Finds the product with the name given as parameter.
7) **Validator<T>:** Validator interface. It is implemented by class for the purpose of validating some of the fields of the model classes before they are inserted.
   a) validate(T): Method to be implemented by a class that implements the Validator interface. It is used to validate a specific field of the object given as parameter.

5. Results

The application yields the results of the executed commands from the input command file. The application will create reports if the command exists and bills for the order commands. These output files are of pdf format, created with the help of iText. They will be created with appropriate names such as report_client2 or bill3, showing the kind of order they were created from and the index of such a command.

The result of thee executed commands can be seen in MySQL Workbench, as well. The tables will be recreated each time the application is ran and the results can be seen in the database's tables. The orderitem table doesn't have its specific report command, but the result of running the application can be seen in MySQL.

6. Conclusions

This was an interesting and challenging project that helped me learn a lot of features that are used in real world applications. This project was useful because it made me learn about such features and use them for the first time.

The layered architecture is used in many real world applications and it provides a simple and organized way of writing your projects. Most of the applications used by companies, work with databases and well-structured package management such as layered architecture can come in handy.

For this solution, I had to implement 3 new maven dependencies, the Guava Splitter, iText and the MySQL J Connector. Because of this, I realized the usefulness of Maven projects and how much easier it is to implement such libraries in your projects.

As I said above, working with databases is almost inevitable in the real world. The knowledge of how to model and construct databases in an efficient way is a most as a programmer. This is why I consider this assignment to be very useful.

For future development, I have 3 ideas:

1) Implementation of a well-designed GUI that shows the field for each database and commands could be inserted there.
2) Better checking of illegal commands.
3) Providing more flexibility for commands, such as lower and upper cases of letters.

7. Bibliography

- http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_3/Assignment_3_Indications.pdf