**ORDER MANAGEMENT**

1. Objectives
   1. Main Objective

Consider an application Order Management for processing customer orders for a warehouse. Relational databases are used to store the products, the clients and the orders. Furthermore, the application should be structured in packages in packages using a layered architecture presented in the support material.

The application should allow processing commands from a text file given as argument, perform the requested operations, save the data in the database, and generate reports in pdf format.

* 1. Secondary Objectives
* Define the classes used to map the tables from the database. (Chapter 2, 3)
* Implement the class in which the connection with the database is realized. (Chapter 3)
* Implement the classes for the data access operations. (Chapter 3)
* Implement the classes that define the business logic. (Chapter 3)
* Define the validator for the table’s fields. (Chapter 3)
* Define a class that interprets the given commands. (Chapter 3)
* Design a file parser. (Chapter 3)
* Implement a class for generating the pdf files. (Chapter 3)

1. Task analysis, Modeling, Scenarios, Uses Cases

The application should receive as argument the path to a file that contains a set of commands. The commands should be one of the following: Insert, Update, Delete, Report and Order having the syntax as shown below.

Insert 🡪 Insert <table name>: <first attribute>, <second attribute>, …

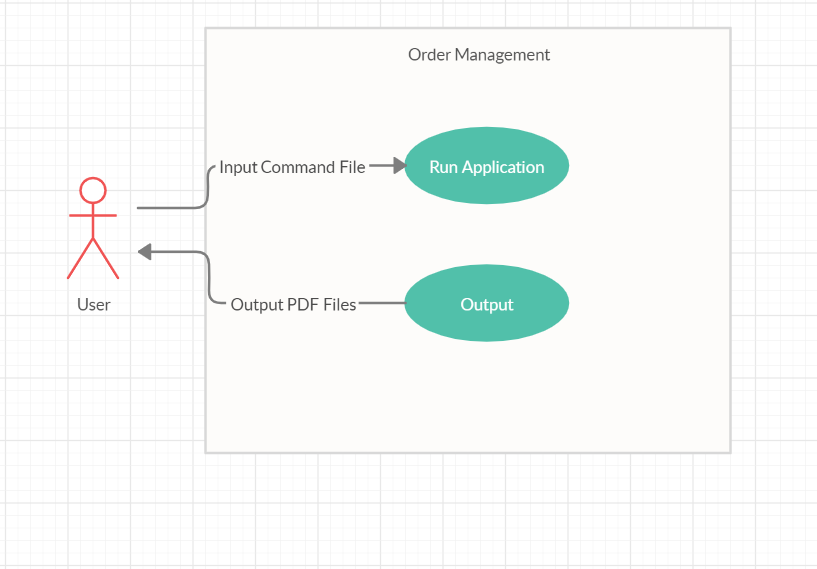
Delete 🡪 Delete <table name>: <identifier attribute>

Update 🡪 Update <table name>: <current attributes>, <updated attributes>

Report 🡪 Report <table name>

Order 🡪 Order: <Client identifier>, <Product identifier>, <amount>

The input file simulates the interaction with the user given the current state of the application not providing a graphical user interface. The commands from the file are considered line by line and interpreted by the application. The way the application communicates with the user is through the generated pdf via the explicit command Report or the automatic pdf generated as bills or messages. Also, the application displays in the console the commands as they are processed giving the user an active feedback of its running state.



* Use Case: Order Management
* Primary Actor: User
* Main Success Scenario:
  + The user provides the application the input file with the correct specified commands.
  + The application processes the commands one by one displaying on the screen the current one.
  + The modifications are made on the database.
  + In case of commands that demand reports, the generated pdfs are stored in folders with intuitive names.
  + The Application reaches the end of the command file and finishes.
* Alternative Sequences:
  + The user does not provide an input file
    - The application will print in the console the usage information.
  + The user provides a bad input file
    - The application will not be able to perform any operation.
  + The user provides a good input file but error generating commands
    - The errors encountered by the application are displayed by the logger in the console.

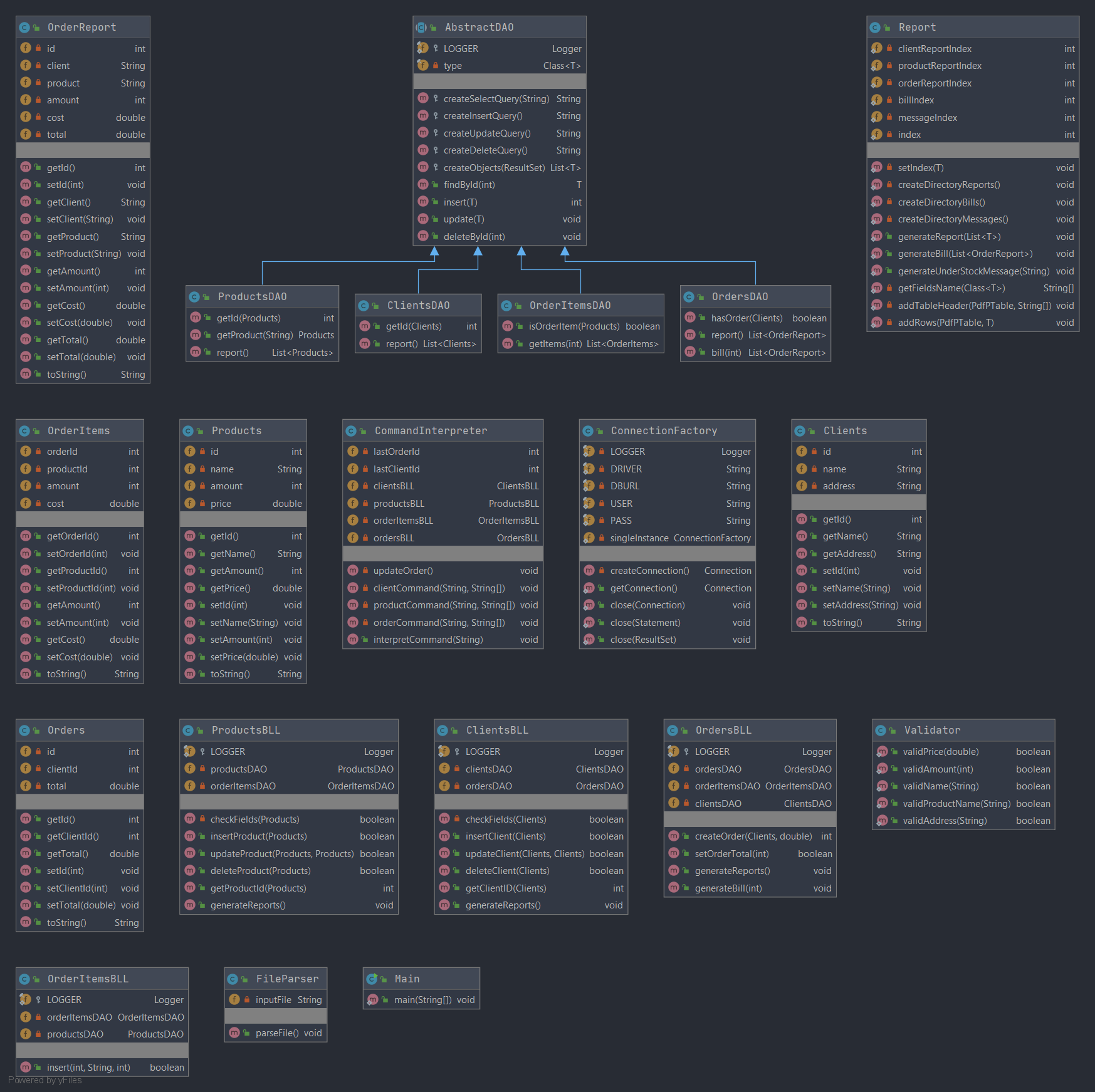
1. Design

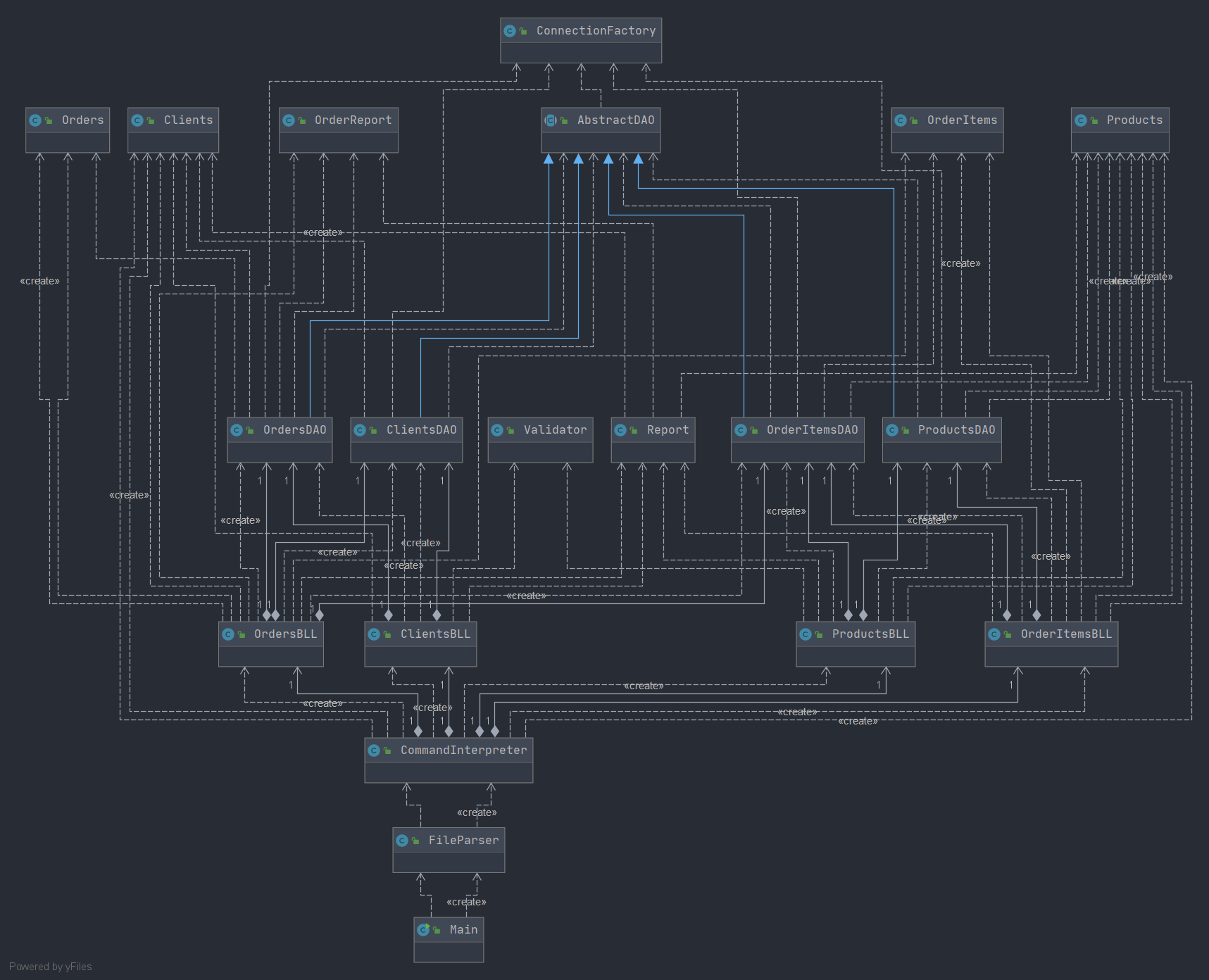
We start off the implementation by creating the class that realizes the connection with the database. This is the ConnectionFactory class that provides a getter for the created connection as well as methods for closing the connection, statement and resultSet. Then we move on to the classes that defines the database in the application. These are the classes that build the model and each should map to one of the tables from the database. We identify all the tables from the database and create the classes: Clients, Products, Orders and OrderItems each having as fields the attributes from the table they map on. The classes provide setters and getters for all their fields and various constructors. One last particular class is the OrderReport class which maps on to a virtual table obtain through a query which is used for generating the data for the orders report. These 5 classes constitute the model of the application and all the other classes rely on them.

The next step is to implement the classes that take care of the data access operations. Visualizing the most often required operations, it can easily be seen that they follow a very similar pattern. This being the case, the basic operations such as find by id, insert, update and delete could be more efficiently implemented in a generic way. We define an abstract class AbstractDAO which provides a generic implementation for the above mentioned operations and represents the base for all the specific DAO classes. Intuitively, we create a class for each table that will implement the required operation to be performed on that table. All the classes: ClientsDAO, OrderItemsDAO, ProductsDAO and OrdersDAO extend the AbstractDAO class which provides them the basic operations and each of them defines the implementation for the specific operations needed to be performed on each table. The DAO classes along with the ConnectionFactory define the Data Access package.

Moving on, we observe the need o some classes that work with the DAO classes at a more abstract level. These classes should define the logic for handling the operations on the tables. Here is where the data that enters the database is validated and the data obtained from the database is processed. A new set of classes building the business logic layer is created: ClientsBLL, ProductsBLL, OrderItemsBLL and OrdersBLL each taking care of all the possible operations performed on the tables as well as managing the data flow. Nonetheless, a new class Validator is required that will provide methods for validating all the fields from the model classes, thus reducing the inconsistency that may appear in the data stored in the database. These classes all together implement the business logic of the application.

The last thing the application must be able to do is providing a way to communicate with the user. Usually this is done through the graphical user interface, but in this simplified implementation the application will receive the commands from an input file receive at runtime. The first thing to be done is a class that manages the parsing of the input file such that each command is identified and ready to be fetch to a command interpreter. The command interpreter class receives a string containing a command that needs to be broken down in order to be evaluated. The command interpreter, based on the requirements of the command, delegates the execution to one of the BLL classes where the command will be executed. The next thing to do is implement a class that will create pdf files containing the data asked by the user. The Report class is responsible with all the needed functionality of the application to generate those pdfs.

The whole design of the application will be better understood from the below diagrams that contain all the classes along with their dependencies.



1. Implementation

* Clients Class

It is the model class that maps the clients table from the database. The fields: id, name and address are the attributes of the table. The class provides all the setters and getters making it very easy to model an object of this class. It also defines a set of multiple constructors gaining flexibility at creation time of the objects.

* Products Class

It is the model class that maps the products table from the database. The fields: id, name, amount and price are the attributes of the table. The class provides all the setters and getters making it very easy to model an object of this class. It also defines a set of multiple constructors gaining flexibility at creation time of the objects.

* OrderItems Class

It is the model class that maps the orderItems table from the database. The fields: orderId, productid, amount and cost are the attributes of the table. The class provides all the setters and getters making it very easy to model an object of this class. It also defines a set of multiple constructors gaining flexibility at creation time of the objects.

* Orders Class

It is the model class that maps the orders table from the database. The fields: id, clientId and total are the attributes of the table. The class provides all the setters and getters making it very easy to model an object of this class. It also defines a set of multiple constructors gaining flexibility at creation time of the objects.

* OrdersReport Class

It is the model class that maps to virtual table form the database generated by the query: SELECT o.id, c.name, p.name, oi.amount, oi.cost, o.total FROM Orders o JOIN OrderItems oi ON (o.id = oi.orderId) JOIN Clients c ON (o.clientId = c.id) JOIN Products p ON (oi.productId = p.id). The fields: id, client, product, amount, cost, total are the attribtues of the table. The objects of this class are mainly used for the generation of the orders reports and the bills pdfs.

* ConnectionFactory Class

The main goal of this class is to create a connection with the database such that the further need operations required could be done. It is a singleton class meaning that only one object of this class time can be available at a time. The fields are related to the information needed such that the connection can be realized. Along with the method for obtaining the created connection, the methods for closing a connection, a statement or a result set are also provided.

* AbstractDAO Abstract Class

This is an abstract class meant to implement generic the basic operations performed on the database such as find by id, insert, update and delete by id. It provides method for all the above specified operations implemented as default. For each model class a new DAO class should be created which will extend the AbstractDAO class. The new DAO class should complete the required operation set for its represented table.

* ClientsDAO Class

This class extends the AbstractDAO class and implements the remaining necessary operations required for the clients table. The method getId returns the id of the client receive as parameter by executing the specific query. The method report returns a list with all the objects that constitute rows in the clients table.

* ProductsDAO Class

This class extends the AbstractDAO class and implements the remaining necessary operations required for the products table. The method getId returns the id of the product receive as parameter by executing the specific query. The method report returns a list with all the objects that constitute rows in the products table. The method getProduct executes a query searching a specific product by name and returns a Products object corresponding to the found product.

* OrderItemsDAO Class

This class extends the AbstractDAO class and implements the remaining necessary operations required for the orderItems table. The method isOrderItem is used for finding if a given product is or is not part of any order in the database. The method getItems returns a list with all the objects included in the specified order.

* OrdersDAO Class

This class extends the AbstractDAO class and implements the remaining necessary operations required for the orders table. The method hasOrder is used to check if the given client is connected to any of the orders registered in the database. The method report returns a list with all the objects that constitute rows in the OrdersReport table. The bill method is similar with report one but in this case only the rows referred to by the orderId given.

* ClientsBLL Class

This class works with the previously described ClientsDAO and OrdersDAO classes and provides a more abstract view of the operations. The role of this class is to control the data access and filter the processed data such that no inconsistencies appear in the database. It provides the methods needed for inserting, deleting and updating the clients table as well as methods for getting a client’s id and generating the report of the clients.

* ProductsBLL Class

This class works with the previously described ProductsDAO and OrdeItemsDAO classes and provides a more abstract view of the operations. The role of this class is to control the data access and filter the processed data such that no inconsistencies appear in the database. It provides the methods needed for inserting, deleting and updating the products table as well as methods for getting a product’s id and generating the report of the clients.

* OrderItemsBLL Class

This class works with the previously described OrderItemsDAO and ProductsDAO classes and provides a more abstract view of the operations. The role of this class is to control the data access and filter the processed data such that no inconsistencies appear in the database. It provides the method needed for inserting rows int the orderItems table.

* OrdersBLL Class

This class works with the previously described ClientsDAO , OrdersDAO and OrderItemsDAO classes and provides a more abstract view of the operations. The role of this class is to control the data access and filter the processed data such that no inconsistencies appear in the database. It provides the methods needed for inserting, deleting and updating the orders table as well as methods generating the report of the orders and the report for the bill.

* Validator Class

This class provides all the needed methods for validating the fields of the model classes. The methods valid price and amount check if the receive values are positive, which is how the values characterizing this fields should be. Another three methods focus on using regex for validating names of clients and products as well as addresses.

* CommandInterprete Class

This is the class that makes the transition from string interpreted commands to actual actions that are done through the previously described business logic classes BLL. It receives a string containing a command which is then divided in sections making it possible to be understand and performed accordingly.

* FileParser Class

The role of this class is pretty straight forward and that is to parse the input command file line by line and feed the read lines to the command interpreter in order for the application to perform them.

* Report Class

This class deals with the creation of the pdf files that contain the tables generated by queries performed on the tables in the report methods of the DAO classes. It creates a folder for each logical category of pdfs that it generates: Reports, Bills and Messages. The generation of the report pdf is done in a generic way making it very code efficient and flexible for later improvements.

1. Results

The testing of the application is done on different parts of it. First the functionality of the CRUD classes is testing by inserting, updating and deleting demo data on the database. Then the we check the behavior of the validators making sure they yield the wanted result. Last thing that need to be checked is the functionality of the file parser along with the command interpreter.

Some of the scenarios for which the application was tested are:

* Inserting data into each table.
* Updating data from the Clients and Products tables.
* Deleting data from the Clients and Products tables.
* Checking the working of the validators on sample data.
* Checking the working of the file pares along with the command interpreter on several sample input files.

1. Conclusions

The main objective of this project was to get used working with databases as well as creating generic classes using reflection techniques. Using reflection in the future projects will be a powerful skill which will improve the overall quality of the project.

In terms of future improvements, the first thing we can take a look at is the database which could be way more complex storing much more useful data regarding other aspects. Another thing that can be done is making the application more user friendly. This would be achieved by getting rid of the command input file and replace it with a proper graphical user interface. This would provide a more controlled way to introduce data into the database.

1. Bibliography

* <https://www.baeldung.com/javadoc>
* <https://www.baeldung.com/java-pdf-creation>
* <http://tutorials.jenkov.com/java-reflection/index.html>
* <https://dzone.com/articles/layers-standard-enterprise>
* Ppt presentation: Java Concurrency