

**TECHNICAL
UNIVERSITY**
OF CLUJ-NAPOCA
ROMANIA

Order management

Faculty of Automation and Computer Science
Specialization – Computer Science

Laboratory teacher: Marcel Antal
Student: Coroian Tudor Florentin

1. Objective of the project

The main objective of the current assignment is to design and implement an application aimed for processing customer orders for a warehouse. Relational databases are used to store products, the clients and the orders. Furthermore, the application should be structured in packages using a layered architecture. This means that the application should use, at the very least, the following types of classes:

- Model classes – the data models of the application
- Business Logic classes – implement the application logic
- Presentation classes – implement the user input/output
- Data access classes – implement the access to the database

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. Other classes and packages can be added to implement the full functionality of the application.

Bellow, a table with the commands that the application should be able to execute is presented:

Command name	Command syntax	Description
Add client to the database	Insert client: Ion Popescu, Bucuresti	Insert in the database a new client with name Ion Popescu and address Bucuresti
Delete Client from the database	Delete client: Ion Popescu	Delete from database the client with name Ion Popescu
Add product to the database	Insert product: apple, 20, 1	Add product apple with quantity 20 and price 1
Delete product from the database	Delete product: apple	Delete product apple from database
Create order for client	Order: Ion Popescu, apple, 5	Create order for Ion Popescu, with apple quantity 5. Also update the apple stock to 15. Generate a bill in pdf format with the order and total price of 5
Generate reports	Report client Report order Report product	Generate pdf reports with all clients/orders/products displayed in a tabular form. The reports should contain the information corresponding to the entity for which reports are asked (client, order or product) returned from the database by a SELECT * query, displayed in a table in a PDF file

```

Insert client: Mihai Dobre, Bistrita
Insert client: Mirel Constantin, Onesti
Insert client: Viorin Copoiu, Onesti
Insert client: Neculai Isac, Babenii
Report client
Delete client: Mircea Nicolae, Onesti
Delete client: Neculai Isac, Babenii
Report client
Insert product: plum, 40, 1.75
Insert product: quince, 50, 3.15
Insert product: kale, 25, 7.20
Insert product: plum, 20, 1.75
Insert product: cucumber, 20, 14
Report product
Delete product: apple
Delete product: cucumber
Report product
Order: Mihai Dobre, plum, 2
Order: Mirel Constantin, quince, 3
Order: Viorin Copoiu, kale, 4
Order: Mihai Dobre, quince, 4
Order: Mihai Dobre, cucumber, 2
Order: Mihai Dobre, kale, 5
Order: Mihai Dobre, plum, 4
Order: Mirel Constantin, kale, 40
Report order
Delete product: plum
Report order
Report product

```

Figure 1. Input file layout example

The output of the simulation should be composed of pdf files that result after executing the commands presented in the table. An example of such an output file is presented below, with the mention that the file was the result of executing the command “Report order”, after executing all the other command from the input file presented above.

ID	Client Name	Order Item		Total
1	Mihai Dobre	Product	Quantity	48.6
		quince	4	
		kale	5	
2	Mirel Constantin	Product	Quantity	9.45
		quince	3	
3	Viorin Copoiu	Product	Quantity	28.8
		kale	4	

Figure 2. Output file layout example

In order to build the application, second objectives have to be considered, such as:

1. designing two classes to read and write to the text files
2. deciding on a proper layout of the output files
3. deciding on a structure for the tables in which the data will be stored
4. deciding on a policy for deleting an entry of the table
5. deciding on means to let the user now which commands failed (either due to syntax errors or logical errors)
6. deciding whether or not the delete or update operations should be performed and then cascaded

2. Problem analysis, problem modeling, scenarios, use cases

Problem analysis

Before moving any further, due to the loose specifications that were given in the objective of the project, some decisions were left to the latitude of the designer. Therefore, those decisions are going to be presented in **Chapter 3, Design decisions**.

Also, as a disclaimer note, I decided to show images and examples for another set of input data than the one given in the Assignment specification, with the mention that the input data that were provided there were also processed and the result can be found in the folder “Output example”. The reason for providing myself with another set of input data was made so that I can design the application better, by giving as input as many corner cases as I could come up with, without exceeding the specification implied by the assignment. Again, I need to reinforce the fact that the input data given in the assignment’s request was solved successfully and that my set of input data should not, by any means, be a substitute to the mandatory set of input data (i.e. the one given in the assignment).

First of all, it is clear that proper structure for the tables should be designed, before moving forward with the rest of the implementation. Consequently, each type of data (i.e. client, product, order) will be stored in a table, in the MySQL database called “ordermng.db”. In addition, a fourth table will be designed, that will store the items that are purchased in each order. Such a table will be called “orderitem”.

In what follows, the columns of this tables will be presented and described. It is worth mentioning now that the following tables are the result of running the command presented as an example in **Chapter 1, Objective of the project**, namely, **Figure1**.

- client table

id_client	name	city	deleted
1	Mihai Dobre	Bistrita	0
2	Mirel Constantin	Onesti	0
3	Viorin Copoiu	Onesti	0
4	Neculai Isac	Babeni	1

Figure 3. client table from MySQL

From the table above, it can be shown that each entry in the “client” table is characterized by the following attributes:

- id_client : uniquely identifies a client
- name : the name of the client
- city : the city of the client
- deleted : flag that shows if the client should be treated as if it was deleted or not

▪ product table

id_product	prod_name	quantity	price	deleted
2	quince	43	3.15	0
3	kale	16	7.2	0
4	plum	54	1.75	1
5	cucumber	20	14	1

Figure 4. product table from MySQL

From the table above, it can be shown that each entry in the “product” table is characterized by the following attributes:

- id_product : uniquely identifies a product
- prod_name : the name of the product; uniquely identifies the product
- quantity : the quantity of that product that is available
- price : the price for one piece of the product
- deleted : flag that shows if the product should be treated as if it was deleted or not

▪ order table

id_order	id_client	total	deleted
1	1	48.6	0
2	2	9.45	0
3	3	28.8	0

Figure 5. order table from MySQL

From the table above, it can be shown that each entry in the “order” table is characterized by the following attributes:

- id_order : uniquely identifies an order
- id_client : the id of the client that made the order; is in a 1:1 relationship with the id_order
- total : the total price of the order
- deleted : flag that shows if the order should be treated as if it was deleted or not

- orderitem table

id_orderitem	id_order	prod_name	quantity	deleted
2	2	quince	3	0
3	3	kale	4	0
5	1	quince	4	0
9	1	kale	5	0
11	1	plum	6	1

Figure 6. orderitem table from MySQL

From the table above, it can be shown that each entry in the “orderitem” table is characterized by the following attributes:

- id_orderitem : uniquely identifies an order item
- id_order : the id of the order to which the entry belongs
- quantity : the quantity of the product purchased
- deleted : flag that shows if the order item should be treated as if it was deleted or not

The *Client*, *Product*, *Order* and *Order Item* tables will then generate the models for this application in a direct way, since for the application to communicate with the database, the models need to be a 1:1 replica of the tables. Nonetheless, these models will be presented and described in **Chapter 2, Problem modeling** and more thoroughly analyzed in **Chapter 3, Models**.

Problem modeling

In what follows, the *Client*, *Product*, *Order* and *OrderItem* models will be presented, but only briefly. Further details on each class will be presented in Chapter 3. It should be mentioned that from now on, even though the model is named *Orders.java*, we will refer to it as simply *Order.java*.

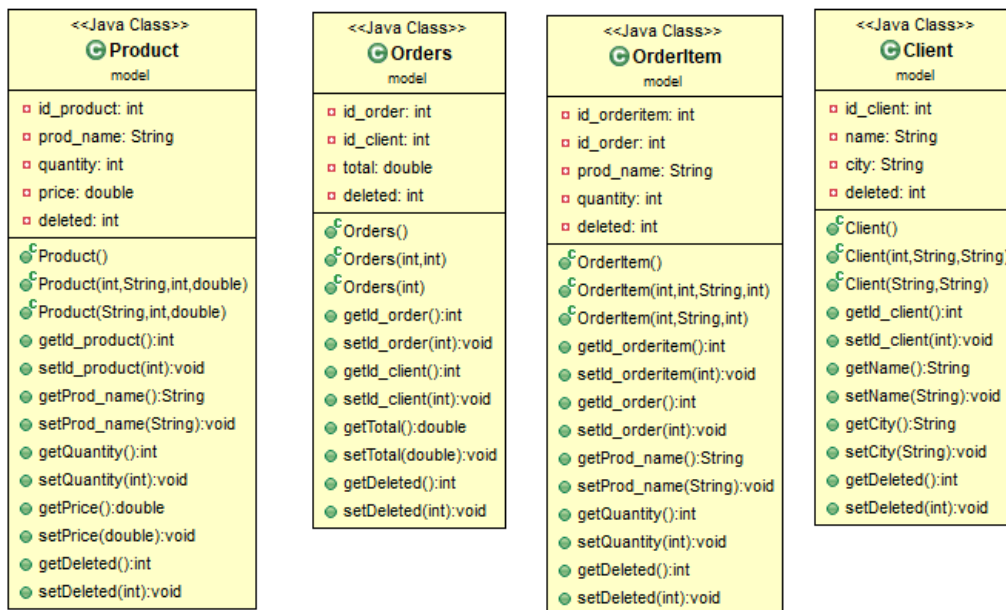


Figure 7. Models of the application

Use-case scenario

Step 1: Input the data into the input text file.

Step 2: Run the *PT2020_30423_Tudor_Coroian_Assignment_3.jar* in the command prompt as follows:

```
java -jar PT2020_30423_Tudor_Coroian_Assignment_3.jar input.txt
```

Step 3: Press enter.

Step 4: Wait for the simulation to end.

Step 5: The result will be shown in the output text files.

3. Application design

The application “Order manager” is designed using the layered architectural design, that splits the execution of the program into three layers: I/O operations with which the user interacts, business logic classes which implement the main logic of the application and data access classes which communicate with the database. The last ones are designed using reflection techniques that allow for the same code to be performed by multiple classes. Also worth mentioning is that the project is designed using an object-oriented language and the paradigms that come with this language.

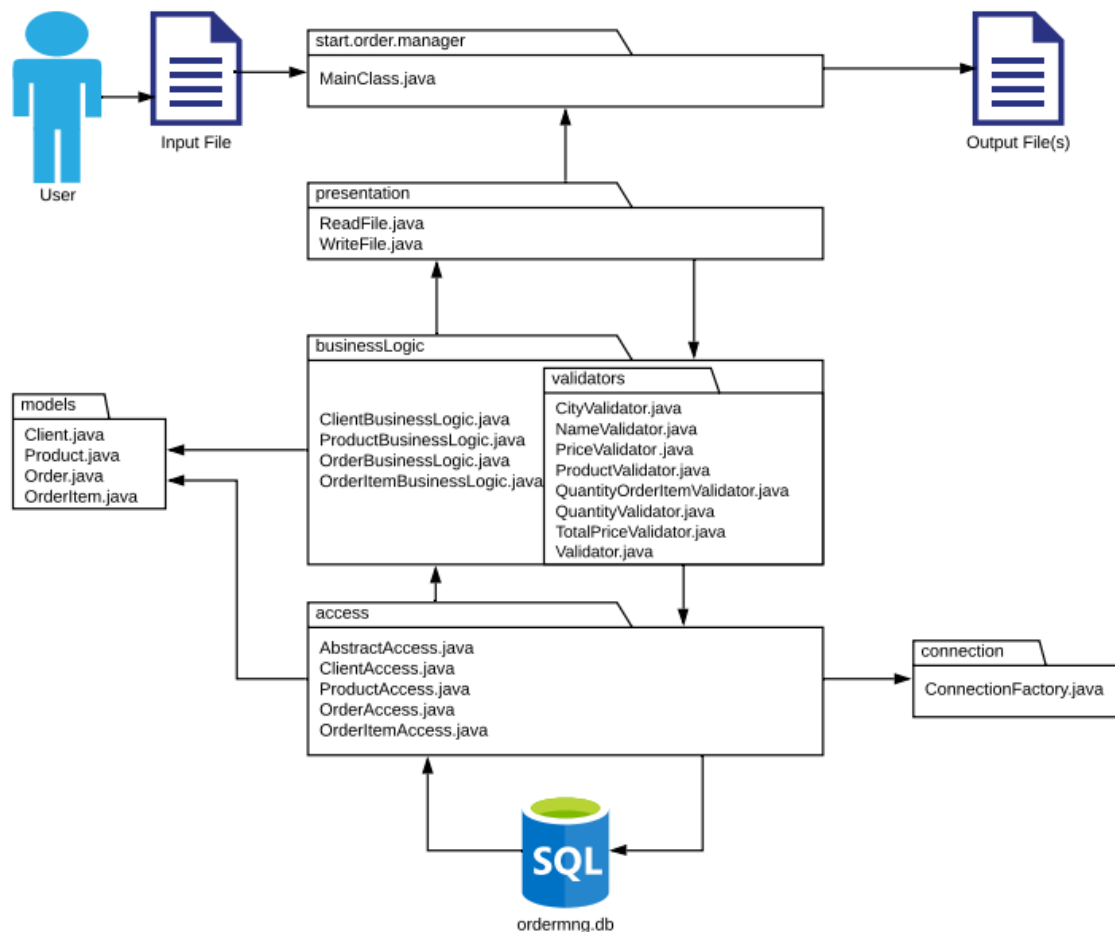


Figure 8. Package design

As it can be seen from the illustration above, there are seven packages:

- *connection* which has only the instance of the *ConnectionFactory.java* class, generated using the singleton paradigm
- *access* which has the classes that directly access the database
- *models* which contains the actual models of the design (i.e. *Client.java*, *Product.java*, *Order.java* and *OrderItem.java*)
- *businessLogic* which contains the classes that perform the actual functionality of the application
- *validators* which contains the classes that validated various attributes of the data that is going to be inserted in the database
- *presentation* which contains the classes that deal with parsing the input file and generating the output file(s)
- *start.order.manager* which holds a single class, called *MainClass.java* in which the star-up of the application is found

It is easy to see from the diagram the layered structure of the application. The user access the application via the input file which is then parsed by the *MainClass.java*. This class contains instances of both the *ReadFile.java* and *WriteFile.java* classes, through which the data is interpreted, sent to the business logic classes and the information is retrieved also from them and sent to the output file(s). The classes in the business logic package validate the input data before sending them further to the classes in the access package, using the classes in the validator package. The classes in the access package are the ones that prepare the SQL statements and queries that are going to be run in the data base. They are also the ones that retrieve the data from the database and send it up through the same layers.

Design decisions ...

Before moving any further with the discussion, some designing decisions have to be stated:

1. The deletion of the entries is not performed as an actual SQL deletion, but rather the deleted column is updated such that it will hold the value 1 indicating that from now on the entry should be treated as being deleted. A deleted entry does not influence the other entries, therefore an entry with the same unique identifier may be inserted in the database.
2. The CRUD (i.e. create, read, update, delete) operations have been implemented and are fully functional for all the models. This is in addition of the assignment instruction types, which only requested certain operations for each model. Therefore, whereas the assignment only requested the following operations:
 - insert product
 - insert client
 - insert order
 - delete product
 - delete client

- select product
- select client
- select order

all the CRUD operations were implemented for each model, including for *OrderItem.java* class such that we can perform the following operations:

<i>Client.java</i>	<i>Product.java</i>	<i>Order.java</i>	<i>OrderItem.java</i>
findClientById	findProductById	findOrderById	findOrderItemById
findClientByName	findAllProducts	findAllOrders	findAllOrderItems
findClientByCity	findProductByName	findOrdersByClientId	findAllOrderItemsFromOrder
findClient	insertProduct	insertOrder	insertOrderItem
findAllClients	updateProduct	updateOrder	updateOrderItem
insertClient	deleteProduct	deleteOrder	deleteOrderItem
updateClient	deleteProductById	deleteOrderById	deleteOrderItemById
deleteClient			
deleteClientById			

- The delete and update operations have a cascading effect such that:
 - if a client is deleted/updated, all the orders of that client are deleted/updated as well
 - if a product is updated (i.e. price is changed, name is changed) all the orders and order items that contain that product are updated as well
 - if a product is deleted, all the order items that contain that product are deleted
 - if an order is updated, (i.e. the id_order is changed), all the order items from that order are updated as well
 - if an order is deleted, all the order items from that order are deleted, but the stock (i.e. an entry in the product table) is not deleted
 - if an order item is updated, the order from which the order item is a part of is updated as well
 - if an order item is deleted, the order from which the order item is a part of is updated (i.e. the total price is decreased), or deleted, if the order item was the last one in the order
 - The client and order tables are in a 1:1 relationship, such that one client will have only one entry in the order table, that will be updated each time the client makes another purchase. This decision, even if strange at first, is a logical one, since the order do not have a date. In this way we may be interested only in how much each client has to pay, rather than how many times a client purchased a product from the stock.
- e.g.1 After running the following commands:
- Order: client1, product1, quantity1
- Order: client1, product2, quantity2

the order table will consist of only one entry and the orderitem table will consist of two entries with the two products.

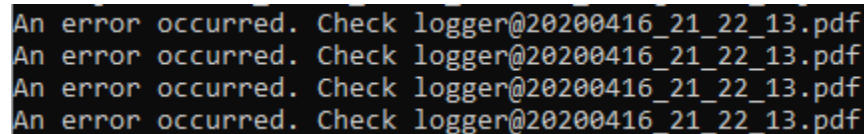
e.g.2 Similarly, after running the following commands:

Order: client1, product1, quantity1

Order: client1, product1, quantity2

the order table will consist of only one entry and the orderitem table will consist of only one entry too, where the quantity of the product1 will be $quantity1 + quantity2$.

5. Two successful insertions of the same product will result in the second one being treated as an update to the first one. Keep in mind that the products are differentiated through id_product and prod_name, therefore, the database will not allow for two entries with the same prod_name. The second update is done regardless if the price of the product is the same as the one found in the database table or not (i.e. the entry will keep the price of the second insert command)
6. In order to allow the user to know what was the outcome of the commands that were run by the application, the decision to create a logger.pdf file was made. The user is notified in the console if an error occurred while performing the commands and it will indicate the name of the logger where further details will be specified.



```
An error occurred. Check logger@20200416_21_22_13.pdf
An error occurred. Check logger@20200416_21_22_13.pdf
An error occurred. Check logger@20200416_21_22_13.pdf
An error occurred. Check logger@20200416_21_22_13.pdf
```

Figure 9. Command line notification of the errors

Above is an example of such a case and below is the logger file generated by that run of the application.

As it can be seen in the picture, the four error signaled in the command line correspond to the four errors written in red in the logger file. It is also worth mentioning now that the logger.pdf file distinguishes between 4 types of lines: (The errors required to be shown by the assignment are the ones colored in red)

- **error** line which is colored in red
 - **update** line which is colored in blue (such an update is signaled if the instruction generating it was intended to be an insert operation)
 - **warning** line which is colored yellow (a warning is signaled if there is a database integrity violation)
 - **normal execution** line which is colored in black
7. All the files generated by the application (except for the logger) have a string of 5 random characters at the beginning of the title to differentiate between them. To be noted that all the files generated by one run of the application will have the same 5 random letters in front of them and will be furthermore differentiated by means that are going to be described in **Chapter 4, Implementation**

21:22:14: Client Mihai Dobre was inserted successfully.
21:22:15: Client Mirel Constantin was inserted successfully.
21:22:15: Client Viorin Copoiu was inserted successfully.
21:22:15: Client Neculai Isac was inserted successfully.
21:22:15: Report for clients was generated successfully.
21:22:15: Client Mircea Nicolae was not found in the database. Operation aborted.
21:22:15: Client Neculai Isac was deleted successfully.
21:22:15: Report for clients was generated successfully.
21:22:15: Product plum was inserted successfully
21:22:15: Product quince was inserted successfully
21:22:15: Product kale was inserted successfully
21:22:15: Product plum was already inserted in the database. It was updated instead.
21:22:15: Product cucumber was inserted successfully
21:22:15: Report for products was generated successfully.
21:22:15: Product apple was not found int the database.
21:22:15: Product cucumber was deleted successfully.
21:22:15: Report for products was generated successfully.
21:22:15: Order of Mihai Dobre was created successfully.
21:22:15: Order item plum was inserted in the database.
21:22:15: Order with id= 1 was updated successfully.
21:22:16: Order of Mirel Constantin was created successfully.
21:22:16: Order item quince was inserted in the database.
21:22:16: Order with id= 2 was updated successfully.
21:22:16: Order of Viorin Copoiu was created successfully.
21:22:16: Order item kale was inserted in the database.
21:22:16: Order with id= 3 was updated successfully.
21:22:16: Order item quince was inserted in the database.
21:22:16: Order with id= 1 was updates successfully.
21:22:16: Product cucumber was not inserted in the database. Operation aborted.
21:22:17: Order item kale was inserted in the database.
21:22:17: Order with id= 1 was updates successfully.
21:22:17: Product plum from the order 1 was already inserted in the database. It was updated instead.
21:22:17: Not enough kale in stock. Operation aborted.
21:22:17: Report for orders was generated successfully.
21:22:17: Product plum was deleted successfully.
21:22:17: Report for orders was generated successfully.
21:22:17: Report for products was generated successfully.

Figure 10. The logger.pdf file resulted after running the commands from Figure1

Model

As presented before, the models of the four classes, *Client.java*, *Product.java*, *Order.java* and *OrderItem.java*, are based on the tables from the database.

The UML diagrams for *Client.java*, *Product.java*, *Order.java* and *OrderItem.java* were presented before. In what follows, the methods of each class will be described.

Client.java

- **GETID_CLIENT**

```
public int getId_client()
```

Returns:

the id of this client

- **SETID_CLIENT**

```
public void setId_client(int id)
```

Sets the id of this client.

Parameters:

id - : int

- **GETNAME**

```
public java.lang.String getName()
```

Returns:

the name of this client

- **SETNAME**

```
public void setName(java.lang.String name)
```

Sets the name of this client.

Parameters:

name - : String

- **GETCITY**

```
public java.lang.String getCity()
```

Returns:

the city from which this client is

- **SETCITY**

```
public void setCity(java.lang.String city)
```

Sets the city from which this client is.

Parameters:

city - : String

- GETDELETED

```
public int getDeleted()
```

Returns:

1 if this client is deleted and 0 otherwise

- SETDELETED

```
public void setDeleted(int deleted)
```

Sets the deleted flag of this class.

Parameters:

deleted - : int

Constructors of this class:

Client()

Constructor for the Client.java class without any arguments.

Client(int id, java.lang.String name, java.lang.String city)

Constructor for the Client.java class when the id_client is known.

Client(java.lang.String name, java.lang.String city)

Constructor for the Client.java class when the id_client is unknown

Fields of this class:

- ID_CLIENT

```
private int id_client
```

The id of the client stored in the database. Must be unique.

- NAME

```
private java.lang.String name
```

The name of the client stored in the database. Multiple entries can have similar names if the "city" column is different.

- CITY

```
private java.lang.String city
```

The city of the client stored in the database. Multiple entries can have similar city names if the "name" column is different.

- DELETED

```
private int deleted
```

Flag to show if the client should be treated as deleted.

Product.java

- GETID_PRODUCT

```
public int getId_product()
```

Returns:

```
the id of this product
```

- SETID_PRODUCT

```
public void setId_product(int id_product)
```

Sets the id of this product.

Parameters:

```
id_product - : int
```

- GETPROD_NAME

```
public java.lang.String getProd_name()
```

Returns:

```
the name of this product
```

- SETPROD_NAME

```
public void setProd_name(java.lang.String prod_name)
```

Sets the name of this product.

Parameters:

```
prod_name - : String
```

- GETQUANTITY

```
public int getQuantity()
```

Returns:

```
the quantity of this product
```

- SETQUANTITY

```
public void setQuantity(int quantity)
```

Sets the quantity of this product.

Parameters:

```
quantity - : int
```

- GETPRICE

```
public double getPrice()
```

Returns:

the price of this product

- SETPRICE

```
public void setPrice(double price)
```

Sets the price of this product.

Parameters:

price - : double

- GETDELETED

```
public int getDeleted()
```

Returns:

1 if this product is deleted and 0 otherwise

- SETDELETED

```
public void setDeleted(int deleted)
```

Sets the deleted flag of this class.

Parameters:

deleted - : int

Constructors of this class:

Product()

Constructor for the Product.java class without any arguments.

Product(int id, java.lang.String name, int quantity, double price)

Constructor for the Product.java class when the id_product is known.

Product(java.lang.String name, int quantity, double price)

Constructor for the Product.java class when the id_product is unknown.

Fields of this class:

- ID_PRODUCT

```
private int id_product
```

The id of the product stored in the database. Must be unique.

- **PROD_NAME**

```
private java.lang.String prod_name
```

The name of the product stored in the database. Must be unique.

- **QUANTITY**

```
private int quantity
```

The quantity of the product stored in the database. Must be greater or equal to zero.

- **PRICE**

```
private double price
```

The price of the product stored in the database. Must be greater than zero.

- **DELETED**

```
private int deleted
```

Flag to show if the product should be treated as deleted.

Orders.java

- **GETID_ORDER**

```
public int getId_order()
```

Returns:

```
the id of the order
```

- **SETID_ORDER**

```
public void setId_order(int id_order)
```

Sets the id of this order.

Parameters:

```
id_order - : int
```

- **GETID_CLIENT**

```
public int getId_client()
```

Returns:

```
the id of the client that made the order
```

- **SETID_CLIENT**

```
public void setId_client(int id_client)
```

Sets the id of the client that made the order.

Parameters:

```
id_client - : int
```


- GETTOTAL

```
public double getTotal()
```

Returns:

the total price to be payed

- SETTOTAL

```
public void setTotal(double total)
```

Sets the total price of this order.

Parameters:

total - : double

- GETDELETED

```
public int getDeleted()
```

Returns:

1 if this order is deleted and 0 otherwise

- SETDELETED

```
public void setDeleted(int deleted)
```

Sets the deleted flag of this class.

Parameters:

deleted - : int

Constructors of this class:

Orders ()

Constructor for the Order.java class without any arguments.

Orders (int client)

Constructor for the Order.java class when the id_order is unknown.

Orders (int id, int client)

Constructor for the Order.java class when the id_order is known.

Fields of this class:

- ID_ORDER

```
private int id_order
```

The id of the order stored in the database. Must be unique.

- ID_CLIENT

```
private int id_client
```

The id of the client that made the order.

- TOTAL

```
private double total
```

The total price of the order.

- DELETED

```
private int deleted
```

Flag to show if the order should be treated as deleted.

OrderItem.java

- GETID_ORDERITEM

```
public int getId_orderitem()
```

Returns:

```
the id of this product (order item)
```

- SETID_ORDERITEM

```
public void setId_orderitem(int id_orderitem)
```

Sets the id of this product (order item).

Parameters:

```
id_orderitem - : int
```

- GETID_ORDER

```
public int getId_order()
```

Returns:

```
the id of the order to which this product (order item) belongs
```

- SETID_ORDER

```
public void setId_order(int id_order)
```

Sets the id of the order to which this product (order item) belongs.

Parameters:

```
id_order - : int
```

- GETPROD_NAME

```
public java.lang.String getProd_name()
```

Returns:

```
the name of this product (order item).
```

- SETPROD_NAME

```
public void setProd_name(java.lang.String prod_name)
```

Sets the name of this product (order item).

Parameters:

```
prod_name - : String
```

- GETQUANTITY

```
public int getQuantity()
```

Returns:

```
the quantity of this order item
```

- SETQUANTITY

```
public void setQuantity(int quantity)
```

Sets the quantity of this order item.

Parameters:

```
quantity - : int
```

- GETDELETED

```
public int getDeleted()
```

Returns:

```
1 if this order item is deleted and 0 otherwise
```

- SETDELETED

```
public void setDeleted(int deleted)
```

Sets the deleted flag of this class.

Parameters:

```
deleted - : int
```

Constructors of this class:

[OrderItem](#)()

Constructor for the OrderItem.java class without any arguments.

[OrderItem](#)(int id, int idOrder, java.lang.String name, int quantity)

Constructor for the Order.java class when the id_orderitem is known.

[OrderItem](#)(int idOrder, java.lang.String name, int quantity)

Constructor for the Order.java class when the id_orderitem is unknown.

Fields of this class:

- **ID_ORDERITEM**

```
private int id_orderitem
```

The id of the order item stored in the database. Must be unique.

- **ID_ORDER**

```
private int id_order
```

The id of the order to which this order item belongs. Must correspond to an existing order id.

- **PROD_NAME**

```
private java.lang.String prod_name
```

The name of the product (order item). Multiple entries can have similar id_order if the product names are different.

- **QUANTITY**

```
private int quantity
```

The quantity of the product. Must be greater than zero. Multiple entries can have similar product names if the id_order is different.

- **DELETED**

```
private int deleted
```

Flag to show if the order item should be treated as deleted.

AccessLayer

..

When it comes to accessing the database, the decision to use reflection techniques was made. This led to the designing of the *AbstractAccess.java* class which implements methods used by all the access classes, which in turn may have their own methods to access the database specified.

Bellow, it can be seen how each of the access class extends the *AbstractAccess.java* class.

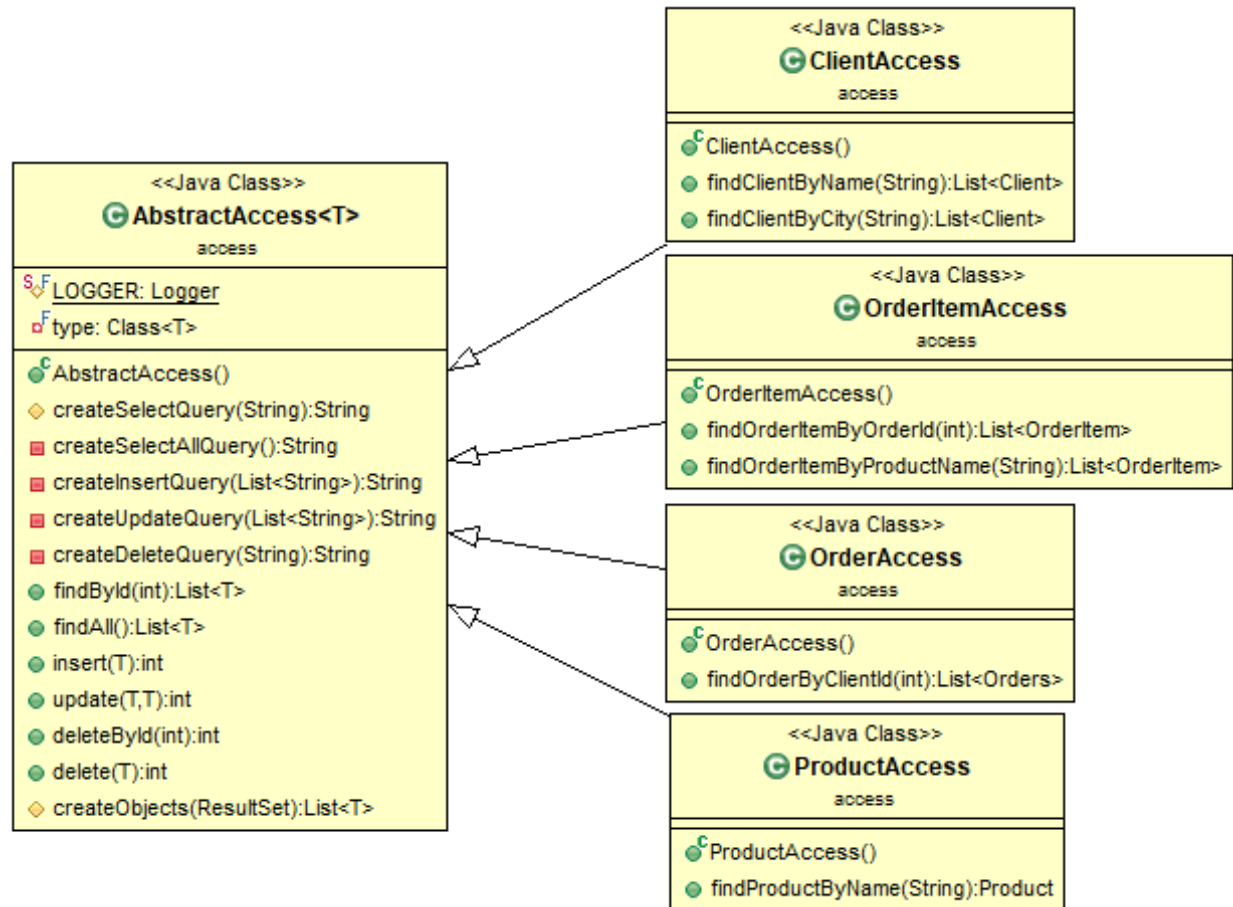


Figure 11. Access Layer

For the sake of simplicity, only the CRUD methods from the *AbstractAccess.java* class will be presented next, the ones from the other classes being just extensions of them.

• CREATESELECTQUERY

```
protected java.lang.String createSelectQuery(java.lang.String field)
```

This method builds a SELECT query with a single condition, of the form: SELECT * FROM T WHERE field=?

Parameters:

field - : String

Returns:

a string with the query

- **CREATESELECTALLQUERY**

```
private java.lang.String createSelectAllQuery()
```

This method builds a SELECT query without any conditions, of the form: SELECT * FROM T

Returns:

a string with the query

- **CREATEINSERTQUERY**

```
private java.lang.String createInsertQuery(java.util.List<java.lang.String>  
> fields)
```

This method builds an INSERT query of the form: INSERT INTO T (column1, column2, ...) VALUES (value1, value2, ...)

Parameters:

fields - : List of String

Returns:

a string with the query

- **CREATEUPDATEQUERY**

```
private java.lang.String createUpdateQuery(java.util.List<java.lang.String>  
> fields)
```

This method builds an UPDATE query of the form: UPDATE T SET (column1=value1, column2=value2, ...) WHERE id = ?

Parameters:

fields - : List of String

Returns:

a string with the query

- **CREATEDELETEQUERY**

```
private java.lang.String createDeleteQuery(java.lang.String field)
```

This method builds and UPDATE query of the form: UPDATE T SET deleted = true WHERE id = ?

Parameters:

field - : String

Returns:

a string with the query

- FINDBYID

```
public java.util.List<T> findById(int id)
```

This method returns a list with all the entries that have as id the id specified as the parameter. In case it finds only one element, it will return a list with that element.

Parameters:

id - : int

Returns:

a list with all the entries found or null in case of errors

- FINDALL

```
public java.util.List<T> findAll()
```

This method returns a list with all the entries in the table specified as a generic parameter.

Returns:

a list with all the entries in the table or null in case of errors

- INSERT

```
public int insert(T object)
```

This method inserts a new entry in the database, with the values specified in the object argument.

Parameters:

object - : T

Returns:

0 if the object was inserted successfully and -1 otherwise

- UPDATE

- ```
public int update(T objectOld,
 T objectNew)
```

This method updates the entry specified as the objectOld argument with the values specified in the objectNew argument.

**Parameters:**

objectOld - : T

objectNew - : T

**Returns:**

0 if the object was updated successfully and -1 otherwise

- DELETEBYID

```
public int deleteById(int id)
```

This method updates the deleted column of the table for the entry with the same id as the one specified as the argument.

**Parameters:**

id - : int

**Returns:**

0 if the object was deleted successfully and -1 otherwise

- DELETE

```
public int delete(T object)
```

This method updates the deleted column of the table for the entry specified as the object parameter.

**Parameters:**

object - : T

**Returns:**

0 if the object was deleted successfully and -1 otherwise

**See Also:**

`deleteById(int id)`

- CREATEOBJECTS

```
protected java.util.List<T> createObjects(java.sql.ResultSet resultSet)
```

This method creates a list from the result set of a query.

**Parameters:**

resultSet - : ResultSet

**Returns:**

a list with all the entries in the result set or null in case of errors



## BusinessLogic layer

...

The actual logic of the application is implemented in the business logic layer. The classes here each has an instance of the classes from the access layer in order to have access to the operations that can be performed on the database.

Bellow, the UML diagrams of these classes are presented.

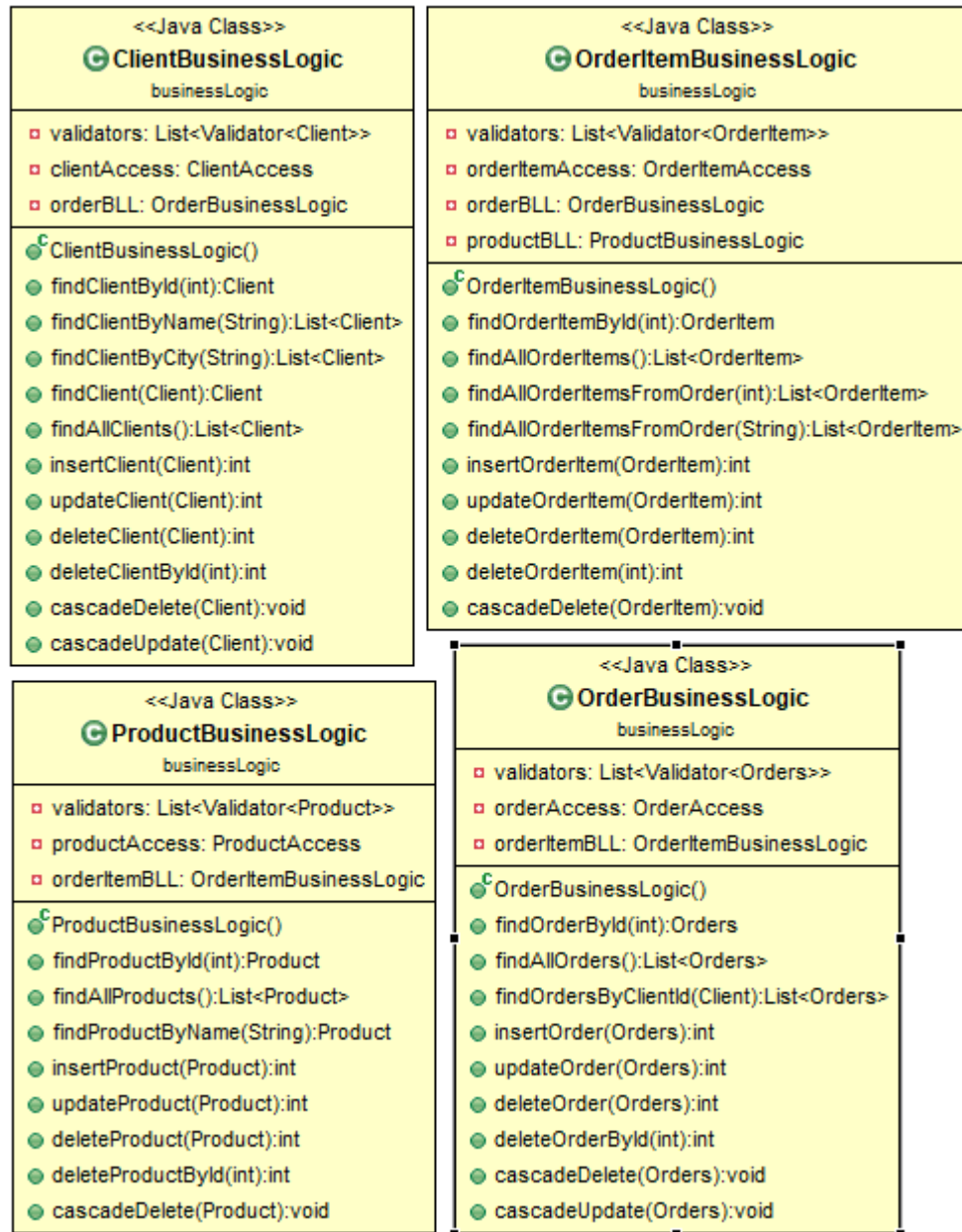


Figure 12.Business Logic Layer

The methods from these classes are self-explanatory and do not need any further description. A mention on the insert method of the *OrderItemBusinessLogic.java* class will be made in **Chapter 4, Implementation**. In the same chapter, the cascading methods will be presented.

## Presentation Layer

...

This is the layer that deals with parsing the input file, generating commands, delegating the commands and finally generating the output files. Mentions of all these aspects were made in the chapters before. Therefore, in what follows, the layout and different types of the output files will be presented.

This application produces 5 types of output files:

- logger.pdf which was already presented
- receipt.pdf which holds the bill generated for each new order item purchased

Client: Mihai Dobre  
Product bought: plum  
Quantity: 4  
Price: 7.0

Figure 13. Example of receipt.pdf file

- report\_clients.pdf which holds a table with all the clients inserted in the database

| ID | Name             | City     |
|----|------------------|----------|
| 1  | Mihai Dobre      | Bistrita |
| 2  | Mirel Constantin | Onesti   |
| 3  | Viorin Copoiu    | Onesti   |
| 4  | Neculai Isac     | Babenii  |

Figure 14. Example of report\_clients.pdf file

- report\_products.pdf which holds a table with all the products inserted in the database

| ID | Name     | Quantity | Price |
|----|----------|----------|-------|
| 2  | quince   | 50       | 3.15  |
| 3  | kale     | 25       | 7.2   |
| 4  | plum     | 60       | 1.75  |
| 5  | cucumber | 20       | 14.0  |

Figure 15. Example of report\_products.pdf file

- report\_orders.pdf which holds a table with all the orders that are found in the database

| ID | Client Name      | Order Item |          | Total |
|----|------------------|------------|----------|-------|
| 1  | Mihai Dobre      | Product    | Quantity | 59.1  |
|    |                  | quince     | 4        |       |
|    |                  | kale       | 5        |       |
|    |                  | plum       | 6        |       |
| 2  | Mirel Constantin | Product    | Quantity | 9.45  |
|    |                  | quince     | 3        |       |
| 3  | Viorin Copoiu    | Product    | Quantity | 28.8  |
|    |                  | kale       | 4        |       |

Figure 16. Example of report\_orders.pdf file

Keep in mind that these values are based on running the application with the input file presented in **Figure 1**.

Furthermore, if an error occurs when making a new order, such as there are not enough products in the database or the kind of product requested is not found in the database, then the receipt file will have the following message:

**Product cucumber was not inserted in the database**  
**Not enough apple in stock.**

Figure 17. Examples of receipts with error messages

Finally, bellow is presented the resulting files of running the application on the input file from the **Figure 1**.



logger@20200416\_21\_22\_13  
QaLzLReceipt\_Mihai Dobre\_1  
QaLzLReceipt\_Mihai Dobre\_4  
QaLzLReceipt\_Mihai Dobre\_5  
QaLzLReceipt\_Mihai Dobre\_6  
QaLzLReceipt\_Mihai Dobre\_7  
QaLzLReceipt\_Mirel Constantin\_2  
QaLzLReceipt\_Mirel Constantin\_8  
QaLzLReceipt\_Viorin Copoiu\_3  
QaLzLReport\_Clients\_1  
QaLzLReport\_Clients\_2  
QaLzLReport\_Orders\_5  
QaLzLReport\_Orders\_6  
QaLzLReport\_Products\_3  
QaLzLReport\_Products\_4  
QaLzLReport\_Products\_7

Figure 18. Files generated after running the application

## 4. Implementation

### Overview

Besides the classes imposed by the structure chosen for this application, some other packages were needed for a better segmentation of the classes and methods. These packages were presented in the **Chapter 3, Application design**.

It is only logical now to present the remaining two packages, namely the validators and start.order.manager.

The package with the validators consists of an interface named *Validator<T>.java* which is implemented by all the other classes of the package. They are all presented bellow.

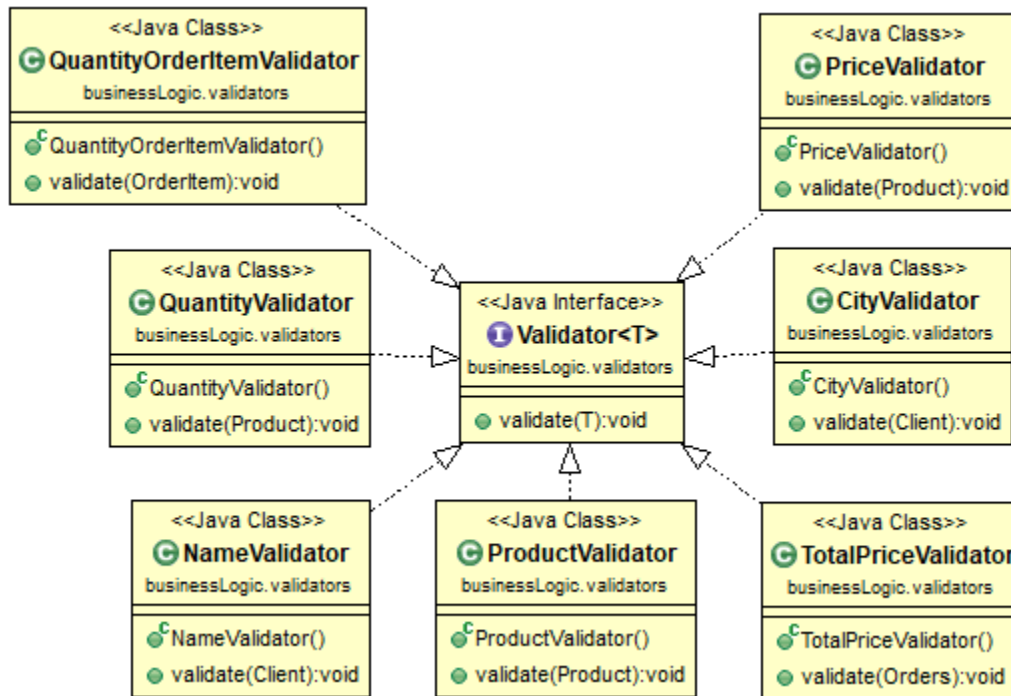


Figure 19. Validator package structure

For the sake of simplicity, only one of the validating method will be presented, the other ones following the same pattern of execution.

```
/**
 * This method checks the name of the client given
 * as parameter, namely it checks if the name of
 * the client is made up of letters only or contains
 * at most two white spaces.
 * @param object : Client
 * @throws IllegalArgumentException
 */
public void validate(Client object) {
 if(!object.getName().matches("^[A-Za-z]+$")) {
 throw new IllegalArgumentException("Client name is not a valid name!");
 }
}
```

Figure 20. The validate method of NameValidator.java

As it can be seen above, the application allows for client names that have at most one white space. In a similar way, the other methods validate certain fields of each model.

## **OrderItemBusinessLogic.java**

...

This class is the most important of the whole application and has the most explicit insert method of the whole package. This method's body is presented bellow.

```
public int insertOrderItem(OrderItem orderItem) {
 productBLL = new ProductBusinessLogic();
 orderBLL = new OrderBusinessLogic();
 for(Validator<OrderItem> validator : validators) {
 validator.validate(orderItem);
 }
 Orders order = orderBLL.findOrderById(orderItem.getId_order());
 if(order == null) return -2; //did not find the order in which to put this item
 Product product = productBLL.findProductByName(orderItem.getProd_name());
 if(product == null) return -3; //did not find the product from this item

 if(order != null) order.setTotal(order.getTotal() + orderItem.getQuantity() * product.getPrice());
 if(product.getQuantity() < orderItem.getQuantity()) return -4; //not enough stock;
 orderBLL.updateOrder(order);
 product.setQuantity(product.getQuantity() - orderItem.getQuantity());
 productBLL.updateProduct(product);

 List<OrderItem> ordersWithSameOrderId = findAllOrderItemsFromOrder(orderItem.getId_order());
 if(ordersWithSameOrderId == null) return orderItemAccess.insert(orderItem);
 List<OrderItem> ordersWithSameProdName = findAllOrderItemsFromOrder(orderItem.getProd_name());
 if(ordersWithSameProdName != null) {
 for(OrderItem found : ordersWithSameProdName) {
 if(found.getId_order() == orderItem.getId_order()) {
 orderItem.setQuantity(found.getQuantity() + orderItem.getQuantity());
 orderItemAccess.update(found, orderItem);
 return 1;
 }
 }
 }
 return orderItemAccess.insert(orderItem);
}
```

Figure 21. insertOrderItem method

This method inserts a new entry in the "orderitem" table. If an entry with the same id\_order as the one passed as the parameter is found, the method updates the "order" table, specifically the entry with the same id as the id\_order and then inserts the new entry in the "orderitem" table. If an entry with the same id\_order and prod\_name as the one passed as the parameter is found, the method updates the "order" table, specifically the entry with the same id as the id\_order and updates the quantity of the entry that was found to be similar to the argument.

In a similar way, the update of an order item is performed, when being called from the *MainClass.java*.

## 5. Results

### **Overall view of the application**

...

Since all the modules have been presented, it is only logical that the whole UML diagram follows. (check the folder of the documentation)

### **Testing**

...

Certain scenarios were tested during and after the development of the “Order Manager” application. These test are stored in their own separate folders and are linked together as follows:

- **in-myTest.txt** contains the input data that generated the output in the folder **Output in-myTest**. Also, the dump files that were generated after running the application with this file as the parameter are stored in the same folder
- **commands.txt** contains the input data that generated the output in the folder **Output example**. Also, the dump files that were generated after running the application with this file as the parameter are stored in the folder **sqldump\_example**. The contents of the input files are presented bellow:

Read the *AAA\_README\_FIRST.txt* file for more details.

### **User interface**

...

For this application, there is not a concrete user interface defined. The way in which a user interacts with the application is through the input and output files that are read and updated by the application.

It is done so in such a way that the input is easy to write and has a solid foundation on which the application can run and also the output is presented to the user in a way that is easy to follow and to further use for future applications.

## 6. Conclusions and further development

Although it benefits the user to not have a user interface, it is possible to develop one and make the application available for more unexperienced users.

Another development that can be made is making the application capable of parsing more instructions, since the methods for these possible instructions are already implemented, as it was presented in **Chapter 3, Design decisions**

## 7. Bibliography

[http://coned.utcluj.ro/~salomie/PT\\_Lic/4\\_Lab/Assignment\\_3/Java\\_Concurrency.pdf](http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_3/Java_Concurrency.pdf)

[https://www.lucidchart.com/documents#/templates?folder\\_id=home&browser=icon](https://www.lucidchart.com/documents#/templates?folder_id=home&browser=icon)

[https://www.programcreek.com/java-api-](https://www.programcreek.com/java-api-examples/?class=com.itextpdf.text.Document&method=newPage)

[examples/?class=com.itextpdf.text.Document&method=newPage](https://www.programcreek.com/java-api-examples/?class=com.itextpdf.text.Document&method=newPage)

<https://www.codota.com/code/java/methods/java.beans.PropertyDescriptor/getWriteMethod>

<https://www.w3schools.com/sql/>

