Order management

Documentation

1. Homework objective

Main objective:

The implementation of an application for processing customer orders for a warehouse. It takes as input a file with operations & data about clients, products and orders and executes the given commands. The output of the given operations will be displayed in PDF files.

Secondary objectives:

-creating an efficient file parser, which parses the lines from the given input file and executes the necessary operations

-creating an efficient PDF generator, which will generate PDF files containing information about the required object

-documenting the code using Javadoc

-creating a relational database & working with queries

-creation of a .jar file

1. Problem analysis, modeling, scenarios, use cases

Problem analysis:

It is required to correctly and efficiently implement an application for processing orders for a warehouse, taking as input a .txt file containing information about the clients, products & orders and different commands. The .txt file is parsed using a file parser, and objects for clients, products and orders are created. The commands are executed by prepared statements on different queries. Some commands require the creation of a .pdf file, which will contain information about the requested objects. The .pdf files are created by a pdf generator, which either creates tables in the output file, a bill containing information about the order or an error message.

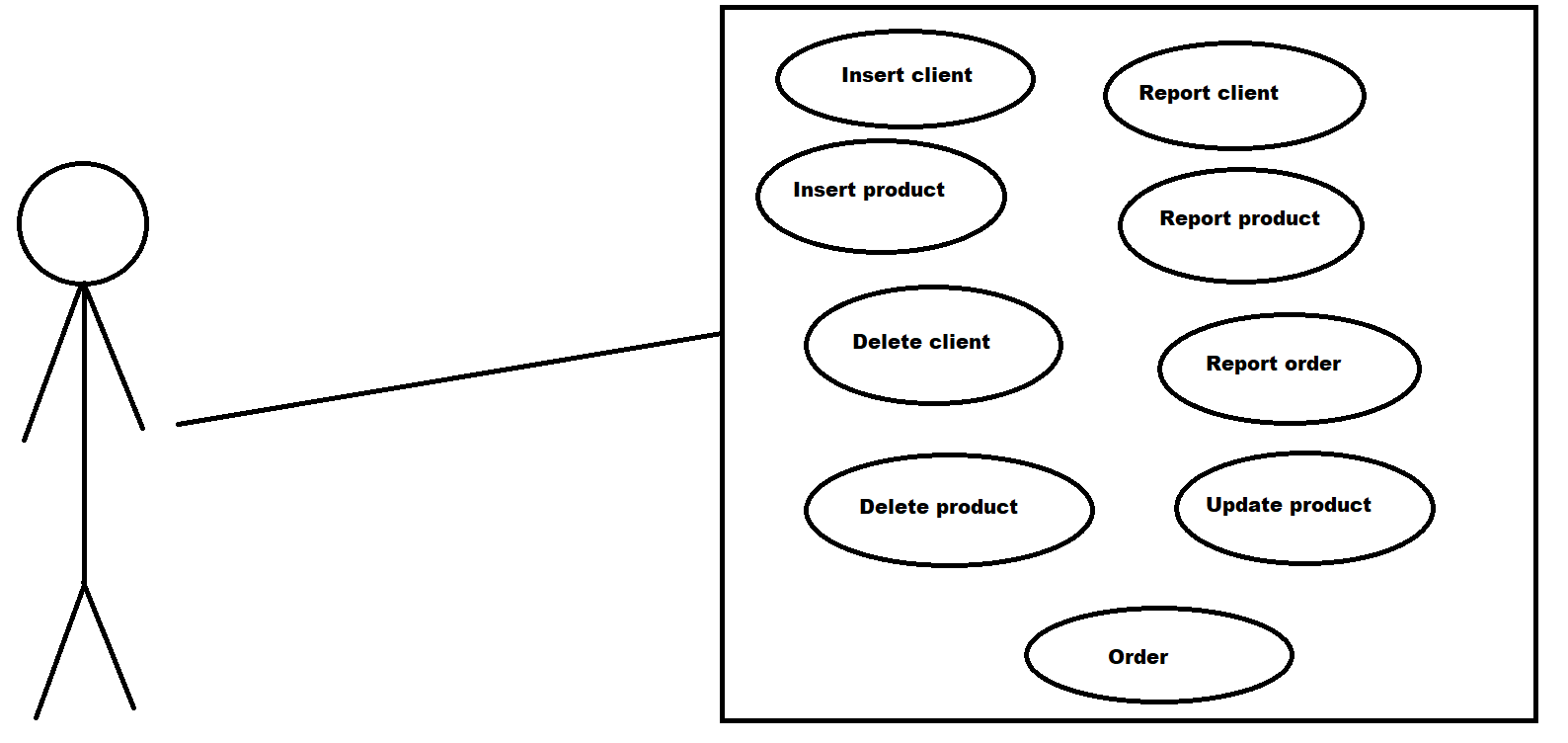
Modeling:

The project is modeled in a layered architecture manner, consisting of 5 different packages: start, presentation, model, dataAccess and DAO. The client, order and product classes are under the mode package. ClientDAO, ProductDAO and OrderDAO classes (classes for accessing the objects) are under the DAO package. The ConnectionFactory which realizes the connection to the database is under the dataAccess package. FIleParser and PdfGenerator, which are the input & output classes, are under the presentation package. The start package only consists of the Main class.

Scenarios:

A scenario represents a sequence of steps taken by the user to correctly interact with the application. In our case, there is a single valid scenario: the input file given as argument will be parsed by the file parser. The input file consists of insert/delete client/product/order & order commands. For any insert/delete command, our database will be modified according to the respective command. For any report command, a .pdf file will be generated, containing information about the requested field. Any other ‘unknown’ command would throw a runtime error.

Use cases:



1. Design

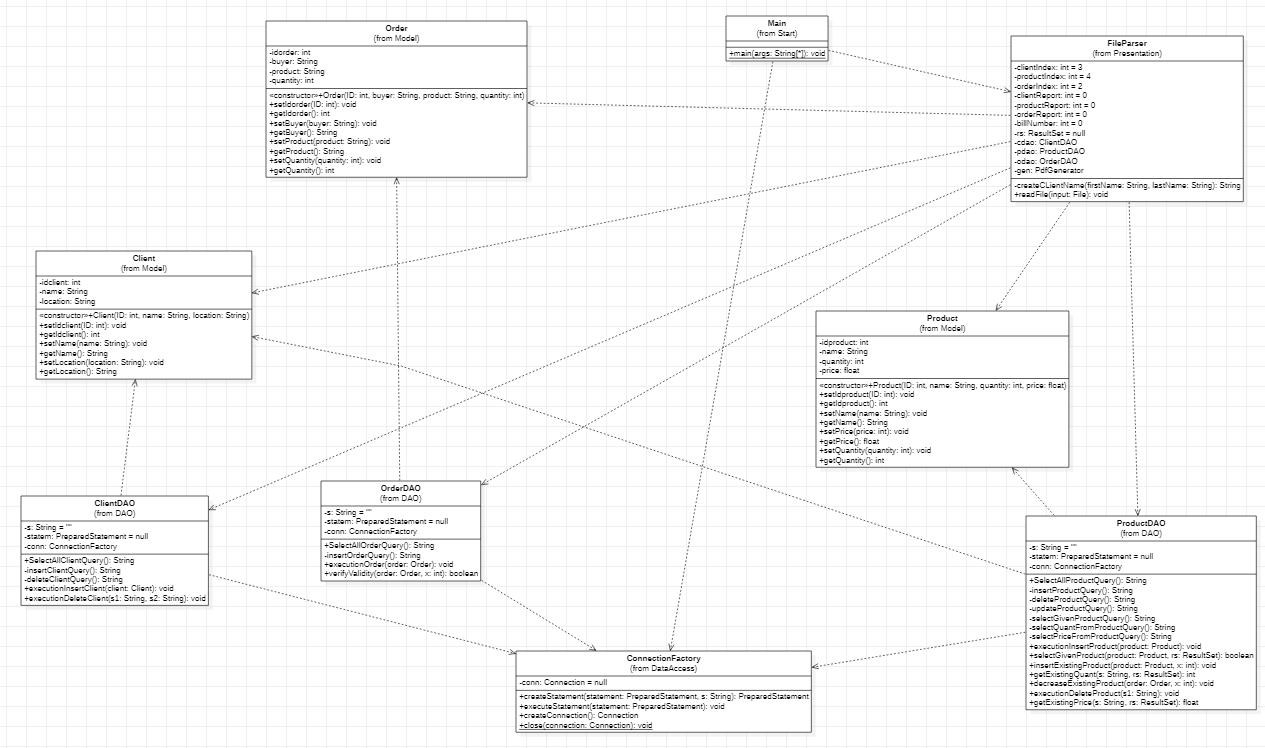
The project is designed under a layered architecture pattern, consisting of 5 different components (packages): Start, DataAccess, DAO, Presentation & Model. Model contains the client, product & order classes. Start contains the Main class. DataAccess contains the ConnectionFactory class, in which the connection to the database is established, and the statement is created & executed. DAO contains the ClientDAO, ProductDAO & OrderDAO classes, in which the requested queries for each one are created: select, insert, delete & update. They also contain the methods for executing these queries, and other additional methods that help with the execution. Finally, the Presentation package contains the FileParser & PdfGenerator classes, which handle the interaction with the input & output files.

Each DAO class contains methods for executing SQL queries. The queries initially contain “?” characters where the specified column should be, which will be eventually set to the necessary values using the prepares statement’s properties. Each of these methods ends with the execution of the statement on the specified query, which will modify the database with respect to the instruction: insert into a specific table, delete from a specific table, update a field in a specific table or select all/a given field from a specific table.

The file parser takes as input a .txt file containing data: instructions to be turned into queries and executed and information about clients/products/orders. Each line is read separately, split by space regex, and computed. All information about clients/products/orders is saved, and the necessary methods are called, with respect to the given command. The instructions could be: insert, delete, report(creation of a .pdf containing information about the given product), order(insert + creating a .pdf containing billing info/an error + update in the product database).

The pdf generator contains methods that create pdf files for clients, products & orders, also with billing information. The reports for clients, products & orders will contain a table with all the information within the respective database table. The billing report will contain some information about the order, such as the buyer, the product, quantity & price. The error report will take the form of a bill, but instead of containing billing information, it will contain an error message, the requested quantity & the available quantity of the wanted product.

The ConnectionFactory class contains methods for opening & closing a database connections, and methods for creating & executing statements. For creating a statement, a null prepared statement is prepared with a given query. For executing a statement, a prepared statement simply calls the execute update command.



1. Implementation

Classes:

Client:

Has the following attributes:

-idclient: the ID of the client

-name: the name of the client

-location: the location of the client

The constructor of the client class sets the id, name and location of the client.

Has the following methods:

-setters for id, name and location – set the specific field

-getters for id, name and location – return the specific field

Order:

Has the following attributes:

-idorder: the ID of the order

-buyer: the name of the person who places the order

-product: the name of the ordered product

-quantity: the quantity of the ordered product

The constructor of the order class sets the ID, buyer, product and quantity of the order.

Has the following methods:

-setters for id, buyer, product and quantity – set the specific fields

-getters for id, buyer, product and quantity – return the specific field

Product:

Has the following attributes:

-idproduct: the ID of the product

-name: the name of the product

-quantity: the quantity of the product

-price: the price of the product (/1 item)

The constructor of the product class sets the ID, name, quantity and price of the product.

Has the following methods:

-setters for id, name, quantity and price – set the specific fields

-getters for id, name, quantity and price – returns the specific fields

ConnectionFactory:

Has the following attributes:

-conn: a Connection type attribute, which holds the connection to the database

Has the following methods:

public PreparedStatement createStatement(PreparedStatement statement, String s) {  
 try {  
 conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/schooldb?autoReconnect=true&useSSL=false","root","tsmtheoddone1");  
 statement = conn.prepareStatement(s);  
 }  
 catch (SQLException ex) {  
 ex.getStackTrace();  
 } return statement;  
}

This method takes as parameters a null prepared statement and a string containing a query. The statement will be prepared with the received query, and then it will be returned.

public void executeStatement(PreparedStatement statement) {  
 try {  
 statement.executeUpdate();  
 } catch (SQLException exception) {  
 exception.printStackTrace();  
 }  
}

This method takes as parameter a prepared statement prepared on a query, and executes it with the executeUpdate() command. This will have effect on a specific table from the database, with respect to the query.

public Connection createConnection() {  
 try {  
 conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/schooldb?autoReconnect=true&useSSL=false","root","password");  
 }  
 catch (Exception exception) {  
 exception.printStackTrace();  
 }  
 return conn;  
}

This method sets the connection to the database and returns it.

public static void close(Connection connection) throws SQLException {  
 connection.close();  
 System.*out*.println("Connection closed.");  
}

The 3rd item from the bibliography section helped me solve connectivity issues I had during this part of the application.

This method takes as parameter an open connection and closes it.

ClientDAO:

Has the following attributes:

-s: initially an empty string, it will serve as the string containing the query which will eventually be executed

-statem: initially a null prepared statement, it will serve as the statement prepared on specific queries, with the purpose of executing them

-conn: a ConnectionFactory object, which allows us to call different methods & attributes within the class

Has the following methods:

public String SelectAllClientQuery() {  
 s = "select \* from client";  
 return s;  
}

This method creates a ‘select all from client’ query and returns it.

private String insertClientQuery() {  
 s = "insert into client (idclient, name, location) values (?, ?, ?)";  
 return s;  
}

This method creates a ‘insert into client’ query and returns it.

private String deleteClientQuery() {  
 s = "delete from client where (name=? AND location=?)";  
 return s;  
}

This method creates a ‘delete from client’ query and returns it.

public void executionInsertClient(Client client) throws SQLException {  
 statem = conn.createStatement(statem, insertClientQuery());  
 statem.setInt(1,client.getIdclient());  
 statem.setString(2, client.getName());  
 statem.setString(3, client.getLocation());  
 conn.executeStatement(statem);  
}

This method takes as parameter a client. It prepares a statement with the insert query, replaces all the ‘?’ values within the query with the expected values from the client object, then executes it. It will insert a client into the clients table of the database.

public void executionDeleteClient(String s1, String s2) throws SQLException {  
 statem = conn.createStatement(statem, deleteClientQuery());  
 statem.setString(1,s1);  
 statem.setString(2, s2);  
 conn.executeStatement(statem);  
}

This method takes as parameters two strings, the name & location of the client that needs to be deleted. It prepares a statement with the delete query, replaces all the ‘?” values within the query with the strings given as parameters, then executes it. It will delete a client from the clients table of the database.

OrderDAO:

Has the following attributes:

-s: initially an empty string, it will serve as the string containing the query which will eventually be executed

-statem: initially a null prepared statement, it will serve as the statement prepared on specific queries, with the purpose of executing them

-conn: a ConnectionFactory object, which allows us to call different methods & attributes within the class

Has the following methods:

public String SelectAllOrderQuery() {  
 s = "select \* from `order`";  
 return s;  
}

This method creates ‘select all from order’ query and returns it.

private String insertOrderQuery() {  
 s = "insert into `order` (idorder, buyer, product, quantity) values (?, ?, ?, ?)";  
 return s;  
}

This method creates a ‘insert into order’ query and returns it.

public void executionOrder(Order order) throws SQLException {  
 statem = conn.createStatement(statem, insertOrderQuery());  
 statem.setInt(1, order.getIdorder());  
 statem.setString(2, order.getBuyer());  
 statem.setString(3, order.getProduct());  
 statem.setInt(4,order.getQuantity());  
 conn.executeStatement(statem);  
}

This method takes as parameter an order object. It prepares a statement with the insert query, replaces all the ‘?” values within the query with the expected values from the order object, then executes it. It will insert an order into the orders table of the database.

public boolean verifyValidity(Order order, int x) {  
 return x - order.getQuantity() >= 0;  
}

This method takes as parameters an order object and an integer, representing the quantity of a product from the products table. It subtracts the ordered quantity of the product from the available quantity. It returns 1 if the result is >= 0 (the order can be placed), and 0 otherwise (the order can not be placed because the stock is insufficient).

ProductDAO:

Has the following attributes:

-s: initially an empty string, it will serve as the string containing the query which will eventually be executed

-statem: initially a null prepared statement, it will serve as the statement prepared on specific queries, with the purpose of executing them

-conn: a ConnectionFactory object, which allows us to call different methods & attributes within the class

Has the following methods:

public String SelectAllProductQuery() {  
 s = "select \* from product";  
 return s;  
}

This method created a ‘select all from product’ query and returns it.

private String insertProductQuery() {  
 s = "insert into `product` (idproduct, name, quantity, price) values (?, ?, ?, ?)";  
 return s;  
}

This method creates a ‘insert into product’ query and returns it.

private String deleteProductQuery() {  
 s = "delete from product where (name=?)";  
 return s;  
}

This method creates a ‘delete from product’ query and returns it.

private String updateProductQuery() {  
 s = "update `product` set quantity=? where name=?";  
 return s;  
}

This method creates a ‘update product’ query and returns it.

private String selectGivenProductQuery() {  
 s = "select \* from product where (name=? AND price=?)";  
 return s;  
}

This method creates a ‘select all from product WHERE’ query and returns it.

private String selectQuantFromProductQuery() {  
 s = "select quantity from product where name=?";  
 return s;  
}

This method creates a ‘select quantity from product were’ query and returns it.

private String selectPriceFromProductQuery(){  
 s = "select price from product where name=?";  
 return s;  
}

This method creates a ‘select price from product where’ query and returns it.

public void executionInsertProduct(Product product)throws SQLException {  
 statem = conn.createStatement(statem, insertProductQuery());  
 statem.setInt(1, product.getIdproduct());  
 statem.setString(2, product.getName());  
 statem.setInt(3, product.getQuantity());  
 statem.setFloat(4, product.getPrice());  
 conn.executeStatement(statem);  
}

This method takes as parameter a product object. It prepares a statement with the insert query, replaces all the ‘?’ values within the query with the expected values from the product object, then executes it. It will insert a product into the products table of the database.

public boolean selectGivenProduct(Product product, ResultSet rs) throws SQLException {  
 statem = conn.createStatement(statem, selectGivenProductQuery());  
 statem.setString(1,product.getName());  
 statem.setFloat(2,product.getPrice());  
 rs = statem.executeQuery();  
 return rs.next();  
}

This method takes as parameters a product object and a result set. It prepares a statement with the select query, replaces all the ‘?’ values within the query with the expected values from the product object, then executes it & saves the result into the result set. It will return 0 if the given product could be found in the products table, and 0 otherwise.

public void insertExistingProduct(Product product, int x) throws SQLException {  
 statem = conn.createStatement(statem, updateProductQuery());  
 statem.setInt(1, product.getQuantity() + x);  
 statem.setString(2, product.getName());  
 conn.executeStatement(statem);  
}

This method takes as parameter a product object and an integer value, the quantity of the object that needs to be added. It prepares a statement with the update query, replaces all the ‘?’ values within the query with the expected values from the product object, then executes it. Considering that the product already exists in the database, it doesn’t need to be added again, but instead, its quantity must be modified. The method will update the ‘quantity’ field of the product (old value + new value).

public int getExistingQuant(String s, ResultSet rs) throws SQLException {  
 statem = conn.createStatement(statem, selectQuantFromProductQuery());  
 statem.setString(1,s);  
 rs = statem.executeQuery();  
 rs.next();  
 return rs.getInt("quantity");  
}

This method takes as parameter a string (the name field of a product object) and a result set. It prepares a statement with the select query, replaces the ‘?’ value within the query with the string given as parameter, then executes it & saves the result into the result set. It will return the value found on the ‘quantity’ column of the given product (the product’s quantity from the table).

public void decreaseExistingProduct(Order order, int x) throws SQLException {  
 statem = conn.createStatement(statem, updateProductQuery());  
 statem.setInt(1, x - order.getQuantity());  
 statem.setString(2, order.getProduct());  
 conn.executeStatement(statem);  
}

This method takes as parameters an order object and an integer, the quantity of a specific product from the products table. It prepares a statement with the update query, replaces all the ‘?’ values within the query with the expected values from the product object, then executes it. When an order is created, the quantity of the ordered product must be modified. The method updates the quantity of the given product in the products table (old quantity – ordered quantity).

public void executionDeleteProduct(String s1) throws SQLException {  
 statem = conn.createStatement(statem, deleteProductQuery());  
 statem.setString(1,s1);  
 conn.executeStatement(statem);  
}

This method takes as parameter a string, the name of a specific product. It prepares a statement with the delete query, replaces the ‘?’ value within the query with the name of the product, then executes it. It deletes a product from the products table of the database.

public float getExistingPrice(String s, ResultSet rs) throws SQLException {  
 statem = conn.createStatement(statem, selectPriceFromProductQuery());  
 statem.setString(1,s);  
 rs = statem.executeQuery();  
 rs.next();  
 return rs.getFloat("price");  
}

This method takes as parameter a string (the name field of a product object) and a result set. It prepares a statement with the select query, replaces the ‘?’ value within the query with the string given as parameter, then executes it & saves the result into the result set. It will return the value found on the ‘price’ column of the given product (the product’s price from the table).

FileParser:

Has the following fields:

-clientIndex: the ID of the client (gets incremented)

-productIndex: the ID of the product (gets incremented)

-orderIndex: the ID of the order (gets incremented)

-clientReport: the ID of the client report (gets incremented)

-productReport: the ID of the product report (gets incremented)

-orderReport: the ID of the order report (gets incremented)

-billNumber: the ID of the bill report (gets incremented)

-rs: a result set, initially empty, will be sent as parameters to different method calls

-cdao: a ClientDAO object, which allows us to call different methods & attributes within the class

-pdao: a ProductDAO object, which allows us to call different methods & attributes within the class

-odao: a OrderDAO object, which allows us to call different methods & attributes within the class

Has the following methods:

private String createCLientName(String firstName, String lastName) {  
 String fullName ="";  
 fullName = firstName + " " + lastName;  
 fullName = fullName.substring(0, fullName.length() - 1);  
 return fullName;  
}

This method takes as parameters two strings representing the first name & last name of a client, read from the input file. It returns a string containing the full name.

public void readFile(File input) throws SQLException, FileNotFoundException, DocumentException {  
 Scanner scan;  
 scan = new Scanner(input);  
 while (scan.hasNextLine()) {  
 String[] s = scan.nextLine().split(" ");  
 switch (s[0].toLowerCase()) {  
 case "insert":  
 if (s[1].toLowerCase().equals("client:")) {  
 Client client = new Client(++clientIndex, createCLientName(s[2], s[3]), s[4]);  
 cdao.executionInsertClient(client);  
 }  
 else if (s[1].toLowerCase().equals("product:")) {  
 Product product = new Product(productIndex, s[2].substring(0, s[2].length() - 1), Integer.*parseInt*(s[3].substring(0, s[3].length() - 1)), Float.*parseFloat*(s[4]));  
 if (!pdao.selectGivenProduct(product, rs)) {  
 pdao.executionInsertProduct(product);  
 ++productIndex;  
 }  
 else  
 pdao.insertExistingProduct(product, pdao.getExistingQuant(product.getName(), rs));  
 }  
 break;  
 case "order:":  
 Order order = new Order(++orderIndex, createCLientName(s[1], s[2]), s[3].substring(0, s[3].length() - 1), Integer.*parseInt*(s[4]));  
 if (odao.verifyValidity(order, pdao.getExistingQuant(order.getProduct(), rs))) {  
 odao.executionOrder(order);  
 pdao.decreaseExistingProduct(order, pdao.getExistingQuant(order.getProduct(), rs));  
 gen.createBill(order, ++billNumber, true, pdao.getExistingPrice(order.getProduct(), rs), pdao.getExistingQuant(order.getProduct(),rs));  
 }  
 else  
 gen.createBill(order, ++billNumber, false, pdao.getExistingPrice(order.getProduct(), rs), pdao.getExistingQuant(order.getProduct(),rs));  
 break;  
 case "delete":  
 if (s[1].toLowerCase().equals("client:"))  
 cdao.executionDeleteClient(createCLientName(s[2], s[3]), s[4]);  
 else if (s[1].toLowerCase().equals("product:"))  
 pdao.executionDeleteProduct(s[2]);  
 break;  
 case "report":  
 if (s[1].toLowerCase().equals("client"))  
 gen.createClientReport(++clientReport);  
 else if (s[1].toLowerCase().equals("product"))  
 gen.createProductReport(++productReport);  
 else  
 gen.createOrderReport(++orderReport);  
 break;  
 default:  
 throw new RuntimeException("Error with input format");  
 }  
 }  
}

This method takes as parameter the input file. It creates a scanner object used to read elements from the file. The file is read line by line, each line being split by the space vertex. Then, it verifies the instruction given: insert/delete/report/order. For each instruction given, it verifies what kind of object needs to be manipulated: product/order/client. Then, it creates the object with the values given as data and performs the operation given as instruction. If none of the {insert, delete, order, report} operations is given as the first parameter of a line, a new runtime exception will be thrown with the “Error with input format” message. The 2nd bibliography item was used during the creation of this method.

PdfGenerator:

Has the following attributes:

-cdao: a ClientDAO object, which allows us to call different methods & attributes within the class

-pdao: a ProductDAO object, which allows us to call different methods & attributes within the class

-odao: a OrderDAO object, which allows us to call different methods & attributes within the class

Has the following methods:

void createClientReport(int index) throws FileNotFoundException, DocumentException, SQLException {  
 Document doc = new Document();  
 PdfWriter.*getInstance*(doc, new FileOutputStream("ClientReport" + index + ".pdf"));  
 doc.open();  
  
 ConnectionFactory conn = new ConnectionFactory();  
 Connection connection = conn.createConnection();  
  
 PdfPTable table = new PdfPTable(3);  
 Stream.*of*("idclient", "name", "location").forEach(columnTitle -> {  
 PdfPCell header = new PdfPCell();  
 header.setPhrase(new Phrase(columnTitle));  
 table.addCell(header);  
 });  
  
 PreparedStatement statem = connection.prepareStatement(cdao.SelectAllClientQuery());  
 ResultSet rs = statem.executeQuery();  
  
 while (rs.next()){  
 table.addCell(String.*valueOf*(rs.getInt("idclient")));  
 table.addCell(String.*valueOf*(rs.getString("name")));  
 table.addCell(String.*valueOf*(rs.getString("location")));  
 }  
  
 doc.add(table);  
 doc.close();  
}

This method takes as parameter an integer, which will be the index of the report. It creates a new document, renames it, sets the connection to the database, creates a new table with 3 cells/row and sets the header of the table. Then, it executes a prepared statement on the ‘select all from client’ query. It goes through each line of the table and sets the cells’ values to the values found in the database. Finally, it adds the table to the document and closes it.

void createProductReport(int index) throws FileNotFoundException, DocumentException, SQLException {  
 Document doc = new Document();  
 PdfWriter.*getInstance*(doc, new FileOutputStream("ProductReport" + index + ".pdf"));  
 doc.open();  
  
 ConnectionFactory conn = new ConnectionFactory();  
 Connection connection = conn.createConnection();  
  
 PdfPTable table = new PdfPTable(4);  
 Stream.*of*("idproduct", "name", "quantity", "price").forEach(columnTitle -> {  
 PdfPCell header = new PdfPCell();  
 header.setPhrase(new Phrase(columnTitle));  
 table.addCell(header);  
 });  
  
 PreparedStatement statem = connection.prepareStatement(pdao.SelectAllProductQuery());  
 ResultSet rs = statem.executeQuery();  
  
 while (rs.next()){  
 table.addCell(String.*valueOf*(rs.getInt("idproduct")));  
 table.addCell(String.*valueOf*(rs.getString("name")));  
 table.addCell(String.*valueOf*(rs.getInt("quantity")));  
 table.addCell(String.*valueOf*(rs.getFloat("price")));  
 }  
  
 doc.add(table);  
 doc.close();  
}

This method takes as parameter an integer, which will be the index of the report. It creates a new document, renames it, sets the connection to the database, creates a new table with 4 cells/row and sets the header of the table. Then, it executes a prepared statement on the ‘select all from product’ query. It goes through each line of the table and sets the cells’ values to the values found in the database. Finally, it adds the table to the document and closes it.

void createOrderReport(int index) throws FileNotFoundException, DocumentException, SQLException {  
 Document doc = new Document();  
 PdfWriter.*getInstance*(doc, new FileOutputStream("OrderReport" + index + ".pdf"));  
 doc.open();  
  
 ConnectionFactory conn = new ConnectionFactory();  
 Connection connection = conn.createConnection();  
  
 PdfPTable table = new PdfPTable(4);  
 Stream.*of*("idorder", "buyer", "product", "quantity").forEach(columnTitle -> {  
 PdfPCell header = new PdfPCell();  
 header.setPhrase(new Phrase(columnTitle));  
 table.addCell(header);  
 });  
  
 PreparedStatement statem = connection.prepareStatement(odao.SelectAllOrderQuery());  
 ResultSet rs = statem.executeQuery();  
  
 while (rs.next()){  
 table.addCell(String.*valueOf*(rs.getInt("idorder")));  
 table.addCell(String.*valueOf*(rs.getString("buyer")));  
 table.addCell(String.*valueOf*(rs.getString("product")));  
 table.addCell(String.*valueOf*(rs.getInt("quantity")));  
 }  
  
 doc.add(table);  
 doc.close();  
}

This method takes as parameter an integer, which will be the index of the report. It creates a new document, renames it, sets the connection to the database, creates a new table with 4 cells/row and sets the header of the table. Then, it executes a prepared statement on the ‘select all from order’ query. It goes through each line of the table and sets the cells’ values to the values found in the database. Finally, it adds the table to the document and closes it.

void createBill(Order order, int index, boolean state, float price, int quant) throws FileNotFoundException, DocumentException {  
 Document doc = new Document();  
 PdfWriter.*getInstance*(doc, new FileOutputStream("OrderBill" + index + ".pdf"));  
 doc.open();  
  
 Paragraph message = new Paragraph("Bill no. " + index + "\n");  
 doc.add(message);  
  
 if (!state){  
 message = new Paragraph("Order could not be placed: product out of stock");  
 doc.add(message);  
 message = new Paragraph("Ordered quantity: " + order.getQuantity() + "\n");  
 doc.add(message);  
 message = new Paragraph("Stock: " + quant + "\n");  
 doc.add(message);  
 doc.close();  
 }  
 else {  
 message = new Paragraph("Order id: " + order.getIdorder() + "\n");  
 doc.add(message);  
 message = new Paragraph("Order recipient: " + order.getBuyer() + "\n");  
 doc.add(message);  
 message = new Paragraph("Item bought: " + order.getProduct() + "\n");  
 doc.add(message);  
 message = new Paragraph("Quantity bought: " + order.getQuantity() + "\n");  
 doc.add(message);  
 message = new Paragraph("Price: " + order.getQuantity() \* price + "\n");  
 doc.add(message);  
 doc.close();  
 }  
}

This method takes as parameters an order object, an integer (the index of the bill), a Boolean state (1 if it needs to create a bill with billing info, 0 if it needs to create a bill with an error message), the price of the ordered object and the quantity of the ordered object (both needed for the bill with an error message). It creates a new document, renames it and sets the connection to the database. Then, it verifies if it needs to create a bill with billing info or a bill with an error message. If it needs to create a bill with billing info, it adds messages into the document containing relevant information about the order, then closes it. If it needs to create a bill with an error messages, it adds an error message containing the ordered quantity & the available stock into the document, then closes it. This block of code was created with help from the 1st bibliography item.

Main:

The class which contains the main method. The main method is the entry point of any java program. The main method consists of a connection to the database, the definition of the input file, creation of a new file parser object and calling of the readFile method on the input file.

1. **Conclusions**

This assignment has given me a better understanding of working with scanners and pdf generators, and helped me get closer to organizing my code in a proper way. Also, it has given me the chance of learning how to work with statements, databases and result sets. In conclusion, this assignment has been a great opportunity to both learn new things and revise the information that I already had.

1. Bibliography

1.<https://www.baeldung.com/java-pdf-creation>

2.<https://www.tutorialspoint.com/java/java_string_tolowercase.htm>

3.<https://stackoverflow.com/questions/34189756/warning-about-ssl-connection-when-connecting-to-mysql-database>