TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

Fundamental Programming Techniques

Order Management

Assignment 3

Hunor Debreczeni

Group: 30423

Table of Contents

Objective

The objective of the application is to manage the orders for the customers in a warehouse. It should be allowed to add new product with specific quantities and specific prices. Clients are also needed to be saved for order generation.

To achieve this, relational databases are used to store the products, the clients and the orders. For an easier layout in the application, it is structured in packages using a layered architecture by using Model, Business Logic, Presentation and Data Access classes.

The software also allows processing commands from an input file whose name can be given as an argument to the program.

Problem Analysis, Modeling, Scenarios, Use cases

The application may be used to help smaller businesses to easily store data about orders, clients and also about products. Each order can have many invoices due to the multiple type of invoices (returned, canceled, normal, etc.).

Firstly, the user of the software should have a database set up ready to use. For this part, there is an SQL file, which creates the necessary database and the tables inside it upon import. In order to be able to connect to the database, there should be a user set up like described below:

The **username** must be **root**

The **password** must be **secret**

Secondly, commands should pe written in the **input** file, according to the user's needs. These commands must follow the following formats:

- Insert
    - Client: <Name of Client>, <Address of Client>
        - Inserts a new client into the DB
    - Product: <Name of Product>, <Quantity of Product>, <Price of Product>
        - Inserts a new product if it doesn't exist, otherwise the quantity of the original product is incremented
- Delete
    - Client: <Name of Client>
        - Deletes the client with the given name if found
    - Product: <Name of Product>
        - Deletes the product with the given name if found
- Order: <Name of Client>, <Name of Product>, <Quantity of Products to be bought>
    - If there is enough stock of the product to make the current order, then an invoice is made with the given details, otherwise an under-stock message is printed through a PDF file.
- Report
    - Client
        - Generates a PDF with all the current clients
    - Product
        - Generates a PDF with all the current products
    - Order
        - Generates a PDF with all the orders that were made

The **output**s of the application are PDF files, that can be called following the commands listed above. Each file's name contains details about the given command or the given situation (in case of the under-stock message) and a timestamp attached to it. This way, it makes easier to follow if multiple commands are used.

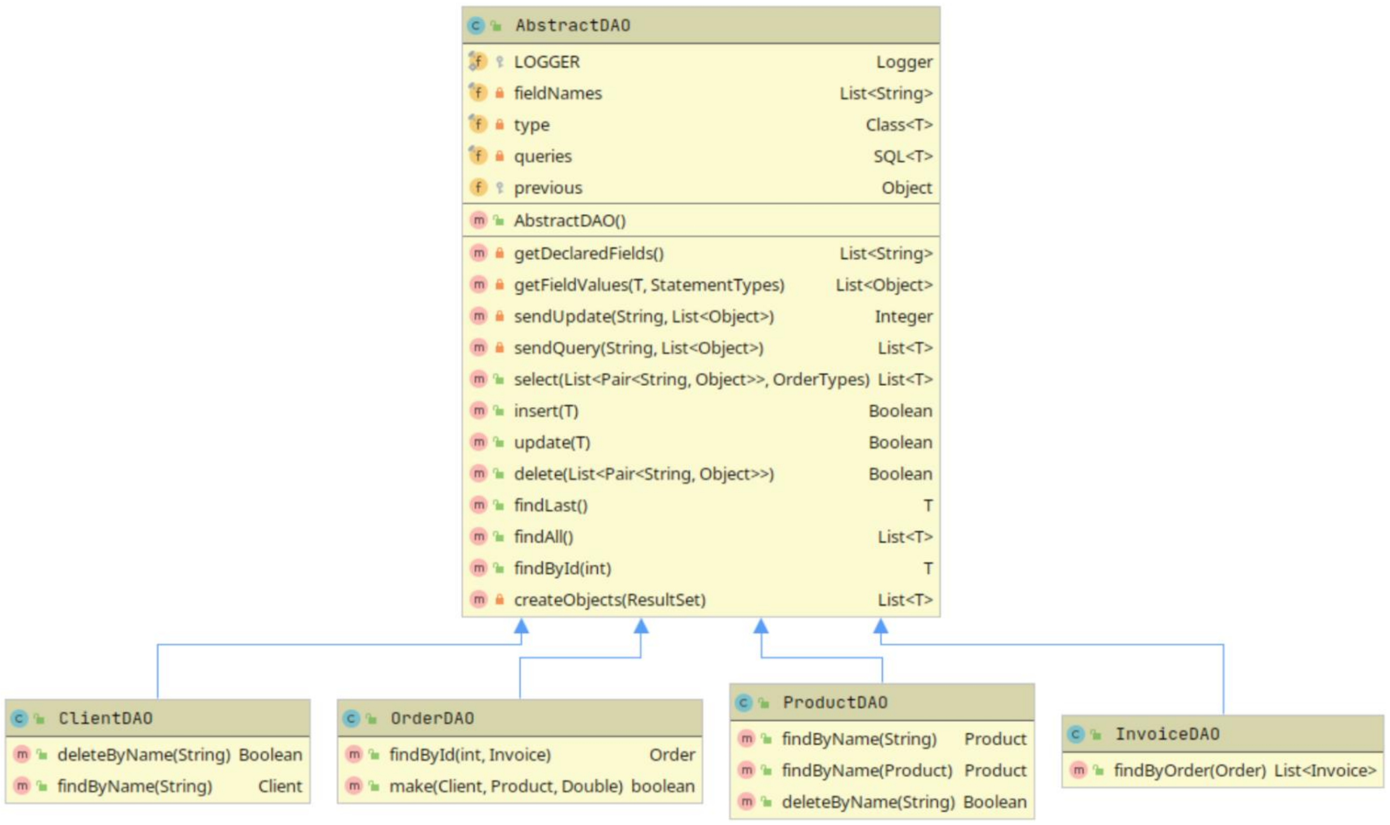
As **arguments** for the program, only one can be used, to specify the name of the input file, that must be in the same folder as the .jar file. In case it is not used, the program will search for the *commands.txt* file by default.

Design

For implementing the problem described above, the best and most efficient approach is to use reflectional techniques. Reflection gives us pieces of information about the class to which a given object belongs to. We may also get details about the methods of that class that can also be executed by referring to the object of that class. This way it's easy to invoke methods at runtime without knowing the type of the current instantiated class object.

Using this technique, it's easy to:

- Get information about the fields of an object even if they are private
- Use its constructors
- Access any method and invoke them


AbstractDAO
LOGGER Logger
fieldNames List<String>
type Class<T>
queries SQL<T>
previous Object
AbstractDAO()
getDeclaredFields() List<String>
getFieldValues(T, StatementTypes) List<Object>
sendUpdate(String, List<Object>) Integer
sendQuery(String, List<Object>) List<T>
select(List<Pair<String, Object>>, OrderTypes) List<T>
insert(T) Boolean
update(T) Boolean
delete(List<Pair<String, Object>>) Boolean
findLast() T
findAll() List<T>
findById(int) T
createObjects(ResultSet) List<T>
ClientDAO
deleteByName(String) Boolean
findByName(String) Client
OrderDAO
findById(int, Invoice) Order
make(Client, Product, Double) boolean
ProductDAO
findByName(String) Product
findByName(Product) Product
deleteByName(String) Boolean
InvoiceDAO
findByOrder(Order) List<Invoice>

This method came in handy while creating the data abstraction layer in this application.

As it can be seen in the figure on the right, the data abstraction classes for the model's does not contain many methods, because most of them are being used from it's super class (they inherit the AbstractDAO class).

The real use of the reflectional technique appears when the software transforms a given object to data that we can use to create our queries. These queries are then sent to the database, to get/ update/ delete data by following the given command. For each select statement that is send, it is needed to process the response from the database.

This is realised by getting the declared fields of the object and afterwards calling their getters in case if the value is needed. In case if the value should be set, then the setters are called.
As an example, in case of the update, the getters are called for the fields, that give back the value if it's not a Model (Product, Client, Order or Invoice), and if it's a model, than the ID is given back, because we need to save the relation in the database. It's not possible to save an object.
In case of the select, we get back a list of rows. For each row, it's necessary to create an object. In case of simple objects, it's easy to create, because we just simply get the value and give it to the setter. On the other hand, if there's a relationship, then we get the ID of the child/ parent. To get the object with that ID, another select query is called with the findByID method for that specific model.

The whole project is made up by using layered architecture. Basically, it structures the software part into four main categories: presentation, application, domain and infrastructure.

- **The presentation layer** contains all of the classes responsible for presenting the UI to the end-user or sending the response back to the client. In our case it's used to generate the PDFs/ Reports, etc. and to give commands from the user to the application.
- **The application layer** contains all the logic that is required by the application to meet its functional requirements and, at the same time, is not a part of the domain rules. In most systems that I've worked with, the application layer consisted of services orchestrating the domain objects to fulfill a use case scenario.
- **The domain layer** represents the underlying domain, mostly consisting of domain entities and, in some cases, services. Business rules, like invariants and algorithms, should all stay in this layer.
- **The infrastructure layer (also known as the persistence layer)** contains all the classes responsible for doing the technical stuff, like persisting the data in the database, like DAOs, repositories, or whatever else you're using.

The benefits of a layered architecture consists of simplicity (easy to understand the code), consistent across different projects also and guarantees the seperation of concerns. Easy to decide which part of the code should be implemented in a specific package.

Packages

Business Level Logic

Contains classes that are specific to the business, more exactly 4 classes, each specific for the models that appear in the project (Client, Product, Order, Invoice). Each class contains methods that are at the same time, also specific only for the business logic, like making an order, finding a product/ client, etc. and they call the same method from the data abstraction layer, that is more technical.

Connection

Realizes the connection between the database and the application. Makes it easier to handle each SQL query and at the same time by also doing it safely (Opening and closing each connection).

Data Abstraction Layer

This is the more technical part of the application. Here appears every object creation mostly and the statement creation that will be executed later on. So we can say that this is the brain of the current project.

Model

Contains the models that are used throughout the whole application. The classes mostly contain the fields that also appear in the DB and the getters/ setters. Simple, yet the most essential part of the application.

Presentation

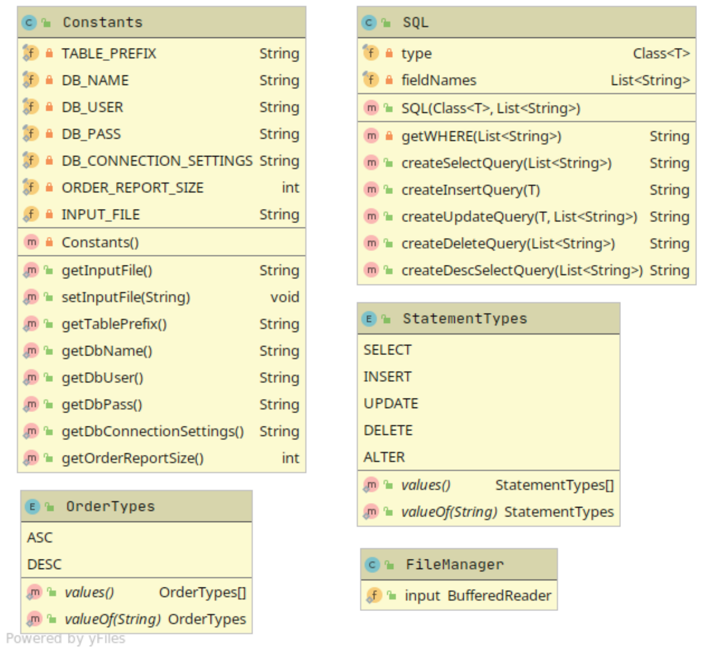
Contains the input/ output processing from/ for the end-user. Parses the input file to see what commands should be ran. And after those commands were ran, creates a new PDF and according to the results, makes a table/ single line output file, that notifies the end-user about a model/ action.

Start

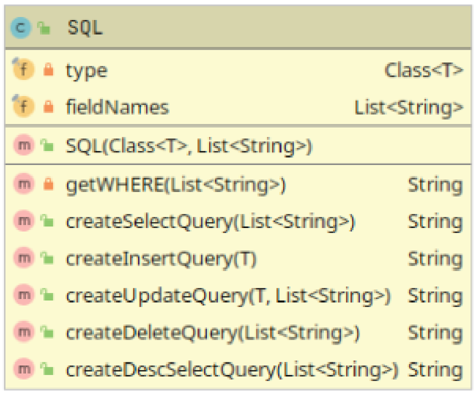
Contains only the main class of the program, that is used to start the application.

Util

Most of the time, classes with static functions can be found in this package or Enum classes that help the other part



Constants
TABLE_PREFIX String
DB_NAME String
DB_USER String
DB_PASS String
DB_CONNECTION_SETTINGS String
ORDER_REPORT_SIZE int
INPUT_FILE String
Constants()
getInputFile() String
setInputFile(String) void
getTablePrefix() String
getDbName() String
getDbUser() String
getDbPass() String
getDbConnectionSettings() String
getOrderReportSize() int
SQL
type Class<T>
fieldNames List<String>
SQL(Class<T>, List<String>)
getWHERE(List<String>) String
createSelectQuery(List<String>) String
createInsertQuery(T) String
createUpdateQuery(T, List<String>) String
createDeleteQuery(List<String>) String
createDescSelectQuery(List<String>) String
StatementTypes
SELECT
INSERT
UPDATE
DELETE
ALTER
values() StatementTypes[]
valueOf(String) StatementTypes
OrderTypes
ASC
DESC
values() OrderTypes[]
valueOf(String) OrderTypes
FileManager
input BufferedReader
Powered by yFiles



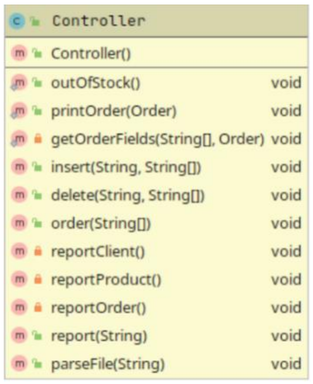
SQL
type Class<T>
fieldNames List<String>
SQL(Class<T>, List<String>)
getWHERE(List<String>) String
createSelectQuery(List<String>) String
createInsertQuery(T) String
createUpdateQuery(T, List<String>) String
createDeleteQuery(List<String>) String
createDescSelectQuery(List<String>) String

of the program to be more readable or to be more easily customizable later.

The most noticeable class is the Constants one, that is used in order to make constant variables to not appear in random positions in random classes. By using a class for this, it's a lot easier to customize some parts of the program as example the connection that is made to the Database.

The SQL Class is used to create the queries as a String which are then executed.

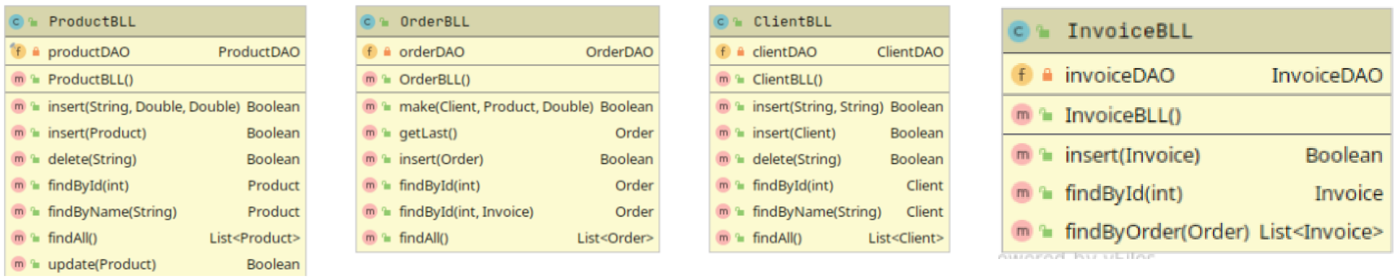
The others are just used to make the code less long and easier to read.



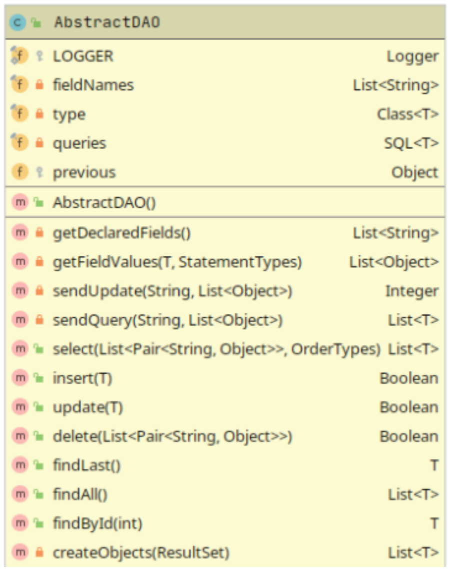
Implementation

The program starts with the **Controller** class, that parses the input file. The parseFile() is first called, which then checks for the main command (insert, delete, order, report) and calles a specific function for that command (insert(), delete(), report()).
Those smaller functions yet again check for the specific model that was given, or make the order. And call the functions from the Business Logic Level Classes.

The next classes that are touched are the BLL classes. They simply make a call to tha Data Abstraction Layers, that do the technical part. The BLL classes function's are used to visualize the logic of the given business and make it clear.



Next up are the DAOs, the core of the application. The **AbstractDAO** class implements most of the work, which are then re-used in the ModelDAOs.



AbstractDAO	
LOGGER	Logger
fieldNames	List<String>
type	Class<T>
queries	SQL<T>
previous	Object
AbstractDAO()	
getDeclaredFields()	List<String>
getFieldValues(T, StatementTypes)	List<Object>
sendUpdate(String, List<Object>)	Integer
sendQuery(String, List<Object>)	List<T>
select(List<Pair<String, Object>>, OrderTypes)	List<T>
insert(T)	Boolean
update(T)	Boolean
delete(List<Pair<String, Object>>)	Boolean
findLast()	T
findAll()	List<T>
findById(int)	T
createObjects(ResultSet)	List<T>

The most used method is the **select()** one, that gets as parameters a List of Pairs where each Pair consists of a String and an Object. The string represents the attribute name, that should be searched for and the Object represents the value of that attribute. This list is used in the creation of the SQL Statement. By using this technique, it is easy to make a query that may contain an infinite number of filters/ references. There is also an OrderTypes paramater that is an Enum class, which tells us the order by the creation date (ordered by the ID field).
Similarly to this function, the **delete**() also uses the same technique to find the elements.

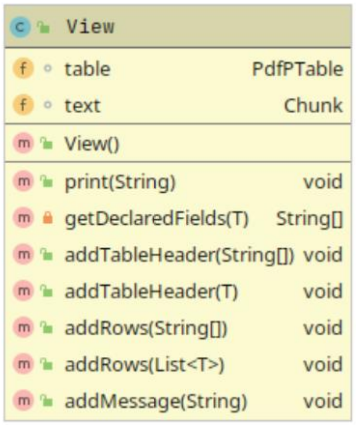
The **insert()** and **update()** method uses an object as parameter, which gets all of it's fields and their values by using reflection, to insert or to find the element and update them.

The **getDeclaredFields()** method is used to prevent duplicate code. This method gets the names of the fields from the current type, by ignoring the ones that start with *x_* . This was added only for the Order model, because of the one-to-many relationship, that shouldn't appear in the Order table, but is needed in the application.

Each one of these methods uses a static function from the **SQL** class that creates the SQL statement by using either objects and reflection, either List of Pairs.

After we get the SQL Statement, there is need for the **ConnectionFactory** class, that establishes the connection to the Database and after a statement is executed, it safely closes everything that is needed to be closed.

Lastly we need to convert the result set to a List of Objects. For this, the **createObjects** method is used. Due to the fact that we don't know what type of object we should create, we create a new one using the current type with reflection. For each field we call the method that uses the setter with the specific type of parameter (Integer, Double or String in our case). Then the logic behind saving that value is implemented in the models. There are cases when we need to create an Object instead of an Integer (Relationships). There's a tricky situation in the case of the Order, because it has a one-to-many relationship (It may have any number of Invoices). For this we do select every Invoice that is the child of the given order and add them to the Orders list of invoices.



View	
table	PdfPTable
text	Chunk
View()	
print(String)	void
getDeclaredFields(T)	String[]
addTableHeader(String[])	void
addTableHeader(T)	void
addRows(String[])	void
addRows(List<T>)	void
addMessage(String)	void

From these we simply return the values back to the Controller, which finally calls the methods from the **View**, which gets the field names and values from the object and prints them into a PDF if possible. The values are added to the table or to the text. And in the print() method, the file is created, those values are added to that file and finally the file is saved/ closed.

Results

As results, the end-user gets a simple application where he/ she can control the products of a business and the orders of any numbers of customers.

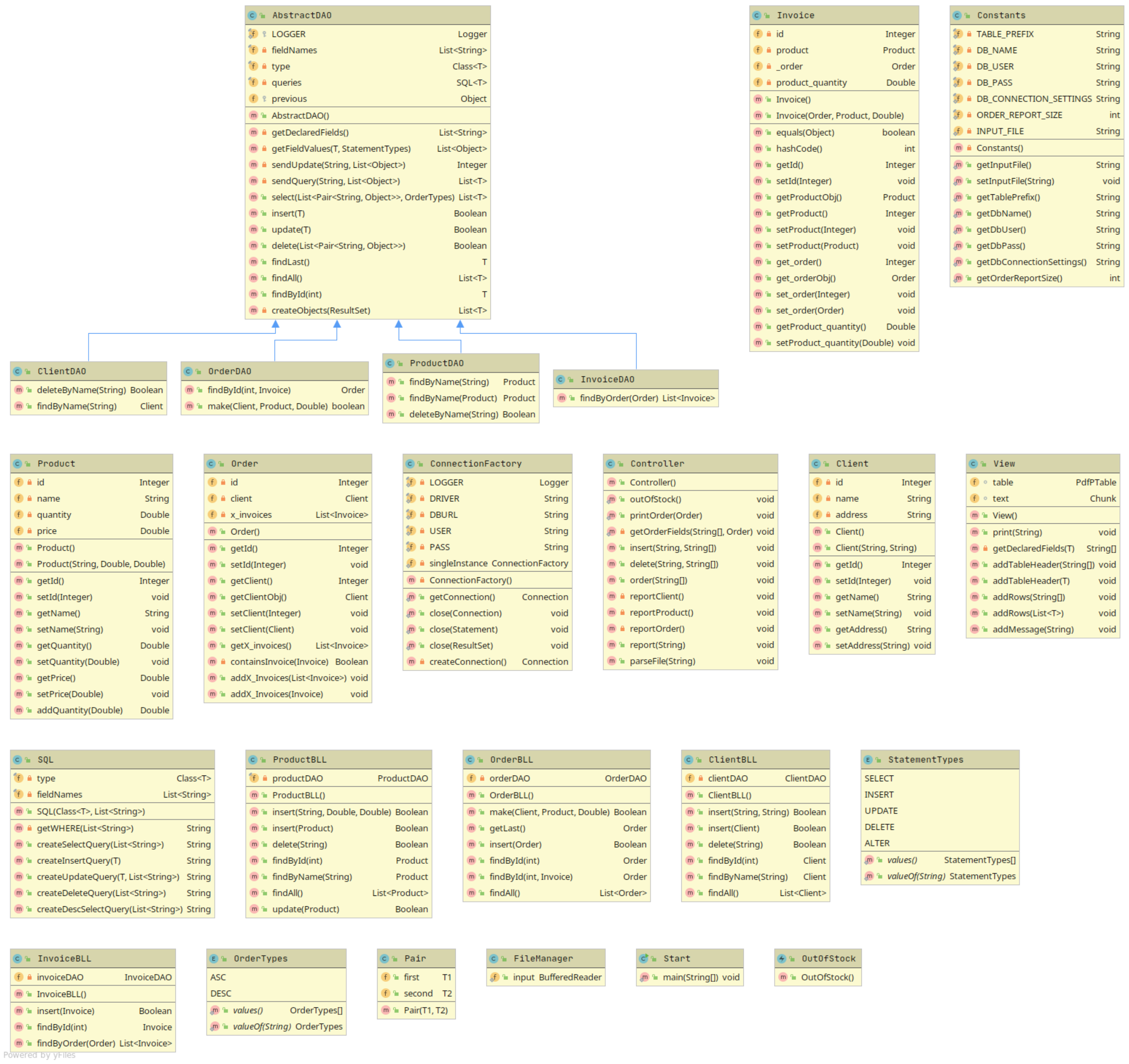
Conclusions

I have learnt a very useful technique that could come in handy many times in object-oriented programming. By using reflection, it's easy to prevent duplicated code fragments and make abstract classes that can be later on inherited. The only drawbacks of this method is that it is performing slower than normal non-reflective codes. In my eyes, the biggest drawback would be the exposure of the private/ protected fields/ methods, that can be used from any part of the code.

Further developments could be made at the commands part, to make the end-user to be able to make reports on specific clients also. Add a way for the end-user to filter the data.

Bibliography

- https://www.geeksforgeeks.org/blockingqueue-interface-in-java/
- https://dzone.com/articles/layered-architecture-is-good


AbstractDAO
LOGGER Logger
fieldNames List<String>
type Class<T>
queries SQL<T>
previous Object
AbstractDAO()
getDeclaredFields() List<String>
getFieldValues(T, StatementTypes) List<Object>
sendUpdate(String, List<Object>) Integer
sendQuery(String, List<Object>) List<T>
select(List<Pair<String, Object>>, OrderTypes) List<T>
insert(T) Boolean
update(T) Boolean
delete(List<Pair<String, Object>>) Boolean
findLast() T
findAll() List<T>
findById(int) T
createObjects(ResultSet) List<T>
Invoice
id Integer
product Product
_order Order
product_quantity Double
Invoice()
Invoice(Order, Product, Double)
equals(Object) boolean
hashCode() int
getId() Integer
setId(Integer) void
getProductObj() Product
getProduct() Integer
setProduct(Integer) void
setProduct(Product) void
get_order() Integer
get_orderObj() Order
set_order(Integer) void
set_order(Order) void
getProduct_quantity() Double
setProduct_quantity(Double) void
Constants
TABLE_PREFIX String
DB_NAME String
DB_USER String
DB_PASS String
DB_CONNECTION_SETTINGS String
ORDER_REPORT_SIZE int
INPUT_FILE String
Constants()
getInputFile() String
setInputFile(String) void
getTablePrefix() String
getDbName() String
getDbUser() String
getDbPass() String
getDbConnectionSettings() String
getOrderReportSize() int
ClientDAO
deleteByName(String) Boolean
findByName(String) Client
OrderDAO
findById(int, Invoice) Order
make(Client, Product, Double) boolean
ProductDAO
findByName(String) Product
findByName(Product) Product
deleteByName(String) Boolean
InvoiceDAO
findByOrder(Order) List<Invoice>
Product
id Integer
name String
quantity Double
price Double
Product()
Product(String, Double, Double)
getId() Integer
setId(Integer) void
getName() String
setName(String) void
getQuantity() Double
setQuantity(Double) void
getPrice() Double
setPrice(Double) void
addQuantity(Double) Double
Order
id Integer
client Client
x_invoices List<Invoice>
Order()
getId() Integer
setId(Integer) void
getClient() Integer
getClientObj() Client
setClient(Integer) void
setClient(Client) void
getX_invoices() List<Invoice>
containsInvoice(Invoice) Boolean
addX_Invoices(List<Invoice>) void
addX_Invoices(Invoice) void
ConnectionFactory
LOGGER Logger
DRIVER String
DBURL String
USER String
PASS String
singleInstance ConnectionFactory
ConnectionFactory()
getConnection() Connection
close(Connection) void
close(Statement) void
close(ResultSet) void
createConnection() Connection
Controller
Controller()
outOfStock() void
printOrder(Order) void
getOrderFields(String[], Order) void
insert(String, String[]) void
delete(String, String[]) void
order(String[]) void
reportClient() void
reportProduct() void
reportOrder() void
report(String) void
parseFile(String) void
Client
id Integer
name String
address String
Client()
Client(String, String)
getId() Integer
setId(Integer) void
getName() String
setName(String) void
getAddress() String
setAddress(String) void
View
table PdfPTable
text Chunk
View()
print(String) void
getDeclaredFields(T) String[]
addTableHeader(String[]) void
addTableHeader(T) void
addRows(String[]) void
addRows(List<T>) void
addMessage(String) void
SQL
type Class<T>
fieldNames List<String>
SQL(Class<T>, List<String>)
getWHERE(List<String>) String
createSelectQuery(List<String>) String
createInsertQuery(T) String
createUpdateQuery(T, List<String>) String
createDeleteQuery(List<String>) String
createDescSelectQuery(List<String>) String
ProductBLL
productDAO ProductDAO
ProductBLL()
insert(String, Double, Double) Boolean
insert(Product) Boolean
delete(String) Boolean
findById(int) Product
findByName(String) Product
findAll() List<Product>
update(Product) Boolean
OrderBLL
orderDAO OrderDAO
OrderBLL()
make(Client, Product, Double) Boolean
getLast() Order
insert(Order) Boolean
findById(int) Order
findById(int, Invoice) Order
findAll() List<Order>
ClientBLL
clientDAO ClientDAO
ClientBLL()
insert(String, String) Boolean
insert(Client) Boolean
delete(String) Boolean
findById(int) Client
findByName(String) Client
findAll() List<Client>
StatementTypes
SELECT
INSERT
UPDATE
DELETE
ALTER
values() StatementTypes[]
valueOf(String) StatementTypes
InvoiceBLL
invoiceDAO InvoiceDAO
InvoiceBLL()
insert(Invoice) Boolean
findById(int) Invoice
findByOrder(Order) List<Invoice>
OrderTypes
ASC
DESC
values() OrderTypes[]
valueOf(String) OrderTypes
Pair
first T1
second T2
Pair(T1, T2)
FileManager
input BufferedReader
Start
main(String[]) void
OutOfStock
OutOfStock()
Powered by yFiles