UCN, UNIVERSITY COLLEGE OF NORTHERN DENMARK

IT-programme

AP Degree in Computer Science

Class: dmaj0920

Title: Workshop – Quality & Testing

Participants:

Mihail Gerginov

Teo Stanic

Antonio Kovacevic

Marko Veljkovic

Lucas Inaschwili-Hongre

Supervisors:

Søren Krarup Olesen

István Knoll

Repository link: https://github.com/TheRandomTroll/ucn-2-ws-persistence

Contents

Introduction

The Persistence workshop is a part of the second semester curriculum for the Computer Science degree at UCN. We were tasked with designing a relational model based on the domain model, writing the SQL scripts for creation of the database and insertion of data. We also developed the system using the DAO pattern to implement persistence and wrote the report.

Our main assignment was to create and develop parts of a computer system for the company named Western Style Ltd. This assignment was provided to us as an opportunity to get us familiar with the system in Java using a relational database.

The domain model given to us was not complete, so we made the adjustments necessary according to design patterns.

While reading the information about the Western Style Ltd. we have learned about the details needed later such as the Club discounts, and free delivery if a private person order is over the selected number. We have also been informed with the companies storing system and their need for being able to track the supplies and products, along with the stock resupply. The company is also considering an expansion, so we kept that in mind.

We were also provided with the information about what kind of IT systems they use, and how often they do the transaction back-ups.

Use cases and diagrams

Starting example of the domain model has been provided but it was not fully completed and correct, so we finished it (Fig. 1), before starting to work on the other diagrams. While working on the use cases we firstly visualized the relations, stuck to the real life as much as possible so that we could manage to make the code as efficient as possible. We created the System Sequence diagram (Fig. 3) which shows us the relations between the user, customer and the system, the Sequence diagram in which we also included the subclasses and their interactions (Fig. 4), and the Design class diagram, for which we used the domain model and the sequence diagram to design classes and packages (Appendix 1). Everything was discussed with our supervisor. For the sake of clarity, these diagrams are also located in the Eclipse project, under the *docs* folder.

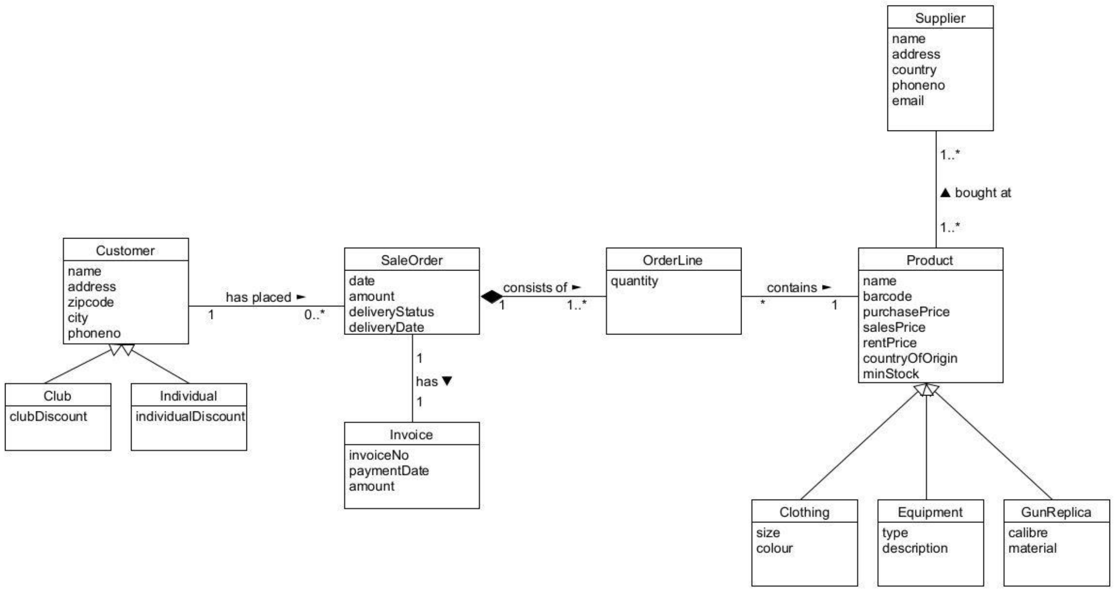

Fig. 1. Domain model

Use Case name	**Order Processing**	
Actors	Customer	
Pre-conditions	Customer has selected desired products; Products have been selected and checked if in stock	
Post-conditions	Payment has been confirmed; Items have been dispatched	
Frequency	Approx. 50 orders/day	
Flow of events	**Actor**	**System**
	1. Customer has selected a product from the catalogue.	2. Checks if the product is in stock.

		3. Notifies the customer that they may proceed to checkout.
	The customer repeats steps 1-3 until he is satisfied with the number of products ordered.	
	4. The customer proceeds to checkout.	5. Displays the order so the user can verify that the ordered products are correct.
		6. Prompts the user to confirm the order.
	7. The customer confirms the order.	8. Registers the order and sends an e-mail invoice to the user.
		9. The system is reset and prepared for a new order.
Alternative flows	**6a. The user decides to cancel the order.**	
		1. The system returns the user to the store, keeping his cart in the state it was before proceeding to checkout.

Fig. 2 Fully dressed use case – Order processing

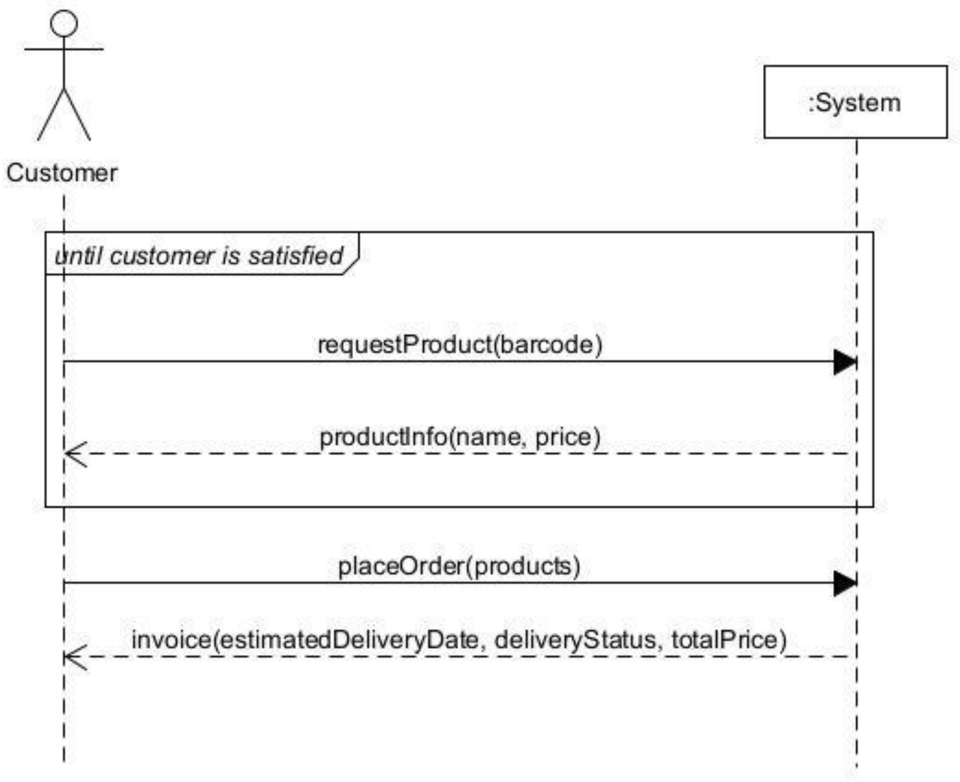
Customer
:System
until customer is satisfied
requestProduct(barcode)
productInfo(name, price)
placeOrder(products)
invoice(estimatedDeliveryDate, deliveryStatus, totalPrice)

Fig. 3. System sequence diagram – Order processing

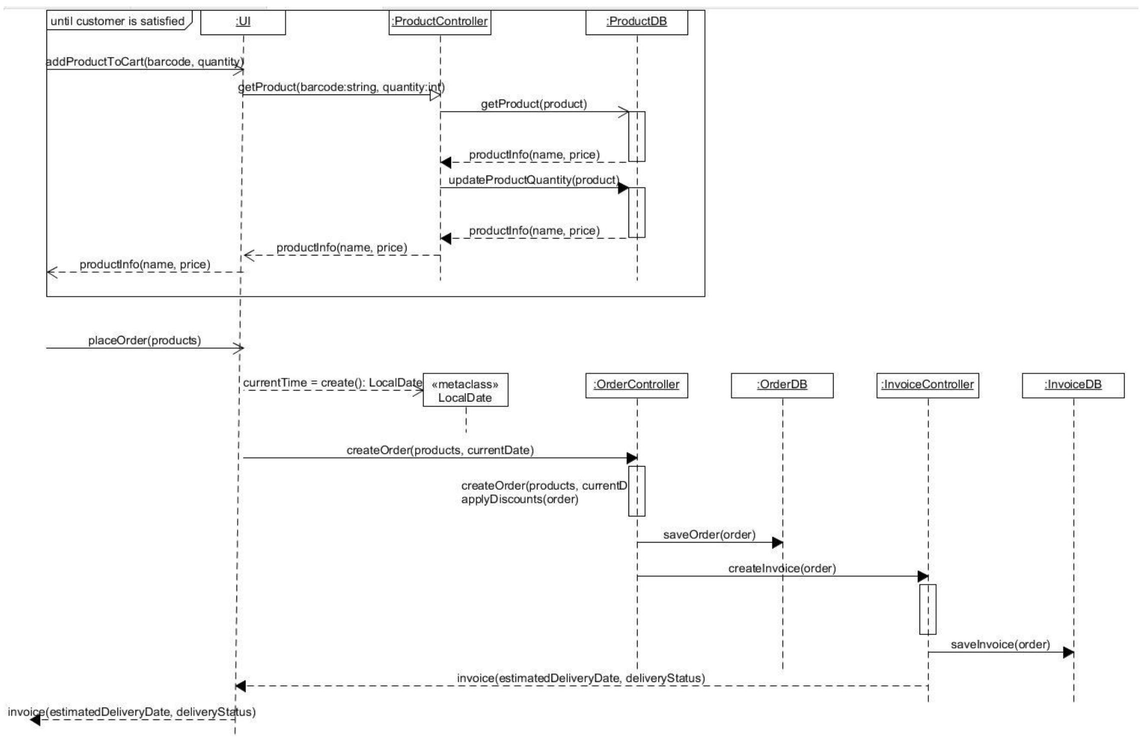
until customer is satisfied
:UI
:ProductController
:ProductDB
addProductToCart(barcode, quantity)
getProduct(barcode:string, quantity:int)
getProduct(product)
productInfo(name, price)
updateProductQuantity(product)
productInfo(name, price)
productInfo(name, price)
productInfo(name, price)
placeOrder(products)
currentTime = create(): LocalDate
«metaclass» LocalDate
:OrderController
:OrderDB
:InvoiceController
:InvoiceDB
createOrder(products, currentDate)
createOrder(products, currentD
applyDiscounts(order)
saveOrder(order)
createInvoice(order)
saveInvoice(order)
invoice(estimatedDeliveryDate, deliveryStatus)
invoice(estimatedDeliveryDate, deliveryStatus)

Fig. 4. Sequence diagram – Order processing

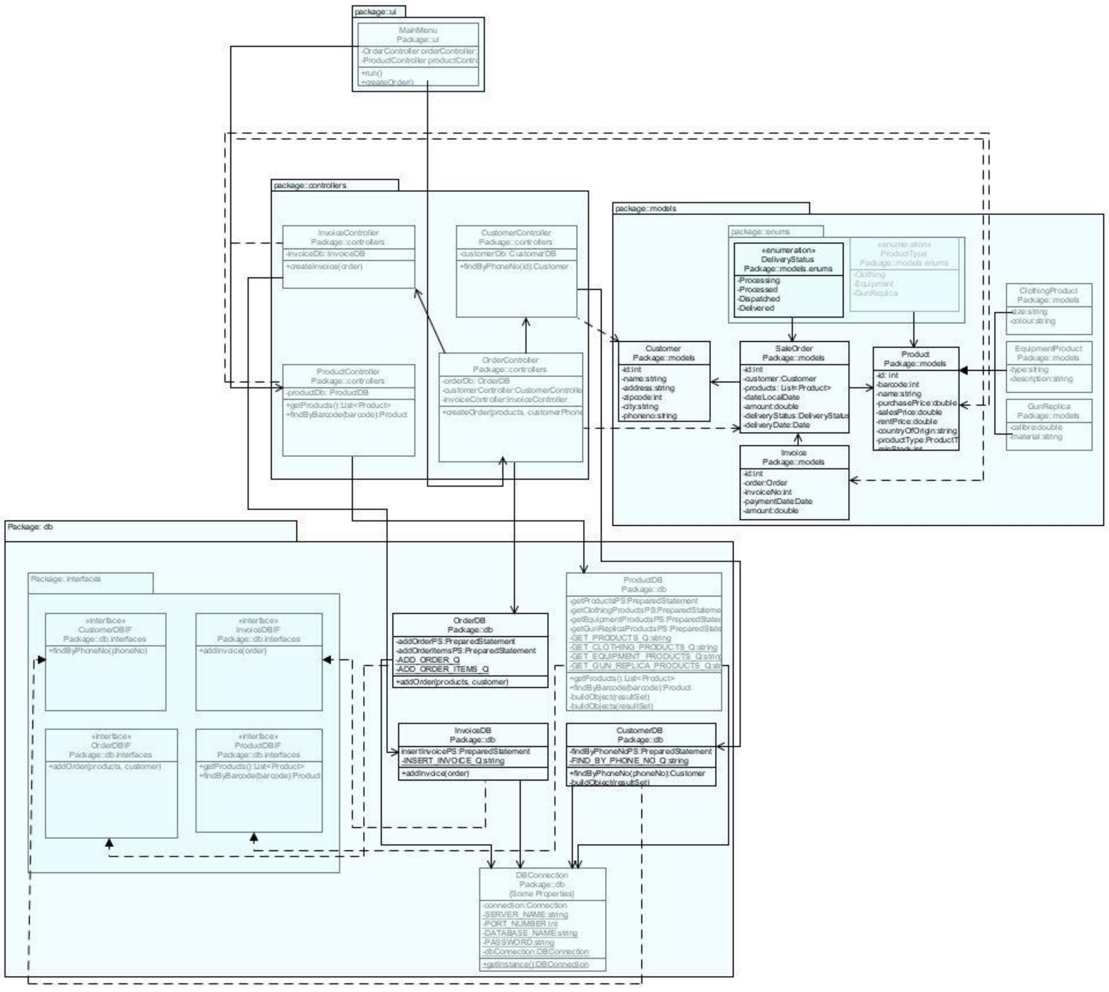

Fig. 5. Design class diagram – Order processing

Programming

Creating and designing the SQL database

The technology of choice for keeping the application persistent is the **Microsoft SQL Server**. The language for writing queries for said technology is **SQL**. The database is located at **hildur.ucn.dk** under the name ***dmaj0920_1086341***. The SQL queries are in the Eclipse project, under the *sql* folder. Fig. 6. displays an example for creating a table for the project.

```sql
CREATE TABLE ProductTypes(
  Id INT IDENTITY(1,1) PRIMARY KEY,
  Type VARCHAR(100)
);
```

```sql
 CREATE TABLE Products(
  Id INT IDENTITY(1,1) PRIMARY KEY,
  Barcode INT,
  Name VARCHAR(100),
            PurchasePrice FLOAT,
            SalesPrice FLOAT,
            RentPrice FLOAT,
            CountryOfOrigin VARCHAR(100),
            MinStock INT,
            ProductTypeId INT,
            FOREIGN KEY (ProductTypeId) REFERENCES ProductTypes(Id)
    );
```

Fig. 6. SQL code for creating a table called *Products*

Since this use case will use classes that inherit from a different class, a workaround for storing the inherited instances in the database must be created. When creating a relational model from a given domain model, there are several methods of converting inheriting objects to tables or parts of tables in the relational model. This project uses the **table each** method, meaning that each child class is mapped to a table and the primary key of the new tables also works as a foreign key for the parent table. Fig. 7. displays the creation of such tables.

```
CREATE TABLE ClothingProducts(
  Id INT PRIMARY KEY,
  Size VARCHAR(3),
  Colour VARCHAR(100)
  FOREIGN KEY (Id) REFERENCES Products(Id)
);
```

Fig. 7. Creating an inheriting table for a child class

Fig. 8. shows the database for the project, which can also be found in the database of the project when browsed with SQL Server Management Studio.

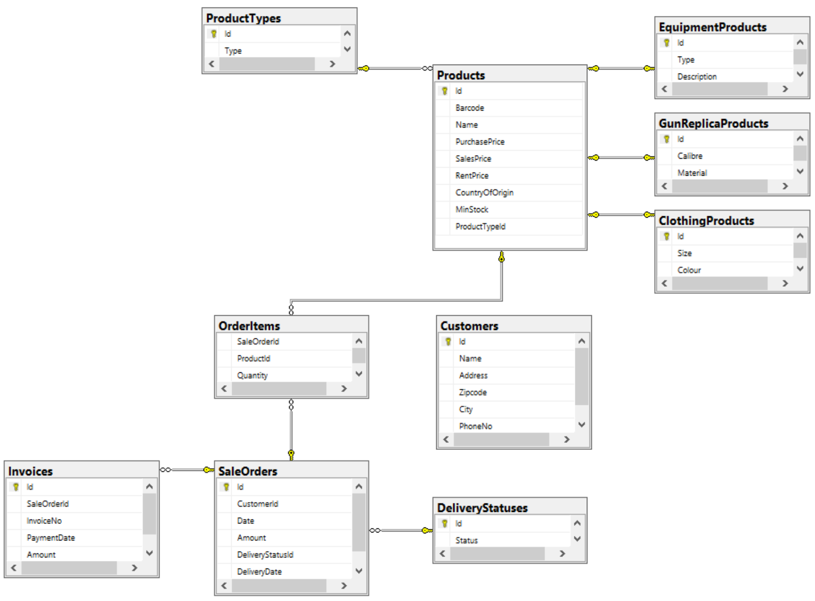

Fig. 8. Database diagram

Developing the solution

For the development of this project, the three-layered architecture was used, meaning that there is a presentation layer for visualizing the data, a business layer for processing commands, and a persistence layer for storing of information. This project also uses the DAO pattern. The package structure of this project is as follows:

- ***models*** - contains the models of the project, which are used both to parse database responses into classes and to use the properties of the classes to create entries in the database. (Persistence layer)
    - ***models.enums*** – contains enumerations which make certain columns from the tables more developer-readable, meaning that, for example – instead of saving a product type as an integer, we can save it as a named value (Clothing, Equipment, etc.)
- ***db*** – contains classes that execute queries, communicating with the database. An especially important class is ***DBConnection***, which handles the main connection with the server. (Persistence layer)
    - ***db.interfaces*** – contains interfaces which are to be implemented by the classes, written in the ***db*** package.
- ***controllers –*** contains classes that communicate with their respective ***db*** class and/or with other controller classes. Their purpose is to execute commands sent by the UI and to return the desired product. (Business layer)
- ***ui –*** contains the user interface, which is to be used by the end user to create requests to and to receive responses from the controller. (Presentation layer)

Comments regarding the code/Code snippets

Section 1. Models

Figures 9. and 10. shows an example of a class being transformed from a model to Java code.

```
public class Product {
        private int id;
        private int barcode;
        private String name;
        private double purchasePrice;
        private double salesPrice;
        private double rentPrice;
        private String countryOfOrigin;
        private int minStock;
        private ProductType productType;

        // Getters and setters
        // toString() method
}
```

Fig. 9. Code for the class models/Product.java

```
public enum ProductType {
```

```
                    CLOTHING, EQUIPMENT, GUN_REPLICA;

                    // Utility methods used when parsing from and to the
database
                    public static Optional<ProductType> valueOf(int value) {
                    return Arrays.stream(values()).filter(x -> x.getValue() ==
value).findFirst();
          }

                    public int getValue() {
                    return ordinal() + 1;
          }
}
```

To maintain the inheritance showed in the domain model, we use the Java keyword **extends** to signify that a class is inheriting another class (Fig. 11).

```
public class ClothingProduct extends Product {
          private String size;
          private String colour;

          // Getters and setters
          // toString() method
          // NB! The toString() method
          // calls its parent method with
          // the super keyword.
}
```

Fig. 11. Java inheritance

Section 2. DBConnection.java

DBConnection.java is responsible for starting and maintaining a connection to the database and acts as a source point for sending requests and receiving responses. The way this connection is done in this project is via the **Java Database Connectivity** (JDBC) application program interface (API). The way JDBC works is by providing it with a hostname, a database name, and credentials for authentication. Below is the realization of the ***DBConnection*** class.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBConnection {
                    private static final String SERVER_NAME = "hildur.ucn.dk";
                    private static final int PORT_NUMBER = 1433;
                    private static final String DATABASE_NAME =
"dmaj0920_1086341";
                    private static final String USERNAME = "dmaj0920_1086341";
          private static final String PASSWORD = "********";
          private Connection = null;

          private static DBConnection dbConnection = null;

          private DBConnection() {
                    String urlString =
String.format("jdbc:sqlserver://%s:%d;databaseName=%s", SERVER_NAME,
PORT_NUMBER, DATABASE_NAME);
                    try {
```

```
				DriverManager.registerDriver(new
com.microsoft.sqlserver.jdbc.SQLServerDriver());
				connection = DriverManager.getConnection(urlString,
USERNAME, PASSWORD);
			} catch (SQLException e) {
				System.out.println("Cannot access database.
Exception: " + e.getMessage());
				return;
			}
		}

		public Connection getConnection() {
			return connection;
		}

		public static DBConnection getInstance() {
			if (dbConnection == null) {
				dbConnection = new DBConnection();
			}

			return dbConnection;
		}
}
```

Fig. 12. DBConnection.java

The reason why the Singleton pattern is used here is because the class meets the following requirements:

- It controls concurrent access to a shared resource.
- Access to the resource will be requested from multiple, disparate parts of the system.
- There can be only one object.

Section 3. Example of a DAO class

The Data Access Object (DAO) pattern is a structural pattern that allows us to isolate the application/business layer from the persistence layer (usually a relational database, but it could be any other persistence mechanism) using an abstract API. (baeldung, The DAO Pattern in Java, 2020)

The functionality of this API is to hide from the application all the complexities involved in performing CRUD operations in the underlying storage mechanism. This allows both layers to evolve separately without knowing anything about each other. (baeldung, The DAO Pattern in Java, 2020)

Figures 13. and 14. show examples for a DAO-pattern interface and class from the source code of the project.

```
public interface CustomerDBIF {
		Customer findByPhoneNo(String phoneNo);
}
```

Fig. 13. CustomerDBIF.java

```
public class CustomerDB implements CustomerDBIF {
			// Private fields and constants
			private static final String FIND_BY_PHONE_NO_Q = "SELECT
Id, Name, Address, Zipcode, City, PhoneNo FROM Customers WHERE PhoneNo =
?";
private final PreparedStatement findByPhoneNoPS;

			public CustomerDB() throws SQLException {
			Connection con =
DBConnection.getInstance().getConnection();
			findByPhoneNoPS = con.prepareStatement(FIND_BY_PHONE_NO_Q);
		}

		@Override
		public Customer findByPhoneNo(String phoneNo) {
			Customer c = null;

			try {
				findByPhoneNoPS.setString(1, phoneNo);
				ResultSet rs = findByPhoneNoPS.executeQuery();
				if (rs.next()) {
					c = buildObject(rs);
				}
			} catch (SQLException e) {
				System.out.println("Error fetching customer from
database: " + e.getMessage());
			}

			return c;
		}

		private Customer buildObject(ResultSet rs) {
			Customer c = new Customer();
			try {
				c.setId(rs.getInt("Id"));
				c.setName(rs.getString("Name"));
				c.setAddress(rs.getString("Address"));
				c.setZipcode(rs.getInt("Zipcode"));
				c.setCity(rs.getString("City"));
				c.setPhoneNo(rs.getString("PhoneNo"));
			} catch (SQLException e) {
				System.out.println("Error parsing customer: " +
e.getMessage());
			}
			return c;
		}
}
```

Fig. 14. CustomerDB.java

Section 4. The controllers

The controllers are straightforward – they accept requests from the UI, transfer the request to the data layer and return a result to it. Fig 15. shows an example of a controller.

```
public class CustomerController {
			private CustomerDB customerDB;

			public CustomerController() {
			try {
```

```
                        this.customerDB = new CustomerDB();
                } catch (SQLException e) {
                        System.out.println("Could not instantiate 
CustomerDB.");
                }
        }

        public Customer findByPhoneNo(String phoneNo) {
                return this.customerDB.findByPhoneNo(phoneNo);
        }
}
```

Fig. 15. CustomerController.java

Section 5. The UI

For the sake of simple presentation, this project uses a text user interface (TUI) to visualize the result of the implementation. The text below shows how the user interface looks now.

```
---- MAIN MENU ----
(1) Create order
Enter your option: 1
Enter the customer's phone number: +4512345678
Enter the product's barcode, or "Checkout" to finish the order: 87612398
Semi-automatic Pistol, Desert Eagle added to order!
Enter the product's barcode, or "Checkout" to finish the order: 16483911
Product with barcode 16483911 not found.
Enter the product's barcode, or "Checkout" to finish the order: 16482911
Men's Shirt added to order!
Enter the product's barcode, or "Checkout" to finish the order: Checkout
Order #1 has been placed!
Estimated delivery date: 2021-03-25Invoice #316645
Customer: {name='Mihail Gerginov', address='Sofiendalsvej 62', zipcode=9000,
city='Aalborg', phoneNo='+4512345678'}
Order date: 2021-03-18
Products:
-----| Product #87612398 |-----
Semi-automatic Pistol, Desert Eagle
Product type: GUN_REPLICA
Price: DKK479.99
Calibre: 0.5
Material: Aluminum

-----| Product #16482911 |-----
Men's Shirt
Product type: CLOTHING
Price: DKK259.99
Size: L
Colour: Beige

=====================
TOTAL: DKK739.98
=====================
```

Since we are using a polymorphic list to store the products in-memory, reflection had to be used to get child-specific methods (models/Invoice.java; toString method). Reflection allows us to inspect or/and modify runtime attributes of classes, interfaces, fields, and methods. This particularly comes in handy when we don't know their names at compile time. Additionally, we can instantiate new objects, invoke methods, and get or set field values using reflection. (baeldung, Guide to Java Reflection, 2020)

```
@Override
public String toString() {
               StringBuilder sb = new StringBuilder();
               sb.append("Invoice #" + invoiceNo + "\n");
               sb.append("Customer: " + order.getCustomer().toString() +
"\n");
               sb.append("Order date: " + order.getDate().toString() +
"\n");
               sb.append("Products: \n");
               for(Product p : order.getProducts()) {
                       Class productClass = p.getClass();
                       if (productClass == ClothingProduct.class) {
                              sb.append(((ClothingProduct)p).toString());
                       }
                       // Analogous for other classes
               }
               sb.append("=====================\n");
               sb.append("TOTAL: DKK" + amount + "\n");
               sb.append("=====================\n");
               return sb.toString();
        }
```

Fig. 16. Using reflection to call child's method in a polymorphic list

Group Process

We were not satisfied with how the 1st semester project went for our group so this time we decided to put more effort into the assignment, as soon as we got the information needed related to the Persistence workshop we have got our hands on the design class diagram and creating the use cases. We were not able to meet up in person, but we were in contact the whole-time using Discord. After wrapping up the system development part we asked our supervisor for his opinion and then based on that improved our system developing part before getting our hands on the code. We started with the code and SQL on the day 3 and as soon as we finished, we started to work on the report.

Conclusion

With what was presented to us as a project the persistence mini project helped us to better understand the usage of the SQL and databases in general, we have also improved our system development skills all while working as a group. Future realization may include:

- System Development
  - Implement more use cases and their respective diagrams;
  - Create tests cases for the given use cases.
- Programming
  - Add unit tests to the project.
  - Create a GUI for the application

References

baeldung. (2020, September 12). *Guide to Java Reflection*. Retrieved from baeldung.com: https://www.baeldung.com/java-reflection

baeldung. (2020, March 21). *The DAO Pattern in Java*. Retrieved from baeldung.com: https://www.baeldung.com/java-dao-pattern