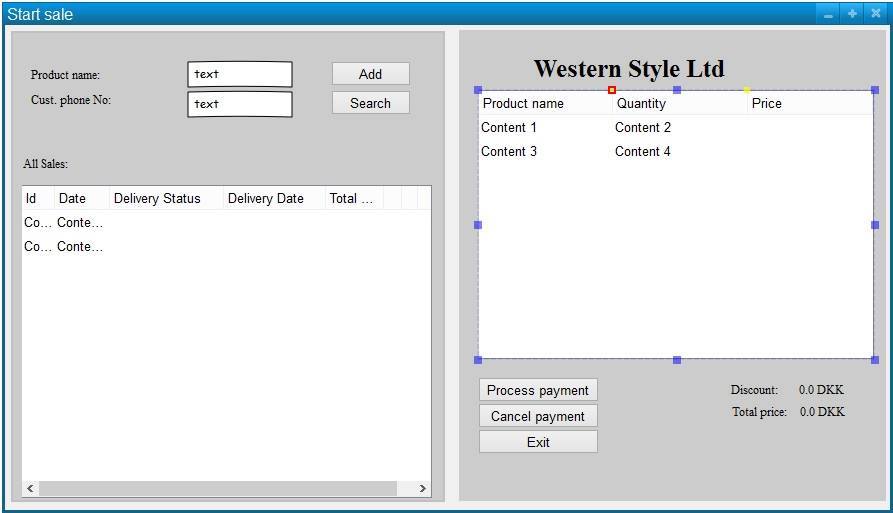
Workshop Documentation

Group 3

Members:

* Ionut Danci-Bumbea
* Janis Indrans
* Leonard Hirja
* Nikita Cimaskevic
* Attila Richard Retezi

Mock Up:



The design follows the principle to keep things in a minimalist way. The interface was created to provide maximum usability and functionality. The design is easy to understand and handle.

Fully Dressed Use Case Diagram comments

The fully dressed use case diagram for the “make sale” use case was created to have a better insight of what actors are participating during the process, how they interact with the system and what inputs the system receive, what outputs the system provides.

The actor in connection with the use case make sale is the Employee. Prerequisite for the use case is that the ordered products are available and are in stock. Post-condition is that the order is created and the related invoice was made. The amount of times the make sale use case is processed is multiple (many) times per day.

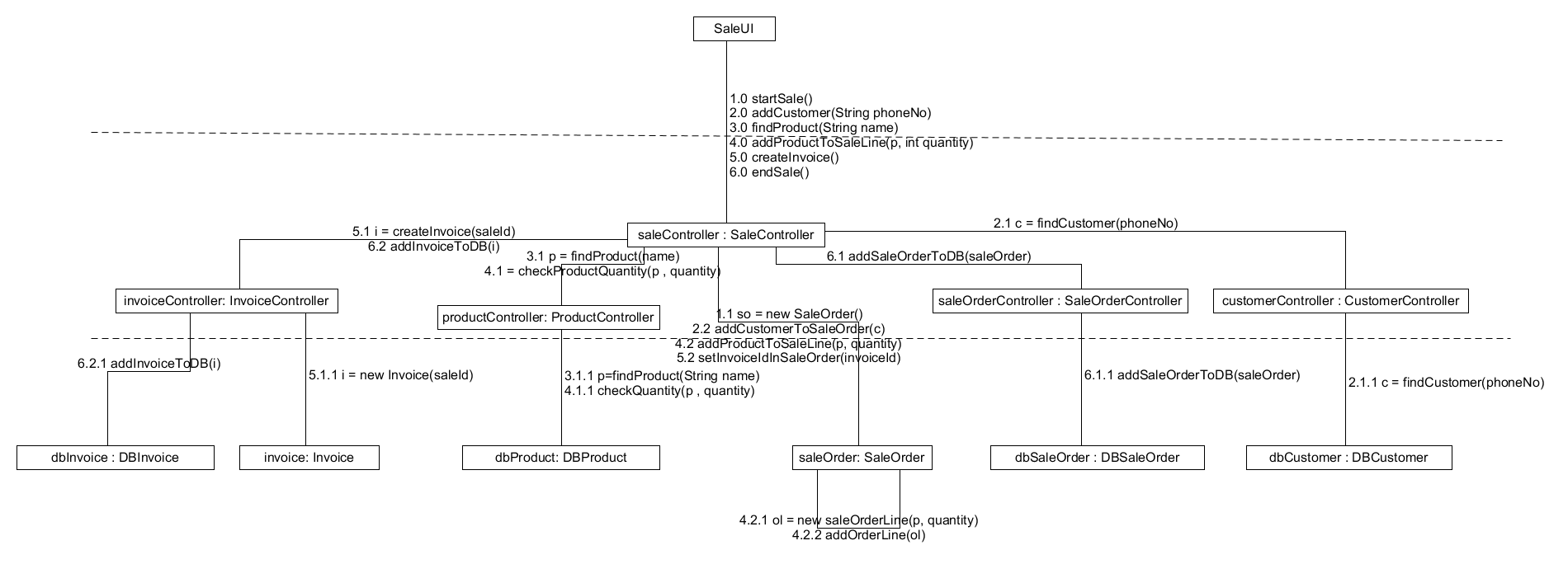
The Main Success Scenario shows the standard procedure in case everything goes according to plan. The Customer makes an order personally, or by phone or by e-mail. The Customer provides the necessary information which the Employee implements in the system. The system processes the order. The Employee registers the customer if he wasn’t in the system before. Customers’ are being searched by their phone number. The Employee starts a new order and checks if the wish products are available (they are in stock). The Employee selects the asked products and the quantity and the system adds them to the order. After all the products are selected, the Employee finalizes the order. The system calculates the discount and delivery fees (if any) and calculates the total price.

In the end, the system completes the order and creates the invoice.

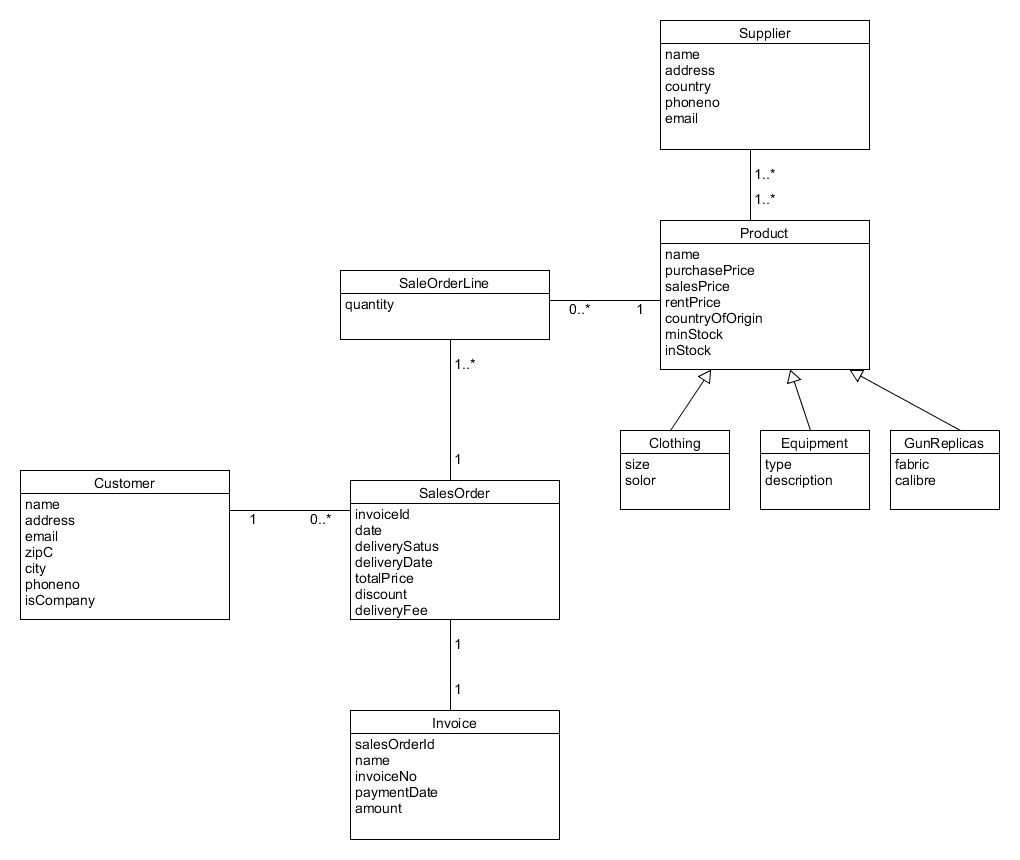
The alternative flow (scenario) is that the item is not in stock (out of stock).

|  |  |  |
| --- | --- | --- |
| **Use case name** | **Make Sale** | |
| **Actors** | Employee | |
| **Pre-condition** | Product are registered in stock | |
| **Post-condition** | The order is created and invoice is made | |
| **Frequency** | \* times per day | |
| **Main success scenario (flow of events)** | **Actor(action)** | **System(response)** |
|
| 1. Customer makes a call and makes an order and specifies information (name,size,type) . Customer provides name and phone number. The Employee implements the information. |  |
|
| 2. Employee registers the customer if it wasn't before or searches for it. | 3. System registers customer by its name if it wasn't in the system before. If was, then returns the customer's information. |
|
| 4. Employee starts a new order | 5. System displays a product list with available products in stock |
|
| 6. Employee chooses desired products. | 7. System register desired products and adds to order |
|
| 8. Employee chooses to complete the order | 9. System calculates the discount and delivery fees (if any). System completes order and makes invoice. |
|
| **Alternative flows** | 6a. Item is not in stock |  |
|  |  |  |
|  |  | |

Interaction Diagram



Domain Model



Domain Model comments

The domain model that was provided in the project description was needed to be modified and added more classes and fields.

Initially a new class, SalesOrderLine was provided. It was a crucial implementation, because the process of the order works as follows:

The selected product is added to the SalesOrderLine where a quantity field is implemented to set the amount of each selected product. Exactly one product can be added to a SalesOrderLine and zero or many SalesOrderLine can contain the same product.

Afterwards, one or many SalesOrderLine can be added to the SalesOrder and we have exactly one SalesOrder per order.

Next, some of the multiplicities given were needed to be changed. The relation between SalesOrder and Invoice looks as follows: A SalesOrder can have only one invoice and an invoice should be connected only to one SalesOrder.

The relation between Supplier and Product also needed to be modified slightly: A Supplier can supply more products and one product can have many suppliers.

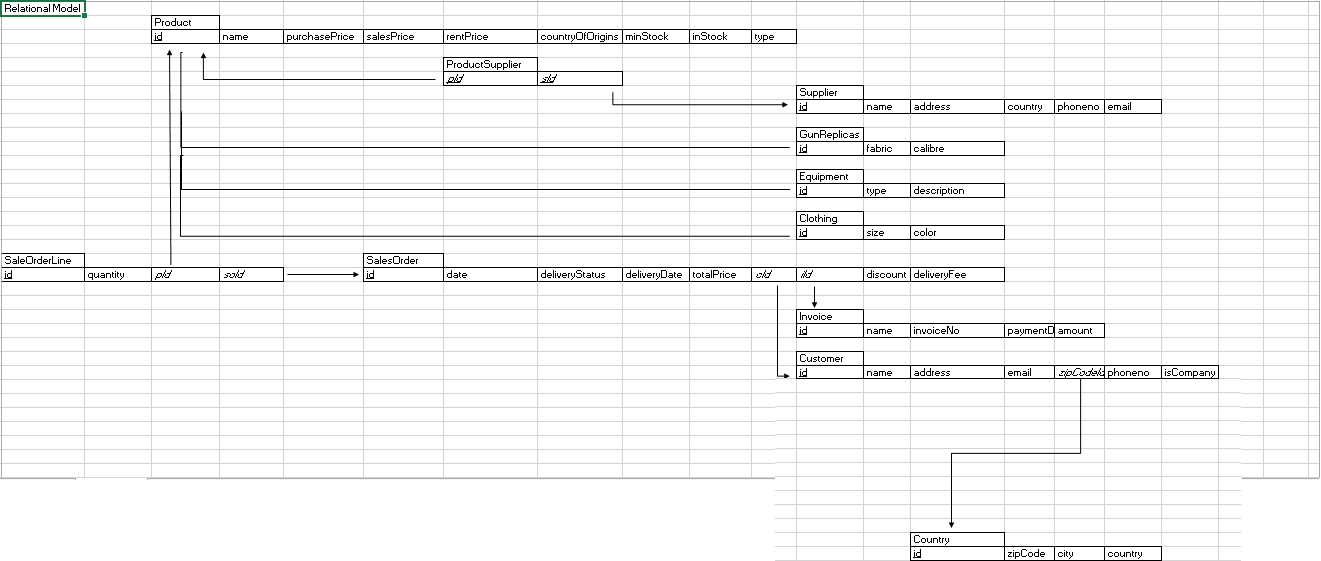
Some additional fields were provided to the classes:

* In the Customer class, an email field was provided, because customers have the possibility to make an order by sending an e-mail. An isCompany field was also provided in order to register if a customer is a private person or company (in this case, mostly clubs).
* In SalesOrder a discount and a deliveryFee attribute was added to store the value of additional fees and discounts for the price calculation methods.
* In the Product class the subclasses of item types (Clothing, Equipment and GunReplicas) were kept and an additional field – inStock – was implemented, which stores the amount of products that are available in the store. Before that point, there was no record of the current amount, only the level of minimum stock (minStock), which is responsible to store a value in order to compare it with the inStock value. If inStock is below minStock, an adequate method calls for an automatic refill for the given product from the given supplier.

Transformation into relational model

The Domain Model needed to be transformed into the relational model in order to have a better understanding of how to implement the information into the SQL-script and queries. The transformation was made “the nice way”, meaning that we put the closely related attributes into the same table and the less related ones into a separate table. The principle should be that all the attributes depend on the primary key and the primary key only. Other close relations should be put into a separate table (normalization). Typical example is the zip code, city, country relation. The customer class contained the field zipC and City, which are closely related. Therefore a new table, Country was created to improve the quality of the relational model.

Between the Supplier and Product tables we have many to many relations, therefore a new table ProductSupplier was created with dummy IDs in order to make the connection between the tables.



Code standard

The code is going to be written according to Oracle Java Code convention as follows:

1. Always put getters and setters for fields.
2. Methods are not so long in order to solve concrete task.
3. All fields on the top of the file.
4. Appropriate comments for most important methods.
5. Give relevant names to classes and methods.