UCN, University College of Northern Denmark
IT-Programme
AP Degree in Computer Science
dmai0919

Mini Project Persistence

Alexandru Stefan Krausz, Sebastian Labuda, Martin Benda, Simeon Plamenov Kolev

27-03-2020

Title page

UCN, University College of Northern Denmark

IT-programme

AP Degree in Computer Science

Class: dmai0919

Participants:

Alexandru Stefan Krausz

Sebastian Labuda

Martin Benda

Simeon Plamenov Kolev

Supervisor: Gianna Belle, Jesper Strandgård Mortensen

Abstract/Resume

In this project, our group was assigned to create a system for company Western Style ltd., which is a retail company for stylish cowboy clothing and accessories. The software solution should be able to create orders, generate invoices and manage customers, products and stocks by using databases. This report presents our work and part of the system we have created.

Repository path: https://kraka.ucn.dk/svn/dmai0919_2Sem_2

Repository number: 47

Database name: dmai0919_1081927

Normal pages/characters: 20 543 characters / 8.55pages

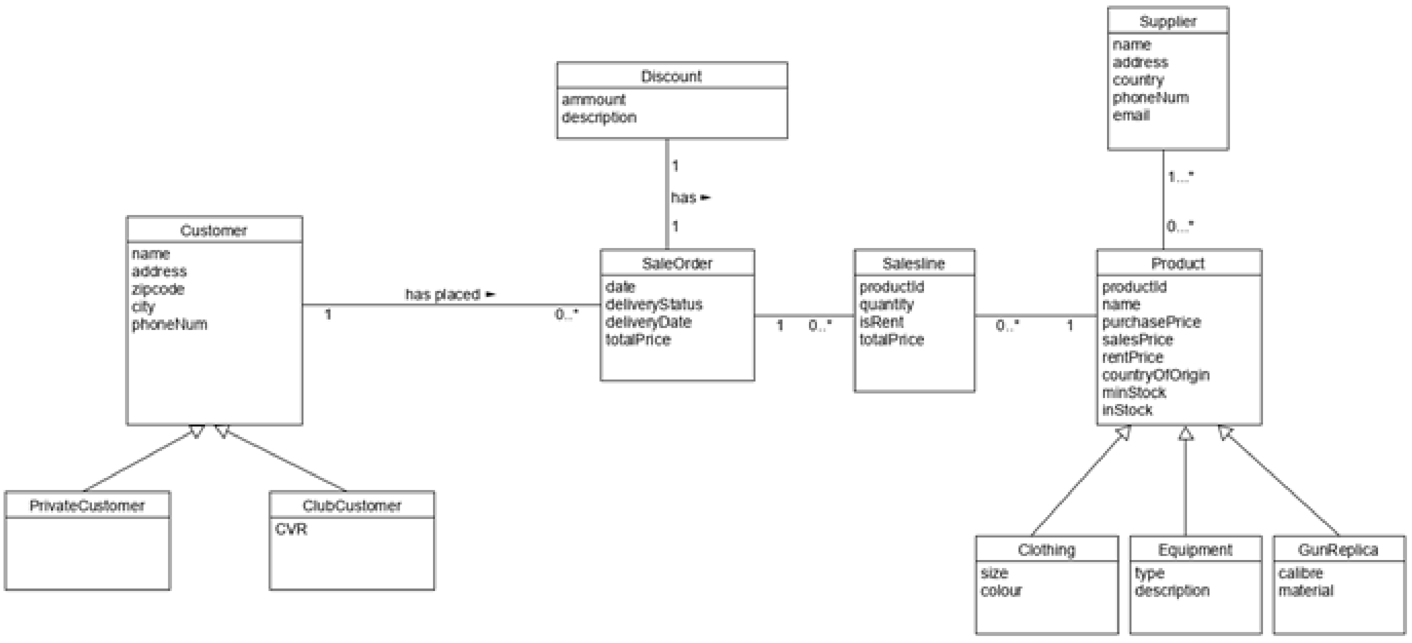
Contents

Abstract/Resume 1
Introduction 3
Domain model 3
Relational model 4
SQL Scripts 5
Use Case 6
Mock-Ups 7
SSD, contract and interaction diagram 7
Design class diagram 10
Testing 10
Test Cases 11
Code Standard 12
Groupwork 12
Conclusion 12
Literature 13
Appendices 13
Appendix 1 – Mockup 13
Appendix 2 - SQL script 13
Appendix 3 19

Introduction

The main reasoning behind this project was to create a brand new system tailored for the company, which is right now using MS Office for all their maintenance of products and orders. Although there can be a lot of modifications and details done, in this case, we were given one week to implement the most important part of this software solution. Most importantly we should set up and work with a database and store all our test data there. We aimed to create a logical structure, which will cover also the future implementation and won't need any changes.

Domain model

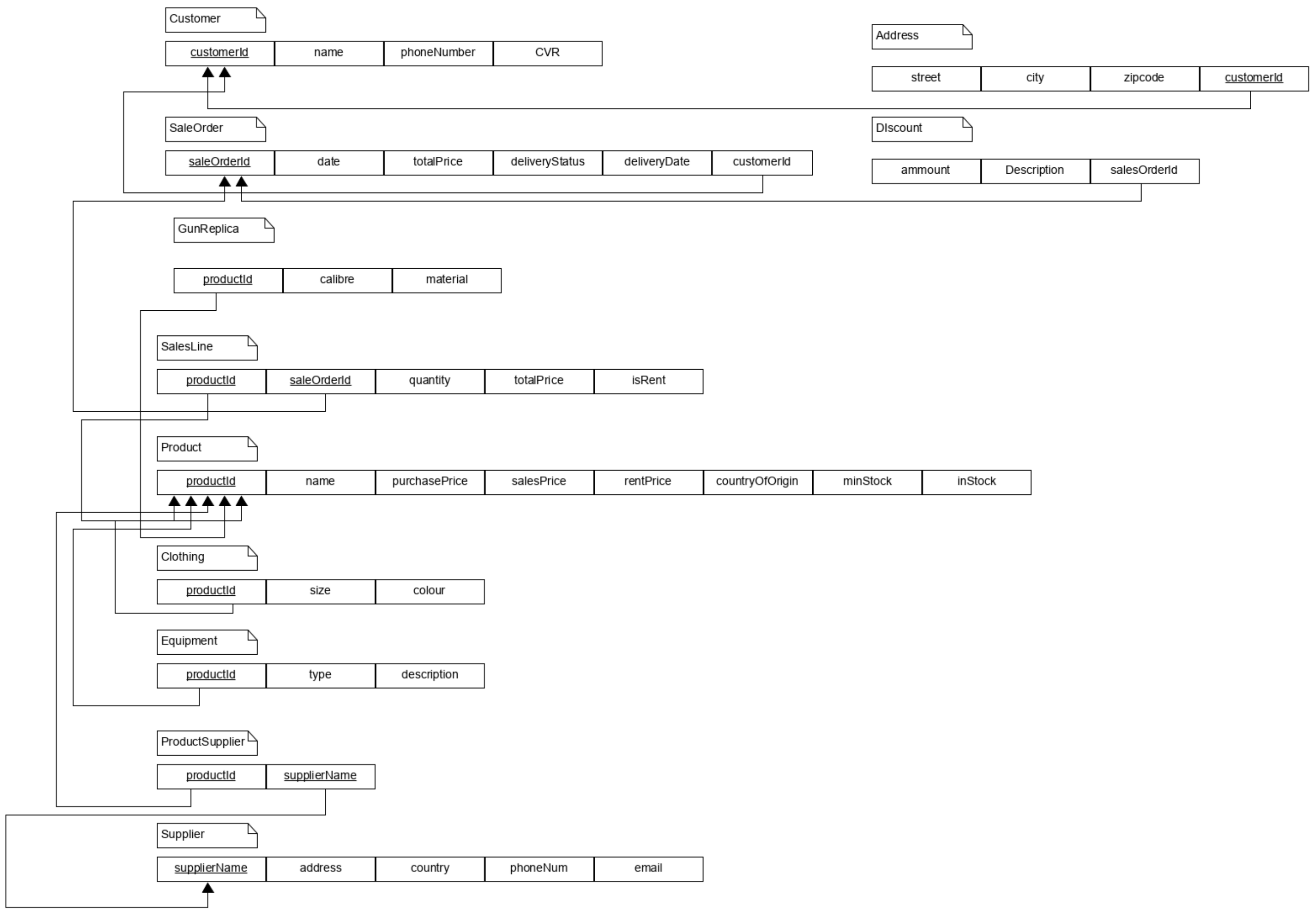


Our first objective was to create a domain model, which is a visual representation of real-life "entities" and relations between them.

Since we received already existing "unfinished" domain model, our main task was to make adjustments in already existing domain model. Our domain model went through several changes during our first three days.

The most discussed class in our domain model was certainly Salesline, since we had to determine its task and meaning in our overall structure. The second discussion consisted, if we should include the middle class between our Supplier and Product class due to many to many relations. At the end, we have come up with the final version of our domain model, as you can see down below.

Relational model

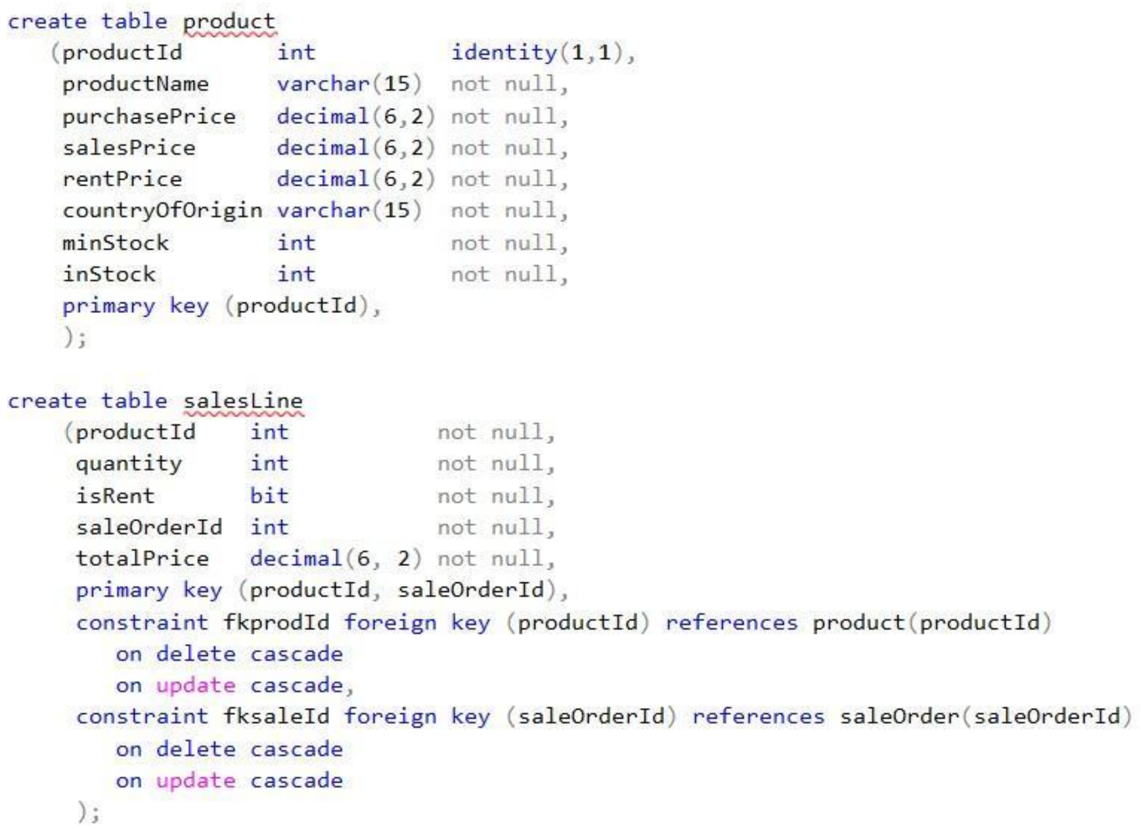


With the domain model finished, we started to work on the relational model. The relational model is created to have an overview of how the database, specifically tables in the database, will look like. In the relational model, you can see foreign keys, which are attributes referencing other attributes in different tables and primary keys (underlined attributes), which uniquely identify that specific table. Basically said each class in the domain model will have its representant in the database.

In our case, we created a separate table for addresses of the customers, which makes the tables more readable. Since there are two types of customers, namely private customers and companies, we decided to use the so-called pull-up tactic. This means that for all the customers we have only one table and each customer also has attribute CVR. Once the CVR is different from null, it means this customer is a company. We decided to use it because subclass private customer has no attributes in the domain model and subclass ClubCustomer has only CVR. Having only one table makes it, in this case, easy to use and it still will be a readable table. The same problem occurred with the product class, but in this case, since there are a lot of attributes in each subclass, we decided to create a separate table for superclass and each subclass. Even though it will be a bit harder to work with those, it will spare space in the database and an enormous amount of nulls. Last more specific case was between classes product and supplier. Since there is many to many relation between them, we had to create a separate table for them called “ProductSupplier”. This will act as a glue between those classes.

We also needed to formally validate our relational model using normal forms. Basically there shouldn’t be any multivalued attribute and each attribute should be dependant only on the primary key. That is why we, for example, created the “ProductSupplier” class. Since each supplier can deliver multiple products and each product can have multiple suppliers, this extra class will ensure that the attributes will be atomic and not multivalued.

SQL Scripts

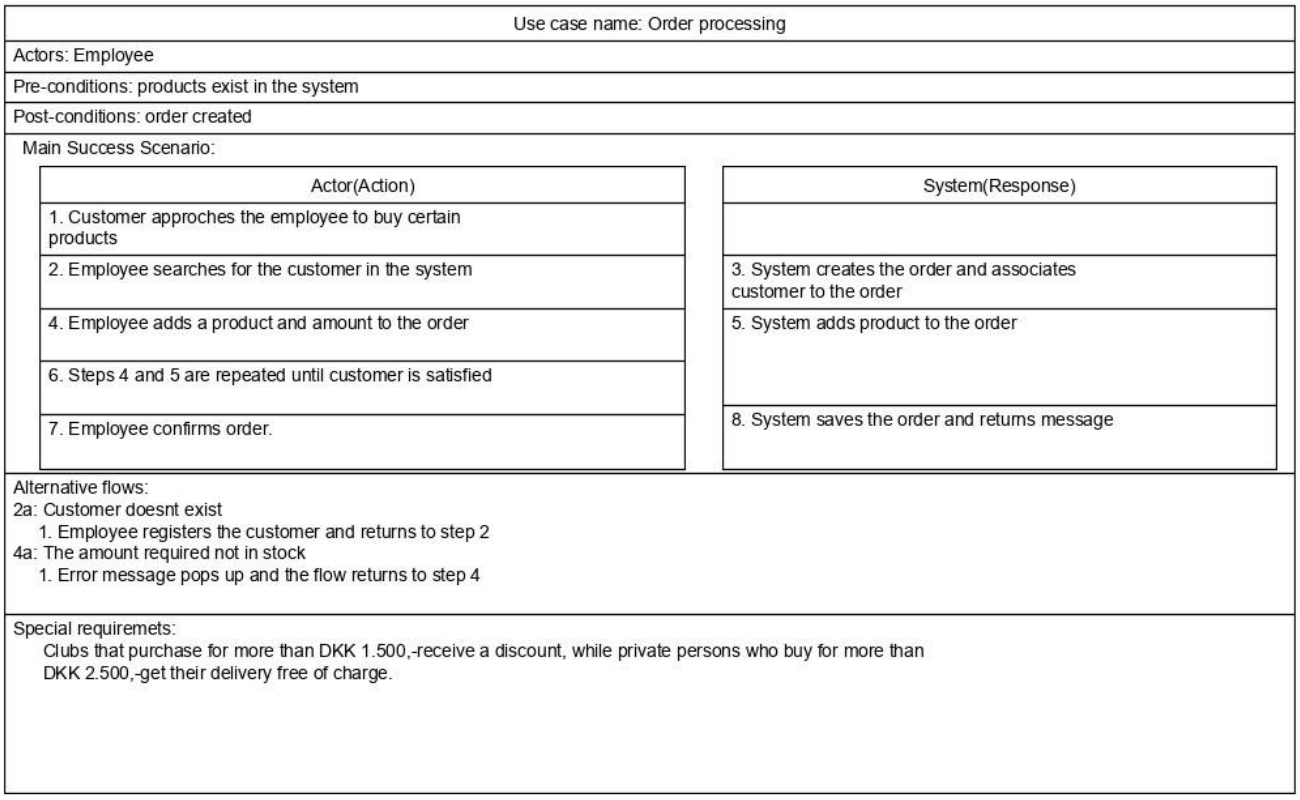


```
create table product
   (productId       int          identity(1,1),
    productName     varchar(15)  not null,
    purchasePrice   decimal(6,2) not null,
    salesPrice      decimal(6,2) not null,
    rentPrice       decimal(6,2) not null,
    countryOfOrigin varchar(15)  not null,
    minStock        int          not null,
    inStock         int          not null,
    primary key (productId),
    );

create table salesLine
    (productId    int            not null,
     quantity     int            not null,
     isRent       bit            not null,
     saleOrderId  int            not null,
     totalPrice   decimal(6, 2) not null,
     primary key (productId, saleOrderId),
     constraint fkprodId foreign key (productId) references product(productId)
        on delete cascade
        on update cascade,
     constraint fksaleId foreign key (saleOrderId) references saleOrder(saleOrderId)
        on delete cascade
        on update cascade
     );
```

Since the relational model is only an overview of tables and attributes in the database, we needed to write the code which will create these tables with all the attributes and relations between them. The script is too long to include it here, so the rest of it is in appendix 2. In the script seen above, there is the creation of 2 tables: product and salesLine. For the productId, we used “identity(1,1)”. This means it will start by 1 and after adding anything to the table it will increment the id by 1, which means the added product will now have id 2. By this, we can ensure unique if for every single product. For the prices we used decimals with 2 decimal places. In the salesLine, the attribute “isRent” is a bit, which will represent boolean logic if it’s true or falls (1 equals true and 0 is false). In both cases we are assigning primary key, in salesLine it has two values. But salesLine has also 2 foreign keys. After there are any changes made in the foreign keys, the same changes will be done in this table thanks to the cascade annotation. This will ensure database consistency. The rest of the script in the appendix is very similar, except the inserting part, where we are inserting test data into our database.

Use Case



Use case name: Order processing

Actors: Employee

Pre-conditions: products exist in the system

Post-conditions: order created

Main Success Scenario:

Actor(Action)	System(Response)
1. Customer approches the employee to buy certain products	
2. Employee searches for the customer in the system	3. System creates the order and associates customer to the order
4. Employee adds a product and amount to the order	5. System adds product to the order
6. Steps 4 and 5 are repeated until customer is satisfied	
7. Employee confirms order.	8. System saves the order and returns message

Alternative flows:
2a: Customer doesnt exist
1. Employee registers the customer and returns to step 2
4a: The amount required not in stock
1. Error message pops up and the flow returns to step 4

Special requiremets:
Clubs that purchase for more than DKK 1.500,-receive a discount, while private persons who buy for more than DKK 2.500,-get their delivery free of charge.

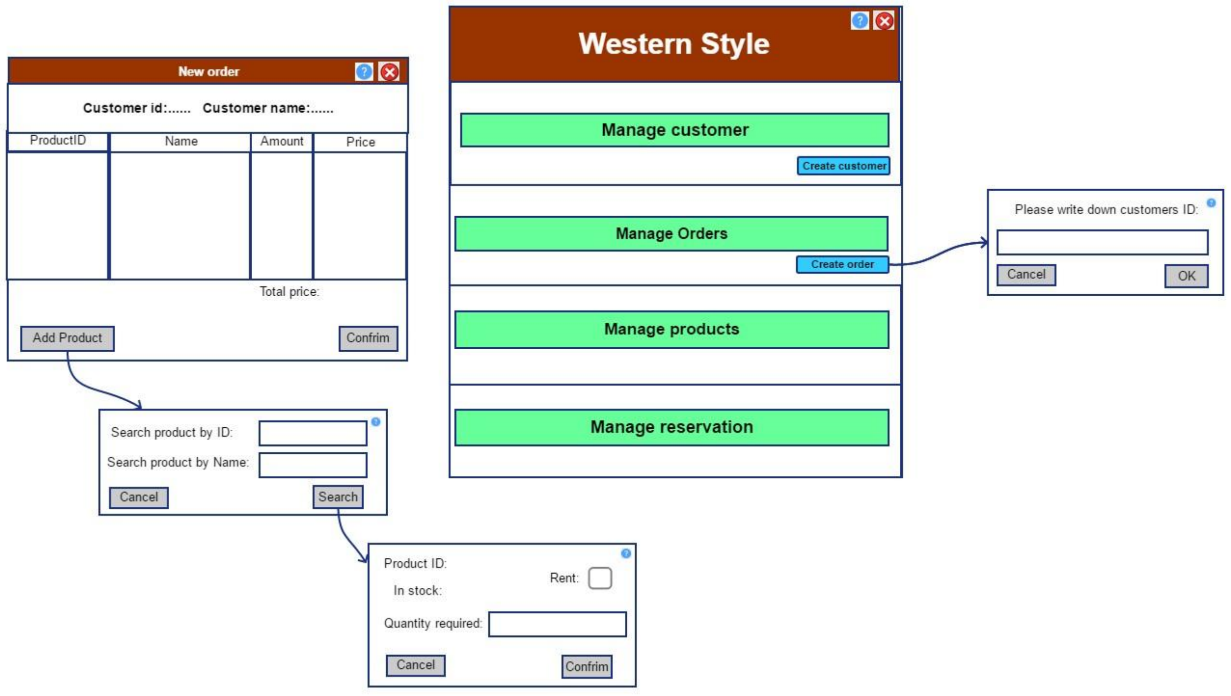
Our primary goal for this project was to implement the most important use case, namely order processing. Use cases, in general, describe which users interact with the system and how they interact with it. This interaction is described in detail, especially in the fully dressed use case. It consists of the main success scenario flow and alternative flows.

Our use case describes a routine order process that an employee will perform in order to sell/rent different products.In it can be followed the main success scenario in which the employee finds the customer in the system after that he adds products to the order until the customer is satisfied.

The use case also includes alternative flows like customer doesn’t yet exist in the system or the amount requested exceeds the current available stock.

As special requirements we included a discount which can and should be applied depending on the customer category and the total price of the order.

Mock-Ups



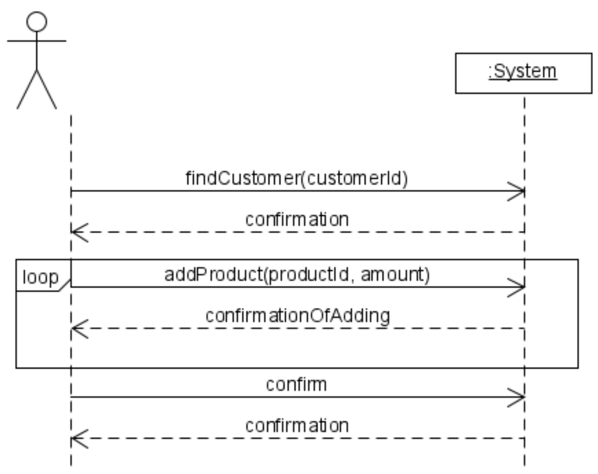
After that, we created mock-ups, which is like the first idea of how our system will look like at the end of development. Even though in the finished version it is mostly different, mock-ups are created to find important aspects which have to be considered, agree on some conventions while creating GUI and during think-aloud testing to discuss with the customer if it is user friendly, if he can find everything he needs etc. In our mock-ups, we focused on creating order, which will be accessed through the main menu. It can be either accessed by clicking on the button “manage order” and from ”manage order” menu go to create order, or user can use the quick button “create order”, which will increase effectiveness since this will be used very often. From there the user is asked for customer ID. After putting the right ID, the user will be in “New order” menu, where they will see customer ID and their name, will be able to add products to the order and see all the products already added. The employee will be able to search for the product either by its ID or name. Ones the product was found, the amount of the products in-stock will be displayed and the user will write the desired amount. In case the product should be rent and not sold, all that will be needed is to mark “Rent” tickbox. Every single part of the user interface will have a small question mark in the top right corner, which will show information about that specific menu.

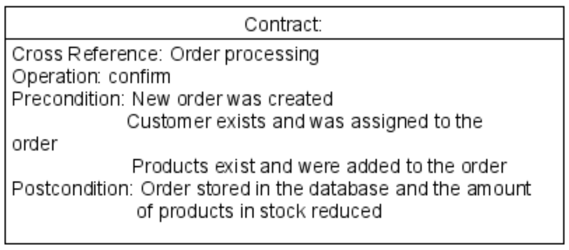
SSD, contract and interaction diagram

The system sequence diagram serves as very helpful tool for achieving more detailed and formal use case description. What is more, system sequence diagram shows, how user of the program interacts system and how the system responds to the actions of the actor. Diagram is created from fully-dressed use case.

Our project includes one system sequence diagram called “Order processing”. First, the users searches for customer through his ID. User then adds products that should be included within the order and confirms his choice.

The contract is meant to give us better understanding of what changes are being made and within our system. Not only that, but thanks to contracts we will be able to see, when the changes are being made. In our case, when we are adding products into our order, the list has already been created.

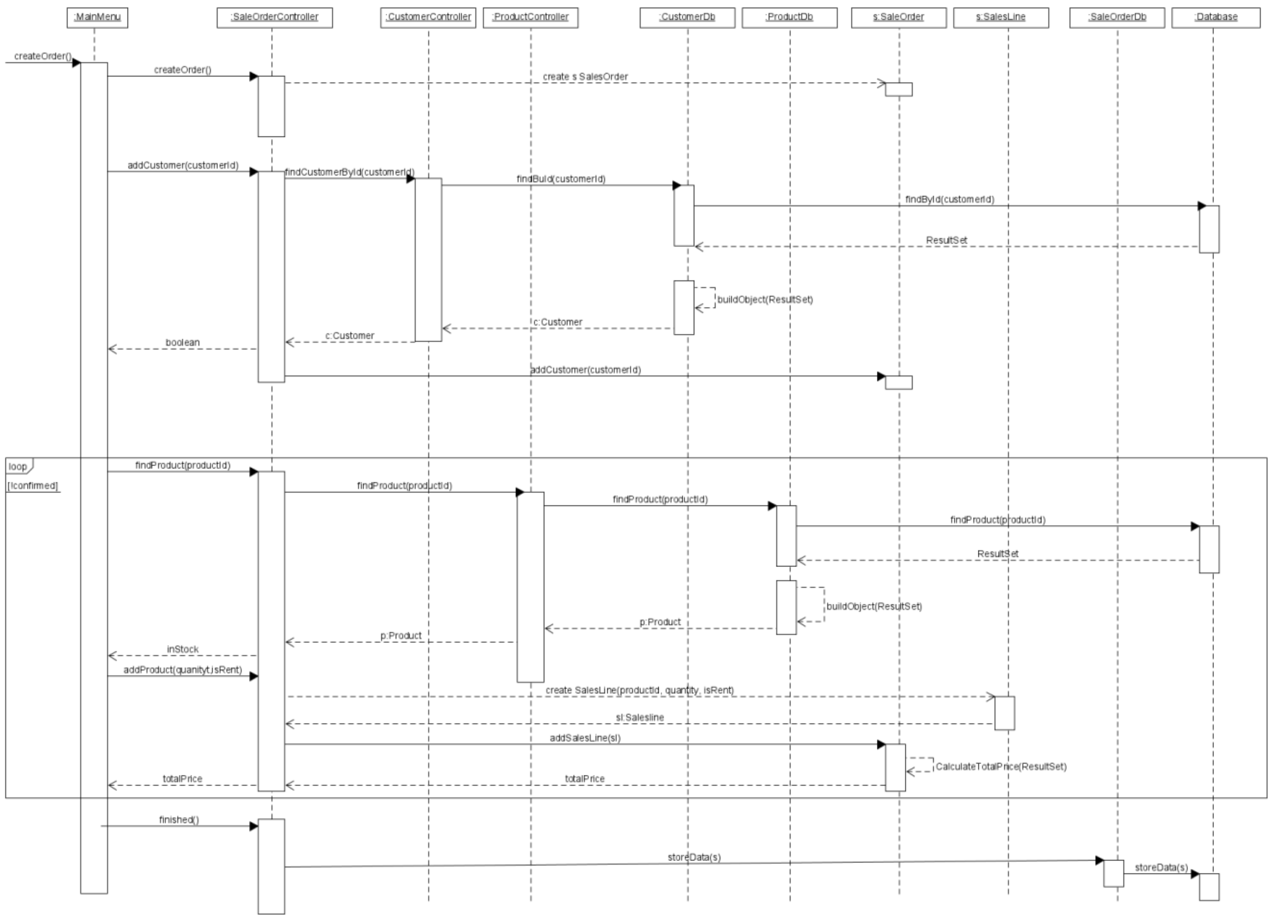



Contract:

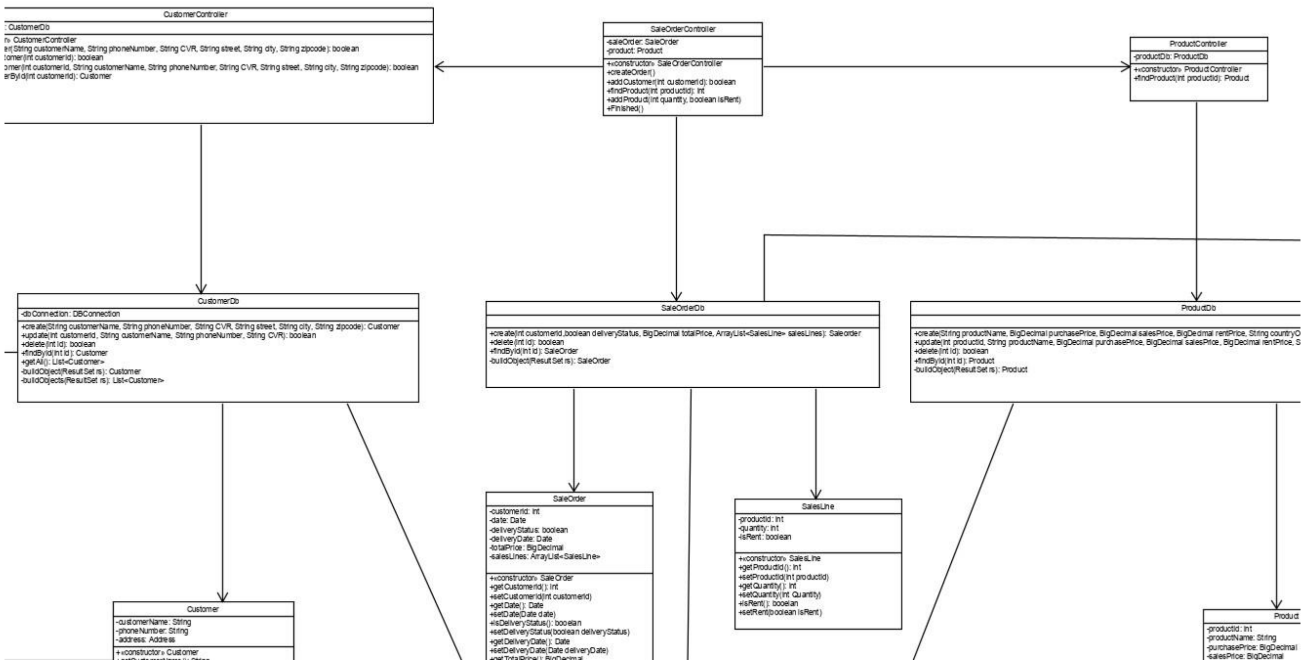
Cross Reference: Order processing
Operation: confirm
Precondition: New order was created
Customer exists and was assigned to the order
Products exist and were added to the order
Postcondition: Order stored in the database and the amount of products in stock reduced

Interaction diagram serves as the model that describes, how objects collaborate between each other. The behaviour as well as messages passed between these objects and new objects being created are included in the diagram. For our project, we have decided to use a sequence diagram. The decision was made since it's easy to read and to work with since we have used this type of diagram several times.

In our interaction diagram, we go step by step through the creation of order for the customer. At the beginning, after you create a new order, you add a customer, to which the order will belong to. After that we search for the product, which will be later included within the created salesline. This process is repeated till the order includes all necessary products. After that, the total price for the whole order will be stated and the user will confirm his choice. At the end, all the data are stored in the database.


:MainMenu
:SaleOrderController
:CustomerController
:ProductController
:CustomerDb
:ProductDb
s:SaleOrder
s:SalesLine
:SaleOrderDb
:Database
createOrder()
createOrder()
create s:SalesOrder
addCustomer(customerId)
findCustomerById(customerId)
findBuId(customerId)
findById(customerId)
ResultSet
buildObject(ResultSet)
c:Customer
c:Customer
boolean
addCustomer(customerId)
loop
[!confirmed]
findProduct(productId)
findProduct(productId)
findProduct(productId)
findProduct(productId)
ResultSet
buildObject(ResultSet)
p:Product
p:Product
inStock
addProduct(quanityt,isRent)
create SalesLine(productId, quantity, isRent)
sl:Salesline
addSalesLine(sl)
CalculateTotalPrice(ResultSet)
totalPrice
totalPrice
finished()
storeData(s)
storeData(s)

Design class diagram


CustomerController
: CustomerDb
» CustomerController
er(String customerName, String phoneNumber, String CVR, String street, String city, String zipcode): boolean
tomer(int customerId): boolean
omer(int customerId, String customerName, String phoneNumber, String CVR, String street., String city, String zipcode): boolean
erById(int customerId): Customer
SaleOrderController
-saleOrder: SaleOrder
-product: Product
+«constructor» SaleOrderController
+createOrder()
+addCustomer(int customerId): boolean
+findProduct(int productId): int
+addProduct(int quantity, boolean isRent)
+Finished()
ProductController
-productDb: ProductDb
+«constructor» ProductController
+findProduct(int productId): Product
CustomerDb
-dbConnection: DBConnection
+create(String customerName, String phoneNumber, String CVR, String street, String city, String zipcode): Customer
+update(int customerId, String customerName, String phoneNumber, String CVR): boolean
+delete(int id): boolean
+findById(int id): Customer
+getAll(): List<Customer>
-buildObject(ResultSet rs): Customer
-buildObjects(ResultSet rs): List<Customer>
SaleOrderDb
+create(int customerId,boolean deliveryStatus, BigDecimal totalPrice, ArrayList<SalesLine> salesLines): Saleorder
+delete(int id): boolean
+findById(int id): SaleOrder
-buildObject(ResultSet rs): SaleOrder
ProductDb
+create(String productName, BigDecimal purchasePrice, BigDecimal salesPrice, BigDecimal rentPrice, String countryO
+update(int productId, String productName, BigDecimal purchasePrice, BigDecimal salesPrice, BigDecimal rentPrice, S
+delete(int id): boolean
+findById(int id): Product
-buildObject(ResultSet rs): Product
SaleOrder
-customerId: int
-date: Date
-deliveryStatus: boolean
-deliveryDate: Date
-totalPrice: BigDecimal
-salesLines: ArrayList<SalesLine>
+«constructor» SaleOrder
+getCustomerId(): int
+setCustomerId(int customerId)
+getDate(): Date
+setDate(Date date)
+isDeliveryStatus(): boolean
+setDeliveryStatus(boolean deliveryStatus)
+getDeliveryDate(): Date
+setDeliveryDate(Date deliveryDate)
SalesLine
-productId: int
-quantity: int
-isRent: boolean
+«constructor» SalesLine
+getProductId(): int
+setProductId(int productId)
+getQuantity(): int
+setQuantity(int Quantity)
+isRent(): boolean
+setRent(boolean isRent)
Customer
-customerName: String
-phoneNumber: String
-address: Address
+«constructor» Customer
Product
-productId: int
-productName: String
-purchasePrice: BigDecimal
-salesPrice: BigDecimal

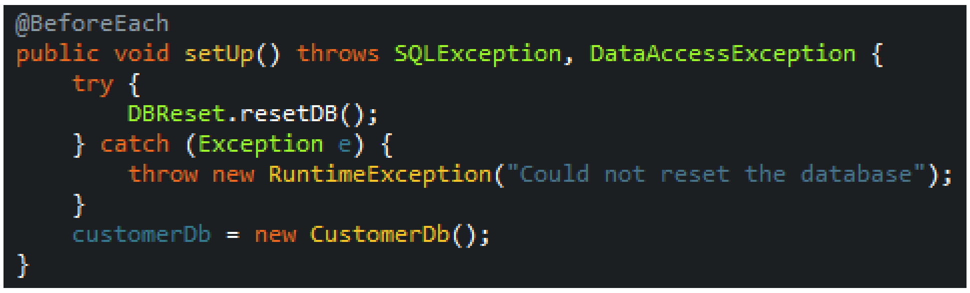
Design class diagrams display all the classes, methods and fields in the system, return types and which classes access which. Since the diagram is too big we included it also in the repository path. This part of the diagram shows SaleOrderController and its access to CustomerController and ProductController. SaleOrderController has only access to SaleOrderDb, so in the order to retrieve products and customers from the database it needs to access their controllers. With that, we ensured a clear and logical structure which is readable and easy to maintain. DAO pattern which we used is partially visible in this part. Controllers have access to the database layer, which has access to the model layer for creation of objects. In the database layer there is a singleton implemented "Connection".

Testing

Testing overall was very important and helped us quite a lot with the implementation of our program. Especially in the beginning when we don't have a working GUI with which we can manually test or a TUI, but even with them testing would be bothersome. At the very beginning we used the well-known System.out.println, but after creating our first Unit Tests for the Customer, which was the first CRUD we started with, the flow of testing and coding became much smoother. Eventually we created our database connection and a Unit test with it, which notified us that we are indeed connected to it. Following that we created a sql script with test data for our database, so that we can test even quicker without needing to write more code for creating objects. And later we implemented a runnable class for resetting the database, instead of executing the script query manually every time in the sql server management studio. Our testing ended or rather pushed us till the end of the implementation of our use case - Order Processing.

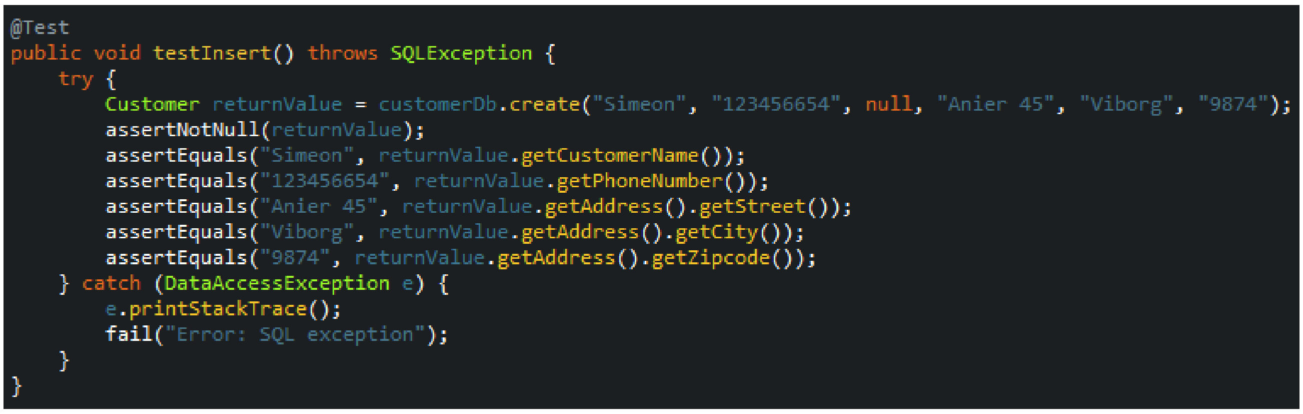
Test Cases

CRUD Customer (2 test cases)



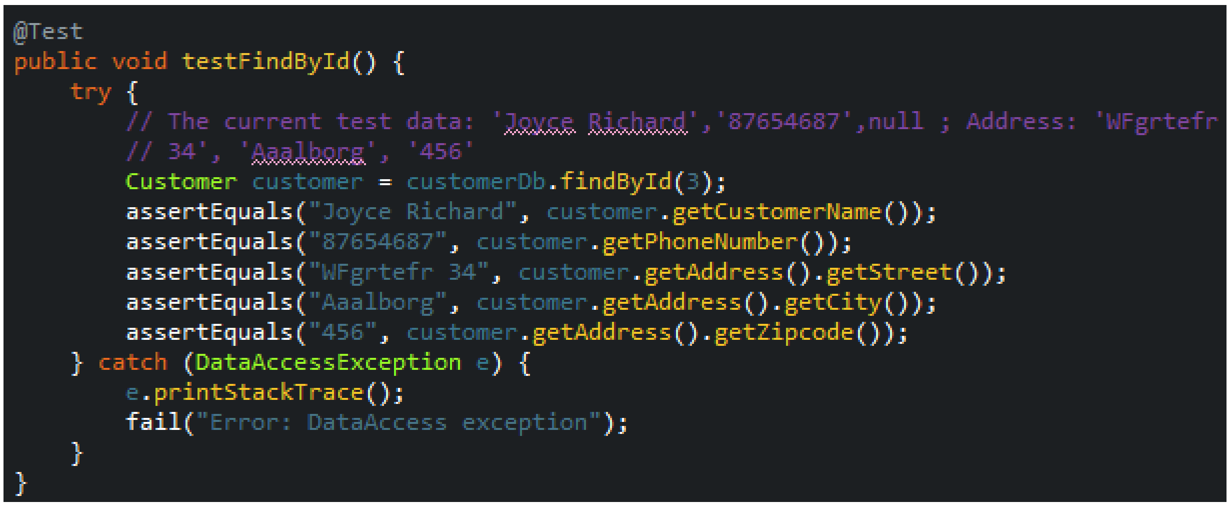
```
@BeforeEach
public void setUp() throws SQLException, DataAccessException {
    try {
        DBReset.resetDB();
    } catch (Exception e) {
        throw new RuntimeException("Could not reset the database");
    }
    customerDb = new CustomerDb();
}
```

Before all tests we created a setUp method, which serves for resetting the database and inserting test data into it, and creating a new object of customerDb which is our connection between the database and the customer. This method is called before each of the test cases so that everything is tested with the same data, which eliminates the possibility of some funny results.



```
@Test
public void testInsert() throws SQLException {
    try {
        Customer returnValue = customerDb.create("Simeon", "123456654", null, "Anier 45", "Viborg", "9874");
        assertNotNull(returnValue);
        assertEquals("Simeon", returnValue.getCustomerName());
        assertEquals("123456654", returnValue.getPhoneNumber());
        assertEquals("Anier 45", returnValue.getAddress().getStreet());
        assertEquals("Viborg", returnValue.getAddress().getCity());
        assertEquals("9874", returnValue.getAddress().getZipcode());
    } catch (DataAccessException e) {
        e.printStackTrace();
        fail("Error: SQL exception");
    }
}
```

One of the test cases is inserting customer data into the database or in other words creating a customer and saving him. Our create method of customerDb creates, stores and returns a customer. Here we just check whether all the information of the customer we created corresponds to the one we initially passed. We are passing a null there, which is the CVR, because Simeon isn't/doesn't own a company. And in case of an exception, it will be caught by the try&catch.

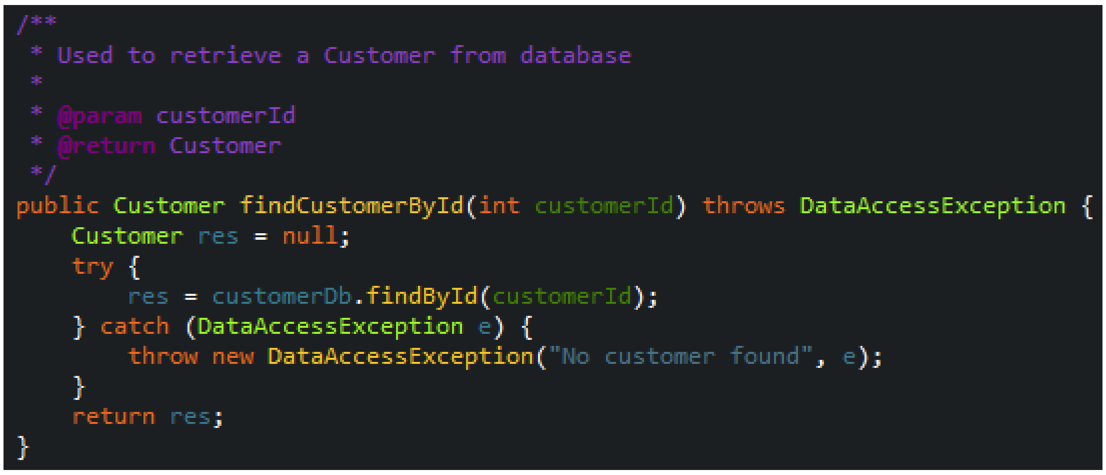


```
@Test
public void testFindById() {
    try {
        // The current test data: 'Joyce Richard','87654687',null ; Address: 'WFgrtefr
        // 34', 'Aaalborg', '456'
        Customer customer = customerDb.findById(3);
        assertEquals("Joyce Richard", customer.getCustomerName());
        assertEquals("87654687", customer.getPhoneNumber());
        assertEquals("WFgrtefr 34", customer.getAddress().getStreet());
        assertEquals("Aaalborg", customer.getAddress().getCity());
        assertEquals("456", customer.getAddress().getZipcode());
    } catch (DataAccessException e) {
        e.printStackTrace();
        fail("Error: DataAccess exception");
    }
}
```

Our second test case is the findById or the Read one from the CRUD Customer. This method is really important in fact it is used by a lot of other methods, so that's why it's really important for us to test this, whenever we modify some of the code.

Code Standard

Implementation of the code required us to agree upon a code standard to make it consistent and readable throughout the entire application. Unified code standards made the project easily maintainable, readable and extendable. Before starting to program, we agreed to use the official Java Code Conventions[1]. By following them, we properly used spacing, indentation and other good practices. Method names should state their function in the system. All methods are catching exceptions, which makes the program more failproof and easier to locate bugs if they appear.



```java
/**
 * Used to retrieve a Customer from database
 *
 * @param customerId
 * @return Customer
 */
public Customer findCustomerById(int customerId) throws DataAccessException {
    Customer res = null;
    try {
        res = customerDb.findById(customerId);
    } catch (DataAccessException e) {
        throw new DataAccessException("No customer found", e);
    }
    return res;
}
```

Groupwork

As always our group work was very well. We had no struggles in communication nor fights or disputes. If there was something we always discussed together and tried to come up with the best solution. Everyone was doing his best and made as much as possible. We needed to split the work pretty often in order to meet the deadline. To ensure that every one of us understands everything, we were asking each other if they understand every part. Thanks to that we created an enjoyable atmosphere which we were happy to work in.

Conclusion

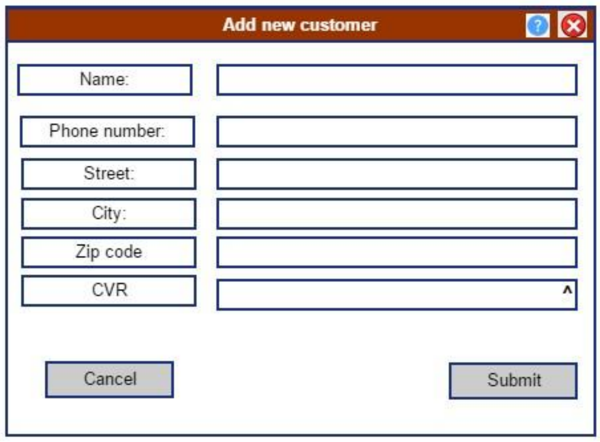
All in all, we were able to do the expected. Despite not having a lot of time we managed to create the most important part of the system and test it. We think that our consideration in architecture and design will ensure the functioning program even after future implementation. Thanks to this project we were able to test our knowledge in creating and setting database, validating it and using it as part of our software solution. Our tests ensured the quality of our code. We realised what needs to be considered during the creation of a system backed by a database and where are often errors. Sometimes we were not sure how to implement certain parts, but we managed to figure it out.

Literature

[1] Oracle.com, 2019. [Online]. Available: https://www.oracle.com/technetwork/java/codeconventions-150003.pdf. [Accessed: 26-Mar-2020].

Appendices

Appendix 1 – Mockup



Appendix 2 - SQL script

```
create table customer

 ( customerId   int                identity(1,1),

          customerName     varchar(30)  not null,

          phoneNumber  varchar(12)             not null,

          CVR        char(9) ,

          primary key (customerId));


create table customerAddress

 (street              varchar(15)   not null,

          city        varchar(15)   not null,

          zipCode  varchar(6)            not null,

          customerId   int                not null,

          primary key (customerId),

          constraint fkcustId foreign key (customerId) references customer(customerId)
```

```
            on delete cascade

            on update cascade,

            );


create table saleOrder

  (saleOrderId      int         identity(1,1),

            saleDate            date      not null,

            totalPricedecimal(6,2),

            deliveryStatus bit  default 0,

            deliveryDate   date,

            customerId          int,

            primary key (saleOrderId),

            constraint customId foreign key (customerId) references customer (customerId)

            on update cascade

            );


create table discount

 (discountAmount            decimal(6,2),

  discountDescription varchar(15),

  saleOrderId      int         not null,

  primary key (saleOrderId),

  constraint fksaleOrdId foreign key (saleOrderId) references saleOrder(saleOrderId)

            on delete cascade

            on update cascade

             );


create table product

  (productId       int                   identity(1,1),
            productName      varchar(15)  not null,
            purchasePrice  decimal(6,2) not null,
            salesPrice        decimal(6,2) not null,
            rentPrice          decimal(6,2) not null,
            countryOfOrigin varchar(15)  not null,
            minStock          int          not null,
            inStock            int          not null,
            primary key (productId),
            );

create table salesLine
            (productId        int          not null,
             quantity    int      not null,
             isRent      bit      not null,
             saleOrderId  int              not null,
            totalPrice     decimal(6, 2)  not null,
             primary key (productId, saleOrderId),
             constraint fkprodId foreign key (productId) references product(productId)
            on delete cascade
            on update cascade,
             constraint fksaleId foreign key (saleOrderId) references saleOrder(saleOrderId)
            on delete cascade
            on update cascade
             );

create table clothing
  (productId        int                not null,
            size       varchar(3)   not null,
            colour                  varchar(8)   not null,
            primary key (productId),
            constraint fkproductId foreign key (productId) references product(productId)
            on delete cascade
            on update cascade,
            );

create table equipment
  (productId          int           not null,
            equipmentType     varchar(8)   not null,
            equipmentDescription  varchar(20)  not null,
            primary key (productId),
            constraint fkproducId foreign key (productId) references product(productId)
            on delete cascade
            on update cascade,
            );

create table gunReplica
  (productId          int                     not null,
            calibre                 decimal(3, 2)   not null,
            material                varchar(8)            not null,
            primary key (productId),
            constraint fkproId foreign key (productId) references product(productId)
            on delete cascade
            on update cascade,
            );

create table supplier
```

```
  (supplierName    varchar(15)  not null,
            supplierAddress varchar(20)  not null,
            country             varchar(10)  not null,
            phoneNum          varchar(12)  not null,
            email                varchar(25)  not null,
            primary key (supplierName)
            );


create table productSupplier
            (supplierName      varchar(15)  not null,
             productId      int     not null,
             primary key (supplierName, productId),
            constraint fksuppName foreign key (supplierName) references supplier(supplierName)
            on delete cascade
            on update cascade,
            constraint fkprId foreign key (productId) references product(productId)
            on delete cascade
            on update cascade,
);


insert into customer values ('James Cameron','79846518',null);
insert into customer values ('Franklin', '86753475','f6ds1f');
insert into customer values ('Joyce Richard','87654687',null);
insert into customer values ('Ahmad','87543876','4946hrt');


insert into customerAddress values('Denarksgade 32', 'Ulborg', '9002', 4);
insert into customerAddress values('ugirewodkpr 123', 'VIborg', '78786', 2);
insert into customerAddress values('WFgrtefr 34', 'Aaalborg', '456', 3);
```

```sql
insert into customerAddress values('Wgerty 634', 'Bike', '678', 1);


insert into product values ('Pen','12.3','8.4','5.4','Denmark','2','20');

insert into product values ('Pistol','16.3','6.4','4.6','Slovakia','7','65');

insert into product values ('Pants','15.3','3.4','5.62','USA','5','23');


insert into gunReplica values(2,'4.2','Iron');


insert into equipment values(1,'Bic','Cool');


insert into clothing values(3,'M','Pink');


insert into saleOrder values('1753-2-15', '2652.23','Not delivered','1999-2-4', 1);

insert into saleOrder values('8-8-1856', '1765.45','Pending','4-2-1999', 3);

insert into saleOrder values('4-9-1985', '8456.56','In stock','4-2-1999', 4);


insert into discount values('25.21','Private', 3);

insert into discount values('654.45','Club', 2);

insert into discount values('5.36','Private', 1);


insert into salesLine values(2, '20', 1, 3);

insert into salesLine values(1, '14', 0, 2);

insert into salesLine values(3, '5', 1, 1);


insert into supplier values('Misfits','Rngiorwe 32','Denmark','56435564','test@dk');

insert into supplier values('TWRhtyh','fergthj 85','USA','894653','test@usa');

insert into supplier values('DWOInoifew','IUjkref 32','Turkey','54896','test@tk');
insert into productSupplier values('TWRhtyh', 3);

insert into productSupplier values('Misfits', 1);

insert into productSupplier values('DWOInoifew', 2);
```

Appendix 3