when we start our project, we see such options as:

Welcome to Online store

Select option:

1. Customer

2. Salesman

3. Admin

0. Exit

**CUSTOMER**

For example, if we choose option “customer” , then we can log in using the username and password we have already created or create a new account. Now we can log in through an already created account (this can be seen in the database tables) .

Aasel03 0304

Then we can choose one of this options:

Welcome, asel

Select option:

1. Order products

2. Show products

3. My orders

0. Logout

1)If we choose show products , then application will demonstrate all products from products table

2) if we select the my orders option, we will be able to see all our orders, but since it is currently empty, the application does not display anything to us

3) and also we can make an order by filling in all the data

**SALESMAN**

If we select the salesman option, we will only log in using the already created username and password. We can take this data from the salesman’s table.

What can the salesman do? Salesman can add and delete products, as well as see all products and customer orders.

**ADMIN**

And what can the admin do? He as the boss can see everything, but this is already for a real and full-fledged application, perhaps I will make this project as a full-fledged application when I have accumulated enough knowledge.

Теория

Single Responsibility

▪ A class should have only one reason to change \*

▪ This means that class should have only one job (responsible for one thing)

▪ There is a place for everything, and everything should be in its own place (Clean room example)

▪ Expectations are often exaggerated

▪ By following the Single Responsibility Principle, you make sure that your class or module has high cohesion, which means that your class does not do more than it should

Open closed

▪ You should be able to extend a class without modifying it \*

▪ Objects or entities should be open for extension, but closed for modification

▪ Extend functionality by adding new code, instead of changing existing one

▪ Separate the behaviors, therefore, the system will not be broken and easily extended

▪ The same as you`ve used packages and libraries. One can use its functionalities but never change

Liskovs substitution principle

▪ Derived classes must be substitutable for their base classes without consumer knowing it \*

▪ Let 𝜑(𝑥) be a property provable about objects of 𝑥 of type T. Then 𝜑(𝑦) should be provable for objects 𝑦 of type S where S is a subtype of T.

▪ Every part of code should get the expected result whatever class instance is sent (knowing that this class implements the same interface)

▪ In short, this principle says that to build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another

Interface Segregation

▪ Make fine-grained interfaces with specific methods \*

▪ A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use.

▪ A client should never depend on anything more that the method which is used

▪ Changing a method should not affect on unrelated class

▪ Fat interface should be changed to several small and specific ones

Dependency Inversion Principle

Depend on abstractions, not on concretions \*

▪ It states that the high-level module must not depend on the low-level module, but they should depend on abstractions

▪ It should be possible to change an implementation easily without altering highlevel code

▪ The code that implements high-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies