Seminar 2

Object-Oriented Design, IV1350

Dana Ghafour, danagf@kth.se

2023-04-18

Contents

1 Introduction

The task is to design a program that can handle all parts, including alternative flows, of the Process Sale scenario specified in the document for seminar one. For this seminar, the program should have a well-designed public interface with high cohesion, low coupling, and good encapsulation. It should be divided into layers, as specified by the MVC and Layer patterns. The solution must also include interaction and class diagrams that describe all functionality of the Process Sale scenario, both the basic and alternative flows, and the main method. For this seminar, I decided to work with Ahmed Nabil Matar.

2 Method

We wanted a design for a program that can handle all parts, including alternative flows, of the Process Sale scenario specified in the document for seminar one. To arrive at this solution, we started by analyzing the Process Sale scenario, both the basic and alternative flows, and identifying the necessary components and their interactions. We used the domain model and system sequence diagram developed in seminar one as a starting point.

We followed the Model-View-Controller (MVC) pattern, where the Controller acts as an intermediary between the View and the Model. The View class is responsible for handling the user interface, while the Controller class handles the logic of the application and communicates with the Sale and ExternalInventorySystem classes. The Sale class handles the sale-related information and calculations, while the ExternalInventorySystem class is responsible for communicating with an external inventory system to retrieve information about items. We also used the Layer pattern to further divide the program into logical layers, such as the external inventory system layer and the item registry layer.

To ensure low coupling and high cohesion, we separated the functionality into different classes and used proper encapsulation by making certain variables and methods private. We also ensured that naming conventions for Java were followed, such as using camelCase for method names and PascalCase for class names. We made sure that parameters, return values, and types were correctly specified. Having a well-designed public interface helped us with ensuring low coupling, high cohesion, and good encapsulation, by carefully defining the responsibilities and dependencies of each component and ensuring that components only interact with those they need to.

We also included a startup main method to initialize the program. To present the interaction and class diagrams that describe all functionality of the Process Sale scenario, both the basic and alternative flows, and the main method, we created an interaction diagram(s) that depict the interactions between components during the sale process. The class diagram shows all classes in the interaction diagram(s).

All-in-all, we made sure to design a program that is modular, follows good design patterns, and has well-defined interfaces between components, ensuring high cohesion, low coupling, and good encapsulation.

3 Result

In the different figures up until Figure 3.7 below, you'll find communication diagrams illustrating the flow of messages between the different objects by using a link. The message, or method call, is illustrated with an arrow that runs along the side of the link, the name of the message (method), its parameters, type, and return value.

Figure 3.8 is the class diagram showing the classes involved in the POS system, including the Controller, View, Main, Sale, Receipt, ExternalInventorySystem, item, ItemRegistry, DiscountDTO, and Discount classes, as well as their relationships and methods. The system allows for starting and ending sales, searching for items, adding discounts, paying for items, and printing receipts. It also involves matching items in an external inventory system and managing item registries and discount information.

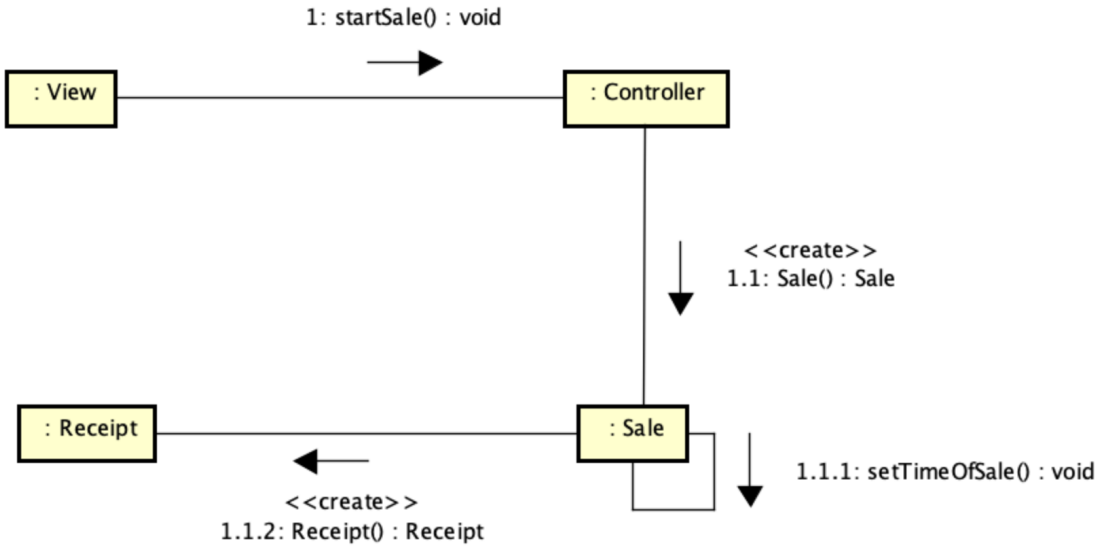

Figure 3.1: Communication diagram displaying system operation startSale.

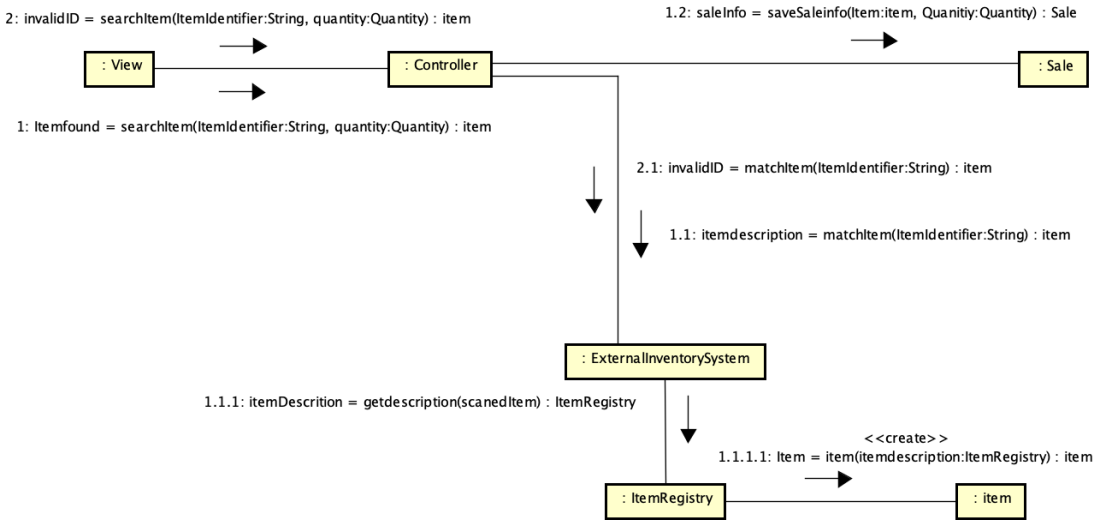

Figure 3.2: Communication diagram displaying system operation RegisterItem.

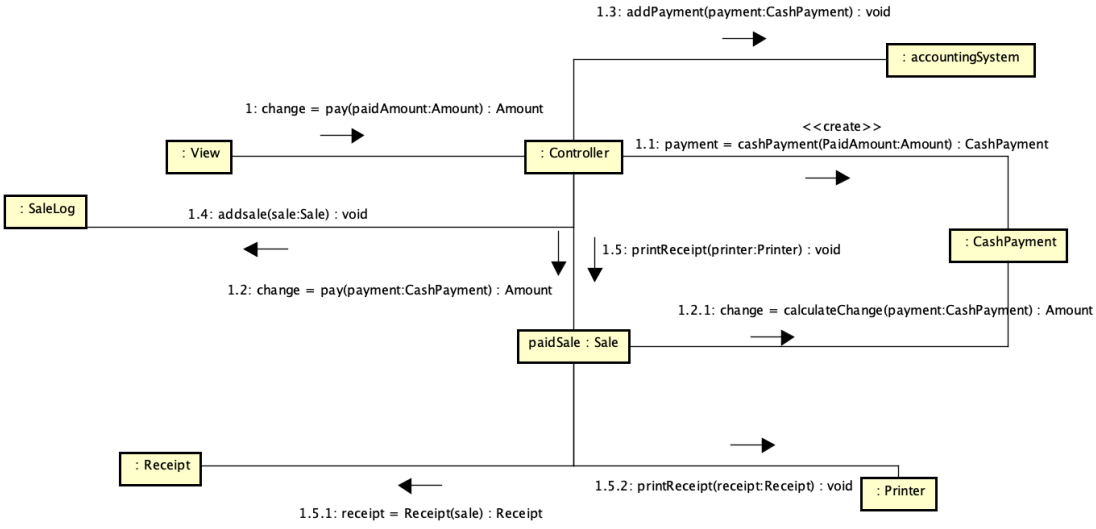

Figure 3.3: Communication diagram displaying system operation pay.

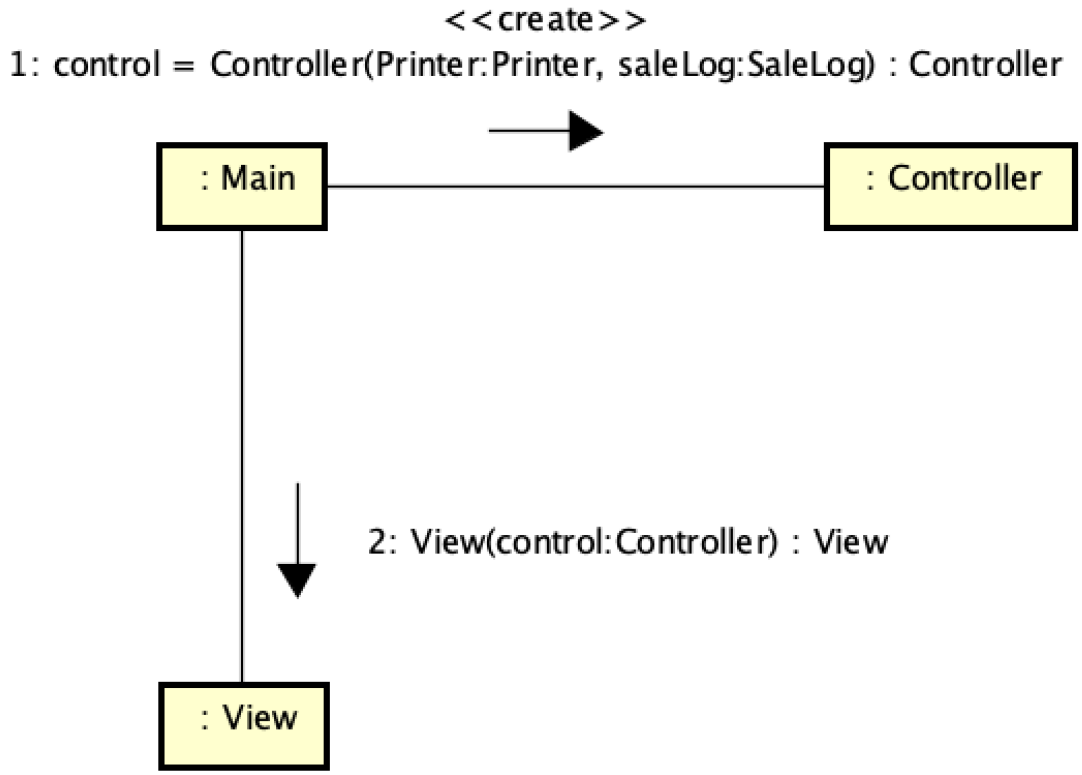

Figure 3.4: Communication diagram displaying the start sequence of the main method.

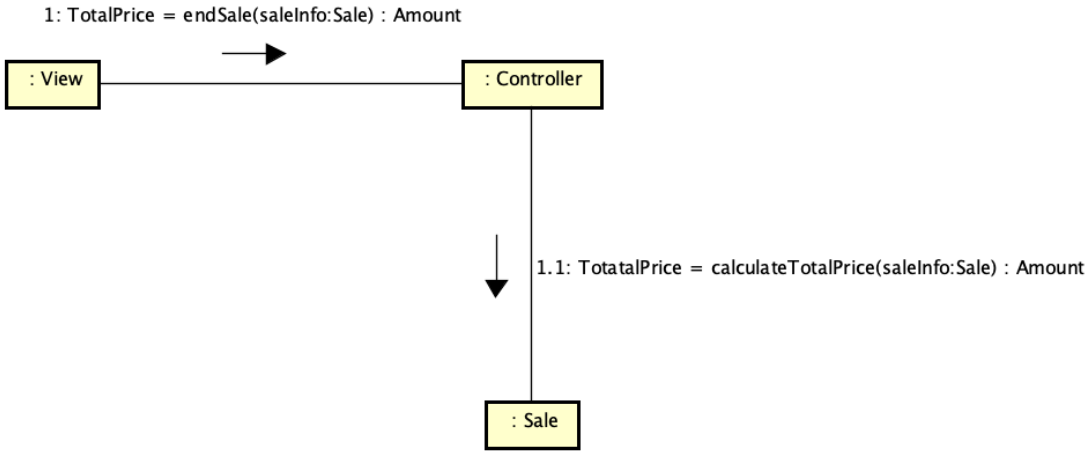

Figure 3.5: Communication diagram displaying the system operation endSale.

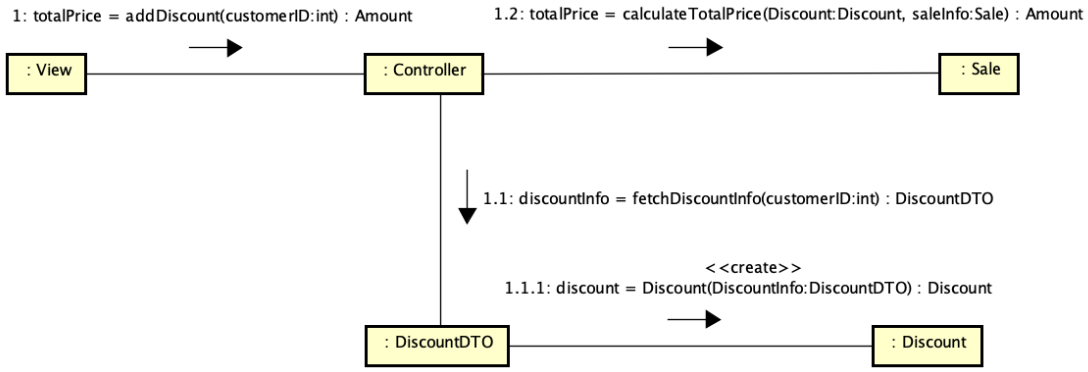

Figure 3.6: Communication diagram displaying the system operation Discount.

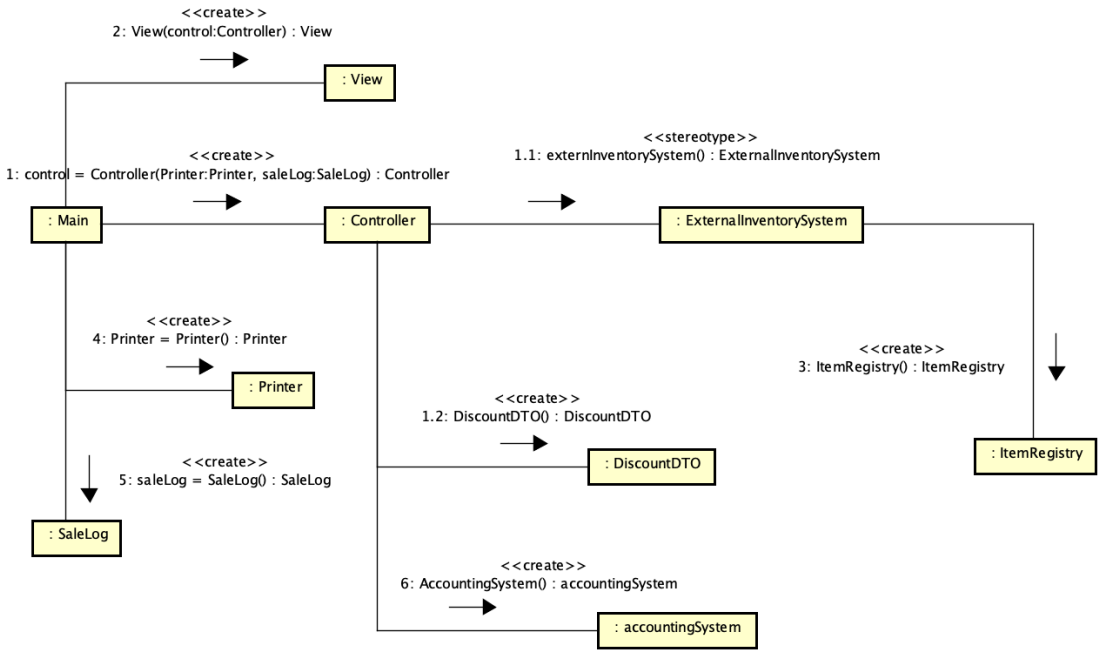

Figure 3.7: Communication diagram displaying the startup, second system operation.

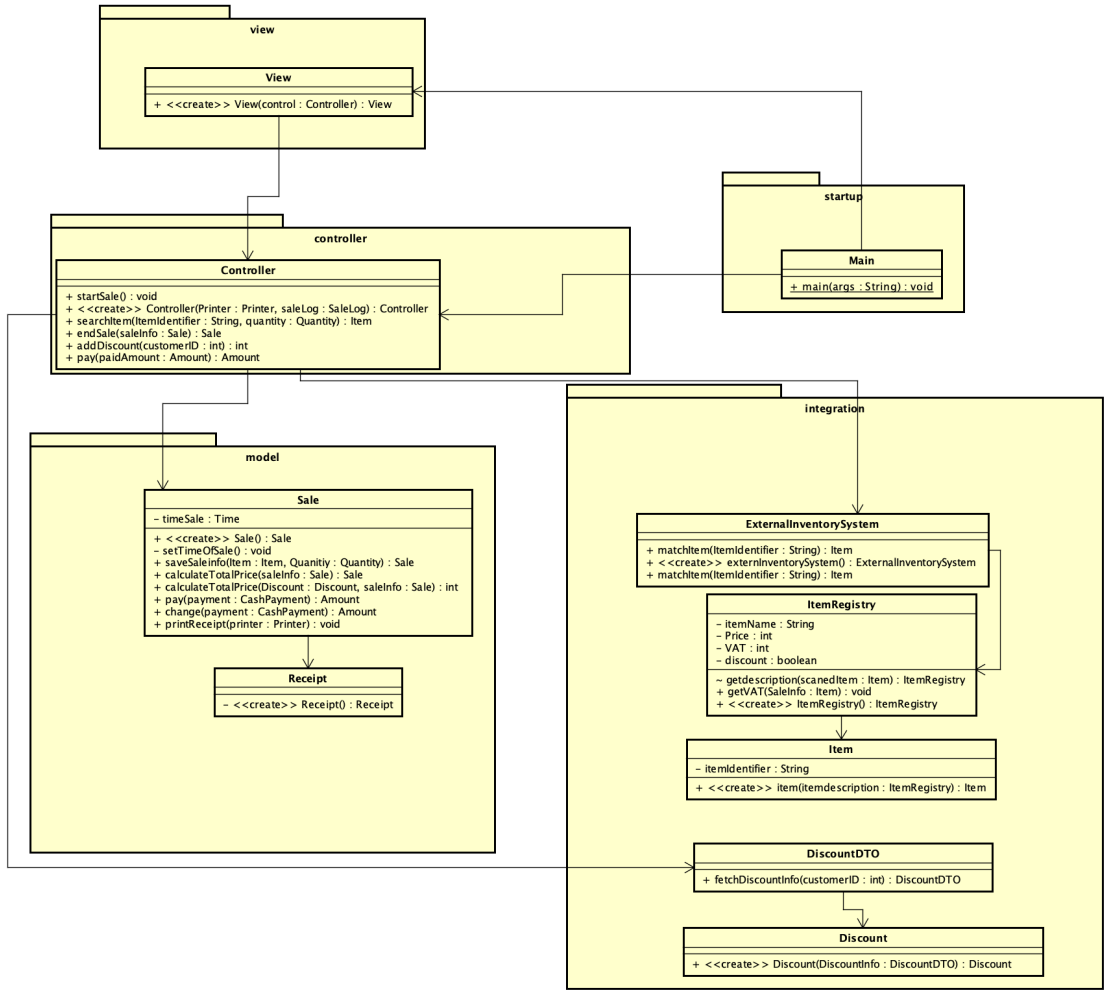

Figure 3.8: Class diagram for the Point of Sale (POS) system.

4 Discussion

The UML class diagram provided depicts the design of a software system that's a point of sale (POS) system for a retail store. The system follows the Model-View-Controller (MVC) design pattern, which separates the system's data, logic, and presentation layers.

The Controller class is part of the central part of the system, responsible for coordinating the communication between the *View* layer and the *Model* layer. It provides methods for starting and ending sales, searching for items, adding discounts, and handling payments. The View class represents the user interface of the system, and it takes care of displaying information and receiving input from the user. The Main class is responsible for starting the system and initializing the necessary components. The Sale class makes up the core of the *Model* layer, which encapsulates the data and logic for a sale. It provides methods for calculating the total price of a sale, handling payments, and printing receipts. The Receipt class represents the receipt that is printed out at the end of a sale, while the ExternalInventorySystem class is responsible for communicating with an external inventory system to match items. The ItemRegistry class is responsible for keeping track of the items available in the store, including their name, price, VAT, and whether they are eligible for discounts. Finally, the DiscountDTO and Discount classes relates to handling discounts in the system.

Also, I believe there's low coupling in the class diagram since there's no communication between view and model. As we can see, the only way to send data from model to view, to be displayed to the user, is as return values to method calls from view, via controller, to model. For instance, the searchItem() method in the Controller class receives an item identifier from the View, and uses it to search for the corresponding item in the model by calling the matchItem() method in the ExternalInventorySystem class. Once the item is found, it is returned to the Controller, which then returns it to the View. The pay() method in the Controller class, which receives an amount from the View and uses it to pay for a Sale by calling the pay() method in the Sale class. The Sale class then calculates the change to be returned to the customer and returns it to the Controller, which then returns it to the View.

One reason why I think there's high cohesion in the class diagram is because the classes have clear and distinct responsibilities, the Sale class is responsible for managing information about a sale, including its time, total price, and receipt. The Item class is responsible for holding information about individual items in the inventory. Then, I also think the classes are organized into different layers, which promotes low coupling. To give an example, the Controller layer communicates with the layer of Sale and ExternalInventorySystem respectively, but their layers do not communicate directly with each other.

One example of why I think there's also good encapsulation is by taking a look at the

addDiscount() method in the Controller class. This method is specific to the Controller's responsibility for managing the sale process, and is not exposed to other classes or methods.

I also don't think there are any static methods or attributes in the classes shown in the UML diagram. Based on the UML diagram, Java naming conventions are followed for the classes and their methods. To give an example, the class name Controller starts with an uppercase letter, which follows the Java naming convention of using PascalCase for class names. The methods in this class, such as startSale() and addDiscount(), also follow the convention of using camelCase for method names. To give two examples, the startSale() method in the Controller class has no parameters and returns void, which is appropriate since it simply starts a new sale without requiring any input from the user. The matchItem() method in the ExternalInventorySystem class takes in a String representing an item identifier and returns an Item object if a match is found. So to answer the question, yes, parameters, return values, and types are correctly specified.