Seminar 1

Object-Oriented Design, IV1350

Dana Ghafour, danagf@kth.se

2023-03-27

Contents

1 Introduction

This seminar assignment involves the creation of a domain model (DM) and a system sequence diagram (SSD) for a retail store's sales process. The reason why DM was useful for this seminar assignment is that it can help us model a domain, the reality, that we want a representation of in a developing program. To help us construct the DM, we used a UML class diagram, with each element in the DM being a representation of things that exist in reality and not just mere classes.

The SSD, on the other hand, is used to show how the developing system interacts with actors that use it. Actor in this case means an individual or system that either gets output from the system, or sends an input to the system. The SSD can be useful for simplifying development due to the fact that it gives us a visual representation of the tasks that a system has to execute and the responses that have to be given. In short, the SSD is an illustration of the interaction between how a program interacts with actors.

A UML modeling tool is required to be utilized in creating the DM and SSD, and this is done with the help of established guidelines for analysis. Further, the DM includes both the basic and alternative flows that are described in the process sale requirements specification. I worked with Ali Madlool on Seminar 1, task 1 and 2.

2 Method

To begin, a bird's-eye view was necessary to understand what the task was all about, and in this case, it was about implementing a point-of-sale system in a System Sequence Diagram and a Domain Model. The methods by which the tasks were tackled are described below.

Task 1. Direct Model (DM)

To find class candidates for the DM, we first used two complementary methods, and those were a category list and noun identification. To start, I used noun identification to find as many class candidates as I possibly could since it was recommended to have enough classes that we can later cancel if deemed redundant.

The process by which nouns were identified was by looking at the set of nouns that are mentioned in the requirements specification for the Process Sale scenarios, and adding them as elements to the class diagram. We also made sure to give them class names that are in singular. An essential part in creating our DM was to also specify the attributes that each of the classes have, seeing that they give a definition of the properties that the class instances have.

Further, we made sure to add associations between the classes in our DM, since they inform us that the instances between classes have a link. This was important because if there's a depiction of entities that exist in the real world by the classes, then there will be a relation between instances of classes. For each association, we wanted to convey an illustrative message that describes the interaction between the target class name and the sequence origin name through the association name. For instance, we chose the association name *finalizes* between target class name *Sale*, and sequence origin name *Payment.*

When we were done with identifying the class candidates by noun, we consulted the category list as mentioned in Chapter 4 of the course book, and we were able to identify more class candidates that we could add. Me and Ali then sat down and worked on identifying classes that we deemed to be redundant to model the sale process.

When this was done, the DM finally looked satisfactory to me and Ali because we thought that the diagram was, in our view, an accurate model of the process sale scenario.

Task 2. System Sequence Diagram (SSD)

For the SSD, we wanted to illustrate the interaction between the *actors* and their utilization of the *system* under development (SUD).

With the help of the requirements specification of the process sale scenarios and the description of SSDs in chapter 4, we were able to distinguish the system from the actor, to avoid the mistake of developing the actor instead of the system that the cashier interacts with. We made sure to include all classes that are part of the SUD.

The classes we decided to choose were, among others, the cashier, the printer, the inventory system, etc. and the reason for that was because we wanted an actor, e.g. cashier, to interact with the system that we want to develop. The system in this case was just called *System*, and the rest were the actors.

For this task, we decided to omit the activation bars since it's practice in SSD and wasn't necessary.

3 Result

In Figure 3.1. you'll find the DM attached that illustrates our implementation of the alternative flow as well as the basic flow that is described in the Process Sale scenarios. The classes that we deemed to be germane to the process sale scenarios and its requirement specifications are also involved, along with the associations between the classes, association names, and attributes for each class.

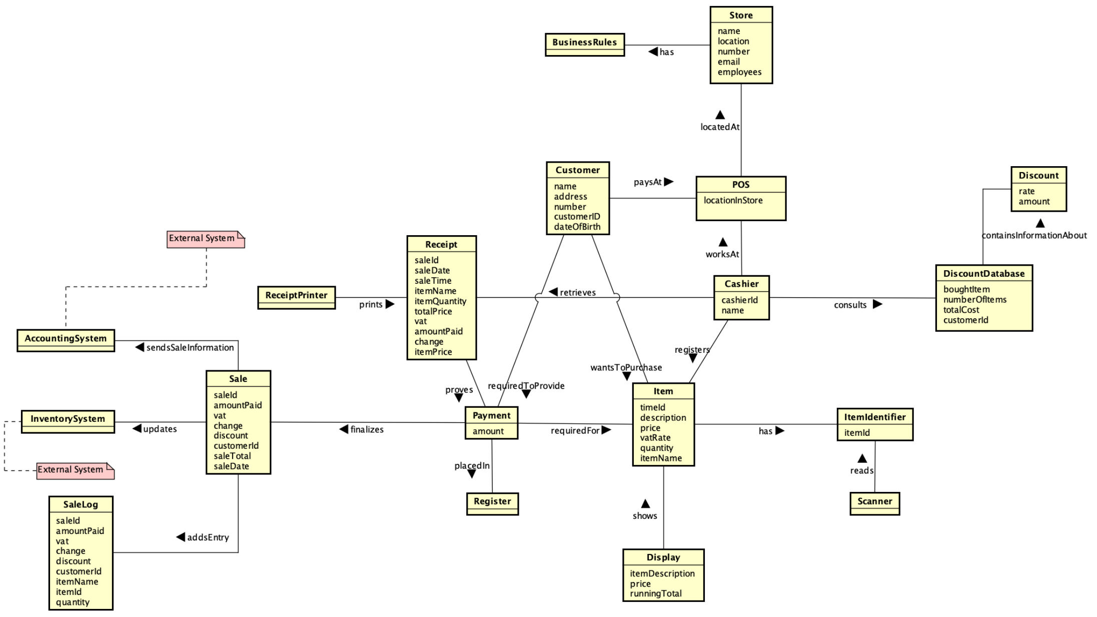

Figure 3.1: Domain Model for a retail store.

In Figure 3.2. you'll find an attachment of the SSD that depicts both the alternative as well as the basic flows of the process sale scenarios. For this diagram, we included the relevant actors and the system that is under development. Here, the central actor that interacts with the system, *System*, is *Cashier*. As we can see, most of the interactions take place between the cashier and the system, seeing that most messages are anchored at the dashed line projecting from those classes, i.e. their lifelines.

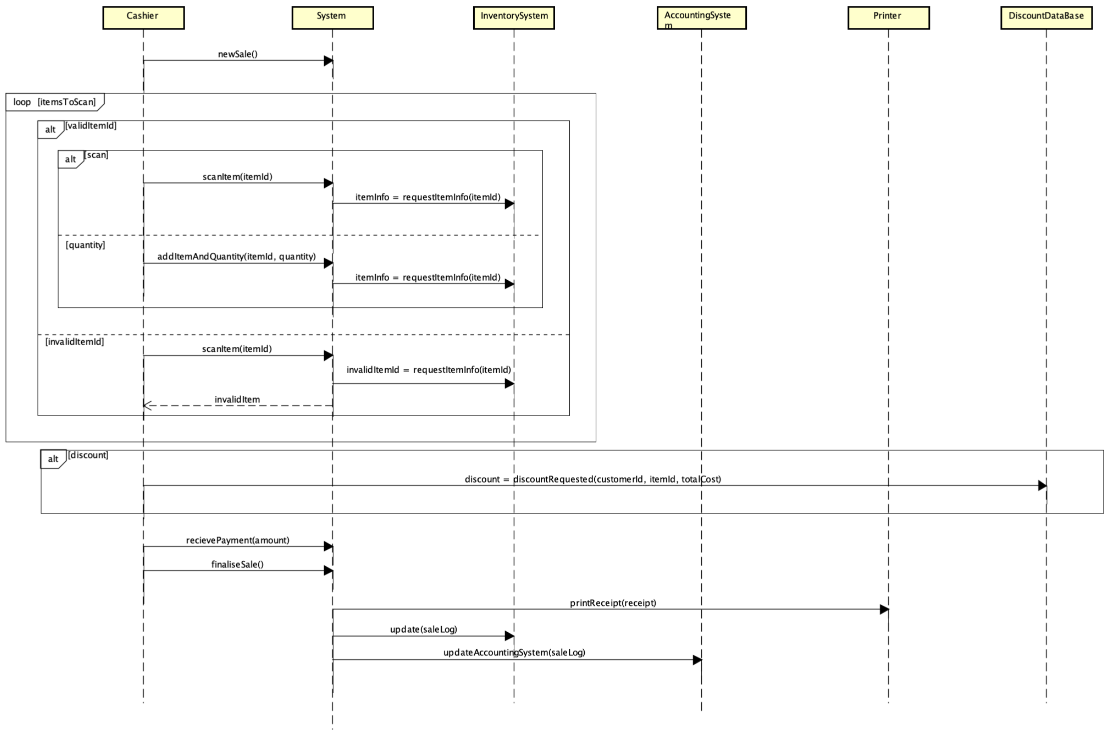

Figure 3.2: The System Sequence diagram illustrating the basic and alternative flows of Process Sale.

4 Discussion

In this assignment, we deemed that the DM was neither a programmatic DM or a naïve DM. The reason for that is that we made sure to convey information that are outside the requirement specification while maintaining a correct model of the reality in the process sale scenarios. For instance, we made sure to give a static picture of what exists rather than merely creating an abundance of outgoing associations from actors. Then there's the part about modeling a program rather than reality, which we don't think our DM did, but instead correctly models the reality. We made sure to avoid making an actor merely the user of a program.

In my opinion, it's possible to understand our implementation of the DM because it shows all the necessary entities to give us a glimpse of what happens in a real process sale scenario. We also worked iteratively on the diagram, every day, to make sure that there was not a missing link somewhere. It's also a matter of analysis of what the optimal DM could look, because people could have their own opinion of what they deem to be optimal. I'm open to ideas of how the DM in question could be improved further, but as it stands I'm satisfied with the implementation.

I also don't think there are any spider-in-the-web classes since given the size of the DM, we've at most 5 associations, which is also a guideline described in the course book to not have more than four or five associations for a class. For example, *Payment* and *Item* have 5 associations. They're however necessary to include because they play a key part in the sale process.

I also think there's a fair enough amount of attributes for each class, for instance in a real scenario where you have a sale process, you'd want to know e.g. the total price on the receipt, which is why I made sure to include it as an attribute *totalPrice* for the *Receipt* class.

As far as naming conventions go for the domain model and the system sequence diagram, it's correctly implemented seeing that we made sure to take into account that messages in system sequence diagrams use camel case notations, whereas classes use pascal case notation. To give an example, we want Receipt to have its first letter in capital case, and the messages in SSD, e.g. *receivePayment*, to be written in lowercase letters for the initial word, with subsequent words starting with an uppercase letter with no spaces.

Further, the system sequence diagram also shows external systems which are our so-called actors that interact with the system, and these were necessary given that there's a clear interaction between first the cashier and the system, and from the system with the other actors, or external systems, in question. To give an example, the cashier scans an item, which the system picks up in form of a message and then interacts with the class called *InventorySystem*, showing that the interaction doesn't just stop there between the

Cashier and the *System*.