# Software Engineering
# Final Project Report

Maxime Chambre
Nathaël Danesi

March 2025

# Abstract

This report details our work for the *Supermarket Checkout System* final project for the 2025 Software Engineering course at CentraleSupélec. It describes our design decisions with their advantages and drawbacks, the design patterns we applied in our code, and the work distribution. It also features our UML use case diagram, class diagrams and sequence diagrams. Our code can be recovered on this public github repository:

`https://github.com/Enzu83/supermarket-checkout-system`.

# Contents

# 1 How to run our code

For Eclipse users, the root directory of our github repository (see **??** ) can be opened as the Eclipse workspace. Then, make sure to switch to the Java perspective. If our project is not visible in the Package Explorer window, go to *File* → *Import*, then select *General / Existing Projects into Workspace*. Tick the "Select root directory" radio button at the top of the window, then click "Browse" on its right and select the *scs* folder within our root directory, and click "Finish".

Our code contains two examples of client-side code for a full use-case scenario within the Main package: UseCase1.java and UseCase2.java. These files can be executed as-is and will display output corresponding to a customer purchasing various items and attempting to pay for them. In UseCase1.java, the customer has enough credit left on their credit account to pay, while in UseCase2.java, they do not have enough money on their debit account to pay. For Eclipse users, right-clicking on one of these files in the Package Explorer window, then clicking on *Run As* → *Java Application* should run the file's code.

Our code contains JUnit tests for our classes in all packages except the Main package: each class has a corresponding JUnit test class with a name ending in "Test". For Eclipse users, in order to run our JUnit tests, the JUnit library must be present in the *scs* project's classpath, which should already be the case. If somehow it isn't, right-click on the *scs* project in the Package Explorer window, click on *Properties* at the bottom, the go to *Java Build Path* → *Libraries*, click on *classpath* then *Add Library*, and proceed to select JUnit. In order to run our JUnit tests for a class, the corresponding test class can be executed as JUnit Test. For Eclipse users, right-clicking on a test class in the Package Explorer window, then clicking on *Run As* → *JUnit Test* should run the tests.

# 2 Use cases

We identified several use cases for the Supermarket Checkout System based on its requirements. They are described in the UML use case diagram in Fig. 1. In our design and code, we decided to ignore the cashier (that is, the task of scanning items is performed by the cash register, and there is no piece of code dedicated to the cashier themselves). In addition, some use cases regarding the customer - namely, subscribing to a different plan and choosing to have their purchases delivered - are covered but are not explicitly present in the *Customer* class (for more details, see Sec. 3.4).

# 3 Design decisions

## 3.1 Structure in packages

Over the course of the project, we identified several parts of the requirements that could be isolated from the rest. These took the form of four packages in our code structure. To ease the explanataion of design decisions, these packages will be studied separately below. In addition, each one will be illustrated only by the fraction of the global UML diagrams relevant to it, in order to ease the visualization of said diagrams.

## 3.2 Summary of design patterns used

Below is a summary of the design patterns which are most prominent in each package. The use of these design patterns, along with their benefits and drawbacks, is discussed at length in each package's subsection.

- Item package (Sec. 3.3): Visitor pattern, Information Expert pattern.
- Bank package (Sec. 3.4): Creator pattern.

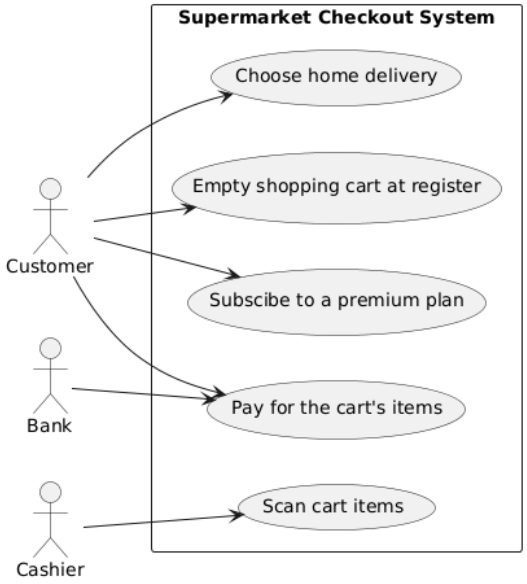


Figure 1: Use case diagram for the Supermarket Checkout System.

- SupermarketData package (Sec. 3.5): Strategy pattern, Information Expert pattern, Creator pattern.
- Main package (Sec. 3.6): Controller pattern.

## 3.3 Item package

First and foremost, we decided to isolate the way the supermarket's products (hereafter "items") are represented and priced. Indeed, items can be priced differently - for instance, fruits and vegetables are priced depending on their weight, while water bottles have a fixed price per unit - but, ultimately, knowing the different ways items can be priced is of little interest to whoever is in charge of computing the price of a customer's cart. According to the Information Expert pattern, each group of items (e.g., a bag of apples, or three water bottles ; hereafter "line items") should be able to know its own price so that the price of a cart can be computed easily by summing these price sub-totals, hence our decision to create a package dedicated to items, line items and their prices.

The class diagram of the Item package can be found in Fig. 2. We began by creating the *Item* abstract class which aggregates identifying data for the supermarket's items: an ID, a label, and a category (for applying special pricing policies, see Sec. 3.5). It has a concrete subclass for each way an item can be priced: *UnitPricedItem* which stores a price per unit, and *WeightPricedItem* which stores a price per kilogram. Then we defined the notion of line items with the *LineItem* interface. There is one concrete class that implements it for each way an item can be priced: *UnitPricedLineItem* which stores the amount of units of the same product that is bought, and *WeightPricedLineItem* which stores how much of the product is bought in kilograms.

The reason *LineItem* and the classes that implement it are unrelated to *Item* and its subclasses *in terms of inheritance* is that there can exist several line items pertaining to the same item in different quantities. As such, each line item refers to the corresponding item via an attribute. Since there is no way to *further specify* the type of an attribute in a subclass in Java, this dependency is only present in the classes that implement *LineItem*.

Regarding the Information Expert pattern, *LineItem* should be the information expert regarding a

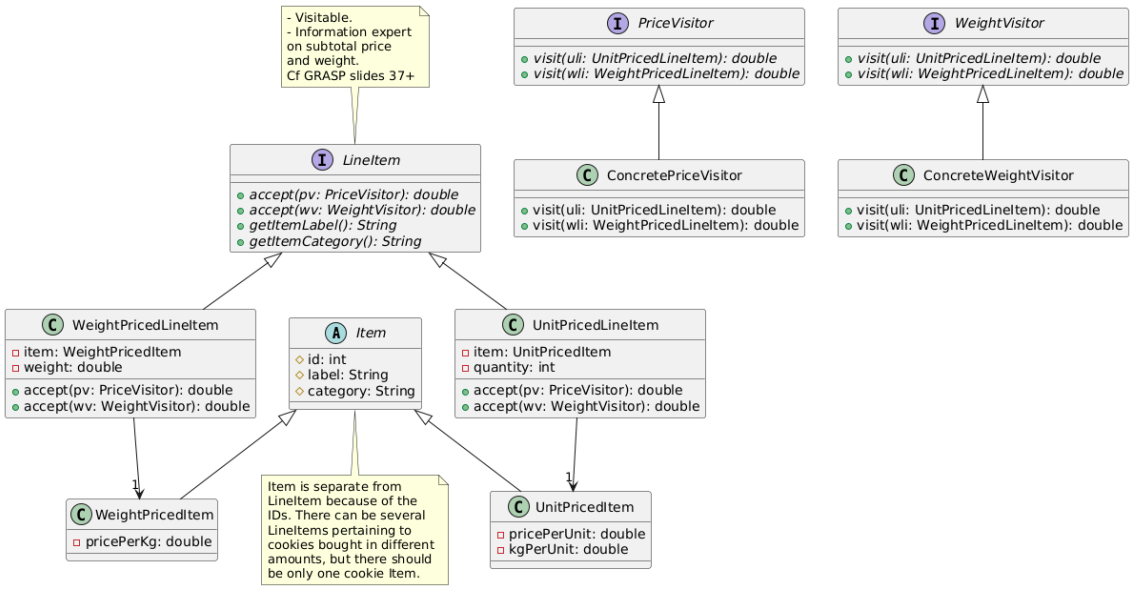


Figure 2: Class diagram of the Item package.

line item's price sub-total (without taking special pricing policies into account), as said above. Since there are several ways that a line item's price can be computed (depending on the way its item is priced), the Visitor pattern is particularly relevant to compute the price. Therefore, we created the *PriceVisitor* interface, with a concrete implementation of it, and made *LineItem* visitable through an abstract *accept* method. We carried out the same reasoning regarding the computation of a line item's weight (which will be needed later for the computation of delivery fees).

This use of the Visitor pattern brings several benefits: it allows to compute various properties of similar yet different items without changing their structure ; it allows to add new ways to price items and new properties to compute on items (e.g., items' carbon footprint) relatively easily ; and the code to compute these item properties is centralized within the visitors and is therefore easy to maintain. However, it also brings some drawbacks, as it breaks the encapsulation of classes that implement *LineItem* and thus, any change made in these classes may entail changes in every visitor interface and concrete implementation, which are rather heavy code modifications.

## 3.4 Bank package

Then, we noticed that all requirements pertaining to the payment, i.e., those regarding the Point of Sale (POS), the Transaction Authorization System (TAS), the bank and its accounts, etc., need absolutely no knowledge of the inner workings of the supermarket. Indeed, these components only intervene once the final price to pay by the customer (cart price and delivery fees, with possible discounts on both taken into account) is computed, hence our decision to isolate them in a package.

The class diagram of the Bank package can be found in Fig. 3. Since the requirements described paying by card, we needed to create classes representing payment cards and bank accounts. We included two different kinds of accounts, *CreditAccount* and *DebitAccount*, as subclasses of an abstract *Account* class for looser coupling. We also created a *Card* class, which simply aggregates data - the ID of the account it is linked to, its card number and its pin. Since *Account* possesses the necessary information to instanciate a *Card* - namely, the account ID - we assigned the responsability of creating them to *Account*, through the *createNewCard* method, as per the Creator pattern principles.

Then, we added a *Bank* class with the responsability to aggregate *Account* instances (and relay

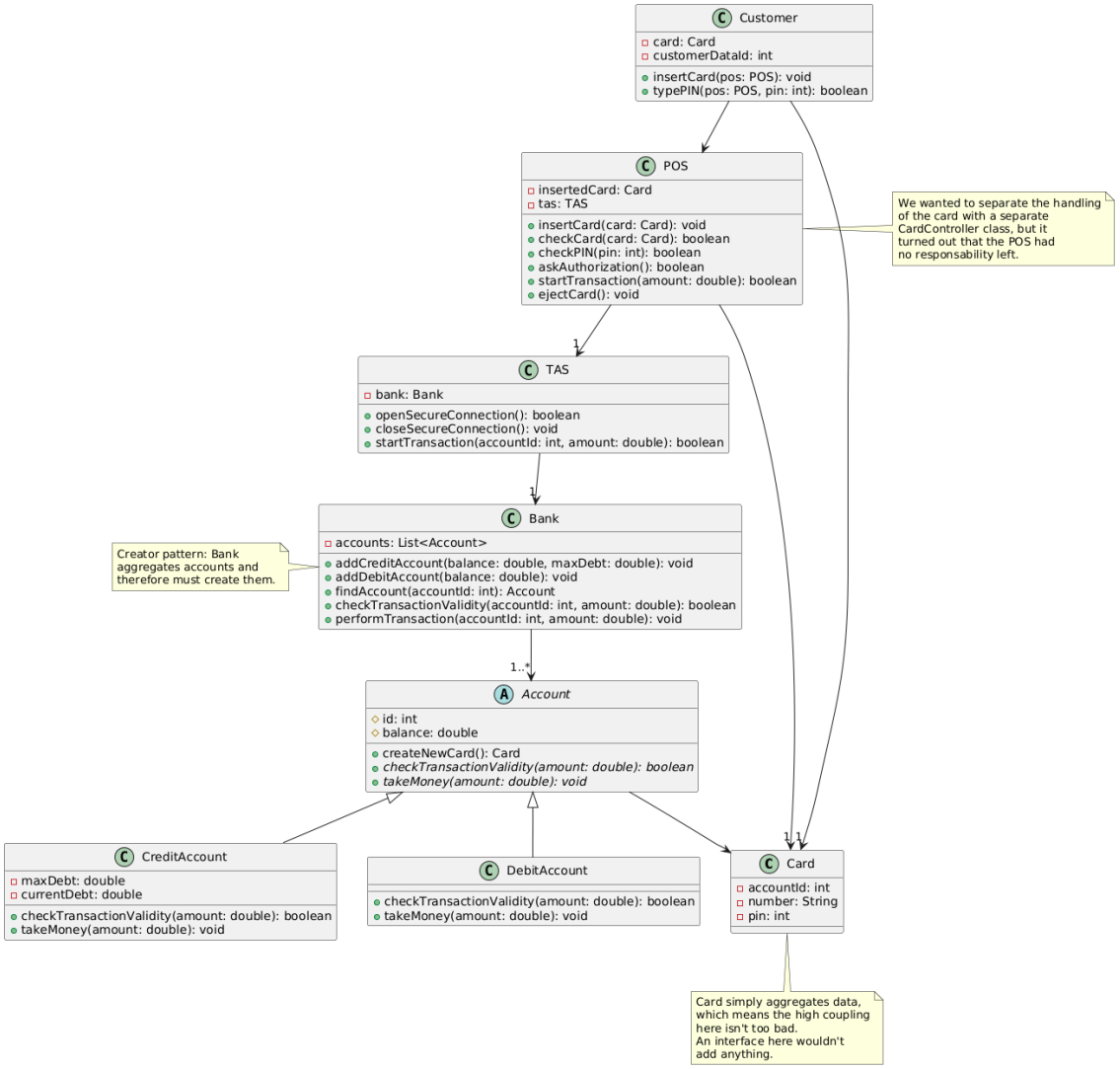


Figure 3: Class diagram of the Bank package.

payment verification and orders to them). As the Creator pattern dictates, *Bank* must create *Account* instances, hence its *addCreditAccount* and *addDebitAccount* methods. We can notice here a drawback of the Creator pattern: when assigning the responsability to instanciate subclasses of an abstract class, one creation method must be written for each subclass, which locally defeats the loose coupling purpose of the abstract class.

Next, we added the *TAS* and *POS* classes. The *TAS*'s sole responsability is to manage the secure connections between the *Bank* and the *POS* and relay transaction requests through them. The *POS* is in charge of initiating transactions and handling the customer's *Card* (checking its number and the PIN entered by the customer). We pondered creating a class dedicated to handling the *Card*, but if we did so then the *POS* would be left with no responsability (as the *TAS* is also in charge of relaying transaction requests), hence our decision to let the *POS* handle the *Card*.

Lastly, we added a class for the *Customer*, which is in charge of holding a card and interacting with the *POS* through inserting the *Card* and typing a PIN. At this point, we can notice that there is a relatively high coupling for the *Card* class, as 3 classes depend on it. However, since *Card* is only

responsible for aggregating data, replacing it with an interface or abstract class wouldn't change anything, hence this relatively high coupling is not a problematic.

Throughout this package, our uses of the Creator pattern foster a looser coupling and stronger encapsulation by assigning instanciation responsabilities to the classes best suited for it.

Since this package was created for the purpose of encapsulating all components related to the payment, we could draw a UML sequence diagram for a sample interaction between said components. It can be found in Fig. 4.

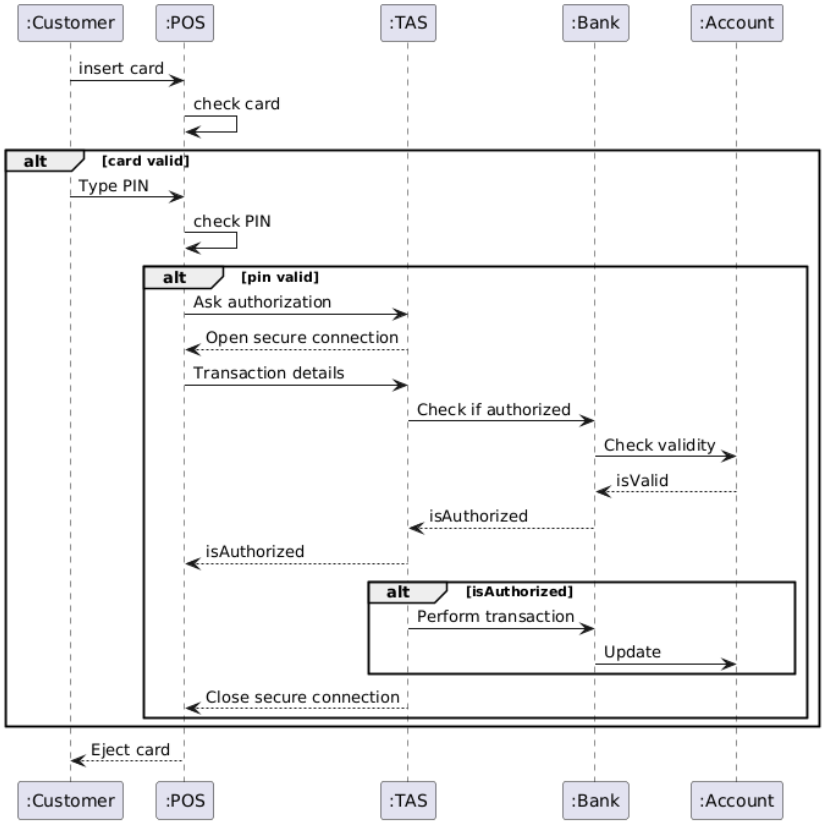


Figure 4: Sequence diagram of the Bank package.

## 3.5 SupermarketData package

Next, we noticed that two spcific requirements - namely, R5b and R6b - required the client of the Supermarket Checkout System to be able to introduce new customer discount plans or pricing policies for categories of items, or modify existing ones. This entails that the client application should be decoupled from the implementation of customer plans and pricing policies, which is clearly the use case of the Strategy pattern. It also entailed that some sort of database-like structure must be present in the code to store the various customer plans and pricing policies that the supermarket is using at a given time. All these reasons, in addition to the fact that customer plans and pricing policies do not need to directly depend on any of the two previously discussed packages, drove us to create a dedicated, separate package for them.

The class diagram of the SupermarketData package can be found in Fig. 5. Most relevantly, it features the *CustomerPlan* interface and *PricingPolicy* abstract class, which both allow to use the

Strategy pattern on their respective subclasses, as mentioned earlier. Said subclasses are not represented on Fig. 5 to keep the figure less cluttered, but they include *StandardPlan*, *PrimePlan* and *PlatinumPlan* for *CustomerPlan*, and *FreshVeggiesPolicy* and *DairyPolicy* for *PricingPolicy*, as specified in the requirements. The main benefit of the Strategy pattern is that the classes detailed on Fig. 5 depend only on *CustomerPlan* and *PricingPolicy*, allowing the client to add new subclasses for them without needing to change the code.

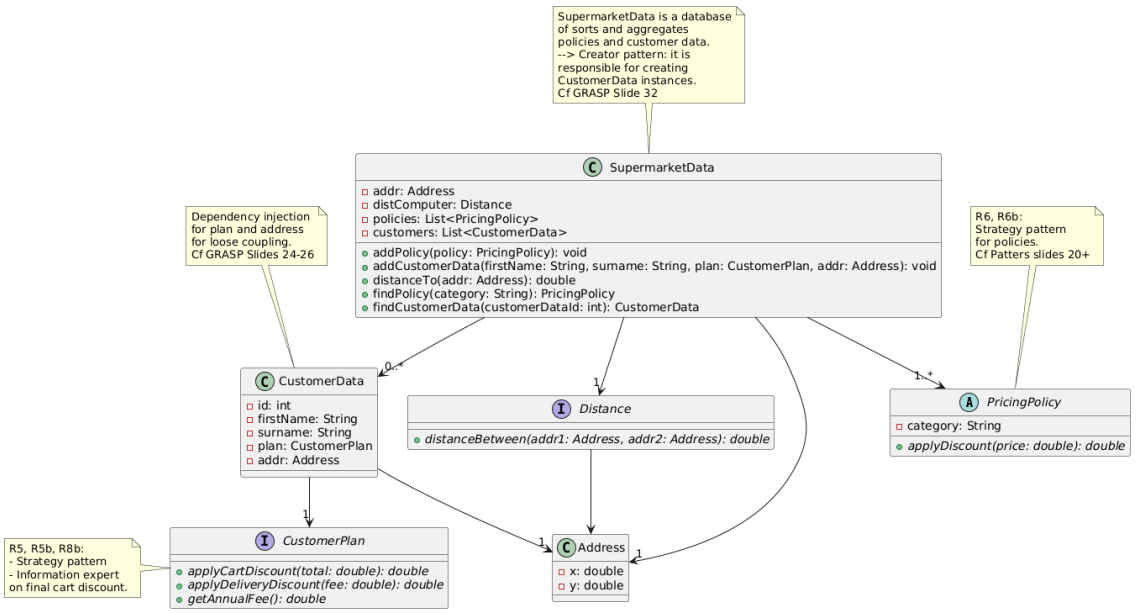


Figure 5: Class diagram of the SupermarketData package. Classes that inherit from PricingPolicy or implement Distance or CustomerPlan not represented for clarity.

On a similar note, this package contains classes dedicated to keeping track of addresses and computing distances between them. The *Address* class is simply responsible for aggregating position data (and thus, similarly to the *Card* class in Sec. 3.4, its relatively high coupling isn't an issue) and is the Information Expert for it, while the *Distance* interface is responsible for computing the distance between *Address* instances and is the Information Expert for said distance. In addition, we applied the Strategy pattern once more on *Distance* to allow to compute distances in various relevant ways and to let the client define which one best suits them. Notably, our code features the *ManhattanDistance* and *EuclideanDistance*, which are omitted from Fig. 5 for clarity.

Lastly, we added two classes which aggregate data and which are responsible for being queried for it by the cash register (notably, to compute the cart's price, as detailed in Sec. 3.6): *CustomerData* and *SupermarketData*. The former contains basic information about a registered customer of the supermarket as well as their address and discount plan, while the latter holds various pieces of information:

- The supermarket's address and a *Distance* to compute distances from anywhere to the supermarket.
- All item pricing policies currently in use in the supermarket, which can be searched for a policy applying to a specific category of items through the *findPolciy* method.
- All *CustomerData* instances pertaining to registered customers of the supermarlet, which can be searched for a specific ID through the *findCustomerData* method. As per the Creator pattern, *SupermarketData* is therefore responsible for instanciating *CustomerData*.

While the role of *SupermarketData* as a large database of sorts for the supermarket made sense from a conception standpoint, we do realize that its cohesion is lower than our other classes and that we could have split it into smaller, more cohesive classes.

### 3.6 Main package

Last but not least, our Main package contains two full use scenarios for the supermarket checkout system, as detailed in Sec. 1, as well as two additional classes that make use of our previous three packages: *CashRegister* and *SCSController*. These two classes appear on the full class diagram, class_full.png, located within the *class-diagram* folder on our github repository.

*CashRegister* is responsible for computing the final price that the customer must pay by summing all line item prices (and taking possible discount policies into account), computing delivery fees based on the cart's full weight, and instructing the customer's discount plan to apply discounts wherever appropriate.

*SCSController* is responsible for orchestrating the classes to handle various the events in a use scenario, as per the Controller pattern. This frees the client from having to worry about organizing all the code and allows them to write their client-side code as in UseCase1.java or UseCase2.java.

## 4 Work distribution

Our work distribution depending on the tasks (design, code, UML Diagrams and JUnit Tests) and the packages can be found in Tab. 1. It boilds down to the following points:

- The both of us worked on the design together, that is to say we reflected on which design patterns to apply and how to structure our UML diagrams together.
- We split the task to write the code: Nathaël wrote the code for the Item and SupermarketData packages, while Maxime wrote the code for the Bank and Main packages. We both reviewed each other's codes and made adjustments as necessary.
- Nathaël used PlantUML to draw clean versions of the UML diagrams.
- Maxime wrote all JUnit tests.

| **Package** | **Design** | **Code** | **UML Diagrams** | **JUnit Tests** |
|---|---|---|---|---|
| Item | Nathaël | Nathaël | Nathaël | Maxime |
| SupermarketData | Maxime & Nathaël | Nathaël | Nathaël | Maxime |
| Bank | Maxime & Nathaël | Maxime | Nathaël | Maxime |
| Main | Maxime & Nathaël | Maxime | Nathaël | Maxime |

Table 1: Table detailing the work distribution in terms of design, code, UML diagrams and JUnit tests for each package of the project. Each cell indicates who did most of the work in a specific task for a specific package, though it should be noted that almost every decision was examined and tuned by the both of us.