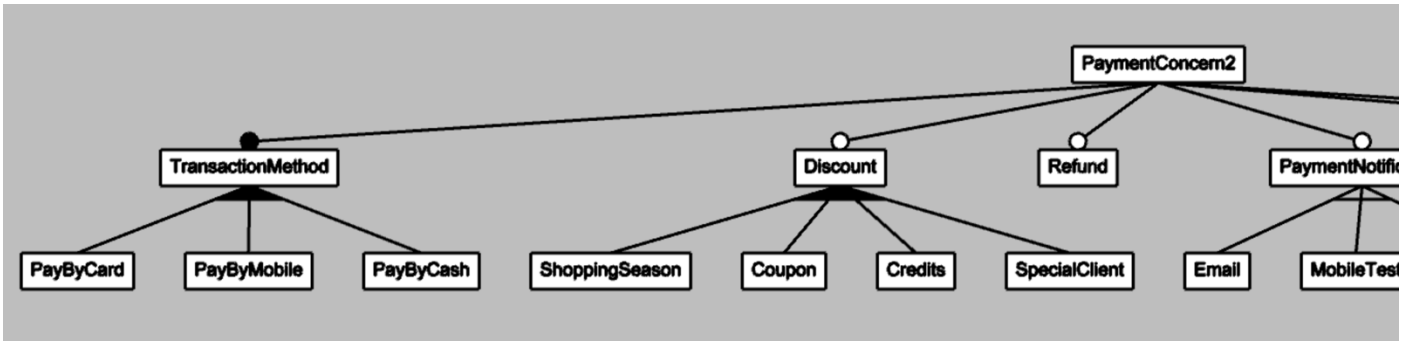
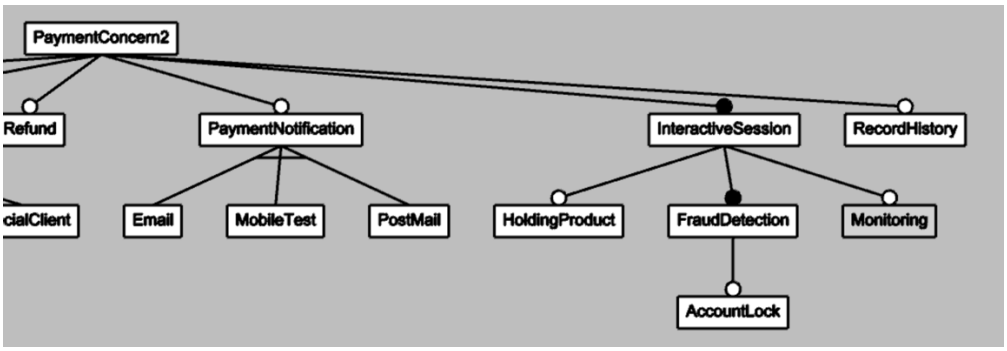
Feature Model:



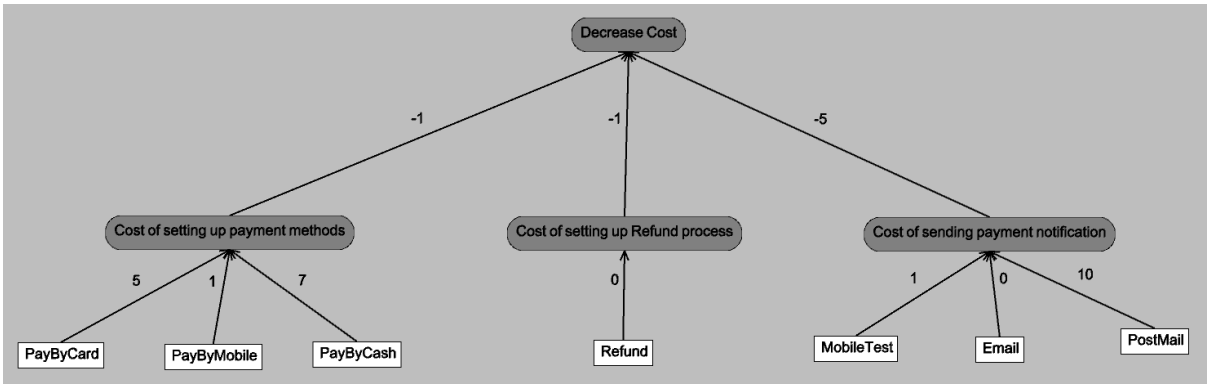


We are aiming to develop a Payment concern which is able to support multiple payment methods, diverse discount effects, refund option, various notification channels, environment safety protection, and transaction record system. Among all these features, we feel it's basically necessary to have definite transaction method and environment protection, which corresponds to FraudDection feature under InteractiveSession. It's normal for users of same shopping platform to pay things by different channels and apply discount in different form, but it's better to have unified agent to receive notifications.

Impact Model:

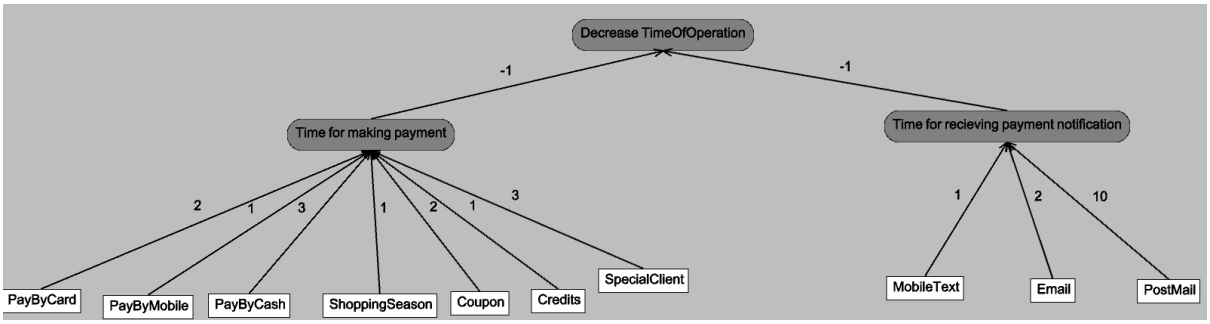
Please find below the impact model diagrams, and a brief description/justification of the weights we felt were not intuitively obvious.

1. Decrease Cost



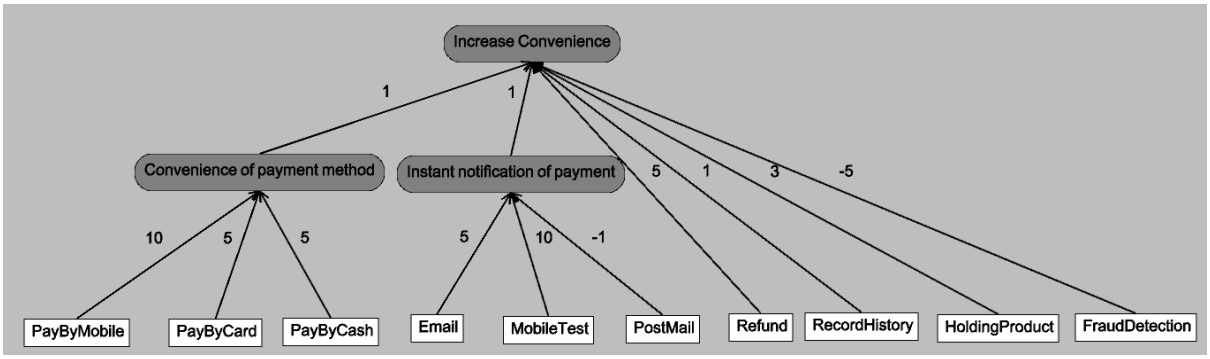
The cost of setting up a payment method and refund process (validating cause of refund, confirming refund method etc.) is a one-time cost, as opposed to the recurring cost of sending out payment notification. Thus, weights are -1 and -5 respectively. Also, for card payment, we need to buy a machine to read the card, for cash payment we need a cash register and cashier, and wallet payment requires only an app. Sending out an e-mail is free, text is marginal cost, but post is pretty expensive. The actual process of refund should not add any cost to the model, hence it's assigned 0 weight.

2. Decrease TimeOfOperation



Getting a discount will definitely add to the time of making payment. However, if it's Shopping season, everyone will be entitled to the discount, so it's 1. Credits also will be automatically deducted from customer's bill, so that's also 1. Coupon requires some kind of coupon code. SpecialClient will require additional validation of the customer. Rest all weights and model seem self-explanatory.

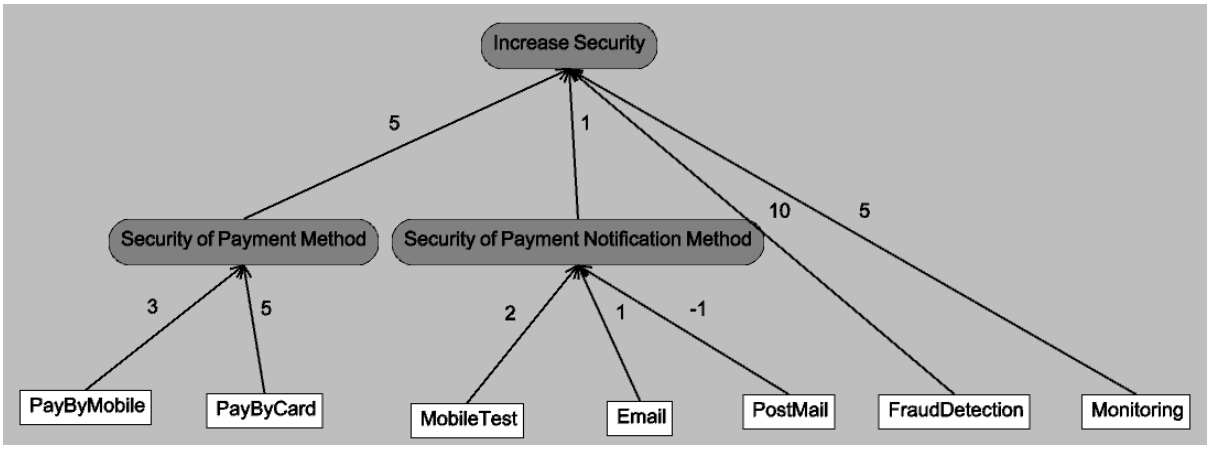
3. Increase Convenience



PayByMobile seems the most convenient method of payment because it doesn't require a second level authentication (CVV, OTP etc) needed like card, and doesn't require calculation and exchange of remaining "change" amount in card transaction.

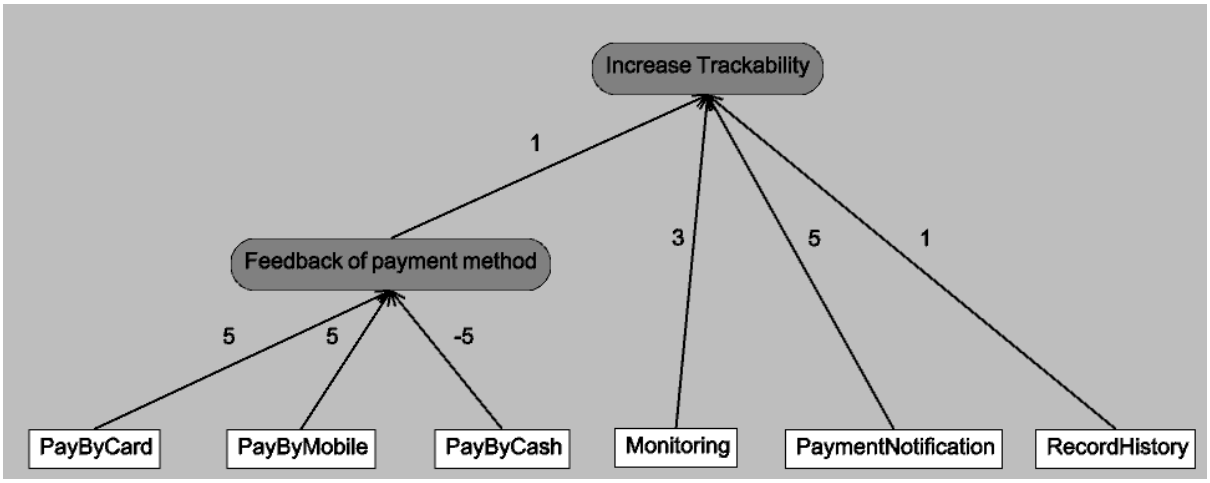
Receiving an email would require internet connection, whereas text can be received anywhere with network connectivity. Post is definitely counter-productive to convenience, since it will be received days later. Refund, RecordHistory and HoldingProduct will all add to customer convenience, but FraudDetection will be hindrance in that.

4. Increase Security



Security of Payment Notification means securing customer's confidential information against hackers. Mobile text is the most difficult to intercept, an e-mail can be hacked more easily, and post is the easiest to intercept since it offers no kind of encryption (basically anyone can tear open the envelope and read the bill). In Security of Payment Method, we only consider card and mobile, since cash doesn't possess any kind of digital security.

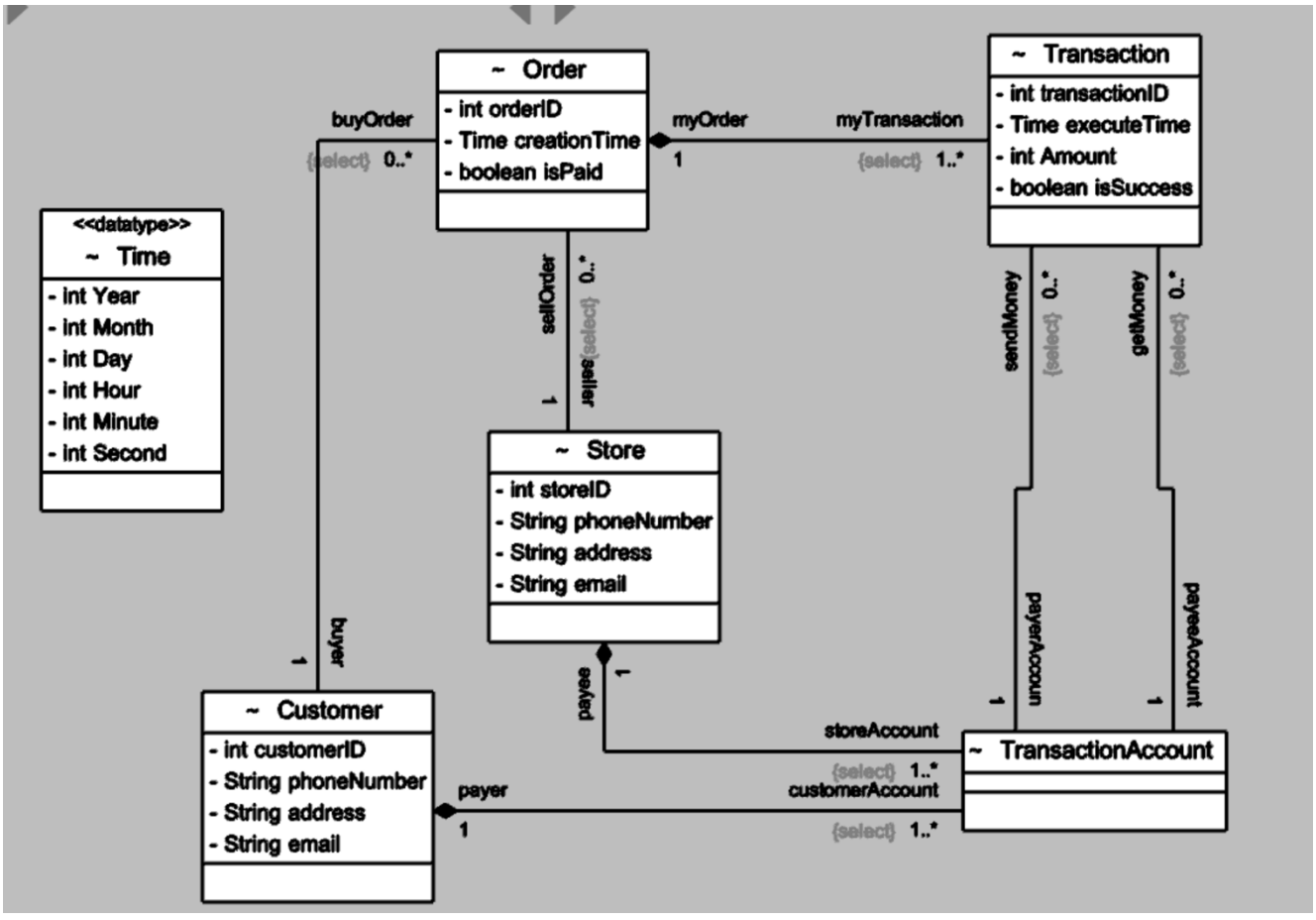
5. Increase Trackability



Cash method is very counter-productive in increasing Trackability since it leaves no digital trace of the transaction. Both card and mobile wallet leave similar trace of transaction (thus same weight). Payment Notification will be biggest proof of transaction completion, followed by Monitoring and then RecordHistory.

Domain Model: all OCL constraints are marked in blue

PaymentConcern Domain Model: (basic root feature)



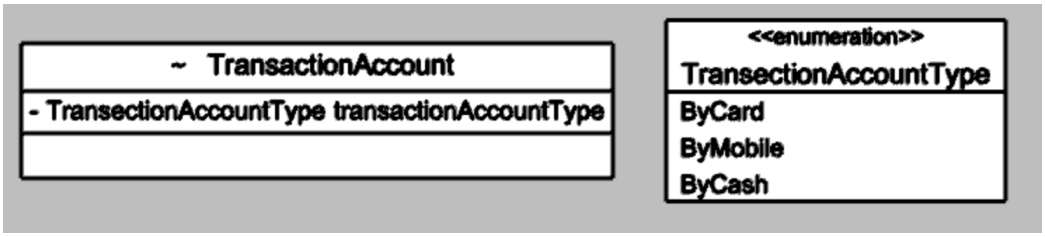
Class Store and Customer represent the two major roles of the trading system, while each role needs to maintain at least one transaction account for realizing payment functionality. Each order can have multiple transactions; imaging if one transaction fails for a given order, the user can retry it in new transaction session. It's important to notice that if one order is destroyed, its attached transaction doesn't exist anymore, similarly for the composition relationship between customer/store verse transaction account.

context Transaction inv: self.amount >= 0;

context Transaction inv: self.myOrder.buyer=self.payerAccount.payer;

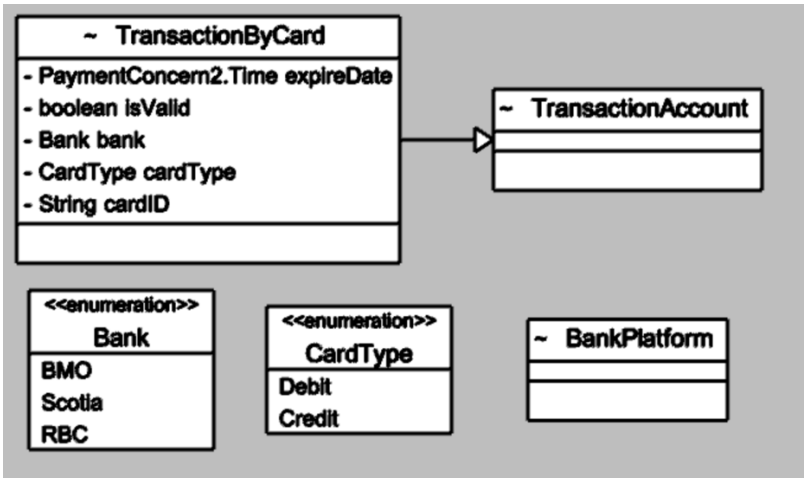
context Transaction inv: self.myOrder.seller=self.payeeAccount.seller;

TransactionMethod Domain Model: (first level sub-feature, mandatory)



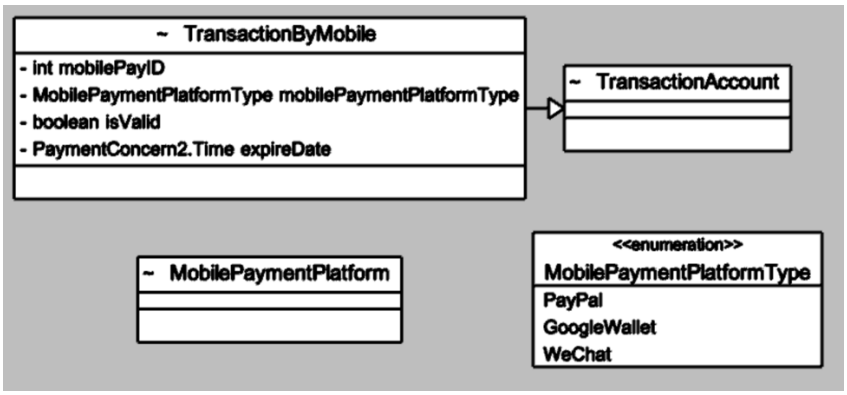
We accept payment by card, mobile e-payment, and cash.

PayByCard Domain Model: (second level-sub feature, OR relationship under TransactionMethod feature)



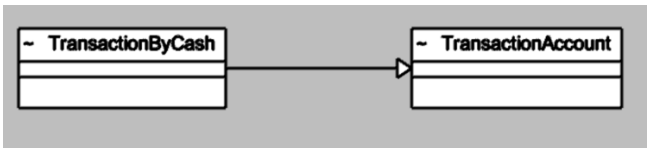
BankPlatform class here represents possible back-support role realizing by banking online system.

PayByMobile Domain Model: (second level-sub feature, OR relationship under TransactionMethod feature)



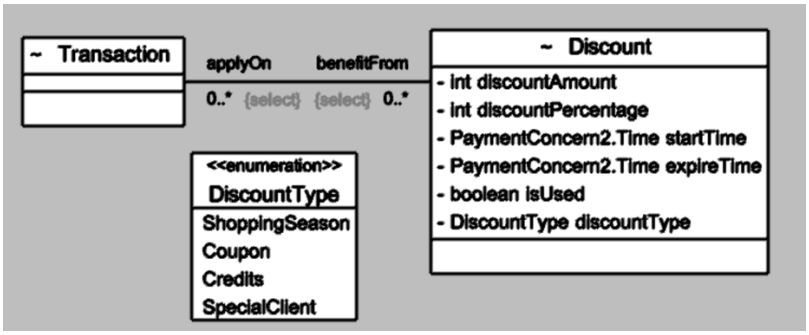
MobilePaymentPlatform class here represents possible back-support realizing by mobile e-payment online system.

PayByCash Domain Model: (second level-sub feature, OR relationship under TransactionMethod feature)



This feature is rarely needed by online shopping system like Amazon. We keep this feature in case if physical store or system with ‘cash on delivery’ function uses our payment concern.

Discount Domain Model: (first level sub-feature, optional)



Discount instance can exist by itself without applying on any Transaction. For example, we can keep a coupon without using it. Transaction surely can have no discount. Single Discount, like the one for Christmas Shopping season, can apply on million clients' transactions, while single transaction is allowed to have multiple ways of discount, like by both Christmas Shopping season and by student identity as the special client. For one Discount object, exactly one of discountAmount value or discountPercentage value can be 0. It represents two different situations: if discountAmount is non-zero, it means this type of discount is to directly reduce payment amount, while discountPercentage should be zero to avoid misunderstanding; if discountPercentage is non-zero, it means this type of discount is to pay target percent of original amount, while discountAmount should be zero respectively.

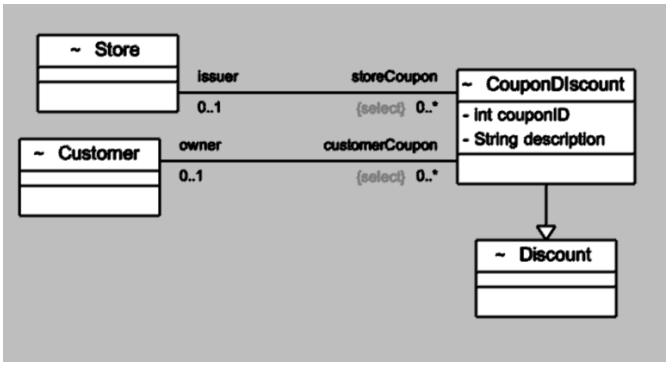
context Discount inv: (self.discountPercentage >= 0) and (self.discountPercentage <= 1);

context Discount inv: self.discountAmount <= self.applyOn.amount

ShoppingSeason Domain Model: (second level sub-feature, OR relationship under Discount feature)

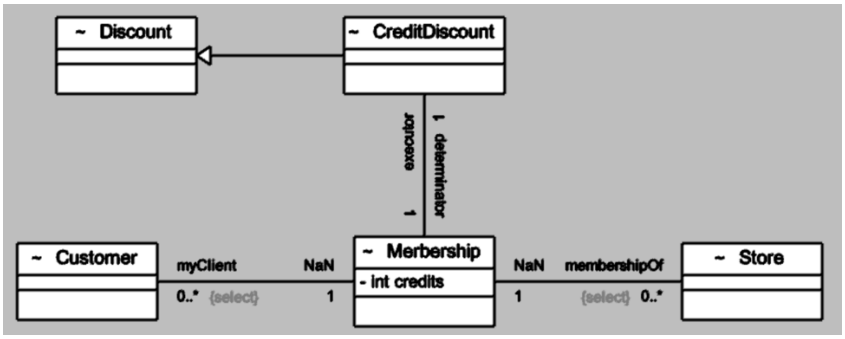
ShoppingSeasonDiscount is a sub-class of Discount. Its innate methods can fill out parent class's needing attributes by matching with corresponding shopping season type. For example, New Year season type may assign 2019.12.31 to startTime and 2020.1.3 to endTime, while Christmas may assign 2019.12.24 to startTime and 2019.12.25 to endTime.

Coupon Domain Model: (second level sub-feature, OR relationship under Discount feature)



Specific stores may grant welfare by issuing coupons to attract new clients or to retain old clients, while coupons are kept by individual customers. The specific reason behind each coupon is recorded down in the attribute 'description'.

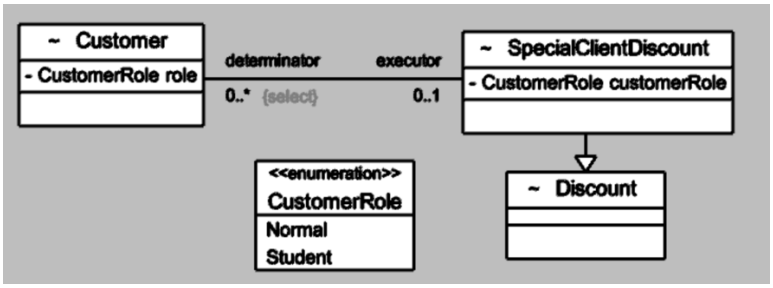
Credits Domain Model: (second level sub-feature, OR relationship under Discount feature)



Membership here should be an association class between each pair of Customer and Store objects. Each store may have 0 to many clients, while each client can belong to 0 to many stores. But the membership of one client under one store cannot duplicate.

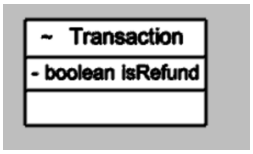
context CreditsDiscount inv: self.determinator.myClient = self.issuer

SpecialClient Domain Model: (second level sub-feature, OR relationship under Discount feature)



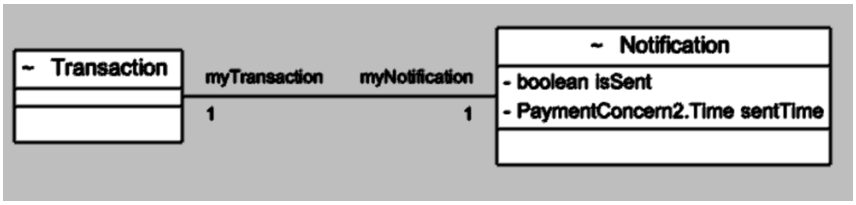
Sometimes, special groups of clients may enjoy discounts based on their identity, for example students.

Refund Domain Model: (first level sub-feature, optional)



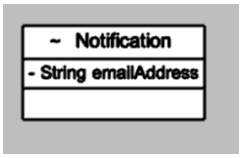
This feature determined whether a given system supported refund option or not. If a transaction is a refund, the direction of normal cash flow will be reversed.

PaymentNotification Domain Model: (first level sub-feature, mandatory)



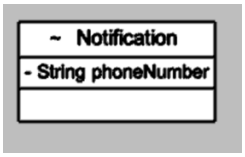
There is one-to-one relationship between each time of transaction and creating notification objects.

Email Domain Model: (second level sub-feature, XOR relationship under PaymentNotification feature)



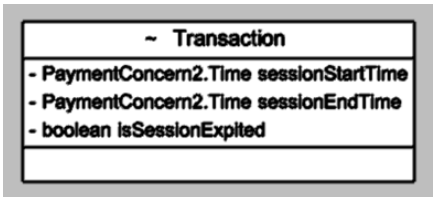
context Notification inv: self.emailAddress = self.myTransaction.buyer.emailAddress

MobileTest Domain Model: (second level sub-feature, XOR relationship under PaymentNotification feature)



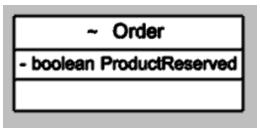
PostMail Domain Model: (second level sub-feature, XOR relationship under PaymentNotification feature)

InteractiveSession Domain Model: (first level sub-feature, mandatory)



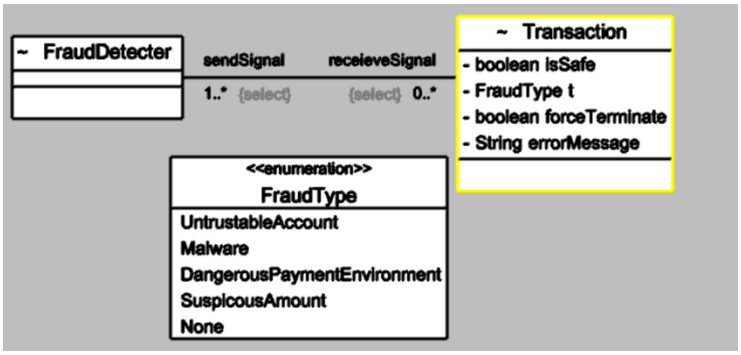
A GUI interface or webpage may be popped out during every transaction for user. If the time exceeds the sessionEndTime, but the transaction has not finished, the session will expire.

HoldingProduct Domain Model: (second level sub-feature, optional)



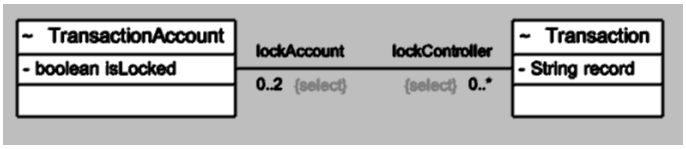
During its related Transaction's sessionStartTime to sessionEndTime, the target product will be reserved for this client only.

FraudDetection Domain Model: (second level sub-feature, mandatory)



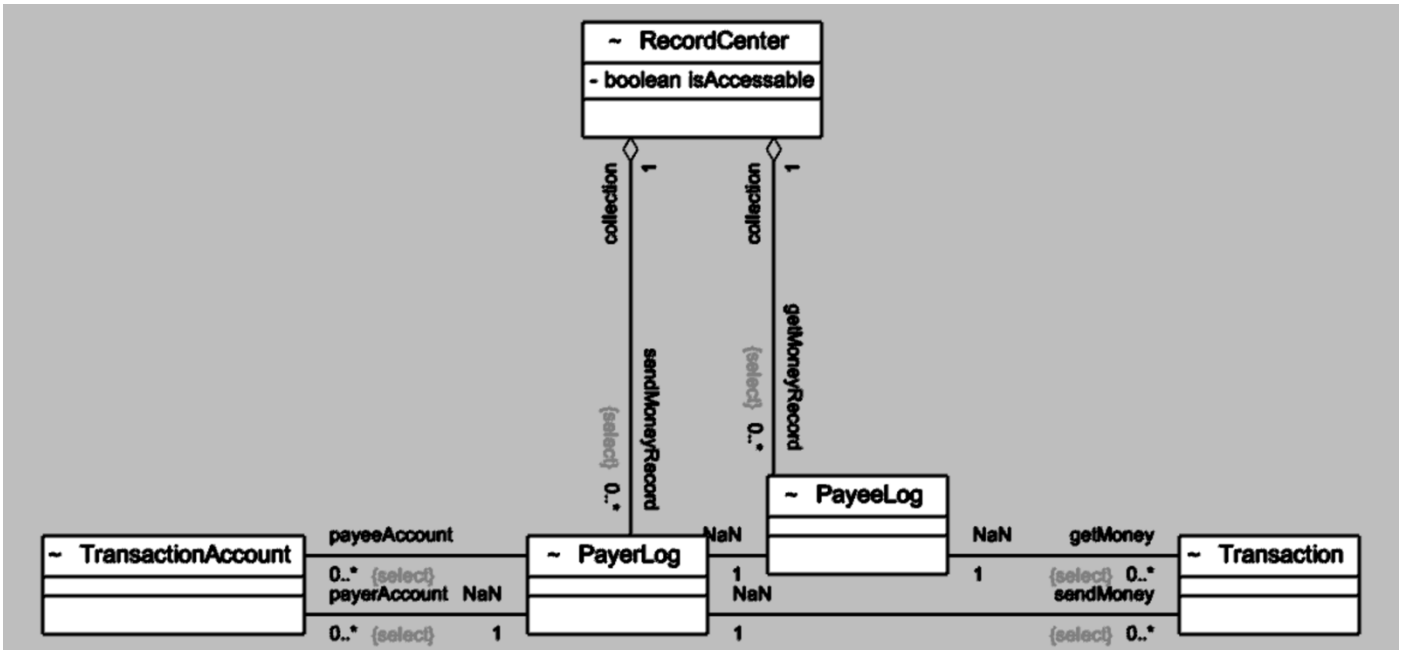
This feature is mandatory. A FraudDtector antivirus software is running to check multiple type of possible fraud and send resulting signal to Transaction objects. A transaction object may use more than one FraudDetectors, while FraudDetector themselves may exist without applying on any transaction process. A transaction may be forced to terminate if the session environment is not safe. And an error message should be printed out to user as explanation.

AccountLock Domain Model: (third level sub-feature, optional)



context Transaction pre: self.forceTerminate = true and self.isSafe= False inv self.isLocked = true

RecordHistory Domain Model: (first level sub-feature, optional)



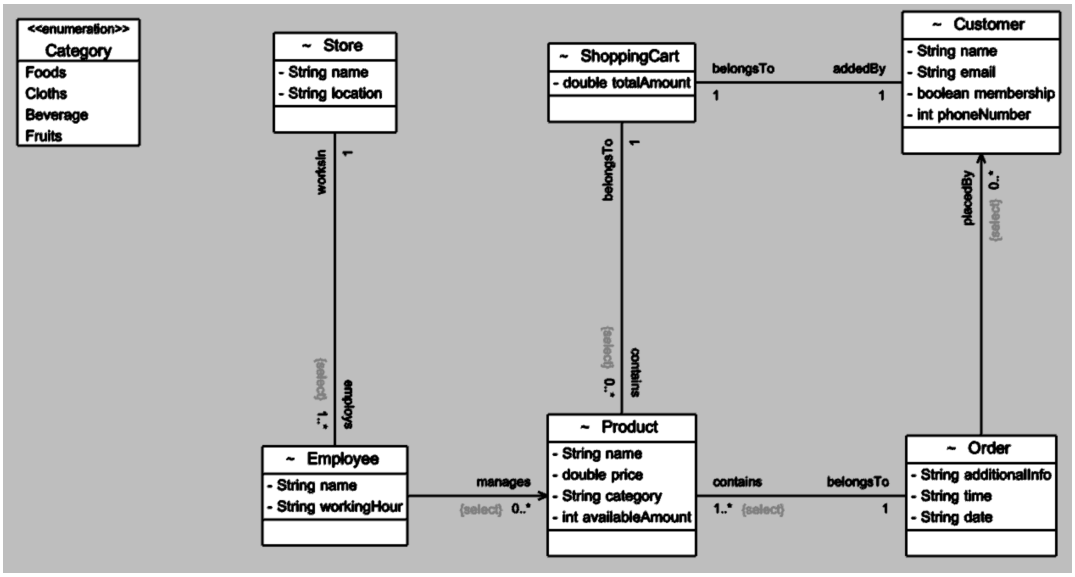
PayerLog and PayeeLog are two association class for two relationships between TransactionAccount and Transaction class. They are in charge of recording down the transaction history. These relationships between TransactionAccount and Transaction, interrupted by PayerLog and PayeeLog, corresponds to the same relationships with same roles' names at two ends depicted in the PaymentConcern domain model.

Note: since TouchCore7 cannot realize association class in design model, we alternatively build concrete classes interrupting relationships. I used 'NaN' on those segments ends since these ends should not exist at the first place.

Application:

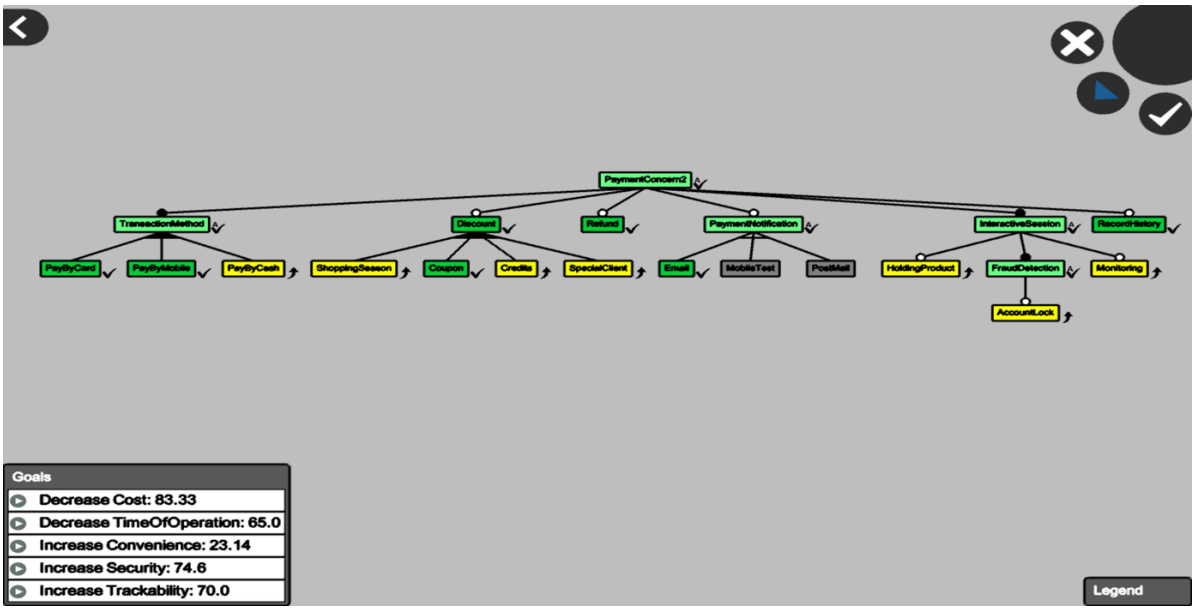
Overview:

This a simple online grocery application designed to demonstrate the reusability of the Payment Concern implemented in previous sections of Project 1.



Selection of Features:

Features are chosen in this sample application to strike a balance between cost and convenience. Cards and Mobile are accepted as valid payment methods. Coupons are selected as the only type of discount. Payment notifications will be pushed via email. Additionally, refund and record history functions are enabled.



Mapping Between Application and Reusable Concern:

As shown in the screenshot below, several classes of Payment Concern are mapped to the components in the application in order to achieve payment functionality. Customer class in the reusable concern will extend the existing Customer class in the demo application. Order and Store will facilitate the online grocery to handle payments. It should be noticed that a list of classes such as Transaction, Refund, and Notification will be automatically injected into the domain with these mappings.