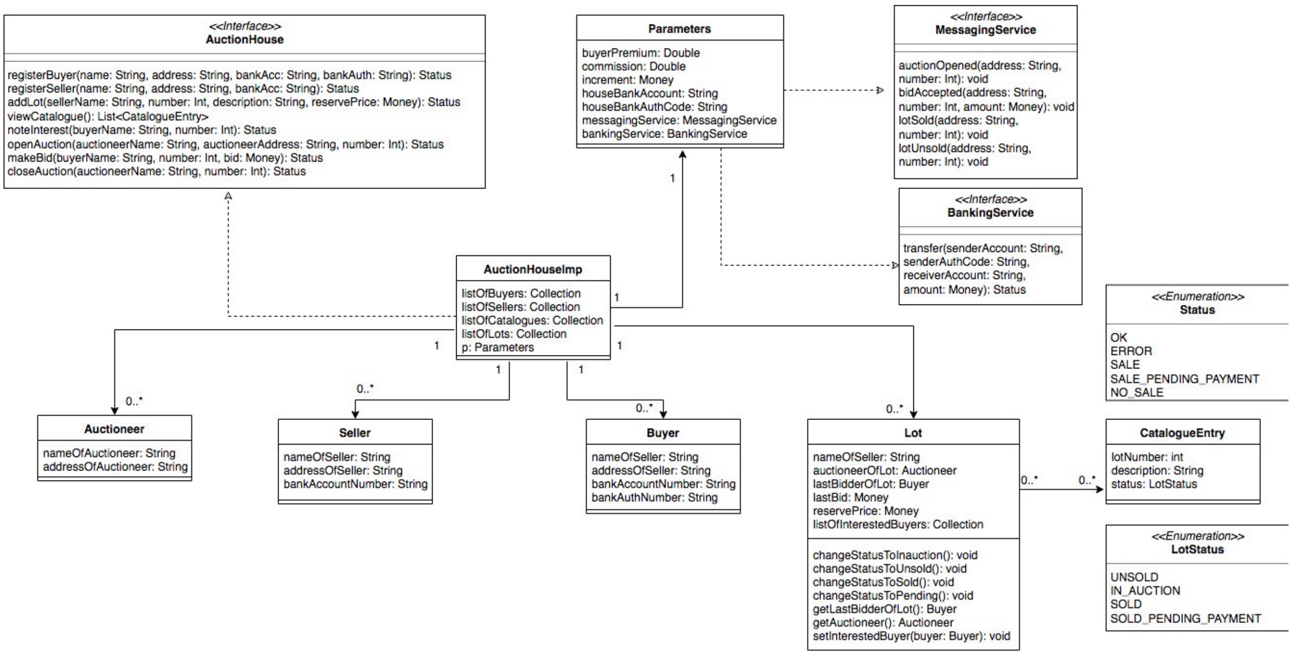
Name: Hanbyeol Joo

UUN: S1705270

UML Class Diagram


<<Interface>>
AuctionHouse
registerBuyer(name: String, address: String, bankAcc: String, bankAuth: String): Status
registerSeller(name: String, address: String, bankAcc: String): Status
addLot(sellerName: String, number: Int, description: String, reservePrice: Money): Status
viewCatalogue(): List<CatalogueEntry>
noteInterest(buyerName: String, number: Int): Status
openAuction(auctioneerName: String, auctioneerAddress: String, number: Int): Status
makeBid(buyerName: String, number: Int, bid: Money): Status
closeAuction(auctioneerName: String, number: Int): Status
Parameters
buyerPremium: Double
commission: Double
increment: Money
houseBankAccount: String
houseBankAuthCode: String
messagingService: MessagingService
bankingService: BankingService
<<Interface>>
MessagingService
auctionOpened(address: String, number: Int): void
bidAccepted(address: String, number: Int, amount: Money): void
lotSold(address: String, number: Int): void
lotUnsold(address: String, number: Int): void
<<Interface>>
BankingService
transfer(senderAccount: String, senderAuthCode: String, receiverAccount: String, amount: Money): Status
AuctionHouseImp
listOfBuyers: Collection
listOfSellers: Collection
listOfCatalogues: Collection
listOfLots: Collection
p: Parameters
<<Enumeration>>
Status
OK
ERROR
SALE
SALE_PENDING_PAYMENT
NO_SALE
Auctioneer
nameOfAuctioneer: String
addressOfAuctioneer: String
Seller
nameOfSeller: String
addressOfSeller: String
bankAccountNumber: String
Buyer
nameOfSeller: String
addressOfSeller: String
bankAccountNumber: String
bankAuthNumber: String
Lot
nameOfSeller: String
auctioneerOfLot: Auctioneer
lastBidderOfLot: Buyer
lastBid: Money
reservePrice: Money
listOfInterestedBuyers: Collection
changeStatusToInauction(): void
changeStatusToUnsold(): void
changeStatusToSold(): void
changeStatusToPending(): void
getLastBidderOfLot(): Buyer
getAuctioneer(): Auctioneer
setInterestedBuyer(buyer: Buyer): void
CatalogueEntry
lotNumber: int
description: String
status: LotStatus
<<Enumeration>>
LotStatus
UNSOLD
IN_AUCTION
SOLD
SOLD_PENDING_PAYMENT
1
1
1
1
1
1
0..*
0..*
0..*
0..*
0..*
0..*

High-level design description

Buyer, Seller, Auctioneer

- In my previous design, the classes **Buyer** and **Seller** were subclasses of the class **User**, and they inherited several attributes from the class **User**. However, in my current design, there is no such class User: I decided to eliminate it because there were fewer attributes for objects of **Buyer** and **Seller** than I previously thought and therefore there was no need for inheriting from another class. In my previous design, there were more attributes for objects of Buyer and Seller, such as zip, telephone and id, all of which are all eliminated in my current design. Another alternative design is to have only one class **User** and include all the attributes and operations of **Buyer** and **Seller** in it. This may have resulted in a simpler diagram, but this would result in no distinction between buyers and sellers in the auction system and therefore implementing the system would have been more difficult.

CatalogueEntry, Lot

- The class **Lot** inherits three attributes from the class **CatalogueEntry**, which are *lotNumber*, *description* and *status*. In comparison to the objects of **CatalogueEntry**, the objects of **Lot** have more attributes like *nameOfSeller*, *auctioneerOfLot*, *lastBidderOfLot*, *lastBid*, *reservePrice* and *listOfInterestedBuyers* and also many more methods. This is because while the objects of **CatalogueEntry** are mostly used to be browsed by users the objects of **Lot** are used in most methods in the class **AuctionHouseImp** and for the methods to be successful the objects of **Lot** need such attributes and operators. Aside from those methods noted in the class **Lot** in the UML class diagram, there are more getters and setters like *setAuctioneer()* or *getLastBid()*. Through the use of these getters and setters, the system can obtain an object of **Auctioneer**, list of **Buyer**s who noted interest, status of lot, last bidder, last bid, reserve price and many others, and it can also set Auctioneer, last bidder, last bid, status of lot and many others. This really makes it easy to implement the system because this allows the class AuctionHouseImp to obtain various information and therefore perform operations smoothly. The classes **Lot** and **CatalogueEntry** have a multiplicity of 0..* because there may be none or many, which we do not know how many.

AuctionHouse, AuctionHouseImp

- Most important operations are all performed in the class **AuctionHouseImp**, which inherits basic method structures from the class **AuctionHouse**. In my previous design, the methods *noteInterest()* and *makeBid()* were in the class **Buyer** and the method *addLot()* was in the class **Seller**. In my current design, these methods are all in the class **AuctionHouseImp**. Also, the methods *openAuction()* and *closeAuction()* were in the class **Auctioneer** in my previous design, but it is in the class **AuctionHouseImp** now. In the class, *listOfBuyers*, *listOfSellers* and *listOfLots* store all the buyers, sellers and lots. Many methods in the class require objects of Buyer, Seller or Lot to perform operations, and they can be obtained from such lists. All the methods in the class **AuctionHouseImp** except for *viewCatalogue()* require inputs. The method *viewCatalogue()* does not have any input because it is required for the system to allow viewing **CatalogueEntry**s to any one.

- Between all the associations between the class **AuctionHouseImp** and other classes, **AuctionHouseImp** has a multiplicity of 1 because there must be only one such main controller for this system; if there is more than one, they are likely to conflict with each other and it is difficult to distribute which responsibility to which and therefore such decisions will make the system more complicated. The classes **Auctioneer**, **Buyer**, **Seller** and **Lot** have a multiplicity of 0..* because there may be none or many, which we do not know how many.

Parameters, MessagingService, BankingService

- The classes **MessagingService** and **BankingService** are given interfaces. They ease the implementation of notifying **Buyer**s, **Seller**s and **Auctioneer**s of events like auctions opening or closing and also of transferring money from **Buyer** to **AuctionHouse** and from **AuctionHouse** to **Seller**. The class Parameters is a given class that makes the use of **MessagingService** and **BankingService**, and therefore the class **AuctionHouseImp** does not need to inherit these classes directly but only need to create an object of type **Parameters**. The class **Parameters** have a multiplicity of 1 because its attributes such as *buyerPremium*, *commission*, *messagingService* or *bankingService* must be consistent.

Status, LotStatus

- The classes **Status** and **LotStatus** are given enumerations. They are used in the class **AuctionHouseImp** because the system must report the status of the system every time it performs a particular operation and the status of lot is also needed to be checked to perform operations in the class.

Implementation decisions

- **TreeMap**<*Integer*, *CatalogueEntry*> is used to store objects of **CatalogueEntry** because the system has a requirement that the method *viewCatalogue()* must return the objects in increasing lot-number order. This is because **TreeMap** is always sorted based on keys: in the **TreeMap**<*Integer*, *CatalogueEntry*>, key is the lot number and the value is the object of type **CatalogueEntry**. Therefore, the objects of type **CatalogueEntry** returned by the method *viewCatalogue()* are in increasing lot-number.

- Instead of **TreeMap**, **ArrayList** is used to store **Buyer**s, **Seller**s and **Lot**s. This is because there is no requirement to the system that **Buyer**s, **Seller**s and **Lot**s should be placed in certain order. If **Buyer**s, **Seller**s and **Lot**s are stored in **Map**, such as **TreeMap** or **HashMap**, I could more easily obtain a particular **Buyer**, **Seller** or **Lot** in several methods without having to use several for loops to look for them. However, I decided to use **ArrayList** because the number of for loops I need to use to find a particular **Buyer**, **Seller** or **Lot** is mostly less than 3, which is very doable. Also, if I decided to use **Map** to store objects, I needed to give each *Buyer*, *Seller* and *Lot* a key, which means more codes and time. So, I thought that choosing **ArrayList** over **Map** was not a bad idea for coding this system.

- In the class **Lot**, **ArrayList** is used to store all the buyers who have noted interest in a particular lot. The **Buyer**s do not need to be placed in certain order, so I used **ArrayList** instead of **Map**.