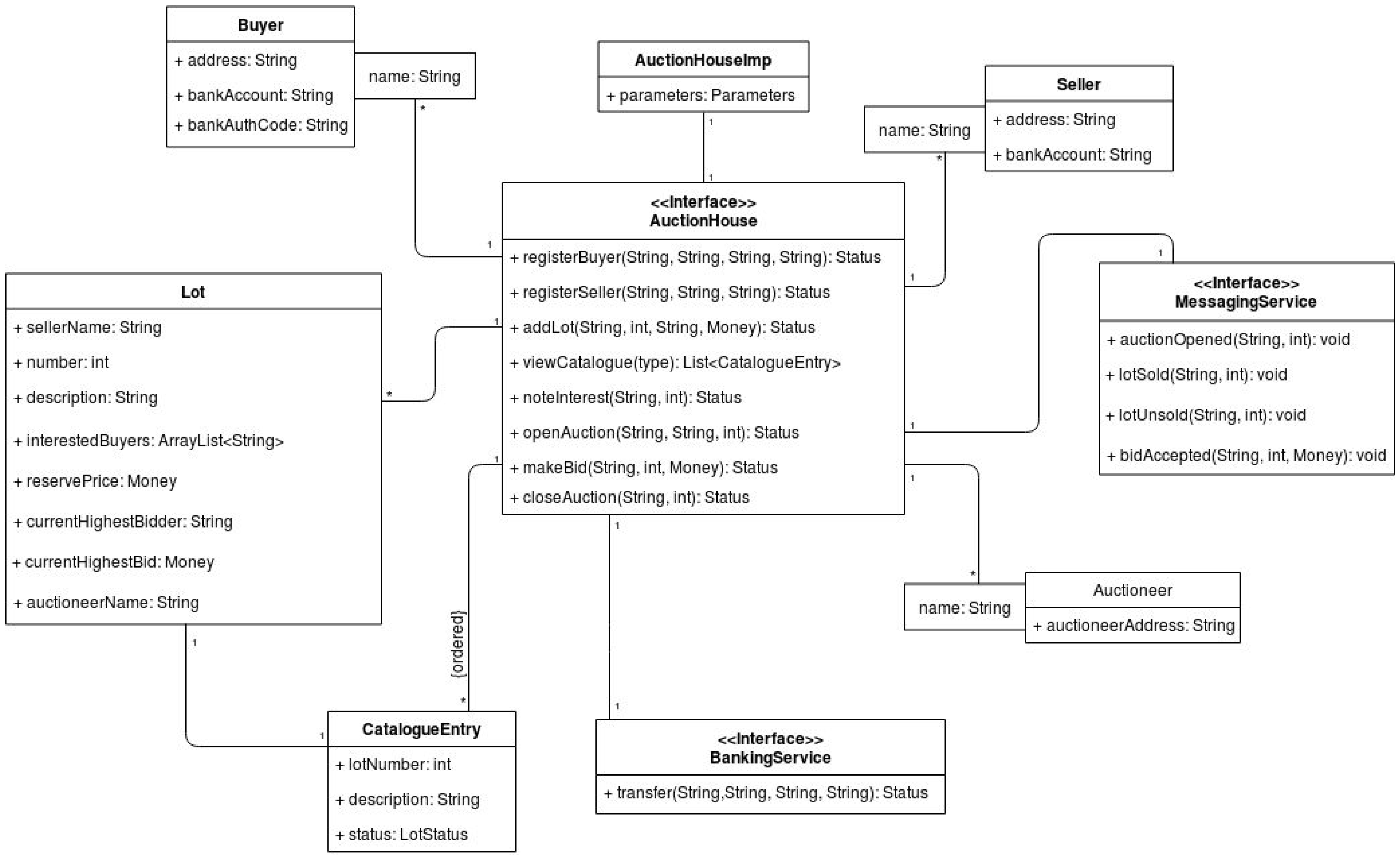
Inf2C Software Engineering 2018-19 Coursework 3

Creating an abstract implementation of a auction house system

Samir Farhat - s1746788
Tomasz Horszczaruk - s1758499

3.7.1 - UML Class Diagram


Buyer
+ address: String
+ bankAccount: String
+ bankAuthCode: String
name: String
*
1
AuctionHouseImp
+ parameters: Parameters
1
1
Seller
+ address: String
+ bankAccount: String
name: String
*
1
<<Interface>>
AuctionHouse
+ registerBuyer(String, String, String, String): Status
+ registerSeller(String, String, String): Status
+ addLot(String, int, String, Money): Status
+ viewCatalogue(type): List<CatalogueEntry>
+ noteInterest(String, int): Status
+ openAuction(String, String, int): Status
+ makeBid(String, int, Money): Status
+ closeAuction(String, int): Status
<<Interface>>
MessagingService
+ auctionOpened(String, int): void
+ lotSold(String, int): void
+ lotUnsold(String, int): void
+ bidAccepted(String, int, Money): void
Lot
+ sellerName: String
+ number: int
+ description: String
+ interestedBuyers: ArrayList<String>
+ reservePrice: Money
+ currentHighestBidder: String
+ currentHighestBid: Money
+ auctioneerName: String
*
1
1
{ordered}
*
1
1
CatalogueEntry
+ lotNumber: int
+ description: String
+ status: LotStatus
1
1
<<Interface>>
BankingService
+ transfer(String,String, String, String): Status
1
1
*
name: String
Auctioneer
+ auctioneerAddress: String

3.7.2 - High Level Description

The most important component of our UML design is the ‘AuctionHouse’ class, as this singleton class functions as the controller for the system, handling the majority of inputs from buyers, sellers and auctioneers. The class only has, methods, as all the arguments for those methods are received by other classes. This class is an interface, which means that it as defined methods that are implemented by some other class (in this case ‘AuctionHouseImp’). ‘AuctionHouseImp’ will be elaborated on in the next section of this report.
The ‘Buyer’ class holds the instance of a buyer, which may be used by ‘AuctionHouse’ to note interest in lots, register, bid, send messages, and send bank transfers. The relationship to ‘AuctionHouse’ is many to one, as multiple instances of buyer can be in relation with the auction house.
The ‘Seller’ class is used in a very similar way to buyer. ‘AuctionHouseImp’ uses the fields from seller to register sellers, register lots, send messages, and manage bank transfers. Once again ‘Seller’ has a many to one relationship with ‘AuctionHouse’ as multiple instances of sellers can interact with the auction house.
The ‘Auctioneer’ class is the simplest and least used class in the system. The fields of the class are simply used to send messages to the auctioneer of a lot when relevant. In this case only when a new bid goes through on an active lot. Multiple instances of Auctioneer can participate with the auction house, hence a many to one relationship.
‘Buyer’, ‘Seller’, and ‘Auctioneer’ use qualified association with the ‘AuctionHouse’ class based on their name. Which means that name is used as a key to refer to the instance of the ‘Buyer’, ‘Seller’, and ‘Auctioneer’ class.
The ‘Messaging Service’ class is a class that handles the notifying of interested parties on occurences during auctions. Once again it is an interface class, so its methods are defined in another class (MockMessagingService). It is important to note that the class uses the address argument in all its methods to be able to direct its messages. The ‘auctionOpened()’, ‘lotSold()’, and ‘lotUnsold()’ methods notifies all interested buyers and the seller of the lot. ‘bidAccepted’ is a bit more complicated, as it notifies the auctioneer every interested buyer except the buyer who made the actual bid. ‘MessagingService’, like ‘AuctionHouse’ is a singleton class, so it has a one to one relationship.
The ‘BankingService’ is another interface class that uses the ‘MockBankingService’ to define its methods. The class has a method ‘transfer()’ that takes banking information, uses it to make a bank transfer, and checks if the transfer went through. Depending on what happens the system will then proceed to update the statuses of the lot, and go to either ‘lotSold’ or ‘lotUnsold’ methods of ‘MessagingService’. ‘BankingService’ is another example of a singleton class, so it has a one to one relationship.
The way that the ‘Lot’ and ‘CatalogueEntry’ class function is different from the other classes in the UML. According to how the system works, the buyers and sellers can only look at logs through the catalogue entry. So every time a lot is created or updated, so must a catalogue entry be updated or created. This is the reason there is a one to one relationship between

them. Furthermore, this is also why the fields for catalogue entry include ‘lotNumber’, ‘description’ and ‘status’, which all come from the lot class. Both ‘Lot’ and ‘CatalogueEntry’ have a many to one relationship with ‘AuctionHouse’ since multiple lots and catalogue entries can interact with the auction house.
The ‘CatalogueEntry’ class has an ordered relationship to ‘AuctionHouse’, this means that any catalogue entry in the auction house is somehow ordered. In this particular case it is ordered according to their lot number(lowest to highest).
Regarding the ‘AuctionHouseImp’ class, it has a single field parameters. It uses parameters to assign certain values to the object of type AuctionHouse.

3.7.3 - Implementation Decisions

<u>Use of Collection Classes:</u>

For our implementation we only used the ArrayList and HashMap collection classes. HashMap was used when implementing the ‘viewCatalogue’ method. The reason for this is that we needed the catalogue to be ordered according to the lot number(lowest to highest). Keeping this in mind, we used the HashMap since it has elements ordered as (key, object). Storing lots into our catalogue entry, we used the lot number as a key. From there it was a simple matter of mapping lot number to catalogue entry, and performing the sort() method to organize the HashMap in the order we wanted it. We created an empty ArrayList, then went through the sorted keys and used a for loop to find their respective lot object, and add it to the Arraylist.
For ‘makeBid’, ‘registerSeller’ and ‘registerBuyer’, we created empty HashMaps mapping names to objects of type Auctioneer, Seller and Buyer respectively. We also created some ArrayLists of Strings which we would use to keep track of the names of interested buyers, interested buyers addresses, seller names and seller addresses. Using these we were able to retrieve objects, names, and addresses of interest for our implementation of several methods.

<u>Classes we created:</u>

For our implementation we created the classes ‘Buyer’, ‘Seller’, ‘Auctioneer’ and ‘Lot’ classes. All these classes carry fields that allow for our implementation. It allows us to keep track of registering sellers and buyers, their names and addresses, any lots the seller wishes to register, any bids and transactions of the buyers, and which auctioneer is assigned to each lot.