Inf2C Software Engineering 2018-19
Coursework 2
Creating a software design for an Auction House System

Diana-Andreea Tanase s1534228
Valentine Dragan s1710228

November 2018

1 Introduction

This document presents the design for the BidIT Auction House System. The System enables Auld Reekie Auction House to have an online catalogue of lots that can be browsed by the general public. In addition, by using the System during an auction, the Auction House wants to replace the traditional bidding way with bids submitted from electronic devices (smartphones or personal computers), allowing some Buyers to be off-site.

This document captures the UML diagram for the classes of System, together with the association between them. The document also presents dynamic models for the use cases. More information about the functionality of the System can be found in the coursework 1 handout. To learn more about the requirements for this design document, please refer to coursework 2 handout.

2 Static model

2.1 UML class model

This subsection presents the static model into two separate class diagrams. Figure 1 highlights the associations between the classes of the System. Figure 2 presents those classes along with their fields and methods. A high-level description of the classes is presented in the next subsection.

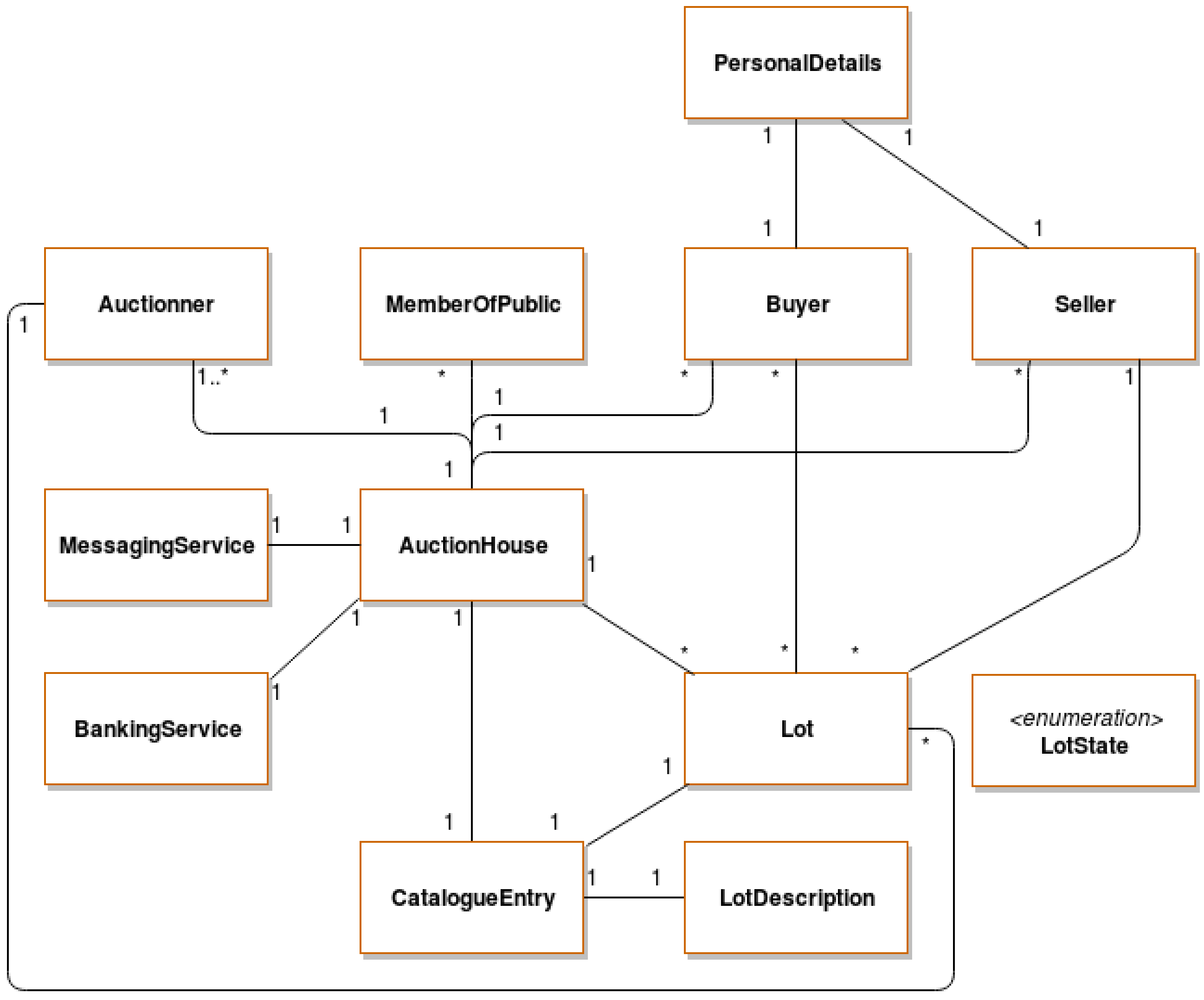


Figure 1: Class relationship diagram

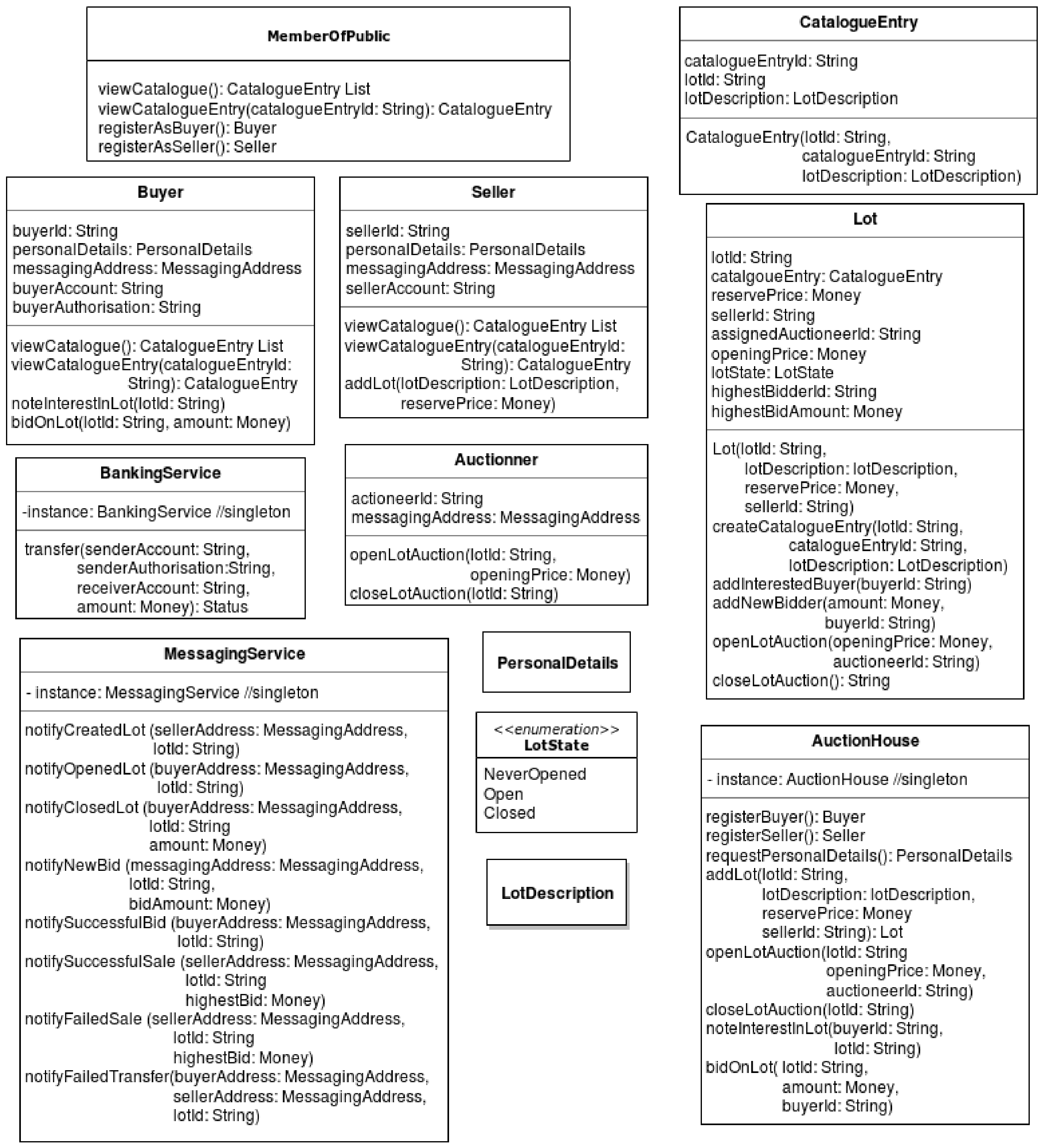

MemberOfPublic
viewCatalogue(): CatalogueEntry List
viewCatalogueEntry(catalogueEntryId: String): CatalogueEntry
registerAsBuyer(): Buyer
registerAsSeller(): Seller
CatalogueEntry
catalogueEntryId: String
lotId: String
lotDescription: LotDescription
CatalogueEntry(lotId: String, catalogueEntryId: String lotDescription: LotDescription)
Buyer
buyerId: String
personalDetails: PersonalDetails
messagingAddress: MessagingAddress
buyerAccount: String
buyerAuthorisation: String
viewCatalogue(): CatalogueEntry List
viewCatalogueEntry(catalogueEntryId: String): CatalogueEntry
noteInterestInLot(lotId: String)
bidOnLot(lotId: String, amount: Money)
Seller
sellerId: String
personalDetails: PersonalDetails
messagingAddress: MessagingAddress
sellerAccount: String
viewCatalogue(): CatalogueEntry List
viewCatalogueEntry(catalogueEntryId: String): CatalogueEntry
addLot(lotDescription: LotDescription, reservePrice: Money)
Lot
lotId: String
catalgoueEntry: CatalogueEntry
reservePrice: Money
sellerId: String
assignedAuctioneerId: String
openingPrice: Money
lotState: LotState
highestBidderId: String
highestBidAmount: Money
Lot(lotId: String, lotDescription: lotDescription, reservePrice: Money, sellerId: String)
createCatalogueEntry(lotId: String, catalogueEntryId: String, lotDescription: LotDescription)
addInterestedBuyer(buyerId: String)
addNewBidder(amount: Money, buyerId: String)
openLotAuction(openingPrice: Money, auctioneerId: String)
closeLotAuction(): String
BankingService
-instance: BankingService //singleton
transfer(senderAccount: String, senderAuthorisation:String, receiverAccount: String, amount: Money): Status
Auctionner
actioneerId: String
messagingAddress: MessagingAddress
openLotAuction(lotId: String, openingPrice: Money)
closeLotAuction(lotId: String)
MessagingService
- instance: MessagingService //singleton
notifyCreatedLot (sellerAddress: MessagingAddress, lotId: String)
notifyOpenedLot (buyerAddress: MessagingAddress, lotId: String)
notifyClosedLot (buyerAddress: MessagingAddress, lotId: String amount: Money)
notifyNewBid (messagingAddress: MessagingAddress, lotId: String, bidAmount: Money)
notifySuccessfulBid (buyerAddress: MessagingAddress, lotId: String)
notifySuccessfulSale (sellerAddress: MessagingAddress, lotId: String highestBid: Money)
notifyFailedSale (sellerAddress: MessagingAddress, lotId: String highestBid: Money)
notifyFailedTransfer(buyerAddress: MessagingAddress, sellerAddress: MessagingAddress, lotId: String)
PersonalDetails
<<enumeration>>
LotState
NeverOpened
Open
Closed
LotDescription
AuctionHouse
- instance: AuctionHouse //singleton
registerBuyer(): Buyer
registerSeller(): Seller
requestPersonalDetails(): PersonalDetails
addLot(lotId: String, lotDescription: lotDescription, reservePrice: Money sellerId: String): Lot
openLotAuction(lotId: String openingPrice: Money, auctioneerId: String)
closeLotAuction(lotId: String)
noteInterestInLot(buyerId: String, lotId: String)
bidOnLot(lotId: String, amount: Money, buyerId: String)

Figure 2: Class diagram

2.2 High-level description

2.2.1 Initial assumptions

This subsection presents some relevant assumptions that have been made for the design:

1. **PersonalDetails, LotDescription, Money, Status** and **String** are classes that are already provided and thus, we have decided not to represent *Money, Status, String* in the class diagram in Figure 2. Their name is self explanatory.
2. The data type **MessagingAddress** is provided to us, hence we have decided not to add further reference for it in the diagrams.
3. Most classes have standard "getters and setters", which have been omitted from the UML class diagrams for a cleaner diagram.
4. The visibility of the fields and methods of each class (whether they are public, private,package protected) is not presented in the UML diagram. This will be decided in the future, ideally after consulting with the stakeholders who will maintain the system.
5. Navigation arrowheads have not been added to the associations, as suggested in the coursework handout.

2.2.2 Classes description

MemberOfPublic consists of the basic functionality to browse through the entries in the catalogue of the Auction House and read the description of one selected entry. Any general member of public can register with the System to become a Buyer or a Seller, as stated in the requirements of the System. Hence, a MemberOf-Public only needs to communicate with the Auction House, either for registering or viewing catalogue entry(s). We have assumed there can exist 0 or more members of public.

Buyer can browse through the entries in the catalogue of the Auction House and read the description of one selected entry. A buyer can also note interest in a lot and bid on a lot, by communicating with the Auction House. Each buyer must also have personal details and bank details stored. We have assumed there can exist 0 or more Buyers. Each buyer has a unique buyerId.

Seller can browse through the entries in the catalogue of the Auction House and read the description of one selected entry. In addition, seller adds a new lot, with its lot description and reserve price. This new lot is added into the System by the Seller communicating with the Auction House. Each seller must also have personal details and bank details stored. We have assumed there can exist 0 or more Sellers. Each seller has a unique sellerId. A Seller can add 0 or more Lots.

Auctioneer opens or closes a bid by communicating with the **AuctionHouse**. We have assumed there must exist at least one Auctioneer. An auctioneer can assist with 0 or more lots, while one lot can be opened and closed by only the same Auctioneer. (so each lot has an assigned Auctioneer).

PersonalDetails is a class provided to use. We have not added further details about the fields it contains. This information can be decided or given to us at the implementation stage. We have decided to keep PersonalDetails separate from any information related to bank details for security. Each Buyer and Seller must have exactly one PersonalDetails object, while each PersonalDetails must belong to exactly one Buyer or Seller. This means one particular PersonalDetails object cannot be shared between different users.

MessagingService is a singleton. Its functionality is essential for the System. It sends out messages (which we sometimes call *notifications* as well) to actors modelling people acting in particular role (buyers, sellers, auctioneer). Because notifications are directed to real actors, we have decided not to connect the MessagingService with the actors' objects in our UML diagram. Each method has an argument of type MessagingAddress for specifying who the message should be sent to, and different arguments for the message being sent. The address is given to the MessagingService by the **AuctionHouse**, forwarded from the fields of the corresponding user class. Each notification should send out a String message. To avoid the possibility of unusual notifications being sent to users, we assumed that these messages are fixed Strings inside MessagingService, to which arguments are appended. For example, a message could be of the format "The auction for the following lot has been closed: " + lotId. We provide below some brief details about the notifications the service sends. Please note that we have chosen to design these notifications only, however a real system might want to send more. Additions could be made after further discussions with the stakeholders.

- *notifyCreatedLot.* The Seller person receives a notification that the new lot has been added into the System.
- *notifyOpenenLot, notifyClosedLot.* The interested buyers are notified of the action of the auctioneer. We have decided not to send a formal notification to the auctioneer as well. Since the auctioneers are the one who trigger the event, they might be able to see, for example, a success message returned from their event.
- *notifyNewBid.* Each interested buyer is notified that a new bid has been made for the lot of interest.
- *notifySuccessfuBid.* The winner of auction is notified that he/she has won the bid.
- *notifySuccessfulSale.* The Sellers are notified that their lots have been sold.
- *notifyFailedSale.* The Sellers are notified that their lots have not been sold, as the reserve price has not been attained. Again, we decided for simplicity that there is no need to inform the auctioneer in this case, as his job is to close the auction even if the highest bid is lower than required.
- *notifySuccessfulSale, notifyFailedSale.* The highest bidder (winner of current auction) and the seller of the corresponding lot are notified if the bank transaction succeeded or failed.

ActionHouse is the central component that all input messages are directed to. The inputs come from *MemberOfPublic*, *Buyer*, *Seller* or *Auctioneer*, when triggered by actors interacting with the System. The inputs are usually forwarded to suitable other classes. The functionality is described below. Please note that the **AuctionHouse** makes calls to getters and setter of other objects to retrieve relevant data for different arguments.

- *requestPersonalDetails* is used to supply the details required when a *MemberOfPublic* wants to register as Buyer or Seller. How the System requests these details is an implementation decision, but we consider using a registration form.
- *registerBuyer* and *registerSeller* first call the *requestPersonalDetails* to supply the required details. It then generates a new ID and forwards the call to the *Buyer* or *Seller* class constructor. We assume that after the new Buyer or Seller are created, the **AuctionHouse** saves the new object in a data structure (could be list, map, dictionary) to keep track of them and their IDs.
- *addLot* generates an ID for the new lot, and forwards the message to the *Lot* class constructor. We assume that after the new Lot object is created, it is saved in a data structure in the **AuctionHouse**.
- *openLotAuction* sends a call to the corresponding *Lot* to set its state to Open and set its opening price.
- *closeLotAuction* sends a call to the *Lot* to obtain the Seller, highest bidder, hammer price, and close the Lot. Then, it obtains the bank details from the corresponding Buyer and Seller. After checking that the hammer price is higher than reserve price, it sends a call to *BankingService* to perform the transfer. If the transfer is successful, the **AuctionHouse** sends a call to the MessagingService to notify the Buyer and Seller of their success. Alternative scenarios are that the MessagingService notifies the Buyer and Seller of an unsuccessful transfer, or notifies the Seller of a failed sale, due to hammer price being too low.
- *noteInterestInLot* forwards the message to the corresponding Lot, who adds a new interested buyer to its list.
- *bidOnLot* forwards the message to the corresponding Lot, who sets its new highest bidder and amount. It then interacts with the MessagingService to send notifyNewBid notifications to all interested buyers of the bid.

BankingService is another singleton, its job is to take care of the money transfers between the sender and receiver (buyer and seller). The transfer is trigger by the **AuctionHouse** after an Auctionner closes the auction for a lot. Each transfer returns a Status and we have assumed a transfer's Status can be fail or success.

Lot is an important class in our design. Every Lot also has an associated unique CatalogueEntry, which only stores information relevant for buyers and is detailed in the next enumeration item. A Lot stores its unique id, its seller and assignedAuctioneer (from the time its auction opens), its current State (whether in auction or not) and the information related to its auction: opening price highest bidder, highest bid amount. In addition, we have decided to save a list of interested buyers as well. When the constructor of a lot is called, it also creates with *createCatalogueEntry* its associated catalogue entry and sets the state to NeverOpened. On every new bid for the lot, a new bidder is added (the lot can save all its bidders in a list) with *addNewBidder*. We can assume this method updates the highestBidderId and highestBidAmount fields. *openLotAuction* and *closeLotAuction* update the lot's state.

CatalogueEntry is associated with a Lot. It only consists of a LotDescription. We have decided to have these two separate classes (Lot and CatalogueEntry) to avoid mixing information related to auction only with the description all users can see. Only a CatalogueEntry is accessible when a user is viewing (browsing) through the System's entries.

LotState is an enumeration for the state a lot can be in: *NeverOpened*, when it was never opened for auction, *Open*, if it is currently open for a present auction, and *Closed* if its auction has ended. Every state has a LotState associated with it. However, a state can correspond to more lots (as there can be more lots closed at the same time, for example). It did not make sense to add a relationship between the Lot and LotState representations in the Class relationship diagram.

LotDescription is a class provided to us. It is meant to represent textual description and graphic images, but no further information is provided on its functionality. Every CatalogueEntry has an associated LotDescription object and each LotDescription object is unique to one CatalogueEntry object. We have not considered necessary to save the LotDescription in the Lot class, as a Lot can access this information from its associated catalogue entry. Since the Lot class is mostly for System use, it wouldn't have been useful to store the LotDescription in it.

Money is a class provided to us. We do not have further information on what methods are accessible or what value it corresponds to. We have made every field related to a financial value (such as amount bidded) of Money type.

String is a class provided to us. No further information is necessary and it can be specific to the programming language used.

Status is a class provided to use and there is no information on what fields it consists of. It represents the Status of a transaction. Since the same Status (success/failure) can be specific to more transactions, we decided not to show any relationship between Status and BankingService. We considered that this would have made the relationship diagram too complex.

2.2.3 Further clarifications

This subsection presents further information on some of the design decisions.

- We have considered having separate **BuyerInfo** and **SellerInfo** classes as suggested in the coursework handout. However, we considered that they would make the design look too congested (especially as they would either have duplicate fields or inherit from a superclass UserInfo) and it was cleaner to only add the required fields in the Buyer and Seller classes.
- Since both **Seller** and **Buyer** have the viewCatalogue and viewCatalogueEntry functions, we have considered inheriting these from **MemberOfPublic** (by making MemberOfPublic) interface. However, we are interested in the registration functionality to be accessed by a general member of public only. The *registerAsBuyer/registerAsSeller* methods would have been accessible to the sub-classes Buyer and Seller and we did not desire such a case to be possible.
- We initially wanted to create a **BankAccount** class to store the data related to a bank account for a user. However, we decided to go with only a *buyerAccount/sellerAccount* String field, to avoid having too many classes with little or not significant functionality that would result in the diagrams looking too complex. If in the future the stakeholders wish that the bank details are more secure, we could create a BankAccount class with encrypted information and link it to each Buyer/Seller.
- It is important to understand that the **Buyer, Seller, Auctionner, MemberOfPublic** classes are for storing data, and also a way to route actions coming from outside the system to the **AuctionHouse** and map the real actors to the actions they can perform. Each buyer actor must have an account with

the System and the **AuctionHouse** can save this as Buyer object. Although the methods for actors' actions could have been only represented in the **AuctionHouse**, we felt that routing them through the corresponding objects would make the design easier to understand and closely related to the functional requirements of the System.

- We assumed that the **AuctionHouse** class will generate unique IDs for new objects being created by the methods *registerBuyer*, *registerSeller* and *addLot*. How it generates those IDs is an implementation decision, but a suggestion would be to count them from 1 onwards.
- We have considered creating a RegisteredBuyers and RegisteredSellers class to store the list of all Buyer and Seller IDs, but instead we ended up saving this data in the **AuctionHouse** for simplicity of design.
- We have considered creating a Catalogue class that would only store all the catalogue entries accessible to the readers. Again, to avoid adding redundant classes in our UML design, we decided to go with these values stored as a private field in the **AuctionHouse**.
- By having two separate Lot and CatalogueEntry classes, we wanted to address the problem of security as well, by ensuring that the readers of an entry cannot have access to any data related to an auction. Moreover, a Lot can access its lotDescription through its catalogueEntry, so there was no need to duplicate this information.
- We decided not to show the **Money, Status, String** classes in the two UML diagrams as they would not have added much significant information about the design.

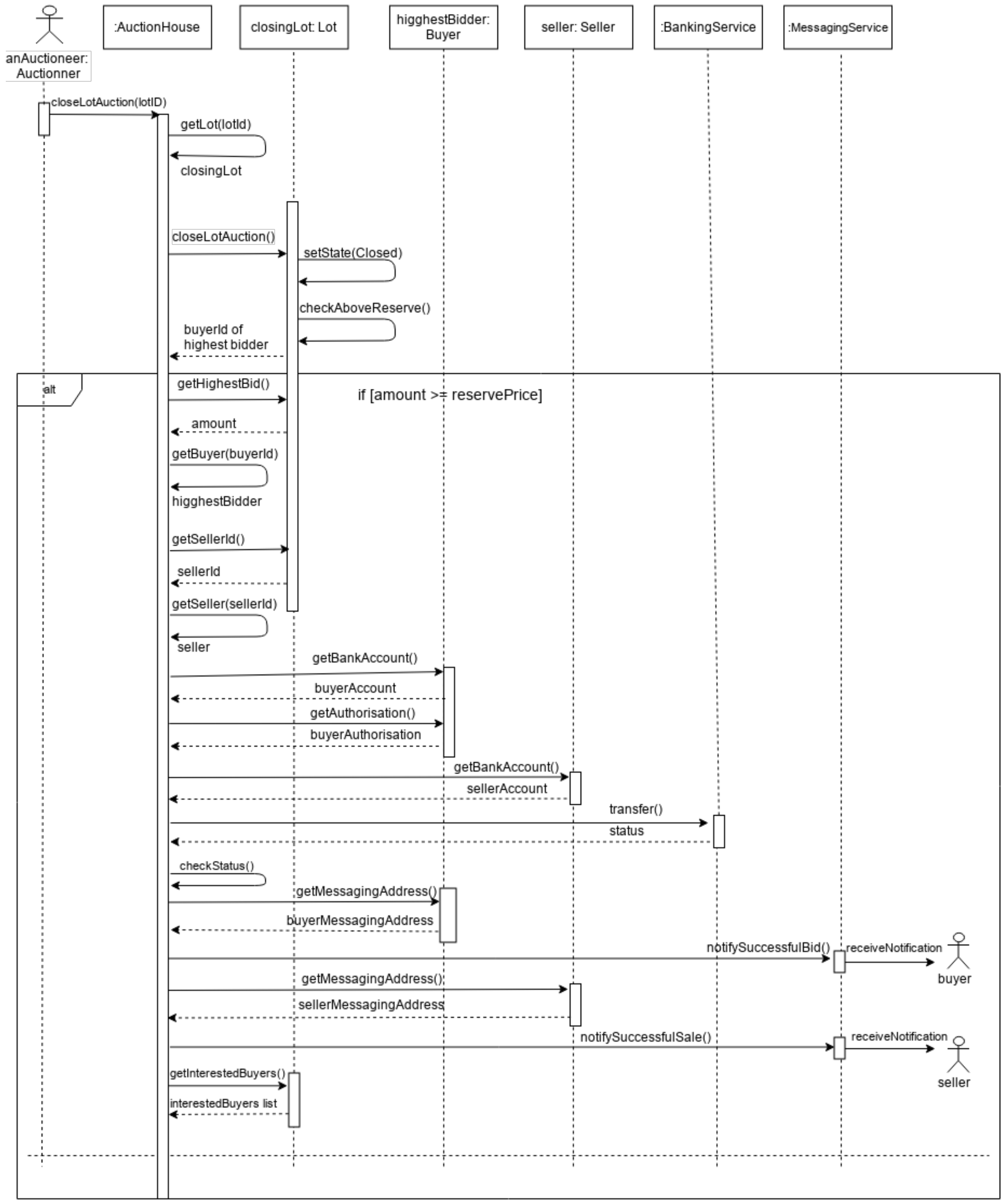
3 Dynamic models

3.1 UML sequence diagram

This section presents in Figures 3 and 4 the UML sequence diagram for the **Close Lot** use case. The use case is presented in the first report that captures the requirements of the System. Please see the following notes and assumptions:

- Some methods that will appear in the sequence diagram may not be present in the UML classes. This decision was made because they were self-explanatory (such as getters and setters, or if statements).
- We have assumed that there will exist a way in the Lot class to check if the higghestBidAmount is above the hammerPrice. Whether this check is done with a simple if statement in the *closeLotAuction* method or through a separate private method is irrelevant. We have decided not to write this in the UML classes diagram in Section 2, as this was an implementation decision and would have made the diagrams too detailed.
- What the Lot returns after the close method is left for discussion.At this stage, we have decided to return a String for the highestBidderId, that is null if the bid did not reach the reserve price. This would make the next actions of the use-case easier, but implementation can be changed easily.
- We have assumed as well the *Buyer's Premium* and *Seller's Commission* can be computed easily before the transfer method, without the need to add extra methods in the class diagrams or extra steps in the sequence diagram.
- For this sequence diagram, we have only presented the case where the bank transaction is successful. The coursework handout suggests to include the financial transactions if the lot *is sold*, which we have assumed is equivalent to a *successful transfer*. To extend the diagram for the fail case, we would need to add an alternative set of steps to send a *failedTransfer* notification to the parties involved (buyer or seller).
- As mentioned in Section 2.2.2, we have decided not to send the Auctioneer any message using the MessageService related to the state of the sale/transfer.

Use case summary: The Auctioneer closes the auction for a lot. The lot's state changes to Closed. The highest bid is compared against the reserve price. The bank transfer is made and buyer and seller are informed. All the interested buyers receive a notification.



[continues on the next page]

Figure 3: Sequence diagram

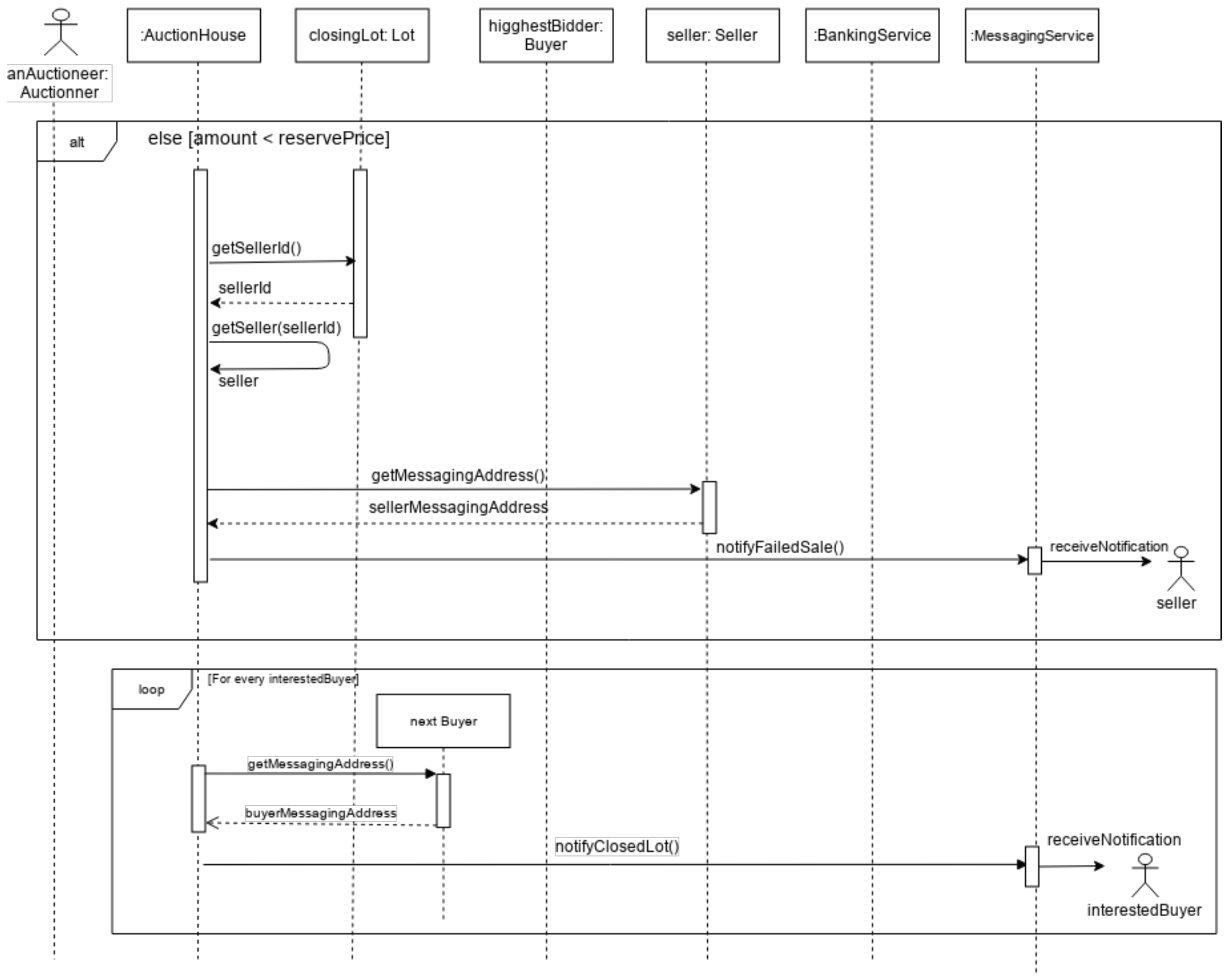


Figure 4: Sequence diagram (continuation)

3.2 Behaviour descriptions

This subsection presents a textual description of the *Add lot, Note interest in lot* and *Make bid* use cases. Please note that some steps involve adding items to a list (when by list we refer to a collection of objects rather than a concrete List data structure) that might not appear in the class diagram in Section 2. We are of the opinion that these instructions, while self-explanatory, would have made the diagrams in Section 2 too detailed. However, they are essential now for understanding the use cases.

- **Add lot**

  Summary: The Seller adds a new lot with its description to the System. A new catalogue entry is created for the new lot. The System saves the new objects.

```
aSeller –– addLot(lotDescription , reservePrice , sellerId) – –> AuctionHouse
  AuctionHouse –– generateNewLotId()  ––> AuctionHouse
  AuctionHouse <–– newLotId –– AuctionHouse
  AuctionHouse  –– new –> Lot
    Lot –– new –> CatalogueEntry
    Lot <–– newCatalogueEntry –– CatalogueEntry
  AuctionHouse <–– newLot –– Lot
  AuctionHouse –– addLotToList(newLot) ––> AuctionHouse
  AuctionHouse ––  getCatalogueEntry() ––> newLot
  AuctionHouse <–– newCatalogueEntry –– newLot
  AuctionHouse –– addCatalogueEntryToList(newCatalogueEntry) – –> AuctionHouse
  AuctionHouse -- getMessagingAddress() --> aSeller
  AuctionHouse <-- sellerMessagingAddress -- aSeller
  AuctionHouse -- notifyCreatedLot(sellerMessagingAddress) - -> MessagingService
    Seller (actor) <-- receive notification -- MessagingService
```

- **Note interest in lot**

  Summary: Buyer notes its interest in the selected lot. System adds Buyer to the list of Buyers interested in the lot.

```
aBuyer -- noteInterestInLot(lotId, buyerId) --> AuctionHouse
  AuctionHouse -- getLot(lotId) --> AuctionHouse
  AuctionHouse <-- interestedLot -- AuctionHouse
  AuctionHouse -- addInterestedBuyer(buyerId) --> interestedLot
```

- **Make a bid**

  Summary: Buyer informs the System that they would like to make a bid on a lot. They make a *jump bid.* System sets the lot's highest bidder to Buyer and updated the highestBid amount and notifies all interested Buyers of the new bid. The Auctioneer is informed about the new bid as well.

```
aBuyer -- bidOnLot(lotId, amount, buyerId) --> AuctionHouse
  AuctionHouse -- getLot(lotId) --> AuctionHouse
  AuctionHouse <-- biddedLot -- AuctionHouse
  AuctionHouse -- addNewBidder() --> biddedLot
    biddedLot -- setHighestBidderId() --> biddedLot
    biddedLot -- setHighestBidAmount() --> biddedLot
  AuctionHouse -- getInterestedBuyers() --> biddedLot
  AuctionHouse <-- interestedBuyers list -- biddedLot

LOOP [For each interestedBuyer]
  AuctionHouse -- getMessagingAddress() --> interestedBuyer
  AuctionHouse <-- buyerAddress -- interestedBuyer
  AuctionHouse -- notifyNewBid(buyerAddress, lotId, amount) --> MessagingService
    Buyer (actor) <-- receive notification -- MessagingService
END LOOP

  AuctionHouse -- getAssignedAuctioneerId() --> biddedLot
  AuctionHouse <-- auctioneerId -- biddedLot
  AuctionHouse -- getAuctioneer(auctioneerId) --> AuctionHouse
  AuctionHouse <-- assignedAuctioneer -- AuctionHouse
  AuctionHouse -- getMessagingAddress() --> assignedAuctioneer
  AuctionHouse <-- auctioneerAddress -- assignedAuctioneer
  AuctionHouse -- notifyNewBid(auctioneerAddress, lotId, amount) --> MessagingService
    Auctioneer (actor) <-- receive notification -- MessagingService
```