# Inf2C: Software Engineering
# Coursework 2

# Creating a software design for an auction house system

Michael Andrejczuk s1703773
Dylan Joseph Thinnes s1737075

November 5, 2018

# Contents

**1 Introduction** **2**

**2 Static model** **2**
2.1 UML class model . . . . . 2
2.2 High-level description . . . . . 2

**3 Dynamic models** **7**
3.1 UML sequence diagram . . . . . 7
3.1.1 Close auction . . . . . 7
3.2 Behaviour descriptions . . . . . 7
3.2.1 Add lot . . . . . 7
3.2.2 Note interest . . . . . 8
3.2.3 Make bid . . . . . 9

# 1 Introduction

We provide a software design for an auction house system, codenamed *Auctionista*. Our design is consistent with the requirements design undertaken in Coursework 1. We provide UML class models for the key actors involved in the system, and sequence diagrams and behaviour descriptions for use-cases identified in Coursework 1. For further details, please refer to the specification for Coursework 2.

# 2 Static model

## 2.1 UML class model

Due to its size, this is provided as the last page of the document.

## 2.2 High-level description

Firstly, an overview on a per-concept basis:

- **Actor:** The actors in the system — Seller, Buyer, Auctioneer, MemberOfPublic — are represented as implementations of the Actor interface. This means that:
  - By separating interface and implementation, we follow good design principles.
  - The Actor class ensures a consistent interface for message exchange using the MessageService singleton.
  - We can encapsulate the handling of messages of different kinds on a per-object basis, making it easy to maintain and add more message types as needed.
  - As each implementation of Actor will have a different way of handling various receiveMessage() instances, we leave these up to the classes to implement, rather than specifying them in Actor. This ensures good cohesion.
- **MessagingService:** Messages are sent and received only for objects which implement the Actor interface. In the case of this system, all messages passed around the system will relate to a Lot. This means that we only define methods for Actors to receive messages which pertain to a given Lot and some change in status (see point below). However, we can easily create new receiveMessage() functions for Actors which take different parameters; this ensures maintainability.
- **LotUpdateMessage:** This class specifies all the possible state-changes a lot may undertake. By providing these states as an enum, we ensure maintainability and modularisation. Particularly, in adding a new possible state to the enum all the classes which use this enum see the update, rather than needing to update each individual class.
- **RegisteredUser:** This superclass represents the actors who have an account on the system, allowing us to separate them from members of the public, who we assume

to not have accounts. Using inheritance, we ensure consistent behaviour for these users. Currently, this is only ensuring that all have a username: however, this is easily extensible to also include, for example, passwords and two-factor-authentication.

- **Client:** A subclass of RegisteredUser, this represents Sellers and Buyers. These two actors use the system in a similar way, so we ensure a similar implementation by combining them into a Client superclass. In particular, we ensure that Sellers and Buyers both have personal and banking details set for their accounts. We considered having both Buyer and Seller inherit from RegisteredUser, as Auctioneer does, but decided that the behaviour of these two classes needed to be enforced to be similar enough to warrant a superclass.

- **BankingDetails:** This class is excluded from the diagram for brevity: we treat it as analogous to the PersonalDetails class, containing the user's account and authorisation details, as needed to make transactions. We chose to separate BankingDetails from RegisteredUser for loose coupling and high cohesion of the components. This way, RegisteredUser does not need to care if the implementation of BankingDetails changes. This could be particularly useful if we decide we need to validate banking details before a user can register: these changes can be handled within the BankingDetails class, which ensures loose coupling as logically bank validation logic may not necessarily belong in a User class.

- **Seller:** This represents a Seller actor. We note the following:

  - An instance of the Seller class has some $n$ lots associated with it, the lots it owns. It makes sense to have this stored as an attribute of the seller: it allows for faster retrieval of the lots associated with a seller, rather than the alternative of polling all lots in the system to check which belong to that seller.
  - A Seller adding a lot calls the addLot method on itself. This method then calls AuctionHouse.addLot, which creates the lot and adds it to the AuctionHouse's list of lots. This is then returned to the Seller who adds it to the list of their lotsOwned. This implementation choice means that the Seller object need not concern itself with exactly how a lot is created, leaving the lot creation logic up to Auctionhouse.
  - We note that the addLot method only needs the LotInformation (described later) and a unique ID associated with the lot, which as per the specification is given to the Seller ahead of time. All other information about a lot can be inferred. However, we note that there is no way given on how an auctioneer is assigned to a lot: we assume this is handled by the auction house staff themselves and do not include how this may be done in the class diagram.
  - The receiveMessage method only includes support for when the LotUpdateMessage is LotSold, wherein the system informs the Seller that their lot has been sold. We assume other LotUpdateMessages are disregarded, but for the purposes of maintainability, maintain the ability to receive them: for example, it may be desirable in the future for the Seller to be informed when bids are placed on their lot.

- **Buyer:** We note the following:
  - Like the Seller class, we maintain a list of lots associated with the Buyer. This has the same benefit of not needing to poll all lots to see which lots the Buyer has marked themselves as interested in every time we wish to display these lots.
  - As per the specification, we have a viewCatalogue() function which returns a List of CatalogueEntries, for the user interface to handle as needed. By separating the retrieval logic from the display logic, we ensure high cohesion.
  - The bidOnLot() method only takes an argument of Bid, as the lot it is associated with can be inferred from the bid object.
  - Both bidOnLot() and markInterestInLot() call the AuctionHouse methods makeBid() and noteInterest() respectively. This means the Buyer object only communicates with the AuctionHouse singleton, which delegates messages and updates its own internal state as needed; particularly, it means the Buyer class does not need to be concerned with the interfaces other objects provide and how they may change.
  - The receiveMessage() method includes support for all three LotUpdateMessage options:
    1. BidPlacedOnLot, so the Buyer is informed when a lot they are interested in has had a bid placed on it;
    2. LotOnAuction, when the lot a Buyer is interested in opens for auction;
    3. LotSold, when the lot a Buyer is interested in is sold. From the Lot object itself we can work out if the Buyer was the winner of this Lot, and change the information presented to the end-user as needed. We originally had three options for this — LotSold, LotSoldAndWon, LotSoldAndLost — for the seller, winning buyer, and losing buyers respectively, but this information can be inferred from the Lot object so including separate enums for this is unnecessary.
- **Auctioneer:** We note the following:
  - An Auctioneer has a set of AssignedLots, which is updated by an implied getter / setter not mentioned on the class diagram for brevity.
  - The Auctioneer has the ability to open and close a lot, delegating the logic of this to the AuctionHouse class.
  - When closing a lot, the auctioneer does not need to specify a winning bid as this can be inferred from the Lot object at the time of the auction closing.
  - The Auctioneer is informed when a bid is placed on a lot they are in charge of (BidPlacedOnLot).
- **Member of Public:** This user has very restricted privileges, being able only to view catalogue entries. We note that it is an implementation of the Actor interface: this is primarily for maintainability purposes, in case it is decided that members of the public

may be allowed to perform more actions — for example, marking interest in a lot and receiving updates when bids are placed on it.

- **Lot:** We note the following:
  - A Lot has a unique ID, provided by the Seller on creation. This can act as a primary key when storing information about this object in a database.
  - We abstract the LotInformation and LotDescription attributes to separate classes, assuming nothing in particular about LotDescription.
  - A Lot stores a list of interested buyers associated with it, simplifying the process of notifying interested buyers when a bid is placed on it or when bidding is opened or closed.
  - We associate a CatalogueEntry with a Lot: we assume this CatalogueEntry is created by AuctionHouse in some internal logic which we omit for brevity.
- **LotInformation:** This provides the (mostly) static data associated with the lot. While it would also make sense to store all this information in the Lot object, it is cleaner to abstract it to a separate class. By doing this, we separate the attributes of Lot which often change from those which are set at creation, which simplifies the interpretability of the Lot class and separates its properties clearly into different roles.
- **LotStatus:** The Lot Status is, as per Coursework 1, the current state of the Lot. We introduced this state for two reasons:
  1. It allows for error-checking: a lot cannot be opened unless it is in a pre-auction state, and closed or bid on unless it is currently on auction.
  2. This provides a convenient method of archiving lots by separating them into those that are still active and those that have been sold and therefore are inactive. This could, for example, allow for catalogue search to show archived lots as well as active lots.
- **CatalogueEntry:** We assume nothing in particular about the contents of this class. We treat it as the externally-visible representation of a lot. We return it when the catalogue is requested for viewing.
- **Bid:** It is logical to create a class to represent a Bid on a lot. A Bid contains all the details required to identify it, allowing us to pass around Bid objects rather than a Bid and its associated Lot object. This ensures loose coupling between objects.
- **BidType:** This is analogous to LotStatus in its usage; it stores the representation of a Bid's type, either Increment or Jump. As with the other enums, we originally considered having this as an integer, with code within the class itself to handle the different cases. However, using an enum is conceptually much simpler and more straightforward to understand, and allows our type annotation to tell us whenever we are manipulating a BidType. With an Integer, this would not be obvious. This is in keeping with good design principles.

- **AuctionHouse:** We use the Facade design pattern here, as recommended by the coursework specification. The AuctionHouse object implements an interface to allow all other objects within the system to communicate. It primarily takes call messages associated with a use-case and passes them on to other objects as needed. Additionally, it performs extra steps before doing this based on the information it holds internally, ensuring objects only need to be aware of the bare minimum about their environment and interaction with other classes.
  We note in particular:
  - The AuctionHouse maintains an internal map of MessagingAddresses to Actors and vice-versa, so an object wanting to send a message to another object need not know its address, only the object instance itself. This allows us to provide good encapsulation for the messaging system and abstract implementation details from the other objects in the system.
  - AuctionHouse sends messages between objects using the MessagingService class, omitted from the class diagram for brevity.
  - A method is available for each use-case. While this does run the risk of creating a 'God class', this is necessary to allow objects to communicate through the AuctionHouse class, and allows the objects themselves to be modular. Through careful delegation of most actual functionality to our other classes, we keep AuctionHouse as a Facade that decouples communication between classes without implementing use cases completely, keeping the dangerous God-class at bay.
  - The viewCatalogue() method does not have a specified argument, and we assume it by default to return the entire catalogue. It may be desirable to allow for filtering the catalogue, either on the back-end or on the front-end, and by not specifying any arguments to this function we leave open the possibility of catalogue filtering on the back-end.
  - The AuctionHouse class also handles payment transfer logic within the closeAuction method. It may be desirable to have this as a separate method, but it is not included for brevity and for further specification by the client. Additionally, it is assumed (eg. in the sequence diagram) that payments are made using the Auction House's bank account as an intermediary in order to allow it to take their commission from sales. We doing direct buyer-seller transfers, but this would make taking commisions more complicated and introduce an element of risk into the transaction.
- **Status:** We assume that a given Status can represent either a success or a failure, allowing us to handle possible errors from classes. Otherwise, we assume nothing in particular about this class and do not include it in the class diagram.

# 3 Dynamic models

## 3.1 UML sequence diagram

### 3.1.1 Close auction

Due to its size, this is provided as the second-to-last page of the document.
Some notes on the sequence diagram itself:

- We leave out details on how the AuctionHouse class gets such variables as account details. These can easily be inferred from the Buyer and Seller objects, and are left out for brevity.
- We simplify the objects at the top of the sequence diagram, again for brevity. In actuality, we would assume that an external actor would initiate the close method in Auctioneer, and that we would be contacting multiple Buyers.
- We leave the process early in two cases:
  - If the lot cannot be closed: this occurs if the lot has already been closed or if it was never opened up to auction. In this case we return a failure Status to the Auctioneer.
  - Similarly, if one bank transfer cannot be completed, we propagate the failure Status down and do not attempt the other transfers, returning a failure Status to the Auctioneer. This is again excluded for brevity.

## 3.2 Behaviour descriptions

### 3.2.1 Add lot

In the *Add lot* use case, we assume the seller has already created an instance of LotInformation and has a uniqueId for it.
Now, they want to create a lot for it and add that to their set of lotsOwned, so they start by invoking the addLot method on themselves.
This will involve the AuctionHouse singleton creating new instances of both Lot and CatalogueEntry, and placing them in internal lists. Then, those lot results will be passed back to the seller to add to their own internal list.

```
seller -- addLot(LotInformation, uniqueId) ---> seller
  seller ---- createLot(LotInformation, uniqueId) ---> AuctionHouse
    AuctionHouse -- new Lot(LotInformation, uniqueId) ---> Lot
    AuctionHouse <- - - - - newLot - - - - - - - - - - - - - Lot
    AuctionHouse -- setLots(newLot) --> AuctionHouse // Add the new lot to the
                                                     // existing "lots" set
                                                     // Use implicit setter
    AuctionHouse -- setCatalogue(newLot) --> AuctionHouse // Create new CatalogueEntry
                                                          // and add to existing
                                                          // internal "catalogue" set
                                                          // Use implicit setter
  seller <- - newLot - - - - - - - - - - - - - - - - - - AuctionHouse
                                                          // Return new lot to seller
  seller ---- setLotsOwned(newLot) -> seller // Push the new lot to the internal set
                                             // of lots owned, "lotsOwned"
                                             // Use implicit setter
seller <- - - - - - - - - - - - - - - - - - - - - - - - seller
```

### 3.2.2 Note interest

In the *Note interest* use case, we assume the buyer has an instance of lot that they would like to note interest in.
This will begin with invoking markInterestInLot on themselves.

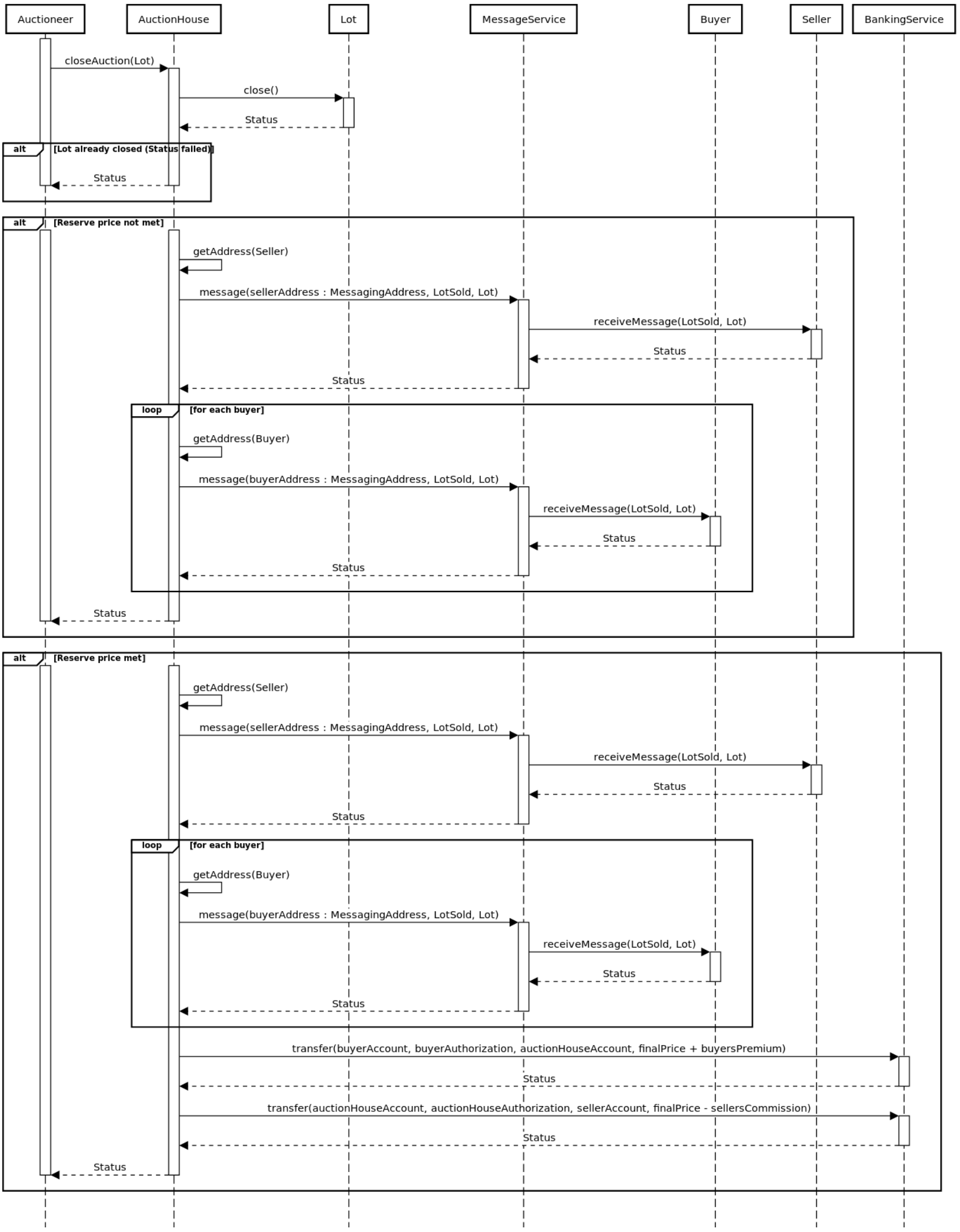
```
buyer -- markInterestInLot(lot) ---> buyer
  buyer ---- noteInterest(lot, buyer) ---> AuctionHouse
    AuctionHouse ---- setInterestedBuyers(buyer) ---> lot
                                                 // Add the buyer to the lot's
                                                 // internal "interestedBuyers" set
                                                 // Use implicit setter
    AuctionHouse <- - - - - - - - - - - - - - - - - - - - lot
  buyer <- - - - - - - - - - - - - - - - - - - AuctionHouse
  buyer ---- setInterestedLots(lot) ----> buyer
                                                 // Add the lot to the buyer's
                                                 // internal "interestedLots" set
                                                 // Use implicit setter
buyer <- - - - - - - - - - - - - - - - - - - - buyer
```

### 3.2.3 Make bid

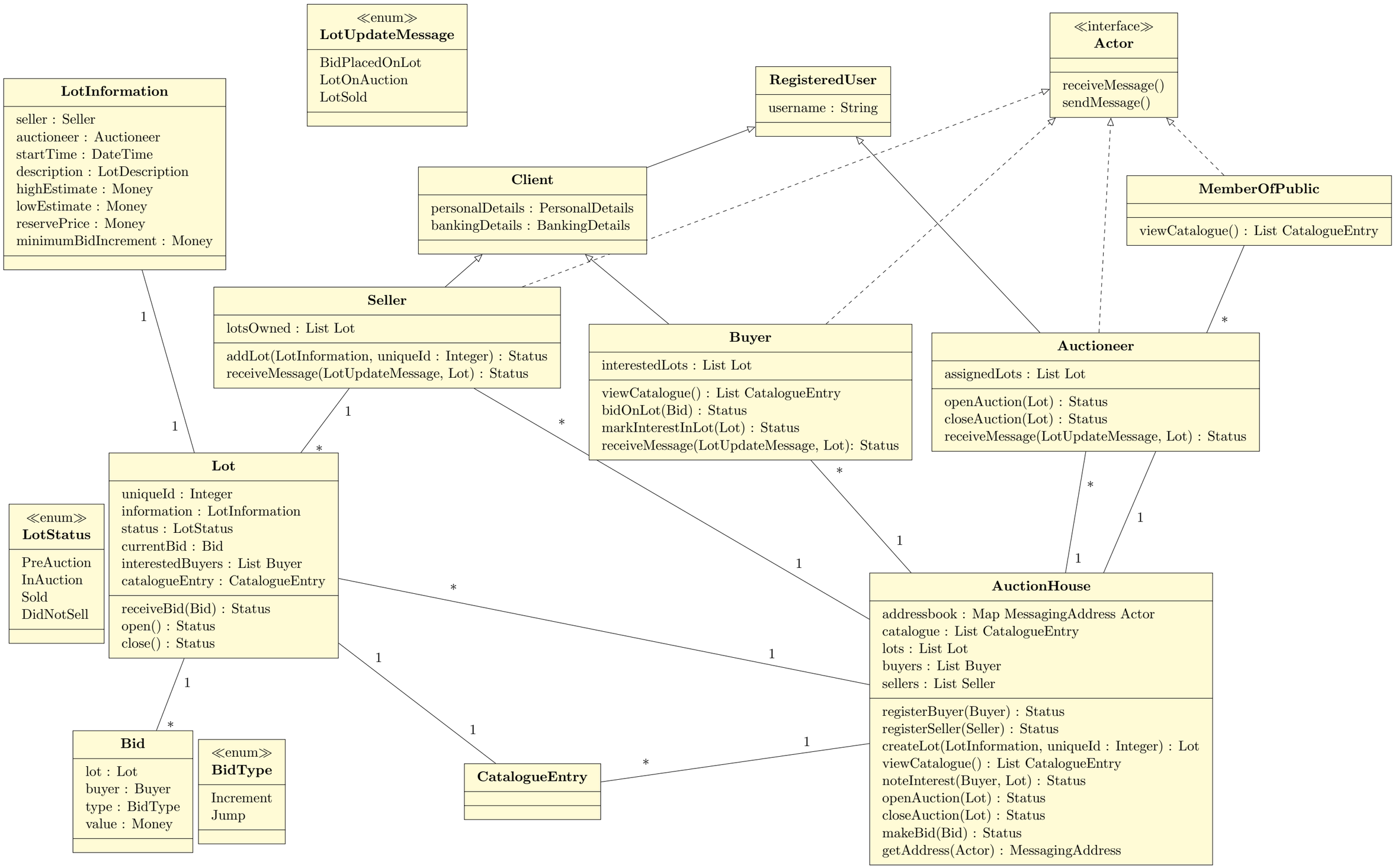
In the *Make bid* use case, we assume the buyer has already created an instance of Bid. The instance of Bid they have created has them as bid.buyer, and the lot they are bidding on as bid.lot. Its value and its type are also already defined.
Now, they want to send that bid out to the lot and have it registered in the system.
This will begin with invoking bidOnLot on themselves. As the message burst propagates, the AuctionHouse will send out relevant changes to the lot in question, and notify other participants.

```
buyer --- bidOnLot(bid) ---> buyer
  buyer ---- makeBid(bid) -----> AuctionHouse
    AuctionHouse --- getLot -------> bid // implicit getter
    AuctionHouse <- - activeLot <- - bid // return lot which bid refers to
    AuctionHouse --- receiveBid(bid) ---> activeLot
    AuctionHouse <- - bidSuccessful - - - activeLot
    // bidSuccessful is a Status variable
    // containing whether the bid was successfully placed
    // If bid is insufficient, simply skip the ensuing loop
ALT Bid is insufficient // simply return to buyer and skip the remaining operations
  buyer <- - - bidSuccessful - - AuctionHouse
buyer <- - bidSuccessful - - buyer
ENDALT
ALT Bid is sufficient // continue to notify all relevant actors of new bid
    AuctionHouse --- receiveMessage(BidPlacedOnLot, bid.lot)
                  --> bid.lot.information.auctioneer
                 // Use implicit getters on current bid to get auctioneer that
                 // is currently in charge of that lot,
                 // and notify auctioneer of new bid
    LOOP [over all bid.lot.interestedBuyers] // Use implicit getters on current
                                             // bid to get all current
                                             // interested buyers
      AuctionHouse --- receiveMessage(BidPlacedOnLot, bid.lot)
                    --> currentInterestedBuyer
    ENDLOOP
  buyer <- - - bidSuccessful - - AuctionHouse
buyer <- - bidSuccessful - - buyer
ENDALT
```

# Sequence diagram for use-case: Close lot auction




On next page: class diagram for system


≪enum≫
LotUpdateMessage
BidPlacedOnLot
LotOnAuction
LotSold
≪interface≫
Actor
receiveMessage()
sendMessage()
RegisteredUser
username : String
LotInformation
seller : Seller
auctioneer : Auctioneer
startTime : DateTime
description : LotDescription
highEstimate : Money
lowEstimate : Money
reservePrice : Money
minimumBidIncrement : Money
Client
personalDetails : PersonalDetails
bankingDetails : BankingDetails
MemberOfPublic
viewCatalogue() : List CatalogueEntry
Seller
lotsOwned : List Lot
addLot(LotInformation, uniqueId : Integer) : Status
receiveMessage(LotUpdateMessage, Lot) : Status
Buyer
interestedLots : List Lot
viewCatalogue() : List CatalogueEntry
bidOnLot(Bid) : Status
markInterestInLot(Lot) : Status
receiveMessage(LotUpdateMessage, Lot): Status
Auctioneer
assignedLots : List Lot
openAuction(Lot) : Status
closeAuction(Lot) : Status
receiveMessage(LotUpdateMessage, Lot) : Status
Lot
uniqueId : Integer
information : LotInformation
status : LotStatus
currentBid : Bid
interestedBuyers : List Buyer
catalogueEntry : CatalogueEntry
receiveBid(Bid) : Status
open() : Status
close() : Status
≪enum≫
LotStatus
PreAuction
InAuction
Sold
DidNotSell
AuctionHouse
addressbook : Map MessagingAddress Actor
catalogue : List CatalogueEntry
lots : List Lot
buyers : List Buyer
sellers : List Seller
registerBuyer(Buyer) : Status
registerSeller(Seller) : Status
createLot(LotInformation, uniqueId : Integer) : Lot
viewCatalogue() : List CatalogueEntry
noteInterest(Buyer, Lot) : Status
openAuction(Lot) : Status
closeAuction(Lot) : Status
makeBid(Bid) : Status
getAddress(Actor) : MessagingAddress
Bid
lot : Lot
buyer : Buyer
type : BidType
value : Money
≪enum≫
BidType
Increment
Jump
CatalogueEntry
1
*