Politecnico Di Milano
Facoltà di Ingegneria dell' Informazione
(Computer Science Engineering)

SOFTWARE ENGINEERING II PROJECT

Part I: **RASD**

Prof.ssa: **Di Nitto**

A.A. 2014/15

Project Title: **BidWin**

Project Repository: **https://github.com/EmanueleLM/BidWin**

Emanuele La Malfa

Matr: **841042**
emanuele.lamalfa@mail.polimi.it

Davide Malvestiti

Matr: **773345**
davide.malvestiti@mail.polimi.it

TABLE OF CONTENTS . . . . 1 / 26

1. Introduction

1.1 Preliminary Observations

Here are reported all the requirements concerning the system fulfillment:

> *"**WinBid** is an application implementing an inverse auction system. An inverse auction works like a regular auction. The difference is that in an inverse auction a user has to propose the **lowest unique bid** to win the auction. Each user has to guess the lowest bid. Bids can be placed till the bidding closing time. Then they are evaluated by the system. The winner is the user who has placed the lowest unique bid. Before the ending of the bidding time each user can provide more than one bid. At each bid the user is informed about its current position with respect to the others. Also, the system provides updates when such current position changes as a result of biddings from other users. Issuing bids has a cost of 2 per bid."*

In case of requirements where the interpretation could be ambiguous, we will choose a non-ambiguous interpretation of the model: moreover we will provide all the motivations which led us up to this point.
This criterion will be applied also in presence of incomplete or lacking requirements, solved by the use of informal (see section below) and formal (see sections from 3 to 5) representations.

1.2 Informal Description

"BidWin" is an application conceived to manage an inverse auction system. We will ensure all the typical features of such systems (e.g. like creating an auction, make an offer for an object).
First of all we need to distinguish from a registered user to a non-registered one: while the former can fully participate to the system of auctions and offers, the latter is required to register: in this way we ensure that our system is *closed* w.r.t. visitors.
Once you are registered, you can look for a particular object or just take a sneak peek of another user's list of objects: this kind of actions requires a search engine, implemented in our system in such a way that allows users to perform queries by their own.
The system must be easy to use, reliable and trustable: that is the reason why we ask our registering users to fill up their profile with name, surname, age of birth and an address - where the won packages will be sent by default- . The more you fill up your profile, the more you are considered trustable: a counter is associated to each user to keep track of the number of his transactions, and a mark from 1 to 5 is associated to each profile; this is a sort of *watermark* for trusted and untrusted users.
Every time an auction is won - there is a unique lowest bid on that object - the

auctioneer should deliver the object: when the winner receives his prize, he is required by the system to send a feedback, which consists in a mark from 1 (not received, broken or equivalent) to 5 (everything is okay, very good auctioneer) stars. This is the most trustable *discrimen* that helps a user to distinguish from a good auctioneer to a bad one.
Since we do not recommend you buying from a low-rank Auctioneer, this category of Users will not be able to use the system any longer.
If a user participates to an auction, he will be informed about the status of his biddings till its closure: on the other hand an auctioneer will receive a notification when one -or more- of his auctions are closed.
From this point on he will carry out his tasks: sending the object, waiting for the receipt of the package and the feedback.

Furthermore, we will ensure that each user can keep track of his history: the system keeps track of all the transactions that led to the winning of an Auction: These information are visible in the user's personal page.
More details in section 4.
Another issue is the time duration of an Auction: each Auction has a limited time during which the system accepts the offers that come out from the Bidders. We will implement a time increment system: if an Offer comes before the end, the Auction's ending time is delayed of a certain amount of time (1 minute).
This will result in a more usable system because an interesting Auction will last more time visible in the stack of the *open auctions*. Even this aspect will be deepened in the further sections.

1.3 Glossary

In order to improve the comprehension of the terms used in this document, we provide the following glossary:

Registered User: someone who is registered and logged -through his unique credentials- into the system. He can become Auctioneer or Bidder -in mutual exclusion w.r.t. the same object-.

Credit: a pocket associated to each user. An Offer decreases the pocket counter by 2 units.
A bonus of 100 credits is assigned to each user who newly registers to the system. Through the graphic interface the system allows users to recharge their pocket.

Auction: an event, with a starting and an expiration term that involves an Auctioneer, a list -even empty- of Bidder and an Object. During the Auction, the bidders try to obtain the object by bidding the lowest unique offer. If a bidder wins, he will receive the object by paying his lowest unique offer on that.

Auctioneer: a registered user who creates and manages an Auction.

Bidder: a registered user who participates to an Auction.

Object: something that a user owns and decides to sell in an Auction.

List Of Objects: a list of objects owned by a specific Registered User. Please note that a ListOfObjects is private: only his owner can access, view and modify it. We guarantee public visibility on Auctions, not on ListOfObjects. Further information on section 4.

Offer: a payment of 2 (two) credits which allows a User to perform an Offer on an Object**.** Please note that an offer is bought before the auction time, so a credit consists on the possibility to place an offer.

Access credentials: specific nickname and password of a registered user.

Auction counter: number of auction completed by a specific User on his own objects.

Rank: mark from 1 to 5 that shows how much a User is reliable.

2. Current System

No previous system available: the system is newly created.

3. Requirement Analysis

3.1 Actors

The following section focuses on the actors who really interact with the system; in particular the respective roles and functions assigned to them, trying wherever possible to make the document complete and easy to read.
As a system heavily based on the interaction between users, we decided to formulate two classes of actors that discharge at all the functions defined in Section **1.1.**
Both Auctioneer and Bidder are Registered User: they are extension of Registered User's class, total and overlapped if we use the ER notation.

Registered User: is the main user of the system, in particular he use the functionality offered by the system to create auctions.

He/she can:

- Perform the log-in through his/her Access credentials
- Check his ListOfObjects, status as Auctioneer and Bidder

Auctioneer: He/she can do all the things that Registered User can do.

Moreover He/she can:

- Add (and eventually remove) an Object to/from his/her ListOfObject. From that point on, an auction can be initialized
- Start an auction on his/her Object by setting its duration
- Deliver an Object, if the Auction is closed and there's a winner (a Bidder)

Bidder: he/she can do all the things that a Registered User can do.

Moreover he/she can:

- Offer on an open Auction. Multiple offers are allowed
- Give a feedback to an Auctioneer in order to evaluate how good he/she is This feature is allowed whenever a Bidder wins an Auction and receives (or not) the prize

Non registered user: is the user without any Access credential; so he/she can only see the first screen of the application that allows the registration or the login.

He/she can:

- Register a personal account (using an unique nickname) in order to obtain Access Credentials

3.2 Non-functional Requirements

Below are reported all the requirements that, even if they are not strictly related to the functionality of the system, are interesting for the final user.

3.2.1 Documentation

As regards the project development**,** we will provide these documents:

- Requirements Analysis Specification Document (RASD)
- Design Document (DD)
- Source code.
- Test cases.
- User manual.
- Validation and acceptance test cases (provided by third part agents).

3.2.2 Reliability

System should be reliable; in particular the crash probability on heavy work charge should be minimal anyway.

3.2.3 Accessibility

Interacting with the system should be intuitive and easy: that's the reason why we try to implement an intuitive navigation system.

3.2.4 Performance

In order to offer a system which can be suitable for an easy interaction with other users, all functionalities should maintain a quick time response.

4 Modeling

4.1 Search Engine

First of all we felt necessary to discern the several functionalities of the Search Engine, which allows us to look for Auctions in four ways, depending on what a Bidder is looking for: Auctions By User, Open Auctions, Auctions By Tags and MyBids.
This means that ListsOfObjects are substantially private.
Please Note that the first 3 kinds of queries are an effort to widen our potential buyers while the last one is an easier way for our users to make again an offer for an auction.
More precisely, the first one guarantees that a good Auctioneer can be easily followed, the second one is meant for registered users that do not use the system very often – maybe when they are more interested in a good offer than in a particular object -, while the third one attracts users interested in a specific offer.

Auctions By User:

This kind of query returns all the open Auctions of a specific user.

Open Auctions:

This kind of query returns all the open Auctions. It will be useful because many users - of this kind of systems - are more interested in a good offer rather than in a specific object. We felt it necessary in order to improve our offer.

Auctions By Tags:

This query returns all the open Auctions on an object, whose tags match with the ones of the query. We felt necessary to introduce this feature in order to allow users to find a specific Auction.
We provide an example - just for reasoning about the system, it will not show how exactly the system processes the queries - : a user is looking for a TV, he can choose among the list of featured objects and the system will display all the open auctions marked with that label.

My Bids:

This query returns all the open Auctions where he/she made at least one offer.

4.2 History

We give the opportunity to the user to see all his previous auctions, both opened and closed: we will organize this section as a time-ordered list, from the oldest to the newest ones.
He/She can easily switch the view from the auctions where he offered to the ones he published.

4.3 System notifications:

Whenever a user performs an offer, he can basically win or lose: the system provides that both these results are notified.
Other stuff - like his current position and the current position of the other offers - will be visible only in the related auction's page.
We know that if we want to keep the system usable, we need to send to the user the <u>right</u> number of notifications: not too much - unusable - , not too many - unreliable -.

4.4 Use Case Diagram

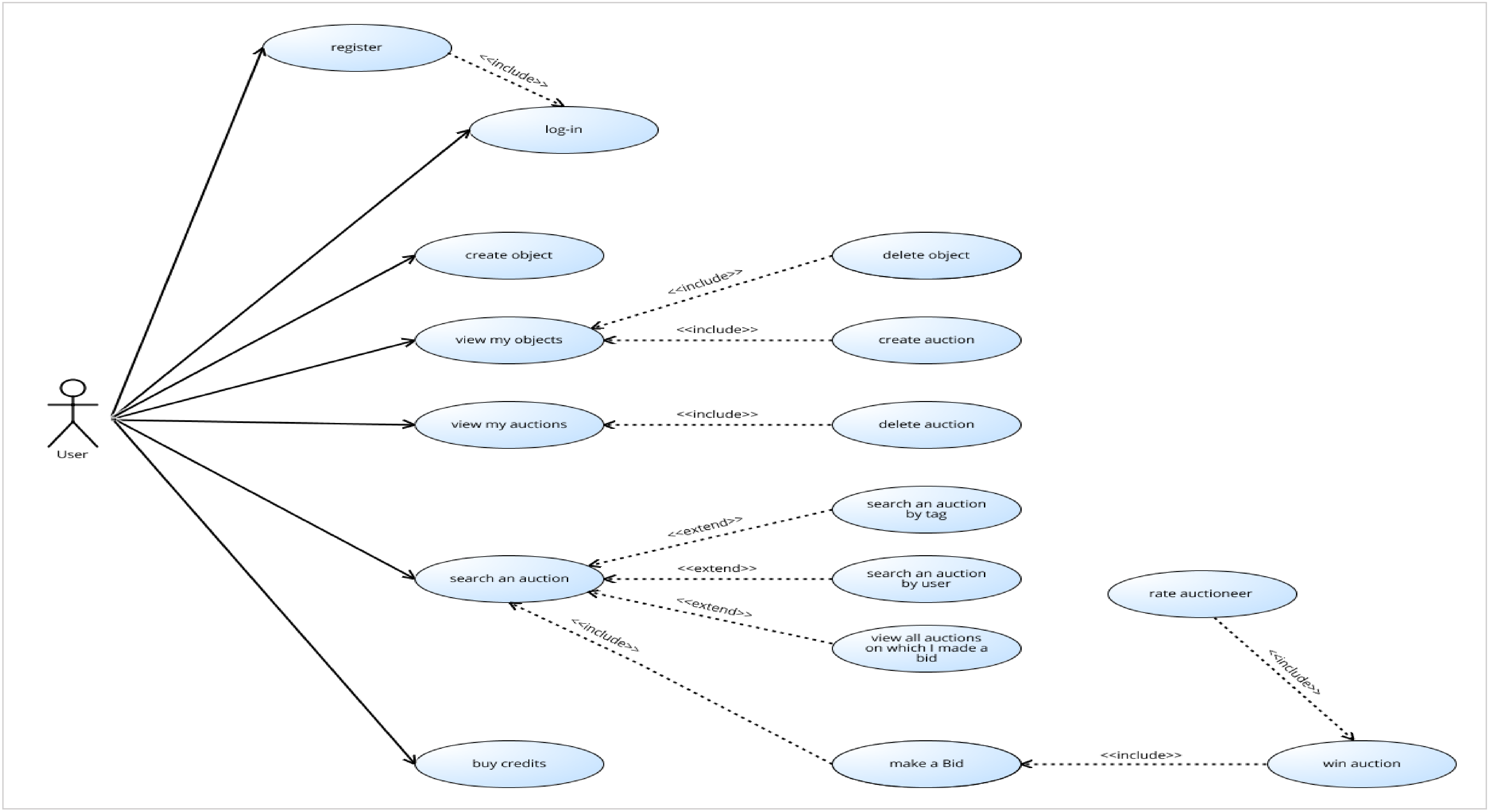
In this section we provide a scenario that describes how the users interact with the various aspects of the system: in particular we focus on the most relevant aspects like *log-in, managing auction* and *making offers.*
Please note that this Use Case is just an overview on the whole system by the user's point of view, every single aspect is deepened in the other sections of the document.

We used all the features that Use Case spec. gave us, like *"include"* and *"extend".*
Sometimes these specifications are mixed up and due to this, often misunderstood: so we preferred to specify how they are meant in our system:

<u>Include:</u> used when a specific action includes at all another one, so for example *"view my objects"* is totally included in *"delete object"* and *"create auction"* features.

<u>Extend:</u> used when an action is a specific implementation of another one: for example *"search an auction"* does the same things of*"search an auction by tag"* and *"search an auction by user"* but the former extends the search auction in a different way from the latter - they will use different queries -.

Picture #1 shows the use case diagram.



Pic 1: Use_case.png

4.2 Sequence Diagrams

In this section we provide several sequence diagrams: "*A **Sequence diagram** is an interaction diagram that shows how processes operate with one another and in what order. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.*" (http://en.wikipedia.org/wiki/Sequence_diagram)

We provide as much sequence diagrams as possible: they let us describe how the user's session flows. Here below follow the various agents with a compact description.

Please Note that we will follow the Jackson & Zave's approach - The World and The Machine -.
The machine is the portion of the system to be developed - typically, software-to-be + hardware – while the world is the portion of the world affected by the machine - the environment -.

User: see Registered User in Glossary, Section 2.

System: all the functionalities that let our system works. This is the main component that interacts with the User.

GUI: graphic user interface, it is the functionality, provided by the system, which allows the communication between User and System.

#1: User Registration: this seq. Diagram shows how a "*non-registered user"* interacts with the registration form provided by the system itself: first of all he/she has to perform a registration request, than he/she inserts a valid nickname - which will have not already been adopted by another user - a password, and a list of information such an address where the delivery pack will be sent.
If all the operations are executed correctly, the registration is successfully performed and he/she becomes a "*registered user".*

Picture #2 shows the use case diagram.

#2: Create Auction: this seq. Diagram shows how a "*registered user"* interacts with the *"auction creation"* functionality provided by the system: first of all he/she has to choose an object, than he will get the object details and a menu where he can create the Auction by specifying the duration.
Once he has asked for the auction creation the system will accept/reject the request - this sequence diagram will describe an auction that has been accepted -. Many reasons can cause a refusal: the most common one is the attempt of creating an auction while it still exists: the Alloy Section will clarify these logical constraints.

Picture #3 shows the use case diagram.

#3: Bid_auction: this seq. Diagram shows how a "*registered user"* interacts with the *"bidding system"* functionalities provided by the system: first of all he/she needs to looks for an object - we already discussed about the search engine -, than he will choose a specific Auction from the list. Now it is up to the user to make or not an offer - we describe the positive scenario-. If the system accepts the offer - action time not ended and many others constraints -he will be notified whether or not he/she has won.
We describe the winning scenario, which includes the notification of the winning, the shipment of the object and the Auctioneer evaluation.

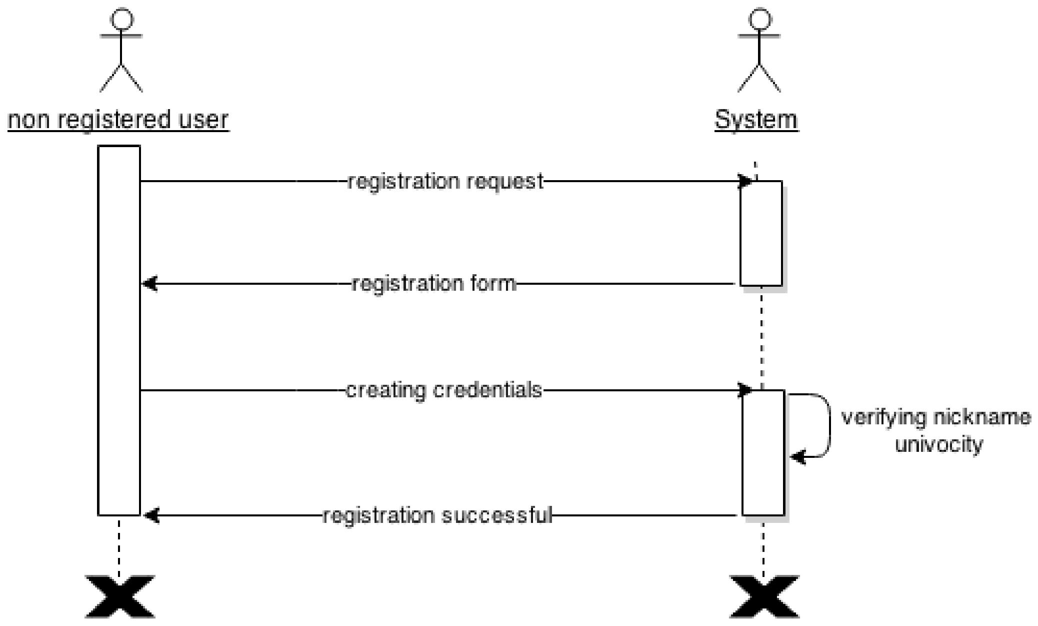
Picture #4 shows the use case diagram.

#4: Auction_delay: this seq. Diagram shows how the system delay's mechanism works. Whenever an offer comes, the system checks if the time is ending: if so, the ending time will be postponed letting other users eventually offer again.

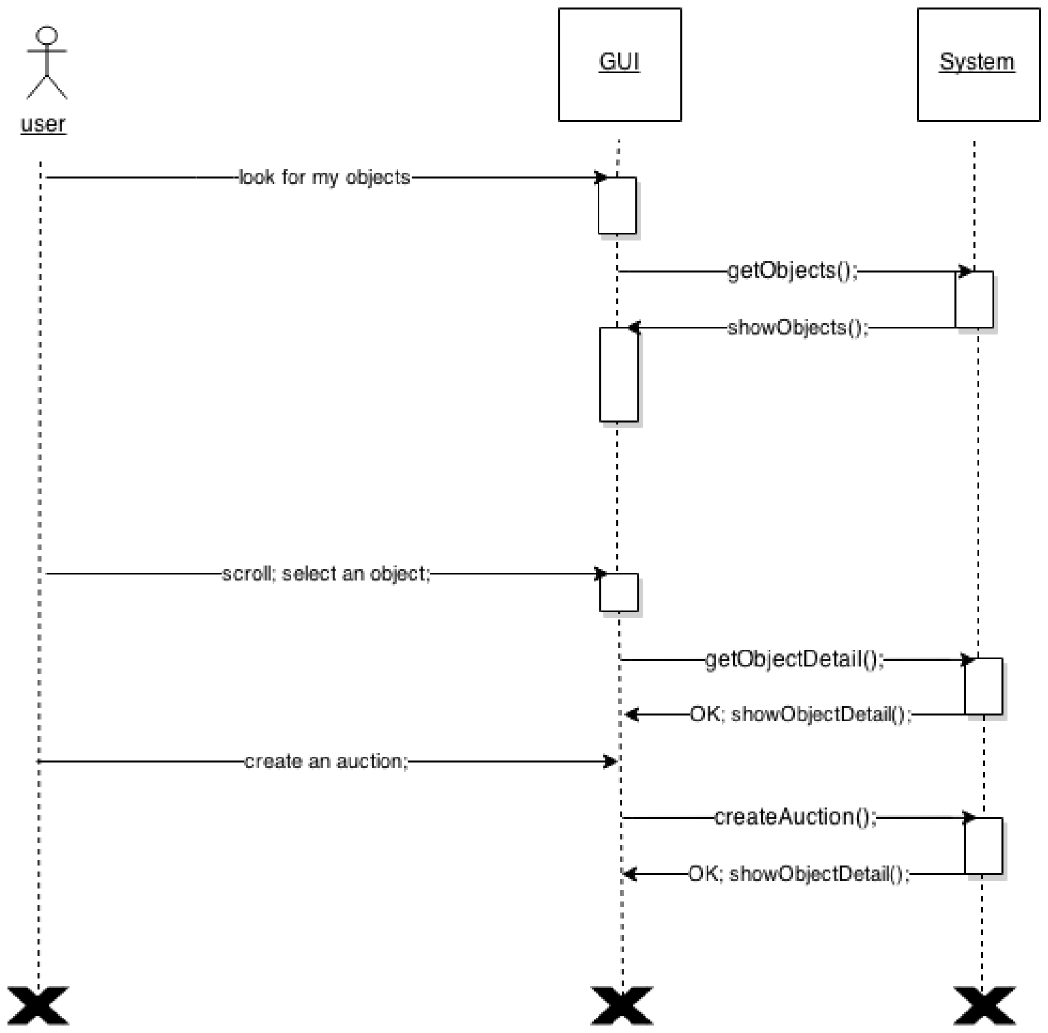
Picture #5 shows the use case diagram.

#5: Recharge_pocket: this seq. Diagram shows how a "*registered user"* interacts with the *"pocket"* functionalities provided by the system: first of all he/she needs to navigate to the user profile, than he/she will insert some values into an input form: these values are the method that he/she would like to use to buy credits, how much of them he/she wants to purchase and some other stuff related to the security side - e.g. the credit card number and its secure code -.

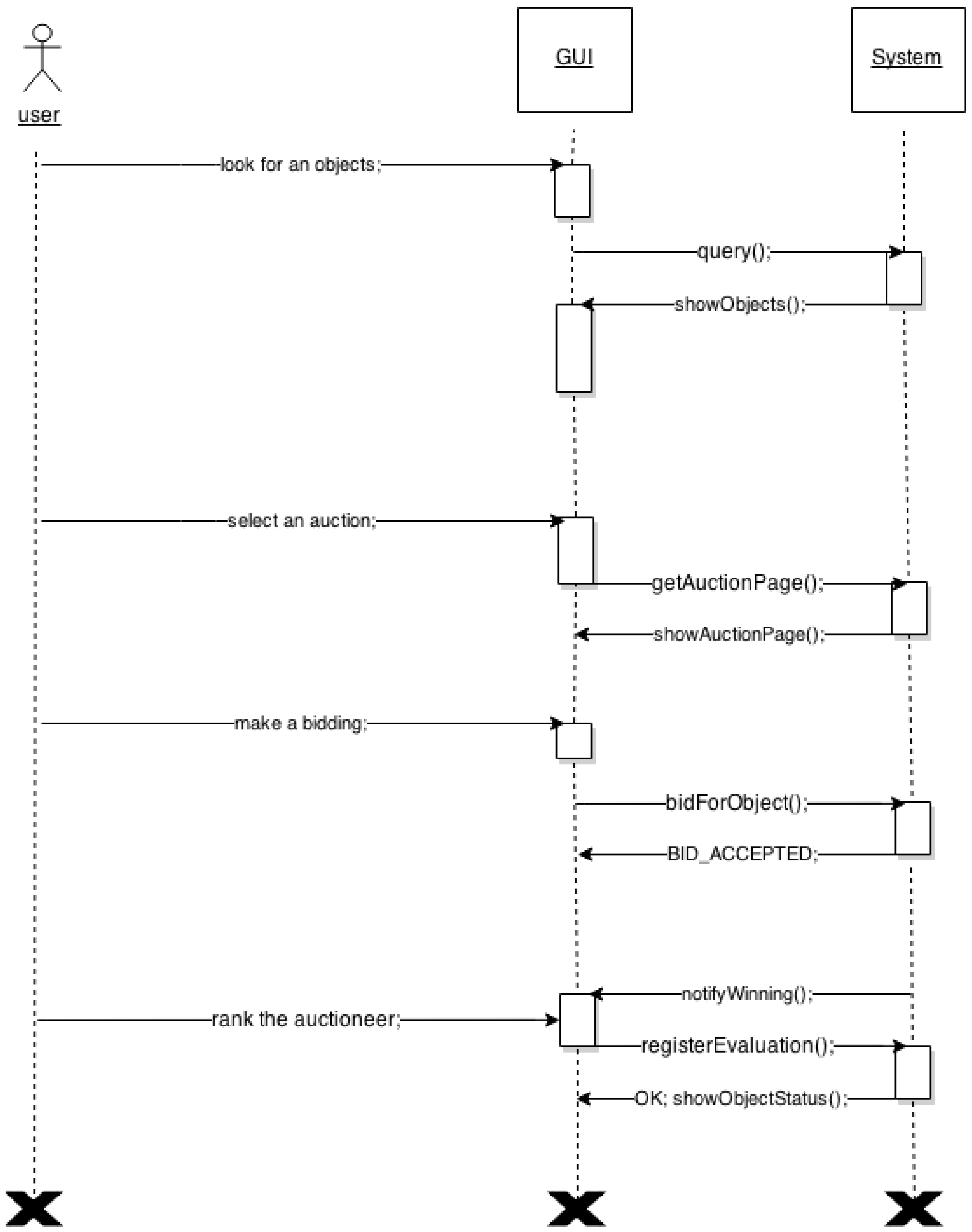
Picture #6 shows the use case diagram.



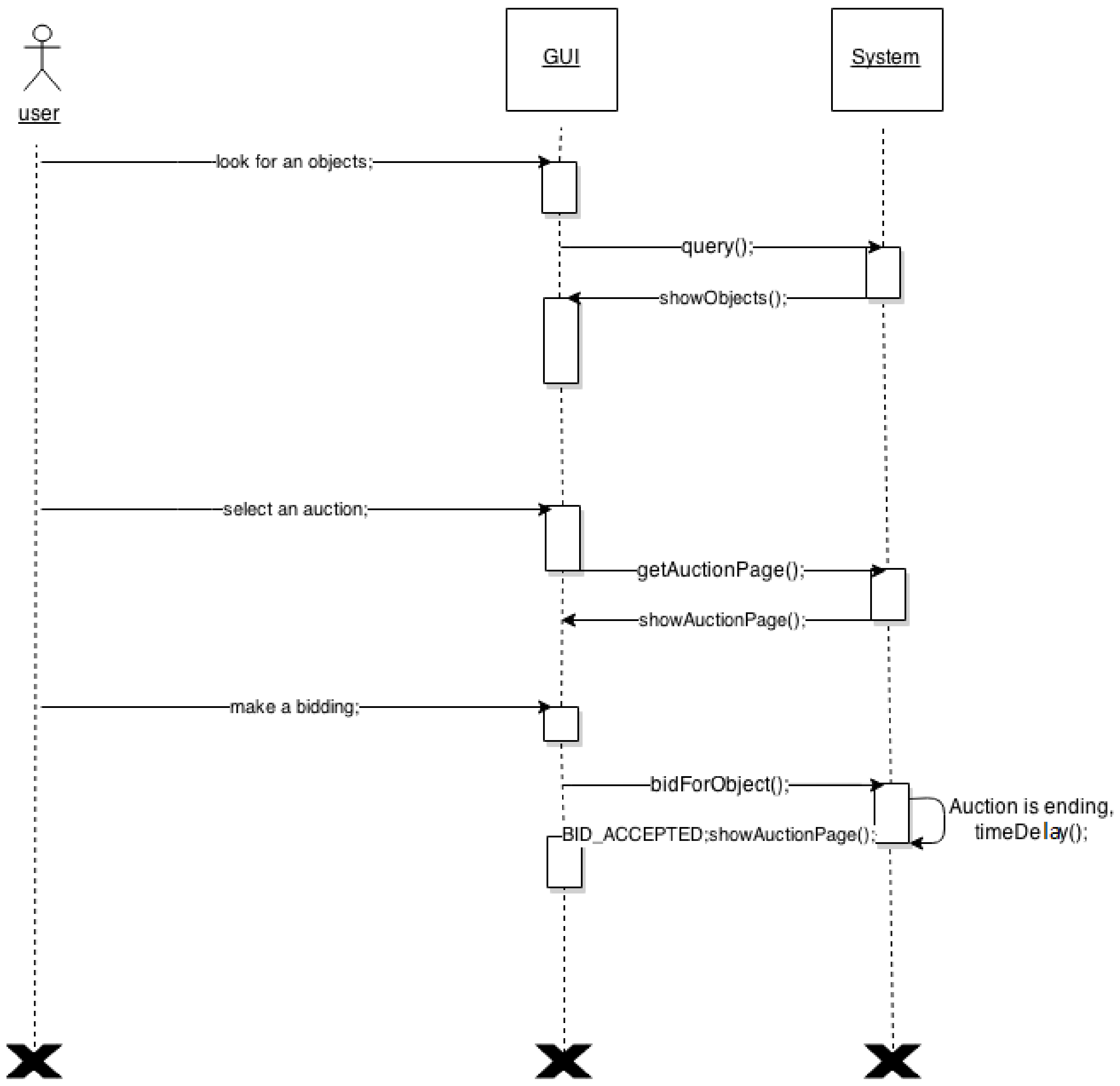
Pic 2: User_registration.png:



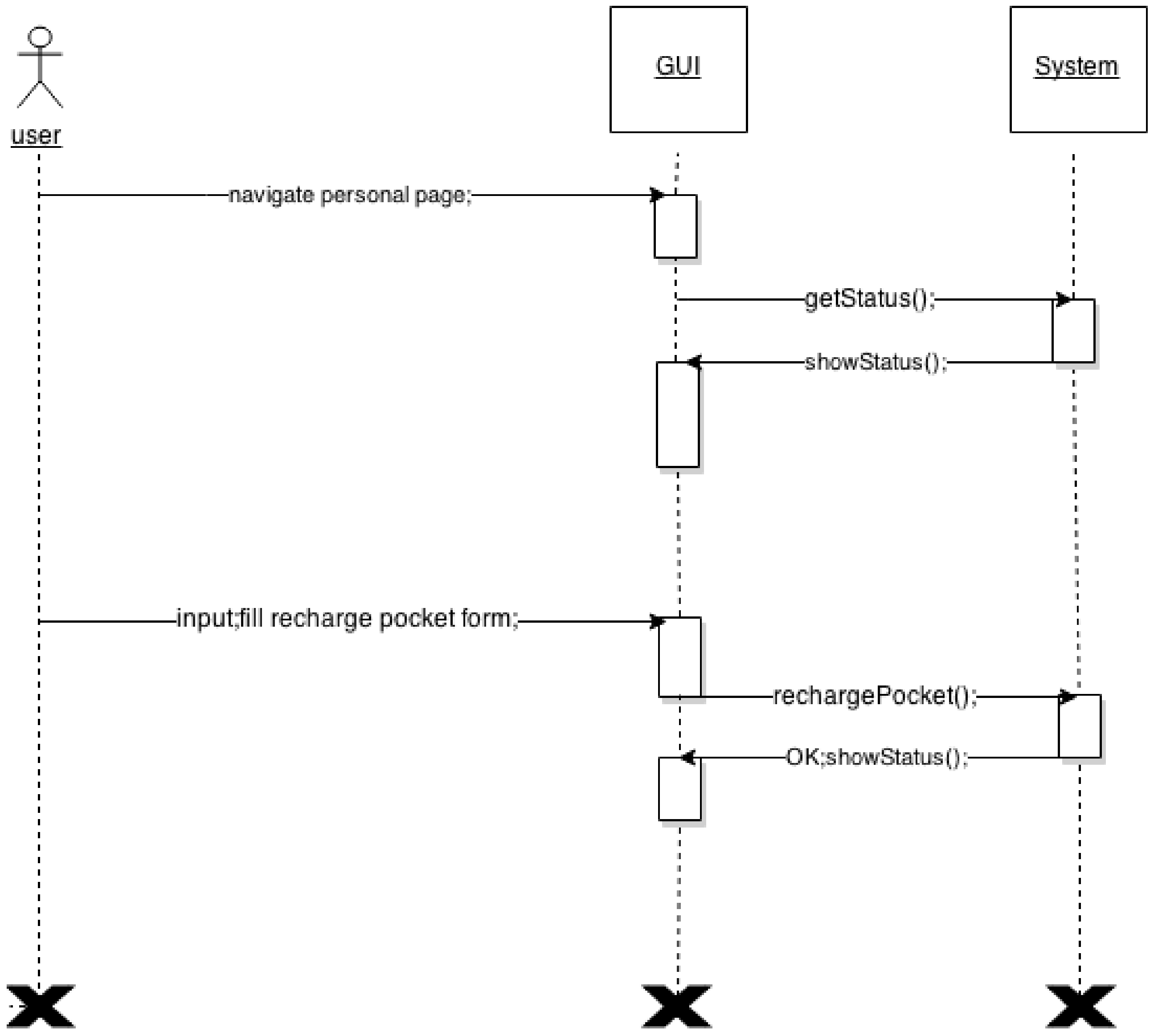
Pic 3: Create_auction.png



Pic 4: Bid_auction.png



Pic 5: Auction_delay.png



Pic 6: Recharghe_pocket.png

5. Specifications

5.1 Alloy Modeling

Although the code is commented, we would like to discuss some details.
Every user owns a unique pocket and a ListOfObjects where the objects are stored, ready to be sold;
They are loaded on the system by the user (this implementation will be explained in the section forward this one): from that point on the system keeps track of the whole movements associated to offers and auctions of such objects.
Please Note: a list can be empty.
An object in the list consists of something that a user wants to propose as an auction to the other users.
An object could not have an associated auction: a user who wants to sell this object can create an auction.
The item will be added only if the object takes part to an auction.
Many reasons can cause a non-sale item and in this case the option described above is very useful. The object would not belong to any auction, even if it still exists: if we didn't have a related object we would lose that granularity, which is indeed very meaningful. We will not guarantee that a pocket is strictly positive because of several reasons: Alloy does not allow us to represent two integers as distinct if they consist of the same value; moreover this constraint looks like an implementation choice detail which has to be guaranteed by further parts (e.g. the real implementation).

The features we used are coherent with the ones we specified in the **Glossary** section.
Moreover we added two signatures - **Time** and **TimeInterval** - which contribute to add to the system the *"time dimension"*.

```
sig User {
c: one Credit ,
l: one ListOfObjects,
}


sig Credit {}


sig ListOfObjects {
o: set Object
}


sig Object {}


sig Auction{
o: one Object,
t: one TimeInterval
}


sig TimeInterval {
st: one Time,
et: one Time
}


sig Time {}


sig Bid {
u: one User,
o: one Auction,
t: one TimeInterval
}



//a single pocket to each user
fact oneUserOnePocket {
all u1,u2: User  | u1.c = u2.c iff u1=u2
}


//every user has a unique ListOfObject, which is eventually empty
fact oneUserOneList {
all u1,u2:User | u1.l = u2.l iff u1=u2
}

//every List has a user in order to exists
fact oneListOneUser {
all u1:User, l1:ListOfObjects |  u1.l!=l1 implies (some u2:User | u2.l=l1 and u1!=u2 )
}


//there's no credit without an associated user -reflexive closure to the fact
oneUserOnePocket-
fact noCreditWithoutUser {
all c1:Credit | one u:User | u.c = c1
}


//every ListOfObjects is disjoint from all the others
fact disjointLists{
no o1:Object,  l1,l2:ListOfObjects |  l1!=l2 and o1 in l1.o and o1 in l2.o
}
```

```
//every object should belong to a ListOfObjects
fact oneObjectOneList {
all o1:Object, l1:ListOfObjects |  o1 not in l1.o implies (some l2:ListOfObjects | o1 in l2.o
and l1!=l2 )
}


//every offer is related to a single
fact oneOfferOneBid {
no b1,b2:Auction | b1!=b2 and b1.o=b2.o
}


//a user cannot bid on his own obects
fact noSelfOffer {
all of:Bid | disjoint [of.u.l.o, of.o.o]
}


//every TimeInterval has a starting and an ending time
fact oneTimeOneEvent {
all t1: Time | one ti1:TimeInterval | ti1.st = t1 or ti1.et = t1
}


//every auction has a starting and an ending time
fact oneAuctionOneStartingTime {
all t1: TimeInterval | one  a:Auction | a.t = t1
}


//every TimeInterval has disjoint starting and ending time
fact oneAuctionOneEndingTime {
all t1: TimeInterval | t1.st != t1.et
}


//every offer belongs to a TimeInterval which is the same of the associated Auction
fact sameTimeIntervalForOffersAndAuction {
all a1:Auction, o1:Bid | o1.o.o = a1.o implies o1.t = a1.t
}


//Predicates
//pred: add an Object to a ListOfObjects
pred addObject[u: User, o1,o2: Object, l1:ListOfObjects] {
u.l->o1 = u.l->o1 + l1->o2
}
```

```
//pred: remove an Object from a ListOfObjects
pred delObject[u: User, o1,o2: Object, l1:ListOfObjects] {
u.l->o1 = u.l->o1 + l1->o2
}


//pred: add an Auction on an Object
pred addAuction[a1:Auction,o1:Object] {
a1.o = o1
}


//pred: remove an Auction from an Object
pred delAuction[a1:Auction] {
a1.o = none
}


//pred: add an Offer to an Object
pred addOffer[a1:Auction, o1:Bid] {
o1.o = a1
}


//pred: remove an Offer to an Object
pred delOffer[a1:Auction, o1:Bid] {
o1.o = none
}


//Assertions
//if I add and remove the same Object, the system is unchanged
assert invariantAddAndDel {
all u1:User, o1,o2:Object, l1:ListOfObjects |
(addObject[u1,o1, o2, l1] and  delObject[u1,o1,o2,l1] and o1!=o2) implies (o1 not  in
u1.l.o)
}


//two List has always disjoint Objects
assert noOverlappedLists {
all l1,l2:ListOfObjects | l1!=l2 implies disjoint[l1,l2]
}


//there are snapshots of the system where a ListOfObjects could be empty
assert usersWithEmptyList {
some  u1:User | #(u1.l.o) = 0
}
```

```
//no users without pocket
assert userWithPocket {
no u1:User | u1.c = none
}

// there are snapshots of the system where an Object doesn’t have an associated Auction
assert objectsWithoutAuction {
some o1:Object | no a1:Auction | a1.o = o1
}

//no Auctions without related Object
assert noVoidAuction {
no a1:Auction | a1.o = none
}

//no Auctions where the same Object is sold
assert noDoubleAuctionOnObject {
no a1,a2:Auction | a1!=a2 and a1.o = a2.o
}

//no Offers in a TimeInterval which is different from the Auction’s one
assert noInvalidBiddings {
no o1:Bid | o1.t != o1.o.t
}

//A predicate that generates a significant system’s snaphot
pred show[] {
#User = 2
#Object = 5
#Auction = 3
#Bid = 4
}
```

5.2 Testing and Show Results

We used the following commands to check the validity of the statements presented so far.
We want to specify that usersWithEmptyList and objectsWithoutAuction generate some valid instances -invalid predicates in Alloy compiler -: we keep them in order to demonstrate that such behaviors **are known and accepted as valid system states.**

```
check invariantAddAndDel for 10
check noOverlappedLists for 10
check usersWithEmptyList for 10
check userWithPocket  for 10
check objectsWithoutAuction for 10
check noVoidAuction for 10
check noDoubleAuctionOnObject for 10
check noInvalidBiddings for 10
```

Then we run the show predicate. The graphical result is generated by the Alloy Show tool:

```
run show for 20
```

Figure 7 shows the result

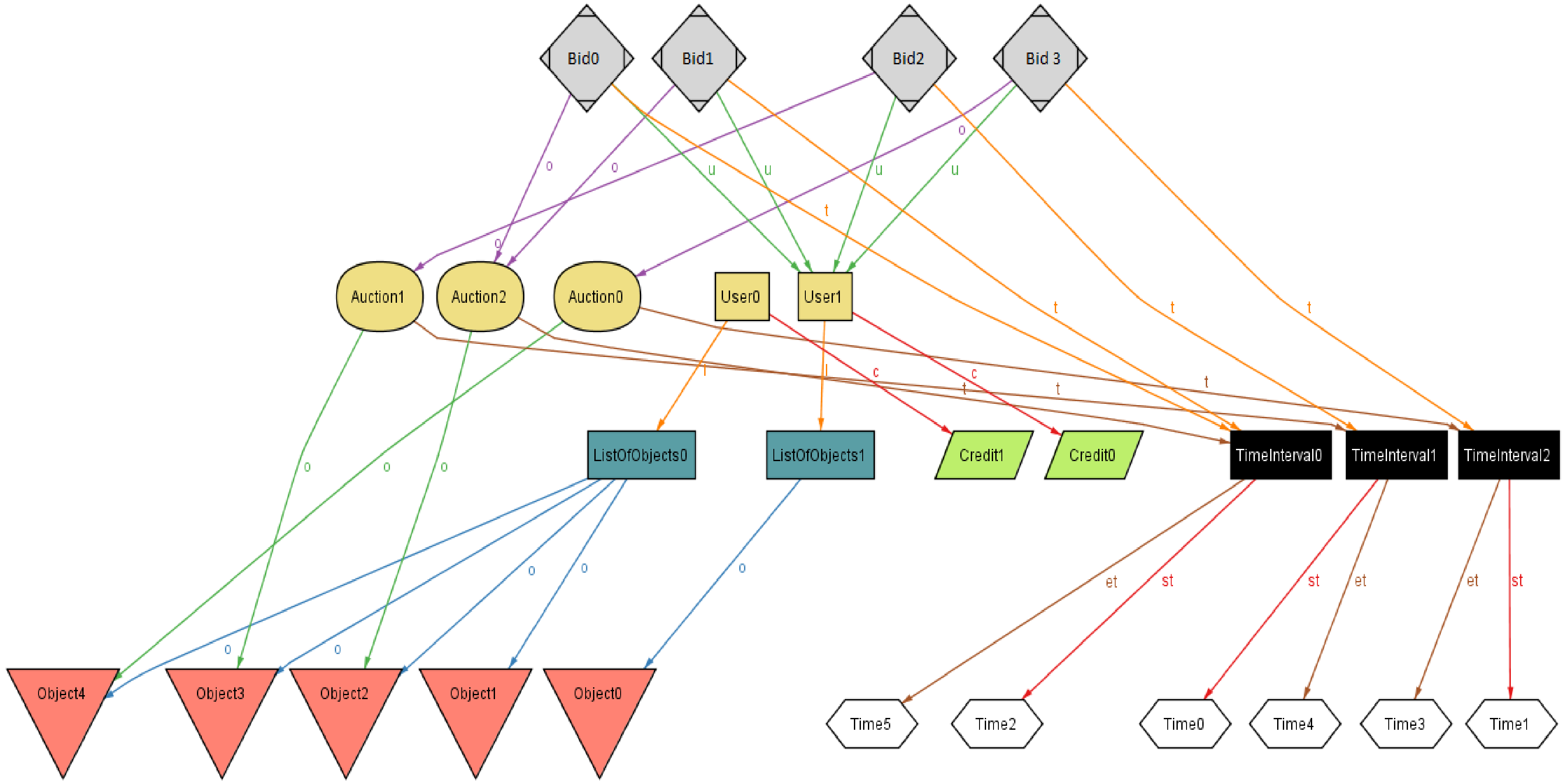


Figure 7: Alloy_pic.png