COMP-533 Object-Oriented Software Development

Midterm

(30% of final grade)

October 9, 2013

Name:

Problem 1: Auction System
Domain Model, OCL invariants and functions (40%)

Note: This problem description is exactly the same as the one that we looked at as an exercise.

Your team has been given the responsibility to develop an online auction system that allows people to negotiate over the buying and selling of goods in the form of English-style auctions (over the Internet). The company owners want to rival the Internet auctioning sites, such as, eBay (www.ebay.com), and uBid (www.ubid.com). The innovation with this system is that it guarantees that all bids are solvent.

All potential users of the system must first enroll with the system; once enrolled they have to log on to the system for each session. Then, they are able to sell, buy, or browse the auctions available on the system. Customers have credit with the system that is used as security on each and every bid. Customers can increase their credit by asking the system to debit a certain amount from their credit card. A customer that wishes to sell initiates an auction by informing the system of the goods to auction, together with a minimum bid price and reserve price for the goods, the start period of the auction, and the duration of the auction, e.g., 30 days. The seller has the right to cancel the auction as long as the auction's start date has not been passed, i.e., the auction has not already started, or at any time provided that no bids have been placed by other users. Customers that wish to follow an auction must first join the auction. Note that it is only possible to join an active auction. Once a customer has joined the auction, he/she may make a bid, or post a message on the auction's bulletin board (visible to the seller and all customers who are currently participants in the auction). A bid is valid if it is over the minimum bid increment (calculated purely on the amount of the previous high bid, e.g., 50 cent increments when bid is between $1- 10, $1 increment between $10-50, etc.), and if the bidder has sufficient funds, i.e. the customer's credit with the system is at least as high as the sum of all pending bids. Bidders are allowed to place their bids until the auction closes, and place bids across as many auctions as they please. Once an auction closes, the system calculates whether the highest bid meets the reserve price given by the seller (English-style auction reserve price), and if so, the system deposits the highest bid price minus the commission taken for the auction service into the credit of the seller (credit internal with the system).

Additional considerations:

- The auction system is highly concurrent — clients can place bids against each other in parallel, and a client can place bids in different auctions and increase his/her credit in parallel.
- For accounting reasons the auction system must keep a history of all auctions and bids.

Task 1.1

Devise a domain model that models the concepts of the auction system. Before you start, read also the description of the OCL constraints that you'll have to write in task 1.2. They might require additional concepts / associations / attributes that you need to add to your domain model. (Use the next page to draw your class diagram).

Draw your domain model for the auction system on this page:

Task 1.2

Write the following constaints and functions in OCL (if your model already models that constraint, then just write: "Is covered by model")

1. The balance of a customer's account must not be negative.
2. A customer is not allowed to place bids in his own auctions.
3. It is not possible to place two bids with the same amount within the same auction.
4. Only users that explicitly joined an auction can post messages to the bulletin board of the auction.
5. Write an OCL function that determines for a given bid if it is currently winning.
6. All bids a customer makes are solvent (in other words, a customer is always able to pay for the items that she wins). (To answer this question you can (but you don't have to) use the OCL function defined in 5.)

Problem 2: Ticket Vending Machine Use Case

The system for which requirements are to be gathered is an automated ticket vending machine like the ones you find for Montreal's subway system ("Le Metro") that allows users to upload tickets onto a chip card called "opus card". For simplicity reasons, we are going to focus only on the simple vending machines that only accept debit or credit card to purchase tickets (single fare, multi-fare, weekly and monthly tickets). A sketch of the input and output devices of the simple ticket vending machine is shown in figure 1. We are also going to assume that the "payment system", i.e., the software that handles the credit/debit card reader and PIN keyboard, is provided by some third-party vendor. In other words, the software we are developing does not need to communicate directly with the card reader, or have to deal with the details of how to handle credit/debit cards (entering and verifying PINs, connecting to credit and financial institutions to validate credit or debit money, etc.), but rather interacts with the payment system.

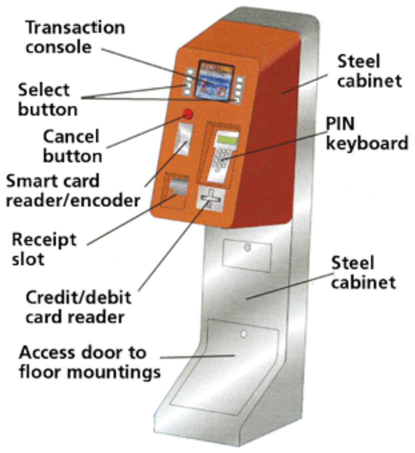

Figure 1: Simple Vending Machine

To use a ticket vending machine, the customer places the opus card into the smart card reader/encoder. The user can then consult the current tickets stored on the card, and is presented with a set of recharge options on the transaction console. The selector buttons are used to determine the desired choice. The user then interacts with the payment system to use the credit/debit card reader and PIN keyboard to pay for the selected ticket. If the payment completes successfully, the tickets are uploaded to the card and a receipt is printed. If payment was unsuccessful, the reason is displayed on the console and no tickets are issued. At any point in time before the payment is completed, the user can cancel his transaction by pressing the cancel button or simply removing his opus card from the smart card reader/encoder. Finally, during the interaction, the system beeps within 30 seconds in the case where the user does not make a selection, or forgets to remove his opus card from the smart card reader/encoder.

Write the user-goal use case *RechargeOpusCard* that describes the complete interaction between the environment and the ticket vending machine software under development. The standard use case template is given in the appendix of this midterm.

Continue the use case here, if needed:

Problem 3: Recycling Machine Use Case Map and Environment Model

The following is a use case description of the interactions that happen when a user interacts with a recycling machine.

RecycleItems Use Case

Use Case: RecycleItems
Scope: RecyclingMachine
Level: User-Goal
Intention in Context: The User wants to recycle bottles and cans in exchange for money.
Multiplicity: Only one User can recycle items at a given time.
Primary Actor: User
Secondary Actors: Recognizer, Display, FinishedButton, Printer
Main Success Scenario:
Step 1, 2 and 3 are repeated for each item the User wishes to recycle.
1. User inserts a can or a bottle into the Recognizer.
2. *Recognizer* informs *System* about the item that was recognized.
3. *System* acknowledges recognition to User by updating the refund total on the *Display.*
4. User informs *System* that s/he has no more items to process using the *FinishedButton.*
5. *System* prints receipt using *Printer.*
Extensions:
2a. Timeout occurs because user has not inserted any item into the recognizer for more than 2 minutes. Use case continues at step 5.
3a. *System* determines that the inserted item is not accepted at this store.
3a.1. *System* informs user that item is not accepted at this store using the *Display.*
3a.2. *System* instructs the *Recognizer* to reject the inserted item. Use case continues with next item at step 1.

Task 3.1

Establish a use case map for RecycleItems.

Task 3.2

Based on the RecycleItems use case and use case map, establish an environment model for the recycling machine. For each message, provide:

- The parameters of the message, if any. The allowed parameter types are all the standard OCL types. If you use other types, please provide their definition using the OCL or UML notation.
- If the name of the message does not describe its functionality in an unambigous way, provide a small textual description of what the message does.

Don't forget to add the multiplicities to the actors and the communications.

Use Case Template

Use Case:
Scope:
Level:
Intention in Context:
Multiplicity:
Primary Actor:
Secondary Actors:
Main Success Scenario:
Extensions:

OCL Summary

Operations of any OCL type:

- =, <>
- oclIsKindOf(OclType) : boolean – true if the object is of type OclType or a subclass
- oclIsTypeOf(OclType) : boolean – true if the object is of type OclType

Operations of any user-defined class:

- allInstances() : Set(user-defined-class) – returns all instances of a given class in a set

Boolean:

- not
- if .. then .. else .. endif
- =, <>
- or, and, xor
- implies

Integer and Real:

- .abs(), .max(), .min()
- For Integers: .div(), .mod()
- For Reals: .floor(), .ceil(), .round(positive position)
- - (negation)
- *, /
- +, -
- <, >, <=, >=
- =, <>

Enumeration type:

- Defined by UML class with stereotype `<<enumeration>>`
- Literals: `class::value`
- =, <>

Operations on Collections: (applied using the →operator)

- size() : Natural – returns the number of elements
- isEmpty() : Boolean
- notEmpty() : Boolean
- count(object) : natural – returns the number of occurrences of *object* in the collection
- includes(object) : Boolean – true if *object* is an element of the collection
- includesAll(collection) : Boolean – true if *collection* is a subset of the current collection
- excludes(object) : Boolean – true if *object* is not an element of the collection
- excludesAll(collection) : Boolean – true if *none* of the objects in *collection* is in the current collection
- any(boolean expression) : Object – selects one object that satisfies the expression at random
- sum() : Real – calculates the sum of all elements in the collection
- = – true if all elements in the two collections are the same. For two bags, the number of times an element is present must also be the same. For two sequences, the order of elements must also be the same.
- union(collection) : Collection
- intersection(collection) : Collection
- including(object) : Collection – returns a collection that includes object
- excluding(object) : Collection – returns a collection where all occurrences of *object* have been removed
- exists(boolean expression) : Boolean – true if expression is true for at least one element of the collection
- one(boolean expression) : Boolean – true if expression is true for exactly one element of the collection
- isUnique(expression) : Boolean – true if expression is unique for each element in collection
- select(boolean expression) : Collection – returns all elements of the collection that satisfy *expression*
- reject(boolean expression) : Collection – returns all elements of the collection that do not satisfy *expression*
- collect(expression) : Bag – computes *expression* for each element, and puts all results in a bag (or in a sequence, if applied to a sequence)
- forAll(boolean expression) : Boolean – true if for all elements in the collection *expression* is true
- asSet() : Set – transforms the collection into a set
- asBag() : Bag – transforms the collection into a bag
- sortedBy() : Sequence – produces a sorted sequence containing the elements of the original set

How to write an Invariant

(words in bold are keywords)

context Class
inv : boolean expression

How to define an OCL function

(words in bold are keywords)

context Class::FunctionName **(** [ParameterList] **) :** TypeName
body : result = Expression (of type TypeName)
or
context Class
def : FunctionName **(** [ParameterList] **) :** TypeName = Expression