COMP-533 Object-Oriented Software Development

Midterm

(30% of final grade)

October 19, 2010

Name:

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

An insurance company wants to automate its functions. The company sells several kinds of insurance policies, including life insurance and car insurance, and has plans to expand its services. Customers can purchase insurance policies, pay insurance fees accordingly, and receive compensation. Each insurance purchase activity is overseen by a sales man and results in a contract, in which the form of payment (e.g. yearly, monthly, ...) is specified. Claim agents are in charge of handling claims. They study the claims received against the insurance plans and decide on how much compensation the client is entitled to. All claims, even those that are not compensated, have to be archived in case of future reclamations.

Life insurance policies
When purchasing a life insurance, the customer has to nominate at least one beneficiary (who will receive the compensation in case of death of the customer). For life insurance policies, the yearly fee is based on the persons age and the date the contract was signed.

Car insurance policies
The fee for a car insurance is based on the price of the car and the year in which the car was built. The base car insurance covers accidents for the primary driver only (i.e. the customer himself/herself). It is however possible to add a maximum of 5 secondary drivers to the contract. Each secondary driver costs $10 extra per year. This total (base fee + secondary drivers) is then further adjusted based on the claim history. For each accident that has been compensated in the past, the fee is increased by 20%.

Task 1.1

Devise a domain model that models the concepts of a (single) insurance company. Before you start, read also task 1.2 and the description of the OCL constraints that you'll have to write in task 1.3. 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).

Task 1.2

Explain how, if ever the company decides to add new insurance policies (like a house insurance) to their portfolio, your model would have to be updated.

Draw your domain model for the insurance company on this page:

Task 1.3

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

1. Every car has to be covered by an insurance policy.

2. A car insurance has to be bought by the owner of the car.

3. A customer's spouse has to be declared as a secondary driver.

4. Write an OCL function that, given a base fee, calculates the adjusted fee for a car insurance policy.

5. Sales agents and claim agents should not be handling policies of their relatives, or policies in which they are involved as beneficiaries or secondary drivers. (You can either write one invariant, or multiple invariants to answer this question.)

Problem 2: Recycling Machine Use Case

The system for which requirements are to be gathered is an automated recycling machine like the ones you find at Lowlaws or Metro stores. A recycling machine has a hole that allows a customer to insert cans or bottles that he wants to return in order to collect the recycling refund. In general, a store only accepts cans and bottles that are also sold in the store (e.g., you cannot return "Harp beer" bottles at the Metro, because Metro does not sell harp beer). In order to enforce that rule, the system has a recognition device that can recognize, based on shape, color or bar code, what kind of can/bottle has been inserted. Each time a valid item is processed, the total amount of refund is updated on the display. Unrecognized or invalid items are ejected. A user inserts cans and bottles one at a time. When finished, the recycling machine prints a receipt that the customer can take to a cashier in order to be refunded.

Write the user-goal use case *RecycleItems* that describes the complete interaction between the user and the recycling machine. You do not have to consider the refunding activity done by the cashier. The standard use case template is given in the appendix.

Problem 3: Environment Model

Based on the use case diagram shown in figure 1 and on the two use case descriptions below, establish an environment model for the *Drink Vending Machine* system. 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 associations

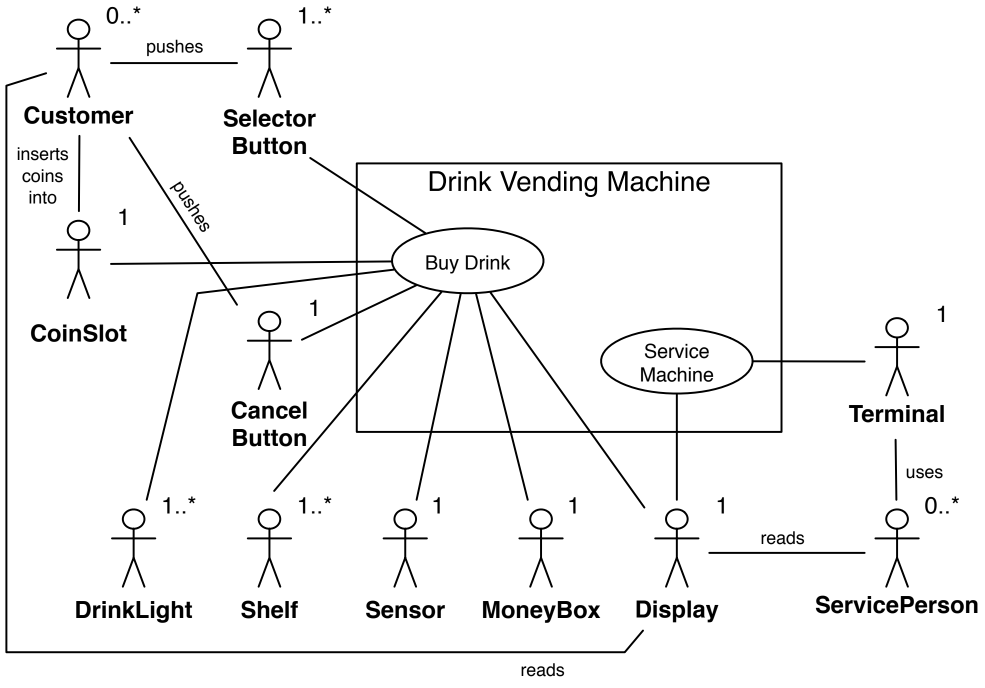

Figure 1: Drink Vending Machine Use Case Diagram

BuyDrink Use Case

Use Case: BuyDrink
Scope: Drink Vending Machine
Level: User Goal
Intention in Context: The intention of the Customer is to buy a drink in exchange of money.
Multiplicity: There can always be only one Customer or Service Person interacting with the system at a given time. There can only be one instance of BuyDrink executing at a given time.
Primary Actor: Customer
Secondary Actors: SelectorButton, CoinSlot, DrinkLight, CancelButton, Sensor, Shelf, MoneyBox, Display, Terminal
Precondition: The system is in service, filled with drinks and change, and the Money Box is not full.
Main Success Scenario:

Customer selects drink by pushing appropriate drink selector button.
1. *Button* notifies *System* of selected drink.
2. *System* displays the price of the selected drink on *Display.*

Customer inserts a coin into CoinSlot.

3. *CoinSlot* recognizes coin and notifies *System.*
4. *System* updates the remaining price on *Display.*

Steps 3 and 4 are repeated until the amount of inserted money reaches or exceeds the price of the drink.

5. *System* validates that there are sufficient funds for the selection and notifies *Shelf* to start dispensing the drink.
6. *Sensor* informs *System* that the drink has been dispensed.
7. *System* asks *Money Box* to collect the specified amount of money and, if necessary, provide the change.

Customer collects the drink and optionally the change.

Extensions:

2a. *System* ascertains that the selected drink is not available and flashes *Drink Lights*; use case ends in failure.
3a. *CoinSlot* fails to identify the coin and ejects it; use case continues at step 3.
(3-4)a. *Customer* informs *System* to abort the sale by hitting the *Cancel* button;
  (3-4)a.1 *System* asks *Money Box* to eject coins; use case ends in failure.
(3-4)b. *System* times out.
  (3-4)b.1 *System* asks *Money Box* to eject the inserted coins; use case ends in failure.
5a. *System* ascertains that the inserted money exceeds the price for the drink and that there is not enough change;
  5a.1 *System* asks *Money Box* to eject inserted coins.
  5a.2 *System* displays "no change" on *Display*; use case ends in failure.

ServiceMachine Use Case

Use Case: ServiceMachine
Scope: Drink Vending System
Level: User Goal
Intention in Context: The intention of the Service Person is to maintain the machine by ensuring that there are drinks available, modifying drink pricing, and by collecting the money earned.
Multiplicity: There can be only one Service Person servicing the machine at a given time, and while the machine is serviced, no one can buy drinks. There can only be one instance of ServiceMachine executing at a given time.
Primary Actor: Service Person
Secondary Actor: Terminal, Display
Main Success Scenario:

Service Person interacts with the system by using the Terminal.
1. *Service Person* identifies herself to *System.*
2. *System* displays welcome message on *Display.*

Steps 3-4 can be repeated for each shelf, in any order.

3. *Service Person* informs *System* of new price for a shelf.
4. *Service Person* replenishes a shelf and informs *System* of new number of drinks for that shelf.

Step 5 can be skipped.

5. *Service Person* empties the *Money Box*, replenishes the change and informs the *System.*
6. *Service Person* informs *System* that maintenance is over.

Extensions:

2a. *System* fails to identify the *Service Person*; use case ends in failure.

Sketch your Environment Model here:

Use Case Template

Use Case:
Scope:
Level:
Intention in Context:
Multiplicity:
Primary Actor:
Secondary Actor:
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