COMP-533 Model-Driven Software Development

Midterm

October 30th 2013: 6pm - 9pm, MDHAR-G10

(30% of final grade)

Name:

McGill ID:

Instructions:

- DO NOT TURN THIS PAGE UNTIL INSTRUCTED
- This is a closed book examination. Non-electronic translation dictionaries are permitted, but instructors and invigilators reserve the right to inspect them at any time during the examination.
- Besides the above, only writing implements (pens, pencils, erasers, pencil sharpeners, etc.) are allowed. The use of any other tools or devices is prohibited.
- Answer **all** questions **on this examination paper** and return it. If you need additional space, use pages 10-12, and clearly indicate where each question is continued.
- You are allowed to detach the OCL summary (last sheet of the exam) from this booklet at your convenience.

The exam has 4 questions, weighted as follows:

- Problem 1 - Grocery Cooperative: Domain Model, OCL invariants and functions (40%)
- Problem 2 - Ticket Vending Machine: Environment Model (25%)
- Problem 3 - Parking Garage: Use Case Model (20%)
- Problem 4 - Strategic Conquest: OCL invariants and functions (15%)

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

The application is about a grocery cooperative: a union of grocery retailers are cooperating when purchasing goods in order to get better conditions from the suppliers. The cooperative buys products, each characterized by a name and a unit size (measured in integral steps of cm^3), from suppliers. A supplier is known by its name and address. Each supplier has its own price for a given product. Goods are delivered by the supplier to one of the warehouses managed by the cooperative.

The cooperative fixes its own retail prices. A retailer, known by its name and address, orders from the cooperative the products in the quantities it needs. Retailers have accounts with the cooperative. The amount a retailer must pay is determined when the order is placed, but only when the goods were shipped and delivered to the retailer the account of the retailer is charged. The cooperative also allocates credit limits to its members. A retailer cannot overdraw this credit limit by an order. Whenever the quantity of a product in stock falls below a certain limit (as a consequence to an order by a retailer), an order to replenish the stock is sent to the supplier that offers the best price for that product.

Task 1.1

Devise a domain model that models the concepts of the grocery cooperative.

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. A retailer is not allowed to overdraw his account over the credit limit.

2. All stock for a given product is stored in the same warehouse.

3. The current stock of a product is not allowed to fall below the stock limit for that product.

4. Write an OCL function that determines for a given product the supplier that currently offers the best price.

Problem 2: Ticket Vending Machine Environment Model

Informal Description:

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.

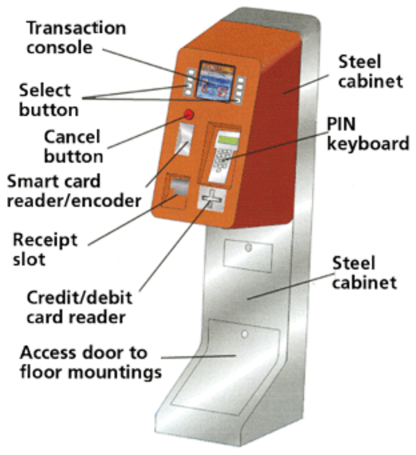

Figure 1: Simple Vending Machine

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.

The use case *RechargeOpusCard* describing the complete interaction between the environment and the ticket vending machine software under development is presented below:

Use Case Model:

Use Case: RechargeOpusCard
Scope: TicketVendingMachine
Level: User-Goal
Intention in Context: The User wants to refill his OpusCard using a credit card.
Multiplicity: Only one User can recharge an opus card at a given time.
Primary Actor: *User*
Secondary Actors: *SmartCardRE*, *PaymentSystem*, *Printer*, *Speaker*
Main Success Scenario:
1. *User* notifies *System* that he wants to recharge his opus card.
2. *System* shows tickets that are currently on the card and recharge options to *User*.
3. *User* informs *System* of recharge choice.
   Step 4 and 5 can happen in any order.
4. *System* displays price of current choice to *User*.
5. *System* informs *PaymentSystem* of price of the ticket.
   User completes the transaction with the payment system.
6. *PaymentSystem* informs *System* of successful completion of the transaction.
7. *System* uploads tickets onto opus card using the *SmartCardRE*.
8. *System* prints receipt using *Printer*.
9. *System* asks the *User* to collect the receipt and remove opus card.

Extensions:
2-6a. *User* informs *System* that he wants to cancel the transaction. Use case ends in success.
3a. Timeout
  3a.1. *System* asks *Speaker* to beep. Use case continues at step 3.
6a. *PaymentSystem* informs *Systems* that payment was unsuccessful.
  6a.1. *System* informs *User* about failed transaction. Use case continues at step 3.
10a. Timeout
  10a.1. *System* asks *Speaker* to beep. Use case continues at step 10.

Task 2

Elaborate a *low-level* Environment Model for the TicketVendingMachine here. Low-level means that you should show that the software interacts with the hardware devices. In other words, you are not allowed to send or receive messages directly from the User. Provide type and message definitions (with parameters) on the next page.

Type and Message Definitions:

Problem 3: Parking Garage Use Case

The following is an informal description of how an automobilist interacts with a parking garage control system (PGCS) when parking his car. The function of the PGCS is to control and supervise the entries and exits into and out of a parking garage. The system ensures that the number of cars in the garage does not exceed the number of available parking spaces.

The entrance to the garage consists of a gate, a state display showing whether any parking space is available, a ticket machine with a ticket request button and a ticket printer, and an induction loop (i.e., a device that can detect the presence or absence of a vehicle). To enter the garage, the driver, receives a ticket indicating the arrival time upon his request. The gate opens after the driver takes the ticket. The driver then parks the car and leaves the parking garage. In case of problems, the PGCS notifies an attendant by means of an attendant call light.

Task 3

Elaborate the *EnterGarage* use case, which describes all the interaction steps between the system and the environment that occur when a driver enters the garage. You do not have to specify what happens when paying for parking and exiting the garage. You also do not have to specify the interactions that the attendant has with the PGCS when servicing the system or when handling problems.

You are asked to write the *EnterGarage* use case at a low-level of abstraction, clearly stating which hardware devices the software is interacting with.

Problem 4: Strategic Conquest Concept Model and OCL

Strategic Conquest is a two player strategy game where the ultimate goal is world domination. The game is fought on land, sea and in the air with different types of military units (e.g. tanks, fighter planes, etc.). The units are produced by cities when they are under the control of a player. Land units have to use transporters to move between islands. Air units use a lot of fuel, and must refill at a city from time to time, or else they crash and are destroyed in the process.

The Concept Model for StrategicConquest is shown in figure 2.

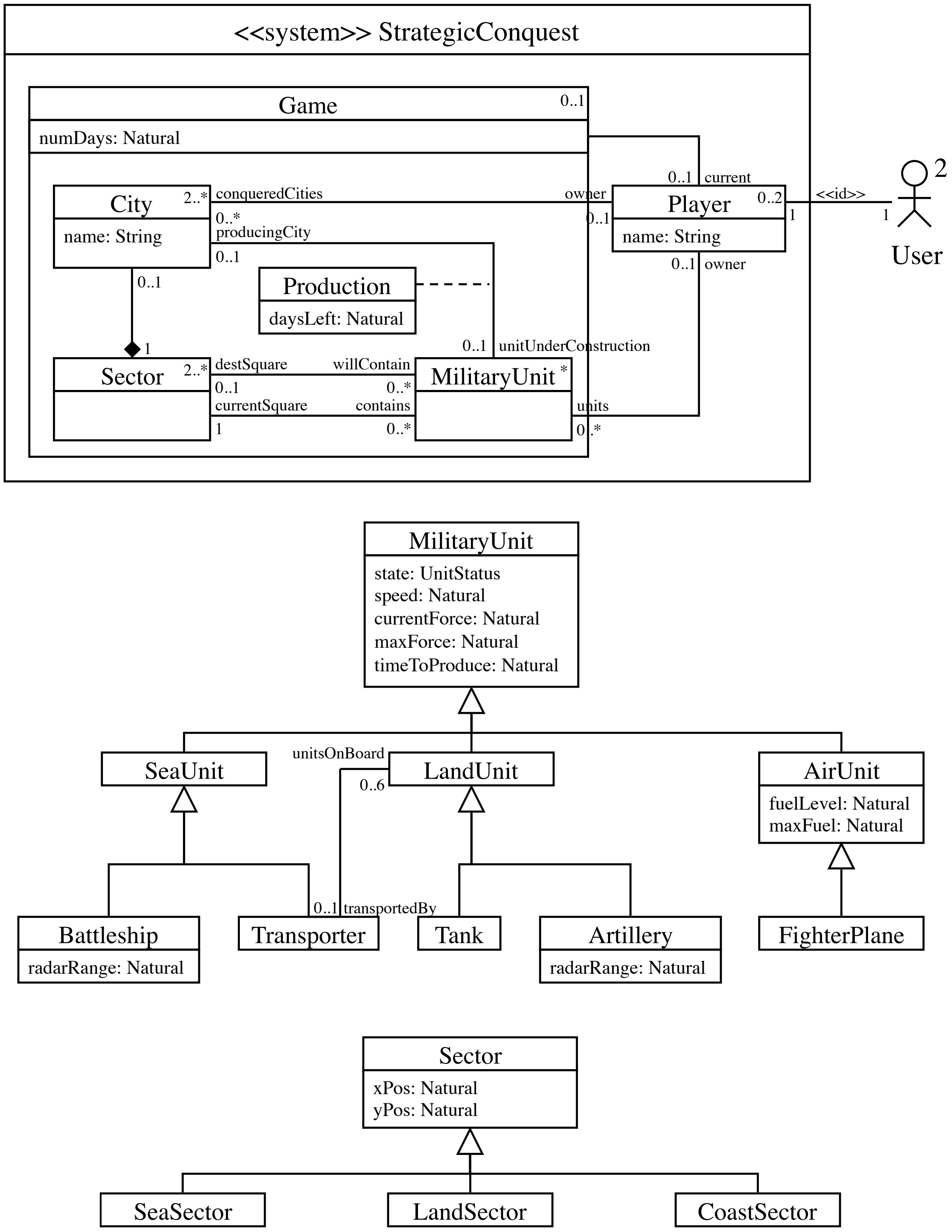

Figure 2: Strategic Conquest Concept Model

Task 4

1. Describe the following invariant in English or French:

   context s: SeaSector
   inv: s.contains→**select**(su | su.**oclIsKindOf**(SeaUnit))→**size**() $\leq$ 1

2. Describe the following invariant in English or French:

   ```
   context l: LandUnit
   inv: if l.transportedBy→notEmpty() then
               not l.currentSquare.oclIsTypeOf(LandSector)
           else
               not l.currentSquare.oclIsTypeOf(SeaSector)
           endif;
   ```

3. Write the following invariant using OCL: Only coastal cities can build sea units.

4. Write the following invariant using OCL:
   When land units are in a transporter, they automatically move where the transporter moves.

5. Write the following invariant using OCL:
   The speed of air units is always higher than the speed of land units.

6. Write an OCL function that determines the number of tanks that will be constructed in the next n days for a given player.

If you use this page for supplying your answer, please state which question your answer belongs to:

If you use this page for supplying your answer, please state which question your answer belongs to:

If you use this page for supplying your answer, please state which question your answer belongs to:

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