Midterm 2013

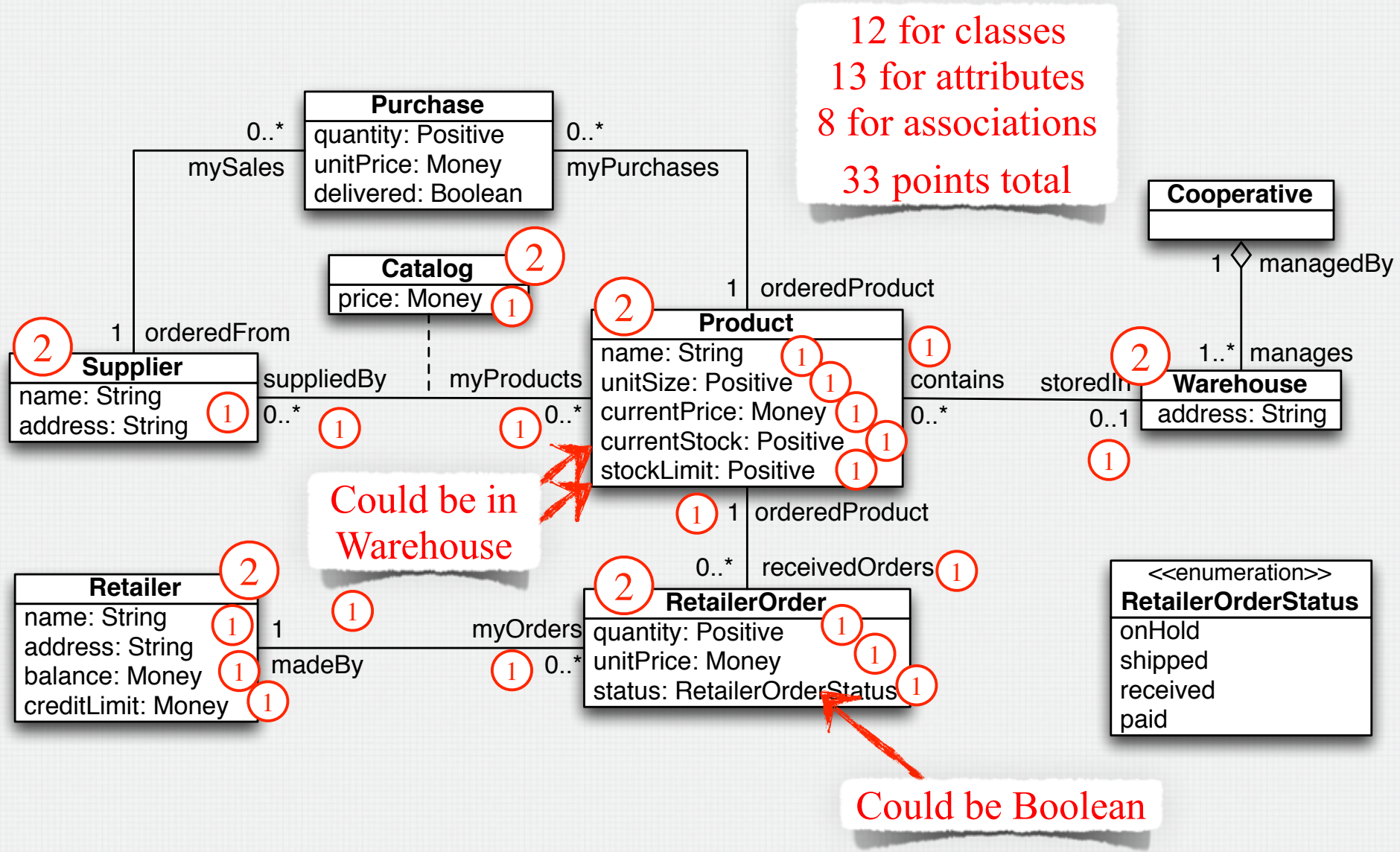
Jörg Kienzle

Grocery Cooperative

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.

Grocery Cooperative



Grocery Cooperative OCL (1)

- Write the following OCL constraints:

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

(1) **context** r: Retailer
inv: r.balance + r.creditLimit >= 0

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

(1) Covered by model

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

(1) **context** p: Product
inv: p.currentStock >= p.stockLimit

Grocery Cooperative OCL (2)

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

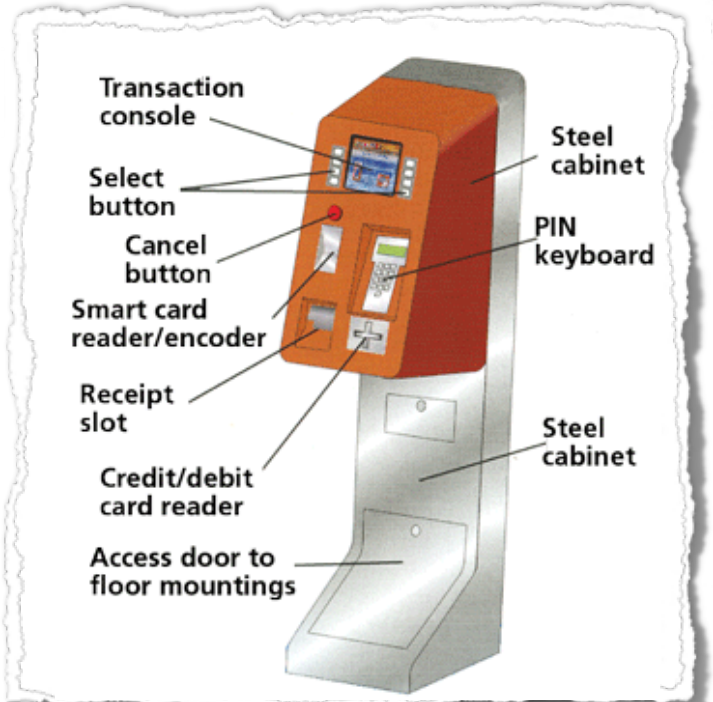
④ **context** p: Product
def: bestSupplier() : Supplier =
p.catalog→**any**
(c | c.price = p.catalog.price→**min**()).suppliedBy

33 for domain model
7 for OCL
40 points total

Ticket Vending Machine (1)

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 below.

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.



Ticket Venging Machine (2)

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.

RechargeOpusCard Use Case (1)

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.

RechargeOpusCard Use Case (2)

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.

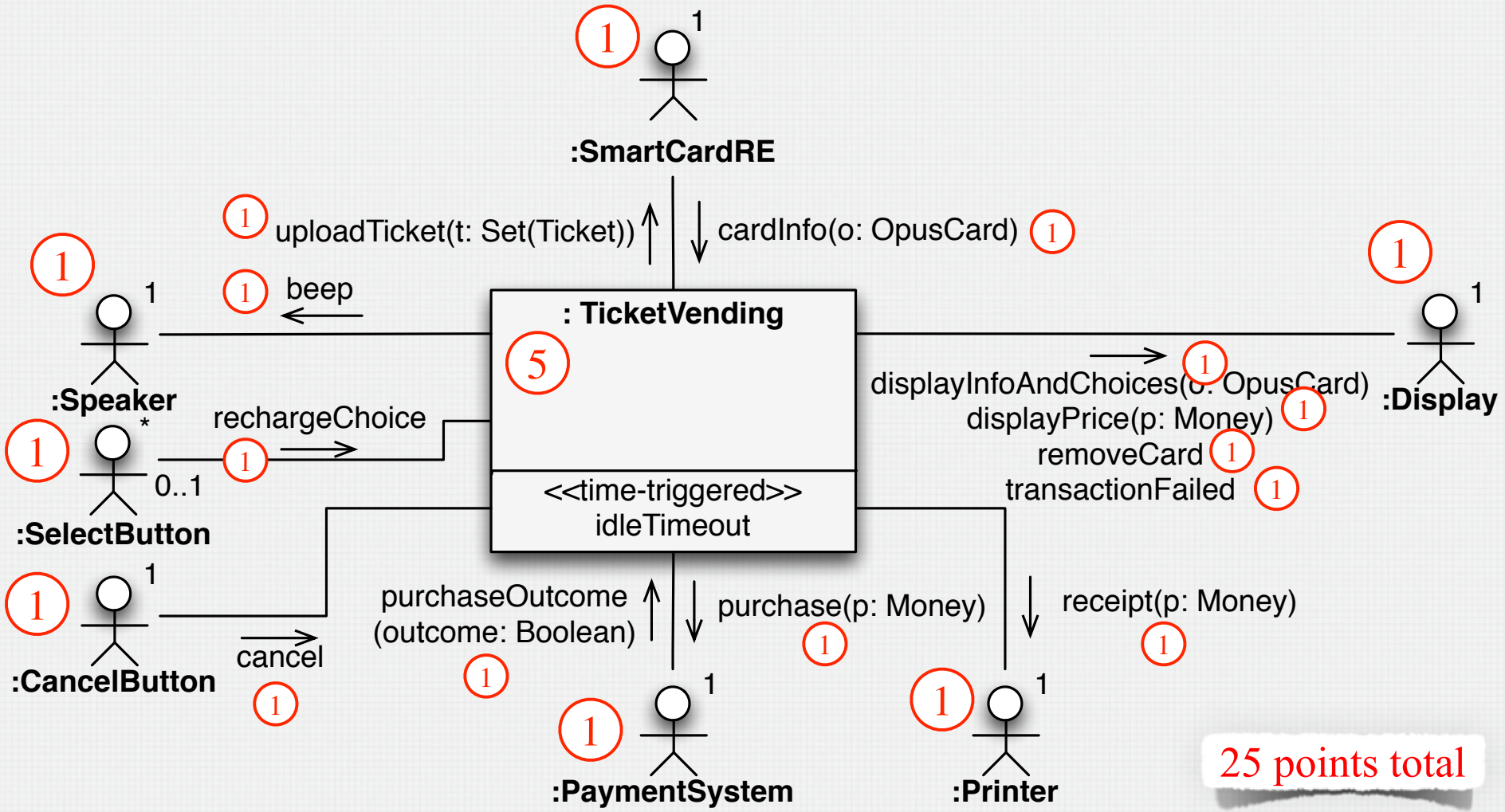
6a. PaymentSystem informs Systems that payment was unsuccessful.

  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.

TicketVending Environment Model



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.

Parking Garage Use Case (1)

1 **Use Case**: EnterGarage
1 **Scope**: PGCS
Level: User-Goal
Intention in Context: The Driver wants to enter the garage with his vehicle.
1 **Multiplicity**: Only one Driver can enter the garage at a given time for each entry. If there are *n* entries, then *n* EnterGarage use cases can execute at the same time.
1 **Primary Actor**: Driver
6 **Secondary Actors**: RequestButton, Display, TicketPrinter, InductionLoop, Gate, Attendant Light
Main Success Scenario:
Driver drives the car to the entrance and stops.
2 1. *RequestButton* informs *System* that a Driver is requesting entry.
2 2. *System* requests *TicketPrinter* to print ticket.
2 3. *TicketPrinter* informs *System* that Driver has taken the ticket.
2 4. *System* instructs *Gate* to open.
Driver drives car passed the gate into the garage.
2 5. *InductionLoop* informs *System* that vehicle has left the parking spot and passed the gate.
2 6. *Gate* informs *System* that it is closed.

Parking Garage Use Case (2)

Extensions:

2a. There are no more parking spots available.

(2) 2a.1 *System* informs User that there are no more parking spots available by displaying "Garage Full" on the *Display*. Use case ends in failure. (1)

(1) 3a. *TicketPrinter* informs *System* that printing has failed.

(2) 3a.1 *System* turns on *AttendantCallLight*. Use case ends in failure.

(1) 3b. Timeout: User has not taken the ticket. (1)

3b.1 *System* turns on *AttendantCallLight*. Use case ends in failure.

5a. Timeout: User has not driven past the gate.

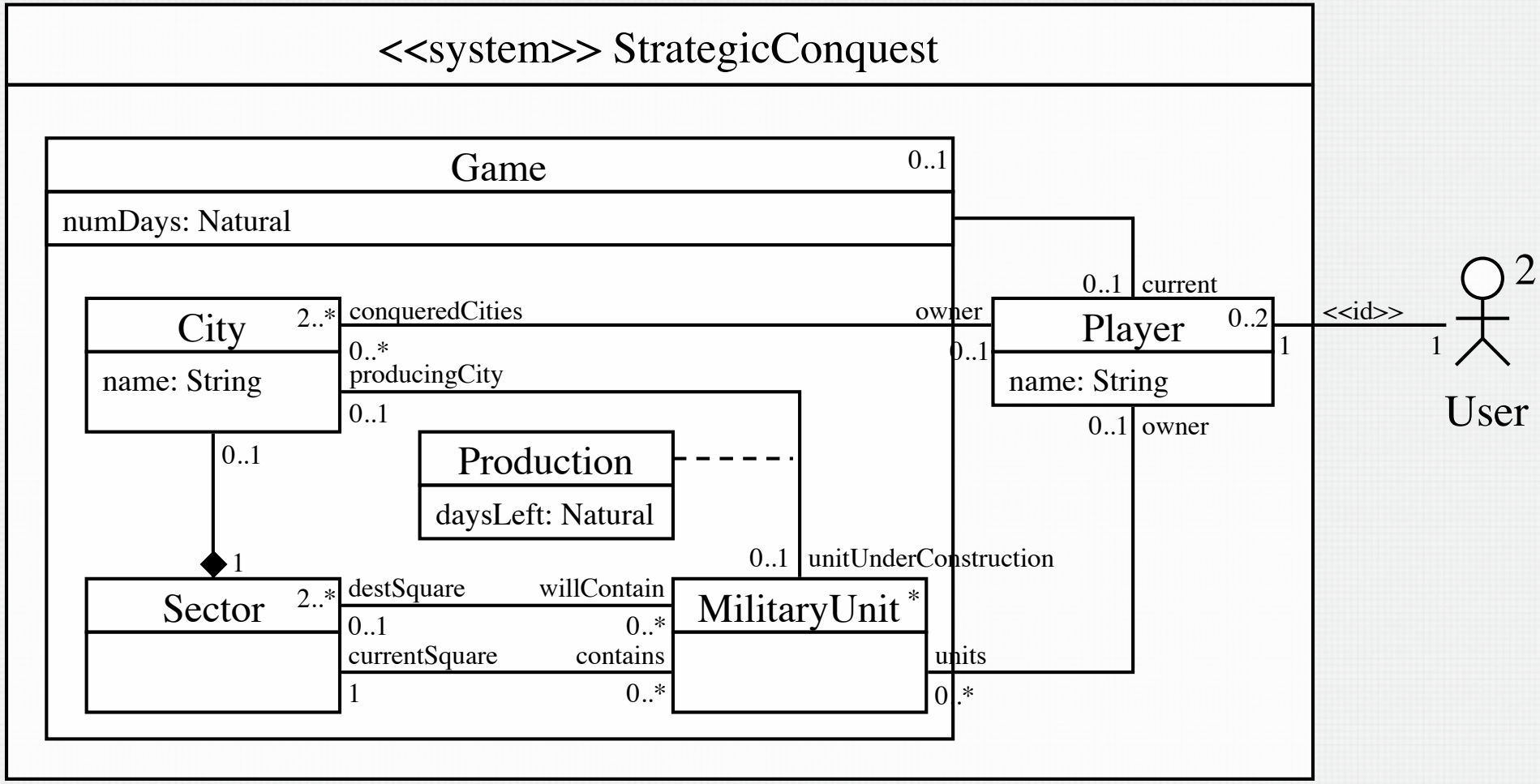
3b.1 *System* turns on *AttendantCallLight*. Use case ends in failure.

30 points total

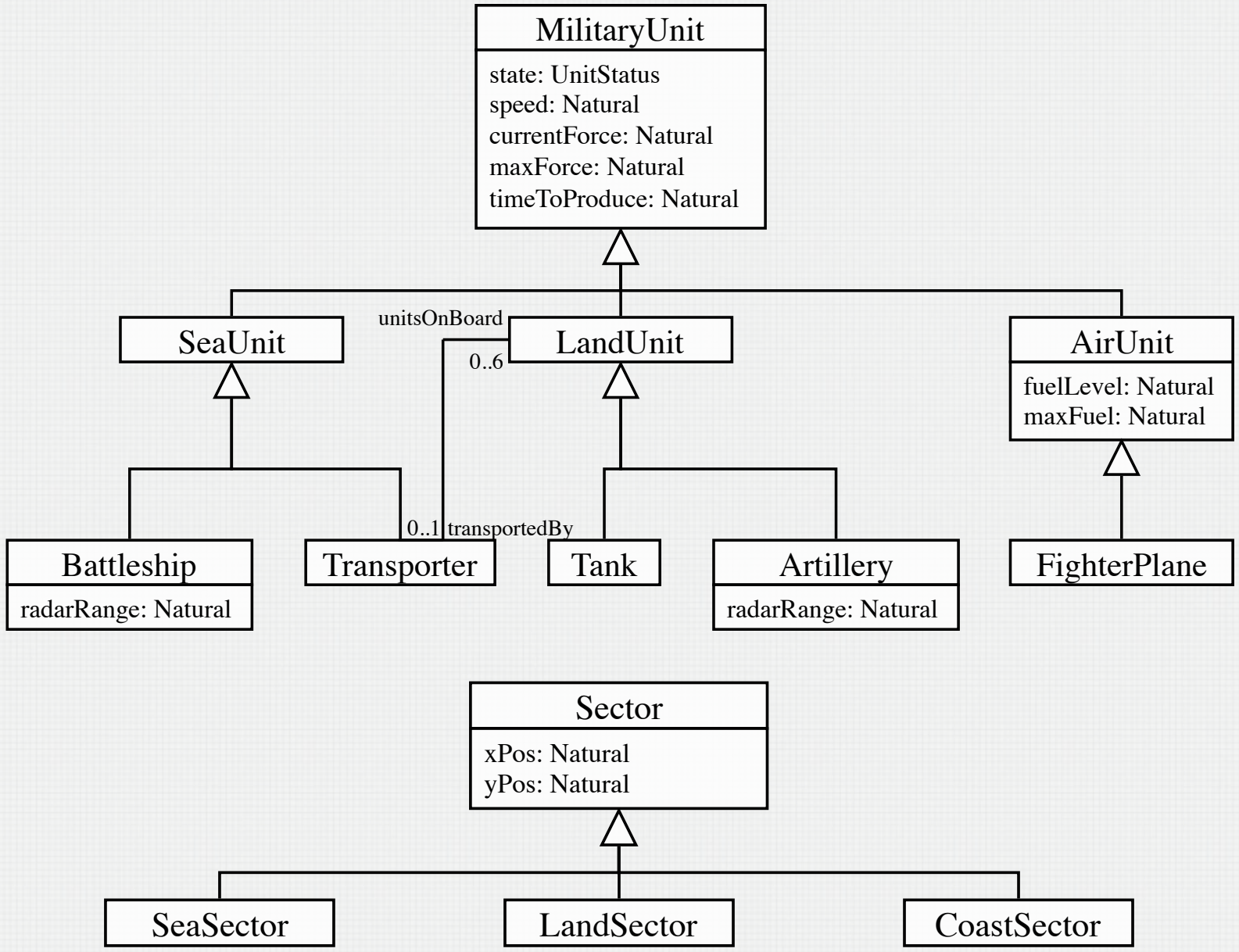
Strategic Conquest

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.

Strategic Conquest Concept Model (1)



Strategic Conquest Concept Model (2)



Strategic Conquest OCL (1)

1. Describe the following invariant in English or French:

```
context s: SeaSector
inv: s.contains→(su | su.oclIsKindOf(SeaUnit))→size() <= 1
```

(1) There can be at most one sea unit on a sea sector.

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
```

(2) Land units must always stay on land or coast sectors, except when they are transported by a transporter, in which case they are on sea or coast sectors.

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

```
context c: City
inv: c.unitUnderConstruction.oclIsKindOf(SeaUnit) implies
        c.sector.oclIsTypeOf(CoastSector)
```

(3)

Strategic Conquest OCL (2)

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

(2)
```
context t: Transporter
inv: t.unitsOnBoard→forAll(u | u.currentSquare = t.currentSquare)
```

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

(3)
```
context StrategicConquest
inv: AirUnit.allInstances()→forAll
        (a | a.speed >= (LandUnit.allInstances().speed→max()))
```

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

(4)
```
context Player
def: numTanks(n: Days) : Integer =  self.conqueredCities→select
    (c : City | c.production.daysLeft <= n and
     c.unitUnderConstruction→notEmpty() and
     c.unitUnderConstruction.oclIsTypeOf(Tank)))→size()
```

15 points total