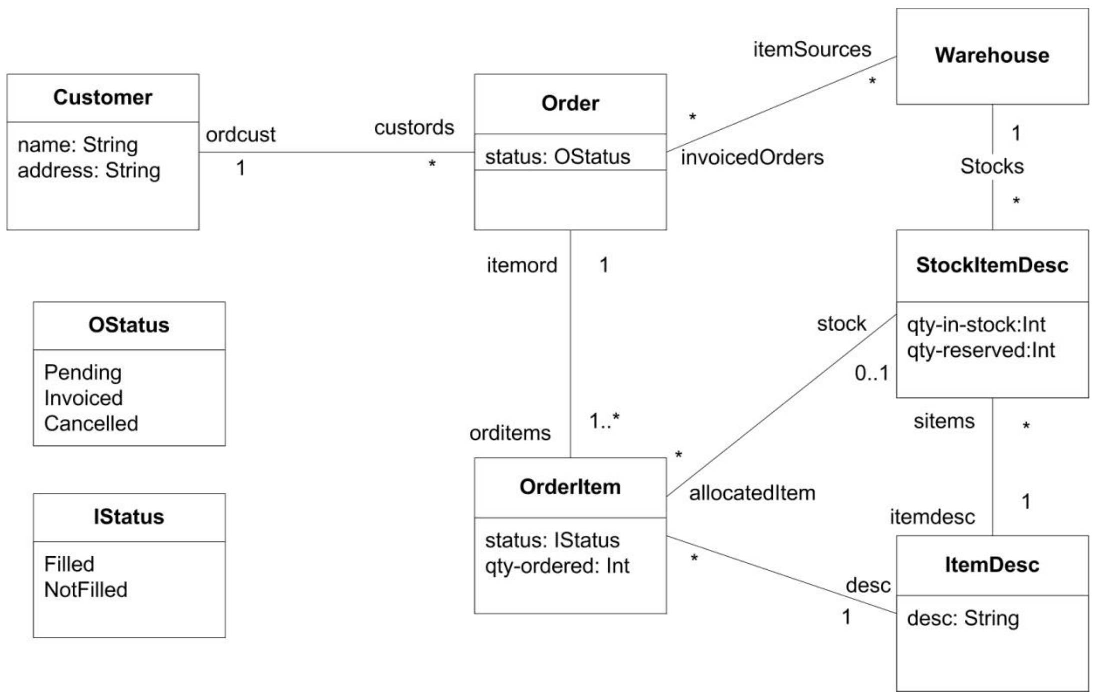
SWEN 745 – Software Modeling
Rochester Institute of Technology

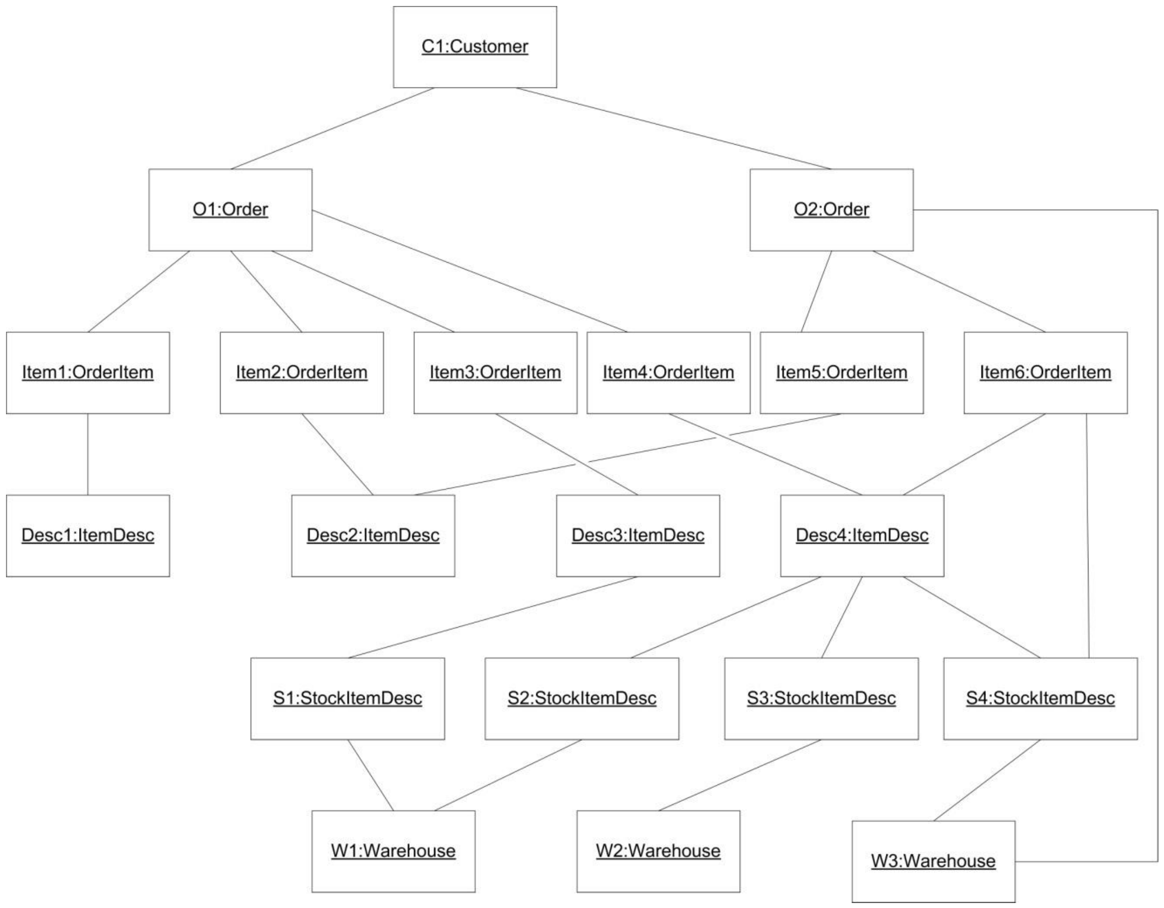
OCL - Modeling HW6 (*50 points*)

Q1:

The following is a Class Model for a simple order processing system (SOPS). In the diagram, the class Order has an attribute, *status* with an enumerated type. Status can be one of the following values: *Pending, Invoiced*, *Cancelled*. A pending order is one that has not been filled. An invoiced order is one that has been filled (i.e., stocked items have been allocated to all the order items of the order) and a cancelled order is one that has been cancelled by the customer. Each order item of an order is also associated with a status that can have one of the following values: *Filled, NotFilled*. The *Warehouse* class represents warehouses that contain products (items). For each type of product (item) stored in a warehouse (a *Warehouse* object) there is a *StockItemDesc* that contains information about the total number of products of the type stored in the warehouse and the total number of reserved products of the type. Order items that are filled are linked to the *StockItemDesc* object from which products have been allocated. An invoiced order is linked to all the warehouses (objects of *Warehouse*) from which order items are filled.



Part A: Evaluate the following OCL expressions using the SOPS object diagram given below. The expressions all return collections of objects. Use object names shown in the object diagram to refer to objects, for example, the expression C1.Order.orditems evaluates to {Item1, Item2, Item3, Item4, Item5, Item6}, that is, C1.O1.orditems = {Item1, Item2, Item3, Item4} **(10 points)**



- (2 points) O2.OrderItem.ItemDesc.StockItemDesc = {S2, S3, S4}
- (2 points) W1.StockItemDesc.itemdesc.OrderItem = {Item3, Item4, Item6}
- (3 points) O1.OrderItem → select(i | i.ItemDesc.StockItemDesc → isEmpty()) = {Item1, Item2}
- (3 points) O1.OrderItem → select(i | i.ItemDesc.StockItemDesc → size() > 1) = {Item4}

Part B: Complete the following OCL constraints (10 points):

- (3 points) If an order is invoiced then all its order items are filled.

  context Order
  inv: if self.status = OStatus::Invoiced

  then self.ordItems → forAll(item | item.status = IStatus::Filled)

- (3 points) If an order item is filled then it is linked to a *StockItemDesc* object.

  context OrderItem
  inv: if self.status = IStatus::Filled

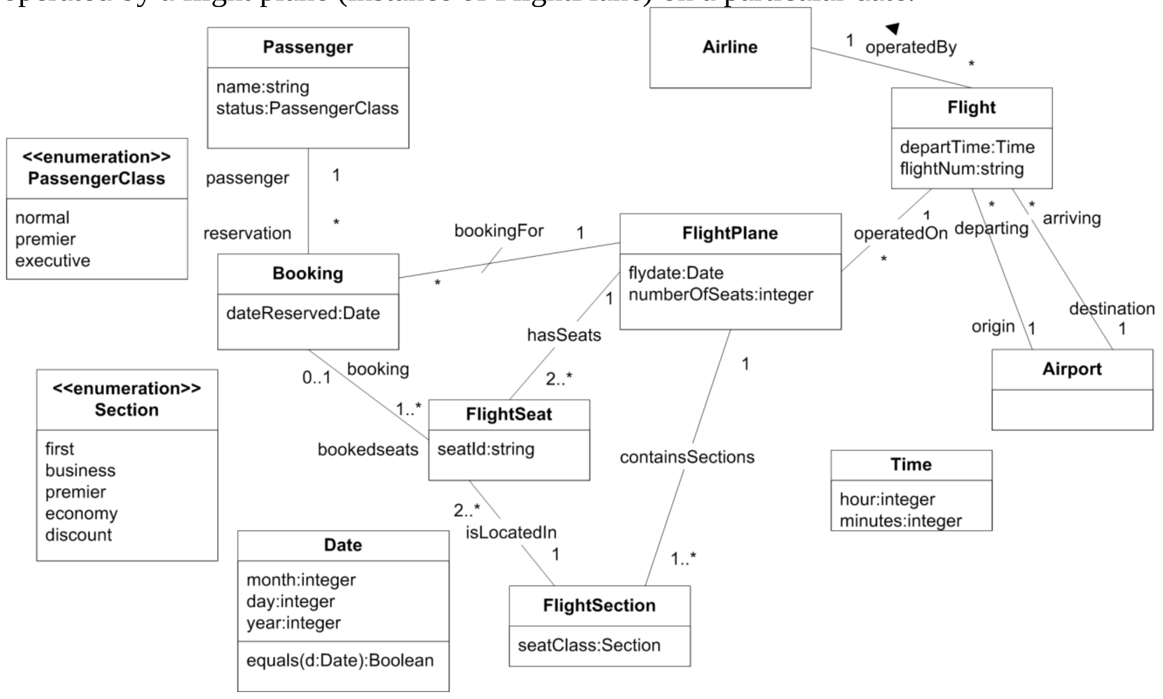
  then self.stock → collect(stock) → notEmpty()

- (4 points) If an order item is linked to a *StockItemDesc* object then the associated order is linked to the warehouse linked to the *StockItemDesc* object.

  context OrderItem
  inv: if self.stock → collect(stock) → notEmpty()

  then self.itemord.itemSources →includesAll(self.stock)

Q2:
The following questions pertain to the class model for an airline reservation system shown below. The diagram describes a system that maintains information about flights (instance of Flight) and bookings (instances of Booking). Passengers are booked on flights and are assigned seats (instances of FlightSeat) at the time of booking. A flight is operated by a flight plane (instance of FlightPlane) on a particular date.



The following are OCL statements associated with model elements in the domain model. State in English the constraints they express:

- **(5 points)**
  Context Booking
  inv: self.bookedseats.hasSeats → forAll(f |f = self.bookingFor)

  All booked seats are a booking for a FlightPlane

- **(5 points)**
  Context FlightPlane
  inv: self.numberOfSeats = self.hasSeats → size()

  The number of seats on a FlightPlane is equal to the size of the seats it has

- **(5 points)**
  Context FlightSection
  inv: (seatClass = Section::first or seatClass = Section::business or seatClass = Section::premier)
  implies isLocatedIn → collect (f.booking.passenger.status) →
  forAll(s | s = PassengerClass:: executive or s = PassengerClass:: premier)

  A seat class that is equal to first, business, or premier must be located in a Passenger Class of either executive or premier

Q3: What are operation contracts? What are the advantages and disadvantages of using OCL in specification of operation contracts? Justify your answer by giving an example. **(5 points)**

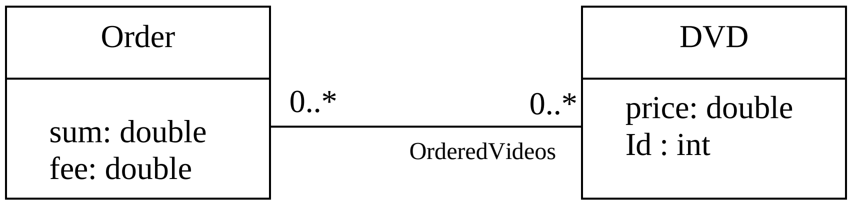
Operation contracts specify a system operation, i.e. behavior invoked by a user, in terms of pre and post conditions. For system operations that are complex, or either awkward to explain with use cases, using OCL to specify operation contracts can be extremely helpful.

An example of using OCL and operation contracts is shown below. The operation contracts, i.e. pre and post conditions, make it clear what must be true in order for a car to change gear. A Vehicle class has the attribute speed and operation changeGear() that returns true if the gear has been changed properly and the vehicle is traveling at the correct speed and false otherwise. A Gear class is referencd that has an attribute gear.

A vehicle that is moving must be in third gear, and traveling between 45 and 60 mph in order to change gears safely. After the car has changed gears, i.e. performed the gear change operation, the car must be in fourth gear and traveling at least 45 mph.

Context Vehicle::changeGears():Boolean
pre: Gear::gear = 3 and (self.speed >= 45 and self.speed <= 60)
post: result = (Gear::gear = 4 and (self.speed >= 45))

Q4: (**10 Points**)
Write the OCL equivalent of the following invariants for the given model



a. **Give an OCL invariant that specifies that the *sum* attribute cannot be negative.**

Context Order
inv: self.sum() >= 0

b. **Give an OCL invariant that specifies that the *sum* attribute will be zero if no DVDs are ordered.**

Context Order
inv: if self.OrderedVideos → isEmpty()
    then self.sum = 0
    endif

c. **Give an OCL invariant that specifies that the *sum* attribute really describes the price of the DVDs ordered.**

Context Order
inv: self.sum → self.OrderedVideos.price → sum()

d. **Give an OCL invariant that specifies that different instances of DVD have different *Id***

Context DVD
inv: DVD::allInstances() → isUnique(Id)