Software Engineering test - April 24, 2012 - solutions

1. Draw a use case diagram for a ticket distributor for a train system. The system includes two actors: a traveler, who purchases different types of tickets, and a central computer system, which maintains a reference database for the tariff. Use cases should include: `BuyOneWayTicket`, `BuyWeeklyCard`, `BuyMonthlyCard`, `UpdateTariff`. Also include the following exceptional cases: `Time-Out` (i.e., traveler took too long to insert the right amount), `TransactionAborted` (i.e., traveler selected the cancel button without completing the transaction), `DistributorOutOfChange`, and `DistributorOutOfPaper`.

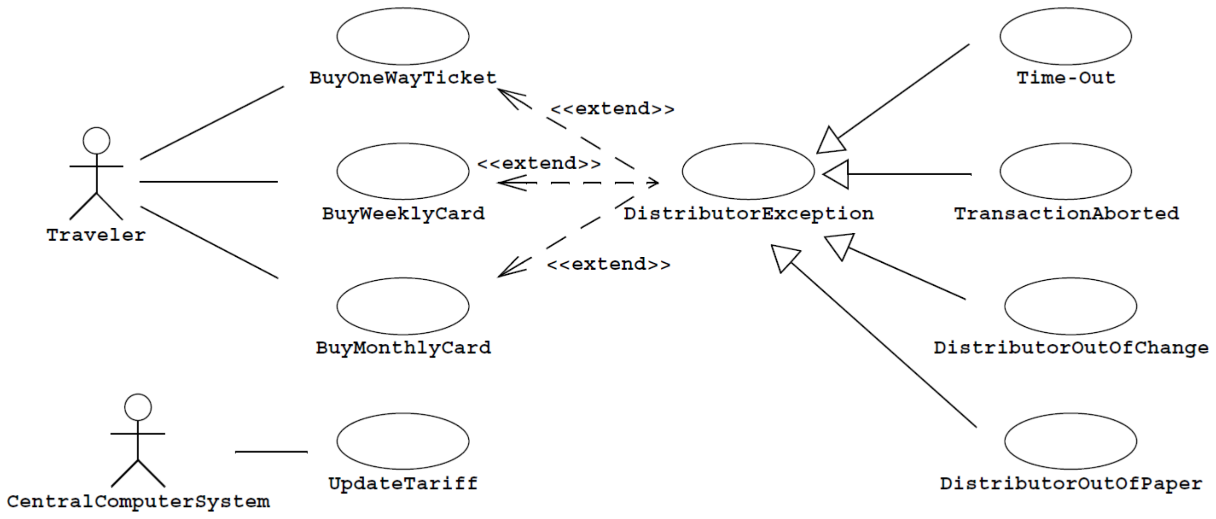


Figure 1 - A UC diagram

This questions can have several correct answers, figure above being a possible answer. The following elements should be present:

- The relationship between an actor and a use case is a communication relationship (undirected solid line).
- The relationship between exceptional use cases and common use cases is an <<extend>> relationship.
- The exceptional use cases described in the exercise only apply to the use cases invoked by the traveler.

The following elements should be present in a "good" answer:

- All exceptions apply to all traveler use cases. Instead of drawing 3x4 relationships between these use cases, an abstract use case from which the exceptional use case inherit can be used, thus reducing the number of <<extend>> relationships to3 at the cost of introducing 4 generalization relationships.
- The student can introduce exceptional use cases not specified in the exercise that apply to the CentralComputerSystem use cases.

2. Consider a file system with a graphical user interface, such as Macintosh's Finder, Microsoft's Windows Explorer, or Linux's KDE. The following objects were identified from a use case

describing how to copy a file from a floppy disk to a hard disk: `File`, `Icon`, `TrashCan`, `Folder`, `Disk`, `Pointer`. Specify which are entity objects, which are boundary objects, and which are control objects. Which is the rationale of classifying objects in these three categories? How are represented the classes corresponding to each category in a class diagram?

Entity objects: File, Folder, Disk
Boundary objects: Icon, Pointer, TrashCan
Control objects: none in this example.

3. Assuming the same file system as before, consider a scenario consisting of selecting a file on a floppy, dragging it to Folder and releasing the mouse. Identify and define at least one control object associated with this scenario.

The purpose of a control object is to encapsulate the behavior associated with a user level transaction. In this example, we identify a `CopyFile` control object, which is responsible for remembering the path of the original file, the path of the destination folder, checking if the file can be copied (access control and disk space), and to initiate the file copying.

4. Please name and briefly describe 3 modeling concepts, that do not have a 1 to 1 correspondence in programming languages. Mention how these concepts can be mapped into appropriate constructs in programming languages.

Association, associationClass, state. Depending on the multiplicity, associationsEnd are mapped into references or collectionsOfReferences. AssociationClasses are transformed into the equivalent design construction represented below and after, transformed in code.

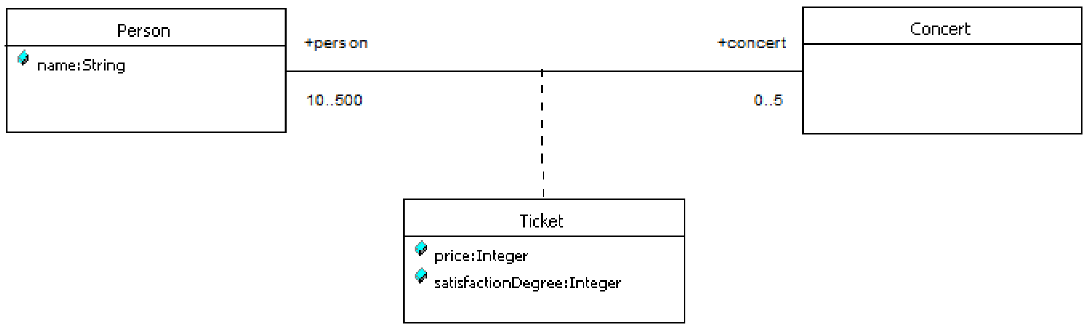


Figure 2 - AssociationClass in UML

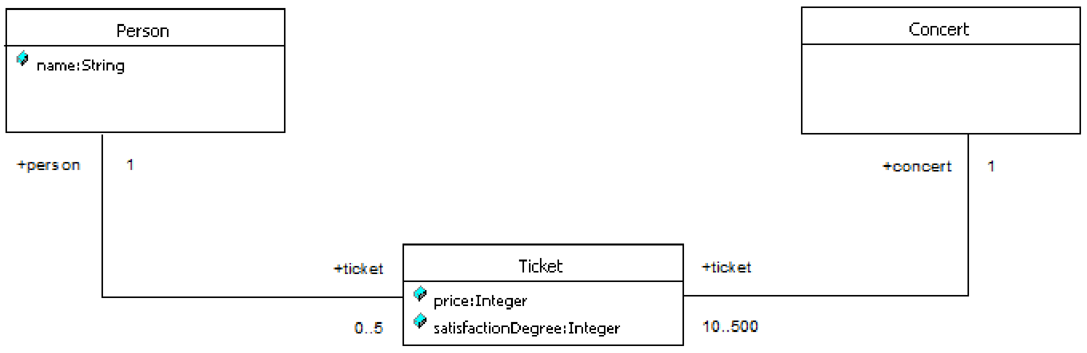


Figure 3 - Equivalent design solution for associationClasses

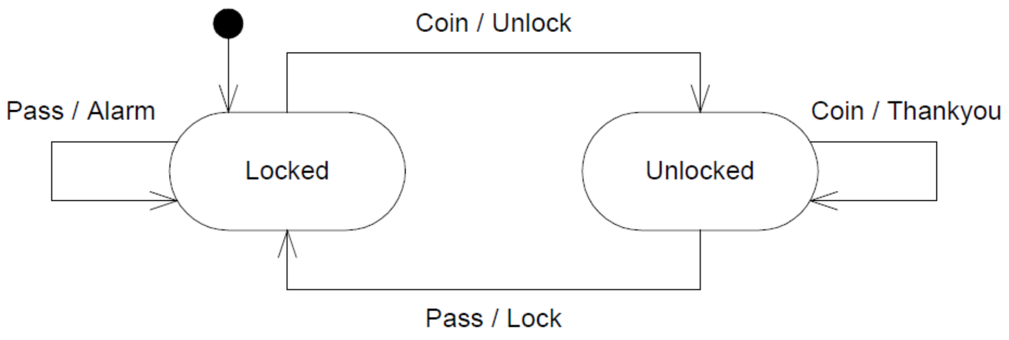


Figure 2 : Turnstile with abnormal events.

Figure 4 - A simple STD

Implementing State Machines.

There are a number of techniques for implementing a finite state machine. One of the most common is nested switch case statements. The code in Listing 1 shows the implementation of the FSM in Figure 4.

Nested switch/case FSM implementation.

```
enum State {Locked, Unlocked};
enum Event {Pass, Coin};
void Unlock();
void Lock();
void Thankyou();
void Alarm();
void Transition(Event e)
{
  static State s = Locked;
  switch(s)
  {
  case Locked:
    switch(e)
    {
      case Coin:
      s = Unlocked;
    Unlock();
    break;
    case Pass:
      Alarm();
    break;
    }
  break;
  case Unlocked:
    switch(e)
    {
    case Coin:
      Thankyou();
    break;
    case Pass:
      s = Locked;
      Lock();
    break;
    }
  break;
  }
}
```

Although this technique is common, it is not pretty. As the FSM grows, the nested switch/case statements become very difficult to read. They can go on for page after page after page of code that all looks the same. Moreover, the logic and behavior of the FSM are inextricably bound together. In C++ there is a better way. (more information in Robert C. Martin - UML Tutorial: Finite State Machines in Engineering Notebook Column - C++ Report, June 98)

5. Describe the behavior specified in Figure 5, and explain if some information is missing or not. In case of an affirmative answer, include the missing information. Name the UML concepts used in this diagram and explain the difference between the two kind of states.

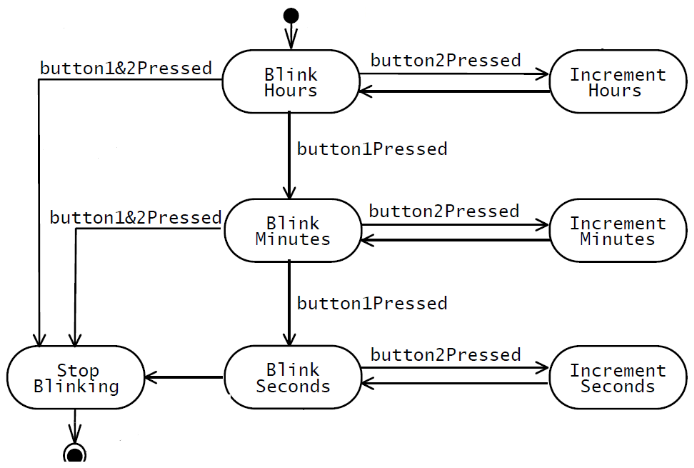


Figure 5 - UML State transition diagram

1. The following class diagrams and associated descriptions were presented as possible solution to the naive model of the Gregorian Calendar mentioned in Bruegge book:

The problems with Figure 5-24 on page 165 are related with the multiplicity of the associations. Weeks can straddle month boundaries. Moreover, the multiplicity on other associations can be tightened up: years are always composed of exactly twelve months, months do not straddle year boundaries, and weeks are always composed of seven days. Figure above depicts a possible revised model for this exercise.

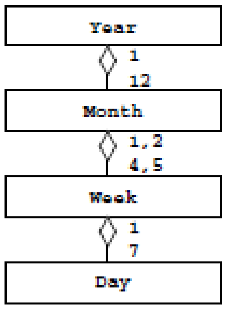


Figure 6 - Figure 5-4 A naive model of the Gregorian calendar (UML class diagram).

Consider the object model of Figure 5-24 on page 165 in the book. Using association multiplicity only, can you modify the model such that a developer unfamiliar with the Gregorian calendar could deduce the number of days in each month? Identify additional classes if necessary.

The purpose of this exercise is to show the limitation of association multiplicities. There is no complete solution to this problem. A partial solution indicating the number of days in each month is depicted in Figure 5-5. We created four abstract classes for each of the possible month lengths, 11 classes for each of the nonvariable months and two classes for the month of February. The part that cannot be resolved with association multiplicities is the definition of a leap year and that the number of days in February depends on whether the year is leap or not. In practice, this problem can solved by using informal or OCL constraints, described in more detail in Chapter 7, Object Design.

```
context Year
    def weekOfCurrentDay:
        let crrDay:Integer = self.month.day->size
        let lWeek:Integer = crrDay.div(7)
        let    weakOfCurrentDay:Integer =
           if crrDay = lWeek
               then lWeek
               else lWeek + 1
           endif
```

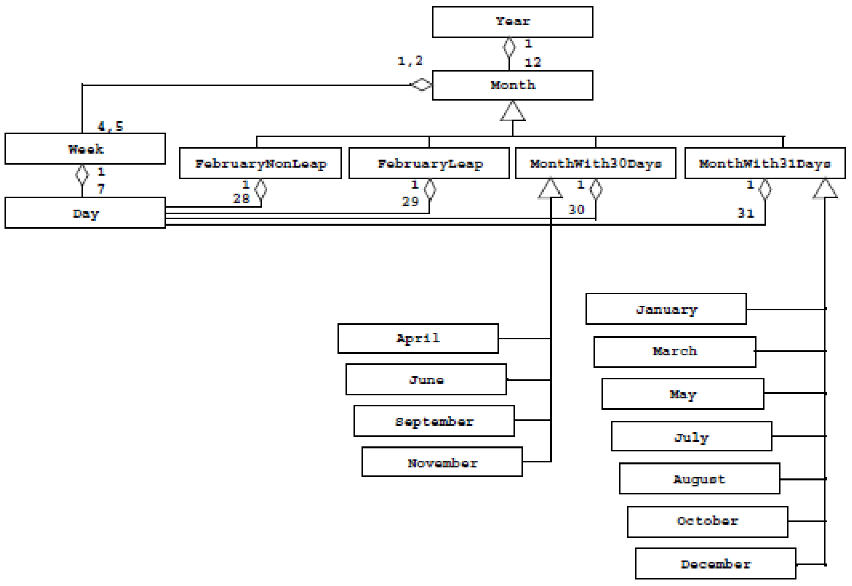
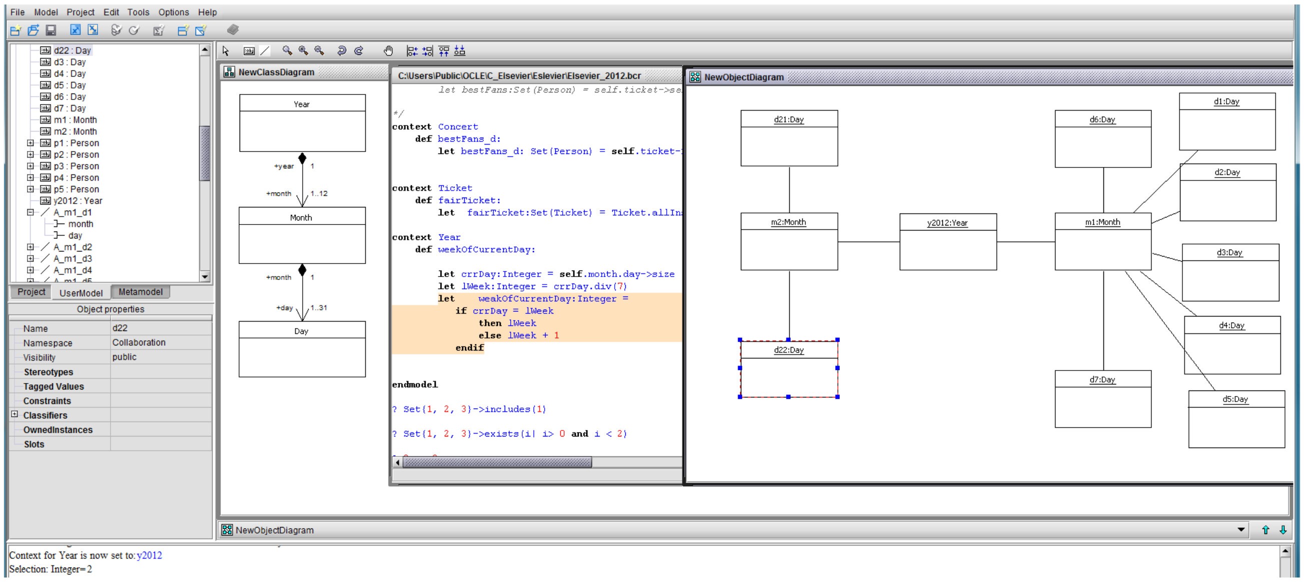


Figure 7 - Figure 5-5 Revised class diagram indicating how many days each month includes.

As mentioned above, even the model described in Figure 6, does not satisfy all the requirements. Which are in your opinion, the drawbacks of the model described in Figure 7? Using OCL, please describe a simpler model, and specify associated constraints and the observer that computes the week of the year corresponding to the current day.



Object properties

Name	d22
Namespace	Collaboration
Visibility	public
Stereotypes	
Tagged Values	
Constraints	
Classifiers	
OwnedInstances	
Slots	

```
        let bestFans:Set(Person) = self.ticket->se

*/
context Concert
    def bestFans_d:
        let bestFans_d: Set(Person) = self.ticket-


context Ticket
    def fairTicket:
        let  fairTicket:Set(Ticket) = Ticket.allIn

context Year
    def weekOfCurrentDay:

        let crrDay:Integer = self.month.day->size
        let lWeek:Integer = crrDay.div(7)
        let    weakOfCurrentDay:Integer =
           if crrDay = lWeek
               then lWeek
               else lWeek + 1
           endif


endmodel

? Set{1, 2, 3}->includes(1)

? Set{1, 2, 3}->exists(i| i> 0 and i < 2)
```

```
Context for Year is now set to:y2012
Selection: Integer=2
```

Figure 8 - A simpler model for the Gregorian calendar