5. Analysis: Solutions

5-1 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.*

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

5–2 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.

5–3 Arrange the objects listed in Exercises 5-1 and 5–2 horizontally on a sequence diagram, the boundary objects to the left, then the control object you identified, and finally, the entity objects. Draw the sequence of interactions resulting from dropping the file into a folder. For now, ignore the exceptional cases.

Figure 5-1 depicts a possible solution to this exercise. The names and parameters of the operations may vary. The diagram, however, should at least contain the following elements:

- Two boundary objects, one for the file being copied, and one of the destination folder.
- At least one control object remembering the source and destination of the copy, and possibly checking for access rights.
- Two entity objects, one for the file being copied, and one of the destination folder.

In this specific solution, we did not focus on the `Disk`, `Pointer`, and `TrashCan` objects. The Disk object would be added to the sequence when checking if there is available space. The TrashCan object is needed for scenarios in which Files or Folders are deleted.

Note that the interaction among boundary objects can be complex, depending on the user interface components that are used. This sequence diagram, however, only describes user level behavior and should not go into such details. As a result, the sequence diagram depicts a high level view of the interactions between these object, not the actual sequence of message sends that occurs in the delivered system.

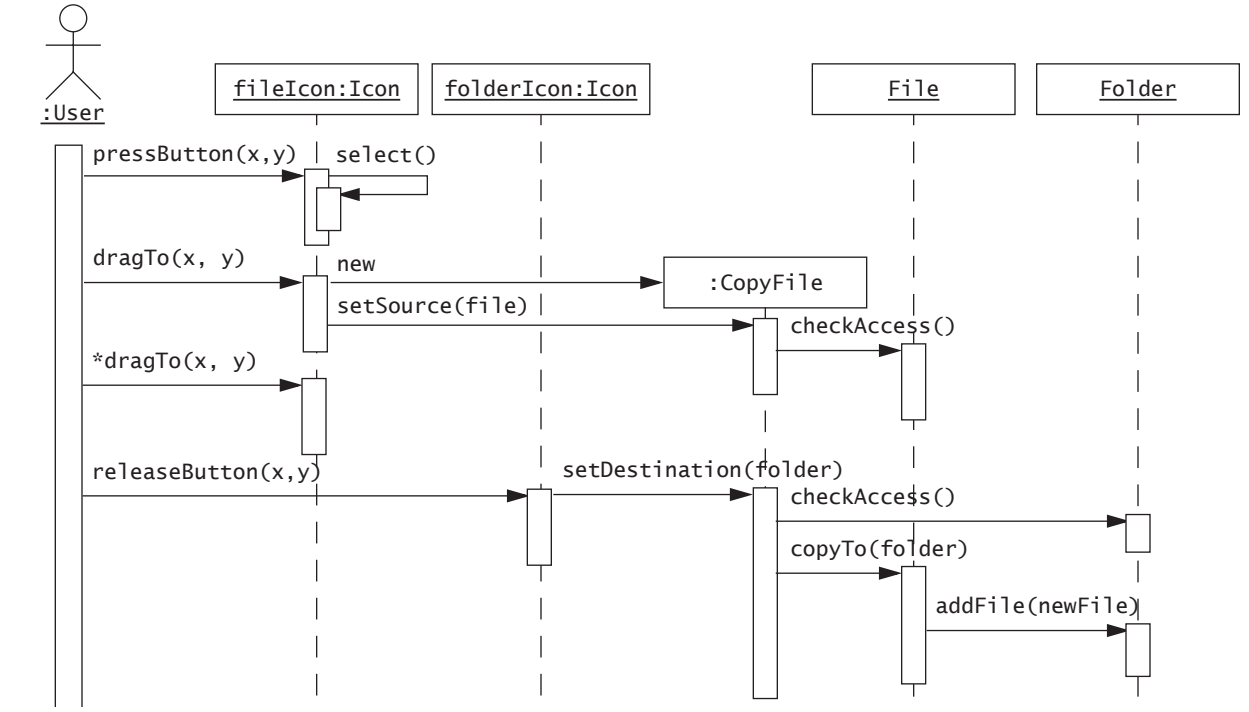

Figure 5-1 Sample solution for Exercise 5–3

5–4 Examining the sequence diagram you produced in Exercise 5–4, identify the associations between these objects.

Figure 5-2 is a possible solution for this exercises. We introduced an Item abstract class such that we can represent common associations to `Files` and `Folders`. A consequence of this decision would be to modify the sequence diagram of Figure 5-1 to use the Item class. The `CopyFile` control object has an association to the source of the `Copy`, which can be either a `File` or a `Folder`. The destination of a copy is always a `Folder`. The Icon boundary object has an association to the Item it represents and to the control object responsible for the copy.

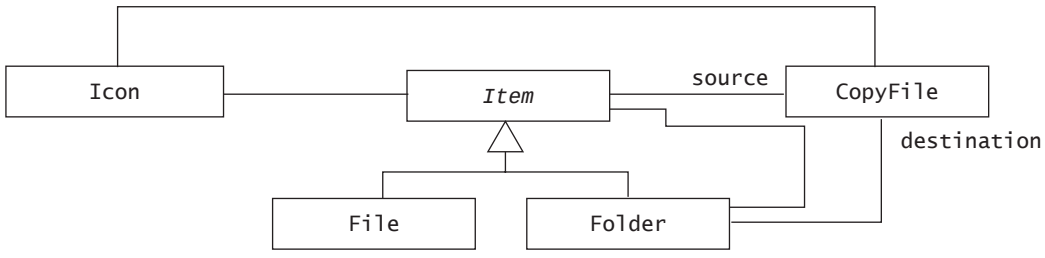

Figure 5-2 Sample solution for Exercise 5–4

5–5 Identify the attributes of each object that are relevant to this scenario (copying a file from a floppy disk to a hard disk). Also consider the exception cases "There is already a file with that name in the folder" and "There is no more space on disk."

The scenario implies that icons have positions (since they can be moved) and that they can be selected. Each `Item`, `Folder` or `File`, has a size attribute that is checked against the available space in the `Disk`. The exception "There is already a file with that name in the folder" implies that `Item` names are unique within a `Folder`. We represent this by adding a qualifier on the relationship between `Item` and `Folder`. Finally, when copying a file, we need to copy its contents, hence we add a contents attribute on the `File` object.

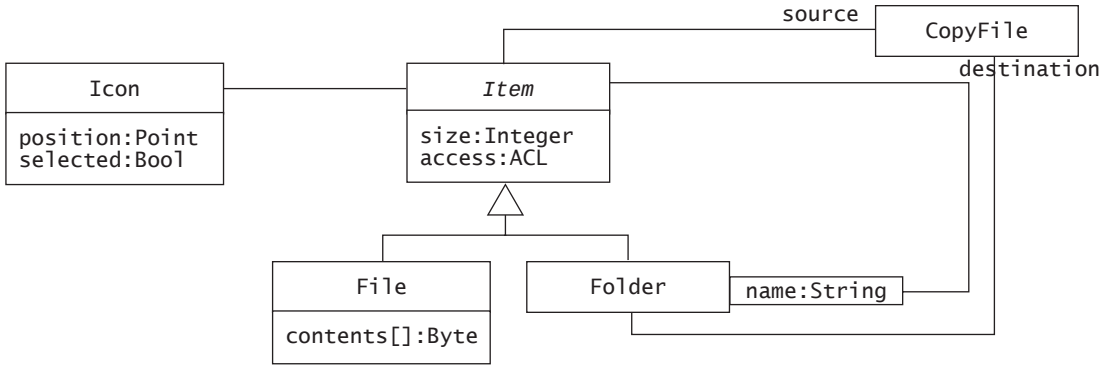

Figure 5-3 Sample solution for Exercise 5–5

5–6 Consider the object model in Figure 5-32 in the book (adapted from [Jackson, 1995]): Given your knowledge of the Gregorian calendar, list all the problems with this model. Modify it to correct each of them.

The problems with Figure 5-32 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 5-4 depicts a possible revised model for this exercise.

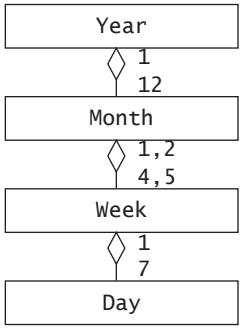

Figure 5-4 A naive model of the Gregorian calendar (UML class diagram).

5–7 Consider the object model of Figure 5-32 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 9, Object Design: Specifying Interfaces.

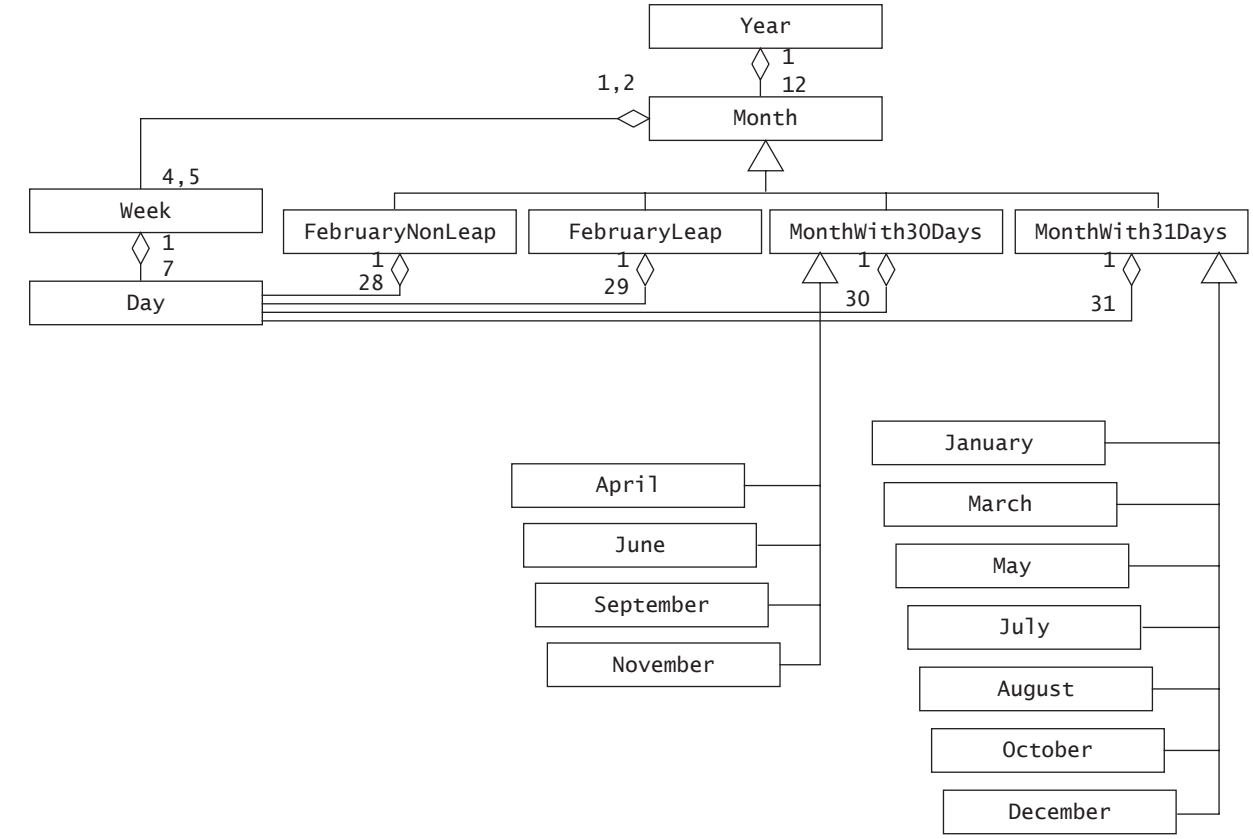

Figure 5-5 Revised class diagram indicating how many days each month includes.

5–8 Consider a traffic light system at a four-way crossroads (e.g., two roads intersecting at right angles). Assume the simplest algorithm for cycling through the lights (e.g., all traffic on one road is allowed to go through the crossroad while the other traffic is stopped). Identify the states of this system and draw a statechart describing them. Remember that each individual traffic light has three states (i.e. green, yellow, and red).

We model this system as two groups of lights, one for each road. Opposing lights have always the same value. Lights from at least one group must always be red for safety reasons. We assume there are no separate lights for left and right turns and that cycles are fixed. Figure 5-6 depicts the startchart diagram for this solution.

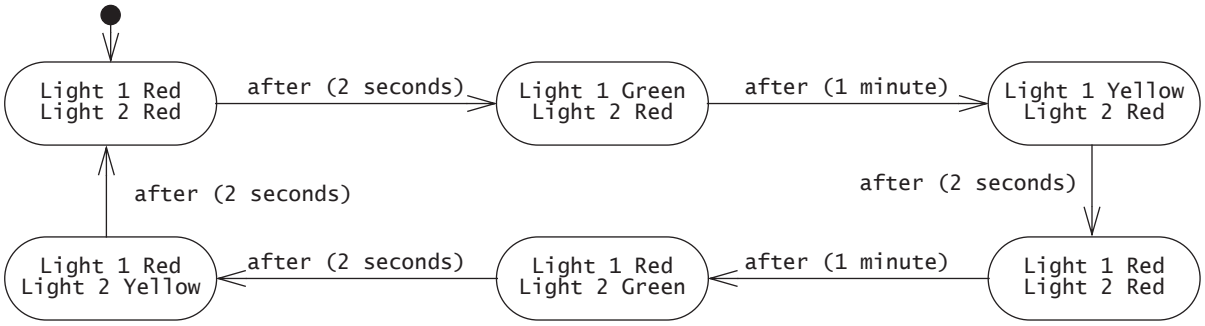

Figure 5-6 Statechart diagram for simplistic traffic light system.

5–9 From the sequence diagram Figure 2-34, draw the corresponding class diagram. Hint: Start with the participating objects in the sequence diagram.

Figure 5-7 depicts a sample solution. The student should figure out from the sequence diagram that there should be access paths among all three classes. One way to ensure that these paths exist is to introduce a top-level object `2Bwatch` which has aggregation associations with the other three classes.

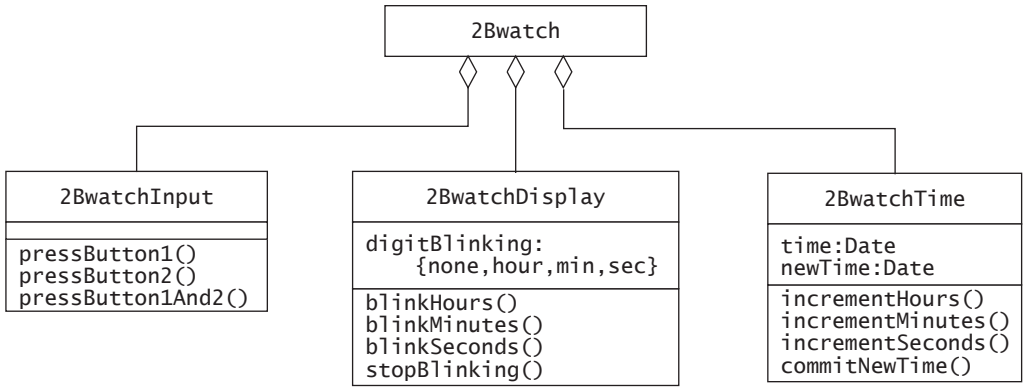

Figure 5-7 Sample solution for Exercise 5–5

5–10 Consider the addition of a nonfunctional requirement stipulating that the effort needed by `Advertisers` to obtain exclusive sponsorships should be minimized. Change the `AnnounceTournament` and the `ManageAdvertisement` use cases (solution of Exercise 4-12) so that the `Advertiser` can specify preferences in her profile so that exclusive sponsorships can be decided automatically by the system.

A new use case, ManageSponsorships, should be written and included in the ManageAdvertisement use case and define the preferences that the Advertiser can specify for automatically accepting an exclusive sponsorship. Figure 5-8 depicts a possible solution for this use case.

Use case name	ManageSponsorships
Participating actors	Initiated by Advertiser
Flow of events	1. The Advertiser requests the sponsorships area..
	2. ARENA displays the list of tournaments for which the Advertiser has exclusive sponsorships and indicates for each tournament whether the exclusive sponsorship was decided by the system or the Advertiser.
	3. The Advertiser may change the conditions under which a tournament becomes a candidate for her exclusive sponsorship, including specifying a set of leagues, a time frame, or a maximum fee that she is willing to pay for an exclusive sponsorship. The Advertiser may also select not to use the automatic sponsorship feature and be notified about such opportunities directly.
	4. ARENA records the changes of preferences for the automatic sponsorships.
Entry condition	• The Advertiser is logged into ARENA.
Exit conditions	• ARENA checks the advertisers automatic sponsorship profile for any new tournaments created after the completion of this use case.

Figure 5-8 ManageSponsorships use case.

In the AnnounceTournament use case, steps 6 & 7 are the modified as follows:

6. The system notifies the selected sponsors about the upcoming tournament and the flat fee for exclusive sponsorships. For advertisers who selected an automatic sponsorship preferences, the system automatically generates a positive answer if the fee falls within the bounds specified by the Advertiser.
7. The system communicates their answers to the LeagueOwner.

5–11 Identify and write definitions for any additional entity, boundary, and control objects participating in the AnnounceTournament use case that were introduced by realizing the change specified in Exercise 5-10.

As specified above, this exercise should result in the identification of two new entity objects:

- SponsorshipPreferences stores the time frame during which a tournament can be considered for an automatic sponsor ship and the maximum fee the Advertiser is willing to pay without being notified. This object also has associations to the leagues that should be considered during the automatic sponsorship decision.
- ExclusiveSponsorship represents an exclusive sponsorship that was awarded to an Advertiser. It is associated with the tournament and Advertiser objects involved in the sponsorship. It stores a boolean indicating whether or not the sponsorship was decided automatically.

However, the instructor may choose to extend the exercise so that the students also consider the objects participating in the ManageSponsorships use case. In this case, the student should identify a control object for the new use case and two boundary objects for displaying the exclusive sponsorships and the preferences, respectively.

5–12 Update the class diagrams of Figure 5-29 and Figure 5-31 to include the new objects you identified in Exercise 5-11.

Figure 5-9 depicts the objects discussed in Exercise 5-11 and their associations.

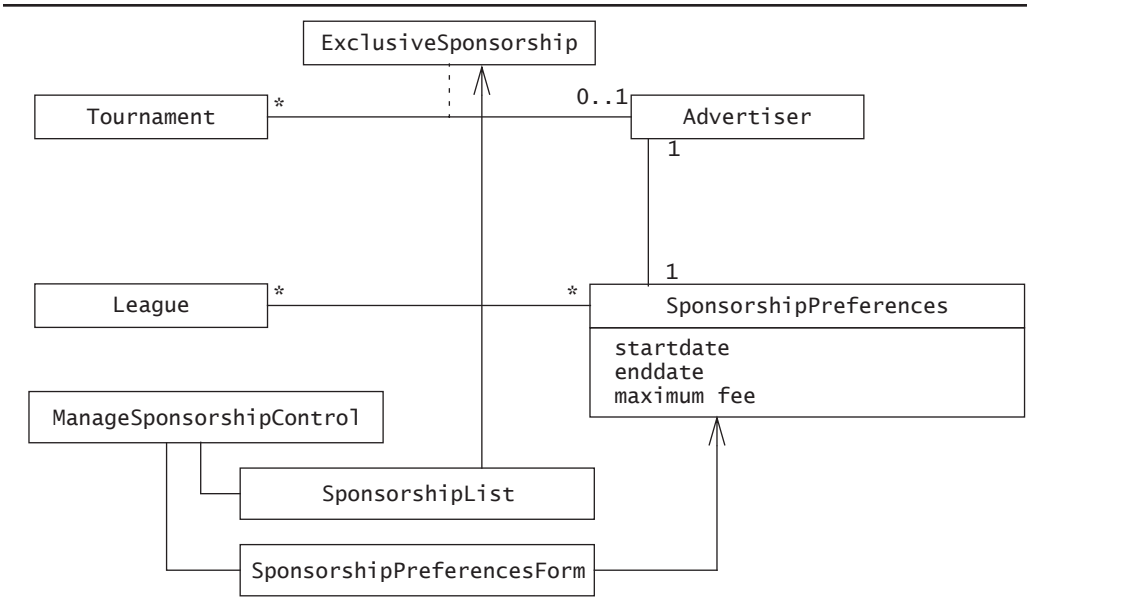

Figure 5-9 objects added after revising the `AnnounceTournament` and `ManageSponsorships` use case.

5–13 Draw a statechart describing the behavior of the `AnnounceTournamentControl` object based on the sequence diagrams of Figures 5-26 through 5-28. Treat the sending and receiving of each notice as an event that triggers a state change.

The `AnnounceTournament` is a simple linear workflow. Hence, this exercise is used to test the students knowledge of statechart diagram syntax and for choosing appropriate names for states and transitions. For modeling exercises, the instructor is advised to select more complex workflows that include decision points and concurrency, similar to the change process described in Figure 5-22.

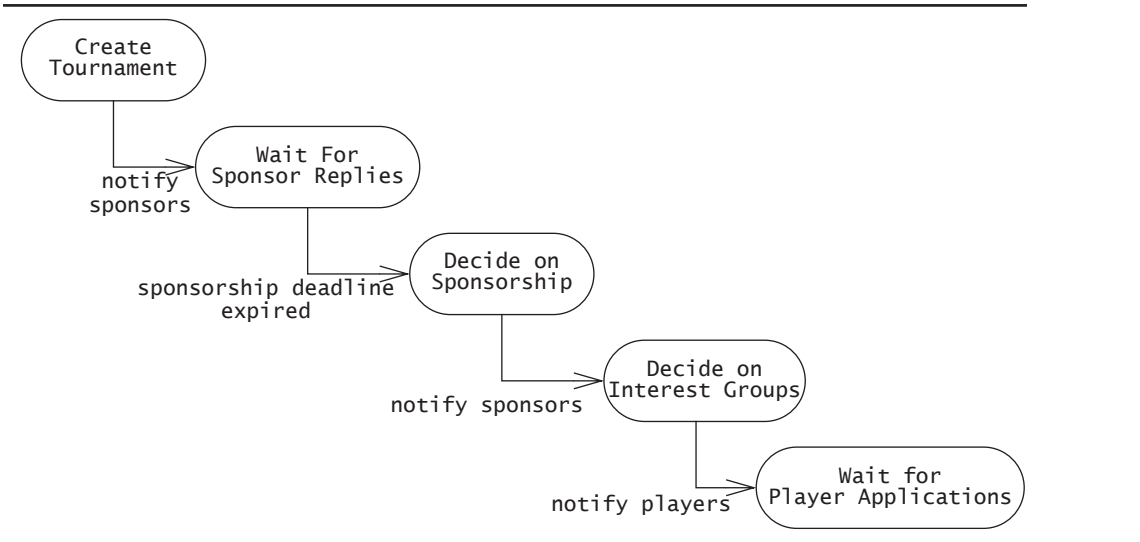

Figure 5-1 A UML startchart diagram for `AnnounceTournamentControl`.

