12

Domain Analysis

12.1 [Do not worry if your answers do not exactly match ours since you had to make assumptions about the specifications. The point of the exercise is to make you think about the examples in terms of the three aspects of modeling.]

a. bridge player. Interaction modeling, class modeling, and state modeling, in that order, are important for a bridge playing program because good algorithms are needed to yield intelligent play. The game involves a great deal of strategy. Close attention to inheritance and method design can result in significant code reuse. The interface is not complicated, so the state model is simple and could be omitted.

b. change-making machine. Interaction modeling is the most important because the machine must perform correctly. A change making machine must not make mistakes; users will be angry if they are cheated and owners of the machine do not want to lose money. The machine must reject counterfeit and foreign money, but should not reject genuine money. The state model is least important since user interaction is simple.

c. car cruise control. The order of importance is state modeling, class modeling, and interaction modeling. Because this is a control application you can expect the state model to be important. The interaction model is simple because there are not many classes that interact.

d. electronic typewriter. The class model is the most important, since there are many parts that must be carefully assembled. The interaction between the parts also must be thoroughly understood. The state model is least important.

e. spelling checker. The order of importance is class modeling, interaction modeling, and state modeling. Class modeling is important because of the need to store a great deal of data and to be able to access it quickly. Interaction modeling is important because an efficient algorithm is needed to check spelling quickly. The state model is simple because the user interface is simple: provide a chance to correct each misspelled word that is found.

f. **telephone answering machine**. The order of importance is state modeling, class modeling, and interaction modeling. The state diagram is non-trivial and important to the behavior of the system. The class model shows relationships between components that complement the state model. There is little computation or state diagrams to interact, so the interaction model is less important.

12.2 [Keep in mind that the requirements are incomplete, thus so are the class models.]

a. Each transmission has a baud rate and a transmission protocol. Transmissions are dynamically initiated and terminated. Each transmission has one source and one destination communication port. The source sends data and the destination receives data. Each port is dedicated to one transmission or is idle. (Note that the structure of the model does not strictly enforce the constraint that a port is associated with at most one transmission. Our model would permit both an *Input* and *Output* association when only one applies. We do not consider it worthwhile to modify the model to capture this constraint.) Each port has a name and is associated with one computer. A computer may communicate through many ports. A port transmits data through a communication line that may serve many ports.

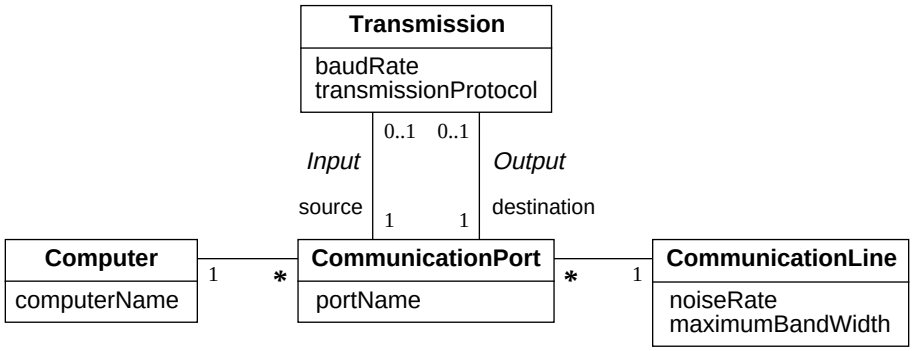

Figure A12.1 Class diagram for a data transfer system

b. A design of a part produces many tapes, each of which is used to control one N/C machine. Each N/C machine may run different tapes over the course of a day. A N/C machine manufactures many parts, and a part may be machined by several N/C machines. Each tape is dedicated to a single part design.

c. Each desktop publishing document contains many text and graphics primitives. (In reality for any reasonable implementation of a desktop publishing system, a document would consist of several intermediate levels of structure. We do not show such structure because it is not obvious from the stated requirements.) Graphics primitives include lines, squares, rectangles, polygons, circles, and ellipses. The *print* operation on the *Document* class interacts with *Printer* objects.

1. The only exception is that objects may not have the same value of the object ID attribute. The object ID is

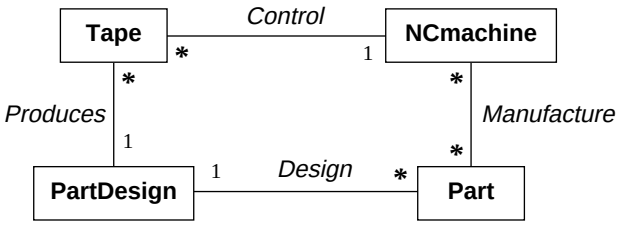

Figure A12.2 Class diagram for a parts machining automation system

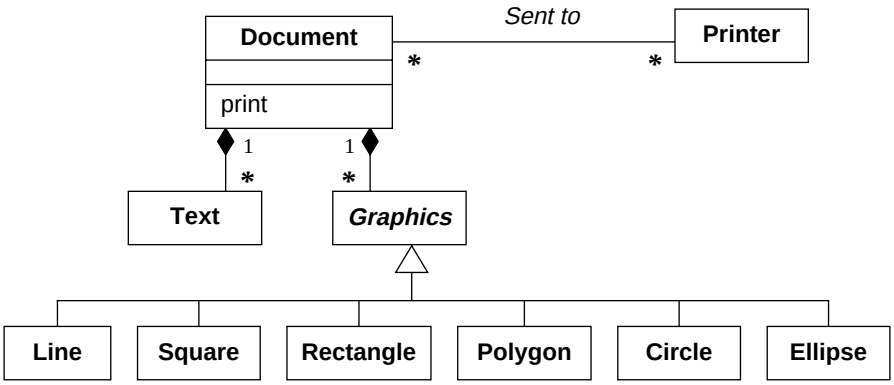

Figure A12.3 Class diagram for a desktop publishing system

d. For each run of the nonsense generator, the user can set two attribute values: *orderOfImitation* and *desiredOutputLength*. The nonsense generator takes an input document and generates an output document. An input document can be used for multiple runs of generating output. A document may or may not be produced from an output.

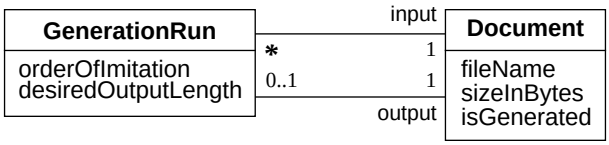

Figure A12.4 Class diagram for a nonsense generator

e. An electronic mail system must service multiple users and handle many individual mail messages. Each user has a set of email default parameters. Users may send and receive multiple mail messages. A mail message is sent by one user and can be received by multiple users. It is often convenient to send a mail message to multiple users via a distribution list. A distribution list is a list of users that has been predefined and named. A file

may contain many mail messages. Users can perform many operations on mail such as save to file, print, send, and forward. A user has many computer accounts but receives mail at only one of these accounts. Each file is owned by some computer account. Multiple accounts may be supported by the same computer.

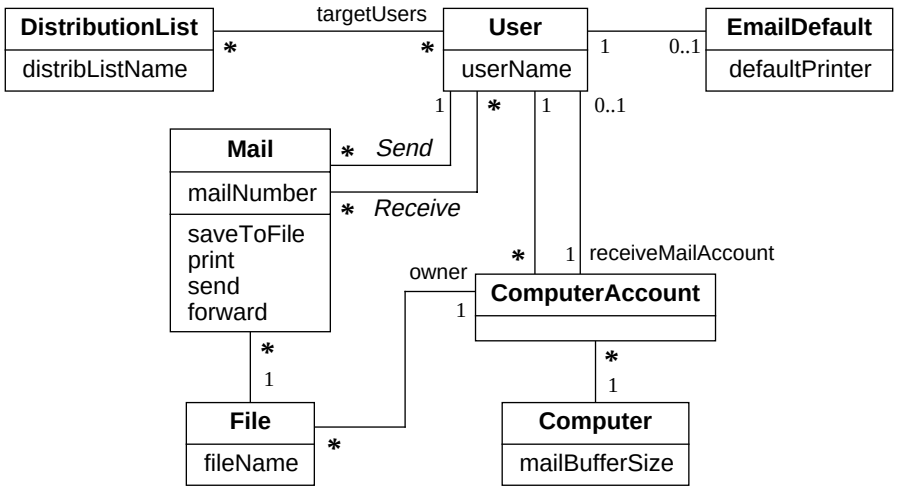

Figure A12.5 Class diagram for an electronic mail system

12.3 The following tentative classes should be eliminated.

- **Redundant classes**. *SelectedObject*, *SelectedLine*, *SelectedBox*, *SelectedText* (redundant with *Selection*), *Connection* (redundant with *Link*).
- **Irrelevant classes**. *Computer* (it is implicit that we are developing a model for the purpose of computer implementation).
- **Vague classes**. *GraphicsObject* (not sure precisely what this is; we do not need it as we have more specific classes).
- **Attributes**. *position*, *length*, *width*, *fileName*, *lineSegment*Coordinate, *name*, *origin*, *scaleFactor*.
- **Implementation constructs**. *x-coordinate*, *y-coordinate* (what about polar coordinates?, also they are really attributes), *Menu* (but you could argue about whether *Menu* should be a class), *Mouse*, *Button*, *Popup*, *MenuItem* (a manner of implementing menus), *CornerPoint*, *EndPoint* (there are other ways to specify a *Box*), *Character* (an implementation construct for *Text*).

After eliminating improper classes we are left with *Line*, *Link*, *Collection*, *Selection*, *Drawing*, *DrawingFile*, *Sheet*, *Point*, *Box*, *Buffer*, and *Text*.

12.4 We only need to prepare a data dictionary for proper classes.

- **Line**—a graphical entity that connects two points; actually a line segment. Lines may be horizontal or vertical.
- **Link**—a connection path between two boxes. A link is represented as a series of joined alternating vertical and horizontal lines.
- **Collection**—a set of two or more lines and boxes, not necessarily connected. A collection is a grouping defined by the user and manipulated as a unit for moves and deletes, for example.
- **Selection**—a set of boxes and links that the user selects with a mouse.
- **Drawing**—a set of sheets that contain boxes and links.
- **DrawingFile**—a file that contains a stored representation of a drawing.
- **Sheet**—a set of boxes and links that fit on a standard piece of printer paper. Each box and link is on exactly one sheet.
- **Point**—a location on a sheet.
- **Box**—a rectangle optionally containing a single string of text.
- **Buffer**—a temporary holder for copied and cut selections. The paste operation uses the buffer contents.
- **Text**—a sequence of characters on a single horizontal line located within a box.

12.5 The following tentative associations should be eliminated because they are between eliminated classes:

- *A box has a position*. (*Position* is an attribute that has been eliminated. Replace by *a box has a point*.)
- *A character string has a location*. (*Location* is an attribute. We are using the term *text* and not *character string*.)
- *A line has length. (Length* is an attribute that has been eliminated.)
- *A line is a graphical object*. (For this problem, we do not consider *GraphicalObject* as a class worth modeling.)
- *A point is a graphical object*. (For this problem, we do not consider *GraphicalObject* as a class worth modeling.)
- *A point has an x-coordinate*. (*X-coordinate* is an attribute that has been eliminated.)
- *A point has a y-coordinate*. (*Y-coordinate* is an attribute that has been eliminated.)

The following tentative associations are irrelevant or implementation artifacts:

- *A character string has characters*. (This is not important enough to include in the model.)
- *A box has a character string*. (This is the same as *a box has text*.)

The following tentative associations are actions:

- *A box is moved*.
- *A link is deleted*.
- *A line is moved*.

The following associations are derived:

- *A link has points*.
- *A link is defined by a sequence of points*. (Replace by *a link corresponds to one or more lines*.)

The following associations were missing from the list given in the exercise:

- *A drawing has one or more sheets.*
- *A drawing is stored in a drawing file*.

We are left with the following correct associations and generalizations: *a box has a point, a box has text, a link logically associates two boxes, a link corresponds to one or more lines, a selection or a buffer or a sheet is a collection, a collection is composed of links and boxes, a line has two points, a drawing has one or more sheets,* and *a drawing is stored in a drawing file.*

12.6 We use a combination of the OCL and pseudocode to express our queries.

a. Find all selected boxes and links.

```
Selection::retrieveBoxesLinks (boxes, links)
   boxes:= self.box;
   links:= self.link;
```

b. Given a box, determine all other boxes that are directly linked to it.

```
Box::retrieveDirectBoxes () returns set of boxes
   boxes:= createEmptySet;
   for each link in self.link
      add set link.box to set boxes;
         /* link.box has two elements: */
         /* self & the linked box.     */
   end for each link
   remove self from set boxes;
   return boxes;
```

c. Given a box, find all other boxes that are directly or indirectly linked to it.

```
Box::retrieveTransitiveClosureBoxes ()
returns set of boxes
   boxes:= createEmptySet;
   return self.TCloop (boxes);
Box::TCloop (boxes) returns set of boxes
   add self to set boxes;
   for each link in self.link
```

```
      for each box in link.box
         /* 2 boxes are associated with a link */
         if box is not in boxes then
            box.TCloop(boxes);
         end if
      end for each box
   end for each link
```

d. Given a box and a link, determine if the link involves the box. This operation is symmetrical, thus it is arbitrary whether we assign it to class *Box* or class *Link*.

```
Box::checkConnection (givenLink) returns boolean
   if self is in givenLink.box then return true
   else return false
   end if
```

e. Given a box and a link, find the other box logically connected to the given box through the other end of the link. This operation is symmetrical, thus it is arbitrary whether we assign it to class *Box* or class *Link*.

```
Box::retrieveOtherBox (givenLink) returns box
   if self is not in givenLink.box then
      error;
      return nil
   else
      boxes:= givenLink.box;
      remove self from set boxes;
      return (boxes.firstElement);
         /* set boxes has a single element */
   end if
```

f. Given two boxes, determine all links between them.

```
Box::retrieveCommonLinks (givenBox2)
returns set of links
   links1:= self.link;
   links2:= givenBox2.link;
   return (links1 intersect links2);
```

g. Given a selection, determine which links are “bridging” links.

```
Selection::retrieveBridgingLinks ()
returns set of links
   selectedBoxes:= self.box;
   bridges:= createEmptySet;
   for each box in selectedBoxes
      for each link in box.link
         otherBox:= box.retrieveOtherBox (link);
         if otherBox is not in selectedBoxes then
            add link to set bridges;
         end if
```

```
      end for each link
   end for each box
   return bridges;
```

12.7 Figure E12.3 promotes the association between *Box* and *Link* to a class. The connection between *Box* and *Link* should be modeled as a class if it has identity, important behavior, and relationships to other classes.

a. Find all selected boxes and links. Same answer as Exercise 12.6a.

b. Given a box, determine all other boxes that are directly linked to it.

```
Box::retrieveDirectBoxes () returns set of boxes
   boxes:= createEmptySet;
   for each link in self.connection.link
      add set link.connection.box to set boxes;
         /* link.connection.box has two elements: */
         /* self & the linked box.                */
   end for each link
   remove self from set boxes;
   return boxes;
```

c. Given a box, find all other boxes that are directly or indirectly linked to it.

```
Box::TCloop (boxes) returns set of boxes
   add self to set boxes;
   for each link in self.connection.link
      for each box in link.connection.box
         /* 2 boxes are associated with a link */
         if box is not in boxes then
            box.TCloop(boxes);
         end if
      end for each box
   end for each link
```

d. Given a box and a link, determine if the link involves the box. This operation is symmetrical, thus it is arbitrary whether we assign it to class *Box* or class *Link*.

```
Box::checkConnection (givenLink) returns boolean
   if self is in givenLink.connection.box then
      return true
   else return false
   end if
```

e. Given a box and a link, find the other box logically connected to the given box through the other end of the link. This operation is symmetrical, thus it is arbitrary whether we assign it to class *Box* or class *Link*.

```
Box::retrieveOtherBox (givenLink) returns box
   if self is not in givenLink.connection.box then
      error;
      return nil
```

```
        else
           boxes:= givenLink.connection.box;
           remove self from set boxes;
           return (boxes.firstElement);
              /* set boxes has a single element */
        end if
```

f. Given two boxes, determine all links between them.

```
Box::retrieveCommonLinks (givenBox2)
returns set of links
   links1:= self.connection.link;
   links2:= givenBox2.connection.link;
   return (links1 intersect links2);
```

g. Given a selection, determine which links are "bridging" links.

```
Selection::retrieveBridgingLinks ()
returns set of links
   selectedBoxes:= self.box;
   bridges:= createEmptySet;
   for each box in selectedBoxes
      for each link in box.connection.link
         otherBox:= box.retrieveOtherBox (link);
         if otherBox is not in selectedBoxes then
            add link to set bridges;
         end if
      end for each link
   end for each box
   return bridges;
```

12.8 The following classes require state diagrams: *Buffer* and *Selection*.

The *Buffer* is used for *copy*, *cut*, and *paste* operations. The state of the buffer is simple: either it is empty or it is full.

The *Selection* state diagram indicates whether one or more objects are selected. Thus there are two states: *Something selected* and *Nothing selected*. The class model contains important information for the *Something selected* state: precisely which boxes and links are selected. The *pick* operation would need to distinguish between picking the first object selected and picking subsequent objects.

12.9 The following tentative classes should be eliminated.

- **Redundant classes**. *Child*, *Contestant*, *Individual*, *Person*, *Registrant* (all are redundant with *Competitor*).
- **Vague or irrelevant classes**. *Back*, *Card*, *Conclusion*, *Corner*, *IndividualPrize*, *Leg*, *Pool*, *Prize*, *TeamPrize*, *Try*, *WaterBallet*.
- **Attributes**. *address*, *age*, *averageScore*, *childName*, *date*, *difficultyFactor*, *netScore*, *rawScore*, *score*, *teamName.*

- **Implementation constructs**. *fileOfTeamMemberData, listOfScheduledMeets, group, number*.
- **Derived class**. *ageCategory* is readily computed from a competitor's age.
- **Operations**. *computeAverage, register*.
- **Out of scope**. *routine*.

After eliminating improper classes we are left with *Competitor*, *Event*, *Figure*, *Judge*, *League*, *Meet*, *Scorekeeper*, *Season*, *Station*, *Team*, and *Trial*.

12.10 We only need to prepare a data dictionary for proper classes.

- **Competitor**—a child who participates in a swimming meet.
- **Event**—a figure performed at a swimming meet.
- **Figure**—a standard sequence of actions performed by each competitor.
- **Judge**—a person who rates the quality of a trial.
- **League**—a group of teams that compete against one another.
- **Meet**—a series of events that are performed by two or more swimming teams on a particular date at a specific site.
- **Scorekeeper**—a person who records scores.
- **Season**—a series of swimming meets that occur in the same summer.
- **Station**—a location around a swimming pool where each contestant performs a figure. All events for a figure at a given meet are held at one station.
- **Team**—a group of children who compete. Each child belongs to exactly one team. A team is treated as a unit for the purpose of awarding team prizes.
- **Trial**—an attempt by a competitor to perform an event.

12.11 The following tentative associations should be eliminated because they are between eliminated classes:

- *A competitor is assigned a number*. (*Number* is an attribute that has been eliminated.)
- *Routines are events*.

The following tentative associations are implementation artifacts:

- *Competitors are split into groups*.
- *The highest score is discarded*.
- *The lowest score is discarded*.
- *Prizes are based on scores*.

The following tentative associations are actions:

- *A competitor registers*.

- *A number is announced.*
- *Raw scores are read.*
- *Figures are processed.*

The following association is derived:

- *A trial of a figure is made by a competitor*. (Replace by *a figure has many events*, *an event has many trials*, and *a competitor performs many trials*.)

The following associations were missing from the list given in the exercise:

- *A judge may serve at several stations.*
- *A station has many scorekeepers.*
- *Scorekeepers may work at more than one station.*

We are left with the following correct associations and generalizations: *a season consists of several meets, a meet consists of several events, several stations are set up at a meet, several events are processed at a station, several judges are assigned to a station, figures are types of events, a league consists of several teams, a team consists of several competitors, a figure has many events, an event has many trials, a competitor performs many trials, a trial receives several scores from the judges, a judge may serve at several stations, a station has many scorekeepers*, and *scorekeepers may work at more than one station.*

12.12 We use a combination of the OCL and pseudocode to express our queries.

[Some of our answers to these problems traverse a series of links (such as *season.meet.event.trial* in answer d). Section 15.10.1 explains that each class should have limited knowledge of a class model and that operations for a class should not traverse associations that are not directly connected to it. We have violated this principle here to simplify our answers. A more robust answer would define intermediate operations to avoid these lengthy traversals.]

a. Find all the members of a given team.

```
Team::retrieveTeamMembers ()
returns set of competitors
   return self.competitor;
```

b. Find which figures were held more than once in a given season.

```
Season::findRepeatedFigures () returns set of figures
   answer:= createEmptySet;
   figures:= createEmptySet;
   for each event in self.meet.event
      if event.figure in figures then
         add event.figure to set answer;
      else add event.figure to set figures;
      endif
   end for each event;
   return answer;
```

c. Find the net score of a competitor for a given figure at a given meet. There are several ways to answer this question, one of which is listed below.

```
Competitor::findNetScore (figure, meet)
returns netScore
   event:= meet.event intersect figure.event;
      /* the above code should return exactly one    */
      /* event (otherwise there is an implementation */
      /* error). This is a constraint implicit in the */
      /* problem statement that is not expressed in  */
      /* the class model.                            */
   trial := event.trial intersect self.trial;
   if trial == NIL then return ERROR
   else return trial.netScore;
   end if
```

d. Find the team average over all figures in a given season.

```
Team::findAverage (season) returns averageScore
   trials:= season.meet.event.trial intersect
    self.competitor.trial;
   if trials == NIL then return ERROR
   else
      sum:=0; count:=0;
      for each trial in trials
         add trial.netScore to sum; count++;
      end for each trial
      return sum/count;
   end if
```

e. Find the average score of a competitor over all figures in a given meet.

```
Competitor::findAverage (meet) returns averageScore
   trials:= meet.event.trial intersect
      self.trial;
   if trials == NIL then return ERROR
   else
      compute average as in answer (d)
      return average;
   end if
```

f. Find the team average in a given figure at a given meet.

```
Team::findAverage (figure, meet) returns averageScore
   trials:= meet.event.trial intersect
      figure.event.trial intersect
      self.competitor.trial;
   if trials == NIL then return ERROR
   else
      compute average as in answer (d)
      return average;
   end if
```

g. Find the set of all individuals who competed in any events in a given season.

```
Season::findCompetitorsForAnyEvent ()
returns set of competitors
   return self.meet.event.trial.competitor;
```

h. Find the set of all individuals who competed in all of the events held in a given season.

```
Season::findCompetitorsForAllEvents ()
returns set of competitors
   answer:= createEmptySet;
   events:= self.meet.event;
   competitors:= self.meet.event.trial.competitor;
   for each competitor in competitors
      if competitor.trial.event = events
         /* test for set equality */
         then add competitor to set answer;
      end if
   end for each competitor
   return answer
```

i. Find the judges who judged a given figure in a given season.

```
Figure::findJudges (season) returns set of judges
   stations:= season.meet.station intersect
      self.event.station;
   return stations.judge->asSet;
```

j. Find the judge who awarded the lowest score during a given event.

```
Event::findLowScoringJudge () returns judge
   lowScore:= INFINITY;
   answer:= NIL;
   for each trial in self.trial
      for each judge in trial.judge
         if link(trial,judge).rawScore < lowScore then
            lowScore:= link(trial,judge).rawScore;
            answer:=judge;
         end if
      end for each judge
   end for each trial
   return answer;
```

k. Find the judge who awarded the lowest score for a given figure.

```
Figure::findLowScoringJudge () returns judge
   lowScore:= INFINITY;
   answer:= NIL;
   for each trial in self.event.trial
      for each judge in trial.judge
         if link(trial,judge).rawScore < lowScore then
            lowScore:= link(trial,judge).rawScore;
            answer:=judge;
```

```
        end if
      end for each judge
end for each trial
return answer;
```

l. Modify the diagram so that the competitors registered for an event can be determined.

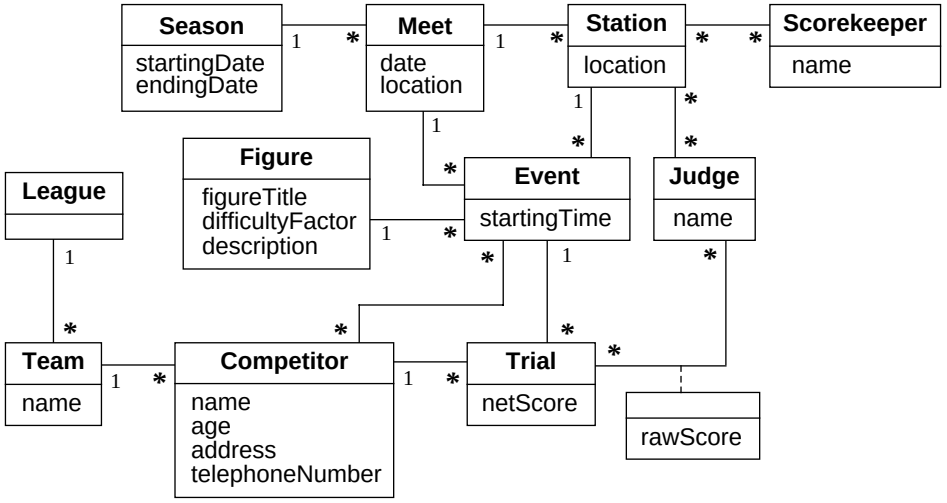

Figure A12.6 Class diagram for a scoring system that supports event registration

12.13 No classes require state diagrams. The swimming league exercises are a data management type of problem. These types of problems have little interesting state behavior.

12.14 The revised diagrams are shown in Figure A12.7-Figure A12.10. Figure A12.7 is a better model than the ternary because *dateTime* is really an attribute. Figure A12.8 is also better than the ternary because the combination of a *Student*, *University*, and *Professor* can occur more than once—a *UniversityClass*. The third ternary is not atomic because the combination of a *Seat* and a *Concert* determine the *Person*. The fourth ternary also is not atomic; this one can be restated as two binary associations.

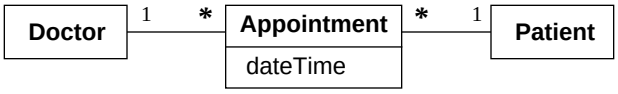

Figure A12.7 Class diagram for appointments

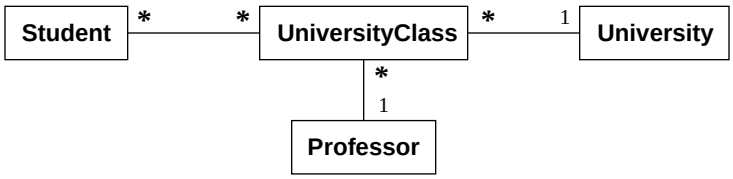

Figure A12.8 Class diagram for university classes

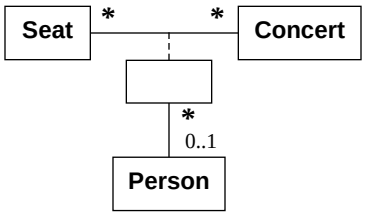

Figure A12.9 Class diagram for reservations

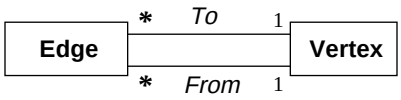

Figure A12.10 Class diagram for directed graphs

12.15 Figure A12.11 models the document manager. Note that by making the model more generic we also make it smaller and more flexible. However, a disadvantage of generic models is that they enforce fewer constraints with their structure than a more tangible application model. For example, the model would allow a book to contain books.

- The *Contains* association eliminates the need for the *Section* and *Page* classes and allows document composition to be arbitrarily deep. The *Contains* association also subsumes the association between *Journal* and *Paper*.
- We have promoted comments to a class. Then the model can support both standard comments that are reusable and custom comments that are specially entered.
- We eliminated the subclasses and capture this information with the enumeration attribute *documentType* that has the following possible values: *section*, *page*, *paper*, *journal*, *book*, *note*, and *file*. The distinction between the different kinds of documents may still be vague, but at least now the imprecision is confined to the *documentType* attribute and does not pervade the structure of the model. We can capture subclass and *Publisher* attributes with the *DocumentProperty* class.

- We can store file path information with a *locationType* of "path name" and the appropriate *locationValue*. We can store journal volume with a *propertyType* of "journal volume" and the appropriate *propertyValue*. Similarly, we can store journal number with a *properType* of "journal number" and the appropriate *propertyValue*. We could have collapsed *Location* and *DocumentProperty* into a single class, but decided to keep them separate in our answer; we would need the tempering of an actual application to make a firm decision.
- We kept *Author* and *DocumentCategory* as distinct classes, because they are especially important to managing documents. In addition, we would need a different generic class than *DocumentProperty* to subsume them, because a document may have many authors and many document categories. Furthermore, the authors are ordered with regard to their significance for creating a document.

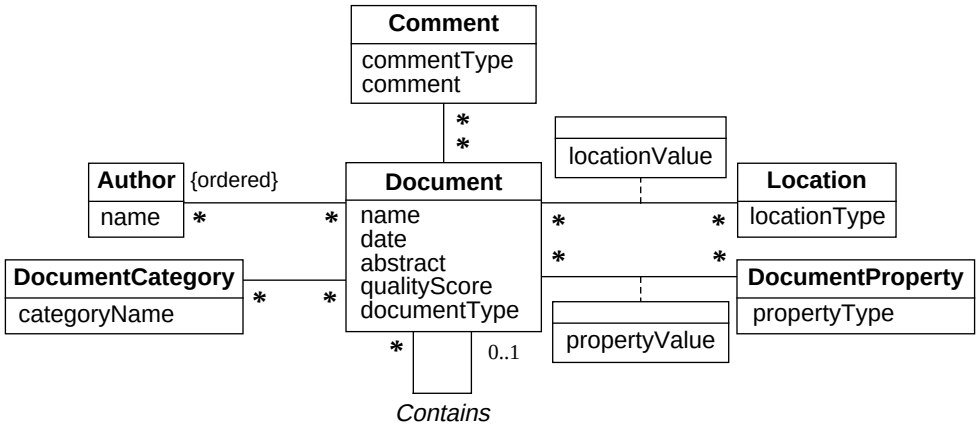

Figure A12.11 Class diagram for a document manager

12.16 The following tentative classes should be eliminated.

- **Redundant classes**. *Vacation* is subsumed by holiday so we discard it. *Meeting entry* is redundant with *meeting*.
- **Irrelevant or vague classes**. We discard *scheduling software, function,* and *network* because they are irrelevant. *Scheduler* refers to the software that is the subject of analysis. *Day* appears in the context of explanation and is not directly relevant to the model. *Repeat information* is vague and refers to attributes of the *Entry* class.
- **Attributes**. We model *start time, end time, start date,* and *end date* as attributes.

We are left with the following classes: *Meeting, Appointment, Task, Holiday, User, Schedule,* and *Entry.*

12.17 We only need to prepare a data dictionary for proper classes.

- *User*—a person who has an account with the scheduler software.
- *Schedule*—a collection of entries for a person. A schedule is an electronic analog to paper-based planning books.
- *Entry*—an individual item in a schedule. There are four kinds of entries: meeting, appointment, task, and holiday. An entry may be defined as being multiply entered with some frequency between repeat start and repeat end dates.
- *Meeting*—an entry in a schedule for several persons to get together for a discussion.
- *Appointment*—an entry in a schedule that lets a person reserve a portion of a day.
- *Task*—a work activity with a duration of one or more days.
- *Holiday*—a non-work day. There are three holiday types: official holiday, personal holiday, and vacation.

12.18 We eliminate spurious associations for the scheduler software.

- **First and second bullets**. We eliminate these because we discarded *scheduling software* and *network* as tentative classes.
- **Third bullet**. We keep this association.
- **Fourth, fifth, and sixth bullets**. The net effect of these bullets is that there is a many-to-many association between *Schedule* and *Entry*.
- **Seventh and eighth bullets**. Both of these concern attributes so they are not bonafide associations.

We are left with the following associations: *user may have a schedule* and a many-to-many association between *Schedule* and *Entry.*

12.19 Figure A12.12 presents a class model for the scheduler software. The generalization was readily apparent from the problem statement.

12.20 The following tentative classes should be eliminated.

- **Redundant classes**. *Invitee* is synonymous with *user* and could be a role, depending on the realization of the class model. *Everyone* refers to *user* and is redundant. *Meeting notice* is the same as *notice*; we keep *notice*. *Attendance* is also extraneous and merely refers to an attendee participating in a meeting. *Meeting* and *meeting entry* are synonymous; we keep *meeting*.
- **Irrelevant or vague classes**. We discard *scheduling software, scheduler, software,* and *meeting information* because they are irrelevant.
- **Attributes**. We model *time* and *acceptance status* as attributes. An *invitation* is a type of notice; we need the enumeration attribute *noticeType* to record whether an invitation is an invitation, reschedule, cancellation, refusal, or confirmation.
- **Roles**. *Chair person* and *attendee* are association ends for user.

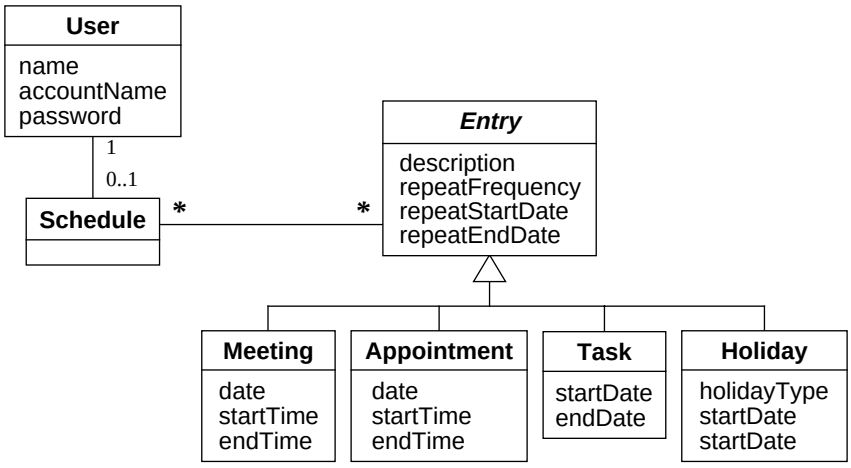

Figure A12.12 Class model for scheduler software

We already noted the classes *User, Schedule,* and *Meeting* (renamed from meeting entry) in our answer to the previous exercise. We add the classes *Room* and *Notice*.

12.21 The following bullets define the additional classes for the scheduler software.

- *Room*—a place in a building where a meeting can be held.
- *Notice*—an email message about a meeting. The values of *noticeType* are: invitation, reschedule, cancellation, refusal, and confirmation.

12.22 We eliminate spurious associations for the extension to the scheduler software.

- **First, seventh, and eighth bullets**. We eliminate these because we discarded *scheduling software* and *scheduler* as tentative classes.
- **Second bullet**. This is a bonafide association. *Chair person* is a role of *user*.
- **Third bullet**. This bullet has no information beyond that for the previous exercise.
- **Fourth bullet**. There is one association in this bullet: *a meeting may have a room*.
- **Fifth bullet**. This only concerns attributes.
- **Sixth bullet**. This bullet implies the association *meetings have users (attendees)*.
- **Ninth bullet**. This bullet implies the association *notices are for users*.

We are left with the following associations: *user (chair person) arranges meetings, a meeting may have a room, meetings have users (attendees)*, and *notices are for users*.

12.23 Figure A12.13 presents a class model for the extension to the scheduler software.

- We made several revisions not directly implied by the problem statement.

- We refined the association *notices are for users* by observing that a notice is sent by one user and received by another user.
- We added the association *a meeting has many notices*.

Figure A12.13 fixes a problem with our answer to Exercise 12.19. In Figure A12.12 we cannot tell which user owns an entry. In our revised model each entry belongs to a single schedule, the owner of the entry. For the *Meeting* subclass, we add another association for the attendees of a meeting.

There is another subtlety to our answer. We decided to associate *Notice* to *Schedule*, rather than to *User*. To be precise, the scheduling software operates on schedules and not on users. Given a schedule, we can readily find the user, since *User* and *Schedule* have a one-to-one association.

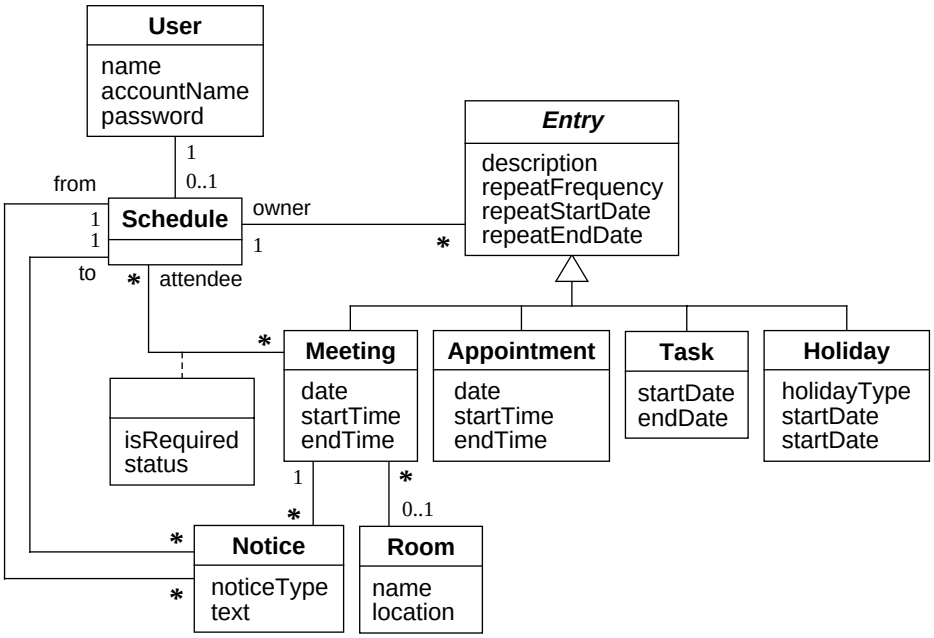

Figure A12.13 Class model for an extension to the scheduling software