3

Class Modeling

3.1 Figure A3.1 shows a class diagram for international borders.

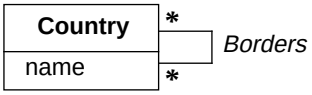

Figure A3.1 Class diagram for international borders

3.2 Figure A3.2 shows a class diagram for polygons and points. The smallest number of points required to construct a polygon is three.

The multiplicity of the association depends on how points are identified. If a point is identified by its location, then points are shared and the association is many-to-many. On the other hand, if each point belongs to exactly one polygon then several points may have the same coordinates. The next answer clarifies this distinction.

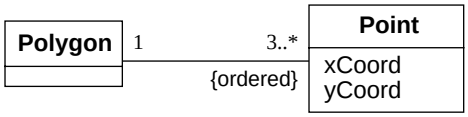

Figure A3.2 Class diagram for polygon and points

3.3 a. Figure A3.3 shows objects and links for two triangles with a common side in which a point belongs to exactly one polygon.

b. Figure A3.4 shows objects and links for two triangles with a common side in which points may be shared.

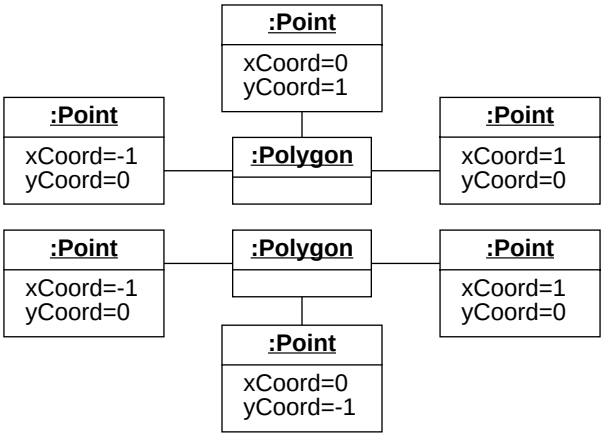

Figure A3.3 Object diagram where each point belongs to exactly one polygon

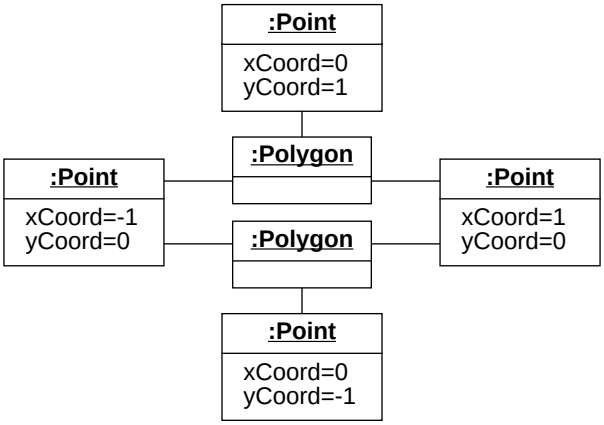

Figure A3.4 Object diagram where each point can belong to multiple polygons

3.4 Figure A3.5 shows the class diagram—this is a poor model. Figure A3.5 has some flaws that Figure A3.2 does not have. Figure A3.5 permits a degenerate polygon which consists of exactly one point. (The same point is first and last. The point is next to itself.) The class diagram also permits a line to be stored as a polygon. Figure A3.5 does not enforce the constraint that the first and last points must be adjacent.

Figure A3.2 and Figure A3.5 share other problems. The sense of ordering is problematic. A polygon that is traversed in left-to-right order is stored differently than one that is traversed in right-to-left order even though both visually appear the same. There is no constraint that a polygon be closed and that a polygon not cross itself. In general it is

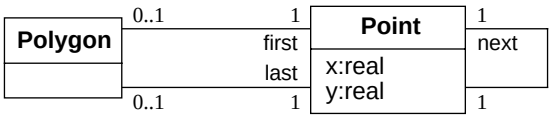

Figure A3.5 Another class diagram for polygon and points

difficult to fully capture constraints with class models and you must choose between model complexity and model completeness.

3.5 **Description for Figure A3.2**. A polygon consists of at least three points; each point has an x coordinate and a y coordinate. Each point belongs to exactly one polygon but whether or not this constraint is required was not made clear by the exercise statement. The points in a polygon are stored in an unspecified order.

Description for Figure A3.5. A polygon has a first and a last point. Each point has an x coordinate and a y coordinate. A point may be first, last, or in the middle of a sequence for a polygon. Each point belongs to at most one polygon. Each point is linked to its next point.

3.6 Figure A3.6 shows a class diagram for a family tree consistent with the exercise. Other models are also possible such as those showing divorce and remarriage. The cousin and sibling associations are logically redundant and can be derived. Chapter 4 discusses derived information.

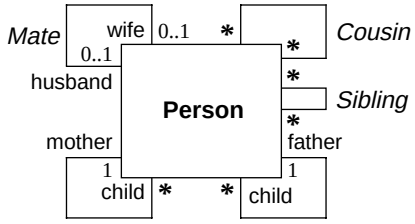

Figure A3.6 Class diagram for family trees

We used our semantic understanding of the exercise to determine multiplicity in our answer. In general, you can only partially infer multiplicity from examples. Examples can establish the need for many multiplicity but does not permit you to conclude that exactly 1 or 0..1 multiplicity applies.

3.7 Figure A3.7 is just one of several possible answers. You could use fewer generalizations and still have a correct answer. The advantage of a thorough taxonomy as we have shown is that it is easier to extend the model to other primitive shapes such as parallelograms, polygons, and 3-dimensional figures.

The class model contains width and height for both square and circle, even though they must be equal. Application software would have to enforce this equality constraint. We also show *Circle* inheriting *orientation* even though the symmetry of a circle makes orientation irrelevant.

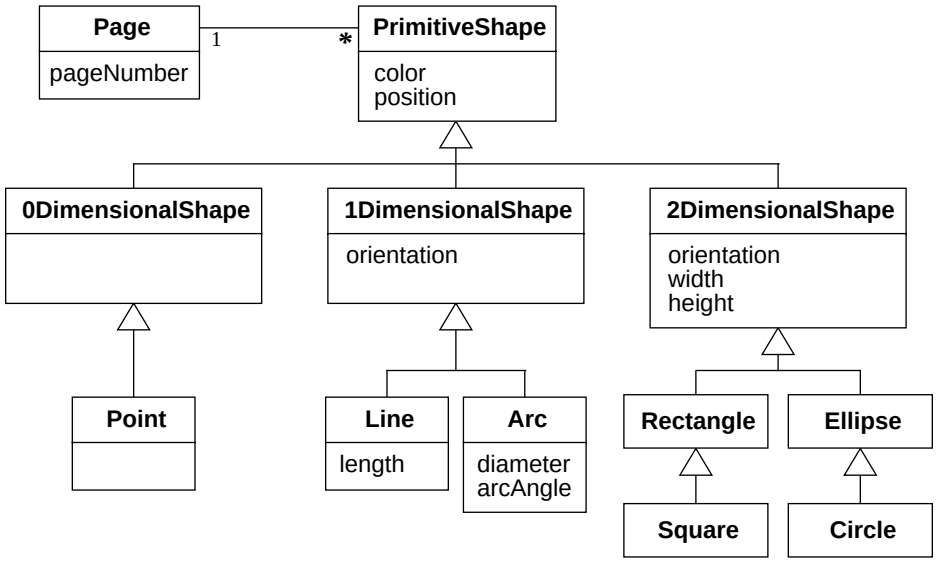

Figure A3.7 Class diagram for geometrical documents

3.8 Figure A3.8 adds multiplicity to the air transportation model. Some associations in Figure E3.6 are unlabeled and require interpretation in order to assign multiplicity. For example *airport locatedIn city* is one-to-many and *airport serves city* is many-to-many. It is important to properly document models to avoid this kind of uncertainty. A possible improvement to the model would be to partition *Flight* into two classes: *ScheduledFlight* and *FlightOccurrence*.

3.9 Figure A3.9 adds operations, association names, and association end names to the air transportation system class model. We chose to add the *reserve* operation to the *Seat* class since *Seat* is the most direct target of the operation. This is not an obvious assignment since *reserve* operates on *Seat*, *Flight*, and *Passenger* objects. Chapter 15 presents guidelines for determining which class should own an operation.

3.10 See answer to Exercise 3.9.

3.11 See answer to Exercise 3.9.

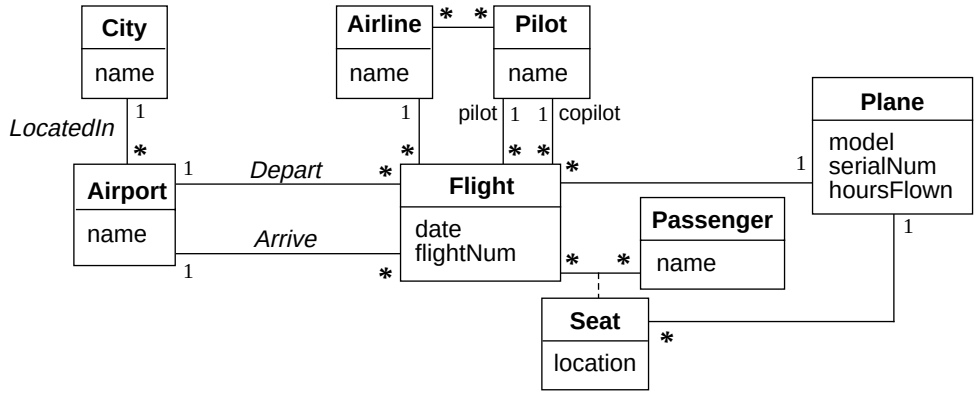

Figure A3.8 Class diagram for an air transportation system

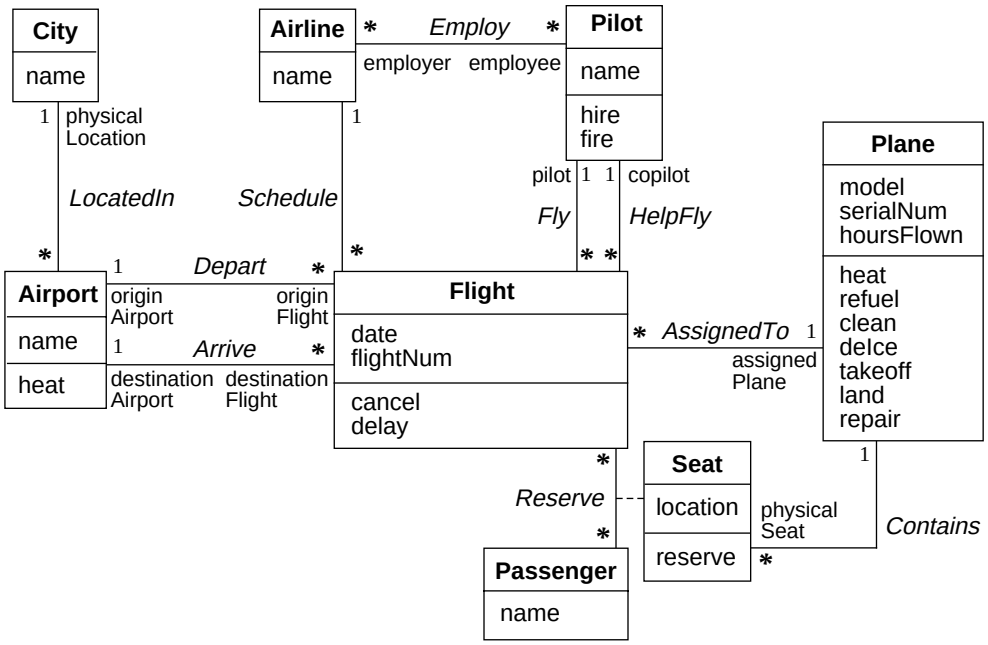

Figure A3.9 Class diagram for an air transportation system with operations, association names, and association end names

3.12 Figure A3.10 shows an object diagram that corresponds to the exercise statement. Note that most attribute values are left unspecified by the problem statement, yet the object

diagram is still unambiguous. All objects are clearly identified by their attribute values and links to other objects. The exercise states that "you took a round trip between cities last weekend"; we make the assumption that this is also a round trip between airports. The two seats connected by the dotted line may be the same object.

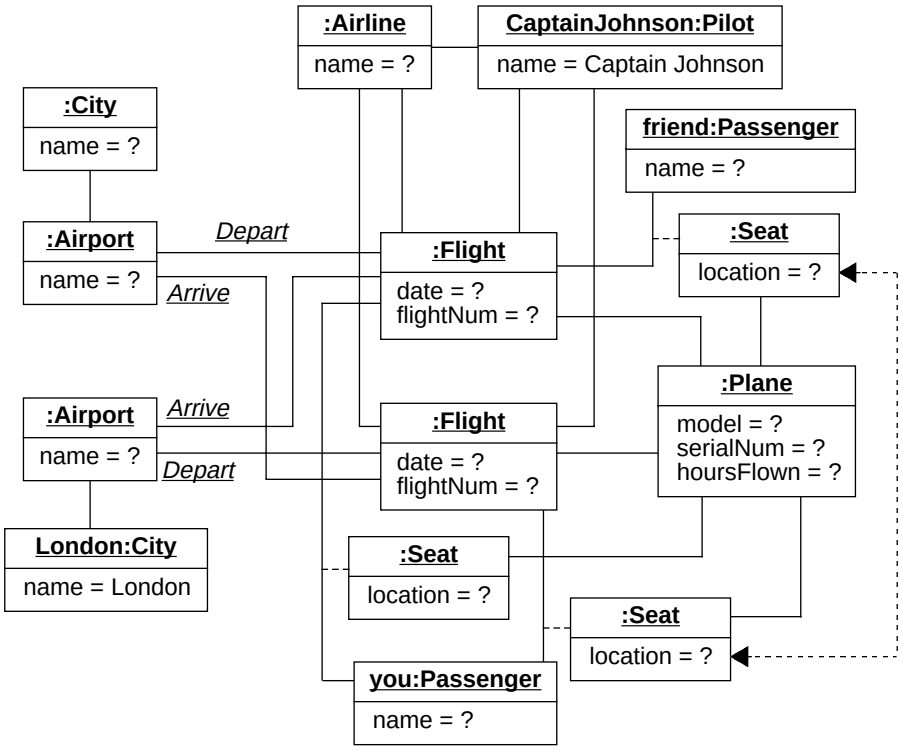

Figure A3.10 Object diagram for an air transportation system

3.13 [Do not be overly critical of the class models for Exercise 3.13 as most of them are for toy problems. Remember that the precise content of a model is driven by its relevance to the problem to be solved. For Exercise 3.13, we have not clearly stated the problem. For example, are we trying to design a database, program an application, or just understand a system.]

a. Figure A3.11 shows one possible class diagram for a school.

A school has a principal, many students, and many teachers. Each of these persons has a name, birthdate, and may borrow and return books. Teachers and the principal are both paid a salary; the principal evaluates the teachers. A school board supervises multiple schools and can hire and fire the principal for each school.

A school has many playgrounds and rooms. A playground has many swings. Each room has many chairs and doors. Rooms include restrooms, classrooms, and the cafeteria. Each classroom has many computers and desks. Each desk has many rulers.

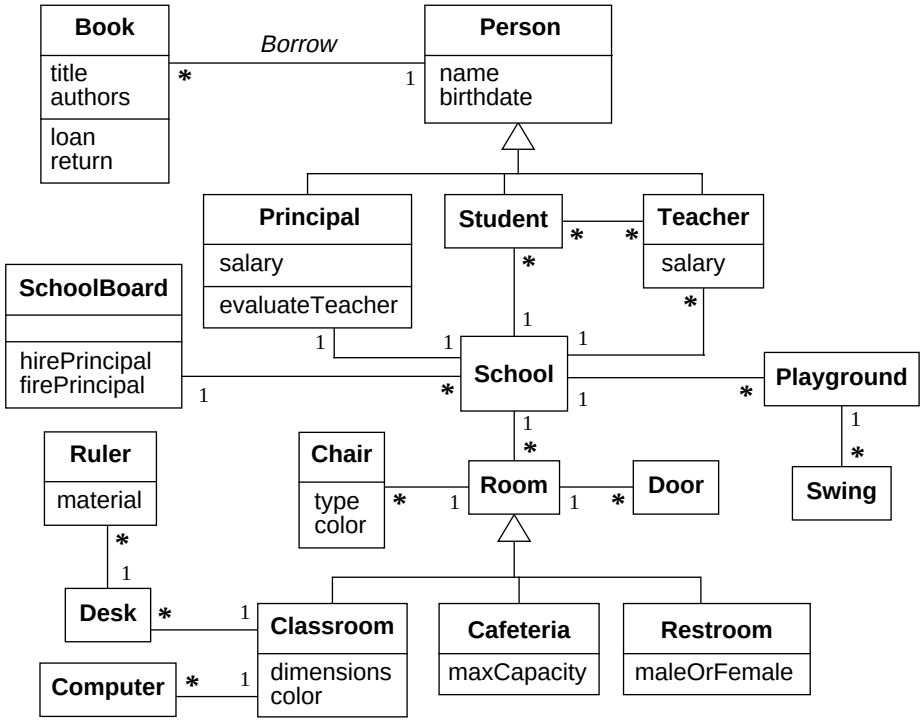

Figure A3.11 Class diagram for a school

b. Figure A3.12 shows a class diagram for an automobile.

An automobile is composed of a variety of parts. An automobile has one engine, one exhaust system, many wheels, many brakes, many brake lights, many doors, and one battery. An automobile may have 3, 4, or 5 wheels depending on whether the frame has 3 or 4 wheels and the optional spare tire. Similarly a car may have 2 or 4 doors (2..4 in the model, since UML2 multiplicity must be an interval). The exhaust system can be further divided into smaller components such as a muffler and tailpipe. A brake is associated with a brake light that indicates when the brake is being applied.

Note that we made manufacturer an attribute of automobile, rather than a class that is associated with automobile. Either approach is correct, depending on your perspective. If all you need to do is to merely record the manufacturer for an automobile, the manufacturer attribute is adequate and simplest. If on the other hand, you want to record details about the manufacturer, such as address, phone number, and dealership information, then you should treat manufacturer as a class and associate it with automobile.

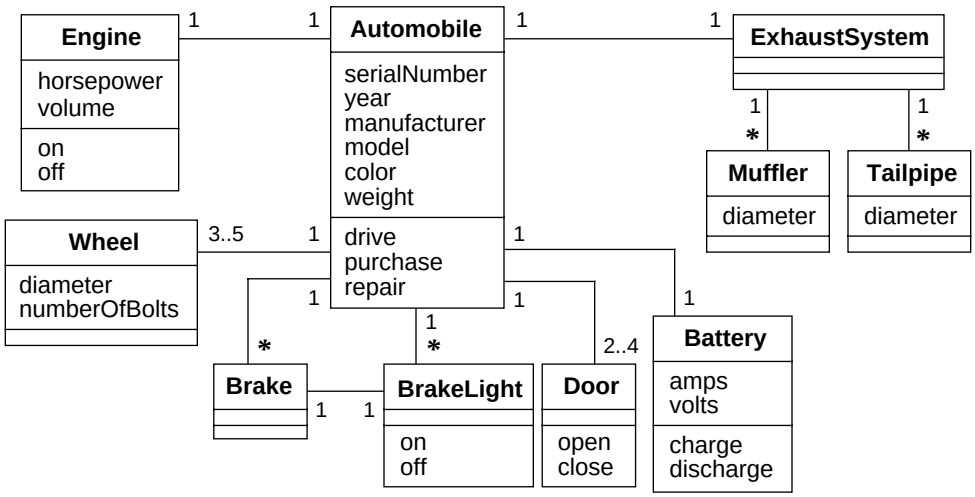

Figure A3.12 Class diagram for an automobile

c. Figure A3.13 shows a class diagram for a castle.

A castle has many rooms, corridors, stairs, towers, dungeons, and floors. Each tower and dungeon also has a floor. The castle is built from multiple stones each of which has dimensions, weight, color, and a composition material. The castle may be surrounded by a moat. Each lord lives in a castle; a castle may be without a lord if he has been captured in battle or killed. Each lady lives in a castle with her lord. A castle may be haunted by multiple ghosts, some of which have hostile intentions.

d. Figure A3.14 and Figure A3.15 show a class diagram for a program. Note that we have added quite a few classes.

In Figure A3.14 a program has descriptive properties such as its name, purpose, date last modified, and author. Important operations on programs include: compile, link, execute, and debug. A program contains global data definitions and many functions. Each function has data definitions and a main block. Each block consists of many statements. Some types of statements are assignment, conditional, iteration, and procedure call. An assignment statement sets a target variable to the result of an expression. A conditional statement evaluates an expression; a then block is executed if the expression is true and an optional else block is executed if the expression is false. An iteration statement continues to execute a block until a loop expression becomes false. A procedure call invokes a function and may pass the result of zero or more expressions in its argument list.

Figure A3.15 details the structure of expressions and parallels the structure of the programming code that could be used to implement expressions. An expression may be enclosed by parentheses. If so, the parentheses are removed and the remaining expression recursively defined. Otherwise an expression contains a relational operator such as ‘>’, ‘=’, or ‘<=’ or is defined as a term. A term is binary addition, binary subtraction, or a factor. A factor involves multiplication, division, or unary expressions. Unary expres-

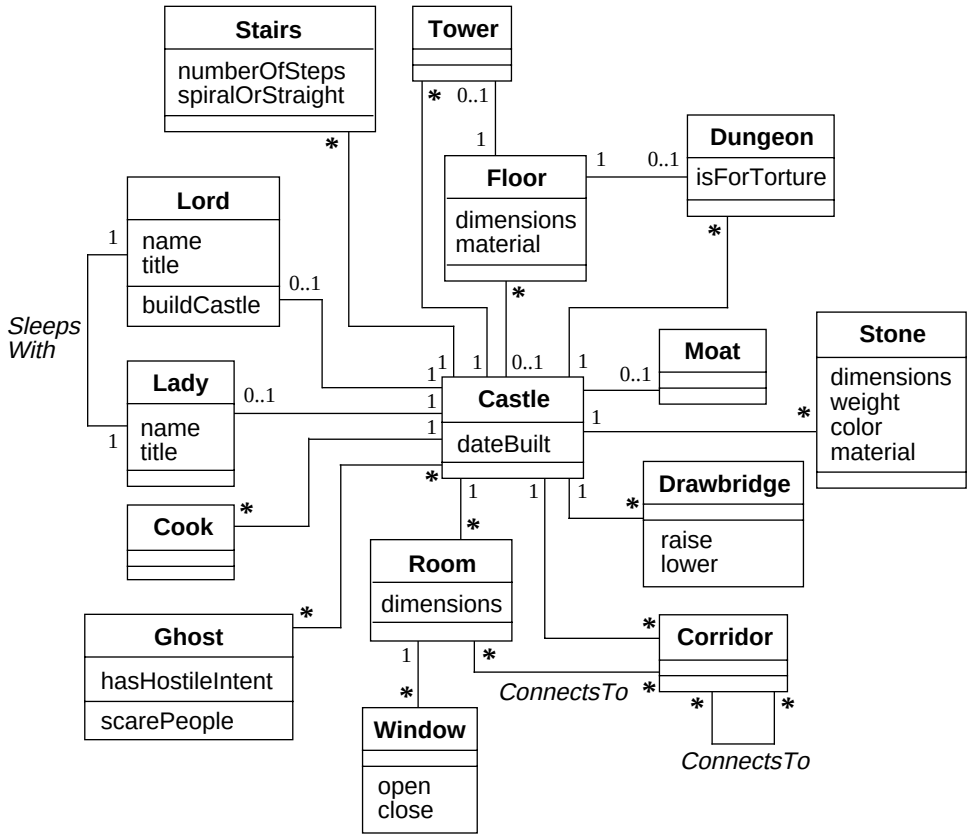

Figure A3.13 Class diagram for a castle

sions can be refined into unary positive or negative expressions or terminals. A terminal may be a constant, variable, or a function reference. The "many" multiplicity that appear on *ParenthesizedExpression* and the other subclasses indicate that expressions may be reused within multiple contexts. (See Exercise 3.3.)

e. Figure A3.16 shows a class diagram for a file system.

A drive has multiple discs; a hard drive contains many discs and a floppy drive contains one disc. (*Platter* may be a better name instead of *Disc*.) A disc is divided into tracks which are in turn subdivided into sectors. A file system may use multiple discs and a disc may be partitioned across file systems. Similarly a disc may contain many files and a file may be partitioned across many discs.

A file system consists of many files. Each file has an owner, permissions for reading and writing, date last modified, size, and checksum. Operations that apply to files include create, copy, delete, rename, compress, uncompress, and compare. Files may be data files or directory files. A directory hierarchically organizes groups of presumably

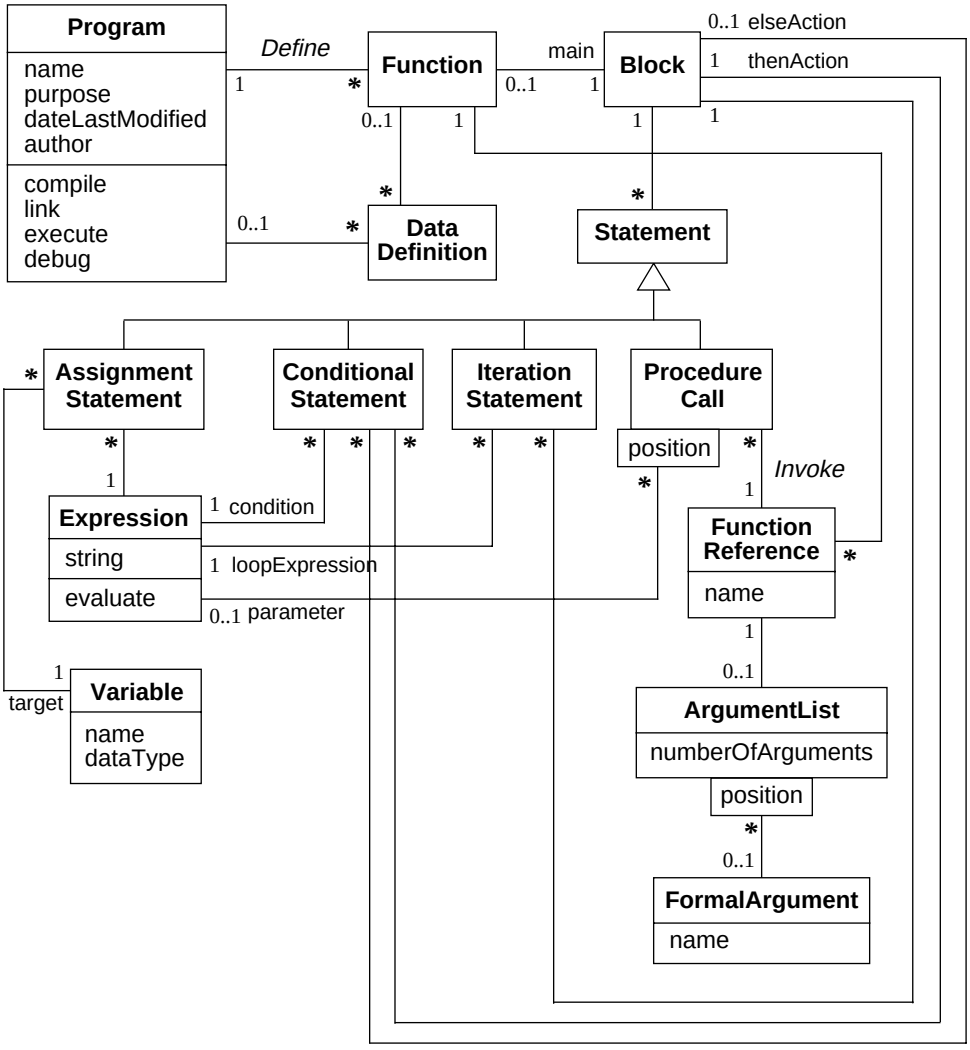

Figure A3.14 Class diagram for a computer program—part 1

related files; directories may be recursively nested to an arbitrary depth. Each file within a directory can be uniquely identified by its file name. A file may correspond to many directory–file name pairs such as through UNIX links. A data file may be an ASCII file or binary file.

f. Figure A3.17 shows a class diagram for a gas fired, hot air heating system. A gas heating system is composed of furnace, humidification, and ventilation subsystems.

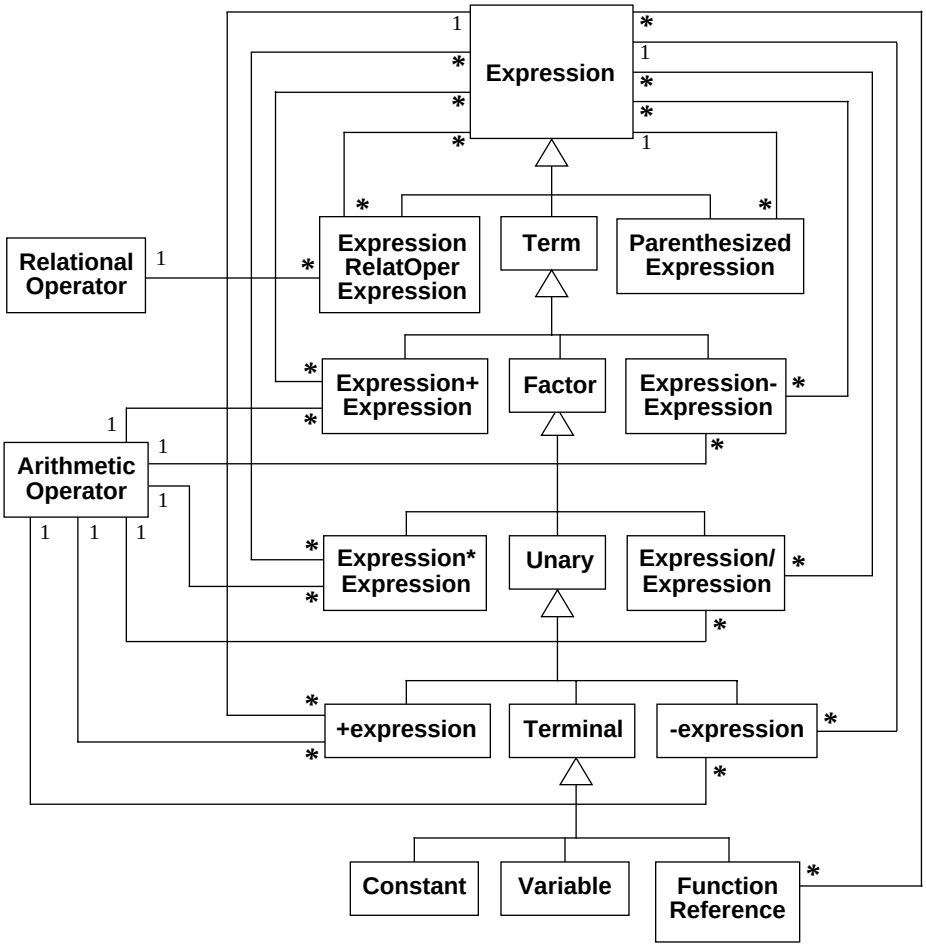

Figure A3.15 Class diagram for a computer program—part 2

The furnace subsystem can be further decomposed into a gas furnace, gas control, furnace thermostat, and many room thermostats. The room thermostats can be individually identified via the room number qualifier.

The humidification subsystem includes a humidifier and humidity sensor.

The ventilation subsystem has a blower, blower control, and many hot air vents. The blower in turn has a blower motor subcomponent.

g. Figure A3.18 shows a class diagram for a chess game. We assume that the purpose of this class diagram is to serve as a basis for a computerized chess game.

A chess game involves many chess pieces of various types such as rooks, pawns, a king, and a queen. A chess game is also associated with a board and a sequence of

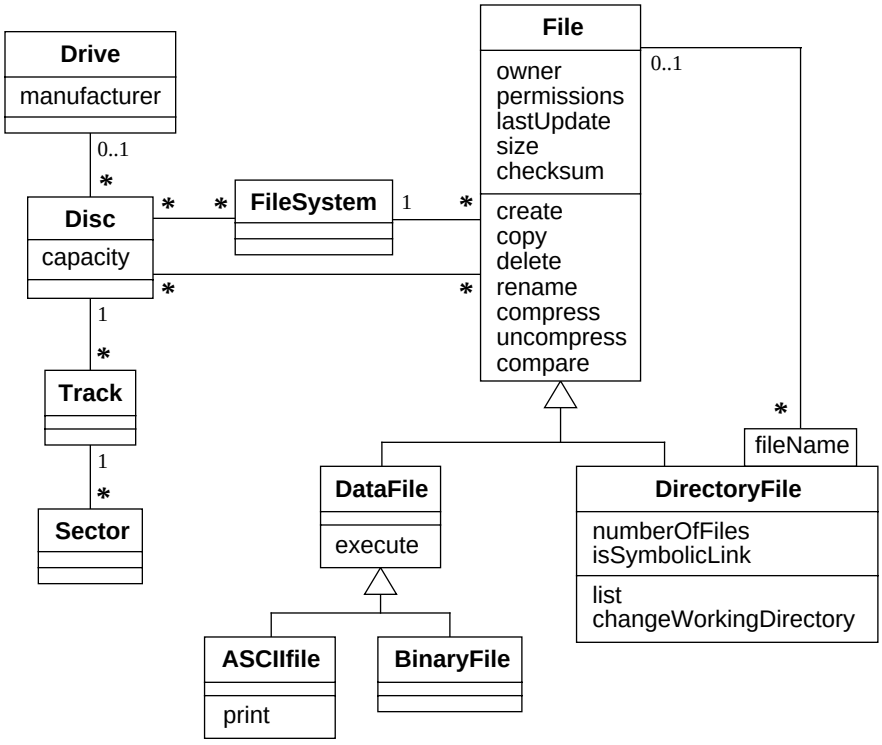

Figure A3.16 Class diagram for a file system

moves. Each time the computer contemplates a move, it computes a tree of possible moves. There are various algorithms which can be used to evaluate potential moves, and restrict the growth of the search tree. The human player can change the difficulty of the computer opponent by adjusting the depth of the strategy lookahead.

Each chess piece is positioned on a square or off the board if captured; some squares on the board are unoccupied. A move takes a chess piece and changes the position from an old position to a new position. A move may result in capture of another piece. The square for a move is optional since the chess piece may start or end off the board. Each square corresponds to a rank and file; the rank is the y-coordinate and the file is the x-coordinate.

h. Figure A3.19 shows a class diagram for a building. This diagram is simple and self-explanatory.

3.14 See answers for Exercise 3.13.

3.15 Figure A3.20 shows a class diagram for a card playing system with operations added.

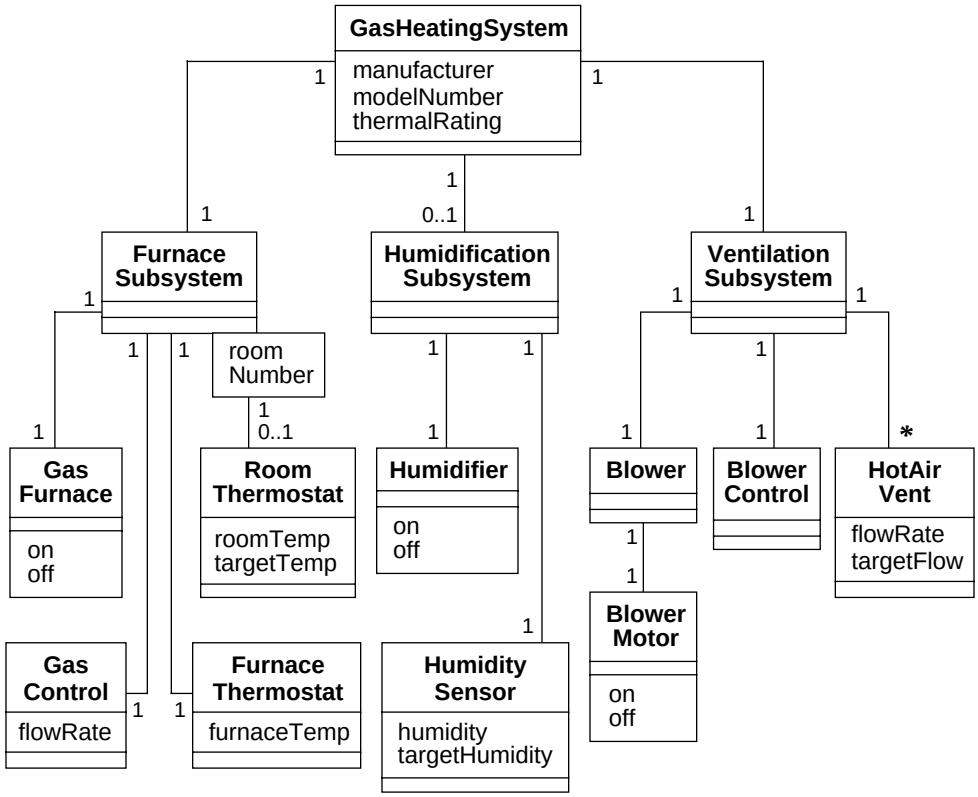

Figure A3.17 Class diagram for a gas-fired, hot-air heating system

Initialize deletes all cards from a *Hand*, *DiscardPile*, or *DrawPile*. *Initialize* refreshes a deck to contain all cards.

Insert(Card) inserts a *Card* into a *CollectionOfCards.* We assume that insertion is done at the top of the pile, but other strategies are possible.

BottomOfPile and *topOfPile* are functions which return the last or first *Card*, respectively, in a *CollectionOfCards.*

Shuffle randomly shuffles a *Deck.*

Deal(hands) deals cards into *hands*, a set of *Hand*s, removing them from the *Deck* and inserting them into each *Hand. InitialSize* determines how many cards are dealt into each hand. Since all hands are the same initial size, a better approach might be to convert *initialSize* into a static attribute (discussed in Chapter 4) or to pass it to *deal* as an argument.

Sort reorders the cards in a *Hand* into a standard order, such as that used in bridge.

Draw is a function which deletes and returns the top card of a *DiscardPile* or a *DrawPile.*

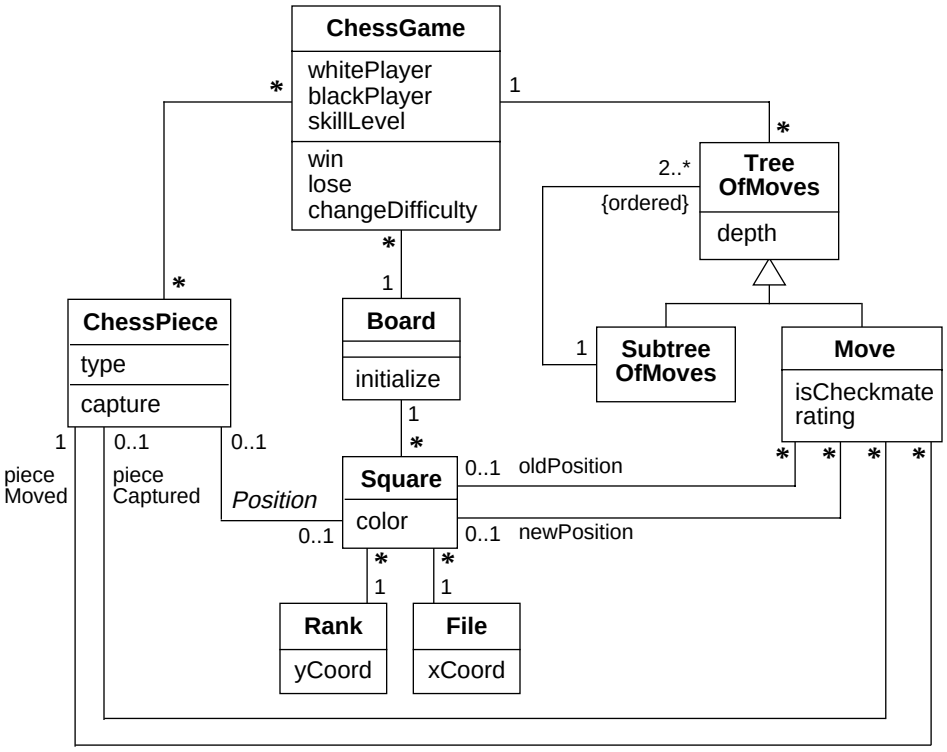

Figure A3.18 Class diagram for a chess game

Display(*location, visibility)* displays a card at the given location. The *visibility* argument determines whether the front or the back of the card is shown.

Discard deletes a *Card* from its *CollectionOfCards* and places it on top of the *DrawPile.*

3.16 Figure A3.21 permits a column to appear on multiple pages with all copies of a column having the same width and length. If it is desirable for copies of a column to vary in their width and length, then width and length should also be made attributes of the association along with x location and y location.

3.17 Figure A3.22 adds multiplicity and attributes to the class diagram for an athletic event scoring system. Age is a derived attribute that is computed from birthdate and the current date. (See Chapter 4.) We have added an association between *Competitor* and *Event* to make it possible to determine intended events before trials are held.

A class model cannot readily enforce the constraint that a judge rates *every* competitor for an event. Furthermore, we would probe this statement if we were actually building the scoring system. For example, what happens if a judge has scored some compet-

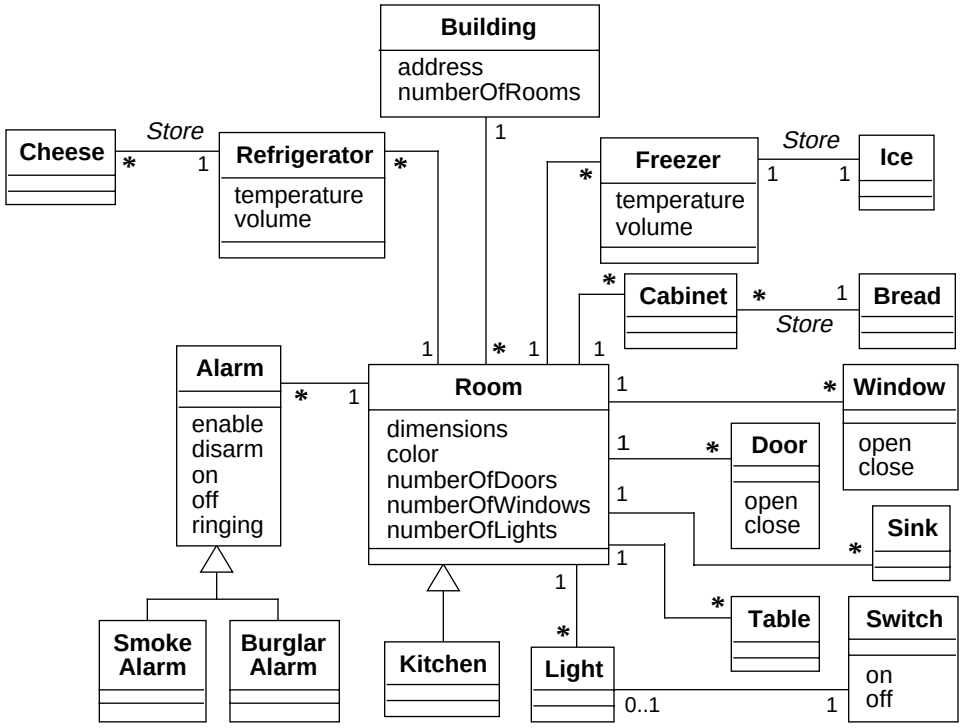

Figure A3.19 Class diagram for a building

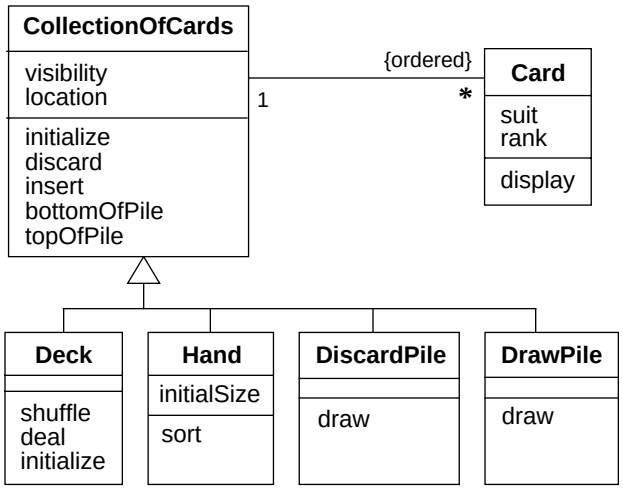

Figure A3.20 Portion of a class diagram for a card playing system

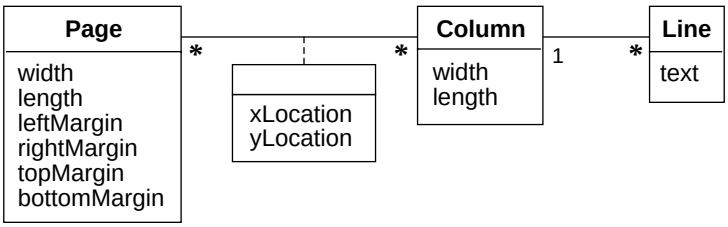

Figure A3.21 Portion of a class diagram for a newspaper publishing system

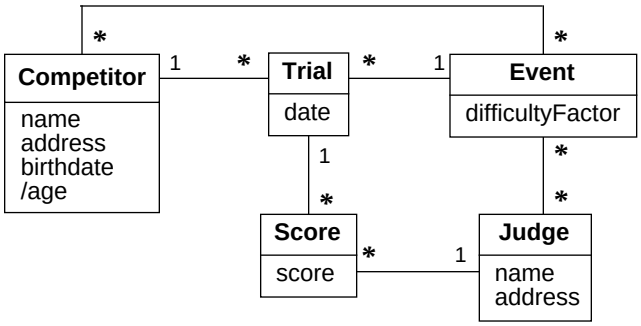

Figure A3.22 Portion of a class diagram for an athletic event scoring system

itors for an event and gets called away? Are the scores of the judge discarded? Is the judge replaced? Or are partial scores for an event recorded? When you build applications you must regard requirements as a good faith effort to convey what is required, and not necessarily as the literal truth.

3.18 See answer for Exercise 3.17. We included birthdate and age for competitors because it affects their grouping for competition. In contrast, birthdate and age are irrelevant for judges.

3.19 See answer for Exercise 3.17.

3.20 This is an important exercise, because graphs occur in many applications. Several variations of the model are possible, depending on your viewpoint. Figure A3.23 accurately represents undirected graphs as described in the exercise. Although not quite as accurate, your answer could omit the class *UndirectedGraph*.

We have found it useful for some graph related queries to elevate the association between vertices and edges to the status of a class as Figure A3.24 shows.

3.21 Figure A3.25 is an object diagram for the class diagram in Figure A3.23.

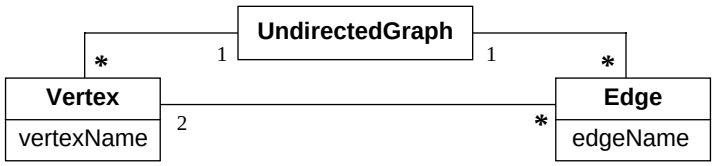

Figure A3.23 Class diagram for undirected graphs

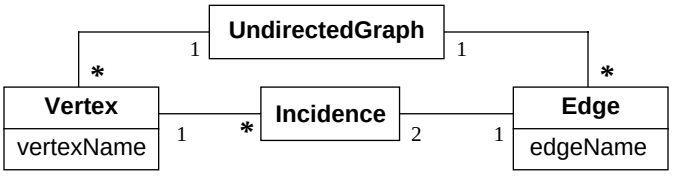

Figure A3.24 Class diagram for undirected graphs in which the incidence between vertices and edges is treated as a class

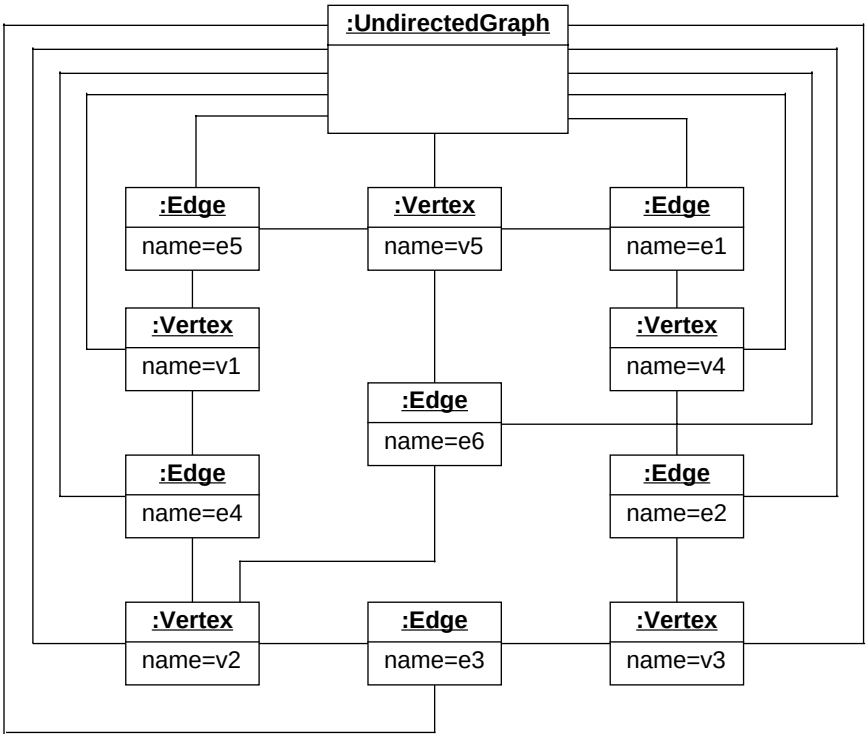

Figure A3.25 Object diagram for the sample undirected graph

3.22 Figure A3.26 adds geometry details to the object model for an undirected graph. It would also be a correct answer to add the geometry attributes to the *Vertex* and *Edge* classes. However, for complex models, it is best not to combine logical and geometrical aspects.

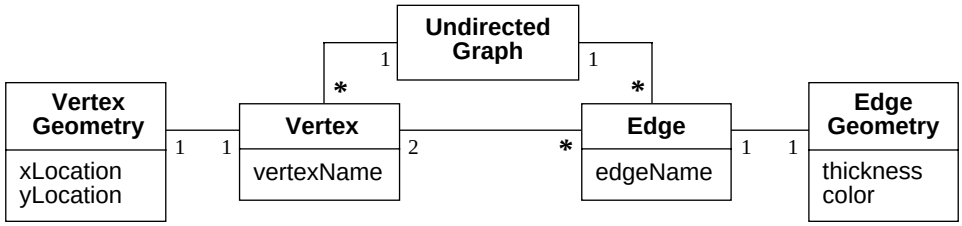

Figure A3.26 Class diagram for undirected graphs with geometrical details

3.23 Figure A3.27 shows a class diagram describing directed graphs.The distinction between the two ends of an edge is accomplished with a qualified association. Values of the qualifier *end* are *from* and *to.*

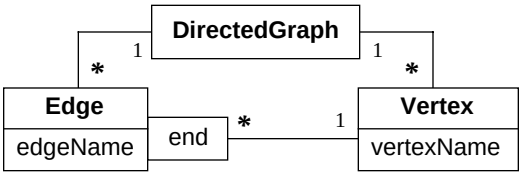

Figure A3.27 Class diagram for directed graphs using a qualified association

Figure A3.28 shows another representation of directed graphs. The distinction between the two ends of an edge is accomplished with separate associations.

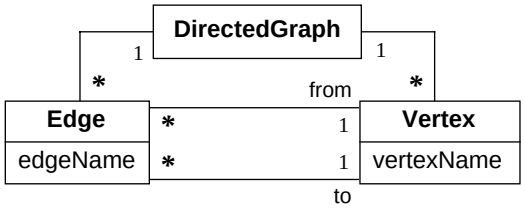

Figure A3.28 Class diagram for directed graphs using two associations

The advantage of the qualified association is that only one association must be queried to find one or both vertices that a given edge is connected to. If the qualifier is not

specified, both vertices can be found. By specifying *from* or *to* for the *end* qualifier, you can find the vertex connected to an edge at the given *end*.

The advantage of using two separate associations is that you eliminate the need to manage enumerated values for the qualifier *end*.

3.24 Figure A3.29 is an object diagram corresponding to the class diagram in Figure A3.28.

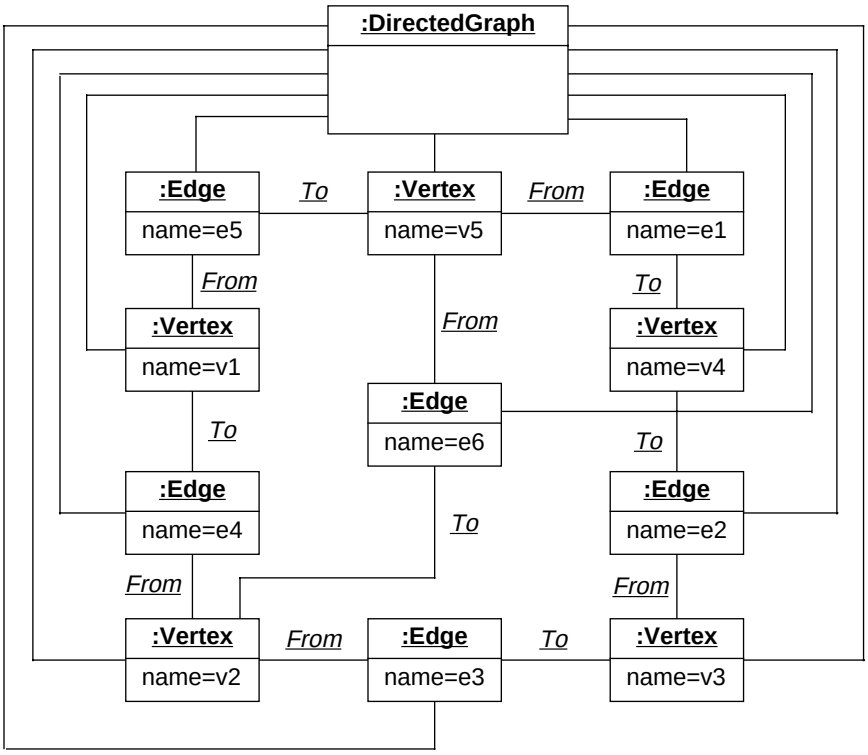

Figure A3.29 Object diagram for the sample directed graph

3.25 Figure A3.30 shows a class diagram for car loans in which pointers are replaced with associations.

In this form, the arguably artificial restriction that a person have no more than three employers has been eliminated. Note that in this model an owner can own several cars. A car can have several loans against it. Banks loan money to persons, companies, and other banks.

3.26 Method *a* will not work. The owner does not uniquely identify a motor vehicle, because someone may own several cars. Additional information is needed to resolve multiple ownership. In general, it is best to identify an object by some intrinsic property and not

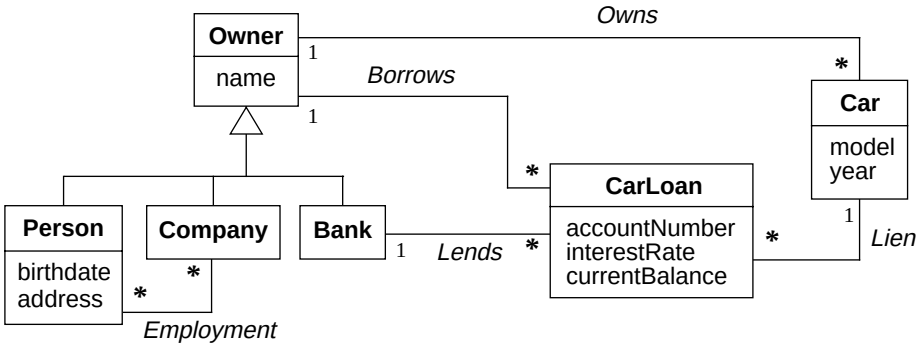

Figure A3.30 Proper class diagram for car loans

by a link to some other thing. For example, what happens when an owner sells a car, a person changes his or her name, or a company that owns a car is acquired by another company?

Method *b* also will not work. The manufacturer, model, and year does not uniquely identify a car, because a given manufacturer makes many copies of a car with the same model and year. The problem here is confusion between descriptor and occurrence. (See Chapter 4.)

Method *c* seems to be the best solution. The vehicle identification number (VIN) uniquely identifies each car and is a real-world attribute. Anyone can inspect a car and read the VIN stamped on it.

Method *d* will uniquely identify each car, but is unworkable if agencies need to exchange information. The ID assigned by the department of motor vehicles will not agree with that assigned by the insurance company which will differ from that assigned by the bank and the police.

3.27 Figure A3.31, Figure A3.32, and Figure A3.33 contain a class model of a 4-cycle lawn mower engine that may be helpful for troubleshooting. A state diagram (discussed in Chapters 5 and 6) would also be helpful in capturing the dynamic behavior of the engine. It is debatable whether "air" and "exhaust" are truly objects. (See definition of *object* in Section 3.1.1). Whether they are objects depends on the purpose of the model which is unclear. [Students may find other correct answers for this exercise.]

3.28 Figure A3.34 shows a class diagram for the dining philosopher's problem. The one-to-one associations describe the relative locations of philosophers and forks. The *InUse* association describes who is using forks. Other representations are possible, depending on your viewpoint. An object diagram may help you better understand this problem.

3.29a. Figure A3.35 shows the class diagram for the tower of Hanoi problem in which a tower consists of 3 pegs with several disks in a certain order.

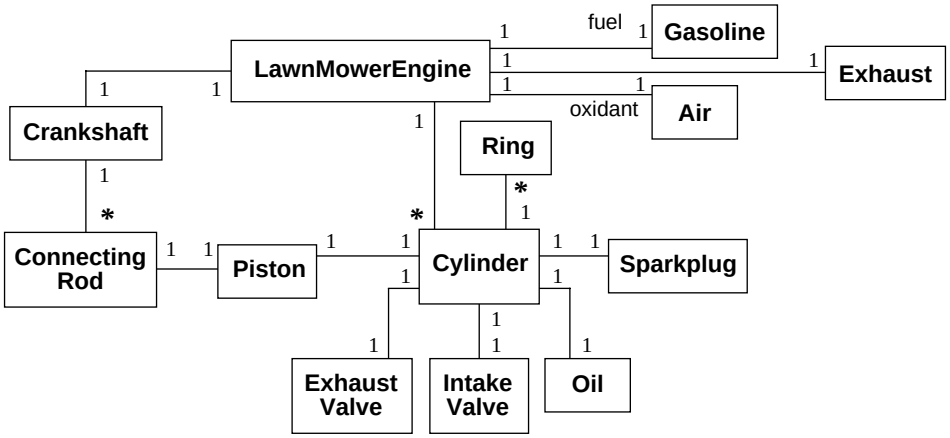

Figure A3.31 Class diagram for 4-cycle lawn mower engine—part 1

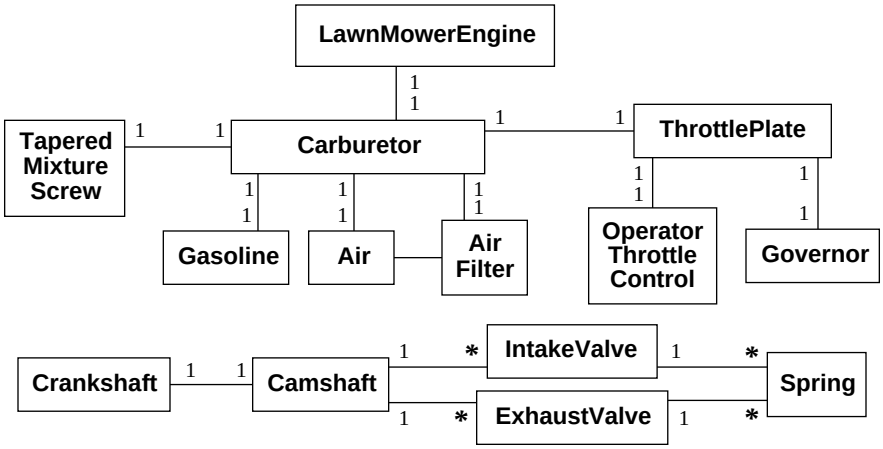

Figure A3.32 Class diagram for 4-cycle lawn mower engine—part 2

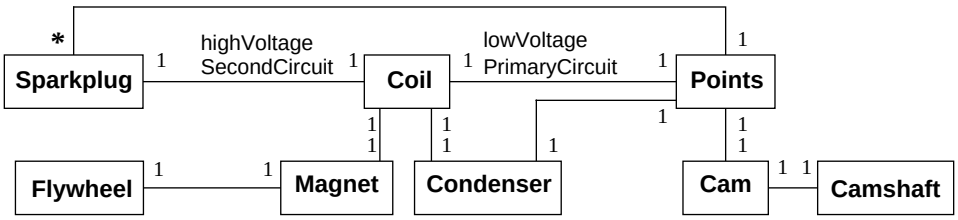

Figure A3.33 Class diagram for 4-cycle lawn mower engine—part 3

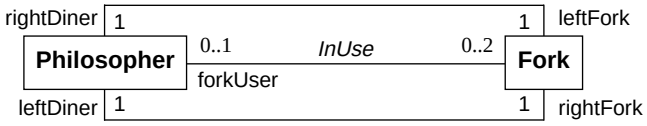

Figure A3.34 Class diagram for the dining philosopher problem

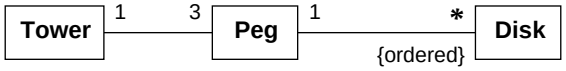

Figure A3.35 Simple class diagram for tower of Hanoi problem

b. Figure A3.36 shows the class diagram for the tower of Hanoi problem in which disks are organized into stacks.

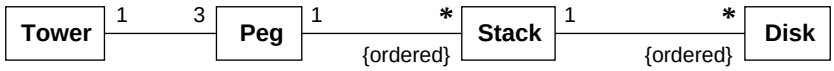

Figure A3.36 Tower of Hanoi class diagram with disks organized into stacks

c. Figure A3.37 shows the class diagram for the tower of Hanoi problem in which pegs are organized into recursive subsets of disks. The recursive structure of a stack is represented by the self association involving the class *Stack*. Note that the multiplicity of the class *Disk* in the association between *Stack* and *Disk* is exactly one.

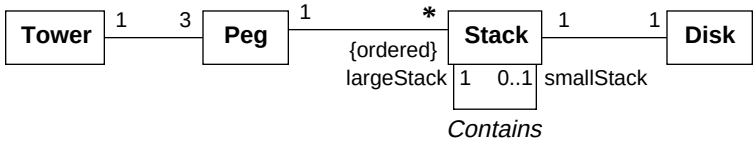

Figure A3.37 Tower of Hanoi class diagram in which the structure of stacks is recursive

d. Figure A3.38 shows the class diagram for the tower of Hanoi problem in which stacks are in a linked list. The linked list is represented by the association *Next*.

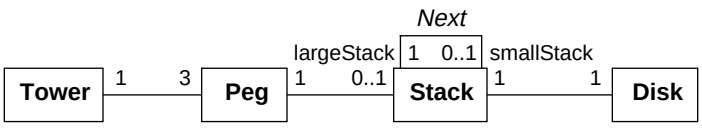

Figure A3.38 Tower of Hanoi class diagram in which stacks are organized into a linked list

3.30 In principle, none of the four class diagrams presented in Figure A3.35 through Figure A3.38 are intrinsically better than the others. All are reasonable models. We will evaluate the models purely on the basis of how well each model supports the algorithm stated in the exercise.

Figure A3.35 is not well suited for the recursive stack movement algorithm, because it does not support the notion of a stack.

Figure A3.36 is better than the previous figure, since it does include the concept of a stack. However Figure A3.36 does not support recursion of a stack within a stack.

Figure A3.37 would be easiest to program if the multiplicity between *Peg* and *Stack* was changed to one-to-one (one stack per peg, the stack may recursively contain other stacks). Figure A3.39 makes this change and adds attributes and operations to the class model. The data structures in Figure A3.39 precisely mirror the needs of the algorithm. The *move(pegNumber)* operation moves a stack to the specified peg. *AddToStack (pegNumber*) adds a disk to the bottom of the stack associated with *pegNumber*. *RemoveFromStack (pegNumber)* removes the disk at the bottom of the stack associated with *pegNumber*.

Figure A3.38 also is a good fit. If a recursive call to the move-stack operation includes as an argument the location in the link list, then descending via recursion is equivalent to moving right through the linked list.

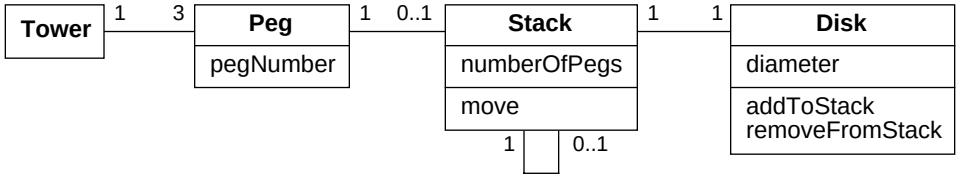

Figure A3.39 Detailed tower of Hanoi class diagram in which the structure of stacks is recursive

3.31 The following OCL expression computes the set of airlines that a person flew in a given year.

```
aPassenger.Flight->SELECT(getYear(date)=aGivenYear).
Airline.name->asSet
```

The OCL *asSet* operator eliminates redundant copies of the same airline.

3.32 The following OCL expression computes the nonstop flights from *aCity1* to *aCity2*.

```
aCity1.Airport.originFlight
->SELECT(destinationAirport.City=aCity2)
```

For this answer we did not apply the OCL *asSet* operator. Given the semantics of the model, there are no redundant flights to eliminate. However, it would also be a correct answer if the *asSet* operator is applied at the end of the OCL expression.

3.33 The following OCL expression computes the total score for a competitor from a judge.

```
aCompetitor.Trial.Score->SELECT(Judge=aJudge).score
->SUM
```

Once again, given the semantics of the problem, there is no need to use the *asSet* operator.

3.34 Figure E3.13 (a) states that a subscription has derived identity. Figure E3.13 (b) gives subscriptions more prominence and promotes subscription to a class.

The (b) model is a better model. Most copies of magazines have subscription codes on their mailing labels; this could be stored as an attribute. The subscription code is intended to identify subscriptions; subscriptions are not identified by the combination of a person and a magazine so we should promote *Subscription* to a class. Furthermore a person might have multiple subscriptions to a magazine; only the (b) model can readily accommodate this.